A Large-Scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-Malware Products

Mahmoud Hammad, Joshua Garcia, and Sam Malek
Department of Informatics
University of California, Irvine
Irvine, California, USA
{hammadm, joshug4, malek}@uci.edu

ABSTRACT

The Android platform has been the dominant mobile platform in recent years resulting in millions of apps and security threats against those apps. Anti-malware products aim to protect smartphone users from these threats, especially from malicious apps. However, malware authors use code obfuscation on their apps to evade detection by anti-malware products. To assess the effects of code obfuscation on Android apps and anti-malware products, we have conducted a large-scale empirical study that evaluates the effectiveness of the top anti-malware products against various obfuscation tools and strategies. To that end, we have obfuscated 3,000 benign apps and 3,000 malicious apps and generated 73,362 obfuscated apps using 29 obfuscation strategies from 7 open-source, academic, and commercial obfuscation tools. The findings of our study indicate that (1) code obfuscation significantly impacts Android anti-malware products; (2) the majority of anti-malware products are severely impacted by even trivial obfuscations; (3) in general, combined obfuscation strategies do not successfully evade anti-malware products more than individual strategies; (4) the detection of anti-malware products depend not only on the applied obfuscation strategy but also on the leveraged obfuscation tool; (5) anti-malware products are slow to adopt signatures of malicious apps; and (6) code obfuscation often results in changes to an app's semantic behaviors.

ACM Reference Format:
Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A Large-Scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-Malware Products. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180228

1 INTRODUCTION

Android is the dominant mobile platform holding 85% of the smartphone OS market share [11]. At the same time, the number and sophistication of malicious Android apps are increasing. For instance, McAfee Labs crawled several app stores over six months in 2016 and detected more than 9 million malicious apps [10]. As another example, Kaspersky discovered more than 4 million new malware in 2016 [9].

Many reasons contribute to this meteoric rise of malware apps including: (1) the relative ease of creating a piggybacked app [35–38, 48], i.e., a mutated version of a legitimate app injected with either malicious code or embedded advertisements; and (2) the prevalence of alternative Android app stores (i.e., app stores other than the official Android app store, Google Play [19]), on which malicious apps may be distributed to users.

To protect mobile devices, users often rely on anti-malware products, which scan apps to determine if they are benign or malicious. However, many malware apps have previously evaded detection by these products. Examples of such malicious apps include Brain Test [7], VikingHorde [12], FalseGuide [18], and DressCode [8]. These apps have infected millions of users before they were detected. To evade detection, malware authors often rely on *code obfuscation.*

Code obfuscation transforms code into a form that is more difficult for humans, and possibly machines, to read, understand, and reverse engineer. These transformations change the syntax of code but not their semantics [30]. These changes could be small (e.g., inserting unused code) or sophisticated (e.g., performing bytecode encryption)[44]. Although code obfuscations are used by malware authors, they are also used by benign app developers to increase the difficulty of reverse engineering their apps.

To better protect the intellectual property of benign app developers and prevent cloning of their apps, several companies have developed obfuscation tools, or *obfuscators* for short, that implement different code transformations (e.g., identifier renaming, string encryption, reflection, etc.). Given the use of obfuscations by malware authors, the goal of this study is to assess the performance of commercial anti-malware products against various obfuscation tools and strategies.

Although some researchers have studied an individual obfuscation tool's effectiveness on a limited number of anti-malware products [32, 34, 40, 43, 44, 47], no study has performed a large-scale assessment of (1) the effect of individual and combined obfuscation strategies provided by multiple obfuscations tools on anti-malware products, (2) the effect the tools and strategies have on the accuracy of anti-malware products for benign apps and not just malicious apps, (3) the effect of time on obfuscated-app detection by those products, and (4) whether the application of obfuscation strategies result in valid, installable, and runnable apps. Due to the lack of a study regarding the effect of specific and combined obfuscation strategies on anti-malware products, it is unclear which strategies evade such products the most. None of the aforementioned studies determine the extent to which anti-malware products erroneously consider obfuscated, benign apps as malicious, which is undesirable for both anti-malware product vendors and benign app developers.

