2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion

Identifying Android Library Dependencies in the Presence of Code Obfuscation and Minimization

Salman A. Baset
IBM Research
sabaset@us.ibm.com

Shih-Wei Li
Columbia University
shihwei@cs.columbia.edu*

Philippe Suter
IBM Research
philippe.suter@gmail.com*

Omer Tripp
IBM Research
trippo@google.com*

I. Introduction

The Android platform has gained tremendous popularity since it was unveiled in 2007. Currently it holds a global market share of 78.4% with 1.5M activations of Android devices per day. In the US alone, there are 76M Android users. The main Android app market, Google Play, offers over 2M apps with a history of over 50B downloads to date. These impressive statistics provide strong motivation to understand, and improve the quality of Android apps. In this work, we focus on a key question in this space, which to our knowledge has not been previously addressed; namely:

> *Can we determine the library dependencies of a given Android application without access to its source code?*

We emphasize that library dependencies include the *exact library version*, which is crucial information for many interesting use cases. First, an application may use a library (version) with known security vulnerabilities, or even worse, a library that is confirmed to be offensive if not malicious (e.g., an advertising library that make use of sensitive user information without proper authorization). Second, many libraries have known functional bugs, such as performance and memory problems [2]. Knowing the library dependencies of an application can help isolate bad behaviors exhibited by the application. Moreover, the ability to automatically extract the library dependencies of an application opens the door for large-scale studies on the use of libraries by Android apps, allowing insight into questions such as what the most popular libraries are, how their popularity changes across app categories, and how often a recent version of a library is used [4], [1].

Interestingly, though it may appear straightforward to compute the dependencies of an Android app without access to its source code, there are two serious challenges that complicate this task, sometimes to the point of rendering nonambiguous identification impossible. First, there is still strong incentive to reduce the size of Android apps (and mobile apps in general), also because many users still own legacy devices. Hence, through a process known as *Minimization*, Android apps typically do not include libraries in their entirety, but rather rely on reachability analysis to determine which parts of the library are used by the app (at the granularity of class members, i.e. fields and methods), such that only those class fragments are incorporated into the image, blurring the boundary between app and library code. Second, since mobile apps are downloaded onto end-user's devices, the end-users can disassemble and inspect the app which might violate sensitive intellectual property and perhaps even uncover security weaknesses that render other users vulnerable. Thus, the app developers apply a process known as *Obfuscation* on the app, which not only obfuscates the source code written by developers but also to library symbols, since the boundary between app and library classes is blurred due to minimization. By the end of minimization and obfuscation, the app's image, in the form of an `.apk` file, is largely a blob of obfuscated code with no immediate hints as to which libraries were incorporated into it. As an illustration, we refer the reader to Figure 1. Notice in particular the static call site highlighted in red with method identifier `La/a/a/a/c;.a`. This is in fact a call to another method from the same Apache Commons Codec class (cf. the clear version in blue).

Our work presents a solution to the challenges highlighted above in the form of MobScanner, a tool for identifying the precise versions of libraries used in the construction of an app without access to its source code. MobScanner employs a combination of information-retrieval and constraint-solving techniques to report an effective set of candidate library dependencies (including their version), even when the app is obfuscated and/or when the set of potential libraries is very large.

II. Technique

In this section, we explain the challenge in identifying the library dependencies of Android apps, namely because code obfuscation and minimization are by now an integral part of the Android build process. We then outline the main steps of our technique.

ProGuard and the Android Build Process. The Android build process packages an app into a single application package (.apk) file. The .apk file contains all the information necessary to run the app on either an emulator or a physical device. It includes compiled .dex files (Java .class files converted to

* The work was done while the authors were at IBM Research.

978-1-5386-1589-8/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-C.2017.79

```
if-eqz v3, 001d
const-string v0, "$apr1$"
invoke-virtual {v3, v0}, Ljava/lang/String;.startsWith:(Ljava/lang/String;)Z
move-result v0
if-nez v0, 001d
new-instance v0, Ljava/lang/StringBuilder;
invoke-direct {v0}, Ljava/lang/StringBuilder;.<init>:()V
const-string v1, "$apr1$"
invoke-virtual {v0, v1}, Ljava/lang/StringBuilder;.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
move-result-object v0
invoke-virtual {v0, v3}, Ljava/lang/StringBuilder;.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
move-result-object v0
invoke-virtual {v0}, Ljava/lang/StringBuilder;.toString:()Ljava/lang/String;
move-result-object v3
const-string v0, "$apr1$"
invoke-static v2,v3,v0,Lorg/apache/.../digest/Md5Crypt;.md5Crypt:([BLString;LString;)LString;
invoke-static v2,v3,v0, La/a/a/a/a/c;.a:([BLString;LString;)LString;
move-result-object v0return-object v0
```

Fig. 1. Dex bytecode to which the method in Figure 2 compiles without obfuscation, and delta due to obfuscation (the red line replaces the blue one). (Some type signatures have been simplified for readability.)

the Dalvik bytecode format), a binary version of the AndroidManifest.xml file, compiled resources (resources.arsc) and uncompiled resource files.

