Characterizing the Use of Code Obfuscation in Malicious and Benign Android Apps

Ulf Kargén
ulf.kargen@liu.se
Linköping University
Linköping, Sweden

Noah Mauthe
noah.mauthe@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Nahid Shahmehri
nahid.shahmehri@liu.se
Linköping University
Linköping, Sweden

ABSTRACT

Obfuscation is frequently used by both benign and malicious Android apps. Since static analysis of obfuscated apps often produces incomplete or misleading results, the problems of identifying and quantifying the use of specific obfuscation techniques in apps has received significant attention. Even though several existing works have addressed these problems, most studies focus on data obfuscation methods such as identifier renaming and string obfuscation, while more advanced code obfuscation methods, such as reflection and control-flow obfuscation, have received less attention. Moreover, existing approaches to detecting Android code obfuscation have significant limitations, as shown by a detailed survey that we present as part of this paper. This is in part due to a fundamental "bootstrapping" problem: since, on one hand, the landscape of Android code obfuscation is poorly known, researchers have very little guidance when designing new detection methods. On the other hand, the lack of detection methods mean that the obfuscation landscape is bound to remain largely unexplored.

In this work, we aim to take the first steps towards addressing this "bootstrapping" problem. To this end, we propose two novel approaches to obfuscation detection and perform a study on over 200,000 malicious apps, in addition to 13,436 apps from Google Play. In particular, we propose a new anomaly-detection-based method for identifying likely control-flow obfuscation, and use it to perform what is, to the best of our knowledge, the first empirical study of control-flow obfuscation in Android apps. In addition to presenting new insights into the use of control-flow obfuscation, we also propose a new approach to characterizing the use of reflection-based obfuscation, which allows us to corroborate earlier findings indicating that this type of obfuscation is much more common in malware than in benign apps.

CCS CONCEPTS

• **Security and privacy** → *Malware and its mitigation*; *Software reverse engineering*; *Software security engineering*; • **Computing methodologies** → *Anomaly detection*; • **General and reference** → **Empirical studies**.

ARES 2023, August 29–September 01, 2023, Benevento, Italy

ACM ISBN 979-8-4007-0772-8/23/08.
https://doi.org/10.1145/3600160.3600194

KEYWORDS

Android, code obfuscation, obfuscation detection, empirical study, graph anomaly detection, malware

ACM Reference Format:
Ulf Kargén, Noah Mauthe, and Nahid Shahmehri. 2023. Characterizing the Use of Code Obfuscation in Malicious and Benign Android Apps. In *The 18th International Conference on Availability, Reliability and Security (ARES 2023), August 29–September 01, 2023, Benevento, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3600160.3600194

1 INTRODUCTION

Recent studies [8, 10] have shown that obfuscation is frequently used by both benign and malicious Android apps. In the former it is used to prevent intellectual property theft and ad-fraud, while in the latter it is used as a means to evade detection by antivirus systems, and to delay analysis of malicious behavior. Static analysis of obfuscated apps often produces incomplete or misleading results. For example, *string encryption* can be used by malware to hide malicious URLs from antivirus scanners, while *reflection* can be used to hide calls to sensitive APIs, or to disrupt static security analysis methods by hiding certain information flows [13]. Similarly, *control-flow obfuscation* can impede static information-flow analysis by hiding the original control-flow structure of an app.

For the above reasons, both the problem of determining whether or not an app uses a specific obfuscation technique, as well as the problem of quantifying the prevalence of different obfuscation techniques, has received significant attention in the Android security community. While several works have addressed the aforementioned problems, most studies to date focus on *data obfuscation* methods such as *identifier renaming* and string encryption, which primarily serve to impede *manual* analysis by hiding semantic clues about program behavior from reverse-engineers. All implementations of both identifier renaming and string encryption share a common trait, namely that they work by scrambling a sequence of characters (an identifier or a string literal). Therefore, these techniques can be detected with reasonable generality by identifying deviations from an expected character-frequency distribution [8, 20, 22]. More advanced *code obfuscation* methods, such as reflection and control-flow obfuscation, exhibit a much greater diversity in terms of possible implementation approaches. Therefore, designing a "universal" detection method for either class of obfuscation technique is very challenging. Furthermore, any attempt to create such a detector is faced with a significant "bootstrapping" problem: on the one hand, since the landscape of Android code obfuscation techniques is not well-known, it is hard to know which app features to focus on, and even harder to estimate a detector's ability

to generalize (since the degree of representativeness of any ground truth cannot be known). On the other hand, the lack of reliable detection methods means that the obfuscation landscape is bound to *remain* largely unexplored. Due to these challenges, the existing literature on Android code obfuscation detection is relatively small and tends to be limited in scope. Moreover, existing detection techniques tend to either be overly simplistic or fail to generalize. For example, in previous work [8, 12, 25], reflection-based obfuscation is simply detected based on the presence of calls to the Java reflection API. During our empirical study, we found this approach to be inadequate, since nowadays the vast majority of apps appear to make use of reflection. Moreover, in the few previous works where control-flow obfuscation is considered, detection is based on machine learning models that are trained and evaluated on (at most) a handful of implementations of a set of specific obfuscation techniques [4, 11, 20, 29]. Such approaches are likely to generalize poorly to in-the-wild apps.

In this work, we aim to take the first steps towards addressing the aforementioned "bootstrapping" problem, by performing an exploratory study of in-the-wild use of code obfuscation on Android. To support our study, we have developed two new methods for triaging Android apps for the presence of, respectively, reflection-based obfuscation and control-flow obfuscation. In contrast to earlier work, we have strived to make minimal assumptions about the nature of code obfuscation techniques when designing our methods. Our exploratory study comprises over 200,000 malicious apps from the AndroZoo [3] dataset, as well as 13,436 apps crawled from Google Play. As part of the study, we have performed a detailed manual analysis of over 400 apps, with the dual purpose of validating the effectiveness of our triaging methods, and to gain deeper insights into Android code-obfuscation use in the wild.

