8-15-2016

LibDetector: Version Identification of Libraries in Android Applications

Zhihao Mike Chi
zc3896@rit.edu

ROCHESTER INSTITUTE OF TECHNOLOGY

MASTERS THESIS

LibDetector: Version Identification of Libraries in Android Applications

Author:
Mike CHI

Supervisor:
Dr. Meiyappan NAGAPPAN

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Software Engineering

in the

Department of Software Engineering
B. Thomas Golisano College of Computing and Information Sciences

August 15, 2016

The thesis "LibDetector: Version Identification of Libraries in Android Applications" by Mike CHI, has been examined and approved by the following Examination Committee:

Dr. Meiyappan Nagappan

Thesis Committee Chair

Assistant Professor

Dr. Mehdi Mirakhorli

Assistant Professor

Dr. Scott Hawker

SE Graduate Program Director

Associate Professor

To my family for their love and support, and making it possible for me to follow my dreams.

To my friends for always believing in me and giving me the strength to persevere.

Acknowledgements

I would like to express my gratitude towards my thesis adviser Meiyappan Nagappan for his guidance, support, consideration and patience in helping me navigate my research. Despite your hectic schedule, you were always available and willing to help with anything and everything. I am sincerely grateful for everything you have done for me.

I would like to thank my amazing colleagues Craig Cabrey, Bushra Aloraini and Joanna Cecilia Santos for helping me during the various stages of my work - your help and contributions have been invaluable to me.

I would also like to thank Paul Hulbert, Austin Malerba and Matthew Mansor for helping me overcome the challenges of crawling the Google Play store.

Abstract

LibDetector: Version Identification of Libraries in Android Applications

by Mike CHI

Supervising Professor: Dr. Meiyappan NAGAPPAN

In the Android ecosystem today, code is often reused by developers in the form of software libraries. This practice not only saves time, but also reduces the complexity of software development. However, like all other software, software libraries are prone to bugs, design flaws, and security vulnerabilities. They too undergo incremental updates to not only add/change features, but also to address their flaws. Unfortunately, the knowledge gap between consumers and maintainers of software libraries presents a barrier to the timely adoption of important library updates.

Therefore we present Libdetector, a tool for identifying the specific version of Java libraries used in Android applications. Using LibDetector, we perform a large empirical analysis of the current trends of library use in the Android ecosystem. We find that a huge proportion of applications currently available on the Google Play Store use outdated libraries. We also explore the potential effects of this lax updating practice. In 2 of the 17 libraries we studied, apps that contain outdated versions of the library had a significantly different average rating than apps that contain more recent versions of the library. Finally, we find in a case study that a vulnerable version of a library is a realistic threat to the security of apps consuming that version of the library.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Code reuse is a widespread software development practice which allows developers to leverage existing code and use it for different purposes. It is a convenience which enables developers to be more productive by focusing on the unique features of their software rather than solving the same problems over and over again(Kapser and Godfrey, 2008)(Thummalapenta et al., 2010).

One major way in which code is reused today in software applications(apps for short) is through the use of software libraries(Mojica Ruiz, Nagappan, Adams, and Hassan, 2012; Li et al., 2016). By importing software libraries for use in their own apps, software developers gain access to many useful functions and features provided by the libraries that they would have otherwise needed to implement themselves. This not only saves time, but reduces the complexity of software development.

For example, let us consider libraries for user authentication. These libraries are tremendously useful in allowing software developers to consistently and reliably add user login capability to their applications. If these libraries did not exist, any developer who wanted a login feature in their application would need to develop the theory and code for it from scratch. This involves a huge amount of additional development effort. More importantly, it requires a significant amount of domain knowledge in software security in order to implement correctly. If done incorrectly, there could be severe consequences. For one, the privacy and security of the app's users would be jeopardized. Furthermore, it could be detrimental to the reputation of the developer or organization responsible for the vulnerable software.

Therefore, it makes sense for developers to take advantage of proven libraries that satisfy their requirements whenever possible. This is especially true for mobile application development, where development cost and time to market are critical factors(Abrahamsson et al., 2004). In fact, Mojica Ruiz et al. have found that on average, a staggering 84% of classes in Android applications are reused(Mojica Ruiz, Adams, et al., 2014). Research by Linares-Vásquez et al. suggest that a big part of the reused code is contributed by software libraries(Linares-Vásquez et al., 2014). This is consistent with findings by Wang et al.(Wang et al., 2015) which reported that libraries are responsible for over 60% of the code in an Android application.

1.2 Motivation

While there are concrete benefits of using software libraries, we must be aware of the consequences. Like any other piece of software, software libraries undergo change and increase in complexity over their lifetime(Lehman, 1980). They too are susceptible to design flaws, bugs, and security vulnerabilities(Constantin, 2014)(Constantin, 2015). In fact, over the years we have witnessed critical flaws in popular software libraries including the Apache Commons Collection(*Arbitrary remote code execution with InvokerTransformer* 2015), Facebook Android SDK(*The FAT Attack. Facebook Social Login Session Hijacking Vulnerability* 2014), OpenSSL(*Vulnerabilities* 2016), and many more.

The good news is that as bugs and vulnerabilities are discovered, software library maintainers are able to efficiently release fixes to all applications via library updates. To make things easier on developers, there are many powerful build tools available today. Build tools help developers manage software library dependencies[1] by installing the necessary libraries before compiling[2] the application. This ensures that whenever an application is compiled, it includes all the code that it is supposed to include in order to work as intended. Popular build tools in the Android ecosystem include Maven[3] and Gradle[4].

[1] Software dependencies refer to components which an application relies on in order to function properly

[2] Software compiling refers to the process of converting source code - code that humans can easily read and understand - into a form that can be read and processed by a machine

[3] https://maven.apache.org/

[4] https://gradle.org/whygradle-build-automation/

Despite the convenience of the build tools, research has shown that software developers do not regularly upgrade to the latest version of software libraries in their application(Bavota et al., 2015). This is because software libraries are typically developed by a third party(developers from outside the organization) and shared with other developers. We refer to these libraries as third-party or external libraries. As a result, the developers using an external library are not intimately familiar with the library or its changes and its flaws(Davies et al., 2013). In fact, this is precisely the reason why developers often specify the exact working version of libraries as the dependencies of their application. It guarantees that their app will build in the same way each and every time and will work as intended. If instead they left the version unspecified, the build tool would retrieve the latest version of the library when compiling the application. This introduces a temporal dependency where their application might build differently just because it was compiled on a different date, which may cause unforeseen problems with their application. For instance, should the library introduce new changes to its application program interface(API[5]), it could mean that their application becomes unstable or even stops working altogether. If this happens, the developers would have to refactor their application to get it working once more with the new version of the library(Robbes, Lungu, and Röthlisberger, 2012).

When we consider the complexity of many modern day software applications, constantly having to resolve these dependencies could be extremely disruptive to software development and entail an enormous cost in time and effort(Dig and R. Johnson, 2006). Furthermore, a library can depend on other libraries, making it exponentially more difficult to manage and resolve all of an application's external dependencies as its dependencies grow. For these reasons, it is not surprising that developers tend to be slow to embrace library updates. This is especially relevant in the case of Android app developers, who Mojica-Ruiz et al. have found to heavily leverage software libraries(Mojica Ruiz, Adams, et al., 2014).

If the updates for a library consist only of minor bug patches or feature changes, then not adopting the changes in a software application is of little to no consequence. As we have seen however, updates can contain security patches as well. In this worst-case scenario, failing to update the library means that the vulnerability lingers in the

[5]The API specifies how to interact with and consume the software component

app and leaves it open to exploitation. This puts the integrity of the app and the security of its users at risk.

Herein lies the problem and main motivation for our work. Our goal is to present a solution which allows all stakeholders[6] to easily identify the libraries being used in an application, whether or not the versions used are out of date, and the degree to which they are outdated[7]. We will be focusing on mobile applications due to their characteristically high dependence on third-party libraries. More specifically, we focus on apps in the Android ecosystem due to its popularity and proliferation in the mobile market. After establishing an effective method for identifying versions of libraries used in Android apps, we aim to explore the current trends of library use in the Android ecosystem. Finally, we wish to determine whether or not there are tangible consequences of using outdated software libraries(libraries where the version used is not the most current) in Android applications.

1.3 Research Questions

During the course of our research, we aim to address the following research questions:

1.3.1 RQ1: Do Android applications use outdated software libraries?

This is the main motivating force behind our work, and presents the bulk of our research effort. The goal is to identify the versions of libraries being used in a large set of Android applications so that we can get a clear picture of current practices in library change adoption in the Android ecosystem.

1.3.2 RQ2: Does using outdated software libraries affect the rating of an Android application?

Mojica Ruiz et al. found that including certain ad libraries in an Android application negatively impacted the ratings of the application(Mojica Ruiz, Nagappan, Adams, Berger, et al., 2014). These results prompted us to examine whether or not using outdated software libraries would have an effect on an app's rating.

[6] Everyone who has a vested interest in the application, including but not limited to its developers, investors, and users

[7] Refers to the number of versions released between the target version and the most recent version

1.3.3 RQ3: Are Android applications that use a vulnerable version of a library vulnerable?

However, we wanted to examine whether or not known vulnerabilities in certain versions of libraries persist in Android applications. We believe that when a vulnerability is discovered, it is reasonable to expect developers to try to address the vulnerability. Perhaps developers who consume a library are aware of the vulnerabilities, and have designed their app to prevent any exploitation. Moreover, mobile software and hardware evolve rapidly. It is possible that the threats that people have previously found have been mitigated in other ways and are no longer an issue. Therefore, we want to address the question of whether or not libraries with previously identified vulnerabilities continue to be a real, tangible threat to the security of Android applications.

1.4 Contributions of the Thesis

- We propose a novel approach for generating and comparing the signatures of Java class files.
- The LibDiff tool which allows for the automatic parsing of differences between consecutive versions of an Android library given the compiled library files as input.
- The LibDetector tool which automates the process of identifying specific versions of libraries used in Android applications given the diff files generated by LibDiff and the APKs for analysis as input.
- An empirical analysis of 21,524 Android applications for the presence of 442 versions of libraries across 17 different libraries, with all of our research data made publicly available on GitHub[8].
- An exploratory study on how outdated library use in Android applications may affect its stakeholders.

[8] https://github.com/zchi88/LibDetector

1.5 Thesis Organization

In this first chapter, we provided the background and motivation for our work and summarized our contributions to the software community. The rest of this thesis is organized as follows:

- **Chapter 2** provides a general overview of the proposed approach for addressing our three research questions.
- **Chapter 3** details our strategy for collecting our data set of Android APKs.
- **Chapter 4** details our strategy for collecting our data set of external Android libraries. We also explain our approach to creating the LibDiff tool, which we use to calculate the change history between versions of a library.
- **Chapter 5** details our strategy for generating class signatures and how LibDetector uses these signatures to determine matches to a version of a library in an Android application.
- **Chapter 6** explores how using outdated libraries in Android applications might impact its stakeholders. We first look at whether or not using outdated libraries can affect the rating of an app. We also look at whether or not using a vulnerable version of a library can affect the security of an app.
- **Chapter 7** presents the related works to our research.
- **Chapter 8** summarizes our findings, and presents directions for future work.

Chapter 2

Approach Overview

In this chapter, we outline our proposed approach to addressing each of our three research questions.

2.1 RQ1: Do Android Applications Use Outdated Software Libraries?

Before we can address this question, we first need a set of Android applications on which to test our approach and perform our analyses on. We aim for a large, randomly selected set of apps to ensure that they are representative of all the apps on the Google Play store(also known as the Android market). We explain in greater detail how we gathered the set of Android apps for our study in the following chapter(chapter 3).

Next, we need a way of identifying the versions of libraries contained in a given Android app. The technique we propose is based on Software Bertillonage which is inspired by Davies and colleagues(Davies et al., 2013). In Software Bertillonage, the purpose is to narrow the list of potential candidates(also known as the search space) of software components. The idea is to automatically filter out impossible candidates. This drastically reduces the number of manual comparisons that must be made in order to identify the best match in a target software system. However, an important distinction is that our goal is not only to reduce the search space, but also to automatically determine the best matching version of any third party library if it exists in the Android app without additional manual effort. There are two main requirements for implementing Software Bertillonage:

1. We must establish a corpus of all potential candidates. This simply means that we need to collect all of the libraries that an Android application might be using. For our research, we refer to this as the Android libraries whitelist. This is the main topic covered in chapter 4.

2. We must devise a strategy for uniquely characterizing each software library. This is so we can qualitatively compare libraries and distinguish between different libraries or different versions of the same library. Since Java libraries are fundamentally a collection of Java class files, Davies et. al proposed that we characterize a Java library by profiling each class that is contained within the library - they refer to this as the "class signature". In order to construct the class signature, they first record information about a class like its fully qualified name[1], its return type, its methods and method parameters, its fields, and its class, method and field modifiers. The final class signature is simply a SHA-1 hash value computed given this information. Using a hash value allows them to quickly and efficiently determine if classes in a library are the same as or different from a class in an app. In fact, by comparing classes in this manner, Davies and colleages were often able to determine the best library match in an application. This is because it is highly improbable for two different libraries(or different versions of the same library) to share the exact same class signatures across all of their classes. We follow a similar approach which we cover in greater detail in Chapter 5.

Figure 2.1 illustrates the overview of our approach for detecting the libraries being used by Android applications and identifying their version. We elaborate on Step One in chapter 4, and address Steps Two and Three in chapter 5.

Finally, given our set of Android apps and having established a method for identifying the versions of libraries in an app, we observe the trends in library use across all of the apps in section 5.4 and reach a conclusion to RQ1 in section 5.5.

[1]Describes the name of the class and the package that it belongs to

2.2 RQ2: Does Using Outdated Software Libraries Affect the rating of an Android Application?

For each library, we will be creating two groups. The "Recent" group will represent the ratings of apps that use more recent versions of the library. The "Outdated" group will represent the ratings of apps that use older versions of the library. Since we cannot assume a normal distribution for app ratings(and in fact they are often skewed), we require a non-parametric test to compare the two groups and determine whether or not there is a statistical difference between the two. For this, We propose the Mann-Whitney U test. We elaborate on this in section 6.2.

2.3 RQ3: Are Android applications that use a vulnerable version of a library vulnerable?

We select apps using various versions of a library. We then attempt to exploit the apps using a known and documented vulnerability in the library, and assess whether or not the apps using vulnerable versions of the library inherit the vulnerability. This is covered in greater detail in section 6.3.