Recently the ProGuard tool, available as an open-source project,[1] has been assimilated into the build process. ProGuard shrinks, optimizes and obfuscates the code of Android applications. With Android Studio or the Gradle build system, for example, the `minifyEnabled` switch controls whether ProGuard is enabled in release builds. The ease of enabling ProGuard, and the benefit of obtaining a more optimized and protected app with a smaller image, are the reasons why many of the apps featured on Google Play are minimized and obfuscated.

The ProGuard reachability analysis is relatively aggressive in deciding which parts of the code are accessible from entry points, operating at a level as low as class members. As a simple illustration, consider the following synthetic example:

class C { void f() {...} void g() {...} }

If f() is seen by ProGuard to be transitively invoked by one or more of the entry-point methods, but the same is not true of g(), then C will be minimized into

class C { void f() {...} }

An analogous transformation is applied by ProGuard in the case of unused fields.

Identifying Library Dependencies. Both obfuscation and minimization complicate the task of identifying the exact library versions that an application is dependent on. Due to obfuscation, symbols are renamed. Due to minimization, classes and class members deemed unused are eliminated.

We tackle these challenges via a unified approach that consists of two main step. The input is an app A, where we assume the existence of a (comprehensive) database $\mathbb{D}$ of candidate library/version pairs. Repositories such as Bintray or Maven Central[2] enable the construction of such a database with relative ease. The first step is to compute a "signature" for each of the classes in A, and compare the signature against those extracted from the classes of the libraries contained in $\mathbb{D}$. Intuitively, the signature is a feature vector, where the features correspond to elements of the code that are resistant

[1] http://proguard.sourceforge.net

[2] https://bintray.com and https://search.maven.org

```
static final String APR1_PREFIX = "$apr1$";
public static String apr1Crypt(final byte[] keyBytes, String salt) {
  // to make the md5Crypt regex happy
  if (salt != null && !salt.startsWith(APR1_PREFIX)) {
    salt = APR1_PREFIX + salt;
  }
  return Md5Crypt.md5Crypt(keyBytes, salt, APR1_PREFIX);
}
```

Fig. 2. Source code of apr1crypt method from class MD5Crypt in the Apache Commons Codec library

to obfuscation, such as instantiation of a core (i.e. java.⋆) or platform (i.e. com.android.⋆) class or use of a string constant.

While identification and extraction of obfuscation-resistant features is relatively straightforward, a key is assignment of weights to different features. As an intuitive example, we refer the reader to Figure 2. Like the MD5Crypt class in that example, many other classes are likely to allocate objects of type java.lang.StringBuilder, which the compiler instantiates to implement string concatenation. However, it is much less likely for another class to define a string constant with value $apr1$. Hence, this feature should be assigned a much higher weight for the purpose of signature matching.

To obtain an effective weighting scheme, we build on results in the area of information retrieval and text mining, specifically the tf-idf algorithm [3]. This algorithm computes the weight of a given feature in terms of (i) its intra-class frequency (i.e., its number of occurrences in the given class) vs (ii) its inter-class frequency (i.e., the number of other classes where it occurs). Intuitively, a feature has high discriminative power, and thus high weight, if it occurs frequently in the given class (e.g., the class makes multiple uses of the constant $apr1$) but infrequently in other classes (e.g., no other class makes use of this constant).

Given the ability to weight features, and thus perform class-level matching, the second main challenge is to lift the class-wise results to the level of library/version pairs. Simply declaring all libraries with classes that match strongly against app classes as dependencies, or doing so only for libraries with at least k strong matches, are overly coarse heuristics that are hard to justify. Indeed, the problem of selecting which library/version pairs to treat as dependencies calls for global reasoning (rather than e.g. greedily accounting for one class at a time). To perform such reasoning efficiently and effectively, we leverage the power of modern constraint solvers in specifying a constraint/optimization system that captures the essential considerations. These include (i) the constraint that two versions of the same library cannot simultaneously act as dependencies of the same app, (ii) the goal of "covering", via the set of dependencies, a maximal number of app classes for which there exist strong matches (other classes are assumed to originate from the app), as well as (iii) the goal of minimizing the overall imprecision of matching for selected libraries.

These requirements and objectives, and in particular the goal of maximizing coverage of appropriate app classes, bias toward high recall, a design choice suitable for clients such as security analysis and optimization. Our techniques achieve this goal, i.e. identify the exact *libraries and their versions* used in the apps; for clear apps the recall is almost perfect at 98%. For obuscated/minimized apps it stands at 85%.

REFERENCES

[1] PlayDrone APKs on archive.org. https://archive.org/details/playdrone-apks. Accessed Nov. 2015.

[2] Zhezhe Chen, Qi Gao, Wenbin Zhang, and Feng Qin. Flowchecker: Detecting bugs in mpi libraries via message flow checking. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[3] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[4] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of Google Play. In *SIGMETRICS*, pages 221–233. ACM, 2014.