To determine if the transformations applied by obfuscation tools break an app's semantics, our study investigates the ability of obfuscation tools to generate valid, installable, and runnable apps. An obfuscated app is not useful to a benign app developer or malicious author if it cannot be executed on a device or if its benign, functional behavior changes. To ensure an app's obfuscation is successful, our study further compares the behavior of an obfuscated app with the behavior of its corresponding original app.

Overall, this paper makes the following contributions:

- We assess the accuracy of over 60 anti-malware products on apps obfuscated using 7 obfuscation tools and 29 obfuscation strategies on 3,000 benign apps and 3,000 malicious apps, totaling over 73,000 obfuscated apps. We further consider the effect of an app's age on that accuracy.
- We evaluate the ability of 7 obfuscation tools to generate Android apps that are valid, installable, and runnable.
- Based on our results, we make suggestions for improving anti-malware products and obfuscation tools.
- To conduct this study, we have implemented a framework for obfuscating Android apps and scanning them using anti-malware products. The framework is reusable, can be extended to include more obfuscation tools and strategies, and is available online [21], along with our resulting dataset of over 73,000 obfuscated apps.

The remainder of this paper is organized as follows. Section 2 covers background information about Android apps and code obfuscation. Section 3 discusses the research questions that this study aims to answer. The research methodology of this study is presented in Section 4. The results and the findings are reported in Section 5. Section 6 discusses the results and provides recommendations to enhance anti-malware products and obfuscation tools. The threats to validity are presented in Section 7. Finally, the paper overviews the related literature (Section 8) and concludes (Section 9).

2 BACKGROUND

This section provides a brief overview of Android apps and obfuscation strategies to help the reader understand the rest of the paper.

2.1 Android Apps

Each Android app is packaged and distributed as an Android Package Kit (APK) file, which is a zipped file that is mainly written in Java. Each APK file contains a manifest file, resources (e.g., images), and the app's bytecode. An app's code is compiled into *Dalvik EXecutable* (*DEX*) format, which can be executed on a customized Java Virtual Machine (JVM). There are two JVMs that can execute the DEX format: Android Runtime (ART), introduced in Android version 5 (Lollipop); and Dalvik Virtual Machine (DVM), for older versions. *classes.dex*, the main DEX file of an Android app, is a file in the APK generated by *dx*, a utility that converts *.class* files into a DEX file.

classes.dex can be disassembled by Baksmali [23] into an Intermediate Representation (IR) format which, in turn, can be assembled by Smali [23] to generate a new variant of *classes.dex*. The new *classes.dex* can be repackaged using a tool such as *Apktool* [1], a reverse-engineering tool for Android APK files, to generate a new APK variant (e.g., an obfuscated app). Different IR formats can be generated from *classes.dex*, including Smali code using Apktool and *.class* files using DARE [42] or dex2jar [16]. Moreover, Soot [45] can generate various IRs such as Baf, Jimple, Shimple, Grimp, or even a low-level IR such as Jasmin.

2.2 Obfuscation Strategies

To study the effectiveness of anti-malware products, we applied several different obfuscation strategies on each Android app. We use the term *obfuscation strategy* to refer to a single transformation or multiple transformations applied to an Android app. We consider three types of strategies: *trivial strategies*, *non-trivial strategies*, and *combined strategies*. Table 1 presents abbreviations of the trivial and non-trivial obfuscations, which will be used throughout this paper.