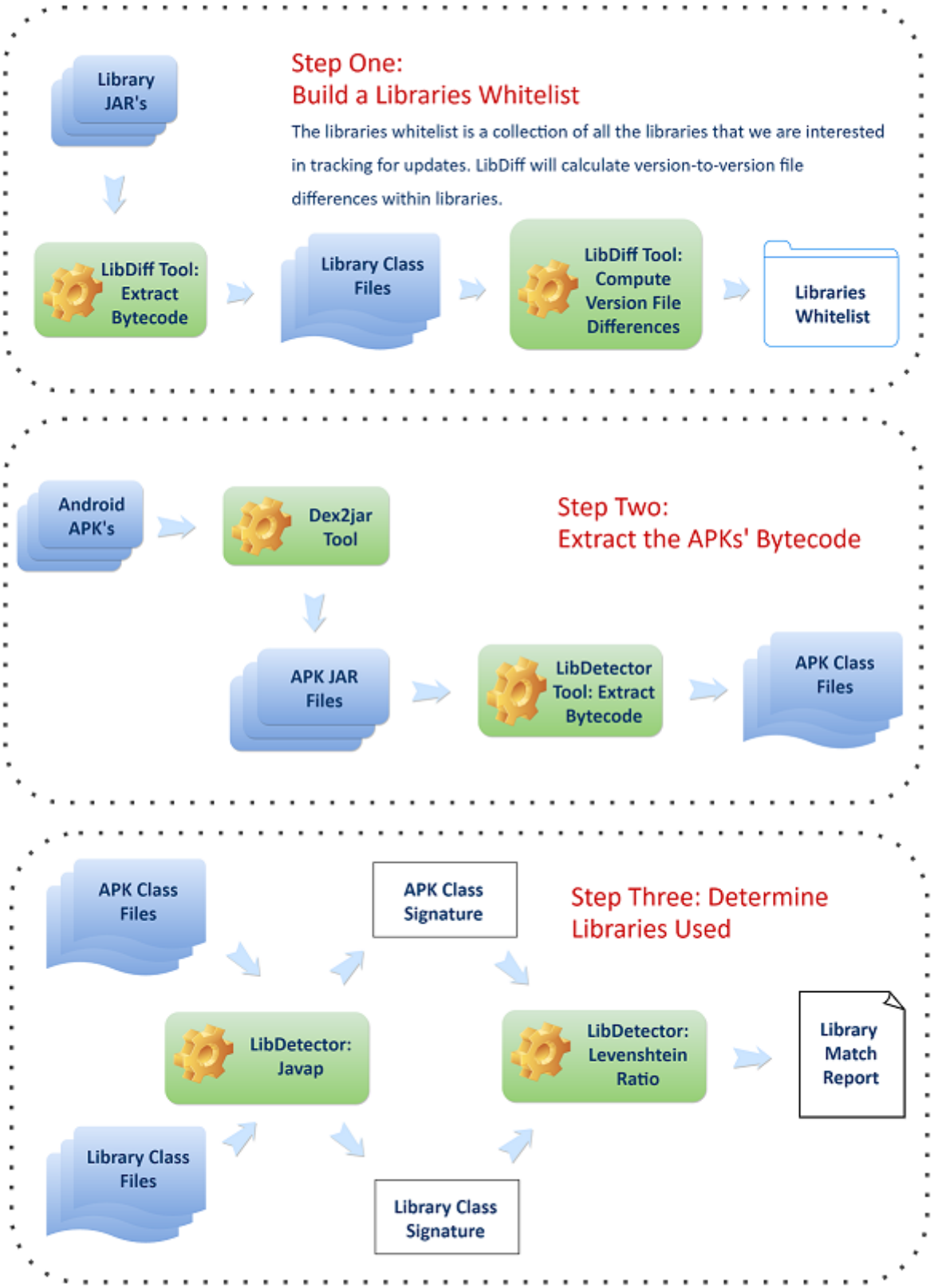

FIGURE 2.1: Proposed Approach for Identifying Versions of Libraries in Android Apps

Chapter 3

Crawling the Google Play Store

3.1 Motivation

In order to conduct our research, we first need a sample set of Android apps. This set has to be sufficiently large and randomly selected so that it is representative of the apps available to the typical Android user. We consider only apps from the Google Play store, the official marketplace for distributing Android apps. Android applications are available from the store in the Android Application Package(APK[1]) format. Therefore when we refer to an APK, we are referring to a single version out of potentially multiple versions of an app.

Unfortunately, Google does not provide an official utility for users to download APKs on the Google Play store in bulk. In fact, Google does not provide any way for users to download APKs to non-Android devices. While there are third-party solutions for this such as APKPure.com[2], they require us to manually download each APKs individually. This might be sufficient for a few apps, but it is extremely impractical for collecting a large number of apps. What we need is a program to crawl the Google Play store and automatically download APKs for us.

Luckily, there are a few open source crawlers available on GitHub. The crawler we use for our work was originally created by Ali Demiroz[3]. This crawler works by emulating an Android device(Samsung GT-I9300), and requesting to download an app from the Google Play store like a real device might. The version of the crawler we

[1]The Android Application Package is the file format used by the Android operating system for the distribution and installation of applications

[2]https://apkpure.com

[3]https://github.com/Akdeniz/google-play-crawler

use has been previously modified by other researchers for their work. These modifications include minor updates to the crawler(since it had not been maintained for nearly 3 years), integrating it with a MySQL database to record downloaded app meta-data(e.g. upload date, average rating, number of ratings, etc.), and error logging.

3.2 Methodology

As we previously stated, our sample set of apps has to be sufficiently large for us to be able to make any meaningful generalizations to all Android apps. Currently, the number of apps available on the Google Play store is estimated to be around 2 million(*Number of available applications in the Google Play Store from December 2009 to February 2016* 2016). With this in mind, we chose to gather roughly 20,000 apps for analysis. We believe this is sufficient because it allows us to report our results with a confidence level of 99% and a confidence interval of 0.91%.

In order for the crawler to work, we need to specify the apps we want it to download from the Google Play Store - it is not capable of browsing and downloading APKs on its own. We took the following steps to ensure that we randomly sampled roughly 20,000 apps from 24 app categories:

1. We obtained a list of the top 100 apps in each of the 24 app categories(business, sports, games, etc.) from AppAnnie(*Top Apps on Google Play, United States* 2016). This set of 100 apps across each of the app categories served as our seed to find other apps.

2. We wrote a python script to crawl the Google Play store for apps similar to the current app in a breadth-first fashion. Given our initial seed, this returned the names of approximately 600,000 apps in total.

3. From the list of 600,000 apps, we randomly select roughly 20,000 for the study.

The final step was to simply run the Google Play crawler, giving it the list of 20,000 apps as an input so that it knows which apps to crawl for. We started the crawler on May 5th, 2016. Whenever the crawler encountered a new APK or an updated APK, it recorded any meta-data associated with the app in a MySQL database and downloaded the APK file. APKs that the crawler has already visited are skipped.

TABLE 3.1: Number of Apps in Each Google Play Store Category

App Category	# Apps
BOOKS AND REFERENCE	1,720
BUSINESS	1,263
COMICS	31
COMMUNICATION	476
EDUCATION	2,791
ENTERTAINMENT	1,293
FINANCE	746
GAMES	2,849
HEALTH AND FITNESS	762
LIBRARIES AND DEMO	40
LIFESTYLE	1,587
MEDIA AND VIDEO	251
MEDICAL	444
MUSIC AND AUDIO	1,067
NEWS AND MAGAZINES	848
PERSONALIZATION	1,800
PHOTOGRAPHY	347
PRODUCTIVITY	875
SHOPPING	398
SOCIAL	335
SPORTS	605
TOOLS	1,849
TRANSPORTATION	472
TRAVEL AND LOCAL	1,097
WEATHER	130
Total:	24,076

3.2.1 Hardware

The crawler was run on a single Linux-based server located at the Rochester Institute of Technology(RIT).

3.3 Resulting Artifacts

After randomly paring down the list of 600,000 apps, we ended up with a total of 24,076 apps. Table 3.1 shows the category distribution of this list of apps.

Out of the 24,076 apps that we crawled for, we were able to successfully download 19,395 of them in the span of the month of May 2016. During this time, some of these apps were updated, and so we had multiple versions of an app(i.e. one app might have one or more associated APKs). We ended up with 21,692 APKs across 19,395

apps. This sample of 21,692 APKs is the data set that we work with throughout the course of our research.

We also generated a MySQL app metadata table with 27,913 total entries and 24,076 unique app entries. Note that there is a discrepancy between the number of APKs we were able to download and the number of entries in the table. This is due to the fact that APKs that we visit but could not download are still recorded to the database in order keep track of the APKs that we have already encountered.

3.4 Limitations

Due to the lack of maintenance on the Google Play crawler in conjunction with API changes in the Google Play Store during that time, almost 5,000(20%) of the apps could not be downloaded. There are two main types of errors thrown when the crawler failed to download the app:

1. The first is a very generic and unhelpful error, which simply states that there was an error when trying to download the app. We were unable to determine the root cause for this error.

2. The second is a more specific device incompatibility error. We believe that one of the reasons for this error is due to an incompatible configuration in the emulated Samsung GT-I9300 device. What this means is that the hardware/firmware requirements for some apps are not met by our device configuration, and so the app could not be downloaded. Another reason is that our method for retrieving a list of 600,000 apps sometimes returned foreign apps that are not available to users in the United States. We were unable to resolve this error despite our best efforts.

Chapter 4

LibDiff: Towards a Corpus of Android Libraries and Their Change History

4.1 Motivation

In the last chapter, we detailed the approach we took to gathering our set of apps for analysis. Our task now is to devise a technique for detecting software libraries in these apps. Therefore, in this chapter we aim to fully address the first requirement for implementing Software Bertillonage - building the libraries whitelist. We also set up to address the second requirement in the following chapter.

4.2 Methodology

4.2.1 Library Selection

Prior research has shown that it is possible to duplicate the entire Maven Repository[1] to generate a corpus of all available Java libraries on Maven(Davies et al., 2013; Ishio et al., 2016). For our research, we select only a few of the most popular Java libraries to include in our libraries whitelist. This is because the focus of our research is to present an approach for the identification of specific versions of libraries in an Android application, not to determine an exhaustive list of all libraries in the app. Furthermore,

[1] http://mvnrepository.com/

by looking at only a select handful of libraries, we are able to perform our analyses with only very average machines.

We based our list of popular libraries on the "Popular" category from the Maven Repository(*Top Projects* 2016) and from PrivacyGrade(Lin, Amini, Luan, et al., 2014). Doing so, we ended up with the following list of 17 libraries which we select for our research:

1. Acra
2. Apache Commons HttpClient
3. Apache Commons IO
4. Apache HttpComponents HttpClient
5. Facebook Android SDK
6. Google aoauth client
7. Google Protobuf
8. Gson
9. Jsoup
10. Mopub
11. Nostra13
12. Okhttp
13. Okio
14. Paypal core
15. Picasso
16. Retrofit
17. Twitter4j core

Note that at the time of writing this thesis, the Apache Commons HttpClient has already long been deprecated. In fact, the last released version (version 3.1) was in August of 2007[2], and since then this library has been succeeded by the new Apache HttpComponents HttpClient. Although HttpComponents HttpClient is still being maintained, Android has already stopped supporting Apache HttpClient in its SDK. This change took place in Android 6.0 Marshmallow, which was released in October of 2015(*Android 6.0 Changes* 2015). For these reasons, we were especially interested in analyzing whether or not these libraries continue to be used by developers in their Android applications.

[2] http://mvnrepository.com/artifact/commons-httpclient/commons-httpclient

4.2.2 Library Collection

We downloaded all the binary versions of the 17 Java libraries from the Maven Repository available in May 2016. While the Maven Repository is not a comprehensive source of all Java libraries that developers use and might not even contain all versions of a given library, it is the most comprehensive source of Java libraries that we are aware of(Davies et al., 2013). On Maven, there are three possible forms in which the code for a Java library are made available:

1. **sources.jar**: The Java Archive(JAR[3]) file format is used to aggregate one or more Java files into a single compressed file - the JAR. The sources.jar is simply the aggregation of all the original source(.java) files of the library.

2. **jar**: Compiling Java source code converts it into binary code, with the corresponding .java files being converted into one or more .class files. The "jar" version of the library on Maven is typically a JAR aggregating the compiled .class files of the library.

3. **aar**: The Axis Archive(AAR[4]) file format is a newer file format specification. It is a zip file format which actually contains within it a JAR of the binary library code.

For our research, we consider only libraries in their binary form(AAR's and binary JAR's).

4.2.3 Why Binary and Not Source?

While ideally we would like to collect the original source code for libraries and compare it to the original source code of an Android application, in reality this is not practical for two main reasons:

1. The source code for an Android application on the Google Play store is rarely accessible to anyone other than the developers.

[3] https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html
[4] http://filext.com/file-extension/AAR

TABLE 4.1: Procyon Decompiler Performance

Library	Extracted From	Failed Classes	Failed Files	Total Files
Facebook 4.11.0	Test App JAR	23	11	173
Facebook 4.11.0	Maven JAR	0	0	172
GSON 2.6.2	Test App JAR	0	0	63
GSON 2.6.2	Maven JAR	0	0	63

2. It is much more common for library developers to upload the binary form of their library code, thereby making the binary form of the library more accessible than the source form (Davies et al., 2013).

However, besides simply being more accessible, there is one major advantage to performing our analysis with only binary libraries - we are able to take full advantage of Oracle's[5] own class file dissassembler, Javap[6].

We believe that this is a huge improvement over many other works in Software Bertillonage on Java projects. We observed that a common decompiler used by those researchers is Apache BCEL[7], which we have found to be quite unwieldy. However, other open source decompilers exist and have been used as well. In fact, in our initial approach we too used an external Java decompiler.

The problem is that among these external Java binary decompilers, even the most powerful often cannot guarantee a 100% decompiling rate. That is, it is hard for external Java decompilers to completely reverse engineer the code and reconstruct the source code from the binary. We know this because when the decompilers fail, they log the failure in the body of the method that could not be decompiled. To illustrate this, we present the following results from the initial stages of our research when we were evaluating the performance of two Java decompilers for decompiling the Facebook Android SDK library and GSON library - Procyon[8] and CFR[9]:

As shown in tables 4.1 and 4.2, different decompilers have different strengths and return different results. While Procyon only failed to decompile files in 1 of the 4 cases, it also produced almost 3 times as many failures as CFR for the Facebook 4.11.0 library

[5]Oracle is the parent company of Sun Microsystems(the creators and maintainers of the Java language)

[6]http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html

[7]https://commons.apache.org/proper/commons-bcel/

[8]https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler

[9]http://www.benf.org/other/cfr/

TABLE 4.2: CFR Decompiler Performance

Library	Extracted From	Failed Classes	Failed Files	Total Files
Facebook 4.11.0	Test App JAR	8	8	173
Facebook 4.11.0	Maven JAR	0	0	172
GSON 2.6.2	Test App JAR	2	2	63
GSON 2.6.2	Maven JAR	0	0	63

which was extracted from our test Android application. It is interesting to note that both decompilers perform perfectly on the original binary libraries. While we do not know for sure which compiler compiled the original JAR's, we see that the compiler included with Android Studio tends to give external decompilers more difficulty.

Finally, we often do not need to reconstruct the complete source code. We only need important identifying information of a class files such as the package and class name, method declarations, and field declarations in order to generate a class signature. For all of the above reasons, we believe that:

- it is sufficient to analyze only binary Java code
- it is reasonable to assume that Javap will give us the most accurate class signature for characterizing Jar files to use in Software Bertillonage

4.2.4 Optimizing the Libraries Whitelist

After collecting our set of libraries, we devised a method to calculate the version differences between successive versions of a library. The purpose of this is to further narrow down the search scope and reduce the amount of comparisons to determine if a software library matched a software component in the target application. This is a departure from the work by Davies et al., which indexed all of the class files of a library JAR in order to generate a characteristic profile for each version of a library.

Since our goal is to construct a change history for a library, we needed a way of automatically figuring out the chronological order of a library. One possible way is to consider the version code of a library. Many developers today follow a best practice for the versioning of their software, also known as semantic versioning(Preston-werner, 2016). In semantic versioning, we describe the version of a software application in the form of **X.Y.Z**, where "X" is the application's major version, "Y" is its minor version,

and "Z" is its patch version. For software applications that follow semantic versioning, it is often very trivial to determine their chronological release order. For example, we can tell that Library-2.0.1 is a more recent release than Library-1.0.1 since its major version is bigger. However, Library-2.0.1 is an older version than Library-2.0.2 since its minor version is smaller.

Unfortunately, discerning the version order is not always so straightforward. For example, in some situations developers may want to try out experimental versions of their application. These versions may be referred to as release candidates(RC), beta, alpha, and so on. Semantic versioning does not provide any guideline for these type of releases. Because of this, we end up with library versions with names like acra-4.7.0 and acra-4.7.0-RC.1. Without a release history available, determining the correct version order of libraries versioned in this manner may be near impossible. In order to address this issue, we present a simple heuristic based on the assumption that the creation date of library JAR's should correlate with its chronological release order.