1.1 Identifying Code Obfuscation Use

Since there are several use cases for reflection in Android apps, detecting reflection-based obfuscation by simply checking for use of the reflection API is insufficient. In contrast to earlier work, our method instead focuses on the way reflective method calls are made. We argue that reflective calls that are truly meant to obfuscate the call target are less likely to provide the class or method name arguments as constant string literals, since this would still allow automated static recovery of the target. Instead, if a reflective call is used with the intent of obfuscating the code, it is more likely that the target name is dynamically computed during runtime (for example, as the return value of another call). We henceforth refer to this case as *dynamic reflection*. By making a distinction between dynamic reflection and reflection lookups using literal arguments, we argue that our empirical results are more likely to give a better indication of actual reflection-based obfuscation use.

For empirically investigating the prevalence of control-flow obfuscation, we note that existing detection approaches are, on a fundamental level, ill-suited for the task. As existing works ([4, 11, 20, 29]) rely on machine learning models that are trained to recognize specific tools or techniques, they make the implicit assumption that these tools or techniques are representative of all (or most) types of control-flow obfuscation encountered in the wild. The authors, however, make no attempt to justify this assumption. We argue that asserting such an assumption to hold true *a priori* would in fact be virtually impossible, given that control-flow obfuscation is such a broad category of techniques. Instead, we propose an approach based on *anomaly detection*. We train an anomaly detector on a set of non-obfuscated apps, and then use that model to detect deviations from "normal" control-flow structure.

1.2 Contributions

In summary, the main contributions of this paper are as follows:

- We perform an in-depth survey of existing efforts to detect and empirically study code obfuscation use in Android apps, and summarize their findings.
- We perform a large-scale study to characterize the use of code obfuscation in both benign and malicious Android apps.
- To support our study, we propose two new approaches to identify, respectively, reflection-use for obfuscation, and control-flow obfuscation[1].

2 BACKGROUND

In this section, we provide some background on the Android system and common methods for obfuscating Android apps.

2.1 Android Runtime Model

Android apps are typically developed using Java or Kotlin, and compiled to bytecode for the Android-specific Dalvik virtual machine. Apps ship as Android Application Package (APK) files, which are compressed ZIP archives containing one or more files of the Dalvik Executable (DEX) format, in addition to various other assets. A DEX file contains several classes, including bytecode for all the methods of each class. Older Android versions used an interpreter to execute Dalvik bytecode, but starting from version 5, Android uses the so-called Android Runtime (ART) system, which instead pre-compiles bytecode to native code during app installation. Android apps can also directly invoke native code using the Java Native Interface (JNI). In this case, native code is distributed with the app as shared library files in ELF format, stored within the APK.

2.2 Android Obfuscation Techniques

Identifier renaming is a very commonly seen type of obfuscation in Android apps, owing largely to the fact that the Android SDK ships with a free tool to apply the obfuscation technique. By using identifier renaming, human-readable identifiers for, e.g., methods, classes or variables, are replaced by short meaningless strings. The technique mostly aims to frustrate manual analysis by hiding semantic clues about program behavior, and by making it harder to remember identifier names. It is also sometimes used for non-obfuscation purposes, since it tends to reduce the size of the final APK.

String encryption is a more advanced method provided by various commercial Android obfuscators. It works by replacing constant strings in an app with encrypted versions. Decryption logic is then injected into to the program at the points where the original strings were used, so that the strings can be decrypted on demand during runtime.

[1]We make our implementation available at https://github.com/NoahMauthe/ObA

Reflection is sometimes used for obfuscation as well. The Java reflection API allows programs to dynamically resolve class and method names during runtime, which can be used to hide the target of method calls from static analysis tools. However, there are many apps that use reflection for purposes other than obfuscation. The ability to test for the availability of a certain class or method during runtime can be used to ensure backwards compatibility with, e.g., legacy APIs. Reflection can also be used to call methods that are declared as private or protected, which can be used by apps for accessing "hidden" parts of, for example, the Android API.

Dynamic code loading is another way to hinder static analysis. Parts of the app's code are stored in encrypted form, and are decrypted on demand during runtime using a key that is either hidden somewhere in the app, or retrieved from a remote server. Android provides the class `DexClassLoader` (and several derived classes) to dynamically load code from a DEX or JAR (Java Archive) file. Methods from the loaded file can then be invoked using reflection. This can be used for obfuscation by, for example, shipping an encrypted DEX or JAR file within the APK, or retrieving such a file from the internet. Another approach, sometimes denoted as *class encryption*, is to store an encrypted class as a static byte array within another class.

Packing is a more sophisticated version of dynamic code loading, where the entire code of the app is stored in encrypted form, and the original DEX file(s) of the app are replaced by a decryption stub. Modern commercial packers typically implement the decryption logic in native code, and only unpack the original bytecode in a piece-wise manner as it is needed [32].

Control-flow obfuscation is a broad category of techniques, which aim to disrupt automated static analysis or delay manual reverse engineering. Examples of simple techniques that fall within this category are dead code insertion or the addition of "fake" branches protected by *opaque predicates* — logical expressions that are invariant during runtime but hard to reason about statically. *Control-flow flattening* [27] is a more advanced obfuscation technique, whereby every basic block of a method is moved into the body of a dispatch loop (for example, in the form of `case` blocks of a large `switch` statement). At the end of each block, a variable is updated, which controls what block to be dispatched next. This way, all static information about the control-flow structure of the method is removed.

Control-flow obfuscation is harder to implement on Android compared to native code or Java programs. This is because the ART will verify during app installation that there are no data type conflicts between register accesses along any path in the control flow graph (i.e., that no registers could ever be written as one type and read as another). However, Balachandran et al. [5] have proposed an implementation of control-flow flattening for Android that does not suffer from this problem. To further obfuscate the control flow, their method additionally makes use of exceptions to transfer execution in statically hard-to-predict ways.