Table 1: Obfuscation-strategy abbreviations

Trivial Obfuscation		Non-trivial Obfuscation			
Disassembly/ Reassembly	DR	Junk code insertion	JUNK	Identifier renaming	IDR
AndroidManifest	MAN	Class renaming	CR	Control flow	CF
Alignment	ALIGN	Member reordering	MR	Reflection	REF
Repackaging	REPACK	String encryption	ENC		

Trivial obfuscation strategies are code transformations that do not change the app's bytecode. For this study, we examined the following trivial strategies:

- Repackaging (REPACK) involves unzipping the APK file and re-signing it with a different signing certificate. This simple transformation thwarts anti-malware products that rely on the app's certificate to determine if the app is malicious or not. For this transformation, we unzip an APK file using the *zipfile* Python library and resign it with our own signing certificate using *jarsigner* [20], a tool for verifying and generating digital signatures for JAR files.
- Disassembling and Reassembling (DR) involves disassembling the app using a reverse-engineering tool, such as Apktool, reassembling the app, and then signing it. By disassembling and reassembling the app, the items in *classes.dex* will be reordered. Anti-malware products that rely on matching *classes.dex* against signatures of known malicious apps would be broken.
- AndroidManifest transformation (MAN): Each Android app contains a configuration file called *AndroidManifest.xml* file, which specifies the principal components that constitute the application, including their types and capabilities, as well as required and enforced permissions. This transformation changes the manifest by adding permissions or adding components' capabilities, called *Intent Filters* in Android.
- Alignment (ALIGN) realigns all uncompressed data, such as images or raw files, in an APK file. This transformation changes the cryptographic hash of an APK file. Therefore, if an anti-malware product identifies malicious apps based on their cryptographic hashes (e.g., MD5), this transformation can thwart it.

Non-trivial obfuscation strategies are code transformations that change an app's bytecode. We study the following non-trivial obfuscation strategies:

- Junk code insertion (JUNK) adds code that does not affect the execution of an app. Junk code insertion can add null operations (nop), comments, and/or debugging information to a *classes.dex* file.
- String encryption (ENC) encrypts the strings in *classes.dex* and adds a function that decrypts the encrypted strings at runtime. Anti-malware products that rely on the string data in an app to determine if it is malicious will be evaded by this transformation.
- Control-flow manipulation (CF) changes the methods' control-flow graph by adding conditions and iterative constructs. In addition, this transformation changes the app's call graph by adding new methods and fake calls to the newly added methods.
- Members reordering (MR) changes the order of instance variables or methods in a *classes.dex* file, which evades anti-malware products that depend on the sequence of members in a class.
- Identifier Renaming (IDR) renames the instance variables and/or the method names in each Java class with randomly generated names. This transformation changes signatures generated from identifiers and changes the *method table* in Dalvik bytecode.

- Class renaming (CR) renames the classes and/or the packages in an app with randomly generated names. This transformation changes the *method table* in the Dalvik bytecode.
- Reflection (REF) transformations convert direct method invocations into reflective calls using the Java reflection API, which can evade static analyses that rely on direct method calls.

Combined strategies are combinations of the aforementioned obfuscation strategies. Previous work [40, 44] has mentioned that combining obfuscation strategies result in stronger obfuscations. Our study leverages different combined strategies to understand which combinations of transformations will result in better evasion of anti-malware products. The majority of our leveraged combined strategies have not been empirically studied in previous work.

3 RESEARCH QUESTIONS

In this paper, our primary goal is to provide a large-scale empirical study that evaluates the effectiveness of anti-malware products against various obfuscation tools and strategies. To that end, this section presents and discusses the research questions this study attempts to answer. Moreover, this section shows who will benefit from answering each research question. In the remainder of this section, we introduce and motivate each research question that we study.

RQ1. How is the accuracy of anti-malware products affected by obfuscation strategies?

The use of code obfuscation in Android apps has become popular and is leveraged by both benign and malicious app developers. Given that smartphone users rely on anti-malware products to protect their devices, it is crucial for anti-malware products to distinguish malicious apps from benign ones with high accuracy, while being resilient to obfuscation. RQ1 aims to measure the accuracy of commercial anti-malware products against a broad range of obfuscation strategies. We measure accuracy in this paper using precision and recall, since these metrics take into account false positives (i.e., benign apps marked as malicious) and false negatives (i.e., malicious apps marked as benign).