Having decided on our strategy for estimating the version order of a library, we can constructed a change history between successive versions. We start by extracting the class files from their binary JAR files. AAR files required an additional step - extracting the JAR file contained within the AAR first. Afterward, the same process is applied to extract the class files from that JAR.

Next, we perform an MD5 hash of each class file in each version of each library. Hashing functions have been proven to a very quick and efficient method for determining file similarity or difference(Davies et al., 2013)(Ishio et al., 2016). When two files have different hash codes, we know for certain that they are different, since hash functions are designed to always return the same hash code for the same input. In the case that two files return the same hash code, it is very likely that they are identical in content. However, there is a small chance that their content is actually different, and that they hashed to the same value by chance. The chance that this occurs is astronomically low, and so we do not consider this scenario. Therefore, by directly comparing the hash values of a class in one version of a library with the hash value of a class with the same name in the successive version of that library, we can determine whether or not a class has been changed from one version to the next. Given two libraries, "Library-1.0.0" and "Library-2.0.0", and a class file "Example.class" we summarize and

classify the 4 scenarios that may occur:

- **Deletion:** Example.class is present in Library-1.0.0 but not in Library-2.0.0
- **Addition:** Example.class is present in Library-2.0.0 but not in Library-1.0.0
- **Modification:** Example.class is present in both libraries, but have differing hash values
- **No Change:** Example.class is present in both libraries, and share the same hash value

We designed our tool, LibDiff, to sequence the versions of libraries based on our heuristic and perform the MD5 hashing and comparison of classes for us. By performing this comparison for all classes of two adjacent versions of a library[10], we effectively construct our version history of each library. In the case where a library is the very first version created, all of its classes are classified as additions, since they are all new. We store the results of the change history between two adjacent versions in a "diff.txt file". The structure of the directory for our libraries whitelist files and the diff.txt file is shown in figure 4.1.

4.2.5 Hardware

LibDiff was run on a 64-bit Windows 10 machine with 12.0 GB of RAM, an Intel i7-2600 processor clocked at 3.40GHz, and a 1.5 GB 7,200 RPM SATA hard drive.

4.3 Results

We observed that out of the 17 libraries we gathered for our study, 8 of the libraries contained at least one version where they deviated from the semantic versioning scheme. There were: Acra, Apache Commons HttpClient, Apache Commons IO, Google OAuth, Google Protobuf, Apache Components HttpClient, Retrofit, and Mopub. However, in the case of Google Protobuf, the developers were faithful to the concept of semantic versioning, which allowed us to discern the intended version order of their library despite not fully conforming to semantic versioning. An example of this can be seen

[10] We define adjacent versions as being versions of libraries where one is the predecessor of the other

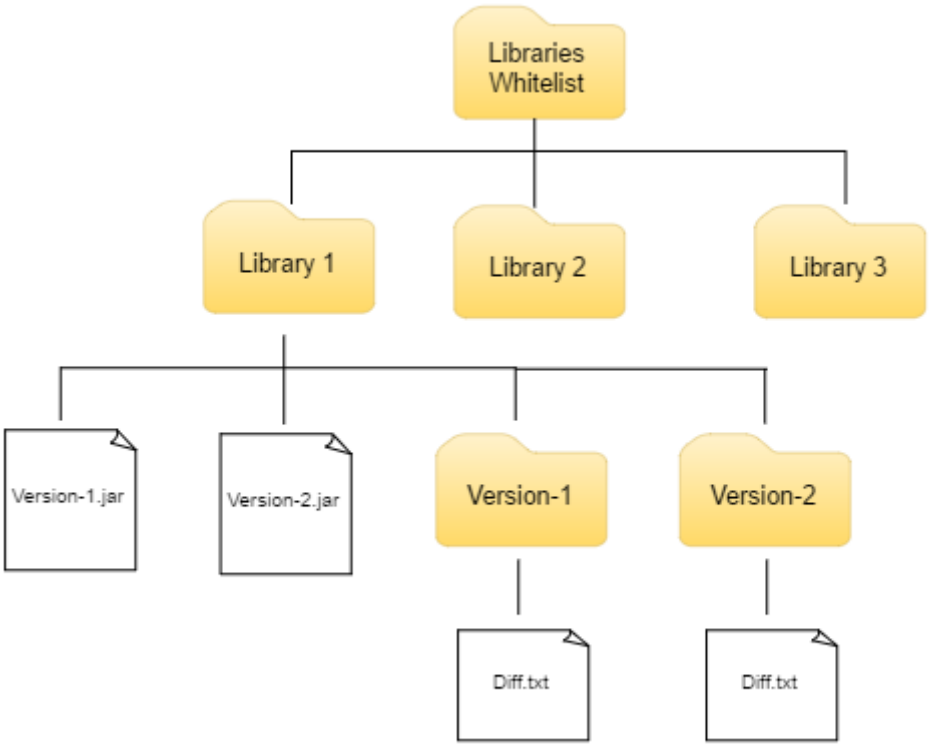

FIGURE 4.1: LibDiff Directory Structure.

in two versions of protobuf: protobuf-java-3.0.0-alpha-2.jar and protobuf-java-3.0.0-alpha-3.1.jar. Although this clearly does not conform to the semantic version model, it is clear and obvious that protobuf-java-3.0.0-alpha-2.jar preceeds protobuf-java-3.0.0-alpha-3.1.jar.

4.3.1 Artifact: The Android Libraries Whitelist

The resulting libraries whitelist for the 442 library JAR files across 17 libraries after running LibDiff expands to 84,344 .class files and 442 diff.txt files. It occupies 520 MB of disk space.

4.3.2 Performance

LibDiff took 1,827,520 milliseconds, or roughly 30 minutes and 27 seconds on our machine to process 442 unique binary library JAR files distributed across 17 different Java software libraries. This equates to roughly 4.072 seconds to unpack the class files

of a single binary library JAR, then calculate and log the version changes with respect to its previous version in the "diff.txt" file.

4.4 Validation

We validated the diff files generated by LibDiff which detail version-to-version differences in the binary files of a library by manually comparing several diff files in Facebook and GSON against their source file change history in the Facebook Android SDK[11] and GSON[12] GitHub repositories, respectively. We confirmed that classes that are reported by LibDiff as being added, modified or deleted coincides with their GitHub commit history.

4.5 Limitations

The heuristic we use for determining the intended version order of a library is not perfect. While developers often release the versions in the order they are intended chronologically, this is not always the case. For example, future versions of a library might be released either early or concurrently with the main version for beta testing.

Another limitation is that we are often limited to the versions of libraries hosted on Maven. While this is a very extensive collection, it is far from being complete. Many older versions of a library are routinely removed from the repository, making them unavailable for us to download. It is certainly possible for applications to contain older versions of those libraries which are no longer available on Maven if they were installed before being removed.

4.6 Threats to Validity

A big part of the LibDiff tool depends on our ability to correctly identify successive versions of a library in order to construct our version change history. Currently, this process relies on a simple heuristic that is not perfect, and does not always correctly identify the intended order of the versions of a library. The result is that where the

[11] https://github.com/facebook/facebook-android-sdk
[12] https://github.com/google/gson

versions are incorrectly ordered, we get a distorted change history. This could affect the internal validity of our library version identification in Android applications.

Chapter 5

LibDetector: Identifying the Version of a Library Used in Android Applications

5.1 Motivation

In the last chapter, we detailed our approach to constructing our libraries whitelist, as well as a Software Bertillonage optimization to avoid having to compare an unknown software component in an Android app against every class file of a given software library. With the whitelist of libraries, the diff.txt files which describe their version change history, and our set of 21,692 Android APK's, we are prepared to address the second requirement for Software Bertillonage and realize our goal of identifying versions of libraries used in Android applications.

5.2 Methodology

5.2.1 Extracting the Binary Code

The first step in our approach is to extract the binary code from an APK file. In every APK file, there is at least one ".dex"(short for **D**alvik **Ex**ecutable) file. The dex format is simply compiled Android application code that Android devices can understand and execute. In order to convert these .dex files into the desired binary JAR's, we require

the use of the dex2jar[1] tool. We observed that in some rare cases, an application contained more than one .dex file. For example, com.glu.deerhunt16 contained 3 .dex files - classes.dex, classes2.dex, and classes3.dex. We made sure that LibDetector checks for those situations, and aggregates the fragmented JAR files that are extracted from each of the .dex files when this occurs.

In some cases the class files for an APK failed to extract, so we skip the APK. This is due to the fact that we were trying to extract them on a Windows machine and the name of the class file contained special reserved keywords of the Windows OS, such as "con" or "aux". In total, 168 of our 21,692 APK's failed to extract, leaving us with a new working set of 21,524 apps.

5.2.2 Constructing the Class Signature

The next step is to formulate the strategy for generating our class signature for comparing classes in the APK to classes in the library. Unfortunately, we cannot reuse the same strategy for comparing library versions using an MD5 hash. The reason for this is that there exist many compilers, and each compiles the code in slightly different manner. With hash functions, even a change as minor as a single whitespace character can completely change the hash value of a file, rendering it completely useless for comparison of files where we cannot expect consistent formatting. Furthermore, we saw in chapter 3 that different compilers can in fact produce very different code.

When working with different versions of a given library, it is safe to assume that the library is being developed within the same environment. Therefore, we can assume that the same compiler is being used to compile the library code and would not expect major code differences to be introduced by the compiler. However, there are a variety of development environments available to Android app developers such as Android Studio[2], Eclipse[3] and NetBeans[4]. Many of these development environments integrate a different compiler, and we do not know which environment a developer is

[1] https://github.com/pxb1988/dex2jar
[2] https://developer.android.com/studio/index.html
[3] https://eclipse.org/home/index.php
[4] https://netbeans.org/

using. This means that we often do not know which compiler has been used to compile an APK. Without this information, we cannot accurately compare library code from an APK against the code from the original library using an MD5 hash.

The solution we arrived at is to calculate a class signature with the same elements as the signature described in the work by Davies et al.(Davies et al., 2013). However, instead of decompiling the binary code of the APK and then parsing for the class name and method and field declarations, we use Javap, Oracle's own Java class file disassembler. We believe it is reasonable to assume that this approach will produce superior results since we avoid decompiler and parsing error. Furthermore, in our approach we do not need to worry about inner classes. When we examine source code, a single .java file might contain several classes besides the main class. However, the compilation process creates a .class file for each of the inner classes, so we can execute Javap on them individually.

To use Javap, we simply make a call to it and pass it the path to the class file we wish to analyze as an argument. By default Javap returns only the package, protected and public classes and members of a class. We want all of the classes and members, so we pass in the "-private" flag as a parameter as well. Figure 5.1 shows the output of Javap when run on the same class extracted from the library versus extracted from an APK. We observe that there is also a huge amount of difference between the two classes.

Output of Javap on Facebook SDK 4.11.0's "AppCall.class" file extracted directly from the Library JAR

```
Compiled from "AppCall.java"
public class com.facebook.internal.AppCall {
  private static com.facebook.internal.AppCall currentPendingCall;
  private java.util.UUID callId;
  private android.content.Intent requestIntent;
  private int requestCode;
  public static com.facebook.internal.AppCall getCurrentPendingCall();
  public static synchronized com.facebook.internal.AppCall finishPendingCall(java.util.UUID, int);
  private static synchronized boolean setCurrentPendingCall(com.facebook.internal.AppCall);
  public com.facebook.internal.AppCall(int);
  public com.facebook.internal.AppCall(int, java.util.UUID);
  public android.content.Intent getRequestIntent();
  public java.util.UUID getCallId();
  public int getRequestCode();
  public void setRequestCode(int);
  public void setRequestIntent(android.content.Intent);
  public boolean setPending();
}
```

Output of Javap on Facebook SDK 4.11.0's "AppCall.class" file extracted from an APK compiled in Android Studio

```
public class com.facebook.internal.AppCall {
  private static com.facebook.internal.AppCall currentPendingCall;
  private java.util.UUID callId;
  private int requestCode;
  private android.content.Intent requestIntent;
  public com.facebook.internal.AppCall(int);
  public com.facebook.internal.AppCall(int, java.util.UUID);
  public static com.facebook.internal.AppCall finishPendingCall(java.util.UUID, int);
  public static com.facebook.internal.AppCall getCurrentPendingCall();
  private static boolean setCurrentPendingCall(com.facebook.internal.AppCall);
  public java.util.UUID getCallId();
  public int getRequestCode();
  public android.content.Intent getRequestIntent();
  public boolean setPending();
  public void setRequestCode(int);
  public void setRequestIntent(android.content.Intent);
}
```

FIGURE 5.1: Javap Example Output for AppCall.class from Facebook SDK 4.11.0

However, upon closer inspection we notice that most of the difference is not in the content, but rather their order. Therefore, we propose a few transformations to help normalize and control for these order differences introduced by the compiler. The first transformation we need is to strip all trailing and leading whitespace from each line of the output. Second, we sort each line alphabetically. Figure 5.2 shows the result of performing these two transforms on the Javap output.

```
1  }
2  Compiled from "AppCall.java"
3  private android.content.Intent requestIntent;
4  private int requestCode;
5  private java.util.UUID callId;
6  private static com.facebook.internal.AppCall currentPendingCall;
7  private static synchronized boolean setCurrentPendingCall(com.facebook.internal.AppCall);
8  public android.content.Intent getRequestIntent();
9  public boolean setPending();
10 public class com.facebook.internal.AppCall {
11 public com.facebook.internal.AppCall(int);
12 public com.facebook.internal.AppCall(int, java.util.UUID);
13 public int getRequestCode();
14 public java.util.UUID getCallId();
15 public static com.facebook.internal.AppCall getCurrentPendingCall();
16 public static synchronized com.facebook.internal.AppCall finishPendingCall(java.util.UUID, int);
17 public void setRequestCode(int);
18 public void setRequestIntent(android.content.Intent);
```

```
1  }
2  private android.content.Intent requestIntent;
3  private int requestCode;
4  private java.util.UUID callId;
5  private static boolean setCurrentPendingCall(com.facebook.internal.AppCall);
6  private static com.facebook.internal.AppCall currentPendingCall;
7  public android.content.Intent getRequestIntent();
8  public boolean setPending();
9  public class com.facebook.internal.AppCall {
10 public com.facebook.internal.AppCall(int);
11 public com.facebook.internal.AppCall(int, java.util.UUID);
12 public int getRequestCode();
13 public java.util.UUID getCallId();
14 public static com.facebook.internal.AppCall finishPendingCall(java.util.UUID, int);
15 public static com.facebook.internal.AppCall getCurrentPendingCall();
16 public void setRequestCode(int);
17 public void setRequestIntent(android.content.Intent);
```

FIGURE 5.2: Javap Output Stripped of White space and Ordered

After these two transforms, we see that the difference between the two files has decreased significantly. However, there are still some compiler differences we need to account for. In the AppCall.class file extracted from the library, we see that its compiler has injected a line describing which .java source file was compiled to produce the .class file. We found this information to be extraneous, so we removed it. Finally, we see that the Android studio compiler tends to remove the synchronized keyword from the method signature. This may seem like an odd behavior at first, since this removal could drastically alter the behavior of the software. However, we discovered that what ends up happening is that the sychronized methods are being converted by the compiler to equivalent synchronized statements. Figure 5.3 shows an example of this in a decompiled method of the AppCall.class file.

AppCall.class code decompiled from Facebook Android SDK 4.11.0

```
private static synchronized boolean setCurrentPendingCall(
        AppCall appCall) {
    AppCall oldAppCall = getCurrentPendingCall();
    currentPendingCall = appCall;

    return oldAppCall != null;
}
```

AppCall.class code decompiled from Android APK Using Facebook Android SDK 4.11.0

```
private static boolean setCurrentPendingCall(final AppCall currentPendingCall) {
    synchronized (AppCall.class) {
        final AppCall currentPendingCall2 = getCurrentPendingCall();
        AppCall.currentPendingCall = currentPendingCall;
        return currentPendingCall2 != null;
    }
}
```

FIGURE 5.3: Compiler Differences: Synchronized keyword

Because the Android Studio compiler was moving the synchronized keyword into the body of a method which we do not consider as part of the class signature, we decided to ignore any occurrences of the synchronized keyword. With these final two transformations, we resolve all of the differences between the classes in the binary code of the libraries and in the binary code of the APK's, and the output of Javap for the same classes become identical. In summary, the class signature in our Software Bertillonage technique is constructed as follows:

1. For a given class, call Javap with the -private flag and the path to the class as a parameter to retrieve the fully qualified class name and all of its method and field declarations.