3 PREVIOUS WORK

In this section we survey previous work on detecting and empirically quantifying obfuscation, with a particular emphasis on the two types of anti-reverse-engineering techniques that is the focus of our work (reflection and control-flow obfuscation).

3.1 Empirical Studies of Mobile App Obfuscation

Kühnel et al. [12] describe a system for detecting identifier renaming, reflection, and encryption (where they make no distinction between string and code encryption). Identifier renaming is detected heuristically, while reflection is detected simply based on the presence of calls to the Java reflection API. Encryption is detected either based on the use of known crypto APIs, or, for custom crypto implementations, based on the frequency of certain types of instructions. Their evaluation on a set of malware retrieved from VirusTota[2]l indicated that, out of the malware samples submitted to VirusTotal during 2014, around 30% used reflection.

Li et al. [13] proposed a constant-propagation method to statically resolve reflective calls in Android apps. They also performed a small-scale empirical investigation of 500 apps from Google Play, and found that 88% used the reflection API.

Qu et al. [25] developed DyDroid, which uses both static and dynamic analysis to detect dynamic code loading in Android apps. Moreover, it is capable of detecting identifier renaming, reflection, and anti-decompilation tricks. Like Kühnel et al., they simply detect the presence of calls to the reflection API. They used their tool to evaluate almost 60,000 apps collected from Google Play in 2016, and found that only 0.24% used dynamic loading of encrypted code for obfuscation, whereas 52% used reflection.

Wang et al. [28] studied the prevalence of obfuscation in iOS apps. They first filtered apps based on the presence of identifier remaining, and then manually analyzed apps that had scrambled identifier names to detect the use of other obfuscation techniques.

A large-scale empirical study on the use of identifier renaming in Google Play apps was performed by Wermke et al. [30].

Another large-scale study of obfuscation use in Android apps was performed by Dong et al. [8] in 2016 and 2017. They scanned over 100,000 apps from three distinct datasets (Google Play, third-party market apps, and malware) for the presence of identifier renaming, string encryption, and reflection. The former two techniques were detected using machine learning (ML), whereas reflection, like in previously mentioned works, was detected by the presence of related API calls. Similar to the results by Qu et al., they found that around 50% of apps used reflection, with no notable difference between benign and malicious apps. Furthermore, in a somewhat similar vein to Li et al. [13], Dong et al. performed intraprocedural static slicing to "reassemble" string arguments to reflection API calls (i.e., method and class names) that were constructed dynamically by the app. The reassembled targets of reflected calls were analyzed to find the most frequently invoked methods in each dataset. This indicated that benign apps typically appeared to use reflection for backward compatibility (checking for the presence of certain APIs), and to call hidden API methods. Malware, on the other hand, appeared to more frequently employ reflection in order to make the code harder to understand.

The study by Dong et al. indicated that string encryption was relatively rare, and almost exclusively used by malware. A more

[2]https://www.virustotal.com/

recent study by Glanz et al. [10], employing a more comprehensive detection method, instead indicates very high adoption rates among both benign and malicious apps.

Mauthe et al. [19] studied the decompilation failure rates (i.e., the percentage of methods that fail to decompile in an app) of four decompilers on both benign and malicious Android apps. While failure rates were generally low, they were about an order of magnitude greater for malicious apps. Ad-supported apps from Google Play were also found to have more decompilation failures than non-ad-supported ones. The authors theorized that this was due to higher adoption rates of code obfuscation in malware and commercial apps. A preliminary manual investigation also showed that several of the decompilation failures in Google Play apps were indeed due to control-flow obfuscation. However, for most apps (both benign and malicious), failures appeared to be induced by high code complexity rather than obfuscation. The authors therefore proposed that the decompilation failures on obfuscated code were a side-effect of increased complexity, rather than the intended purpose of applying obfuscation.

3.2 App Obfuscation Detection

In addition to the detection methods proposed in the aforementioned empirical studies, several works propose dedicated methods for detecting specific types of obfuscation.

APKiD [1], is a well-known signature-based detector, which can detect, among other things, packing and obfuscation, by scanning for signatures of specific obfuscation tools. An obvious limitation of signature-based detection is that custom obfuscation schemes cannot be detected.

Several works also propose machine-learning-based approaches for obfuscation detection. Wang and Rountev [29] proposed an ML based technique to learn signatures of specific obfuscators and packers, based on the presence of certain strings in an app. Similarly, Kaur et al. [11] proposed a fingerprinting approach based on visual representations of Android APKs, for the same purpose.

Bacci et al. [4] applied several different ML algorithms for the problem of identifying obfuscation in Android apps. They considered identifier renaming, string encryption, and several code obfuscation methods, and found that these techniques could be detected with an accuracy of 80–100%. However, since their training and evaluation data was generated using their own implementations of various code obfuscation techniques, it is unclear to what degree (if at all) their approach generalizes to in-the-wild apps.

Another ML based approach was proposed by Mirzaei at al. [20]. Their system AndrODet uses online learning for the detection of identifier renaming, string encryption, and control-flow obfuscation. A comment paper by Mohammadinodooshan et al. [21], however, pointed out that the datasets used for training and evaluating string encryption detection is biased, and demonstrated that the accuracy of AndrODet drops dramatically if non-biased datasets are used. AndrODet's control-flow obfuscation detection also has several limitations. Firstly, it has a limited accuracy of around 70%, and secondly, the fact that only a single obfuscation tool (Allatori) was used to generate training and evaluation data might reduce its capacity to generalize. Moreover, the construction of the training and evaluation data sets for control-flow obfuscation detection potentially suffer from the same bias as described in the aforementioned comment paper.

Mohammadinodooshan et al. proposed an alternative ML approach based on anomaly detection for identifying obfuscated strings in apps [22].

