A Large Scale Investigation of Obfuscation Use in Google Play

Dominik Wermke
Leibniz University Hannover
wermke@sec.uni-hannover.de

Nicolas Huaman
Leibniz University Hannover
huaman@sec.uni-hannover.de

Yasemin Acar
Leibniz University Hannover
acar@sec.uni-hannover.de

Bradley Reaves
North Carolina State University
bgreaves@ncsu.edu

Patrick Traynor
University of Florida
traynor@cise.ufl.edu

Sascha Fahl
Ruhr-University Bochum
sascha.fahl@rub.de

ABSTRACT

Android applications are frequently plagiarized or repackaged, and software obfuscation is a recommended protection against these practices. However, there is very little data on the overall rates of app obfuscation, the techniques used, or factors that lead to developers to choose to obfuscate their apps. In this paper, we present the first comprehensive analysis of the use of and challenges to software obfuscation in Android applications. We analyzed 1.7 million free Android apps from Google Play to detect various obfuscation techniques, finding that only 24.92% of apps are obfuscated by the developer. To better understand this rate of obfuscation, we surveyed 308 Google Play developers about their experiences and attitudes about obfuscation. We found that while developers feel that apps in general are at risk of plagiarism, they do not fear theft of their own apps. Developers also report difficulties obfuscating their own apps. To better understand, we conducted a follow-up study where the vast majority of 70 participants failed to obfuscate a realistic sample app even while many mistakenly believed they had been successful. These findings have broad implications both for improving the security of Android apps and for all tools that aim to help developers write more secure software.

KEYWORDS

Obfuscation, Android, User Study

ACM Reference Format:

Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. 2018. A Large Scale Investigation of Obfuscation Use in Google Play. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3274694.3274726

1 INTRODUCTION

While smartphones have changed society in countless ways, application markets are perhaps an underappreciated development. These markets enable the simple distribution of new software, but they have also enabled numerous studies of application security [17–19] and provided mechanisms to identify malware before or after infection [9, 37]. Much of this research depends on software analysis techniques, and these techniques face challenges in the presence of *software obfuscation* [14, 26, 28, 34, 53], software transformations designed to frustrate automatic or manual analysis.

Despite the impacts of obfuscation, to-date there is very little *data* on how Android apps are obfuscated in practice apart from limited or small-scale studies [19, 36]. In this paper, we present the first holistic, comprehensive analysis of the state of the use of software obfuscation in Android applications. We begin with a study of obfuscation usage (and techniques) on over 1.7 million apps collected from Google Play. We follow this with a survey of 308 application developers about their experiences and perceptions of software obfuscation. We conclude with a development study with 70 professional Android developers to investigate usability issues with ProGuard, which is by a large margin the most popular obfuscation tool for Android. We address three research questions:

RQ1: *How many apps are obfuscated, and what techniques are used?* For researchers who develop automated analysis tools, it is critical to understand how often and what types of obfuscation are commonly applied so they can ensure correct analysis of apps. It is also an important measurement for the Android ecosystem. Software obfuscation is a defense against app repackaging, an abusive practice where applications are cloned and redistributed to build trojan apps or steal ad revenue. App repackaging is an epidemic threat to the entire ecosystem: in recent studies, 86% of malware samples collected were repackaged versions of benign applications [57], and apps are repackaged by the thousands [15, 51]. Up to 13% of entire third party markets consist of repackaged apps [55, 56]. Thus, software obfuscation protects not just individual apps and developers, but also users and the ecosystem at large.

We find that roughly 25% of apps are obfuscated, but that number rises to 50% for the most popular apps with more than 10 million downloads. This is high enough that it would have a significant impact on research – especially for projects that ignore obfuscated apps [41, 48]. However, it is also still low enough to indicate that the vast majority of apps are unprotected.

RQ2: *What are developers' awareness, threat models, experiences, and attitudes about obfuscation?* These factors provide insight into root causes of the low rates of obfuscation in Android. We examine whether developers are aware of obfuscation, whether they have attempted or successfully used obfuscation, which tools they have used, and whether they found the tools were sufficiently easy to use. We find that while developers are aware of the benefits of obfuscating their apps on a theoretical level, a perceived negligible personal impact and the time-consuming use of obfuscation tools for real applications is a large deterrent to obfuscation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274726

		Obfuscation							Other				
Name	License	Package name	Class name	Method name	Field name	Overloading	Debug Data	Annotations	String Enc.	Class Enc.	Optimization	Minimization	Watermarking
Allatori[1,†]	$290	●	●	●	●	●	●	○	●	○	●	●	●
DashO[†]	On request	●	●	●	●	●	●	●	●	○	●	○	●
DexGuard[2,†]	On request	●	●	●	●	●	●	●	●	●	●	●	○
DexProtector	$800	●	●	●	●	○	○	○	●	●	○	○	○
GuardIT	On request	●	●	●	●	●	●	○	●	●	○	○	○
Jack[2,†]	Free	●	●	●	●	○	●	●	○	○	●	●	○
ProGuard[†]	Free	●	●	●	●	●	●	●	○	○	●	●	○
ReDex[2,†]	Free	●	●	●	●	●	●	●	○	○	●	●	○
yGuard[†]	Free	●	●	●	●	●	●	○	○	○	○	●	○

[1] Multiple obfuscation patterns, default can be detected
[2] Mirrors ProGuard's obfuscation with same configuration format
[†] Obfuscation features (partially) detected by Obfuscan

Table 1: Selected features of popular obfuscation software for the Android environment.

RQ3: *How usable is the leading obfuscation tool ProGuard?* Our developer survey also found that 35% of our participants reported difficulty obfuscating their apps, while over 61% — more than double the Play market average — claim to obfuscate their apps. To better understand this paradox, we asked 70 developers to obfuscate two sample apps. We found that while most developers successfully managed to complete a simple obfuscation task, 78% failed to correctly use ProGuard in a more complex and realistic scenario. Moreover, 38% mistakenly believed they had successfully obfuscated their app. This highlights that even when developers attempt to use obfuscation, tool usability likely has a negative impact on its effectiveness.

We conclude our paper with a discussion of lessons learned and recommendations in Section 7. While software obfuscation is by no means a perfect defense against reverse engineering, previous work shows that even simple forms of obfuscation (like identifier renaming) significantly increase the effort required to successfully reverse engineer software [7, 8]. Additionally, the significant challenges obfuscation presents researchers (as shown in prior work [26, 28, 34, 53]) make this topic worthy of study. Our focus is on obfuscation used by legitimate applications; we leave the topic of obfuscation of malware for future work.

We note that the implications of this study go beyond the Android ecosystem. In contrast to other secure practices with a variety of costs and trade offs, software obfuscation is in an ideal position for adoption: ProGuard is one of the very few secure development tools in existence that is free, already available in the IDE of most developers, and can automatically enhance security while simultaneously improving performance. *Understanding why developers do or do not use such an ideal tool has broad implications both for the development of better developer support and as a measure of barriers to a more security-conscientious software development community.*

Source

```
public class Matrix {
  private int M;
  public Matrix(int M);
}
```

Obfuscated

```
public class a {
  private int a;
  public a(int b);
}
```

Listing 1: Example code before and after obfuscation with ProGuard.

2 ANDROID OBFUSCATION TECHNIQUES

Available obfuscation tools for the Android ecosystem range from free, open-source obfuscation solutions providing only basic obfuscation features such as ProGuard, up to premium tools with high licensing fees such as DexGuard (cf. Table 1). Basic obfuscation features include the following:

Name obfuscation. *Package*, *class*, *method*, and *field* names are commonly obfuscated by replacing their original values with meaningless labels. For example, ProGuard implements name obfuscation by generating name replacements using characters from the `[a-zA-Z]` alphabet. Obfuscated names are generated by iterating through the alphabet resulting in the following renaming patterns: `[a, b, ..., z]`, `[A, B, ..., Z]`, `[aa, ab, ..., zz]`, and so on. Allatori and DexGuard build on ProGuard's name obfuscation alphabet and add reserved Windows keywords ("AUX", "NUL"). Some of the tools allow users to add their own word lists to the renaming alphabet.
Name overloading. Obfuscation tools commonly use Java method overloading to assign the same name to methods with different signatures (i.e., different arguments or return types). In addition to using the same name for different methods, method parameters are also renamed to common names.
Debug data obfuscation. Removing debug information like line numbers or method names complicates the reverse engineering of code structures. Obfuscation tools often include means to reverse this information removal to allow for debugging by developers.
Annotation obfuscation. Another information removal feature strips annotations from classes and methods. Annotations provide additional functional context in class bytecode, including annotations for inner classes or methods that contain "throws" statements. Similar to debug information, the removal of class file annotation and the removal of class source file information complicates the reverse engineering of code structures by tracing class attributes.
String encryption. Strings can be encrypted to hide information. A trade-off has to be made between encryption strength and performance impact by decryption. The decrypter has to be provided in the program, making encryption unsuitable to hide sensitive information. Strings are encrypted to deter simple string searches over the code base and hide information about the program flow.
DEX file encryption. The *classes.dex* file can be encrypted to increase the difficulty of decompilation. Decryption of encrypted classes at run time can cause large performance impacts.