2. Strip any trailing and leading whitespace in the output

3. Remove the "compiled from" line in the output

4. Remove the synchronized keyword from the output

5. Order the output alphabetically line by line

5.2.3 Class Signature Comparison

With our class signature defined, we next need to specify how we will compare the similarity between two classes should their signatures differ. Davies et al. proposed hashing this signature and storing it in a table for efficient comparison. We propose the use of the Levenshtein Distance (also known as edit distance) instead. The way the Levenshtein distance works is that it calculates the least number of changes required to transform one string to another. For example, consider the two strings "cat" and dog". In order to transform "cat" into "dog", two edits are needed - "c" -> "d" and "a" -> "o". Therefore, the Levenshtein Distance is 2.

The main advantage of using the edit distance over the hashing approach is that we can explore the degree to which two classes are similar or different. With hash values, we can only determine if two class signatures are the same or not, with no room for error. Although this would more or less eliminate the possibility of a false positive match, it is extremely inflexible in dealing with situations where the code may have been slightly altered by the compiler, or has been obfuscated. By using the Levenshtein Distance to compare signatures, we design for some flexibility in the signature matching.

Furthermore, we propose keeping track of the degree of similarity between components in the APK and a version of a software library via a Levensthein Ratio metric. The Levenshtein Ratio normalizes the edit distance with respect to the size of the signatures being compared. This is important because on its own, the edit distance is not very useful. For example, in a class signature with 20 characters, an edit distance of 20 that the two classes under comparison are completely different. However, an edit distance of 20 where the class signature contains 100 characters means that the two classes are roughly 80% identical. To calculate the Levenshtein Ratio, we simply divide the Levenshtein Distance by the longer length of the two class signature strings being compared.

5.2.4 Library Version Identification

To identify the version of a library in an APK, we first iterate through each library in reverse chronological order, starting with the most recent version and working our

way towards the oldest version. We first reduce the search space by filtering out all libraries where we expect a file to be deleted but is present in the APK, or where the file should exist but is not present in the APK. This leaves us with a subset of libraries which are potential matches. For these libraries, we only have to compare the class signatures of the modified class files reported in the diff.txt files against the class signatures of the class files present in the APK. By keeping track of an aggregated Levenshtein ratio for a library during this analysis, we are able to determine the best match and its degree of similarity to the unknown component in the app. We continue this process for every version of every library until one of three situations occur:

1. The initial filtering based on presence/absence of files leaves zero candidates, which means the APK does not use any versions of the library

2. The files in a version of a library is a perfect match to the files in the APK with a total Levenshtein Ratio of 0, at which point we can stop and declare that version as the best match

3. We compare the files in the APK with each relevant file in every version of the library, and went through all the candidates without finding a perfect match. We report the match (or matches) with the lowest Levenshtein Ratio.

The results of the LibDetector analysis is stored in an individual "libraryMatchResults.txt" file in a directory named after the APK that was analyzed.

We split our data set of apps into 2 subsets of roughly the same size(about 10,500). This was done in order to decrease the computation time required to analyze the versions of libraries used in all 21,524 apps. Half of the results were computed on a Windows machine, while the other half was computed on a Linux machine.

5.2.5 Hardware

LibDetector was run on a 64-bit Windows 10 machine with 12.0 GB of RAM, an Intel i7-2600 processor clocked at 3.40GHz, and a 1.5 GB 7,200 RPM SATA hard drive(as used for LibDiff in chapter 4). We also leveraged several Linux-based servers for parts of our research computation.

5.3 Evaluation

We evaluated LibDetector by using it to analyze a simple Android test application created in Android Studio. We specified the dependencies of our application to be Facebook Android SDK 4.11.0, GSON 2.6.2, and Twitter4jcore 4.0.4. The results are promising, with all three of the libraries identified with a Levenshtein Ratio of 0.0 (meaning a perfect match), and no false positives or false negatives. Since we correctly identified 3 out of 442 possible versions of libraries, we have a true positive of 3/3 and a true negative of 439/439. This means that for our test app, LibDetector performed with a precision of 100% and recall of 100%.

5.4 Results

Both machines ran LibDetector and completed their analysis in roughly the same amount of time - 3 days. Therefore, the total amount of time required to analyze 21,524 apps was roughly 6 days, which comes out to less than half a minute to analyze a single APK for the inclusion of 442 versions of software libraries.

Since we examined over 20,000 APKs, we are not able to provide our full set of results in this paper. However, everything is made available on our GitHub page[5]. Our results are documented in the following files on our GitHub page:

- The **"LibDetector Results Per App.zip"** file contains the direct output from the LibDetector tool.
- **"APKLibUse.xlsx"** combines the individual LibDetector results with the recorded metadata for each APK into an excel spreadsheet.
- **"LibVersionUseFrequency.xlsx"** breaks down the results by each library, and shows the trends in library use among the APKs.

[5] https://github.com/zchi88/LibDetector/tree/master/LibDetector%20Results

TABLE 5.1: Library Use Trends in Android Applications

Library Name	Total Apps Using Library	Apps 1+ Minor Versions Behind	% Apps 1+ Minor Versions Behind	Apps 1+ Major Versions Behind	% Apps 1+ Major Versions Behind
Acra	341	341	100.00%	0	0.00%
Okio	717	717	100.00%	100	13.95%
Apache Commons IO	656	655	99.85%	105	16.01%
Jsoup	661	660	99.85%	0	0.00%
Google Protobuf	72	71	98.61%	71	98.61%
Facebook Android SDK	1,848	1,822	98.59%	1,213	65.64%
Google Oauth Client	103	101	98.06%	0	0.00%
Gson	2,188	2,124	97.07%	95	4.34%
Retrofit	699	672	96.14%	672	96.14%
Okhttp	1,734	1,666	96.08%	945	54.50%
Mopub	221	199	90.05%	160	72.40%
HttpComponents HttpClient	338	291	86.09%	0	0.00%
Twitter4j Core	648	427	65.90%	427	65.90%
Picasso	1,308	599	45.80%	105	8.03%
Commons HttpClient	43	12	27.91%	0	0.00%
Nostra13 ImageLoader	914	219	23.96%	0	0.00%
Paypal Core	0	N/A	N/A	N/A	N/A

5.5 Analysis

Before we analyzed our results, we first filtered out multiple versions of an app, leaving us with only 19,395 APKs which are the most recent version of each app. This is is prevent any one app from having more influence on our results than other apps.

Table 5.1 shows the number of apps that use each of the 17 libraries we looked at. Our results show that for any given library, it is not uncommon for apps to be using an outdated version of the library. In fact, in 11 of the 17 libraries we looked at, the proportion of apps using a version of the library that is at least one minor version behind exceeded 90%.

More importantly, for 10 of the 17 libraries we looked at, we saw that there were apps that used a version that was behind by at least one major version. In the case of Google Protobuf, as many as 98.61% of the apps containing that library was using a version that was a major update behind. Note that the Paypal Core library was not found in any of the apps we examined with LibDetector.

Figures 5.4, 5.5, 5.7, 5.8, 5.9, 5.10, 5.6, 5.11, 5.12, 5.13, 5.14, 5.15, 5.16, 5.17, 5.18, 5.19 provide a more detailed breakdown of the number of apps using the various version

of each library.

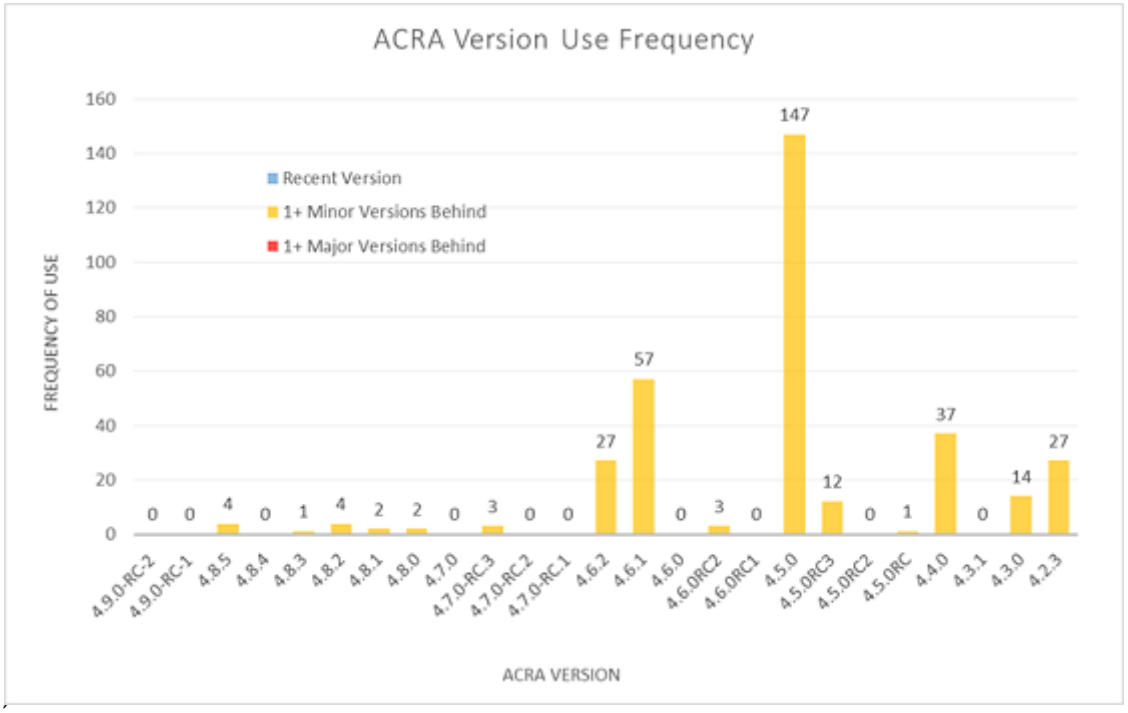

FIGURE 5.4: Distribution of APK's Using Versions of Acra

Based on the results, it is evident that many Android developers use outdated versions of Java libraries in their applications. This is consistent with the research findings by Bavota et al. (Bavota et al., 2015). Interestingly, even the very outdated and deprecated Apache Commons HttpClient library continues to exist a a dependency for some apps.

5.6 Summary

In the last 3 chapters (chapters 2, 3, 4) we detailed our approach to identifying not only the library being used in an Android application, but the specific version of the libraries as well. Based on the results of our LibDetector tool, we arrive at the following conclusion to our first research question:

5.6.1 RQ1: Do Android Applications Use Outdated Software Libraries?

Based on the results shown in table 5.1 and in figures 5.4, 5.5, 5.7, 5.8, 5.9, 5.10, 5.6, 5.11, 5.12, 5.13, 5.14, 5.15, 5.16, 5.17, 5.18, 5.19, we conclude that Android apps do commonly use outdated software libraries.

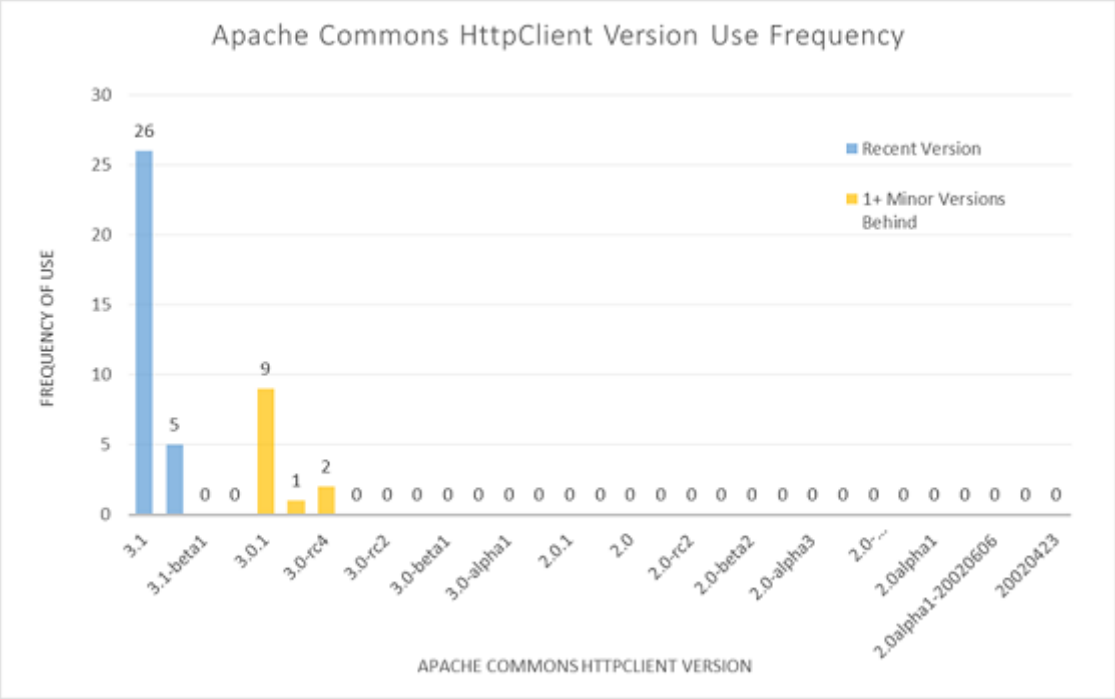

FIGURE 5.5: Distribution of APK's Using Versions of Apache Commons HttpClient

5.7 Limitations

A key component of our approach is the libraries whitelist. However, it is impossible to gather all of the software libraries that may be used by mobile applications. There are two main reasons for this:

1. Maven Repository, the most comprehensive collection of Java libraries we know about, is incomplete and does not contain all libraries. In many cases, the libraries that it does host are incomplete since older versions of a library periodically get removed.

2. Not all libraries are made publicly available. Developers may use in-house libraries(software libraries that are developed for use within the same organization) which are never shared with the public. Our assumption is that with in-house libraries, developers are more aware of their evolution and that there is a lower barrier to adoption, therefore making the accurate identification of these libraries less of a concern than external software libraries. However, the fact remains that these libraries will typically not be available for a libraries whitelist.

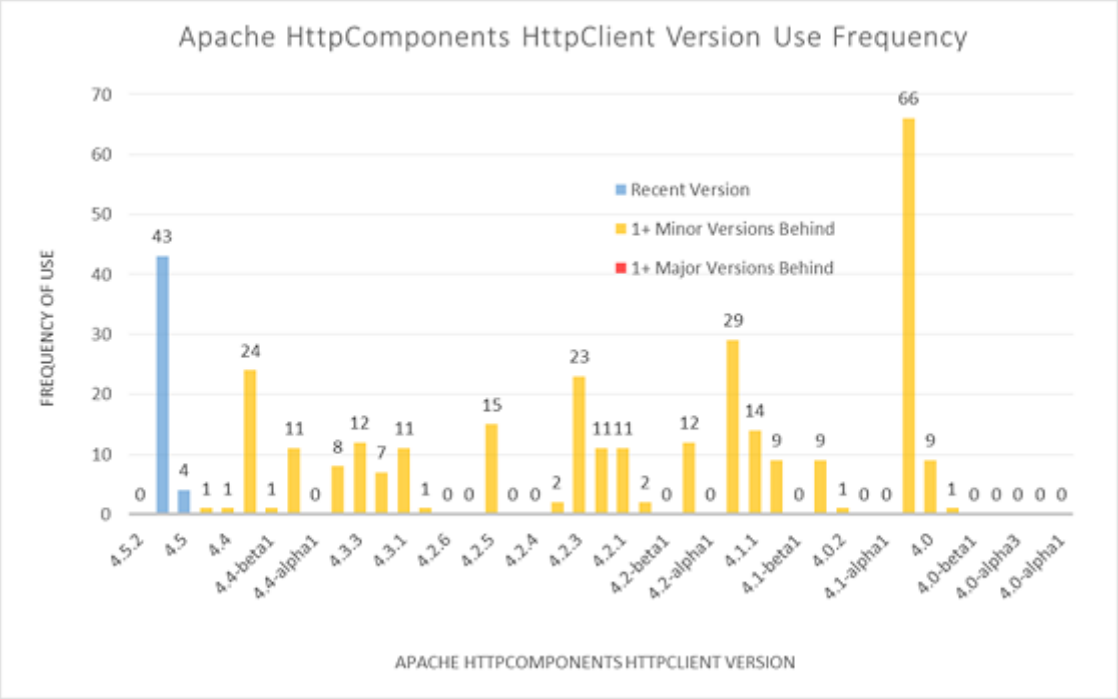

FIGURE 5.6: Distribution of APK's Using Versions of Apache HttpComponents HttpClient

Because it may be impossible to ever compose a complete libraries whitelist, it would be impossible to identify any version of any library in a given application with 100% accuracy using our approach.