3.3 Summary of Previous Work

To summarize, to the best of our knowledge, all previous efforts to quantify the use of reflection-based obfuscation rely on simply checking for usage of the reflection API. Such approaches cannot distinguish between legitimate uses of reflection, and reflection for obfuscation purposes. As for detection of control-flow obfuscation, three approaches can be identified: (1) detection based on manually-crafted obfuscator signatures (as used by, e.g., APKiD), (2) using ML for automatically learning such signatures ([11, 29]), and (3) attempting to train an ML classifier to detect general signs of control-flow obfuscation ([4, 20]). Both the works in the latter category, however, use training data based on single implementations of a small set of obfuscation transformations, making it unlikely that the detectors will generalize well.

Moreover, as a general observation, it is also evident that empirical results can vary significantly, depending on the approach used for compiling datasets and the techniques used for detection.

4 APPROACH

In this section we describe the methodology of our study. We begin by giving a brief description of the datasets used, and then outline the design and implementation of our analysis framework, including our proposed new methods for identifying likely use of reflection-based obfuscation and control-flow obfuscation.

4.1 Datasets

We used two primary datasets in the study. The first one consists of 13,436 apps crawled from Google Play in 2020. The dataset was compiled by downloading all apps in the "most popular" and "top-grossing" subcategories of the 34 app categories present in Google Play at the time, with the intention of focusing on apps with the greatest degree of end-user exposure. (Paid apps were excluded.) We divided the Google Play apps into two categories, based on whether they appeared to be commercial (i.e., for-profit) or not. Apps were categorized as "commercial" if they either used ads (according to Google Play metadata), or if they requested permission for in-app payments (the "`com.android.vending.BILLING`" permission). This yielded a split with 9,725 "commercial" apps and 3,711 "non-commercial" ones, referred to henceforth as the GPlay-C and GPlay-NC datasets, respectively.

The second dataset consists of malware from the AndroZoo dataset [3], which is a large and continuously updated collection of Android apps. Each app was scanned at Virus Total (VT) at the time it was added to the dataset, which allows us to filter apps based on the total number of antivirus products that flagged it as malicious. We choose a threshold of at least 20 detections for considering an app as malicious, corresponding to roughly one third of all scanners used by VT. Randomly sampling apps based on this threshold yielded 236,888 samples, added to AndroZoo between

2012 and 2021 (according to the `vt_scan_date` entry in the listing of app metadata). (See Appendix A for the exact yearly distribution.)

The AndroZoo dataset contains 10,186 "synthetic" malware samples from the PRAGuard dataset [18], which were created by running a commercial obfuscator on a set of malicious apps. Since we are interested in characterizing obfuscation use of in-the-wild malware, we excluded all PRAGuard apps from sampling.

Finally, we additionally make use of 3,037 open-source apps retrieved from the F-Droid[3] repository in 2020.

4.2 System Design

Our system is implemented using the Python-based Android app analysis platform Androguard[4], which can extract the bytecode of each method present in an APK, and perform various types of analysis on it. Additionally, it implements its own decompiler for reconstructing Java source code or the corresponding abstract syntax tree (AST) from the Dalvik bytecode of a method.

An overview of our system design is shown in Figure 1. The primary contributions of our work are highlighted in boldface. We use Androguard to extract a bytecode-level control-flow graph (CFG), which is used by our system to detect potentially control-flow obfuscated methods. A random sample of suspect methods is then analyzed manually, in order to draw conclusions about the use of control-flow obfuscation in different types of apps. There are two main reasons why we operate on the bytecode-level CFG, rather than using decompiled Java code: firstly, it allows a canonical representation of a method's control-flow, which is independent of any particular decompiler's interpretation of the code. Secondly, it allows us to handle CFGs that are non-reducible or otherwise hard to represent with higher-level code constructs.

In contrast, when detecting dynamic reflection, it is necessary to lift the Dalvik bytecode to a higher abstraction level, since we need to reconstruct the arguments to API calls. Therefore, we make use of Androguard's built-in decompiler, and its feature to allow user-defined analysis directly on intermediate ASTs.

Below, we describe the design of our detectors for control-flow obfuscation and reflection-based obfuscation in detail.

4.2.1 Control-Flow Obfuscation. Detectors for control-flow obfuscation that rely on signatures are only able to detect obfuscation applied by specific tools. Methods that rely on *discriminative* ML classifiers have a similar limitation: such classifiers are trained on examples from two classes (obfuscated and non-obfuscated), and learn to recognize differences between the two. However, presented with a type of obfuscation that was not in the original training data, this type of classifier is unlikely to perform well. To make matters worse, due to the "bootstrapping" problem that we discuss in Section 1, feature engineering, hyperparameter tuning, and model evaluation is essentially impossible to carry out in a reliable way, since the generality of any validation or testing data cannot be estimated. In this work, we instead aim to use ML in an *exploratory* fashion, allowing us to take the first steps towards characterizing the Android control-flow obfuscation landscape.

Based on the aforementioned challenges, the following requirements for an ML detector can be identified:

- **R1 Scalability.** Since we need to analyze millions of methods, scalability is an important baseline requirement.
- **R2 No labeled data.** Since no unbiased labeled datasets are available, the approach cannot rely on labeled datasets.
- **R3 No hyperparamter tuning.** Most ML algorithms have several tunable parameters, which must be determined empirically, typically using cross-validation on a validation dataset. Since, in our setting, no (unbiased) validation datasets are available, we cannot rely on hyperparameter tuning *at all*. In other words, it must be possible to assign reasonable values to all hyperparameters based on a clear intuition, rather than using cross-validation. This is a very strong requirement, which rules out most "off-the-shelf" approaches.

