Forensic Science International: Digital Investigation 39 (2021) 301285

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations

Xiaolu Zhang [a, *], Frank Breitinger [b, *], Engelbert Luechinger [c], Stephen O'Shaughnessy [d]

[a] Department of Information Systems and Cyber Security, University of Texas at San Antonio, San Antonio, TX, 78249, United States
[b] School of Criminal Justice, Faculty of Law, Criminal Justice and Public Administration, University of Lausanne, 1015, Lausanne, Switzerland
[c] Hilti Chair for Data and Application Security, Institute of Information Systems, University of Liechtenstein, Fürst-Franz-Josef-Strasse, 9490, Vaduz, Liechtenstein
[d] Department of Informatics, Technological University Dublin, Blanchardstown Campus, Dublin 15, Ireland

ARTICLE INFO

Article history:
Received 7 May 2021
Received in revised form
9 September 2021
Accepted 12 September 2021
Available online 6 October 2021

Keywords:
Android application forensic
Obfuscation
Deobfuscation
Obfuscation detection
Literature review
Survey
Reverse engineering

ABSTRACT

Android obfuscation techniques include not only classic code obfuscation techniques that were adapted to Android, but also obfuscation methods that target the Android platform specifically. This work examines the status-quo of Android obfuscation, obfuscation detection and deobfuscation. Specifically, it first summarizes obfuscation approaches that are commonly used by app developers for code optimization, to protect their software against code theft and code tampering but are also frequently misused by malware developers to circumvent anti-malware products. Secondly, the article focuses on obfuscation detection techniques and presents various available tools and current research. Thirdly, deobfuscation (which aims at reinstating the original state before obfuscation) is discussed followed by a brief discussion how this impacts forensic investigation. We conclude that although obfuscation is widely used in Android app development (benign and malicious), available tools and the practices on how to deal with obfuscation are not standardized, and so are inherently lacking from a forensic standpoint.

1. Introduction

Android is undoubtedly the most popular mobile device operating system (OS), holding a market share of 87% (Chau and Reith, 2019) with new Android apps or updates for existing ones appearing daily or even more frequently. Given this seemingly ever-increasing pace of production, there is inevitably a corresponding increase in the misuse of mobile technology and applications for nefarious purposes. As a direct consequence, law enforcement and forensics investigators struggle to cope with the digital evidence backlog (Hamilton, 2020). This is compounded by the pace with which the complexities of technology are growing and often, techniques and procedures can quickly become outdated. For instance, Zhang et al. (2017) manually reversed Android Vault applications to understand the encryption mechanisms. All these apps have been updated since then, making the work outdated (presented techniques most likely will not work anymore), yet these findings were utilized by local law enforcement to solve a case as no commercial tool at that time had the capability. Although in this case, investigators were lucky that Zhang et al. (2017) were working on this problem, we argue that examiners may get to the point where they have to manually analyze an application. While reverse engineering alone is already a challenge, obfuscation techniques add an additional burden and complicate the process. These techniques are frequently employed in the development of Android applications (benign and malicious) and are commonly found in the public domain on Google's Play Store. Wermke et al. (2018), for example, investigated the use of obfuscation

* Corresponding authors.
E-mail addresses: Xiaolu.Zhang@utsa.edu (X. Zhang), Frank.Breitinger@unil.ch (F. Breitinger), Engelbert.Luechinger@uni.li (E. Luechinger), Stephen.Oshaughnessy@tudublin.ie (S. O'Shaughnessy).

https://doi.org/10.1016/j.fsidi.2021.301285

techniques in the Google Play Store and analyzed a dataset of over 1.7 million apps. They concluded that approximately 25% of apps are using some obfuscation techniques, even reaching the ratio of 50%, when only considering the most popular apps with more than 10 million downloads.[1] Note, the Android developer studio recommends obfuscation (e.g., with `ProGuard`) for code shrinking and optimization. With respect to malicious applications, Zhou and Jiang (2012) showed that 86% of their tested malware samples were repacked versions of benign Android apps, proving that malware developers are using obfuscation techniques as well. Though, they usually use these methods with the intent to circumvent anti-malware products; a recent survey concluded that the use of obfuscation techniques significantly reduces the detection rate of malware detectors (Hammad et al., 2018). From an investigative perspective, we can differentiate between two major motives to dissect an app:

M1 The examiner suspects that the (benign) app contains evidence about a certain crime and there is currently no tool that can assist (e.g., photos locked away in a vault app).
M2 The examiner suspects that the app is malicious and s/he wants to access the underlying code and structure (note that an app can be malicious by design or a benign app that has been modified to be malicious).

Regardless of the motivation, an essential first step is to determine the utilized obfuscation method where some can be identified automatically. For instance, Mirzaei et al. (2019) introduced an obfuscation detector that can analyze an app and discover the applied obfuscation techniques. Nevertheless, these detectors are usually limited to a small number of obfuscation techniques. Depending on the utilized technique, an examiner might have to deobfuscate the app before performing static analysis. This process usually includes several manual steps, since the support of automated tools is sparse. Yoo et al. (2016), for example, introduced a deobfuscation schema that can decrypt encrypted strings; however, this idea never made it into actual implementation. As we see malware analysis as a subdomain of digital forensics, this paper aims at supporting forensic investigators and researchers by:

- providing a comprehensive literature overview to capture the current status-quo of obfuscation techniques preparing them for a possible manual analysis.
- providing a summary of obfuscation detection and deobfuscation techniques allowing practitioners to identify the most suitable approach/application for their problem.

Although several reviews and articles have already been published in this domain, their emphasis was primarily on Android obfuscation techniques, leaving the areas of detection and deobfuscation relatively unexplored. Furthermore, existing articles mainly focused on discovering the impact of obfuscation techniques for malware detection which is in contrast to this work where we consider the affect on forensic investigations.

Limitations. The articles have been searched manually and the changes on this topic happen frequently, therefore we do not claim to be complete. As indicated by the title, this work focuses on the Android platform and therefore techniques for other operating systems (e.g., Windows, iOS) may be different.

Outline. The remainder of this work is organized as follows. In Sec. 2, we introduce the package structure and the building process of Android applications followed by Sec. 3. The methodology of this research is covered next in Sec. 4. The core of this article are Sections 5, 6 and 7 which discuss Obfuscation techniques, Obfuscation detection and Deobfuscation, respectively. The related work is summarized in Sec. 8 followed by Impact of obfuscation on forensic investigations (Sec. 9). Sec. 10 concludes the paper.

[1] The fact that popular apps tend to use obfuscation more frequently may be attributed to companies being more concerned about intellectual property theft.

2. Android basics

This section outlines some basics of the Android platform which are essential to understand prior to discussing the obfuscation and deobfuscation techniques. In detail, we briefly summarize the Android package structure (Sec. 2.1) followed by the building process (Sec. 2.2).

2.1. Android package (APK) structure

An APK file is a compressed file format used by the Android platform as a container for an app(lication). According to developer.android.com, a typical APK file consists of the directories and files as depicted in Fig. 1:

lib directory contains the application's shared libraries compiled for different CPU architecture such as ARM and MIPS. Since the shared libraries are processor dependent, they are usually written in C.

res contains additional resources that are not already pre-compiled into the resources.asc file.

assets hold application-specific supplementary assets. These are usually raw files like images that can be loaded later as a byte-stream.

META-INF can be considered as an internal Java *meta* directory which should not contain user data. Common files are the developer signature file (similar to a certificate; named *.RSA or *.DSA); MANIFEST.MF which lists all files in the package including a cryptographic hash; and many more.

Besides these directories, there are several important files (technically, only the *AndroidManifest.xml* file is mandatory (Google, 2020c,a)):

AndroidManifest.xml contains essential details about the app such as the permissions and hardware requirements.

classes.dex is the executable file running on the Dalvik virtual machine (DVM) or the Android runtime (ART). Note that the DEX file usually contains the main part of the program and can easily be decompiled. Therefore, more and more applications start outsourcing important code into shared libraries.

resources.arsc contains pre-compiled resources and is usually used for performance reasons.