5.8 Threats to Validity

One of the biggest threats to validity was the small scale of our proof of concepts. We tested our proposed approach on only a very small sample.

For example, we tested LibDetector on just one very simple Android application developed in Android Studio to evaluate its accuracy. This does not consider or account for differences in APKS built using for example Eclipse or NetBeans. When we tested our approach to generating class signatures, we only looked at one distinct library version of two libraries - Facebook Android SDK and Gson. Because we did not do more thorough testing, there may be things that we failed to consider and account for in the design of our study, thereby reducing the internal validity of our findings.

As we mentioned in chapter 2, the crawler was unable to download nearly 20% of the apps that we instructed it to crawl for. We also had to remove another 168 apps from our analysis due to our inability to extract their byte code onto a Windows

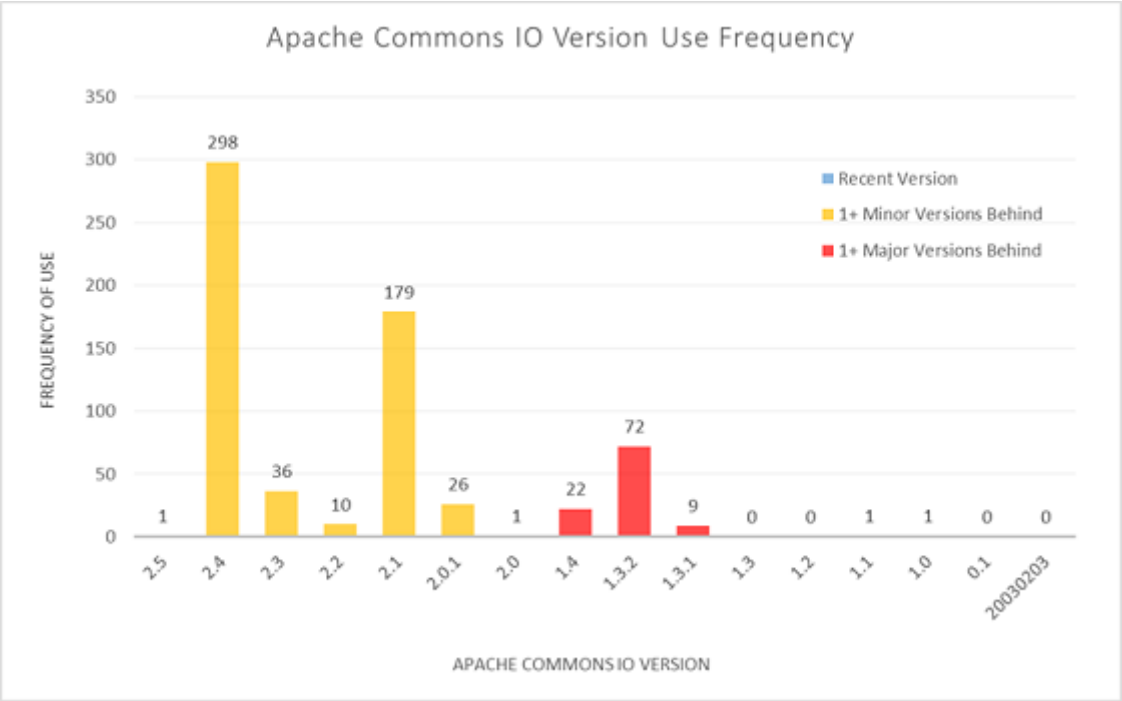

FIGURE 5.7: Distribution of APK's Using Versions of Apache Commons IO

machine. It is possible that there is a systematic difference between these apps that differentiate them from the apps that we studied, thereby reducing the external validity of our findings.