Due to **R2**, we have chosen to base our approach on anomaly detection, which is an instance of so-called *one-class classification*. Instead of learning to recognize differences between classes, an anomaly detector learns a model of the distribution of one class (the "normal" class). The model can then be used to identify outliers by measuring how much a specific sample deviates from the learned distribution. In our setting, the detector will identify methods that exhibit control-flow patterns that deviate substantially from typical code. While the use of anomaly detection has the benefit of not being biased towards specific techniques or implementations, it is important to note that the lack of validation data means that it is not possible to gauge the recall of our model in the general case. While our manual analysis (Sections 4.2.2 and 5.2) allows us to investigate the precision (or false positive rate), it is quite likely that the detector is incapable of detecting certain types of control-flow obfuscation (i.e., it likely exhibits false negatives). In this study, we have chosen to acknowledge this limitation, which fundamentally stems from the aforementioned "bootstrapping" problem. Instead, we use the detector in an exploratory fashion to perform the first empirical study of in-the-wild use of control-flow obfuscation, in the hope that this could pave the way for future investigations into the subject. In the following, we outline the design of our control-flow anomaly detector.

Like most machine learning methods, classical anomaly detection algorithms expect a set of fixed-size feature vectors as input. For our use case, however, anomaly detection needs to be applied to the CFGs of methods in an app. While a relatively large body of work on techniques for applying anomaly detection on graphs exist (see for example the survey by Akoglu et al. [2]), most existing methods focus on identifying anomalies *within* a graph, e.g., anomalous nodes or edges. In comparison, relatively few works have addressed the problem of identifying anomalous samples within a *set* of graphs.

Classical methods for computing pairwise graph similarity, such as graph edit distance [6] or graph kernels [26], do not scale to millions of graphs, and can therefore be ruled out due to **R1**. Instead, it is necessary to employ a so-called *graph embedding*, which transforms a graph to an approximate vector representation, so that a classical ML algorithm can be used. The vector representation is constructed in such a way that similar graphs tend to be located close to each other in the vector space. Many recent works (for example, [9, 14, 31, 33]) have shown neural-network-based graph embeddings to be highly effective for estimating code similarity. Such embeddings are created by training a neural network on a

[3] https://f-droid.org/
[4] https://androguard.readthedocs.io/en/latest/

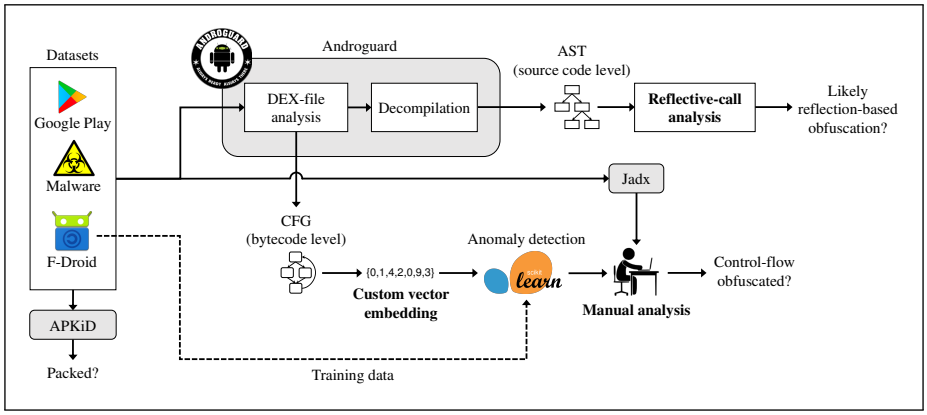


Figure 1: Overview of our system design.

(large) set of representative samples. After training, a graph can be fed into the network, yielding the equivalent vector representation. However, in our quite unique setting, this requirement for pre-training constitutes a significant disadvantage for mainly two reasons. Firstly, existing neural-network-based embedding techniques were not originally designed with anomaly detection in mind. As pointed out by, e.g., Ma et al. [17], this could lead to poor detection performance. For example, if the neural network is trained on "normal" graphs, the resulting embeddings might fail to capture precisely those properties that are important for identifying anomalous graphs. Secondly, many neural-network-based methods require extensive hyperparameter tuning to perform well for a particular task, violating R3. Consequently, we have chosen to design our own custom graph embedding, which we describe next.

Graph embedding. Our vector-embedding approach works by performing a fixed-depth traversal from each node in the CFG. We exhaustively traverse down every path of (at most) n hops, and record the frequency of every encountered node subsequence of length $1..n$. (That is, the frequency of every subsequence of length k for every $k \in \{1..n\}$.) We denote such a node subsequence as a *control-flow n-gram*. Feature vectors consist of n-gram frequencies normalized on the CFG node count, so that each possible n-gram has a corresponding dimension in feature space. An important property of this approach is that it can capture both local and global characteristics of a graph. Specifically, sequences of control-transfer operations (n-grams) capture local context, whereas the total n-gram counts capture information about global context.

The node attributes that make up our n-grams reflect the different types of possible control-transfer operations between basic blocks in a CFG. We define 7 ways in which the flow of execution can be transferred from one basic block to another:

(1) **Conditional branch** (`if-`*test* instructions)
(2) **Unconditional branch** (`goto` instructions)
(3) **Switch** (`packed-switch` or `sparse-switch` instructions)
(4) **Fall-through** to next basic block
(5) **Throw** (explicitly raised exceptions using `throw`)
(6) **Exception** (implicitly raised exceptions)
(7) **Return** from method

In addition to different kinds of branch operations, we also included control-transfers that happen due to exceptions, since some advanced obfuscation schemes use exceptions to obscure a method's control-flow structure (see Section 2.2). An illustrative toy example of applying our embedding technique can be found in Appendix B.

The only parameter that needs to be tuned with our approach is the maximum size n of n-grams, where larger n gives more local context. If we want to capture context both before and after each node, n should intuitively be at least 3. On the other hand, since feature vectors contain one element for each possible n-gram, the dimensionality of feature vectors is exponential in n. As a trade-off, we opted to use $n = 5$, which yields 10,885 unique n-grams, after pruning n-grams with **Return** nodes that are not the last element of the sequence (and therefore cannot exist).