To the best of our knowledge, while the most well-known obfuscation techniques target DEX files and shared libraries, some

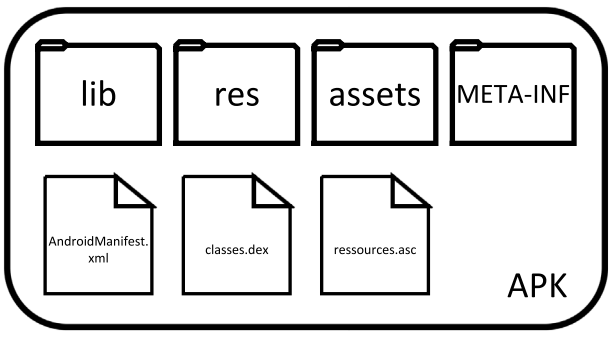

Fig. 1. Contents of an APK file.

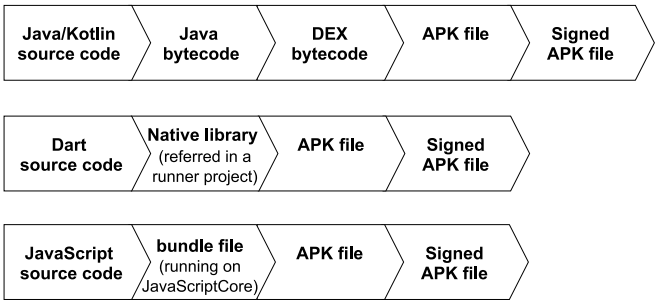

Fig. 2. APK build process.

obfuscation techniques are applied to application manifest and resource files. More details can be found in Sec. 5.

2.2. APK build process

To create an APK file, a specific build process has to be followed. This process is also explained in detail in the Android developer guide. Nevertheless, Fig. 2 shows a simplified version of the compilation process.

First, the source code (written in Java or Kotlin) together with other custom libraries is compiled with a compiler (*javac/kotlinc*), into Java bytecode (.class). In order to run the program in Android, the code has to be further compiled into DEX bytecode (.dex). Only in this structure, the application can run on DVM or ART (since Android V5.0). This format is then also used to distribute the app to the end-users. The virtual machine performs the final transformation of translating the DEX bytecode into machine instruction during runtime. After all required files (in Fig. 1) are prepared, they are compressed into an APK file, which, in the last step, must be signed by the author for declaring the legitimacy of the application. The understanding of this process is essential since every code obfuscation approach works on a different level and affects different resources.

Note that the build process could be different if the app developer tended to build a cross-platform mobile application with Flutter[2] or React Native.[3] On Flutter, the source code is written in another language called Dart and compiled directly to a native library which will be referred in a small 'runner' Android project. With the React Native platform, the source code written in JavaScript will be kept in a bundle file (called *index.android.bundle* located in the assets folder by default) in the APK file and will be executed in a JavaScript VM (called JavaScriptCore) along with the application. Consequently, although the complied APK file is structured the same, the main logic of the application, which is relevant to the forensic investigators, can be found in different locations. Of course, this may also impact the obfuscation techniques applied. To the best of our knowledge, Proguard (see Sec. 5) is still the major obfuscation tool utilized for the applications that are built on these platforms.

3. Android application forensics basics

Due to the simplicity of unpacking, disassembling and repacking, it is a straight forward process for attackers to inject malicious code and advertisements to benign apps. Since "Android has no requirement for a key-pair to be certified by a Certificate Authority (CA)" (Vidas and Christin, 2013), a modified app can be signed with the attacker's key as a 'new' app and will run on Android device legitimately. Attackers may even sign the same repacked app with different keys to trick security products that rely on hash value of APK files for identification.

From a forensic investigator's perspective, unpacking and disassembling are playing an important role in reverse engineering Android apps (Lin et al., 2018; Spreitzenbarth et al., 2013) as they allow investigators to extract hard-coded evidence such as crypto-keys and user credentials (Zhang et al., 2017), as well as examining malicious activities of Malware (Cen et al., 2014). Prior to the prevalence of obfuscation techniques in Android development, reverse engineering (or statically analyzing) an Android application was significantly less complicated.

3.1. Unpacking and repacking

To unpack the APK content, the first step is to decompress the APK file which can be done by any software that handles ZIP files. Alternatively, one may use a more sophisticated tool such as the `apktool` which not only unpacks the APK but performs additional steps, e.g., decoding the manifest file (details see Sec. 5.1). Decompressed APKs may be modified and repacked, resulting in a valid new/modified app which in the last step must be re-signed which is commonly done using jarsigner (ORACLE, 2013a). Since this is commonly used for Android Malware, prior to disassembling the app under investigation, the examiner should identify if the app is an untampered app (see M1 in the Introduction) or a malicious/tampered one (see M2). This can be done in several ways:

1. The most straight forward approach is to generate the cryptographic hash of the app and look it up in a reference database. Alternatively, one may simply use a search engine.
2. Another approach is to analyze the certificate that comes with the app and is usually located in the META-INF folder and often has the file extension RSA. Once found, this binary file can be interpreted using Java keytool:

```
$ keytool -printcert -file META_INF/<NAME>.RSA
```

While these are frequently self-signed certificates, searching for the certificate fingerprint may reveal additional information.

3. Lastly, one may investigate the assets folder so see if one can find a familiar logo or other indicators of the original app.

Completing these previously mentioned steps then allows a general assessment of the app: (i) if all are true, the app is likely to be benign; (ii) if all are false, this is likely a malicious by design application; and (iii) if only 3 is true, this may be a benign application that has been tampered with. In the latter case, one may compare against the original APK to identify tampered files/resources. A possible procedure would be comparing the MANIFEST.MF files (name may be different) in the META-INF directory as these contain the SHA256-hashes as Base64 of all files. Note, that the file has to start with `Manifest-Version: 1.0`; there is a similar looking file with a.SF-extension which starts with `Signature-Version: 1.0` and contains the digital signatures of all files.

3.2. Disassembling

To investigate the functionality, the DEX file and shared libraries are disassembled. With respect to the DEX file, one can either decompile it back to Java source code or disassemble it to an intermediate language (e.g., `smali`):

[2] https://flutter.dev/(footnote links were last accessed 2021-09-09).
[3] https://reactnative.dev/.

Java Decompilation. To decompile the app, one can rely on *dex2jar* (Google, 2020b), a tool that transforms DEX bytecode into Java bytecode. After acquiring the Java bytecode, one can use a Java decompiler to retrieve Java source code. Examples would be *JD* (Dupuy, 2020), a Java decompiler that can extract the Java source code files or JEB, a commercial dissembler for DEX bytecode and native machine code.

Smali dis-/reassembling. Disassembling describes the procedure of converting the *classes.dex* file into an intermediate language named `smali` commonly done with the tool baksmali. Compared to DEX, `smali` is human readable. Another peculiarity of `smali` is that it can be recompiled back into a functioning DEX file (reassembling). Note, during dis- and reassembling, program components are likely to be reordered (i.e., arranged differently) even if no changes were made resulting in different hashes.

Library analysis. In addition to the Java executable, the shared libraries are usually written in C and compiled for different CPU architectures. Therefore, traditional Reverse Engineering tools for desktop computer such as `IDA pro` and `Ghidra` became the common options for forensic investigators.

Of course, if obfuscation techniques were successfully applied, this could severely hinder the process to a point where the original state of an Android app is not reconstructable anymore. How obfuscation can impact forensic analysis is discussed in Sec. 5.

4. Methodology

To find relevant source material, Google Scholar (GS) as well as other article search engines (e.g., IEEE, ACM) were searched to identify relevant literature. Additionally, general search engines such as google.com and duckduckgo.com were queried to include relevant knowledge in forums or blog posts.

General survey. To receive a general overview, the first query on GS with the search term *android obfuscation* was conducted. The results of the first five pages were then further segmented using a matrix dividing the content into three main categories: *obfuscation technique, obfuscation detection and deobfuscation*. Obfuscation technique is further partitioned into sub-categories of various obfuscation techniques (e.g., identifier renaming).