Complications for Obfuscation. While the previous section has discussed a number of techniques for transforming software, configuring obfuscation tools for Android is more complicated than merely choosing from the available features. In fact, there are a

```
-optimizationpasses 5

-dontusemixedcaseclassnames
-overloadaggressively
-printmapping mapping.txt

-keep public class * extends project.Interface
-dontwarn project.example.**
```

Listing 2: Example ProGuard configuration. Configuration path is set in the build system, e.g. in a *gradle.build* file.

number of complicating situations that make it difficult or impossible to obfuscate certain pieces of code, and if that code happens to be obfuscated the app can no longer function. These situations for partial obfuscation include classes that need to be accessible from an outside context: the names and class names of native methods and similarly classes that extend native Android classes such as activities, services or content providers should remain unobfuscated in most cases so that the library/system can invoke callbacks.

ProGuard. The free ProGuard enjoys preferential treatment in the Android ecosystem. It is included with the Android SDK and the official Android Studio IDE. In addition, other obfuscation tools inherit most of their functionality from ProGuard; the now deprecated alternative tool chain Jack is configured by ProGuard configuration files and provides ProGuard's obfuscation with reduced options. Similarly, ReDex accepts ProGuard's configuration files and mirrors the renaming functionality closely. DexGuard is a commercial ProGuard extension and utilizes name obfuscation with the same basic functionality as ProGuard, but with some advanced features.

ProGuard was integrated with the Android Software Development Kit (SDK) in August 2009 and can be activated in the build setup of a project. The "`minifyEnabled`" option activates ProGuard obfuscation for the release build of an app. Additional configuration files can be specified with the "`proguardFiles`" option. In the ProGuard configuration file, different program options are activated/deactivated by setting a number of flags that are relevant to later presented results (cf. Listing 2). Some processing steps of ProGuard can be completely disabled with flags such as "`-keep`".

3 DETECTING PROGUARD OBFUSCATION

To answer "**RQ1:** *How many apps are obfuscated, and what techniques are used?*" we built a tool we call Obfuscan to conduct a large scale measurement study of obfuscation practices. Obfuscan is able to detect a number of obfuscation features in compiled Android binaries. In particular, Obfuscan is able to detect all of ProGuard's obfuscation features and many features of other obfuscation tools (as shown in Table 1).

How Obfuscan Works. Obfuscan takes an Android binary as input and analyzes certain parts of the binary to detect specific obfuscation features and outputs the list of all detected features. Obfuscan analyzes package, class, method and field names to detect name obfuscation. To detect method name overloading, Obfuscan analyzes the distribution of obfuscated method names for duplicates and relies on the content of debug entries to detect debug information removal. Annotation removal is detected by analyzing an app binaries for the removal of corresponding class attribute fields. To detect further obfuscation features, Obfuscan relies on the *classes.dex* file format and specific function calls (see below).

Feature Detection. Obfuscan implements many heuristics to detect obfuscation features. For accuracy, many of these are developed deterministically and directly from the ProGuard source code.

For name obfuscation, Obfuscan detects both lower- and uppercase obfuscated names by simulating the obfuscation process of ProGuard and comparing the generated names to the actual names encountered on the app, package, or class level. Obfuscan also considers possible flags such as the usage of mixed-case characters if corresponding strings are detected in the scope. Finally, Obfuscan also looks for instances where tools replace class names with restricted keywords in the Windows operating system utilized by DexGuard and some Allatori configurations. To detect method name overloading, Obfuscan investigates names that follow the obfuscation pattern and occur multiple times on the same class level. Obfuscan detects missing debug information by parsing and storing the entries of the Java *LineNumberTable* which maps bytecode instructions to source code line numbers. Similarly, the removal of the source file data from classes removes information about the source file where the class (or at least its majority) is defined. Obfuscan detects this feature by directly accessing the source file attribute of classes and storing the string content of the attribute. Removal of annotations is detected by Obfuscan by directly accessing and storing the attribute field of classes.

Other Tools. Although we built Obfuscan with a focus on detecting the use of ProGuard, it is able to detect apps that were obfuscated with other tools (cf. Table 5). Obfuscan is able to detect apps that were obfuscated using ReDex, Jack and DexGuard name obfuscation using Obfuscan's name obfuscation detection feature since all three tools use name obfuscation patterns that are identical with ProGuard's name obfuscation. Additionally, Obfuscan is able to detect DexGuard's more advanced removal of debug line numbers and annotations obfuscation features. We extended Obfuscan's name obfuscation detection feature to also cover the name obfuscation patterns implemented by yGuard and DashQ. To be able to detect Allatori's non-alphanumeric name obfuscation scheme, we extended Obfuscan and added detection support for restricted Windows keywords such as "AUX" or "NUL".

Evaluation. We implemented Obfuscan in Python and evaluated its efficacy by conducting a lab experiment using 100 real Android applications randomly selected from the F-Droid open source app market. We compiled two different versions of each sample app: One version did not use any means of obfuscation and one version that had ProGuard's name obfuscation for all application scopes, method name overloading, debug information removal, annotation removal, and source file removal enabled. Additionally, we acquired and tested 26 apps obfuscated with DexGuard, an expensive commercial tool, correctly identifying obfuscation in all 26.

Obfuscan correctly identifies nearly all obfuscation features of the 200 APKs dataset with a low false-positive rate and a high correlation coefficient (cf. Table 2). We manually investigated false positives and false negatives. Obfuscan falsely detected few class and method names as not obfuscated. In these cases, structures of the app were exempt from obfuscation, e.g., due to classes being

Feature	TP	TN	FP	FN	MCC
Class name obfuscation	98	100	0	2	0.980
Method name obfuscation	99	100	0	1	0.990
Field name obfuscation	100	92	8	0	**0.923**
Method name overloading	99	100	0	1	0.990
Debug information removed	100	100	0	0	1.000
Annotations removed	100	88	12	0	**0.886**
Source files removed	100	100	0	0	1.000

Table 2: Performance of OBFUSCAN for sample set of 200 APKs. Shown are true positive (TP), true negative (TN), false positive (FP), false negative (FN) predictions, and Matthews correlation coefficient (MCC).

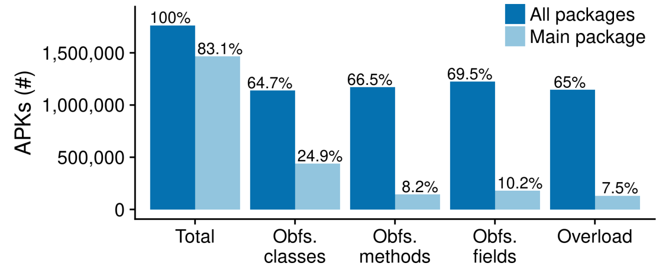

Figure 1: Comparison of obfuscation for different app structures including all packages and main package only. Overall obfuscation of apps considering all packages is increased due to library obfuscation.

marked as an interface. The false positive rate for field names is slightly higher than for other features. This is because ProGuard uses short strings for names (e.g., a and b) that are sometimes used as variables in unobfuscated apps. OBFUSCAN had no false positives for the debug information and source files removal feature. However, it falsely detected 12 apps as using the annotations removal feature. These false positives affect apps that do not use the code characteristics that are compiled to annotations (like inner classes).

Limitations. There are several obfuscation features that OBFUSCAN does not measure. Since OBFUSCAN focuses on the detection of the benign application of obfuscation, we do not look for packers or other techniques specifically used by malware. We excluded the heuristics for resource name and content obfuscation from our large scale measurement study for performance reasons. We evaluated a test set of 1,000 random apps from Google Play and could not find a single app using these features. Additionally, we did not implement class and string encryption detection. Both are advanced features and DexGuard, DexProtector, or GuardIT provide them as extensions to the more basic name obfuscation features. Finally, OBFUSCAN focuses on the detection of name obfuscation as implement by common tools. These heuristics conservatively estimate the prevalence of obfuscation at the cost of missing the use of name obfuscation algorithms by less popular tools. However, because OBFUSCAN reliably detects the removal of debugging information, we believe that this estimates a strong upper bound of the potential uses of other tools that are not ProGuard-related.

OBFUSCAN's annotation removal detection looks for app packages that do not include annotations. However, this heuristic might mislabel unobfuscated apps that naturally do not use annotations. Since it is hard to estimate the false positive rate for this heuristic, we excluded it from our measurement study in Section 4.

To test OBFUSCAN, we used apps from F-Droid rather than Google Play because we needed source code. While there is a chance that F-Droid apps differ from Google Play apps, this methodology was better than alternatives like writing test apps.