Anomaly detection. Due to R3, we have chosen to use *Isolation forest* [16] for anomaly detection, since it, in contrast to most other anomaly detection algorithms, does not require hyperparameter tuning. Similar to algorithms like, e.g., *Random forest*, Isolation forest trains an ensemble of decision trees, with the important difference that the split at each node of a tree is chosen completely *at random* during training The algorithm continues to expand the tree until every sample is *isolated* into its own leaf node. The gist of the algorithm is that, when applying such random decision trees to new samples, the traversal tends to terminate at leafs at a *lower depth* for anomalous samples. The average termination depth across the ensemble can therefore be used to compute an anomaly score, reported as a real value between -1.0 and 1.0, where lower means more anomalous. The anomaly threshold is often determined based on the training data, as a percentile of the training sample scores. In addition to requiring essentially no hyperparameter tuning, Isolation forest has two other desirable features for our use-case: being an ensemble-of-trees method, it can capture non-linear relationships between features, and, more importantly, it can effortlessly handle very high-dimensional data, which is not the case for many other anomaly detection algorithms.

Training data was gathered by using Androguard to extract bytecode-level CFGs for every method in the 3,037 F-Droid apps. As developers of open-source apps have little incentive to obfuscate their code, we expect control-flow obfuscation to be very rarely

encountered in this dataset[5]. Since it is not meaningful to apply control flow obfuscation to methods with trivial control flow, we considered only methods with a bytecode size of at least 600 B and at least 30 nodes in the CFG, both during training and classification. Pruning F-Droid methods below these thresholds yielded a final selection of 182,380 methods, which we used for training. We used the Isolation forest implementation in scikit-learn [23]. The ensemble size was set [6] to 300 and trees were grown to their full depth.

During analysis of the datasets, we ran the anomaly detector on each method that met the aforementioned minimum-size requirement. In order to avoid excessive disk usage during data collection, we only recorded the anomaly score for methods with a score below -0.30, corresponding roughly to the lower (i.e., most anomalous) 5th percentile of anomaly scores in the F-Droid training data.

4.2.2 Manual analysis of control-flow obfuscation. For our manual analysis, we randomly sampled 200 apps from each of the GPlay-C, GPlay-NC, and malware sets. Apps that had no methods with an anomaly score below the threshold were removed from the three samples, and the most anomalous method in each of the remaining apps where analyzed manually using the decompiled source code. We choose the decompiler Jadx[7] for the analysis, as it was shown to be the most effective decompiler in the study by Mauthe et al. [19].

4.2.3 Reflection-Based Obfuscation. During the initial stages of our study, we found that 99.5% of the Google Play apps contain at least one reflective method lookup. (The corresponding figures for the malware and open-source apps where 84.7% and 74.8%, respectively.) Therefore, the approach used in earlier works of simply checking for use of the Java reflection API is clearly inadequate for modern apps. Instead, it is necessary to differentiate between typical use-cases for reflection, as opposed to reflection-based obfuscation. Using scriptable analysis environments like Androguard, it is trivial to extract the names of reflectively called methods, if the name is provided as a string literal when invoking the reflection API. Moreover, it is also often straightforward to reconstruct the fully qualified name, using an approach similar to Li et al. [13] or Dong et al. [8] (see Section 3.1). For this reason, we argue that developers who truly intend to use reflection as a means of obfuscation are more likely to refrain from exposing class and method names as constant string literals. While there are many legitimate reasons why an app might use dynamic reflection, our approach is based on the hypothesis that apps where an abnormally large proportion of reflective method lookups use non-literal arguments are strong suspects for reflection-based obfuscation.

To detect dynamic reflection, we make use of Androguard's built-in decompiler to generate ASTs for each method, and traverse ASTs to detect calls to the methods `getMethod` and `getDeclaredMethod` in the `Class` class. These methods are used to reflectively resolve public and non-public methods, respectively. Using the AST, we can determine if the name arguments to these methods are provided as literals, or, for example, as the return value from another method. Note that Androguard performs constant propagation. This allows us to detect cases where a string literal is assigned to a local variable, which is later used (unmodified) in a call to the reflection API.

Table 1: App distribution based on the number of reflective method lookups.

Lookups	Malware	F-Droid	GPlay-C	GPlay-NC
0	45,302	770	10	52
1–10	112,516	399	104	116
10–100	65,391	1,735	1,173	1,017
100–1000	12,437	133	8,428	2,510
≥1000	1,242	0	10	16

An abnormality threshold for the ratio of non-literal method lookups can be decided by using the F-Droid apps as a baseline for non-obfuscated apps, similar to the way we use F-Droid as a baseline for control-flow anomaly detection. Concretely, we (admittedly, somewhat arbitrarily) define the threshold as the 99th percentile of this ratio for F-Droid apps.

In order to derive a rough estimate of the precision of our detection approach (in the absence of actual ground truth data), we also performed a manual analysis of a random subset of apps that ended up above the threshold in each dataset. Apps where classified as true positives if they appeared to use dynamic reflection for no other apparent reason than to impede static analysis, whereas apps that appeared to have a legitimate reason to use dynamic reflection was classified as false positives. A detailed breakdown of the results are presented in Section 5.1. It should be noted that, similar to the control-flow anomaly detector, we cannot estimate the recall of our dynamic-reflection obfuscation detector. It is, for example, possible that our 99th percentile threshold is too high, causing us to miss some obfuscated apps. Moreover, apps that use naive non-dynamic reflection-based obfuscation cannot be detected with our approach, although from a practical point of view, such obfuscation should be less of a concern for advanced static analysis tools, as explained above.