Extended searches. Based on these initial results, an extended search was performed with the aim to find additional literature. This search was primarily based on the identified sub-categories (e.g., android obfuscation reflection) as well as analyzing references of downloaded articles (backward searches). When searching online, again the first five result pages were considered. New sources were added to the matrix based on the headline/title. In case a result was ambiguous, we additionally evaluated the abstract, introduction and/or conclusion.

Finalizing the collection. As the last step, we read and assessed all identified articles. In case there was no relevance found, the entry was deleted. Additional sources like the general Android developer guide (developer.android.com) are not part of the matrix.

5. Obfuscation techniques

Obfuscation mechanisms have been widely implemented in the Android ecosystem where we, like Maiorca et al. (2015), "refer to the term obfuscation as actions that perform changes on the application while preserving its semantics" and usually complicates an analysis. Table 1 reflects the structure of this section and summarizes the most popular Android obfuscation techniques grouped by how they operate. The techniques require a different **implementation effort (IME)** which is an *subjective rating*. Meaning that approaches that are integrated into common tools and perform the transformation fully automated are labeled *low*. On the other hand, if manual work is required, it is rated *mid* or *high* depending on the complexity. For instance, asset file encryption can be completed with two helper functions (encrypt(), decrypt()) while reflection is considered more complex requiring more programming. Before discussing the techniques, we briefly want to highlight three frequent reasons we identified:

Table 1
Overview of obfuscation techniques.

Obfuscation technique	PTT	MBE	OPT	IME
APK				
Aligning		x	x	low
Manifest transformation		x	x	low
Renaming				
Identifier renaming	x	x		low
Package renaming	x	x		low
Native library stripping	x	x		low
Control flow modification				
Code shrinking (or tree-shaking)		x	x	low
Call indirection	x	x	x	low
Junk Code insertion	x	x		low
Code reordering	x	x		low
Reflection	x	x		high
Evasion attacks		x		high
Encryption				
Data encryption	x	x		low
Asset file encryption	x	x		mid
Class encryption	x	x		high
Native code encryption	x	x		high

Note that although Android obfuscation has been reviewed in existing works (e.g., Hammad et al. (2018); Dalla Preda and Maggi (2017)), this work intends to expand these reviews with new techniques and its impact on digital forensic analysis.

Protection's (PTT) goal is to safeguard the application (e.g., prevention from being modified into a malicious application) and/or the intellectual property (IP) by limiting the ability to manipulate or reverse engineer applications. Therefore, apps frequently rely on encryption (Sec. 5.4) but may also utilize other mechanisms as they are easy to implement.

Malicious Behavior Evasion (MBE) aims at bypassing malware analysis software. By altering specific information that is used to classify an app as benign or malicious, the decision can be influenced. Note, all presented obfuscation techniques are commonly used for MBE.

Optimization (OPT) is used to improve code because the output of the transformation can be stored more compactly, or the efficiency of the app can be improved. Concepts described in Sec. 5.1 or 5.2 are commonly used.

We believe that knowing the IME as well as a reason, helps an investigator to be better prepared. For instance:

- The app under investigation seems to be only repacked which makes it very unlikely that this is an original app; it is likely to be an app trying to hide malicious behavior.
- The app under investigation utilizes native code encryption (Sec. 5.4) and reflection (Sec. 5.3); it looks like it was not repacked (=original app). It is less likely to contain malicious behavior but reverse engineering will be difficult.

5.1. APK

Both presented techniques are directly related to the APK file: Aligning is commonly used for optimization purposes; the AndroidManifest.xml is often modified to hide malicious behavior. These techniques could mislead the forensic analysis where no code analysis is involved but relying on the target APK file's hash value or the requested App permissions.

Aligning. According to Google (2020d), "archive alignment [...] provides important optimization to Android application (APK) files [..] It causes all uncompressed data within the APK, such as images or raw files, to be aligned on 4-byte boundaries" allowing faster access and less RAM usage. While malicious apps may care less about the performance, it is an easy way to obfuscate as it can be done automatically using zipalign:

```
$ zipalign -v 4 infile.apk outfile.apk
```

where 4 is the alignment and -v forces verbose output. The same tool can also be used to verify the alignment running:

```
$ zipalign -c -v 4 existing.apk
```

where -c confirms the alignment of the given app.

AndroidManifest.xml transformation. The AndroidManifest contains metadata about the app itself and is saved in XML format. During the APK generation process, the manifest file is converted into a binary XML file which has to be undone before modifying it:

```
$ file Folder/AndroidManifest.xml
AndroidManifest.xml: Android binary XML
$ Java -jar apktool_2.5.0.jar d existing.apk
```

While `apktool` may be the most prominent tool, one could also use aapt[4] (Android Asset Packaging Tool) or online services such as apkdecompilers.com. The AndroidManifest is commonly used by tools to assess an app and thus simple alterations like adding unnecessary permissions or adding fake component capabilities may drastically decrease the performance of detection tools.

5.2. Renaming

These obfuscation techniques manipulate the program code directly by replacing expressions with less meaningful and often shorter expressions (e.g., only single characters), resulting in more compact source code, without modifying the logic. It may also modify non-code files such as assets. This evades searching for strings and analyzing file names.

Identifier renaming[5]. Following software development guidelines, variables, classes and methods should be meaningful to improve code readability and reusability. However, from a programming standpoint, the identifier names do not influence the program's logic. Thus, it is possible to replace every code identifier with a meaningless string (Dong et al., 2018). An example is given in Listing 1 and 2 representing identical logic.

Listing 1: Original Java code.

```
private void calculate(MyList listVariable) {
  while(listVariable.HasMore()) {
    variable = listVariable.GetNext(true);
    variable.calculate();
    StaticClass(variable);
  }
}
```

Listing 2: Obfuscated Java code.

```
private void a(a b) {
  while(b.a()) {
    a = b.a(true);
    a.a();
    a(a);
  }
}
```

`ProGuard`, for example, implements a simple version of this obfuscation technique, where the identifiers are replaced using the English alphabet characters [a-zA-Z]. DexGuard, a more powerful tool than ProGuard, on the other hand is using non-ASCII characters which makes it more difficult to work with (Apvrille and Nigam, 2014). Since automatically detecting these kinds of obfuscation is straight forward (average length of identifiers is short; analyzing used charset), one may instead replace each character of a given identifier with a random one, e.g., `Variable.GetNext()` turns into `Bszyoadf.ZbtNqvp()` or using MD5 values of original names (Hammad et al., 2018). In general, possibilities are endless and easy to implement.

Package renaming. Every Java program is structured into packages which are used to group related classes together building a common namespace; classes inside the same package can access each other without using the fully qualified name. Packages have a hierarchy, and the dot-character separates them. While Android strongly recommends using the package naming according to owned URLS in reverse order (e.g., Google Maps app should be `com.google.maps`), developers are free to choose the name of the package themselves, with the only requirement that all characters are lower case (ORACLE, 2013d). Wang and Rountev (2017) in their experiments used a tool called Legu. Legu[6] obfuscated the packet structure by renaming items and adding new helper classes to hide the original classes.

Native library stripping. Similar to identifier/package renaming for Java executable(s), a developer may strip/remove debug symbols that are not required for execution from a native (shared) library. If a library is compiled with Native Development Kit (NDK), stripping is done by default. Such a technique could significantly hurt the readability of the reverse-engineered code and impact automated tools that rely on Dalvik bytecode analysis. Due to the popularity of this technique, Android Malware and obfuscation tools tend to hide their malicious activity or cryptographic algorithms (for releasing the encrypted code at runtime) in stripped native libraries (Alam et al., 2017).