4 LARGE SCALE OBFUSCATION ANALYSIS

With OBFUSCAN we can answer our "**RQ1:** *How many apps are obfuscated, and what techniques are used?*" Therfore, we analyzed 1,762,868 [1] current free Android apps from Google Play to investigate the real-world use of the ProGuard family of Android obfuscation tools. To the best of our knowledge, this is the largest obfuscation detection analysis to-date for Android applications. Of those applications, OBFUSCAN detected the renaming obfuscation pattern implemented by the ProGuard family of obfuscation tools (cf. Section 2) in 1,137,228 (64.51%) apps.

Main Application Code Obfuscation. The high percentage of apps with obfuscated code would seem to indicate that many developers are obfuscating their apps. However, this statistic is misleading because a large percentage of apps are not intentionally obfuscated by the original developer. Instead, many apps simply include third-party libraries that use obfuscation, and the presence of an obfuscated library does not indicate that core application code is obfuscated. This fact means we need additional analysis to determine how often developers are actually obfuscating their apps.

To distinguish between apps that are obfuscated by their developer and apps that simply include obfuscated libraries, we analyze the obfuscation used by the declared main package of the application[2]. The main package is used as the universal identifier of the application (e.g. *com.google.maps*) and is necessarily implemented by the developer, so a choice to obfuscate the main package strongly indicates a choice to obfuscate at least some (if not all) of the original application code. We note that determining whether code is from a library or written by the developer is non-trivial, and this approach has the advantage of being scalable to millions of apps while not relying on potentially incomplete lists of libraries [2].

Our main package analysis found that only 24.92% of apps (439,232 apps) are intentionally obfuscated by the developer. In other words, *the vast majority of apps — representing millions of man-hours of development — are not protected using ProGuard as recommended for use in the official Android developer documentation [27].*

Obfuscation in Libraries. To get a better understanding of the included libraries in the Android ecosystem, we investigated the names of Android packages in all apps. Android packages follow Java naming conventions, allowing for the identification of larger scopes (e.g. the *com.google.ads.interactivemedia.v3.api* package can

[1] All free Android applications we were able to download from our geographic location.
[2] This distinction of main package vs. other packages was also performed by Linares-Vásquez et al. [34]

Scope	Packages	**Unique APKs**
com.google.ads.*	1,919,976	**681,102**
com.google.android.gms.*	24,095,920	**651,952**
android.support.v4.*	1,811,806	**192,497**
com.unity3d.*	432,856	**152,668**
org.fmod.*	135,524	**135,524**
android.support.v7.*	992,843	**117,680**
com.facebook.*	1,309,276	**106,178**
com.startapp.*	2,234,609	**88,242**
com.chartboost.*	491,612	**87,781**
com.pollfish.*	537,046	**44,851**

Table 3: Top 10 obfuscated libraries by total number of packages and number of APKs containing the libraries. Our analysis considers both main application code and libraries separately to determine the actual rates of use by end developers.

be traced to the *com.google.ads.** scope). Examining the included packages, we find that most of the external library obfuscation stems from a few, popular library frameworks (cf. Table 3). Examples include the Google Ad framework and the Google Mobile Service (GMS) framework used for Google authentication and search. Other commonly included obfuscated frameworks include the Facebook integration library and the FMOD audio playback library. The presence of these very popular libraries explains why many applications have obfuscated code, yet so few main packages are obfuscated.

Obfuscation Feature Popularity. Obfuscan provides the ability to examine use of individual ProGuard obfuscation features, and the use of name features for both entire applications and main packages only is shown in Figure 1. The "all package" category is measured as the number of apps containing any package with the obfuscation feature. This includes all libraries and the declared main package. The "main package" category is the number of apps with the obfuscation feature considering only the app's main package. We note that percentages of features used in the main package results are only among those apps with code in the main package.

We see first that class name obfuscation is the most popular feature, with 64.7% of all packages and 24.9% of main packages using it. Looking at other features shows a marked difference in feature use between libraries and main packages. While features that obfuscate method names, field names, and exploit function name overloading are used about as often as class name obfuscation in the all package analysis, they are infrequently used in main packages. One explanation is that library developers have a greater incentive to protect proprietary or sensitive internal APIs.

Overall, our findings indicate that the vast majority of app developers do not obfuscate their core code, and even when they do they do not use all of the available features. These results might indicate that developers either only obfuscate critical parts of their application or do not fully understand the concept of obfuscation.

Non-Proguard Obfuscation. While Obfuscan comprehensively covers features used by ProGuard, it also provides information about other forms of obfuscation. First, apps that do not contain debug info or source files are likely obfuscated, and so looking for those characteristics provides an upper bound on the number of apps in our dataset that are obfuscated by any non-ProGuard tool. As shown in Figure 1, we find that between 7.4 and 7.5% of apps in our data have these features for the main package, while between 11.7 and 13.2% of apps have these features for any class in the application. Additionally, we found 2,799 (0.16%) apps that use the advanced obfuscation feature of replacing class names with restricted keywords of the Windows operating system (e.g. "AUX", utilized by DexGuard and some Allatori configurations). By analyzing *classes.dex* files, we found 794 (0.05%) apps that were obfuscated with DexProtector and 207 (0.01%) apps obfuscated with Bangcle. Ultimately, these results together allow us to conclude that ProGuard is far more popular than any other obfuscation tool. This is because the classes using ProGuard-style name obfuscation greatly outnumber the scrubbed debugging or source files, which provide an upper bound on all other obfuscation tools.

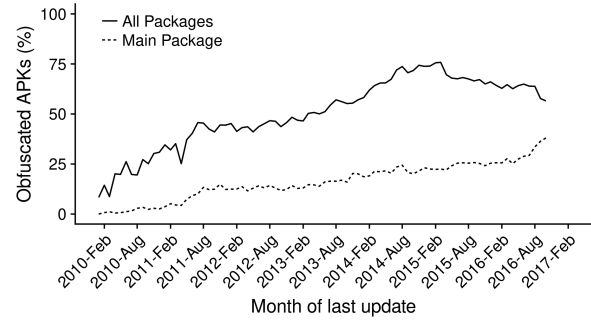

Figure 2: On average, more recently updated apps are more likely to be obfuscated.

Download Counts	Total Apps	**Obfs. Main Package**
0+	115,683	**27.30%**
10+	343,652	**26.34%**
100+	499,018	**24.74%**
1,000+	383,046	**24.13%**
10,000+	234,213	**23.95%**
100,000+	80,302	**25.50%**
1,000,000+	16,335	**29.15%**
10,000,000+	1940	**36.80%**
100,000,000+	160	**50.00%**

Table 4: Distribution of main package obfuscation for different download counts. More popular apps have a higher rate of main package obfuscation.

4.1 Obfuscation Trends

By comparing our obfuscation findings with Google Play metadata for all analyzed apps, we can develop further insights into the use of obfuscation in Android. In this subsection, we consider an app "obfuscated" if classname obfuscation is used, as this is the most common obfuscation feature supported by most obfuscation tools. As before, we distinguish between "all packages" and "main packages" for our analysis. We investigate the following trends in app obfuscation: main package obfuscation rate in relation to download numbers; average main package obfuscation by number of apps per developer; and obfuscation by app update date.

App Popularity. Google Play apps range from rarely downloaded side projects to popular and complex apps with dozens of developers

Apps per Account	Unique Accounts	**Avg. Obfs. of MP**
1	311,908	**21.83%**
2+	155,220	**21.24%**
10+	27,397	**26.50%**
100+	642	**34.37%**
250+	112	**35.29%**
500+	36	**68.41%**

Table 5: Average main package obfuscation for number of apps by Google Play account. Accounts with more apps have a higher average rate of main package obfuscation.

and millions of installs. Hence, different apps will have different incentives to obfuscate their code. We hypothesized that popular apps would be more likely to obfuscate their code as these apps are often more sophisticated and complex and face the greatest risks of plagiarism. To test this hypothesis, we examine the obfuscation rates for each download count category reported by Google Play.

Table 4 shows these results. We find that most apps — the 98.9% (1,655,914 apps) of apps with less than 1 million downloads — are obfuscated at roughly the same rate, ranging from 23.9% – 27.3%. As download counts increase further, we see an increase in obfuscation in the most downloaded apps from 29.15% of apps with more than one million downloads to 50.0% of apps with more than 100 million downloads. While this does confirm our initial expectation, we were surprised that even the most popular apps are only obfuscated on average half of the time.

Obfuscation by Google Play account. We also investigated if the number of published apps per Google Play account plays a role in the decision to obfuscate apps. Our hypothesis was that accounts with more submitted apps either belong to experienced developers or even companies specialized in app development and that apps from these accounts would show a higher obfuscation rate either due to a higher awareness or even previous experience of intellectual property theft or due to a higher perceived investment.

Table 5 shows the results. We find that apps from accounts with less than 100 apps have roughly the same average obfuscation rate between 21.8% – 26.5%. For accounts with 100 or more submitted apps this increases to about 35% and even to 68.4% for accounts with 500 and more apps. This increase in average app obfuscation seems to confirm our hypothesis that experienced developers or specialized companies with a large number of submitted apps use obfuscation more often. A likely explanation for this could be that more experienced developers and companies want to protect their intellectual property further. This could be the results from previous experiences of intellectual property theft, or the result of placing a higher value on their apps, as they are likely an important source of income for professional developers and specialized app companies.