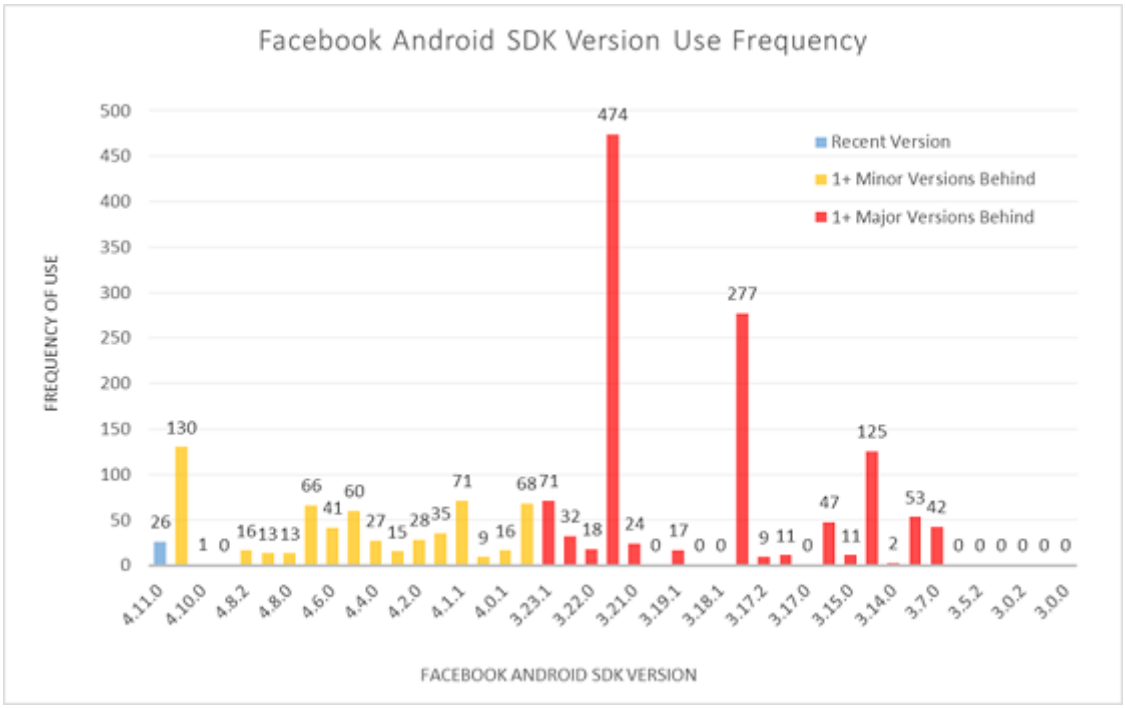

FIGURE 5.8: Distribution of APK's Using Versions of Facebook Android SDK

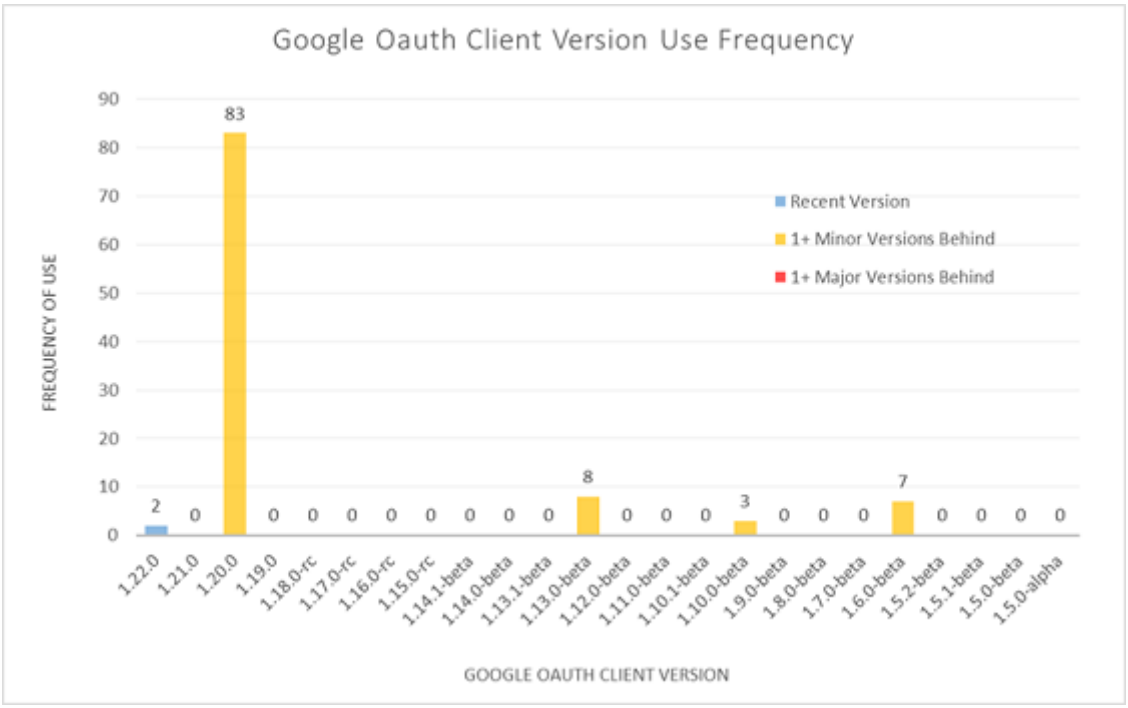

FIGURE 5.9: Distribution of APK's Using Versions of Google OAuth

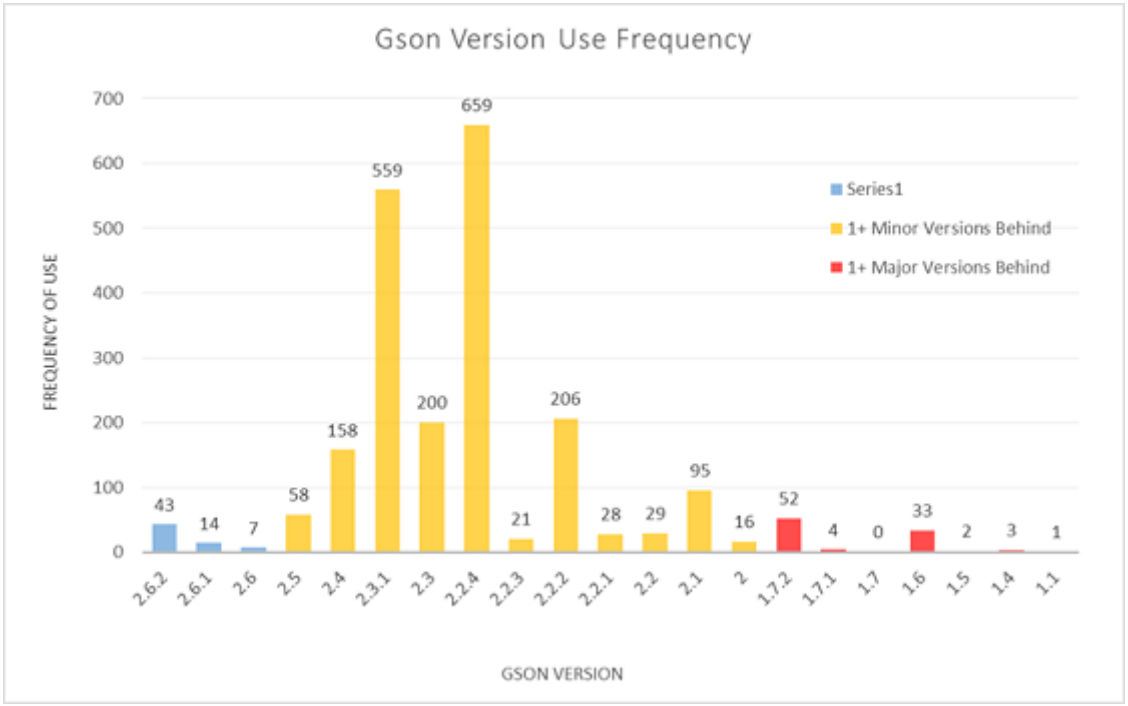

FIGURE 5.10: Distribution of APK's Using Versions of GSON

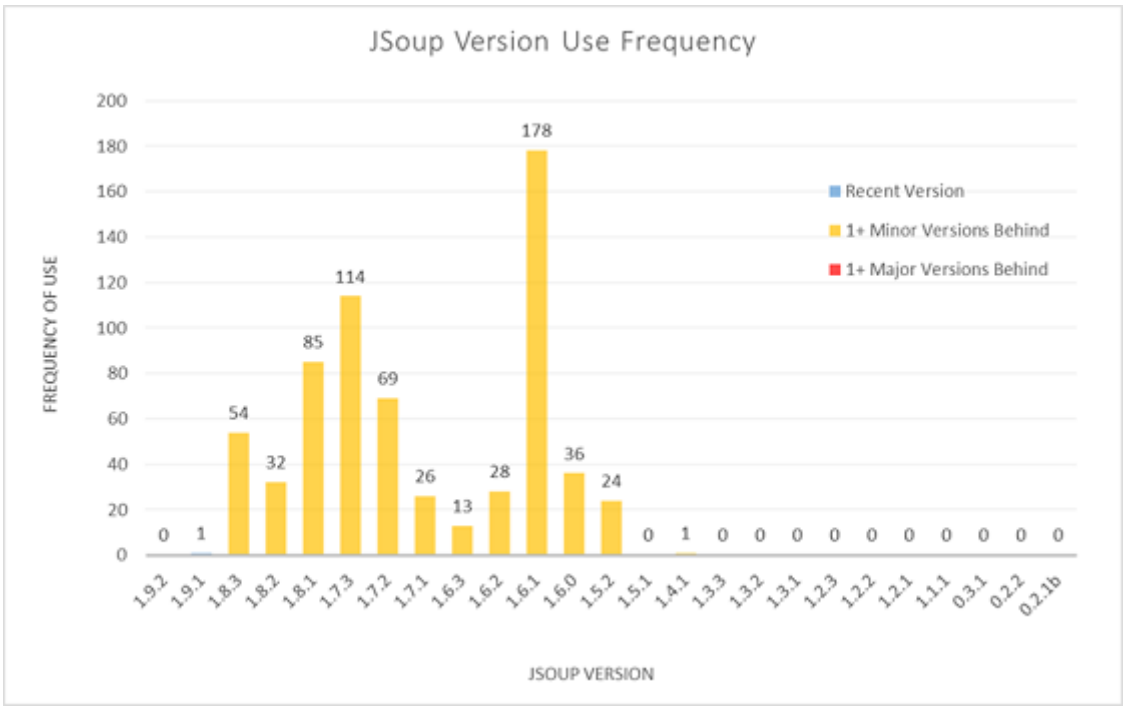

FIGURE 5.11: Distribution of APK's Using Versions of JSoup

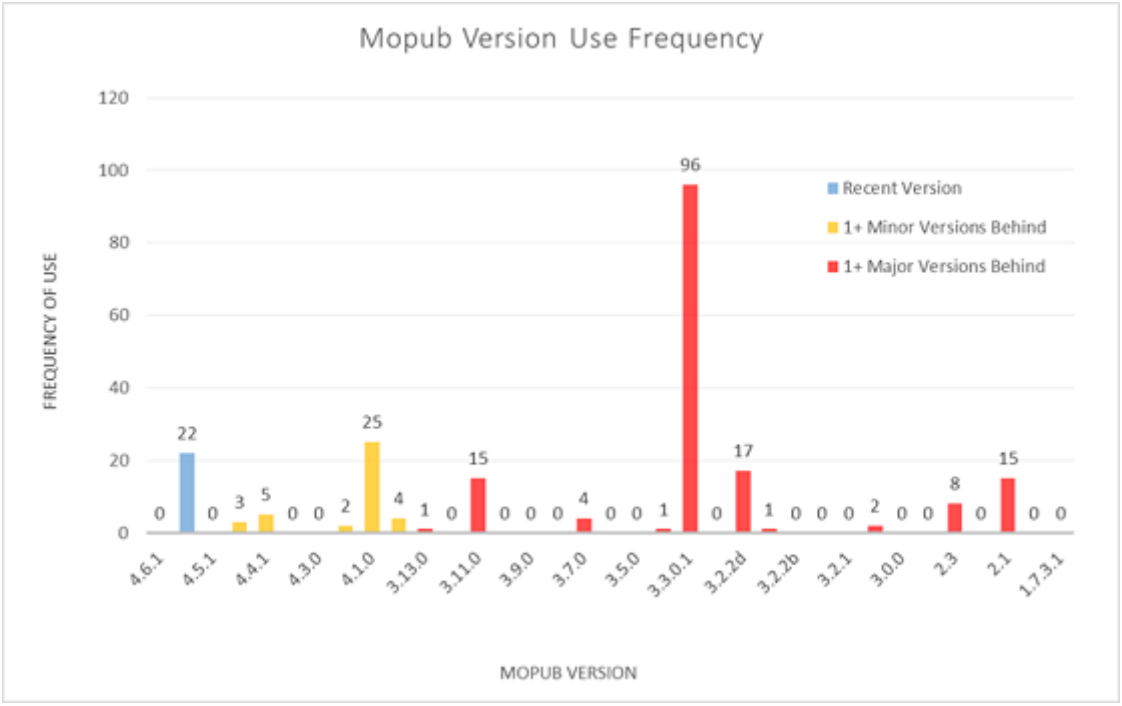

FIGURE 5.12: Distribution of APK's Using Versions of Mopub

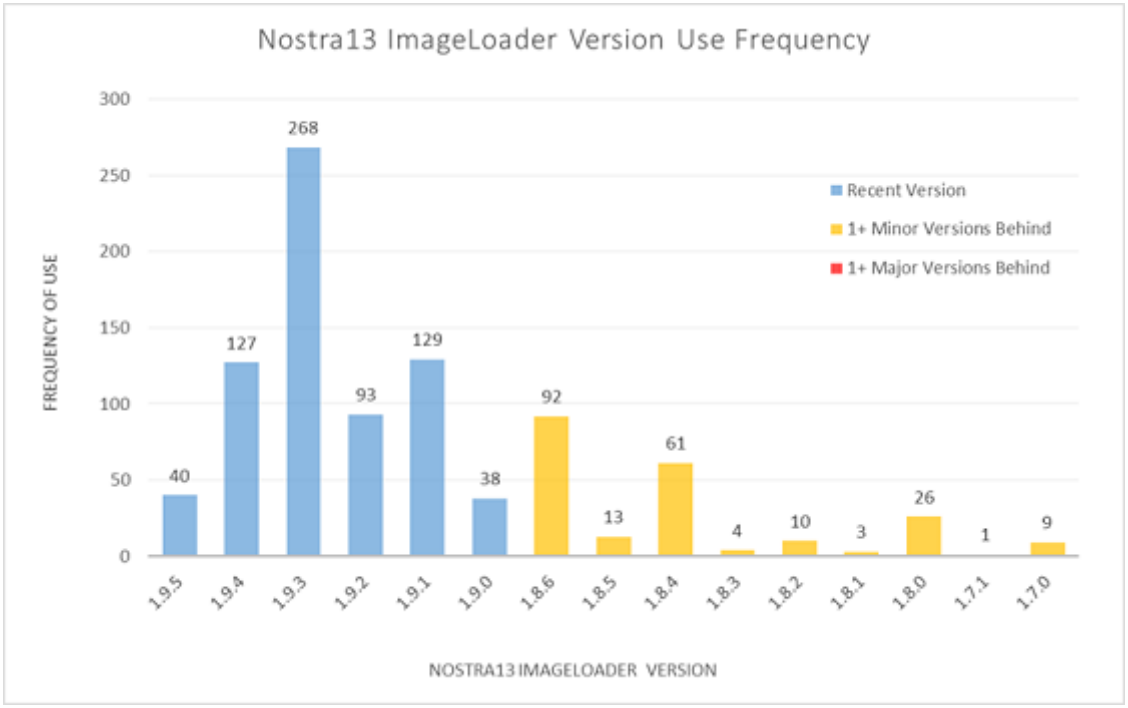

FIGURE 5.13: Distribution of APK's Using Versions of Nostra13 ImageLoader

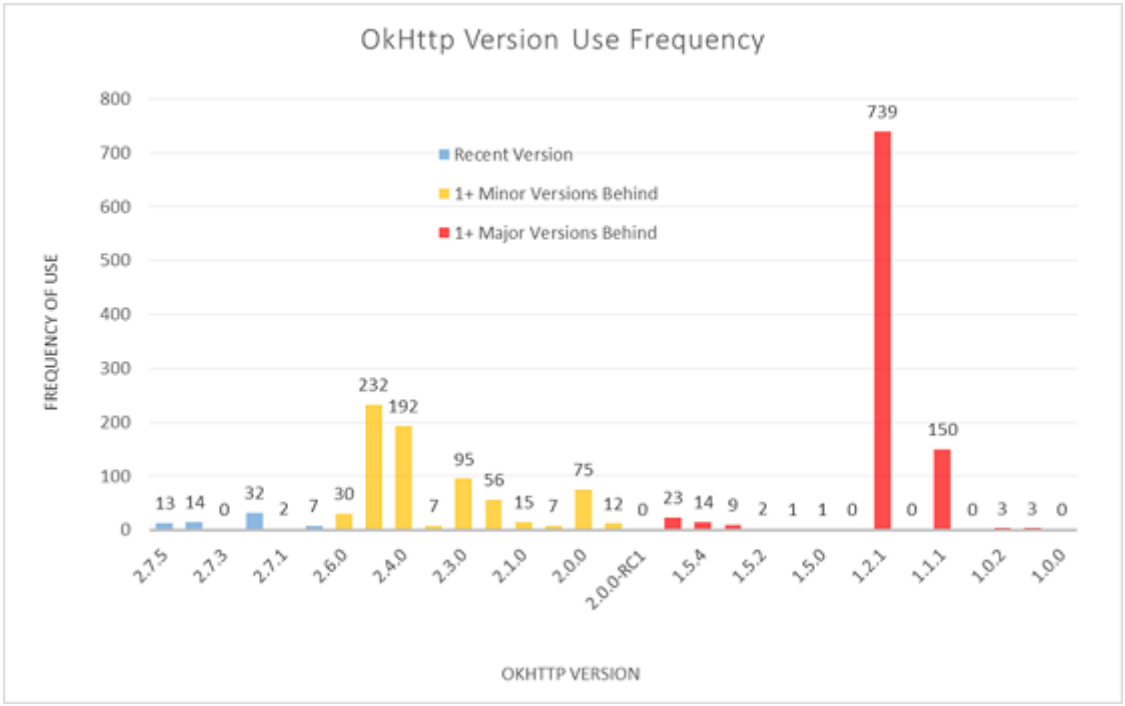

FIGURE 5.14: Distribution of APK's Using Versions of OkHttp

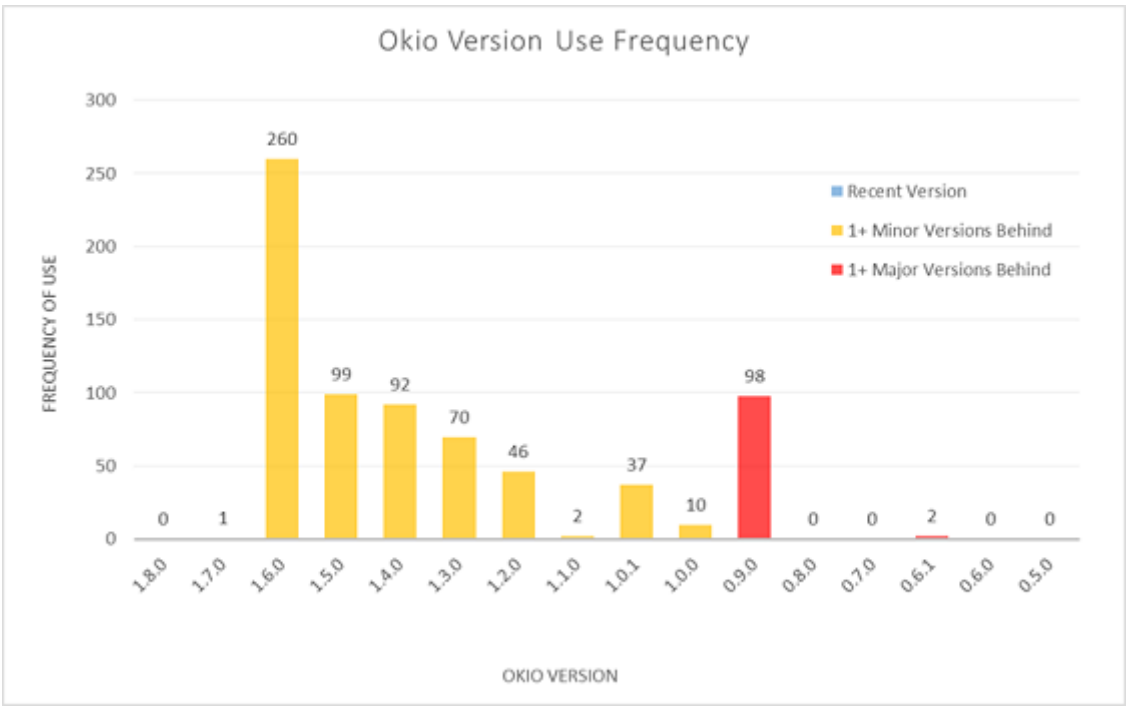

FIGURE 5.15: Distribution of APK's Using Versions of Okio

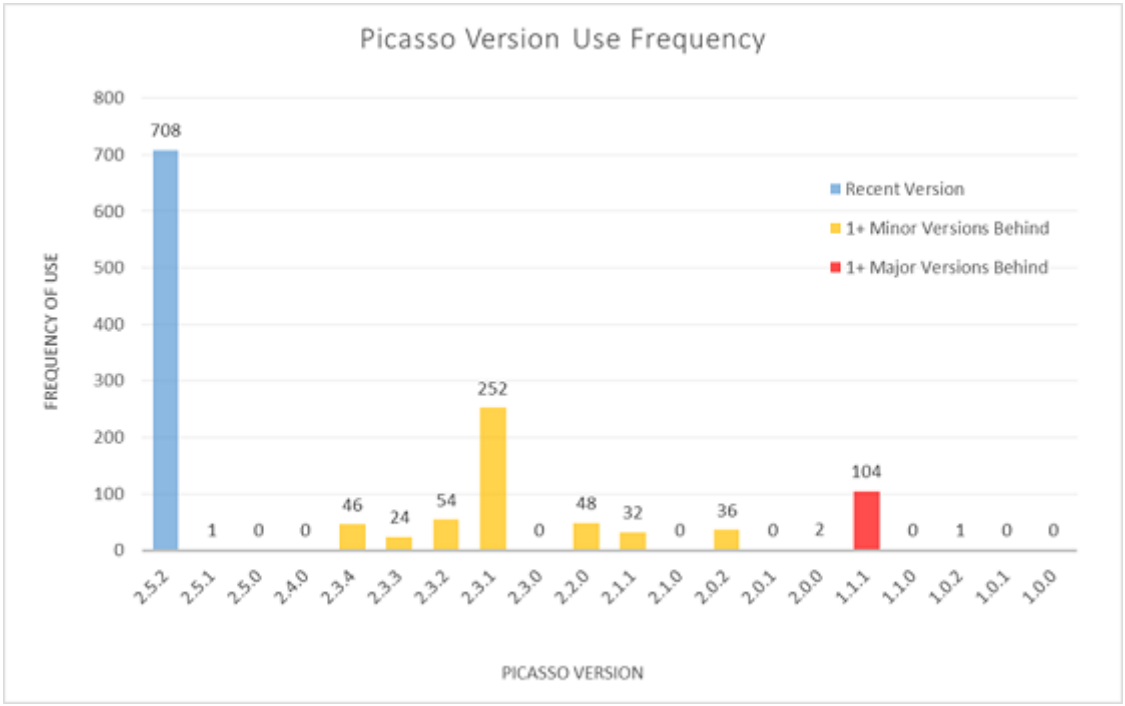

FIGURE 5.16: Distribution of APK's Using Versions of Picasso

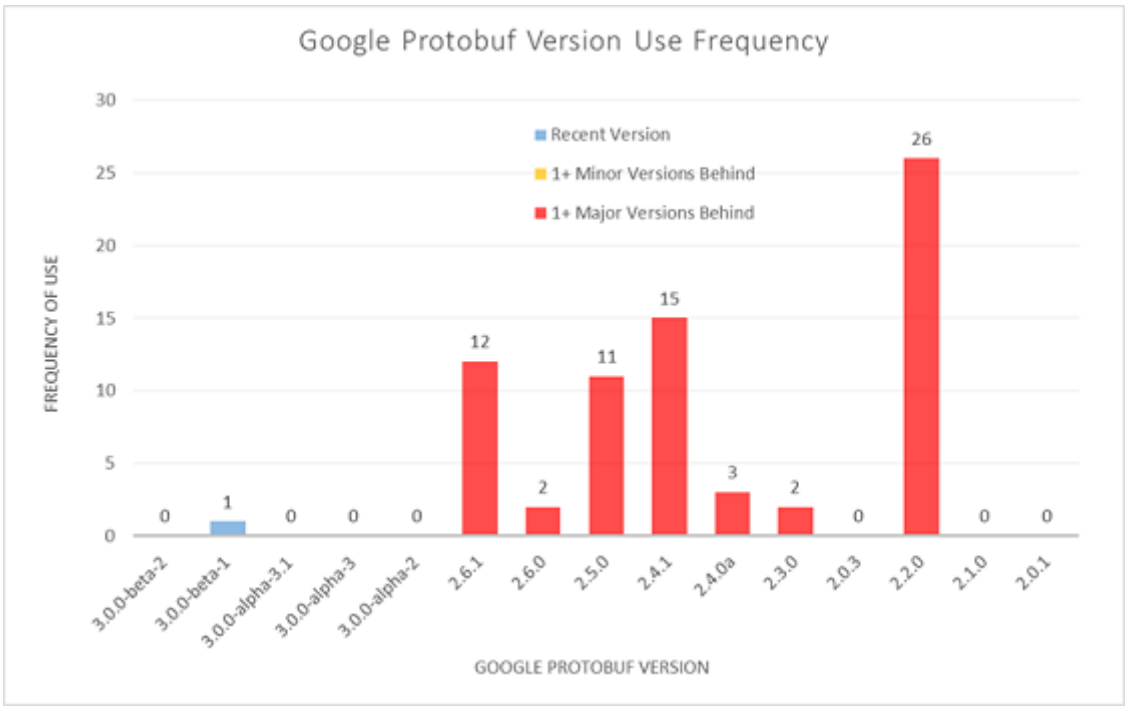

FIGURE 5.17: Distribution of APK's Using Versions of Google Protobuf

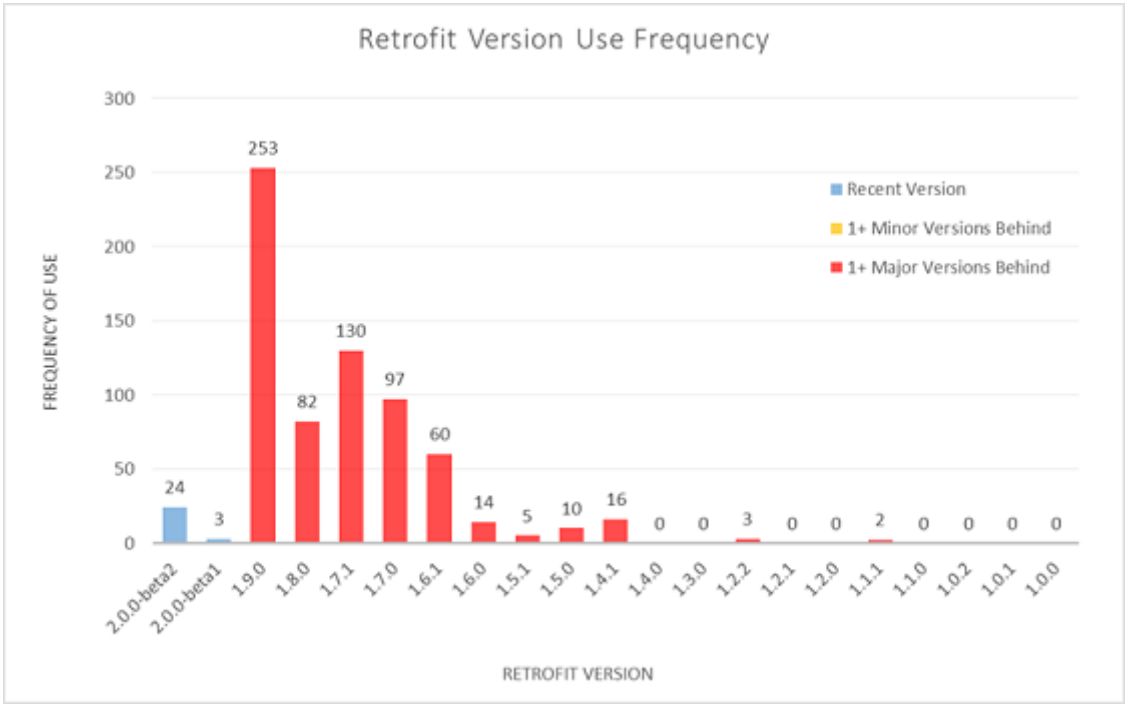

FIGURE 5.18: Distribution of APK's Using Versions of REtrofit

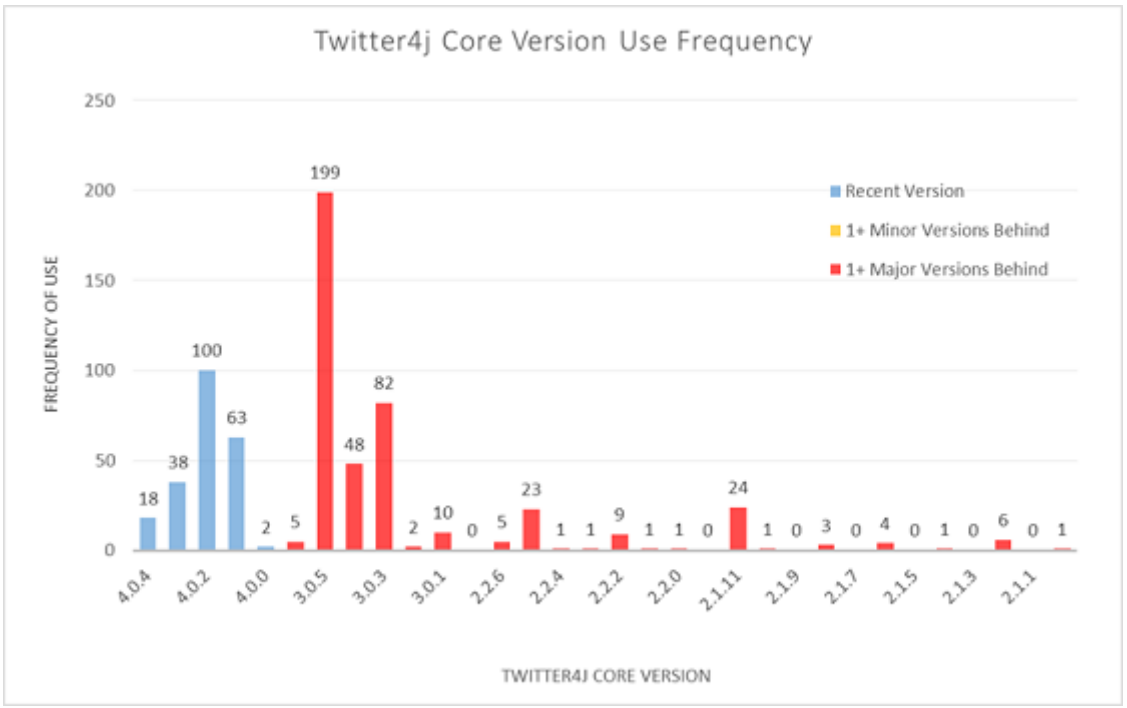

FIGURE 5.19: Distribution of APK's Using Versions of Twitter4jCore

Chapter 6

Impact of Outdated Library Use in Android Applications

6.1 Motivation

The results we gathered using our LibDetector tool show that many applications on the Google Play Store today do in fact continue to use outdated versions of libraries. In many cases, this is true even when newer libraries are available to the applications' developers at their time of release or update. In this chapter, we perform an exploratory study of whether or not this behavior impacts the apps.

6.2 App Rating

Mojica Ruiz et al. found that including certain ad libraries in an Android application negatively impacted the ratings of the application(Mojica Ruiz, Nagappan, Adams, Berger, et al., 2014). Therefore, we were interested in seeing whether or not the same holds true for Android applications that use significantly outdated versions of libraries.

Methodology

Our hypothesis is that there is a difference between the ratings of apps that use more recent versions of libraries versus those that use more outdated versions of libraries. Therefore, the null hypothesis is as follows:

H_0: There is no difference between the ratings of apps that use more recent versions of libraries versus those that use more outdated versions of libraries.

To test our null hypothesis, we first filter out all apps with less than 10 ratings from our results. We do this filtering in order to reduce rating bias. This is because the fewer the ratings there are, the more likely it is for biased ratings to skew the average, resulting in an rating that is unrepresentative of the actual rating of the app.

For each library, we split the APKs consuming the library into two groups. We put the APKs that contain a more recent version of the library in the "Recent" group, and APKs that contain a more outdated version of the library in the "Outdated" group. We tried to split the APKs in half based on the version of the library they were using. In the event that an APK is using a version that falls between the two groups, it is placed into the group that results in the groups being most equal in size. This is because it does not make sense for different apps using the same version of the same library to be place into different groups.

Results

Table 6.1 shows the results of the Mann Whitney test comparing the ratings of apps in the Recent versus the Outdated group. We see that in 14/16 of the libraries, there is no statistically significant difference between the two groups. However, for Retrofit and OkHttp, there was a significant difference. Interestingly, for Retrofit the Recent group had a significantly lower rating than the Outdated group. For OkHttp, the opposite was true.

RQ2: Does Using Outdated Software Libraries Affect the rating of an Android Application?

Based on the results of table 6.1, we are able to reject our null hypothesis that there is no difference between the ratings of apps that use more recent versions of libraries versus those that use more outdated versions of libraries. We conclude that in some cases, including outdated versions of certain libraries in an app can affect the rating of the app.

TABLE 6.1: Mann-Whitney Test for Ratings of Apps Using Newer vs Older Versions of a Library

Library Name	n Recent	Med. Rating Recent	n Outdated	Med. Rating Outdated	Z-Score	P-Value
Retrofit	243	4.1662	216	4.3082	-2.941	0.003
Okhttp	483	4.2655	494	4.1325	-2.896	0.004
HttpComponents HttpClient	103	4.2216	112	4.0903	-1.729	0.084
Acra	74	4.2139	136	4.0671	-1.496	0.135
Commons HttpClient	14	4.0205	6	4.0371	-1.155	0.248
Apache Commons IO	195	4.1539	191	4.0881	-1.153	0.249
Jsoup	184	4.2137	176	4.1520	-1.057	0.290
Facebook Android SDK	559	4.1951	722	4.2265	-0.987	0.323
Google Protobuf	31	4.3486	24	4.1779	-0.950	0.342
Google OAuth Client	58	4.2656	13	4.2222	-0.847	0.397
Okio	260	4.2142	217	4.1793	-0.733	0.463
Nostra13	297	4.2237	305	4.1837	-0.580	0.562
Mopub	72	4.3165	96	4.2327	-0.497	0.619
Twitter4j Core	150	4.1665	243	4.1632	-0.339	0.735
Picasso	497	4.2500	362	4.2559	-0.093	0.926
Gson	718	4.1905	750	4.1619	-0.002	0.999

6.3 App Exploitation

We have seen several reports and instances where well known and trusted libraries are found to have sever security flaws that put the security of their users at risk(Constantin, 2014; Constantin, 2015; *Arbitrary remote code execution with InvokerTransformer* 2015; *The FAT Attack. Facebook Social Login Session Hijacking Vulnerability* 2014; *Vulnerabilities* 2016). However, the mobile software domain is a rapidly evolving one. Perhaps the flaws and vulnerabilities that once posed a threat to the security of mobile software have been mitigated. Another possibility is that app developers are responsibly using vulnerable software libraries and designing with the exploits in mind. In this section, we perform a case study on the apps using the Facebook Android SDK to examine whether or not apps are inheriting vulnerabilities from vulnerable versions of the Facebook SDK.

6.3.1 Case Study: Exploiting Applications Using Outdated Facebook Android SDK

Methodology

Due to the Facebook vulnerability reports by Mi3Security, we are aware of a security vulnerability with Android applications that use the Facebook SDK 3.15.0 and earlier (*The FAT Attack. Facebook Social Login Session Hijacking Vulnerability* 2014). Given our LibDetector results, we have already determined the apps that contain a version of Facebook Android SDK. We see that in total, there are almost 300 apps that contain a version of the Facebook Android SDK version 3.15.0 and earlier.

To test whether or not these apps are exploitable because of this documented flaw in the Facebook Android SDK library, we recruited the help of a PhD student(we will refer to this student as the evaluator) from RIT specializing in mobile security.

In order to minimize any biases in the evaluation of the app for vulnerabilities, we did not tell the evaluator any details about the research. The request made to the evaluator was simple: given a set of apps known to be using a vulnerable version of Facebook SDK, try to exploit the app using the exploit method described by Mi3Security.