Forensic implication. Dong et al. (2018) investigated the Google Play Store and other Third-party markets using a custom detection model. They concluded that 43% of the Google Play Store apps and 73% of the third-party apps used some renaming obfuscation. Wermke et al. (2018) reported even higher numbers with 64% of the apps using renaming obfuscation. They note, however, that "a large percentage of apps were not intentionally obfuscated by the original developer but contained third-party libraries that used obfuscation". These renaming mechanisms have become a default choice for app/IP optimization protection Such techniques, which forensic investigators should familiarize themselves with, can defeat some existing library detection methods that rely on the names/symbols in APKs during an investigation. For example, Crussell et al. (2012, 2013)'s library detection methods were revised for similarity matching based on constructing Program Dependency Graphs, in which names and symbols in DEX bytecode are required. Although renaming mechanisms can pose an obstacle to static code analysis, many advanced approaches (introduced in Sec. 7) can bypass them because they do not intend to change the syntax or workflow of the program and they are usually limited, e.g., system calls and library calls may cause running issues.

5.3. Control flow modification

This group of obfuscation techniques alters program code in a way to control the execution path of the Android app. These can usually happen entirely automated.

[4] https://www.jgwebdesigns.net/android-development-tips-and-tricks/extracting-manifest-info-from-an-apk-file.

[5] In computer programming, identifiers are language entities such as class names, method names and field names.

[6] According to the authors this is a tool from China and we could not find a source.

Code shrinking (tree shaking). Tools such as `ProGuard` and `R8`[7] (which has replaced `ProGuard` to be the default option since Android Studio 3.3 beta) can also apply code shrinking and tree shaking to remove unreachable code and uninstantiated types. Although the aim of these tools is not necessarily for obfuscation purposes, the optimized code could cause confusion for some automated forensic tools. For example, as a unique feature of `R8`, class inlining can remove classes that are only used locally. Since this approach could inline (move and remove) code in the dex executable, tools/approaches relying on matching the application's Control Flow Graph (CFG) for third-party library detection or Malware detection could be potentially impacted by this code shrinking mechanism. Unfortunately, as `R8` is relatively new, we had barely found any existing literature that has claimed that their detecting approach tolerates the `R8` optimization.

Call Indirection. A call graph, also known as CFG, can be extracted from a computer program using either a dynamic approach, by recording the execution of the app with a profiler for example, or by a static approach, where the code itself is analyzed. The graph represents the calling relationship between the methods within the program code and can aid humans in understanding the program code (Eisenbarth et al., 2001). Anti-malware products utilize CFGs to determine if an Android app is malicious or not, by comparing the signature of the CFG to the CFG of known malware. Bacci et al. (2018a) designed a program that added a new method for every method invocation where the new method simply invokes the original one. Thus, creating an additional call of a method but leaving the original flow of the program intact. Dalla Preda and Maggi (2017) described a similar approach but defined more precisely that the new proxy methods have to be public and static, have the same parameters and have the same return type as the original method. Additionally, in their approach, they created distinct new proxy methods for methods that were called multiple times. This technique is usually executed on the DEX bytecode because an efficient Java compiler would detect such unnecessary calls and try to optimize the CFG accordingly and eventually get rid of the desired effect.

Junk Code Insertion. According to Li et al. (2019), Junk Code Insertion includes methods like operation (nop) instructions (e.g., `0x00` for DEX bytecode), unconditional jumps and additional register for garbage operations. These techniques do not change the program's logic and have a less significant impact on a detector. On the other hand, Sindhu et al. (2014) concluded that nop instructions are easily detectable and can be undone. In addition, the use of opaque predicate is described by Balachandran et al. (2016). An opaque predicate is a conditional instruction that always results in the same outcome but creates two branches. One of the branches leads to the original code, and the other contains unreachable junk instructions.

Code Reordering. During the compilation process (see Sec. 2) the DEX compiler orders the code in a structure that the Dalvik virtual machine can execute it. Bacci et al. (2018b) reported a random reordering method by inserting `goto` instructions, producing unnecessary jumps inside the code. This obfuscation technique has to be done without changing the original runtime execution. However, this could only be done to methods that did not contain any other type of jump instructions, like for example if, switch, while, and more. If the method is applied randomly, You and Yim (2010) considers this obfuscation technique as a strong way to circumvent anti-malware products that check the code signature, since the reordering is done randomly. Dalla Preda and Maggi (2017) described a different approach. Nevertheless, in this case, they grouped the DEX code into uniquely labeled basic blocks. Afterwards, these blocks were randomly shuffled. To preserve the original execution sequence, unconditional jumps were inserted, between the block, to link these together again.

Reflection. Reflection is a Java-specific feature used to examine or modify the runtime behavior of an application. In particular, it uses the `Java.lang.reflect.*` API (ORACLE, 2013c) which allows creating new and modifying existing classes during code execution. According to Li et al. (2016), "reflection usage is widespread in Android apps, with 87.6% (438/500) of apps making reflective calls". The most popular pattern in which Reflection is used is the sequence: `Class.forName()` followed by `Class.getMethod()` into `Method.invoke()`. Therefore, in most cases, reflection is used to access class methods that may be loaded at runtime. Geinimi, a popular malware family, uses reflection for dynamic method invocation (Kazdagli et al., 2014). Load parts of the code dynamically hamper static analysis and often requires dynamic approaches.

Evasion attacks. Evasion attacks are adversarial examples for machine-learning-based techniques. Since many Android malware detection and obfuscation detection techniques (introduced in Sec. 6) rely on Machine Learning, Chen et al. (2019) proposed an adversarial example to defeat two well-known machine-learning-based Android malware detection systems, MaMaDroid and Drebin, by obfuscating the semantic features (CFG is one of them) leveraged in these tools. The result shows that the presented adversary can drop the detection rate of an Android malware from 96% (MaMaDroid) and 97% (Drebin) down to 0%. Rather than targeting to specific Android malware detectors, Gu et al. (2020) presented adversarial examples, which are based on a modified one-pixel attack, to evade any Android Malware classifier that relies on Malware's gray-scale image. The result shows that with the proposed approach, 93% of 500 Malware examples were successfully classified as benign apps.

5.4. Encryption

The most sophisticated obfuscation mechanism is encryption as without the matching decryption method and the key, static analysis is useless. Investigators have to fall back on dynamic methods to get a full understanding of the app. Depending on the utilized technique, it may still be possible to reverse parts of the app. Generally, we differentiate between 'data encryption' and 'asset file encryption'. The use of encryption methods as an obfuscation method has increased over the evolution of the Android platform. For example, the support for file-based encryption in the Android Version 7.0 (Nougat) (Crowley and Lawrence, 2016), made the handling of obfuscation techniques like aligning with the additional use of encryption easier to implement since the encryption methods were part of the standard Android API. Therefore, no extra library had to be included, which made the app smaller and less suspicious to malware detectors that consider external libraries as problematic. One of the downsides of encryption is that it uses a large amount of computing power to encrypt and decrypt the data. However, modern hardware capabilities make the use of encryption more feasible.

Data Encryption. Naturally, examiners and analysis tools analyze cleartext strings looking for certain key words such as IP addresses or login credentials which are directly embedded into the DEX file (e.g., *stringsclasses.dex*). To hamper this analysis, Bacci et al. (2018a,b) implemented a simple Caesar cipher with a fixed key of 3; "the original string will be restored during application run-time." Of course, one may use stronger encryption mechanisms which in return will impact runtime.

[7] https://r8.googlesource.com/r8/.

Asset file Encryption. Asset files are not compiled but simply added to the APK file without modification. Since these files can be valuable for examiners, it became common to also encrypt them. For instance, Dalla Preda and Maggi (2017) renames the file and path names to the MD5 of their string values and then encrypts the raw objects. The authors point out that the `AssetManager` class, which is responsible for loading these asset files, cannot be overridden. Therefore, to decrypt them during runtime again, every call to the `AssetManager` class is redirected to a proxy method, handling the decryption.

Class Encryption. This obfuscation technique is based on the reflection technique. A possible implementation is discussed by Maiorca et al. (2015), to completely encrypt each program class and store the result in a byte array. The code has to be altered by adding a new method responsible for decrypting the data at runtime and loading the class into memory by using the reflection API. Accordingly, it inherits the disadvantage of using reflection by increasing the overhead of running the app.