Update Date. Figure 2 shows how all package and main package obfuscation rates vary when compared to the month of their most recent update; recent updates on average imply frequent maintenance of apps [43].[3] ProGuard is distributed with the Android SDK starting August 2009. The base ProGuard name obfuscation algorithm remained functionally unchanged, allowing Obfuscan to detect obfuscation for all included apps over the study period.

The figure shows a clear upward trend for both all packages and main packages, though as seen previously the overall obfuscation rate for all packages is much greater than main package obfuscation rate. More recently updated apps are more likely to be obfuscated as well. This could be indicative of greater developer sophistication or greater investment in terms of development time and intellectual property. In any case, it is clear that more recently updated apps are more likely to be obfuscated, though still at a low rate overall.

RQ1 Conclusions: This section addressed our **RQ1:** *How many apps are obfuscated, and what techniques are used?.* We found that a significant minority of apps are obfuscated by developers (24.92%), though obfuscated libraries are present in far more apps (64.51%). We also found that ProGuard was overwhelmingly the most popular obfuscation tool. Although these numbers a low compared to an ideal high rate of adoption, they are high enough that software tools and research should be compatible with obfuscated apps.

5 DEVELOPER SURVEY

To answer our second research question, "**RQ2:** *What are developers' awareness, threat models, experiences, and attitudes about obfuscation,*" we conducted an online survey of Android developers covering their obfuscation experience, the tools they use and their general knowledge and risk assessment concerning obfuscation and reverse engineering. We asked them if they had heard of obfuscation, if they knew what it was, if they had ever used it or decided against using it, and why. Additionally, we measured their awareness of "repackaging," "reverse engineering," "software plagiarism," and "obfuscation". We asked how strongly they feel that apps in general and their own apps in particular are threatened by the first three concepts. We followed this with a set of general questions about their Android development practices.[4] In this section, we briefly discuss the design of this survey as well as the results. The online study was approved by the Institutional Review Boards of all involved universities (See Appendix A for more details).

Depending on participants' prior answers, we asked up to three free text questions, the results of which we analyzed by using open coding with two researchers, developing an initial codebook and refining it iteratively, using it independently on the answers and resolving all conflicts with the help of a third researcher [11].

We collected a random sample of 62,462 email addresses of Android application developers listed in Google Play. We emailed these developers, introducing ourselves and asking them to take our online survey. A total of 561 people clicked on the link to our survey, visited our website and agreed to the study's consent form. Of these 561, 186 dropped out before answering the first question; another 67 participants were removed for dropping out later during the survey or providing answers that were nonsensical, profane, or not in English. Results for our survey are presented for the remaining 308 valid participants.

As common for developer studies [1], we compared participants to the larger population from which we sampled: we compared metadata of 3,159 Android apps associated with our survey participants to the metadata of 1.1M free and paid applications associated

[3]Unfortunately, our data collection only allowed us to collect the most recent data on an application, preventing us from getting ground truth on the changes in obfuscation of individual apps over time.

[4]Full questionnaire included in the appendix

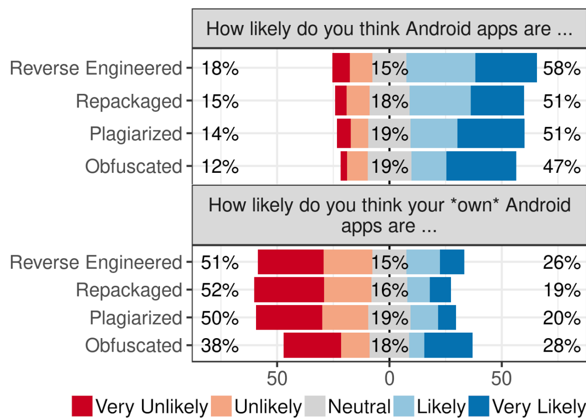

Figure 3: Likert plots of questions on risks of apps in general and risks to the participant's apps show that developers see themselves at much lower risk than the "average" app.

with the 62,462 email addresses to which we sent survey invitations (shown in Figure 4 in the Appendix).

We found a close resemblance in download counts per app (mean invited: 5.75, mean participated: 5.89, category 5 corresponds to 100–500 downloads, category 6 to 500–1,000 downloads), the average user rating (mean invited: 3.07, mean participated: 3.29) and the date of the last update as a measure of app age and long-term developer support (mean invited: 2015-11-18, mean participated: 2015-09-01). These similarities suggest that the developers who opted into our survey study resemble the random sample of Google Play Android developers.

5.1 Results and Takeaways

Obfuscation Experience. We found that the majority (241, 78%) of our participants had heard of software obfuscation in general, while 210 (68%) knew about obfuscation techniques for Android in particular. 187 (61%) had considered obfuscating one or more of their applications, of whom 148 (48%) actually did obfuscate one or more applications. While the majority of developers had heard of reverse engineering (253, 82%), software plagiarism (201, 65%) and software repacking (189, 61%) and felt that Android applications in general were severely threatened by plagiarism and malicious repacking, they had the impression that their own apps were less likely to face those threats than apps "in general". Figure 3 shows a Likert plot of these responses.

Reasons to obfuscate. The following results are reported for 101 developers who voluntarily specified reasons for using obfuscation in a free text answer. 63 developers (62.3%) used obfuscation to protect their intellectual property against malicious reverse engineering and theft. Interestingly, 14 (13.9%) participants used ProGuard only because it came pre-installed with Android Studio and was easy to use. 18 (17.8%) participants needed ProGuard's optimization features and stated that adding obfuscation was trivial. 4 (4%) participants apparently misunderstood the concept of obfuscation and enabled ProGuard to provide their users additional security, similar to encrypting files or using secure network connections. 7 (6.93%) configured obfuscation because there was a policy (either given by the company they worked for or a customer) that required it.

Verifying that obfuscation works. The following results are reported for the 69 participants who gave a free text answer on their method of verifying the success of obfuscating their app. 48 (69.6%) developers verified the correct use of obfuscation by decompiling the application and manually looking for obfuscation features (e.g., obfuscated package, class or method names). Six (8.7%) participants relied on the Android Studio toolchain and interpreted no warning or error messages as successful obfuscation. Four (5.8%) participants checked their apps' logfiles to verify their obfuscation. Finally, six (8.7%) other participants verified obfuscation by comparing the size of the non-obfuscated with the obfuscated version of an application.

Reasons to not obfuscate. Out of the 185 developers who gave reasons to not obfuscate in a free text answers, 81 (54.8%) thought about obfuscation and then decided against using it because they saw no reason to protect their application(s) against malicious reverse engineering, either because they open sourced their applications (17) or included no valuable intellectual property (64). 52 (35%) participants tried to use obfuscation and gave up because they felt overwhelmed by ProGuard's complexity. They could not get third party libraries working or had other issues such as non-working JavaScript interfaces. Five (3.2%) tried to understand the concept of obfuscation but failed. Eight (5.8%) participants mentioned company policies that did not allow them to obfuscate code. However, no one elaborated on those policies in more detail.

Use of Obfuscation Tools. Furthermore, 148 participants gave details on the obfuscation tools they had used. Most of them (127, 85.8%) had used ProGuard. 12 participants (8.1%) used the Jack toolchain[5]. 11 participants (7.4%) used DexGuard and 6 participants (4%) used ReDex. 4 participants mentioned other less popular obfuscation tools with only one appearance, like an obfuscation tool built into the Unity engine. Overall, 144 (97.3%) of the participants had used ProGuard or similar tools.

RQ2 Conclusions: This section addressed **RQ2:** *What are developers' awareness, threat models, experiences, and attitudes about obfuscation?* We found that survey participants are aware of obfuscation, but estimated the risk to their own apps as low. Many participants noted that obfuscation was simply not worth the extensive effort.

We also learned that many Android developers suffer from misconceptions (e.g., using obfuscation to secure network connections) and seem to be overwhelmed by using obfuscation correctly (e.g., the inability to obfuscate an app, but exclude certain components from obfuscation). Generally, we also observed the lack of a threat model: one participant explicitly stated "I wasn't sure my apps would be even popular enough so that someone would bother to copy them. If they would get popular, I'd release an update with obfuscation on." Many developers did not see a reason to obfuscate their own app(s) despite being aware of an abstract risk. One participant explicitly spoke of their experiences with piracy, stating "I see it as highly unlikely, that someone is actually interested in reverse engineering my code. However, I have encountered several fraud

[5]The Jack toolchain was deprecated in March 2017 (cf. https://android-developers.googleblog.com/2017/03/future-of-java-8-language-feature.html)

cases as an Android developer. All consisted of minimum reverse engineering efforts, i.e. people decompiled my app, changed the advertising ID code, repacked it, and published it under a different name." We find that the lack of concrete threat models explains a low motivation to obfuscate; to obtain a better understanding of the barriers to obfuscation, we decided to investigate the usability issues mentioned by a substantial number of participants in depth.