5 RESULTS

In this section, we present the results of our empirical evaluation

5.1 Reflection-based obfuscation

As explained in Section 4.2, we base our analysis for detecting likely use of reflection-based obfuscation on the ratio of reflective method lookups that use non-literal method names. As the first step of our exploratory analysis, we studied the distribution of this ratio for different types of apps. However, one factor that makes comparing the ratios for different datasets more complicated is that the total number of reflective method lookups differ significantly between the datasets, as can be seen from Table 1. For example, around 20% of both the malware and F-Droid apps contain no calls to `getMethod` or `getDeclaredMethod`, whereas almost all the Google Play apps have at least one such call. To reduce the risk of drawing incorrect conclusions due to this imbalance, we bin all apps based on the order of magnitude of reflective method lookups in the app (the

[5] A small amount of "pollution" in the training dataset does typically not adversely affect the performance of Isolation forest [16].
[6] We opted to increase the ensemble size compared the default 100 trees, since our model is very complex with many features. Using more trees will never hurt the accuracy of Isolation forest, but comes at the cost of increased computation time.
[7] https://github.com/skylot/jadx

same way as in Table 1), and analyze each bin separately. Figure 2 shows violin plots of the per-bin distributions for each dataset. Since it is difficult to draw any conclusions for apps with very few lookups, we omit the "1–10" bin. Clearly, the malware dataset stands out in comparison to the others for both the "10–100" and "100–1000" bins, with a significantly larger number of apps with high ratios. For example, in the "100–1000" bin, around 40% of the malware apps have ratios higher than 0.3, while the corresponding figures for the F-Droid and Google Play apps are 0% and 0.9%, respectively. Interestingly, for the (very small) bin with a thousand or more lookups, the malware apps have very low ratios, while the handful of Google Play apps (26 ones) that end up in this bin have very high ratios. When invoked on these Google Play apps, APKiD found signatures of commercial obfuscators (Specifically, Arxan, DexGuard and Gemalto) for all apps except one. It is possible that these apps use class encryption, which would require reflective invocation of methods in encrypted classes. DexGuard, for example, is known to implement this kind of obfuscation [18].

Figure 3 shows the distributions for all apps with at least 10 reflective method lookups. The same pattern seen in the binned case can be seen here for the malware apps.

The horizontal dashed line in Figure 3 denotes the 99th percentile for F-Droid apps, which we define as the abnormality threshold (see Section 4.2.3). 16,107 malware apps, corresponding to 20.4% of malware with at least 10 lookups, or 6.8% of all malware, end up above the threshold. The corresponding numbers for the Google Play and F-Droid apps are 170 and 19 apps, respectively.

We also scanned the apps with APKiD 2.1.2, in order to compare our findings with a state-of-the-art signature-based obfuscator detector. Running APKiD on the 170 Google Play apps revealed that 77 (45%) matched signatures for commercial obfuscators or packers, compared to around 2.5% of Google Play apps overall. This indicates that much of the (suspected) reflection-based obfuscation in Google Play apps is due to the use of third-party obfuscation tools. In comparison, only around 12% of malware above the reflection threshold match a corresponding APKiD signature, while 6.3% of all malware apps match such a signature, indicating a weaker correlation. This, in turn, would indicate that malware authors more frequently apply reflection-based obfuscation by hand or using custom tools.

Manual analysis. The above results are in line with the qualitative analysis by Dong et al. [8], which indicated that reflection-based obfuscation was significantly more common in malware. However, as explained in Section 4.2.3, in order to investigate the validity of our approach, and to gain further insights into the use of reflection-based obfuscation in Android, we also performed a manual analysis of a random sample of all apps that ended up above the threshold (i.e., above the F-Droid 99th percentile) within each dataset. 35 apps were sampled from each dataset. In addition, we also analyzed the 19 apps from F-Droid that constituted the 99th percentile. Here, we considered an app as using reflection-based obfuscation if it applied transformations to method-name arguments that appeared to serve no other purpose than to prevent static resolution of reflective call targets. As expected, none of the F-Droid apps showed signs of this kind of obfuscation. However, for the other datasets, we found evidence of obfuscated call targets in a large proportion of the apps that were analyzed. For the Google Play dataset, we observed

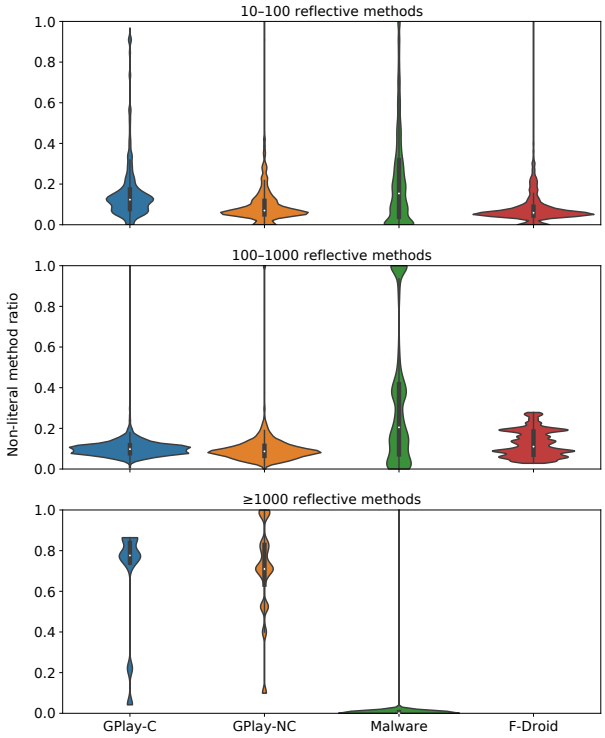


Figure 2: Distributions of non-literal method lookups binned on the number of lookups per app.