Native Code Encryption. The Java Native Interface (JNI) is a Java feature that allows executing and being executed by native applications (i.e., closer to the hardware such as C or Assembly) (ORACLE, 2013b). This is especially useful if the standard Java class library does not support platform-specific features. To bring this to the next level, Dong et al. (2018) describes a more sophisticated approach where the original APK is encrypted and an additional wrapper APK is created that is responsible for decrypting the original APK, loading it into memory and then executing it. This approach makes it impossible for tools depending on static analysis to determine if the original app is benign or malicious. Additionally, it hinders reverse engineering as well. Schulz (2012) outlined a three-step process to modify DEX bytecode during runtime: (1) using JNI to run native code; (2) a magic byte constant that the native code will look for and (3) the new Davlik bytecode which will replace the original one. As pointed out by Wong and Lie (2018), "native code contains no symbol information—variables are mapped to registers and many symbols are just addresses. Thus, static analysis of native code yields significantly less useful results and the inclusion of native code in an application can hide malicious activity or sensitive API invocations from an analyzer."

6. Obfuscation detection

Obfuscated apps pose a real challenge and hamper the investigative process as outlined in Sec. 3. A starting point for examiners is the detection of the utilized mechanism and therefore researchers and practitioners have developed concepts in the form of schemes and implementations to tackle this challenge. In the following, we highlight some approaches that resulted in tools and are not only conceptional. An overview of identified detectors is shown in Table 2. More details about the different tools is provided in the corresponding paragraphs.

RepDroid. Yue et al. (2017) specifically address repacking and proposed a dynamic detection method based on layout group graph (LGG). LGGs can be extracted by executing the Android app and tracing the layout transitions during the interactions. Under the assumption that a repacked app behaves similar to the original version, the two LGGs are expected to be similar. Thus, one of two analyzed apps is a clone, when the LGGs resemble each other. This approach has excellent scaling potential since it can detect multiple clones of an app. However, the limitation of this work is that once apps are considered similar, the actual original benign app still cannot be determined. Furthermore, we could not find the implementation of RepDroid publicly, although the authors claimed that an implementation exists.

WLTDroid. A recent work continued the research on repacking and proposed a more efficient tool: `WLTDroid`. In contrast to RepDroid, this implementation uses a new approach called whole layout tree (WLT), which is a purpose-based measure. Since it is a dynamic technique, the app is executed. WLTDroid then extracts the external view layout information and the inner view transfer information to determine the similarity of apps by comparing the WLT similarity. This approach reaches a comparable detection rate, but the generation of a WLT is less time consuming compared to the LGG approach of RepDroid. Obviously, WLTDroid inherits the same limitations in case of identifying the original app. Additionally, no publicly available implementation of WLTDroid could be found (Guo et al., 2020).

Table 2
Obfuscation detection tools.

Tool	Obfuscation detection	Technique	Available
RepDroid	Repacking	dynamic	x
WLTDroid	Repacking	dynamic	x
IREA	Identifier Renaming Reflection Data Encryption	static	x
AndrODet	Identifier Renaming Data Encryption Control Flow Obfuscation	static	✓

IREA. Kühnel et al. (2015) developed `IREA` which is capable of detecting identifier renaming, reflection and data encryption.[8] The tool applies multiple heuristics to detect each obfuscation technique separately which are briefly summarized: 1) For identifier renaming, four different heuristics are used targeting various characteristics renamed identifier such as average length, variance of length, absolute number of used different characters and ratio of capital to lower case letters (CamelCase detection). 2) For reflection, the authors use pattern matching to check the presence of `java.lang.Class` or `java.lang.reflect` packages and opcodes of the form `invoke-kind`. 3) For encryption, IREA proposes two heuristics. One targets detecting the standard encryption API or common third party crypto libraries such as `java/security`, `javax/crypto` or `Lorg/jasyp`. The second heuristic targets custom encryption "by utilizing bit or byte operations on bit or byte arrays".

AndrODet. `AndrODet` by Mirzaei et al. (2019) is based on three detector algorithms working independently and with different feature sets. Thus, a single sample app can be tested on one obfuscation method (e.g., Identifier Renaming) or on multiple obfuscation techniques (e.g., identifier renaming and data encryption), by running multiple detectors in sequence. The detectors work on features that are extracted from DEX code. For identifier renaming, they used similar heuristics like IREA. In contrast, for encryption detection, they rely on analyzing the encrypted strings and extract features that include, entropy, word size in byte, string length and the number of special characters (e.g., equals, dashes, slashes). The control flow obfuscation detector works by analyzing the extracted control flow graph. Next, features like number of nodes, number of sinks, number of edges, for example, were considered to determine if the code used any control flow obfuscation techniques. The Python-implementation is publicly available on Github. Another practical advantage of AndrODet is that it supports online learning algorithms through Data Stream Mining (DSM), which means that AndrODet does not have to be re-trained. However, Mohammadinodooshan et al. (2019, ArXiv article) recently commented on encryption detection capabilities of AndrODet. The authors criticized that "the accuracy of string

[8] Additionally, it includes the detection of mobile advertising but is not relevant for this paper.

Table 3
Approaches and tools for deobfuscation.

Article	Tool	Adversary	Scope	Functionality	Application	Approach	Availability
Bichsel et al. (2016)	DeGuard	Identifier & Package renaming	Dalvik bytecode	Code deobfuscation	(Static) Code analysis	Probabilistic learning based on dependency graph	Free service at http://apk-deguard.com/
Yoo et al. (2016)	N/A	String obfuscation & encryption	Dalvik bytecode	Code deobfuscation	(Static) Malware analysis	Recover the encrypted string with the decrypted value captured at runtime	N/A
He et al. (2018)	DEBIN	Native library stripping	Native code (x86, x64, ARM)	Debug information recovery	(Static) Malware analysis	Machine learning based on Dependency Graph	Open-sourced at https://github.com/eth-sri/debin
Rasthofer et al. (2016)	HARVESTER	Reflection, data encryption	Dalvik bytecode	Runtime value extraction	Runtime data monitoring, dynamic analysis	Similarity matching	Base version for scientific research and advanced version for commercial usage at https://www.sit.fraunhofer.de/de/harvester/
Backes et al. (2016)	LibScout	Identifier renaming, reflection, control-flow modification	Dalvik bytecode	Third-party library detection	Vulnerable library detection	Profile matching based on known libraries	Open-sourced at https://github.com/reddr/LibScout
Wang et al. (2018)	Orlis	Control flow modification	Dalvik bytecode	Third-party library detection	Large scale library detection	Similarity digest matching based on call graphs	N/A
Baumann et al. (2017)	Anti-ProGuard	Identifier & Package renaming	Dalvik bytecode	Package detection	Malware detection	Similarity matching	Open-sourced at https://github.com/ohaz/antiproguard
Ikram et al. (2019)	DaDiDroid	Identifier Renaming, String Encryption, Control flow modification	Dalvik bytecode	Malware detection		Machine learning based on API call graphs	N/A
Li et al. (2019)	Obfusifier	Identifier Renaming, String Encryption, Identifier Renaming, Control flow modification	Dalvik bytecode	Malware detection		Machine learning based on Method graph, API Path, etc.	N/A

encryption detection is evaluated using samples from the AMD and PRAGuard malware dataset [.. which] are highly similar due to the fact that they come from the same malware family." Additionally, they proposed an alternative by analyzing per-string bases and defining a threshold to determine if an app is using data encryption.

As Table 2 showed, the number of publicly available tools is limited. From a forensic standpoint, this becomes a problem since an investigator does not have access to the tools themselves and conclusively cannot inspect and most critically cannot make improvements to the code. Researchers in the obfuscation community may have to consider publishing not only the research paper and the results but also the code used publicly, for example, on Github.

7. Deobfuscation

With the rise in popularity of obfuscation, there has also been an increase in research for defeating the obfuscation techniques. This section focuses on reviewing the deobfuscation techniques proposed for assisting forensic investigators. Additionally, we included techniques that are obfuscation resilient, which could potentially help forensic investigator to identify obfuscated third-party libraries (Sec. 7.3) and detecting obfuscated Malware (Sec. 7.4). An overview of all identified tools and articles is provided in Table 3.