6 OBFUSCATION EXPERIMENT

The large scale measurement study and developer survey described above raised an interesting paradox: Roughly 50% of our survey participants claimed to have tried obfuscation in the past, but only 25% of the apps in our measurement study were obfuscated. We hypothesized that this discrepancy may be explained by the fact that developers may *attempt* obfuscation, but be unsuccessful due to difficulties in using their obfuscation tool. To test this hypothesis that the leading obfuscation tool might suffer from *usability problems*, we conducted an online experiment to investigate how developers interact with the ProGuard obfuscation framework. This study addresses our **RQ3:** *How usable is the leading obfuscation tool?*

Study Design. We designed an online, within-subjects study to compare how effectively developers could quickly write correct, secure ProGuard configurations. Again, we recruited developers with demonstrated Android experience from Google Play. Participants were assigned to complete a short set of Android obfuscation tasks, using ProGuard. All participants completed the same set of two ProGuard tasks. After finishing the tasks, participants completed a brief exit survey about the experience. We examined participants' submitted ProGuard configuration for functional correctness and security. The study was approved by our institutions' ethics review boards (see Appendix A for more details).

We chose to use ProGuard as the obfuscation tool for our experiment because it is pre-installed with Android Studio, the standard IDE for Android app development, and also because our online survey participants overwhelmingly used ProGuard.

Recruitment and Framing. Similar to our survey, we recruited Android developers from Google Play to participate in our developer study. We emailed 91,177 developers in batches, asking them to volunteer for a study on how Android developers use ProGuard to obfuscate apps. We did not mention security or privacy in the recruitment email. We assigned each invitee a unique pseudonymous ID to link their study participation to Google Play metadata without directly identifying them. Recipients who clicked the link to participate were directed to a page containing a consent form. After affirming they were over 18, consenting to the study, and indicating comfort completing a study in English, they were introduced to the study, given access to an Android Study project containing our skeleton app and instructions (including screenshots) on how to import it and set it up. We also provided brief instructions about the study infrastructure, which we describe next.

Experimental Setup. After reading the study introduction, participants were instructed to work on the tasks themselves. Our aim was to have developers write and test ProGuard configurations. We wanted to capture the ProGuard configuration and the Android application code that they typed. To achieve this, we prepared a Gradle-based Android application development project for Android Studio as a skeleton, compressed the project to a zip file, and provided a download link. We asked participants to download the zip file, import the project into their Android Studio development environment, work on the tasks, put their solutions in a new zip file, and upload this file to our study server. After uploading the solution zip file, we provided a link to the exit survey that allowed us to connect the ProGuard solutions to the survey responses.

To investigate possible usability issues with ProGuard, we asked participants to use ProGuard to complete two obfuscation tasks on the skeleton app we provided in the zip file.

We designed tasks that were short enough so that uncompensated participants would be likely to complete them before losing interest, but sufficiently complex to offer insights into the usability of ProGuard. Most importantly, we designed tasks to model real world problems that Android developers using ProGuard could reasonably be expected to encounter. We chose both tasks after investigating ProGuard centered StackOverflow discussions and GitHub repositories. Both tasks are amongst the most popular ProGuard related discussions on StackOverflow and represent the most popular modifications in ProGuard configuration files on GitHub.

For each task, participants were provided with stub code and some commented instructions. These stubs were designed to make the task clear without providing too much scaffolding and to facilitate our evaluation. We also provided Android application and ProGuard code pre-filled so participants could test their solutions.

Task 1 - Configure: The first task required participants to activate ProGuard within the default Gradle configuration file. The goal was to fully obfuscate the Android application. Participants were asked to solve this task so we could investigate their ability to complete a basic ProGuard configuration. Possible errors include the inability to activate obfuscation or a misconfiguration of ProGuard that disables obfuscation.

Task 2 - Obfuscate and Keep: The second task required developers to configure ProGuard to obfuscate one specific class (*SecretClass*) of our app, while keeping a second class (*OpenClass*) and its function (*doStuff()*) unobfuscated. To solve this task, developers were expected to use ProGuard's "`-keep`" flag for the *OpenClass* class.

The challenge for this task was to correctly use the "`-keep`" flag. Depending on the specified arguments, developers could potentially leave the *SecretClass* unobfuscated or obfuscate *OpenClass* instead.

Exit Survey. Once both tasks had been completed and the zip file was uploaded, participants were directed to a short exit survey.[6] We asked for opinions about the completed tasks, their assessment of their configurations for both tasks, general questions related to obfuscation and reverse engineering, and their previous experience with ProGuard and other Android obfuscation tools.

Evaluating Solutions. Once participants submitted solutions, we evaluated their correctness. Every solution was independently reviewed by two coders, using a codebook prepared ahead of time based on the official ProGuard configuration documentation. Differences between the two coders were reconciled by a third coder.

[6] We used LimeSurvey for this; the full questionnaire is available in the Appendix.

We assigned correctness scores to valid solutions only. To determine a correctness score, we considered several different ProGuard parameters. A participant's solution was marked correct (1) only if their solution was acceptable for every parameter; an error in any parameter or a parameter that weakened the ProGuard configuration security resulted in a correctness score of 0. To assess the correctness of Task 1, we evaluated the Gradle and ProGuard flags in participants' solutions. Whenever participants enabled ProGuard using both the "`minifyEnabled true`" and "`proguardFiles proguard-rules.pro`" options in the configuration file, we rated the solution correct. Solutions that did not specify one of these options or included the "`-dontobfuscate`" flag were rated incorrect.

For Task 2 correctness, we evaluated whether participants enabled obfuscation for the *SecretClass* class and its *doSecretStuff()* method but left the *OpenClass* class and its method *doStuff()* unobfuscated. Similar to Task 1, we required participants to enable obfuscation by using the "`minifyEnaled true`" and the "`proguardFiles proguard-rules.pro`" options. Additionally, correct solutions had to specify one of the following options "`-keep`", "`-keepclassmemebers`", "`-keepclasseswithmembers`", "`-keepnames`", "`-keepclassmembernames`", or "`-keepclasseswithmembernames`" for both the *OpenClass* class and the *doStuff()* method without including the *SecretClass* and its *doSecretStuff()* method. Solutions that did not meet these criteria were considered incorrect.

6.1 Results and Takeaways

In total, we sent 91,177 email invitations. Of these, 999 (1.9%) requested to be removed from our list, a request we honored.

766 people clicked on the link in the email. Of these, a total of 280 people agreed to our consent form; 202 (72.1%) dropped out without taking any action. We received zip files from the remaining 78 participants. We excluded eight submissions from further evaluation: one participant submitted a broken zip file, five submitted zip files without a ProGuard configuration file included, two submitted unmodified ProGuard configuration files.

The remaining 70 participants proceeded through at least one ProGuard task; of these, 66 started the exit survey, and 63 completed it with valid responses. Unless otherwise noted, we report results for the remaining 63 participants, who proceeded through all tasks and completed the exit survey with valid responses. Almost all (60, 95%) of our participants had heard of the concept of software obfuscation before, and 54 (85%) had been using ProGuard at least for one Android application in the past.

Most participants (49, 77%) mentioned an abstract threat of reverse engineering or malicious repackaging for Android applications in general. However, similar to the online survey we conducted in Section 5, only a small number of participants estimated a high risk for malicious repackaging for their own app(s).

Surprisingly, all of the 70 participants who changed the configuration for Task 1 submitted a correct solution by adding both the "`minifyEnabled true`" and "`proguardFiles proguard-rules.pro`" options.

Task 2 was correctly solved by only 17 (22%) participants, all of whom correctly solved Task 1 as well. Of the 53 incorrect solutions for Task 2, 30 solutions did not include the *-keep* option for the *OpenClass* class. These mistakes resulted in obfuscated classes that should be kept unobfuscated. 17 of the 53 incorrect solutions did include the *-keep* option but misspelled the package name for the *OpenClass* class. Six of the 53 incorrect solutions included the wildcard option for class names which disabled obfuscation for the *SecretClass* class.

41 of our participants rated their solutions as correct. However, only 11 of them actually submitted correct solutions for both tasks. Overall, 52 participants self-reported previous experience with ProGuard of which 13 correctly solved both tasks. Only one of the 11 participants with no previous ProGuard experience was successful.

RQ3 Conclusions: This section addressed our **RQ3:** *How usable is the leading obfuscation tool?* We found that all participants, regardless of their experience with ProGuard, were able to solve the trivial task to obfuscate the complete app with ProGuard. However, we found a low success rate for the task that required more complex configuration, which substantiated the usability problems mentioned in our developer survey. Being unfamiliar with ProGuard use essentially disqualified participants from configuring partial obfuscation. Critically, participants were unable to verify whether ProGuard had been configured correctly and whether it obfuscated successfully. These results underline a critical usability problem with ProGuard that likely contributes to low obfuscation rates in the wild.