We randomly selected 28 apps that are potentially vulnerable because of the Facebook library they use, and ask the evaluator to try to break into them given the method described by Mi3Security. We also randomly selected 30 apps using Facebook Android SDK version 4.x, and gave the evaluator the same instructions.

Results

A few of the apps were indeed exploitable because of the security flaw that is introduced by Facebook. More specifically, 5 out of the 28 apps using Facebook Android SDK version 3.14.1 were able to be exploited. The results for the rest of the 23 apps are as follows:

- 11/28 contained Facebook SDK but made no calls to it
- 7/28 were vulnerable, but not because of Facebook
- 3/28 required an update upon startup

- 2/28 were unresponsive

The results for the 30 apps that using version 4.x of the Facebook Android SDK are as follows:

- 0/30 were vulnerable because of outdated Facebook
- 13/30 contained Facebook SDK but made no calls to it
- 13/30 had no vulnerability detected
- 2/30 were unresponsive
- 1/30 were vulnerable, but not because of Facebook
- 1/30 crashed upon logging in via Facebook

Discussion

It is possible that software libraries will never be completely free of vulnerabilities and that we might never find all of the ways in which a library is vulnerable. In fact, understanding that software flaws are sometimes inevitable despite our best efforts, some organizations like Facebook and Google have even started bug bounty programs to award people for actively searching for and reporting bugs and security vulnerabilities in their software(*Facebook Bug Bounty Program* 2016; *Google Security Reward Programs* 2016).

However, we think it is reasonable to assume that the older a library is, the more exposure the public has had with it. This means it is more likely that an older version of a library will have more known vulnerabilities associated with it. Therefore, we make the argument that the more outdated the version of a library in an app is, the more vulnerable it is. This is consistent with our results which show that for a known vulnerability, only the older versions of the library are vulnerable. None of the apps using the updated Facebook Android SDK version 4.x were found to be vulnerable by the exploit identified in our study. However, without more empirical data, this is purely conjecture at this time. It is possible that for example Facebook Android SDK version 4.0.0 is the version that contains the largest number of vulnerabilities, but that we do not know about them. What our results do show at this time is that for the 5

apps that not only contain a vulnerable version of Facebook Android SDK but also actually make calls to its Login API, all 5 became vulnerable. Therefore, apps with vulnerable versions of a library can very well inherit the vulnerability.

Interestingly, many of the apps that LibDetector identified as using the Facebook Android SDK library did not seem to actually implement any of the library's functionality in their app. To the user, it looked like there was no integration with Facebook whatsoever. We double checked the apps by decompiling their source code, and saw that they did in fact contain the code for the Facebook library. We think the explanation for this is that some app developers are using a templating strategy or automatic tool to set up their app, but do not use all of the libraries included in that template. Considering how popular Facebook integration is, it would not be surprising that such a template would provide the Facebook library by default.

RQ3: Are Android applications that use a vulnerable version of a library vulnerable?

The results of our small case study show that apps that apps using a vulnerable version of a library can very well inherit the vulnerability and become exploitable.

Threats to Validity

One major threat to the validity of this case study is that we rely exclusively on one person's ability to evaluate the exploit-ability of an Android app. While this person was chosen because of their domain knowledge in mobile security, there is nonetheless room for human error which could affect the validity of the results. Unfortunately, we did not have other evaluators which would have helped to minimize error and improve our confidence in the results.

6.4 Summary

In summary for this chapter, we conclude that in some cases including outdated versions of certain libraries in an app can affect the rating of the app. However, their causality and the direction of the relationship is still unclear at this time.

We also conclude that apps using vulnerable versions of a library do become exploitable. In fact, 5/5 apps that contain a vulnerable version of Facebook Android SDK and actually made calls to it were exploitable.

Chapter 7

Related Work

There are three main areas of research that our work is closely related to. They are Software Bertillonage, code clones, and mobile ad libraries.

7.1 Software Bertillonage

Davies et al. pioneered the idea of Software Bertillonage(Davies et al., 2013), which our work is most closely related to and draws inspiration from. The idea of Software Bertillonage is to narrow down the search space when trying to match a software component against a large number of possible candidates. Their motivation was to be able to determine artifact provenance, or the origin of a software component, in any given application. For their work, Davies et al. focused specifically on software libraries in Java applications.

In order to implement Software Bertillonage, Davies and colleagues had to first devise a strategy for measuring the components so that they can be compared. The technique they proposed was to measure and compare the class files using an "anchored class signature". To construct the signature, they modify the java compiler javac to extract method and field declarations in a class file. In order to make comparison of these class signatures more efficient, they use an SHA1 hash function to get the hash value of the class, instead of comparing whole string literals. They found that this method was effective in narrowing down the potential library matches in a target system to a manageable number of candidates, which could then be manually inspected to determine the best match.

Ishio et al. extended the work by Davies et al. by introducing a greedy algorithm to identify the best library version match in a software systen(Ishio et al., 2016). Their

work is based on the assumption that the best library JAR match should be the one that contains as many of the same classes that the target system has as possible. For example, imagine that a target system contains class files A, B, C, and D. Now consider two libraries - library 1 contains an match to class files A and C, while library 2 contains an exact match to class files A, C and D. In Davies et al.'s approach, both library 1 and library 2 would be reported as potential matches, and it would be up to the person interpreting the results to determine which library is actually used in the target system. However, Ishio et al. propose that we can avoid this extra effort. They postulate that the clear match here is library 2 since it accounts for a greater proportion of the class files found in the target system.