7.1. Code deobfuscation

Identifier renaming is one of the most popular techniques found in the app market. Besides the automated deobfuscation approaches that we are about to introduce in this section, traditionally, forensic investigators could utilize some reverse engineering platforms (such as JEB[9] and IDA pro[10]) for renaming the obfuscated identifiers (in both Dalvik bytecode and native libraries) manually during static analysis. Such tools are still the most reliable option for fine-grained analysis (Zhang et al., 2017). `DeGuard`, presented by Bichsel et al. (2016), aims to reverse the Android applications obfuscated with ProGuard via a probabilistic learning model trained over thousands of non-obfuscated Android application. The model was stated as being able to defeat identifier renaming, detect third-party libraries, and is helpful for inspecting obfuscated malware. The basis of DeGuard is the "dependency graph" generated for the obfuscated app, which is formalized with the relationship between the un-obfuscated information left in the app (e.g., the unrenamed elements such as names of fields, methods, classes, and packages) and the obfuscated information that DeGuard tries to predict. The authors of the article claimed a 79.1% recovery rate for renamed elements, and 91.3% accuracy for third-party library detection. Although the implementation of DeGuard is close-sourced, DeGuard is showcased online on apk-deguard.com, allowing investigators to upload obfuscated APKs. Yoo et al. (2016) outlined a string deobfuscation scheme that targets the string code obfuscation technique specifically. It extracts the smali code of the Android app, intercepts all results coming from the decrypt method and replacing the encrypted string with the decrypted value. However, there is currently no implementation but maybe there will be in the future. DEBIN is a deobfuscation tool for predicting the stripped (debug) information from binaries (compiled for a specific CPU architecture) via machine learning (He et al., 2018). To build a machine learning model for name prediction, `DEBIN` first lifts the assembly code of non-striped binaries (which are decompiled from the binaries) and then converts the assembly code to an intermediate representation which is the basis for extracting dependency graphs (over which the model is built). Although this approach is not proposed solely for the Android platform, we argue that it could be very helpful for forensic investigators to apply static analysis on Android native libraries due to the fact that 1) comparing with similar tools such as DIVINE and TIE, DEBIN is the

[9] https://www.pnfsoftware.com/jeb/.
[10] https://www.hex-rays.com/products/ida/.

only tool that supports Advanced RISC Machines (ARM) executable,[11] and 2) the only tool that supports name prediction for functions and variables.

7.2. Runtime value extraction

Rasthofer et al. (2016) introduced HARVESTER and is a deobfuscation approach aiming to extract runtime values. When using HARVESTER, an analyst must specify the parameters (i.e., one or more variables or function arguments) that are targeted for extraction. HARVESTER then finds the context of the targeted parameters from the static code of the application and removes statements that may restrict the creation of them (i.e., resulting in a stripped source code). Lastly, this reduced application is executed to extract the runtime value dynamically. If the values of the targeted parameters are generated by a reflective call, HARVESTER will get the runtime values that are needed for making the reflective call first and then trace the targeted values by executing the reflective call as an ordinary call. In the article, HARVESTER outperforms other existing code (static/dynamic) analysis tools such as FlowDroid and TaintDroid.[12] Although HARVESTER helps defeating obfuscation techniques such as reflection and native code encryption, it does not provide any benefits for renaming or control flow modification. HARVESTER is available for scientific research in a base version[13] and in an advanced version for commercial usage.

7.3. Obfuscation-resistant third-party library detection

Detecting known third-party libraries that have been obfuscated can significantly reduce an investigator's workload or bypass the deobfuscation process completely if, for example, the third-party library being identified is open-sourced. One possible implementation is the light-weight third-party library detector LibScout which was introduced by Backes et al. (2016). The authors utilize a profile matching algorithm based on a hash (variant of Merkle) tree which is built upon the non-obfuscatable information extracted from an obfuscated library. LibScout has shown to be efficient for library detection (even fine-grained to its version) as well as resilient against common renaming techniques. A known flaw of this approach is that due to the way the hash tree is built it cannot be fully resilient against other techniques such as listed in Table 1 under “Control flow modification”. Another library detector named ORLIS was presented by Wang et al. (2018) and it is claimed to be resilient against control flow modifications. ORLIS can identify third-party library in an Android Application with a repository of known third-party libraries and a two-stage similarity matching approach. In the first stage, the sdhash ‘digest’[14] is calculated for the application and is compared with the same type of digests precalculated for the known third-party libraries. In the second stage, the comparison becomes fine-grained. Instead of App-to-libraries, ORLIS calculates and matches the ssdeep digest for each chain of classes of the application and the libraries (the libraries that are remained from the first stage). In the article, ORLIS was also proved outperforming some other detectors such as FDroidData and LibDetect. However, the tool has not been released publicly for forensic investigators. The second tool named Anti-ProGuard was presented by Baumann et al. (2017) and also aims to deobfuscate renaming techniques. However, instead of building a probabilistic model, Anti-ProGuard requires smali files as input and then utilizes a database storing obfuscated snippets and their cleartext counterparts. To improve the accuracy, Anti-ProGuard is not looking for exact matches but utilizes similarity hashing (SimHash by (Charikar, 2002)) Overall Anti-ProGuard was able to identify over 50% of packages in Android apps correctly, despite admitting of shortcoming like high processing time, e.g., one Android app took four to 6 h to process. However, the implementation code is not publicly available, which makes possible improvements for interested people harder.

7.4. Obfuscation-resistant malware detection

Since obfuscation heavily impacts the detection of malware (Pomilia (2016); Maiorca et al. (2015), some of the latest research focuses on utilizing features that cannot be (easily) obfuscated. These tools are often called obfuscation resilient or obfuscation resistant. Note, as the focus of this article is slightly different, we only showcase two recent approaches. For instance, Ikram et al. (2019) added additional features in their extraction process. For example, they parsed the names in the recovered source code and made the decision if more than 50% of the names were three letters or shorter, that the code has been processed with an Identifier renaming algorithm. They used this specific feature together with other& features to determine if an Android sample app was benign or not. By acknowledging the possibility of obfuscation and therefore adding an additional feature to represent it, the whole system itself gets more robust against obfuscation methods. Li et al. (2019) used a different approach by constructing a call graph based on the DEX file. By analyzing the graph, they extracted relevant features to determine malicious apps from benign apps. In this example, this approach is not disturbed by an Identifier renaming algorithm. Thus, the whole extraction process became more resilient against obfuscation approaches.

8. Related works

There have been various studies on obfuscation and its usage and impact on the Android ecosystem which can roughly separate into three different domains:

Usage of obfuscation techniques. Dong et al. (2018) studied the popularity of obfuscation techniques by looking at different app markets like Google Play Store, third-party markets (i.e., Huawei, Xiaomi, App China), and malware datasets from VirusShare and VirusTotal. In total, they examined more than 114 thousand apps and showed that, for example, identifier renaming was used more in third-party market apps and malware than in Google Play Apps. Similarly, Wermke et al. (2018) analyzed a dataset of over 1.7 million Google Play Apps and concluded that approximately 25% of the apps were intentionally obfuscated by the app developer (many more used libraries that were obfuscated). Additionally, they surveyed Android developers on their knowledge and use of obfuscation. Although tools like ProGuard are free and well known by developers, the paper pointed out that there are still misconceptions around obfuscation and that the correct use of obfuscation tools is a challenge.

Application of obfuscation techniques. The second group focused on applying obfuscation techniques for various purposes. Zheng et al. (2012) introduced ADAM, an automated and extensible system, with the idea to transform a (malware) sample into different variations using various obfuscation methods, including renaming obfuscation, control flow manipulations, and encryption. The goal of the authors was to test and “evaluate the detection of these

[11] ARM is one of the most commonly used CPU architectures for smart phones and the architecture, for which most native libraries of Android applications are complied.

[12] FlowDroid (Arzt et al., 2014) and TaintDroid (Enck et al., 2014) are well-known tools for static taint-analysis and dynamic taint-analysis for Android applications.