7 DISCUSSION

Security through insignificance? Our large-scale analysis showed that the majority of developers do not take basic steps to protect their apps. Even for the most popular apps with upwards of 10,000,000 downloads, who are high risk candidates for obfuscation-related threats, the intentional obfuscation percentage remains below 50%. In our studies, participants assigned a low risk of obfuscation-related attacks to their apps while assuming a greater risk for the whole app ecosystem. Through provided write-ins we learned that many developers perceive their apps as too insignificant to ever fall prey to intellectual property theft or plagiarism. This "security through insignificance"-approach could prove fatal to the increasing number of small developers in the Android ecosystem.

Optional obfuscation: In addition to low initial motivation, the complexity of correctly using obfuscation further contributes to developer unwillingness to obfuscate. Cryptic error messages and confusing documentation do not increase motivation. Perhaps as a result, a certain mind-set seems to have contributed further to the rejection of obfuscation: some participants voiced concerns that obfuscation would destroy their "completed" applications. This view of obfuscation usage as an optional — not essential — development practice could play a larger role in hampering the acceptance of software obfuscation among developers.

Recommendations: Our findings indicate that there are two critical problems preventing widespread adoption of obfuscation in the Android ecosystem. The first is technical, and may have a technical solution: ProGuard is difficult to use correctly. We believe that it may be possible to automatically detect complicating factors (like WebView use) and automatically generate valid ProGuard configurations for developers. If successful, this would allow obfuscation

to be enabled by default within Android Studio and other development environments. The second problem is that developers are not motivated to deploy obfuscation given a low perceived risk and high perceived effort. Developers also view obfuscation as an optional, possibly "app destroying" step instead of an integral part of the build process. While improved interfaces and automation for obfuscation may improve the perceptions of effort, more research and education regarding the risks of plagiarism is needed. A technical solution may take the form of new obfuscation techniques or obfuscations applied by the market instead of relying on developers to protect themselves, their users, and the ecosystem at large.

8 THREATS TO VALIDITY

In this section, we detail issues that may have affected the validity of our results and the steps we have taken to ensure that our results are as accurate as possible.

App Analysis. Our dataset of 1.7 million apps was downloaded from public accessible Google Play Android apps. This is a common methodology, and like all similar studies we run the risk that paid apps or apps in other markets have different properties. These populations (paid apps in particular) may have additional incentives to obfuscate. However, we believe that the high overlap of apps that are available as both free and paid apps, and identical apps available in multiple markets, minimizes this risk.

Our choice of measuring main package obfuscation is not perfect; it is possible that a developer does not obfuscate the main package but obfuscates the remainder of the app. To estimate the frequency of this practice, we examine how many apps without main package obfuscation have obfuscated packages that do not have multiple occurrences in the overall dataset. We found that only 22,868 apps (1.30% of all apps in the dataset) meet this criteria. This establishes an upper bound on the error of this heuristic. We note that an alternative approach to main package analysis would have been to remove third-party library packages after identification with obfuscation-resistant library detection tools such as LibRadar [38], LibScout [2], or LibD [33]. This whitelist approach to package filtering would by design miss new or rarely used libraries, so we opted for the conservative approach of main package analysis.

Online Survey and Developer Study. As with any user study, our results should be interpreted in context. We chose an online study because it is difficult to recruit "real" Android application developers (rather than students) for an in-person lab study at a reasonable cost. Conducting an online study resulted in less control over the study environment, but it allowed us to recruit a geographically diverse sample.

Because we targeted developers, we could not easily take advantage of services like Amazon's Mechanical Turk or survey sampling firms. Managing online study payments outside such infrastructures is very challenging; as a result, we did not offer compensation and instead asked participants to generously donate their time. As might be expected, the combination of unsolicited recruitment emails and no compensation may have led to a strong self-selection effect, and we expect that our results represent developers who are interested and motivated enough to participate. However, as the recruitment in Figure 4 demonstrates, while our participants have higher average app ratings, the sample represents Google Play developers both in app popularity and frequency of updates.

In any online study, some participants may not provide full effort or may answer haphazardly. In this case, the lack of compensation reduces the motivation to answer non-constructively; unmotivated participants typically do not opt in to the study. We attempted to remove obviously low-quality data (e.g., responses that are entirely invective) before analysis, but cannot discriminate perfectly.

9 RELATED WORK

Software obfuscation has been studied as defense against reverse engineering [13], to prevent intellectual property attacks [14], as disguise for malware [52], and to avoid user profiling [50]. Researchers successfully employed code obfuscation techniques to avoid detection tools, including anti-malware software [44, 45, 54], repackaging detection algorithms [31], and app analysis tools [29], although performance of anti-malware software improved in a more recent study [39]. A number of works detail different obfuscation techniques in general [6, 12, 13, 52], for the Java programming language [10, 30, 46], and for Android apps in particular [22, 25, 44]. Research on Android app obfuscation has focused on reversing obfuscation [5, 49], analyzing an app in spite of obfuscation [26, 28, 34, 53], the detection of repacked malware [23, 24, 35, 47], or identification of third-party libraries [2, 38].

Previous Android developer studies were performed in the context of privacy, Trusted Layer Security/Secure Sockets Layer (TLS/SSL) security, and cryptographic Application Programming Interfaces (APIs). Balebako et al. performed interviews and online surveys to investigate how app developers make decisions about privacy and security, identifying several hurdles and suggesting improvements that would help user-privacy [3, 4]. Jain et al. suggested design changes to the Android Location API based on the results of a developer lab study [32]. Fahl et al. and Oltrogge et al. conducted developer surveys and interviews, revealing deficits in the handling of TLS/SSL and suggesting several improvements [20, 21, 41]. Nadi et al. found in a study that Java developers struggle with perceived low-level cryptography APIs [40]. Concerning obfuscation on the Android platform, Ceccato et al. assessed in experiments the impact of Java code obfuscation on the code comprehension of students, finding that obfuscation delays, but not prevents tampering [7, 8]. Pang et al. surveyed 121 developers about their knowledge concerning app energy consumption [42]. Compared to these works, our root cause analysis focuses on obfuscation knowledge and ability to use the obfuscation tool ProGuard among Google Play developers. Similar to recent work on how information sources influence code security [1], we find that developers are generally aware of benefits and basic use, but fail to correctly obfuscate in complex scenarios.

Finally, in a pre-print concurrent with our work, Dong et al. also investigate the use of obfuscation in the Android ecosystem [16]. While that work is solely focused on technical measurements of obfuscation (similar in focus to our Sections 3 and 4), our research works with the developers responsible for obfuscation to determine the root causes of why apps are or are not obfuscated. Our app measurements are more comprehensive (1,762,868 apps from Google Play market vs. 114,560 apps) and use measurement techniques

grounded in specifications of the most common obfuscation tools (instead of machine learning approaches).

10 CONCLUSION

This paper presents the first comprehensive evaluation of the state of software obfuscation for benign Android applications. We built Obfuscan to analyze the use of obfuscation in 1,762,868 free Android applications available in Google Play. Our investigation reveals that 439,232 were obfuscated by their developers, leaving more than 75% unprotected against malicious repacking. In an online survey with 308 Google Play developers, 78% of the participants had heard of obfuscation while only 48% actually used software obfuscation – more than 85% of the participants used ProGuard – in the past. Interestingly, the majority of the participants recognized that software obfuscation in general is a laudable approach to protect against malicious repackaging. However, only few of them saw a reason to protect their own apps. Finally, in a within-subjects study with 70 real Android developers, we learned that 78% of the participants could not correctly complete a realistic ProGuard obfuscation task. Participants who self-reported no previous experience with ProGuard had a negligible chance to correctly obfuscate the study application beyond the trivial option to obfuscate it entirely.

Overall, our studies show that the current use of software obfuscation for benign Android applications leaves manifold challenges for future research. We find that both misconceptions about software obfuscation many of our participants suffered from and the challenges in using ProGuard correctly seem to be the root cause for the low adoption rate of software obfuscation in the Android ecosystem. Hence, future research needs to find more effective ways to make the concept and relevance of software obfuscation concepts accessible to Android developers and should work on more usable software obfuscation tools.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grant numbers CNS-1526718 and CNS-1562485. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *Proc. 37th IEEE Symposium on Security and Privacy (SP'16)*. IEEE, IEEE, Los Alamitos, CA, USA, 289–305. https://doi.org/10.1109/SP.2016.25

[2] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *Proc. 23nd ACM Conference on Computer and Communication Security (CCS'16)*. ACM, New York, NY, USA, 356–367. https://doi.org/10.1145/2976749.2978333

[3] R. Balebako and L. Cranor. 2014. Improving App Privacy: Nudging App Developers to Protect User Privacy. *IEEE Security & Privacy* 12, 4 (July 2014), 55–58. https://doi.org/10.1109/MSP.2014.70

[4] Rebecca Balebako, Abigail Marsh, Jialiu Lin, Jason I Hong, and Lorrie Faith Cranor. 2014. The privacy and security behaviors of smartphone app developers. In *Proc. Workshop on Usable Security (USEC'14)*. The Internet Society, Reston, VA, USA.