Although we also employ a Software Bertillonage technique towards identifying the versions of libraries being used in software systems, our research differs from the work by Davies et al. and Ishio et al. in a few major ways. First, we focus on Android applications. Secondly, we use Javap to extract a class signature directly from binary code instead of using Java decompilers to decompile the binary code and parse through the decompiled source code. This is not only more efficient, but also avoids introducing potential code anomalies during the decompiling and parsing process. Finally, we propose using edit distance instead of signature hashing to compare the similarity of two classes. While hash comparisons are very efficient for determining whether two files are the same or not, it is very sensitive to minor code alterations. There are several factors that can contribute to code differences. For example, different compilers optimize code differently, resulting in code that looks different but behaves the same. Code obfuscation can also drastically alter the code while preserving the functionality of a class. By using edit distance, we ensure that our technique is flexible in dealing with code alterations. Furthermore, it serves as a good metric of how dissimilar two libraries are if there are differences in them.

7.2 Clone Detection

Although researchers have identified many ways for code clones to be created, the term "code clone" typically refers to code that has been copied and pasted in a software system, with or without minor modifications(Roy and Cordy, 2009). Therefore,

code clones are essentially another means of code reuse. Over the years, researchers have actively studied code clones and produced a wide variety of strategies for detecting them in a software system. Here, we present a summary of some of the most influential works in code cloning research.

One of the first approaches proposed for clone detection is the textual approach. This approach is very direct and straightforward - the source code of a system is taken as-is for analysis and comparison. In some cases, the source code may be transformed to normalize the code and aid with text comparison. For example, the leading and trailing white space in lines of code might be removed. Johnson et al. were one of the earliest pioneers of textual clone detection(J. H. Johnson, 1993). More recently Roy presented NiCad, a textual clone detector to find near miss clones(clones with minor to significant modifications)(Roy, 2009). To detect near miss clones, NiCad implements a longest common sub-sequence algorithm. It also takes advantage of flexible pretty printing to pre-process the code and ensure consistent formatting. This helps to minimize the effect of structural differences in clones.

Another popular approach that has been proposed for clone detection is the lexical(also known as the token-based) approach. Like NiCad, the main purpose of this approach is to allow for the detection of code clones where the clone has been slightly altered. However in the lexical approach, a lexical parser is first used to parse the source code, apply necessary transformations, and convert the code into a stream of tokens that represents the code. Baker was one of the first to present an effective token-based clone detector with Dup(Baker, 1995). Since then, Kamiya et al. have developed CCFinder, one of the most influential token-based clone detectors to date(Kamiya, Kusumoto, and Inoue, 2002). They use a suffix tree algorithm with tokens as nodes on the tree to represent the entire application. This allows them to compare suffix trees of different software systems. Wherever segments of the suffix trees match, they conclude that the segment is a code clone snippet.

The major distinction between our work and code clone detection is that our technique is optimized for detecting libraries in mobile applications. Furthermore, our technique does not operate on the entire source code. In order to generate the class signature which we use to compare classes in a library against classes in an app, we only really care about its method and field declarations. We do not need to consider

the code that is inside the method body at all. In fact, unlike clone detection techniques, our approach does not even require us to have source code. This saves a lot of computational effort since the amount of information we need to compare is drastically reduced. This is a huge advantage since the scalability of clone detection tools has always been a challenge due to the amount of processing and parsing they often need to perform on the entire source code(Sajnani, 2016). Not requiring source code is also very beneficial when we consider the fact that we will likely not have access to the source code of Android apps.

Of notable mention is the CloneTracker tool developed by Duala-Ekoko and Robillard for the purposes of simplifying the process of code clone maintenance(Duala-Ekoko and Robillard, 2007). This is a tool that applies clone detection in order to help developers track and maintain code clones in their software, which is along the same veins as our research goals. In order to use CloneTracker, developers must first use any clone detection tool to find clones within their project. The code clone output from the clone detection tool is then fed into CloneTracker, and developers select the code clones that they are interested in tracking. When any of the clones are modified, the developers receive a notification. This allows them to decide whether or not the same modifications should be made to all the clones. With this process, CloneTracker is able to ease the maintenance of code clones within a project. However, while these researchers developed CloneTracker as a means for keeping track of code that is cloned internally within a project, our focus is on code being reused in Android applications in the form of external third party libraries.

7.3 Mobile Ad Libraries

Ad libraries are a subset of software libraries. They have been extensively researched due to the unique privacy and security concerns that they present. Most of the research on ad libraries examines how these libraries affect user privacy. This is because ad libraries often require access to personal data in order to serve targeted ads to its users.

Recently, Lin et al. proposed a crowd-sourcing approach to determine which sensitive resources an app should have access to(Lin, Amini, Hong, et al., 2012). By leveraging Amazon's Mechanical Turk system, they were able to survey over 5,000 participants about the resource access that the participants expect an app to have. This helped to reveal odd/suspicious access to resources in apps that a user would not expect, such as a simple flashlight app requiring access to Network Location information. The same researchers followed up on this work with Privacygrade, where they used static code analysis with Androguard to determine the libraries being used by apps(Lin, Amini, Luan, et al., 2014). By identifying which libraries are in an app and mapping that to the permissions that the libraries require, they were able to deduce many of the resource permissions that the app will require. When combined with their crowd-sourcing technique for determining the expected resource permissions, they are able to determine the degree of discrepancy between the expected and actual resource permissions in a mobile application. However, they are not concerned with the version of the library being used, which is a main focus in our research.

Researchers have also looked at how ad libraries may affect user acceptance and satisfaction of apps by looking at the app rating as an approximate indicator. Mojica Ruiz et al. found that unsurprisingly, more intrusive ad libraries tend to cause more user dissatisfaction and lower ratings for the app(Mojica Ruiz, Nagappan, Adams, Berger, et al., 2014). The results from this study inspired us to examine whether or not the version of a library has an effect on the ratings of an application.

Chapter 8

Conclusion

8.1 Summary

In this thesis, our main goal was to inform Android stakeholders of the library dependencies of Android applications and to explore the current trends in Java library use. Our work culminated in the creation of the LibDiff and LibDetector tools, which present an automated solution towards the efficient and accurate identification of specific versions of libraries being used by Android apps.

We found that many applications that are available on the Google Play Store do in fact use outdated versions of libraries. Next, we saw that in some cases the version of a library used by an Android app can affect the rating of the app. Finally, we empirically validated that there are tangible security consequences to using vulnerable versions of a library.

8.2 Future Research Directions

Due to the small scale and exploratory nature of much of our work, we would like to extend it by performing our evaluation and analyses on a much larger sample to mitigate the threats to validity that we have identified.

Furthermore, given our findings one very important area of research would be whether or not we can map changes to libraries, especially security fixes, to the versions of the library. When used in conjunction with LibDetector's ability to identify versions of libraries in Android applications, it would help to bridge the knowledge

gap between software library developers and the consumers of the libraries. We believe that this would be prove to be extremely beneficial in helping developers manage their software dependencies and prioritizing their effort.

Bibliography

Abrahamsson, Pekka et al. (2004). "Mobile-D: An Agile Approach for Mobile Application Development". In: *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '04. Vancouver, BC, CANADA: ACM, pp. 174–175. ISBN: 1-58113-833-4. DOI: 10.1145/1028664.1028736. URL: http://doi.acm.org.ezproxy.rit.edu/10.1145/1028664.1028736.

Android 6.0 Changes (2015). Android Developers. URL: https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html#behavior-apache-http-client (visited on 02/16/2016).

Arbitrary remote code execution with InvokerTransformer (2015). Apache Commons Collections. URL: https://issues.apache.org/jira/browse/COLLECTIONS-580 (visited on 06/20/2016).

Baker, B. S. (1995). "On finding duplication and near-duplication in large software systems". In: *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pp. 86–95. DOI: 10.1109/WCRE.1995.514697.

Bavota, Gabriele et al. (2015). "How the Apache community upgrades dependencies: an evolutionary study". In: *Empirical Software Engineering* 20.5, pp. 1275–1317. ISSN: 1573-7616. DOI: 10.1007/s10664-014-9325-9. URL: http://dx.doi.org/10.1007/s10664-014-9325-9.

Constantin, Lucian (2014). *Flaws in third-party software libraries often find their way into products, a problem that will occupy developers and sysadmins next year*. URL: http://www.computerworld.com/article/2864053/hey-devs-those-software-libraries-arent-always-safe-to-use.html (visited on 06/19/2016).

— (2015). *Developers often unwittingly use components that contain flaws*. URL: http://www.itworld.com/article/2936575/security/software-applications-

have-on-average-24-vulnerabilities-inherited-from-buggy-components.html (visited on 06/19/2016).

Davies, Julius et al. (2013). "Software Bertillonage". In: *Empirical Software Engineering* 18.6, pp. 1195–1237. ISSN: 1573-7616. DOI: 10.1007/s10664-012-9199-7. URL: http://dx.doi.org/10.1007/s10664-012-9199-7.

Dig, Danny and Ralph Johnson (2006). "How do APIs evolve? A story of refactoring". In: *Journal of Software Maintenance and Evolution: Research and Practice* 18.2, pp. 83–107. ISSN: 1532-0618. DOI: 10.1002/smr.328. URL: http://dx.doi.org/10.1002/smr.328.

Duala-Ekoko, E. and M. P. Robillard (2007). "Tracking Code Clones in Evolving Software". In: *29th International Conference on Software Engineering (ICSE'07)*, pp. 158–167. DOI: 10.1109/ICSE.2007.90.

Facebook Bug Bounty Program (2016). Facebook. URL: https://www.facebook.com/whitehat (visited on 04/20/2016).

Google Security Reward Programs (2016). Google Application Security. URL: https://www.google.com/about/appsecurity/programs-home/ (visited on 04/20/2016).

Ishio, Takashi et al. (2016). "Software Ingredients: Detection of Third-party Component Reuse in Java Software Release". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Austin, Texas: ACM, pp. 339–350. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2901773. URL: http://doi.acm.org.ezproxy.rit.edu/10.1145/2901739.2901773.

Johnson, J. Howard (1993). "Identifying Redundancy in Source Code Using Fingerprints". In: *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*. CASCON '93. Toronto, Ontario, Canada: IBM Press, pp. 171–183. URL: http://dl.acm.org/citation.cfm?id=962289.962305.

Kamiya, Toshihiro, Shinji Kusumoto, and Katsuro Inoue (2002). "CCFinder: A multilinguistic token-based code clone detection system for large scale source code". English. In: *IEEE Transactions on Software Engineering* 28.7. Copyright - Copyright Institute of Electrical and Electronics Engineers, Inc. (IEEE) Jul 2002; Last updated

- 2011-07-20; CODEN - IESEDJ, pp. 654–670. URL: http://search.proquest.com.ezproxy.rit.edu/docview/195579324?accountid=108.

Kapser, Cory J. and Michael W. Godfrey (2008). ""Cloning considered harmful" considered harmful: patterns of cloning in software". In: *Empirical Software Engineering* 13.6, pp. 645–692. ISSN: 1573-7616. DOI: 10.1007/s10664-008-9076-6. URL: http://dx.doi.org/10.1007/s10664-008-9076-6.

Lehman, M. M. (1980). "Programs, life cycles, and laws of software evolution". In: *Proceedings of the IEEE* 68.9, pp. 1060–1076. ISSN: 0018-9219. DOI: 10.1109/PROC.1980.11805.

Li, Li et al. (2016). "An Investigation into the Use of Common Libraries in Android Apps". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1, pp. 403–414. DOI: 10.1109/SANER.2016.52.

Lin, Jialiu, Shahriyar Amini, Jason I. Hong, et al. (2012). "Expectation and Purpose: Understanding Users' Mental Models of Mobile App Privacy Through Crowdsourcing". In: *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. UbiComp '12. Pittsburgh, Pennsylvania: ACM, pp. 501–510. ISBN: 978-1-4503-1224-0. DOI: 10.1145/2370216.2370290. URL: http://doi.acm.org/10.1145/2370216.2370290.

Lin, Jialiu, Shahriyar Amini, Song Luan, et al. (2014). *PrivacyGrade: Grading The Privacy Of Smartphone Apps*. URL: http://www.privacygrade.org/ (visited on 06/01/2016).

Linares-Vásquez, Mario et al. (2014). "Revisiting Android Reuse Studies in the Context of Code Obfuscation and Library Usages". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, pp. 242–251. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597109. URL: http://doi.acm.org.ezproxy.rit.edu/10.1145/2597073.2597109.

Mojica Ruiz, I. J., B. Adams, et al. (2014). "A Large-Scale Empirical Study on Software Reuse in Mobile Apps". In: *IEEE Software* 31.2, pp. 78–86. ISSN: 0740-7459. DOI: 10.1109/MS.2013.142.

Mojica Ruiz, I. J., M. Nagappan, B. Adams, T. Berger, et al. (2014). "Impact of Ad Libraries on Ratings of Android Mobile Apps". In: *IEEE Software* 31.6, pp. 86–92. ISSN: 0740-7459. DOI: 10.1109/MS.2014.79.

Mojica Ruiz, I. J., M. Nagappan, B. Adams, and A. E. Hassan (2012). "Understanding reuse in the Android Market". In: *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pp. 113–122. DOI: 10.1109/ICPC.2012.6240477.

Number of available applications in the Google Play Store from December 2009 to February 2016 (2016). Statista. URL: http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ (visited on 06/23/2016).

Preston-werner, Tom (2016). *Semantic Versioning 2.0.0*. URL: http://semver.org/ (visited on 03/12/2016).

Robbes, Romain, Mircea Lungu, and David Röthlisberger (2012). "How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. Cary, North Carolina: ACM, 56:1–56:11. ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393596.2393662. URL: http://doi.acm.org.ezproxy.rit.edu/10.1145/2393596.2393662.

Roy, Chanchal K. (2009). *Detection and analysis of near-miss software clones*. English. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2016-05-29. URL: http://search.proquest.com.ezproxy.rit.edu/docview/762330560?accountid=108.

Roy, Chanchal K. and James R. Cordy (2009). "A survey on software clone detection research". English. In: *Science of Computer Programming* 74.7, pp. 470–495. URL: http://search.proquest.com.ezproxy.rit.edu/docview/1776183909?accountid=108.

Sajnani, Hitesh (2016). *Large-Scale Code Clone Detection*. English. URL: http://search.proquest.com.ezproxy.rit.edu/docview/1776183909?accountid=108.

The FAT Attack. Facebook Social Login Session Hijacking Vulnerability (2014). Mi3Security. URL: https://www.mi3security.com/the-fat-attack-facebook-social-login-session-hijacking/ (visited on 06/15/2016).

Thummalapenta, Suresh et al. (2010). "An empirical study on the maintenance of source code clones". In: *Empirical Software Engineering* 15.1, pp. 1–34. ISSN: 1573-7616. DOI: 10.1007/s10664-009-9108-x. URL: http://dx.doi.org/10.1007/s10664-009-9108-x.

Top Apps on Google Play, United States (2016). AppAnnie. URL: https://www.appannie.com/apps/google-play/top/ (visited on 03/01/2016).

Top Projects (2016). Maven Repository. URL: http://mvnrepository.com/popular (visited on 04/09/2016).

Vulnerabilities (2016). OpenSSL. URL: https://www.openssl.org/news/vulnerabilities.html (visited on 06/21/2016).

Wang, Haoyu et al. (2015). "WuKong: A Scalable and Accurate Two-phase Approach to Android App Clone Detection". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: ACM, pp. 71–82. ISBN: 978-1-4503-3620-8. DOI: 10.1145/2771783.2771795. URL: http://doi.acm.org.ezproxy.rit.edu/10.1145/2771783.2771795.