[13] https://www.sit.fraunhofer.de/de/harvester.

[14] ‘Digest’ is derived from many fuzzy call graphs, of which each is generated for a method.

variants against commercial anti-virus systems". Afterward, these new APK samples can be used to test and evaluate either custom detection tools or commercial systems. This allows tool developers to test their systems against various obfuscation techniques. Over the years new studies and results on testing Android analyzers were published (Rastogi et al., 2013; Maiorca et al., 2015; Hammad et al., 2018). After several studies, Dalla Preda and Maggi (2017) published a systematization of knowledge paper including a unified methodology.[15] Additionally, the authors expanded the idea of ADAM and designed a new framework called AAMO (Automatic Android Malware Obfuscator), with the main improvement that AAMO can combine multiple obfuscation techniques.

Detection of obfuscation techniques. Faruki et al. (2016) present a comprehensive analysis of obfuscation techniques as well as obfuscation detection techniques. In detail, they "review code obfuscation and code protection practices, and evaluate efficacy of existing code de-obfuscation tools". Accordingly, the focus of that work covered primarily tools used in manual reverse engineering and less on automated deobfuscation tools or frameworks.

All of the aforementioned articles include sections on obfuscation-related techniques. However, none of them provides an all-round view for obfuscation, obfuscation detection and deobfuscation techniques. In fact, the obfuscation techniques reviewed in these articles are limited to their own purpose. For example, when review obfuscation techniques, Wermke et al. (2018) does not "look for packers or other techniques specifically used by malware" (e.g., methods in the Control flow modification category in Table 1), and Dong et al. (2018) only counts those applications obfuscated with Identifier Renaming, String Encryption, or Java Reflection (which are attached to the Renaming and Encryption categories), because these obfuscation methods can be detected by the tool/platform proposed in these articles. For the same reason, reading an article written for testing malware detectors against code (Zheng et al., 2012) or for reviewing deobfuscation tools (Faruki et al., 2016) may be not efficient for a digital forensic investigator or Android app developer to understand this overall field.

9. Impact of obfuscation on forensic investigations

We see two major implications for forensic investigations:

Misleading automation Obfuscation techniques can be/are used to evade automatic detection and inhibit forensic examination. For instance, evasion attacks trick machine-learning-based obfuscation detection techniques, and simple methods such as repacking and manifest transformation can invalidate application fingerprinting (via hash values) techniques.

Complicate the manual analysis In addition to rendering automatic approaches useless, some obfuscation techniques can also significantly complicate the manual analysis of apps to a point where some methods are impractical, e.g., static analysis will not work for applications whose code is encrypted or is inserted with a great load of junk code.

Both points can be summarized to the point that the analysis (particularly, those that tend to be applied automatically) gets more difficult which in return requires more advanced practitioners and more backlogs. To counteract existing Android obfuscation techniques, several tools have been discussed that help to detect obfuscation and sometimes even to deobfuscate (parts of) the application. Approaches that deobfuscated and compare obfuscated code show promising results and have the advantage that they could also handle layered obfuscation techniques on Dalvik bytecode. However, according to our review, most of the automated approaches are not publicly released which coincides with findings from Wu et al. (2020) that tools are often developed but not released. In addition, the existing deobfuscation and obfuscation detection approaches reviewed barely cover native libraries although almost all the research we reviewed in this article (e.g., Mirzaei et al. (2019); Li et al. (2019)) have asserted that an obfuscated native library could hide malicious activities and hinder some library detection techniques. The results from more recent research are tools and techniques that are less impacted by obfuscation. For instance, Pasetto et al. (2020) presented a tool named `StrAndroid` which is "not affected by certain types of code obfuscation such as structural transformations (modifying the CFG of the methods) and dummy code insertion". Although obfuscation resilience features become essential for Android malware detection, it does not truly provide many benefits for reverse engineering (i.e., understanding) the application, as typical deobfuscation techniques for malware detection-resilient obfuscation work by extracting non-code features rather than recovering the source code of the application. An aspect that was not addressed in this article but is becoming more relevant, is the possibility of using multiple obfuscation methods in sequence, i.e., creating multiple layers of obfuscation. You and Yim (2010) already concluded that obfuscation techniques could be combined, resulting in a more complex obfuscation level, making malware analysis and deobfuscation approaches even harder. Recent work like Mirzaei et al. (2019) confirms this observation.

10. Conclusion

This work presented a condensed summary of the status quo of Android obfuscation, obfuscation detection and deobfuscation. The obfuscation techniques have been separated by their utilized technique (e.g., identifier renaming or encryption) which allows examiners to easily find them in this article and inform themselves. In contrast, obfuscation detection and deobfuscation have been organized by tools as we believe from a practitioner's perspective, finding sophisticated tools is most helpful. Consequently, this article provides a good overview for beginners and intermediates alike. Some of the key takeaways are that obfuscation techniques have advanced in recent years for various reasons and by now are frequently part of developer tools, i.e., they will become even more common in the future (obfuscation by default). On the other hand, not all techniques are easy to implement and some require more sophisticated knowledge which in return means they may be not used by 'script-kiddies'. Lastly, these advances often target and prevent static analysis wherefore dynamic analysis skills will be even more important in the future.

References

Alam, S., Qu, Z., Riley, R., Chen, Y., Rastogi, V., 2017. Droidnative: automating and optimizing detection of android native code malware variants. Comput. Secur. 65, 230–246.

Apvrille, A., Nigam, R., 2014. Obfuscation in android malware, and how to fight back. Virus Bull. 1–10.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P., 2014. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. ACM Sigplan Not. 49, 259–269.

Bacci, A., Bartoli, A., Martinelli, F., Medvet, E., Mercaldo, F., 2018a. Detection of obfuscation techniques in android applications. In: Proceedings of the 13th International Conference on Availability. Reliability and Security, pp. 1–9.

Bacci, A., Bartoli, A., Martinelli, F., Medvet, E., Mercaldo, F., Visaggio, C.A., 2018b. Impact of code obfuscation on android malware detection based on static and dynamic analysis. In: ICISSP, pp. 379–385.

Backes, M., Bugiel, S., Derr, E., 2016. Reliable third-party library detection in android and its security applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 356–367.

Balachandran, V., Tan, D.J., Thing, V.L., et al., 2016. Control flow obfuscation for

[15] This paper provides a good starting point for investigators to get even more insides into obfuscation.