[5] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *Proc. 23nd ACM Conference on Computer and Communication Security (CCS'16)*. ACM, New York, NY, USA, 343–355. https://doi.org/10.1145/2976749.2978422

[6] Jean-Marie Borello and Ludovic Mé. 2008. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology* 4, 3 (2008), 211–220. https://doi.org/10.1007/s11416-008-0084-2

[7] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. 2009. The effectiveness of source code obfuscation: An experimental assessment. In *Proc. 17th IEEE International Conference on Program Comprehension (ICPC'09)*. IEEE, IEEE, Los Alamitos, CA, USA, 178–187.

[8] Mariano Ceccato, Massimiliano Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. 2014. A Family of Experiments to Assess the Effectiveness and Efficiency of Source Code Obfuscation Techniques. *Empirical Software Engineering* 19, 4 (Aug 2014), 1040–1074. https://doi.org/10.1007/s10664-013-9248-x

[9] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. 2013. MAST: Triage for Market-scale Mobile Malware Analysis. In *Proc. 6th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'13)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2462096.2462100

[10] Jien-Tsai Chan and Wuu Yang. 2004. Advanced Obfuscation Techniques for Java Bytecode. *Journal of Systems and Software* 71, 1-2 (April 2004), 1–10. https://doi.org/10.1016/S0164-1212(02)00066-3

[11] Kathy Charmaz. 2014. *Constructing grounded theory*. SAGE Publications, Thousand Oaks, CA, USA.

[12] Christian Collberg and Jasvir Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection* (1st ed.). Addison-Wesley Professional, Boston, MS, USA.

[13] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A taxonomy of obfuscating transformations*. Technical Report. Department of Computer Science, The University of Auckland, New Zealand.

[14] C. S. Collberg and C. Thomborson. 2002. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering* 28, 8 (Aug 2002), 735–746. https://doi.org/10.1109/TSE.2002.1027797

[15] J. Crussell, C. Gibler, and H. Chen. 2015. AnDarwin: Scalable Detection of Android Application Clones Based on Semantics. *IEEE Transactions on Mobile Computing* 14, 10 (Oct 2015), 2007–2019.

[16] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. 2018. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. *arXiv:1801.01633 [cs]* 1801 (Jan 2018). http://arxiv.org/abs/1801.01633

[17] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 393–407. http://dl.acm.org/citation.cfm?id=1924943.1924971

[18] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2, Article 5 (Jun 2014), 29 pages. https://doi.org/10.1145/2619091

[19] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A Study of Android Application Security. In *Proc. 20th Usenix Security Symposium (SEC'11)*. USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=2028067.2028088

[20] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)*. ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/2382196.2382205

[21] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013. Rethinking SSL Development in an Appified World. In *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/2508859.2516655

[22] P. Faruki, A. Bharmal, V. Laxmi, M. S. Gaur, M. Conti, and M. Rajarajan. 2014. Evaluation of Android Anti-malware Techniques against Dalvik Bytecode Obfuscation. In *Proc. 13th IEEE International Converence on Trust, Security and Privacy in Computing and Communications (TrustCom'14)*. IEEE, Los Alamitos, CA, USA, 414–421. https://doi.org/10.1109/TrustCom.2014.54

[23] Parvez Faruki, Vijay Ganmoor, Vijay Laxmi, M. S. Gaur, and Ammar Bharmal. 2013. AndroSimilar: Robust Statistical Feature Signature for Android Malware Detection. In *Proc. 6th International Conference on Security of Information and Networks (SIN'13)*. ACM, New York, NY, USA, 152–159. https://doi.org/10.1145/2523514.2523539

[24] Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek. 2015. *Obfuscation-resilient, efficient, and accurate detection and family identification of Android malware*. Technical Report. Department of Computer Science, George Mason University, Virginia.

[25] Sudipta Ghosh, S. R. Tandan, and Kamlesh Lahre. 2013. Shielding Android Application Against Reverse Engineering. *International Journal of Engineering Research & Technology* 2, 6 (2013). http://www.ijert.org/view-pdf/4095/shielding-android-application-against-reverse-engineering

[26] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. 2017. CodeMatch: Obfuscation Won't Conceal

Your Repackaged App. In *Proc. 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, New York, NY, USA, 638–648. https://doi.org/10.1145/3106237.3106305

[27] Google Inc. 2017. Shrink Your Code and Resources. https://developer.android.com/studio/build/shrink-code.html. (Apr 2017).

[28] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. 2013. Juxtapp: A Scalable System for Detecting Code Reuse among Android Applications. In *Proc. 9th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'12)*, Ulrich Flegel, Evangelos Markatos, and William Robertson (Eds.). Springer, Berlin, Heidelberg, 62–81. https://doi.org/10.1007/978-3-642-37300-8_4

[29] Johannes Hoffmann, Teemu Rytilahti, Davide Maiorca, Marcel Winandy, Giorgio Giacinto, and Thorsten Holz. 2016. Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation. In *Proc. 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*. ACM, New York, NY, USA, 139–141. https://doi.org/10.1145/2857705.2857737

[30] T. W. Hou, H. Y. Chen, and M. H. Tsai. 2006. Three control flow obfuscation methods for Java software. *IEE Proceedings - Software* 153, 2 (April 2006), 80–86. https://doi.org/10.1049/ip-sen:20050010

[31] Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu. 2013. A Framework for Evaluating Mobile App Repackaging Detection Algorithms. In *Proc. 6th International Conference on Trust and Trustworthy Computing (TRUST'13)*, Michael Huth, N. Asokan, Srdjan Čapkun, Ivan Flechais, and Lizzie Coles-Kemp (Eds.). Springer, Berlin, Heidelberg, 169–186. https://doi.org/10.1007/978-3-642-38908-5_13

[32] Shubham Jain and Janne Lindqvist. 2014. Should I Protect You? Understanding Developers' Behavior to Privacy-preserving Apis. In *Proc. Workshop on Usable Security (USEC'14)*. The Internet Society, Reston, VA, USA.

[33] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo. 2017. LibD: Scalable and Precise Third-Party Library Detection in Android Markets. In *Proc. 39th IEEE/ACM International Conference on Software Engineering (ICSE'17)*. IEEE, Los Alamitos, CA, USA, 335–346. https://doi.org/10.1109/ICSE.2017.38

[34] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2014. Revisiting Android Reuse Studies in the Context of Code Obfuscation and Library Usages. In *Proc. 11th Working Conference on Mining Software Repositories (MSR'14)*. ACM, New York, NY, USA, 242–251. https://doi.org/10.1145/2597073.2597109

[35] Cullen Linn and Saumya Debray. 2003. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proc. 10th ACM Conference on Computer and Communication Security (CCS'03)*. ACM, New York, NY, USA, 290–299. https://doi.org/10.1145/948109.948149

[36] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. 2015. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, USA, 89–103. https://doi.org/10.1145/2742647.2742668

[37] Hiroshi Lockheimer. 2012. Android and Security. http://googlemobile.blogspot.com/2012/02/android-and-security.html. (Feb 2012).

[38] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In *Proc. 38th IEEE/ACM International Conference on Software Engineering (ICSE'16)*. ACM, New York, NY, USA, 653–656. https://doi.org/10.1145/2889160.2889178

[39] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. 2015. Stealth attacks: An extended insight into the obfuscation effects on Android malware. *Computers & Security* 51 (2015), 16–31. https://doi.org/10.1016/j.cose.2015.02.007

[40] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs?. In *Proc. 38th IEEE/ACM International Conference on Software Engineering (ICSE'16)*. ACM, New York, NY, USA, 935–946. https://doi.org/10.1145/2884781.2884790

[41] Marten Oltrogge, Yasemin Acar, Sergej Dechand, Matthew Smith, and Sascha Fahl. 2015. To Pin or Not to Pin Helping App Developers Bullet Proof Their TLS Connections. In *Proc. 24th Usenix Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 239–254. http://dl.acm.org/citation.cfm?id=2831143.2831159

[42] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. 2016. What Do Programmers Know About Software Energy Consumption? *IEEE Software* 33, 3 (May 2016), 83–89. https://doi.org/10.1109/MS.2015.83

[43] R. Potharaju, M. Rahman, and B. Carbunar. 2017. A Longitudinal Study of Google Play. *IEEE Transactions on Computational Social Systems* 4, 3 (Sept 2017), 135–149.

[44] M. Protsenko and T. Müller. 2013. PANDORA applies non-deterministic obfuscation randomly to Android. In *Proc. 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE'13)*. Research Publishing Services, Singapore, 59–67. https://doi.org/10.1109/MALWARE.2013.6703686

[45] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. 2014. Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks. *IEEE Transactions on Information Forensics and Security* 9, 1 (Jan 2014), 99–108. https://doi.org/10.1109/TIFS.2013.2290431

[46] Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. 2005. Java obfuscation approaches to construct tamper-resistant object-oriented programs. *IPSJ Digital Courier* 1 (2005), 349–361.

[47] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. 2013. Mobile-sandbox: Having a Deeper Look into Android Applications. In *Proc. 28th ACM Annual Symposium on Applied Computation (SAC'13)*. ACM, New York, NY, USA, 1808–1815. https://doi.org/10.1145/2480362.2480701

[48] Vasant Tendulkar and William Enck. 2014. An application package configuration approach to mitigating Android SSL vulnerabilities. In *Proc. 2014 Mobile Security Technologies Workshop (MoST'14)*. IEEE, Los Alamitos, CA, USA.

[49] S. K. Udupa, S. K. Debray, and M. Madou. 2005. Deobfuscation: reverse engineering obfuscated code. In *Proc. 12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE, Los Alamitos, CA, USA, 10pp. https://doi.org/10.1109/WCRE.2005.13

[50] Imdad Ullah, Roksana Boreli, Salil S. Kanhere, and Sanjay Chawla. 2014. ProfileGuard: Privacy Preserving Obfuscation for Mobile User Profiles. In *Proc. 13th Workshop on Privacy in the Electronic Society (WPES'14)*. ACM, New York, NY, USA, 83–92. https://doi.org/10.1145/2665943.2665961

[51] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A Measurement Study of Google Play. *ACM SIGMETRICS Performance Evaluation Review* 42, 1 (June 2014), 221–233. https://doi.org/10.1145/2637364.2592003

[52] I. You and K. Yim. 2010. Malware Obfuscation Techniques: A Brief Survey. In *Proc. 2010 International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA'10)*. Springer, Berlin, Heidelberg, 297–300. https://doi.org/10.1109/BWCCA.2010.85

[53] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: Towards Obfuscation-resilient Mobile Application Repackaging Detection. In *Proc. 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'14)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2627393.2627395

[54] Min Zheng, Patrick P. C. Lee, and John C. S. Lui. 2013. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems. In *Proc. 9th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'12)*, Ulrich Flegel, Evangelos Markatos, and William Robertson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 82–101. https://doi.org/10.1007/978-3-642-37300-8_5

[55] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. 2013. Fast, Scalable Detection of "Piggybacked" Mobile Applications. In *Proc. 3rd ACM Conference on Data and Application Security and Privacy (CODASPY'13)*. ACM, New York, NY, USA, 185–196.

[56] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proc. 2nd ACM Conference on Data and Application Security and Privacy (CODASPY'12)*. ACM, New York, NY, USA, 317–326. https://doi.org/10.1145/2133601.2133640

[57] Y. Zhou and X. Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *Proc. 8th Symposium on Usable Privacy and Security (SOUPS'12)*. IEEE, Los Alamitos, CA, USA, 95–109. https://doi.org/10.1109/SP.2012.16

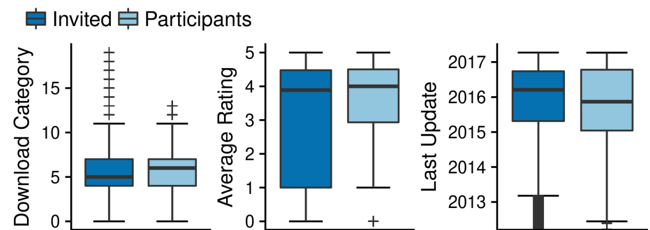

Figure 4: App metadata associated with invited email addresses compared to metadata from our participants: We find a close resemblance in download category (as classified by Google Play), ratings and currentness of last update. We compared the distributions using Mann-Whitney-U tests, but our results were inconclusive due to a number of outliers. Nonetheless, we observe similar interquartile ranges: while the invited population leans to being more spread out than our participant population, the populations are similar in median and mean with the invited population having heavier tails.

A ETHICAL CONSIDERATIONS

We conducted two user studies in the context of this paper. Both the survey presented in Section 5 and the developer study in Section 6 were approved by the Institutional Review Board (US) and ethical review board (Germany) of all involved universities. Additionally, the strict data and privacy protection laws in Germany were taken into account for collecting, processing and storing participants' data. Our user studies were targeted towards Android developers who had made their app public by offering it on Google Play. For ecological validity reasons we decided against recruiting local computer science students. To reach this rather specific group of Android developers, we gathered email addresses from developers who had published apps on Google Play from their public Google Play profiles. We selected a random sample and emailed them an invitation to one of our studies (This participant recruitment procedure is in line with work by Acar et al. [1]). Our invitation email included a link to our website, where they could access information about the purpose of our research, a consent form that explained how participant data would be used and a contact form. The email further included a link to be blacklisted; hashes of the blacklisted email addresses are shared across several research groups participating in similar developer studies.

B ONLINE SURVEY

General Questions

- Which of these have you heard of in the context of Android apps? Please check all that apply.
  (Reverse Engineering, Repackaging of Software, Software Plagiarism, Obfuscation)
- How likely do you think Android apps are …
  (Reverse Engineered, Repackaged, Software Plagiarism, Obfuscated), scale: (Very Unlikely, Unlikely, Neutral, Likely, Very Likely, I don't know)
- How likely do you think your *own* Android apps are...
  (Reverse Engineered, Repackaged, Software Plagiarism, Obfuscated), scale: (Very Unlikely, Unlikely, Neutral, Likely, Very Likely, I don't know)
- How much do you feel the intellectual property of your *own* Android apps is threatened by...
  (Reverse Engineering, Software Plagiarism), scale: (Very Unlikely, Unlikely, Neutral, Likely, Very Likely, I don't know)

Terminology

- Reverse engineering is:
  (Translate binary files to source code, Translate source code to binary files, Analysis of pure source code, Analysis of binary files, Reconstruction of app logic, Testing an app's functionality, I don't know, Other [with free text])
- Reverse engineering can be used for:
  (Understanding an app's logic, Circumvention of licence or security checks, Repackaging of an app, Stealing IP addresses, Attacks on Android users who have your app installed, Remote attacks on mobile phones, I don't know, Other [with free text])
- Software plagiarism is:
  (Repackaging existing software and rebranding it as your own, Use of third party open source code in your software, Imitating software to trick users, Copy pasting code found on the internet, I don't know, Other [with free text])
- Software plagiarism can be used for:
  (Obtaining software revenue, Distributing disguised malware, Attacking users that have your app installed, Attacking distribution services, I don't know, Other [with free text])
- Obfuscation is:
  (Making source code unreadable or difficult to understand so only authorized developers can work on it, Making source code unreadable or difficult to understand before compilation, Hiding binaries from the user, Preventing acces to the deployed application, I don't know, Other [with free text])
- Obfuscation can be used for:
  (Making reverse engineering more difficult, Prevent others from attacking vulnerabilities within your application, Hiding the logic within your application, Optimization of app performance, I don't know, Other [with free text])
- Have you heard of obfuscation before?
  (Yes, No, Uncertain)
- Have you ever thought about using obfuscation?
  (Yes, No, Uncertain)
- Did you obfuscate at least once before?
  (Yes, No, Uncertain)

Obfuscation tools

- Please select all Android obfuscation tools that you have heard of prior to this study.
  (ProGuard, DexGuard, Jack, DashO, ReDex, Harvester, Other [with free text])
- Please select all Android obfuscation tools that you have used before.
  (ProGuard, DexGuard, Jack, DashO, ReDex, Harvester, Other [with free text])
- Please select all Android obfuscation tools that you have actively decided against using.
  (ProGuard, DexGuard, Jack, DashO, ReDex, Harvester, Other [with freetext])
- Which tools do you use to remove unused library code?
  (ProGuard "Minify", Android Studio "Minify", I remove it manually, I never remove unused library code from my apps, Other [with free text])

Obfuscation 1

- How did you first encounter obfuscation?
  [Free text]
- How many apps have you worked on?
  [Number input]
- How many of those where obfuscated?
  [Number input]
- Why did you use obfuscation on those apps?
  [Free text]
- Why did you decide against obfuscating apps?
  [Free text]
- Did you verify that obfuscation was successful?
  (yes, no)
- How did you verify if obfuscation was successful?
  [Free text]
- Why did you decide against using obfuscation?
  [Free text]

B.1 ProGuard Study - Exit Survey

After completing the programming task, developers were asked to fill out a final survey.

Tasks

Do you think you solved the tasks correctly?
(Task1, Task2), scale: (Yes, No, I don't know)
Do you have additional comments on the tasks?
[Free text]
Followed by the `General Questions, Terminology` and `Obfuscation tools` question groups from the online survey (cf. Appendix B)

ProGuard

- What do you use Proguard for?
(Testing, Minifying Code, Optimization, Obfuscation)
- After using Proguard, how did you verify that it achieved its goal?
(I do not verify that Proguard worked, Reverse Engineering, Other [with free text])
- Why have you never used Proguard before?
(No need, Never heard of it, Too complicated, I have other tools, Other [with free text])