Anti-malware providers will benefit from answers to RQ1 in order to improve their products, especially against the obfuscation strategies that thwart their products the most. In addition, the answers to RQ1 can help smartphone users choose between anti-malware products. Benign app developers will benefit from answers to RQ1 by learning which obfuscation strategies prevent their apps from being flagged as malicious.

RQ2. How is the accuracy of anti-malware products affected by obfuscation tools?

Each anti-malware product's effectiveness likely varies based on the implementations of obfuscation strategies provided by an obfuscation tool. To make that determination, RQ2 measures the accuracy of anti-malware products on obfuscation tools, where accuracy is again measured in terms of precision and recall.

Anti-malware product vendors, benign app developers, and obfuscation tool developers can benefit in several ways from the answers to RQ2. Anti-malware product vendors can use this information to determine which specific implementations of obfuscation strategies may cause false positives (i.e., benign apps marked as malicious) in their products. Similarly, these vendors can benefit from learning which obfuscations result in successful evasion from detection by malicious apps. Answers to RQ2 can aid benign app developers in choosing the obfuscation tools that will prevent their apps from erroneously being marked as malicious. Furthermore, if false positives or false negatives (i.e., malicious apps marked as benign) are due to obfuscation tools, as opposed to the anti-malware products, then this information is useful for correcting obfuscation tools.

RQ3. How is the accuracy of anti-malware products affected by the year an app is created?

RQ3 aims to study the accuracy of anti-malware products on non-transformed and transformed apps over different time periods, where each time period for our study spans two years. We consider transformed apps as belonging to the same time period as their non-transformed versions. For example, if we transform apps created in time period 2012-2013, we still consider the resulting obfuscated apps as created in 2012-2013, for the purposes of RQ3.

This research question allows us to understand the effectiveness of anti-malware products when applied to different time periods and to determine if those products' detection accuracies are affected by time. Anti-malware vendors can use this information to determine the time periods that result in poor accuracy for their products, aiding them with improving results for apps created during those problematic time periods.

RQ4. To what extent do obfuscation tools result in valid, installable, and runnable apps?

Although an obfuscated app does not need to be runnable when scanned by an anti-malware product, developers of benign apps and obfuscation tools rely on those tools to produce valid, installable, and runnable apps. Similar to [34], we consider an APK to be *valid* if an obfuscation tool successfully generates a signed APK package that includes a *classes.dex* file containing correct Dalvik bytecode syntax. An app is *installable* if it can be successfully deployed into the Android runtime. For our purposes, a transformed app is *runnable* if its runtime behavior is similar to its non-transformed version. RQ4 is particularly useful for obfuscation tool developers since answers to that question provide information about transformations that result in malformed apps.

4 RESEARCH METHODOLOGY

This section describes the research methodology that we pursued in terms of our study subjects, selected obfuscation tools, our evaluation framework, and our selected anti-malware products.

4.1 Study Subjects

We used a dataset of benign apps consisting of 3,000 apps from Google Play and 3,000 malicious apps. To avoid having malicious apps in our ground-truth dataset of benign apps, we obtained benign apps from AndroZoo [26], which is a collection of more than 5.5 million apps collected from several sources, including Google Play. AndroZoo apps have been scanned by commercial anti-malware products using the VirusTotal service [4], a free online service provided by Google that scans URLs, files, and Android apps. Approximately 25,000 Google Play apps out of nearly 2 million apps in AndroZoo are marked as benign by all anti-malware products. From these 25,000 apps, we have randomly selected 3,000 apps for this study. The malicious apps belong to several malware repositories including Android Malware Genome [49], Contagio [6], AndroTotal [39], the Drebin dataset [27] and VirusShare [5]. In addition to these malware repositories, we used the VirusTotal service to include recently discovered malicious apps that belong to the following malware families: BrainTest [7], VikingHorde [12], and FalseGuide [18].

4.2 Obfuscation Tools

We have included the following obfuscation tools for our study, whose supported obfuscation strategies are depicted in Table 2: