Vision: Alleviating Android Developer Burden on Obfuscation

Geoffrey Hecht
ISCLab, Department of Computer Science, University of Chile

Cyprien Neverov
ISCLab, Department of Computer Science, University of Chile

Alexandre Bergel
ISCLab, Department of Computer Science, University of Chile

ABSTRACT

Mobile applications (apps) have gained an increasing importance in the field of software engineering as they are becoming one of the most widely used type of software. In the Android ecosystem, obfuscation tools are available to optimize, reduce the size and protect the intellectual properties of apps. However, despite the clear advantages provided by obfuscation most apps do not use it, often because of the difficulties induced by the usage of obfuscation which requires writing rules to keep a usable app. In this paper, we identify the concrete challenges encountered by app developers who wish to use obfuscation in their apps. In addition, we propose an approach using crowdsourcing to automatically generate rules, when static analysis is not sufficient. With the knowledge gained from hundreds of projects, we hope to lighten the burden on developers when writing rules.

ACM Reference Format:
Geoffrey Hecht, Cyprien Neverov, and Alexandre Bergel. 2020. Vision: Alleviating Android Developer Burden on Obfuscation. In *IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3387905.3388611

1 INTRODUCTION

The Android documentation recommends to use a program optimizer, generally called obfuscators, such as ProGuard[1] or the new Google's R8[2] to optimize, shrink, and obfuscate apps. One of their main functions is to shorten the identifier names and remove the unused code and resources, thus greatly reducing app's size. By default, identifiers are renamed using lexicographic order {a,b,...,aa,ab,...} (see Listing 1)[3] but custom mapping file can also be configured by developers to use other identifiers.

Obfuscation is also used to protect intellectual property [12] and offers a protection against piracy, including apps cloning. Cloning is a very serious threat since more than 13% of apps available on Android markets are clones or repackagings of legitimate apps, modified with malicious code or advertisements to the attacker's benefits [3].

[1]ProGuard by Dexguard: www.guardsquare.com/en/products/proguard
[2]R8 optimizer by Google: https://r8.googlesource.com/r8
[3]Inspired by https://github.com/realm/realm-java/issues/4909

MOBILESoft '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7959-5/20/05…$15.00
https://doi.org/10.1145/3387905.3388611

However, despite the default integration of obfuscators in the Android Gradle plugin and the listed advantages for both end-users and developers, the use of obfuscators is still not widespread enough, in particular on the Google Play Store where only 25% of apps use obfuscation [13]. This could be explained by the ignorance of the developers on such tools, but a previous study [13] has shown that 75% of them are aware of the benefits of obfuscators. 55% of them thought about using it, but saw no valid reasons and 35% participants tried to use obfuscation, but gave up because of the complexity of the tasks. Indeed, in theory, it is only required to change the value of a boolean from false to true in the `build.gradle` to activate obfuscation. But in practice, it will often be necessary for the developer to write rules in a configuration file to keep the app usable. For Listing 1, a rule that would not obfuscate the subclass of DBObject would be: `-keep class * extends app.database.DBObject`. The aforementioned study also tested 70 developers among which only 17 were able to obfuscate a realistic sample app leading to the conclusion that "more work is needed to make obfuscation tools more usable" [13].

While previous researches investigated how and to what extent obfuscators are used by apps [3–5, 12, 13], to the best of our knowledge, none of them aimed to identify or to propose solutions to the concrete challenges that are encountered by developers willing to use obfuscators. While faster, more robust and more efficient obfuscation are being proposed [2, 8, 14], especially since the arrival of R8 [11], it appears that the difficulties encountered by developers while writing rules are neglected since the same rules have to be written by developers. This is why in this paper, we propose an approach to automatically generate obfuscator's rules.

Listing 1: Example of a database related class using reflection

```
//Without obfuscation
public class DataClassFetcher {
 private String fetchUUID(DBObject dbobject) {
  Class dataClass = loadDataClass(dbobject);
  String field = retrievePKField(dataClass);
  Method getter =
       dataClass.getDeclaredMethod("get"+field);
  return String.valueOf(getter.invoke(getter));
}

//With obfuscation
public class a {
 private String a(abc a) {
  Class b = e(a);
  String c = k(b);
//Crash if the method "get"+c has not been preserved
  Method d =  b.getDeclaredMethod("get"+c);
  return String.valueOf(d.invoke(d));
}
```

The vision defined in this paper is summarized as follows:

- We raise awareness about the difficulties that may discourage developers to use obfuscators and advocate for the development of tool-based approach to ease this burden;
- We propose the usage of crowdsourcing as a complement to the already-used static analysis;
- We present the risks associated with our approach and an inconclusive experience with machine learning.

2 THE DEVELOPER'S BURDEN

2.1 Identification of challenges

***i)* Introduction of bug:** The first major drawback of using an obfuscator is that in some situations the use of obfuscators may lead to bugs or crashes in the apps. Common causes of bugs are the usage of reflection, code called from Java Native Interface, data classes, opening of resources inside an APK or annotations [7]. Indeed, if a package, class or a method has been renamed, it may not be found by the system at runtime. For example, in Listing 1 a crash will occur when using `getDeclaredMethod("get"+c))` since the original method is now renamed to another identifier. Obfuscators use static analysis to determine which parts of the code can be removed or safely renamed. However, this static analysis is limited to simple cases and bugs may still appear. Use of libraries, configuration files, requests to a database or over a network make static analysis insufficient to know if a class should be kept or not. To avoid this, a developer can configure the obfuscation process with a file that specifies a set of `keep` rules that tells the obfuscator to keep the code it would otherwise remove or rename.

***ii)* Need for expertise:** Even if the syntax for writing rules is well-documented and not particularly complex, developers must master it to write efficient rules. In addition, developers must be able to guess when a class or a method might need specific rules, this implies a good knowledge of the apps and its libraries but also of the limits of the obfuscator's static analyzer. Sometimes it is even necessary to know what will happen at compilation time. For example, the bytecode produce by Kotlin classes will contains some Kotlin specific metadata which is, among other things, used for reflection. But the default behavior of Proguard is to remove this metadata [9], therefore leading to bugs if developers does not write rules to keep them.

***iii)* Complicated debugging:** In addition to that, obfuscation increases the difficulty of debugging because the error's output may refer to obfuscated identifiers in the stack trace. Developers must therefore de-obfuscate the stack trace or search for corresponding source code's methods in a mapping file, which can contain several thousand lines of correspondence between original and obfuscated identifiers. Each version of an app has a distinctive mapping file, therefore in case of a user bug reports, the version should be identified and the correct mapping file retrieved. This problem is further amplified by the fact that the default build for development under Android Studio is the *debug* build that does not include obfuscation. Thus, bugs related to this phase, can only be detected in the *release* build, which should be tested at runtime.

***iv)* Handling of warnings:** In addition to the rules they have to write, developers must also determine whether or not the warnings provided during the static analysis are relevant or not, especially since the build of the app will fail as long as there are such warnings. Often, warnings can be ignored by the developer, since they will not lead to any bugs. In this case, the `-dontwarn` command should be used to tell the obfuscator that it could ignore the problem. In most cases the warning is triggered by a class that cannot be referenced by the obfuscator and those cases are irrelevant for the app.

***v)* Impact of libraries:** Android apps heavily rely on the usage of external libraries [1], which has the effect of amplifying all the previously listed challenges. Not only developers must be able to write the rules specific to apps, but they also need to handle bugs, warnings and rules related to libraries obfuscation. In theory, libraries can include a `consumerProguardRules` file containing the libraries specific rules which will be automatically processed during the compilation. However the use of `consumerProguardRules` is not systematic, it is common for library developers to list the needed rules in their documentation instead. Sometimes, no rules are provided whatsoever [7]. In any cases, other rules might be needed when a developer is going to use, for example, heritage or reflection mechanisms on classes in the library. In this case, the developer should be able to write library-related rules specific to his case, or find them in the library documentation if available. An example of such a rule could be something like: `-keep class * implements com.some.library.NonObfuscableObject` which means keep from being obfuscated all of the classes that implement this interface from the library. In Listing 1, the data class might be a part of some library, as it is indeed the case in the example which inspired us on Github.

2.2 A concrete example

To illustrate how problems may arise from the usage of obfuscator, consider the open-source app Budget Watch (version 0.21.4)[4]. This small app contains only 43 classes and the developer tried to write rules for Proguard before finally disabling obfuscation. A comment highlights debugging reasons (challenge *iii*). For such a small number of classes, a mapping file of more than 27,400 lines is generated, and without rules 1,075 warnings are triggered (challenge *iv*). 13 rules are written for the libraries (challenge *v*), essentially guava, and although it seems that the developer found some rule example online, most of the classes are marked by both a `keep` and a `dontwarn` which demonstrate that the developer did not really understand them (challenge *ii*). In this case, only one keep is really needed, without it the app crashes (challenge *i*). Interestingly this is the class `android.support.v7.widget.SearchView` which is part of the support librairies provided by Google to allow support of multiple API versions and devices in one app. The Android documentation does not mention the need to write this rule [6] and a research on stackoverflow shows that it is a common problem encountered by developers.

This example shows that using obfuscation can be challenging, even for small apps. To help developers, this paper proposes an approach to tackle the challenges *i*,*ii*, *iv* and *v* to automatically generate the `keep` and `dontwarn` rules specific to an app and its libraries.

[4]Budget Watch on F-droid: https://f-droid.org/en/packages/protect.budgetwatch/

3 THE VISION

As mentioned previously, the current static analysis performed by Proguard is limited, in particular concerning reflection. The documentation goes so far as to state that "It is generally impossible to compute which classes have to be preserved (with their original names) " leading to "Obfuscating code that performs a lot of reflection may require trial and error, especially without the necessary information about the internals of the code" [10]. Obviously, it is possible to improve this static analysis to cover more cases, by including, for example, string analysis of potential reflection. However, not all cases may be covered, in particular because of the many possible sources of reflection outside the source code of the app (libraries, configuration files, database...) determined at runtime. This is all the more complicated in an ever-changing ecosystem like Android. Moreover, a complex static analysis could extend even more the compilation time when using obfuscation, discouraging developers even more.

This is why, rather than seeking to propose an omniscient static analysis, this paper exploits the wisdom of the crowd and machine learning to automatically generate rules. Indeed, developers who have successfully overcome the challenges of obfuscation provide us with a source of knowledge and experience on when and how to write rules adapted to an app.

The problem with `android.support.v7.widget.SearchView` of our example (2.2) was encountered several times by other developers. Therefore, if we have a large enough base of knowledge, it is reasonable to expect that a rule was written to solve this problem. So we should be able to automatically generate this rule for all future apps using this class. Such solution can be updated automatically as apps evolve.

Rules specific to an app are more difficult to generate, here some patterns, keywords, features or class roles should be identified in order to know if a class should be kept or not. The good news is that the precision of such approach does not have to be perfect, indeed if some classes are kept in excess this will only have a slight effect on the obfuscation results. However, the recall should be maximized, since missed rules might lead to bugs.

4 WHY IS IT NEW?

When Proguard (or R8) is enabled, developers need to write the rules adapted to their apps in specific files. Then using these rules and its static analyzer, the obfuscator determine which classes will be obfuscated or not before the conversion to Android bytecode.

We propose not to leave the writing of the rules to the sole responsibility of the developer. To do this, our approach will generate an additional rule file which will be processed by Proguard/R8 the same way as a traditional developer file. The novelty lies not only in the automatic generation of rules, but also in the fact that we generate these rules relying on a crowdsourced base of knowledge.

For example, by analyzing the libraries used by the current app, we can extract the related libraries rules from apps in the base of knowledge which are using the same library. Complex heuristics based on machine learning may predict rules for the classes of the app, by identifying recurring patterns, keywords or features in the base of knowledge. To the best of our knowledge, this is the first approach proposing to generate rules using knowledge gained from other apps.

5 EXAMPLE OF DATASET

To test the feasibility of our approach, we created a first dataset of open-source apps from the F-Droid repository [5]. We downloaded the latest version of the 2,038 apps available in July 2019[6]. Among these 2,038 apps, only 460 enabled an obfuscator.

We also removed from our dataset many apps which were abusing of wild cards in rules (e.g. `-keep class org.myapp.**`). Although this kind of rules are easy to write, they defeat the very purpose of obfuscation. Consequently, we have decided to ignore these rules so that our model does not learn from such bad practices. In total 352 apps were considered. We obtained around 1,200 classes kept by rules for our base of knowledge.

It is early to tell if this dataset is sufficient to cover most cases of rules, but it shows that it is possible to recover knowledge from developers of open-source apps.

6 THE RISKS

6.1 Tale of an inconclusive attempt

We tried to exploit our dataset as a ground truth to automatically generate app-specific rules using machine learning. To characterize a class we used as inputs the source code of our open-source apps and the rules attached to them, allowing us to label each class as kept or not. We extracted two types of features: the word-based features and metrics.

The word-based features were extracted by counting the occurrences of certain words in the source code like `android` or `string`, hoping that the usage of particular keywords might help us to determine if a class should be kept or not. The metrics used an abstract syntax tree to count more high-level entities like numbers of methods or reflections usage. From the keywords we selected a set of the 30 most relevant features.

Concerning the metrics, for a given class, the number import statements, fields, constructors and methods was counted as well as the number of times the name of this given class is referenced inside the classes that use reflection (imports `java.lang.reflect`) in the same application. They are used to give an insight about the complexity and role of a class. For example, if a class has lots of fields and few methods or if the only methods in a class are getters and setters, it is more likely that it is a data class which should not be obfuscated. Our problem is a binary classification on features from a 37-dimensional space (30 word-based features and 7 metrics). Each classes is labeled as kept or not according to Proguard rules and is processed to extract the features.

We compared the performance of three different models: SVM, neural network with one hidden layer, 500 neurons and random forest. A 10-fold crossvalidation shows that the random forest performed consistently better than the other models with an f1 score of 0.85. The Table 1 presents the precision, recall and f1 score of our model. The variance of the results is less than 0.001 in all cases, except for the recall of neural network (0.0018).

Although our results seemed to be good with an f1 score of 0.84, they are given for one class while developers may want to know if this approach may work for their whole app. Therefore, we intended two validations.

[5] F-droid apps repository: https://f-droid.org
[6] List of apps of our dataset https://pastebin.com/QZp3VH4i

First, we validated our approach on already obfuscated apps with at least one kept class, for a total of 68 apps. We tested our model on each app, for each app the model was trained on the whole dataset minus the app under validation. If we have no false negatives, then the model was able to successfully identify all the classes to keep and we consider that the rules are well-generated. Given that the class-wise recall is of 0.85, the probability for the model to determine all of the n classes to keep correctly is of 0.85^n. The model properly generate rules for 18 apps out of a total of 68 (29%). Concerning the apps in which the approach was imperfect, on average only 5 correct rules out of 21 were missing.

Second, we tried to validate our model on 50 random apps which were not already obfuscated. We decided to build them with obfuscation enabled and a command to ignore the warnings. We then only selected app which were crashing because of obfuscation to see if we could solve the bugs. The aim of this test was to see if obfuscation causes any crashes at runtime. We tested each app manually (login, main features, exploration of menus...) during a minimum of five minutes. Once the rules are generated, we apply them to the app source code, build the app and run the test again. If there is no crash, then our approach solved the problem related to obfuscation. Alternatively, if the app does not works properly, then the model was not able to provide a functioning set of rules. With this protocol we identified 15 crashing apps, unfortunately our approach was not able to fix any of these apps. Indeed, after further investigation, we discovered that most of the crashes were related to libraries which are not yet considered. These bugs could be due to missing libraries related to `keep` rules or could be due to the warning we automatically ignored (often thousands per apps). A more comprehensive approach including librairies `keep` rules and `dontwarn` rules is therefore needed to fix all crashes related to obfuscation.

	SVM	Neural Network	Random Forest
Precision	0.70	0.76	0.84
Recall	0.63	0.61	0.85
F1 score	0.67	0.68	0.84

Table 1: Model performance (mean)

6.2 Identified risks

Our prototype was designed to assess whether available rules could be exploited. However, a number of risks and threats were identified.

***i)* Open source apps might not be the most relevant apps to build a base of knowledge:** Our approach relies on open-source apps, which are not necessarily representative of all Android apps. Not only do open-source apps are generally small but in addition we cannot guarantee the quality of the rules used by these apps even if we try to filter them. Obfuscation is not a major concern for open-source developers, as we observe in our dataset. For now, it is the only way to get the source code and rules associated with them. We are currently investigating the possibility to infer some rules from the bytecode of store apps, which could allow us to build a better base of knowledge.

***ii)* Multiple library versions:** The generated rules should not only cover the last versions of libraries since apps may be compiled with older versions. In our first dataset, we observed different versions of the same libraries, therefore it is possible to build a base of knowledge covering this. However, some version might not be covered but this risk is mitigated since we observed that the needed rules rarely evolve between different version of the same library.

***iii)* Detecting reflection is hard:** Our metrics for reflection may be improved using static analysis. We do not think that one approach can accurately detect all possible cases for the reasons mentioned in section 3. However, we observed that some keywords and import (related to libraries) could help us to predict if a class is likely to be reflectively used. We therefore believe that the knowledge we could get from our dataset could work when static analysis is not suitable.

***iv)* Challenges to validate our approach:** Ideally our approach should assess whether and app does not crash at runtime due to the obfuscation. Although automatic solutions exist to test apps at runtime, they are often unable to get past a login screen or may miss some app functionalities. Therefore for the moment, we should rely on a time consuming manual test. This test does not guarantee a perfect coverage of the code, and therefore some crashes might been missed.

***v)* The recall should be close to perfection:** To ensure that this approach could convince developers to enable obfuscation in their apps, it is of the utmost importance that it is not seen as a source of bugs. Even if the approach does not introduce bug in itself (compare to a project with a basic proguard file), this could not be enough from the developer point of a view. Every missed rule is a potential bug. To achieve this, it is possible to lower precision while maximizing recall, but we do not know the order of magnitude yet.

***vi)* Explaining the generated rules is a challenge:** Ideally, developers should know why each rules has been generated for their apps. This might be challenging for a fully-automated approach which could be based on many criteria. Worse, the results could be even confusing for developers when the approach fails, since they will not necessarily be able to write the necessary extra rules if they do not understand the generated ones.

7 NEXT STEPS

This paper presents the challenges encountered by developers willing to use Android obfuscator's. It proposes the vision of a crowd-sourced approach to assist developers by automatically generating `keep` rules, an essential configuration step of obfuscators. Our approach may complement a traditional static analysis in order to alleviate Android developer's burden with obfuscators. We briefly presented the results of an inconclusive attempt with machine learning and the risks we identified from this experimentation. We plan to improve this approach for app rules, by improving the detection of reflection and recall. Using our dataset and first results, we also plan to implement heuristics to generate libraries related rules (`keep` and `dontwarn`). It should allow us to fix some of app crashes related to obfuscation we observed, and therefore generate successfully the rules of some apps. Finally, we are considering to infer rules from the bytecode of non open-source apps, which would allow us to expand our dataset and possibly increase its quality. We also hope that the community will propose other approaches and tools that will eventually convince developers to systematically use obfuscation.

Acknowledgements: This work is supported by Proyecto FONDECYT Postdoctorado N°3180561.

REFERENCES

[1] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in Android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 356–367.

[2] Vivek Balachandran, Darell JJ Tan, Vrizlynn LL Thing, et al. 2016. Control flow obfuscation for Android applications. *Computers & Security* 61 (2016), 72–93.

[3] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 175–186.

[4] 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. In *International Conference on Security and Privacy in Communication Systems*. Springer, 172–192.

[5] Felix C Freiling, Mykola Protsenko, and Yan Zhuang. 2014. An empirical evaluation of software obfuscation techniques applied to Android APKs. In *International Conference on Security and Privacy in Communication Networks*. Springer, 315–328.

[6] Google. 2019. Support Library. https://developer.android.com/topic/libraries/support-library. [Online; accessed 15 October-2019].

[7] Wojtek Kaliciński. 2018. Practical ProGuard rules examples. https://medium.com/androiddevelopers/practical-proguard-rules-examples-5640a3907dc9. [Online; accessed 15 Jan-2020].

[8] Aleksandrina Kovacheva. 2013. Efficient code obfuscation for Android. In *International Conference on Advances in Information Technology*. Springer, 104–119.

[9] Proguard. 2019. Proguard : Kotlin Beta. https://www.guardsquare.com/en/products/proguard/proguard-manual-kotlin-beta. [Online; accessed 15 Jan-2020].

[10] Proguard. 2019. Proguard Manual. https://www.guardsquare.com/en/products/proguard/manual/introduction. [Online; accessed 15 Jan-2020].

[11] Leo Sei. 2018. R8, the new code shrinker from Google, is available in Android studio 3.3 beta. https://android-developers.googleblog.com/2018/11/r8-new-code-shrinker-from-google-is.html. [Online; accessed 15 October-2019].

[12] Yan Wang and Atanas Rountev. 2017. Who changed you?: obfuscator identification for Android. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 154–164.

[13] 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 *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 222–235.

[14] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2016. Generalized dynamic opaque predicates: A new control flow obfuscation method. In *International Conference on Information Security*. Springer, 323–342.