android applications. Comput. Secur. 61, 72–93.
Baumann, R., Protsenko, M., Müller, T., 2017. Anti-proguard: towards automated deobfuscation of android apps. In: Proceedings of the 4th Workshop on Security in Highly Connected IT Systems, pp. 7–12.
Bichsel, B., Raychev, V., Tsankov, P., Vechev, M., 2016. Statistical deobfuscation of android applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 343–355.
Cen, L., Gates, C.S., Si, L., Li, N., 2014. A probabilistic discriminative model for android malware detection with decompiled source code. IEEE Trans. Dependable Secure Comput. 12, 400–412.
Charikar, M.S., 2002. Similarity estimation techniques from rounding algorithms. In: Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing, pp. 380–388.
Chau, M., Reith, R., 2019. Smartphone market share. URL: https://www.idc.com/promo/smartphone-market-share/os.
Chen, X., Li, C., Wang, D., Wen, S., Zhang, J., Nepal, S., Xiang, Y., Ren, K., 2019. Android hiv: a study of repackaging malware for evading machine-learning detection. IEEE Trans. Inf. Forensics Secur. 15, 987–1001.
Crowley, P., Lawrence, P., 2016. Pixel security: better, faster, stronger. URL: https://security.googleblog.com/2016/11/pixel-security-better-faster-stronger.html.
Crussell, J., Gibler, C., Chen, H., 2012. Attack of the clones: detecting cloned applications on android markets. In: European Symposium on Research in Computer Security. Springer, pp. 37–54.
Crussell, J., Gibler, C., Chen, H., 2013. Andarwin: scalable detection of semantically similar android applications. In: European Symposium on Research in Computer Security. Springer, pp. 182–199.
Dalla Preda, M., Maggi, F., 2017. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. Journal of Computer Virology and Hacking Techniques 13, 209–232.
Dong, S., Li, M., Diao, W., Liu, X., Liu, J., Li, Z., Xu, F., Chen, K., Wang, X., Zhang, K., 2018. Understanding android obfuscation techniques: a large-scale investigation in the wild. In: International Conference on Security and Privacy in Communication Systems. Springer, pp. 172–192.
Dupuy, E., 2020. Java decompiler. URL: https://java-decompiler.github.io/.
Eisenbarth, T., Koschke, R., Simon, D., 2001. Aiding program comprehension by static and dynamic feature analysis. ICSM 2001. In: Proceedings IEEE International Conference on Software Maintenance. IEEE, pp. 602–611.
Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N., 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. 32, 1–29.
Faruki, P., Fereidooni, H., Laxmi, V., Conti, M., Gaur, M., 2016. Android Code Protection via Obfuscation Techniques: Past, Present and Future Directions arXiv preprint arXiv:1611.10231.
Google, 2020a. App manifest overview. URL: https://developer.android.com/guide/topics/manifest/manifest-intro.
Google, 2020b. Github - dex2jar. URL: https://github.com/pxb1988/dex2jar.
Google, 2020c. Reduce your app size. URL: https://developer.android.com/topic/performance/reduce-apk-size.
Google, 2020d. zipalign. URL: https://developer.android.com/studio/command-line/zipalign.
Gu, S., Cheng, S., Zhang, W., 2020. From image to code: executable adversarial examples of android applications. In: Proceedings of the 2020 6th International Conference on Computing and Artificial Intelligence, pp. 261–268.
Guo, J., Liu, D., Zhao, R., Li, Z., 2020. Wltdroid: repackaging detection approach for android applications. In: International Conference on Web Information Systems and Applications. Springer, pp. 579–591.
Hamilton, F., 2020. Police Struggling to Clear Evidence Backlog of 12,000 Devices. URL: https://www.thetimes.co.uk/article/police-struggling-to-clear-evidence-backlog-of-12-000-devices-rpmhmfnpt.
Hammad, M., Garcia, J., Malek, S., 2018. A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In: Proceedings of the 40th International Conference on Software Engineering, pp. 421–431.
He, J., Ivanov, P., Tsankov, P., Raychev, V., Vechev, M., 2018. Debin: predicting debug information in stripped binaries. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1667–1680.
Ikram, M., Beaume, P., Kâafar, M.A., 2019. Dadidroid: an Obfuscation Resilient Tool for Detecting Android Malware via Weighted Directed Call Graph Modelling arXiv preprint arXiv:1905.09136.
Kazdagli, M., Huang, L., Reddi, V., Tiwari, M., 2014. Morpheus: benchmarking computational diversity in mobile malware. In: Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy, pp. 1–8.
Kühnel, M., Smieschek, M., Meyer, U., 2015. Fast identification of obfuscation and mobile advertising in mobile malware. In: 2015 IEEE Trustcom/BigDataSE/ISPA, vol. 1. IEEE, pp. 214–221.
Li, L., Bissyandé, T.F., Octeau, D., Klein, J., 2016. Droidra: taming reflection to support whole-program analysis of android apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 318–329.
Li, Z., Sun, J., Yan, Q., Srisa-an, W., Tsutano, Y., 2019. Obfusifier: obfuscation-resistant android malware detection system. In: International Conference on Security and Privacy in Communication Systems. Springer, pp. 214–234.
Lin, X., Chen, T., Zhu, T., Yang, K., Wei, F., 2018. Automated forensic analysis of mobile applications on android devices. Digit. Invest. 26, S59–S66.
Maiorca, D., Ariu, D., Corona, I., Aresu, M., Giacinto, G., 2015. Stealth attacks: an extended insight into the obfuscation effects on android malware. Comput. Secur. 51, 16–31.
Mirzaei, O., de Fuentes, J.M., Tapiador, J., Gonzalez-Manzano, L., 2019. Androdet: an adaptive android obfuscation detector. Future Generat. Comput. Syst. 90, 240–261.
Mohammadinodooshan, A., Kargén, U., Shahmehri, N., 2019. Comment on "Androdet: an Adaptive Android Obfuscation Detector" arXiv preprint arXiv:1910.06192.
ORACLE (2013a). jarsigner. URL: https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html.
ORACLE (2013b). Java native interface specification contents. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html.
ORACLE, 2013c. Trail: the reflection api. URL: https://docs.oracle.com/javase/tutorial/reflect/index.html.
ORACLE, 2013d. Using package members. URL: https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html.
Pasetto, M., Marastoni, N., Dalla Preda, M., 2020. Revealing similarities in android malware by dissecting their methods. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). IEEE, pp. 625–634.
Pomilia, M., 2016. A Study on Obfuscation Techniques for Android Malware. Sapienza University of Rome, p. 81.
Rasthofer, S., Arzt, S., Miltenberger, M., Bodden, E., 2016. Harvesting runtime values in android applications that feature anti-analysis techniques. In: NDSS.
Rastogi, V., Chen, Y., Jiang, X., 2013. Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, pp. 329–334.
Schulz, P., 2012. Code Protection in Android. Insititute of Computer Science. Rheinische Friedrich-Wilhelms-Universitgt Bonn, Germany, p. 110.
Sindhu, P., Babu, S.S., Vijayalakshmi, Y., Kalyankar, N., 2014. Evaluating android antimalware against transformation attacks. Journal Impact Factor 5, 9–14.
Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., Hoffmann, J., 2013. Mobile-sandbox: having a deeper look into android applications. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, pp. 1808–1815.
Vidas, T., Christin, N., 2013. Sweetening android lemon markets: measuring and combating malware in application marketplaces. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy, pp. 197–208.
Wang, Y., Rountev, A., 2017. Who changed you? obfuscator identification for android. In: *2017* IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft). IEEE, pp. 154–164.
Wang, Y., Wu, H., Zhang, H., Rountev, A., 2018. Orlis: obfuscation-resilient library detection for android. In: *2018* IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft). IEEE, pp. 13–23.
Wermke, D., Huaman, N., Acar, Y., Reaves, B., Traynor, P., Fahl, S., 2018. A large scale investigation of obfuscation use in google play. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 222–235.
Wong, M.Y., Lie, D., 2018. Tackling runtime-based obfuscation in android with <inline>\left\{\right.</inline>TIRO<inline>\left.\right\}</inline>. In: In *27th <inline>\left\{\right.</inline>USENIX<inline>\left.\right\}</inline> Security Symposium (<inline>\left\{\right.</inline>USENIX<inline>\left.\right\}</inline> Security 18)*, pp. 1247–1262.
Wu, T., Breitinger, F., O'Shaughnessy, S., 2020. Digital forensic tools: recent advances and enhancing the status quo. Forensic Sci. Int.: Digit. Invest. 34, 300999.
Yoo, W., Ji, M., Kang, M., Yi, J.H., 2016. String deobfuscation scheme based on dynamic code extraction for mobile malwares. IT Convergence Practice 4, 1–8.
You, I., Yim, K., 2010. Malware obfuscation techniques: a brief survey. In: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications. IEEE, pp. 297–300.
Yue, S., Feng, W., Ma, J., Jiang, Y., Tao, X., Xu, C., Lu, J., 2017. Repdroid: an automated tool for android application repackaging detection. In: *2017* IEEE/ACM 25th International Conference on Program Comprehension (ICPC). IEEE, pp. 132–142.
Zhang, X., Baggili, I., Breitinger, F., 2017. Breaking into the vault: privacy, security and forensic analysis of android vault applications. Comput. Secur. 70, 516–531.
Zheng, M., Lee, P.P., Lui, J.C., 2012. Adam: an automatic and extensible platform to stress test android anti-virus systems. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 82–101.
Zhou, Y., Jiang, X., 2012. Dissecting android malware: characterization and evolution. In: *2012* IEEE Symposium on Security and Privacy. IEEE, pp. 95–109.