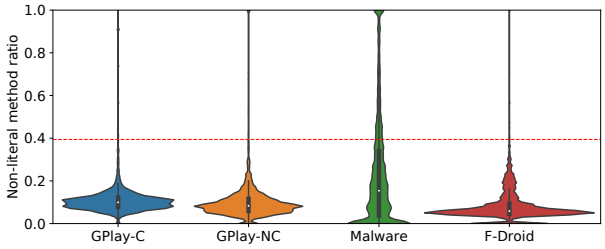


Figure 3: Distributions of non-literal lookups for all apps with more than 10 reflective method lookups.

reflection-based obfuscation in 29 (83%) of the "noncommercial" apps, and in 13 (37%) of the "commercial" ones. The corresponding figure was 20 (57%) for the malware apps. These findings clearly indicate that, in a large proportion of apps with high ratios of non-literal reflective call targets, obfuscation is indeed the underlying reason. Furthermore, combined with the distributions of non-literal lookups (Figure 3), our manual analysis indicate that dynamic reflection appears to be a relatively common form of obfuscation in malware. APKiD found signatures of packers or obfuscators in 17 out of the 29 GPlay-NC apps, and in 10 of the 13 GPlay-C apps. In contrast, only 3 out of the 20 malware apps matched a signature. This is in line with the above hypothesis that malware authors more frequently apply custom obfuscation, rather than using commercial tools.

The most common pattern, observed among the vast majority of the obfuscated apps, was applying string encryption to the method-name argument when invoking the reflection API. This was the case for all of the 42 Google Play apps, and for 18 out of the 20 malware apps. It should be noted, however, that it is generally not possible to determine if the developers of an app have targeted the reflection API specifically when applying string encryption, or if the obfuscation of call targets is simply a side effect of liberal overall use of string encryption. Since we observed instances of string encryption unrelated to reflective calls in many of the analyzed apps, we suspect that the latter case is common.

Two malware apps were observed to use techniques other than string encryption to obfuscate call targets. One app loaded method names from an array, using a complex array-index calculation, while another app instead fetched method names from an array that was dynamically loaded from a file. One Google Play app also combined string encryption with loading method names from a file.

Turning our view to false positives, the majority of such cases were due to code from the Google Mobile Services (GMS) API, constituting 77% of all false positives in the malware and Google Play datasets. Most false positives appeared to be due to a particular method, which seems to load advertisements by invoking a class loader, and subsequently calling methods from the loaded class. Other common cases of false positives included custom wrappers around the reflection API for implementing, e.g., error handling.

5.2 Control-flow obfuscation

Here, we present the manual analysis of the 200 apps that were sampled from each dataset, as described in Section 4.2.2. Interestingly, we also noticed here that many false positives were due to methods from the GMS API. These methods often make use of large switch-case blocks for implementing optimized string lookup tables, where the string is compared using a hard-coded `hashCode` instead of using a slower character-by-character comparison. Incidentally, this makes their control structure similar to that of methods obfuscated with control-flow flattening. Prior to conducting the manual analysis, we filtered out GMS methods based on their fully qualified name, using the regular expression "`com/google/android/.*gms/`". This reduced the sample sizes to, respectively, 57, 100, and 142 apps, for the GPlay-C, GPlay-NC, and malware sets.

Overall, the manual analysis revealed 10 cases of control-flow obfuscation in the GPlay-C sample, 15 in the GPlay-NC sample, and 12 among the sample of malware apps. There was no clear correlation among our high-anomaly samples between the anomaly score and the likelihood of a method actually being obfuscated, with a range of anomaly scores between -0.305 and -0.366 being observed for obfuscated methods. Running APKiD on the obfuscated apps yielded only two matches (one GPlay-C app and one malware app), both matching a signature for the DexGuard obfuscator.

The types of obfuscation that we identified could broadly be divided into two classes:

- *Control-flow flattening*, characterized by "`while(true)`" loops with large `switch` statements using random "magic numbers" for the `case`s. We observed 4 instances each of this for the two Google Play datasets, and 6 among the malware.
- *Artificially convoluted control-flow structure*, including large nested `if`, `switch`, or `while` statements, or nested combinations of those. Other observed techniques were excessive use of fall-throughs in `switch` statements, and one case of using nested `while` and `try` statements inside the `case`s of a large `switch` statement. In several methods, highly complex arithmetic was also used in conditional statements.

Sometimes, combinations of the two were also used.

We also observed one instance of what appeared to be reflection-based obfuscation (among the malware), one instance of excessive class overloading, and one instance of package flattening (i.e., to collapse a hierarchical package structure into a flat one), both of the latter in the GPlay-C sample.

The GPlay-C and malware sets contained a pair each of identical obfuscated methods, i.e., methods that were found in two different sampled apps. In all three datasets, there were also several cases of highly similar obfuscated methods encountered in different apps. This could be due to the use of obfuscated libraries within the apps.

Large switch-case blocks (similar to those in GMS) were common culprits for false positives. `try` statements with many `catch` blocks, or nested `try` statements, also caused several false positives.

As a final observation, the decompiler used for the analysis (Jadx) failed on many of the high-anomaly methods. However, only in one out of the 37 cases where Jadx failed the method was actually obfuscated. Instead, the decompiler appeared to predominantly fail when confronted with "legitimate" use of large switch-statements or deep levels of nesting. This is in agreement with the results by Mauthe et al. [19] (c.f., Section 3.1).

6 DISCUSSION

As mentioned in Section 3.3, the exact figures obtained from empirical studies of obfuscation tend to vary significantly, depending on the methodology and datasets that are used. Therefore, attempting to determine definitive figures for the adoption rate of different obfuscation techniques is not meaningful, and this has not been the purpose of our work. Instead, we have aimed to better characterize the in-the-wild use of reflection-based obfuscation and control-flow obfuscation. Below, we summarize the main takeaways of our work, identify avenues of future work, and discuss threats to validity.

6.1 Summary and Main Takeaways

Reflection-based obfuscation. Since the vast majority of apps today make use of reflection, the presence of calls to the reflection API is entirely inadequate as an indicator of reflection-based obfuscation. Naive attempts at obfuscation that use hardcoded arguments to the reflection API can be defeated with limited effort, using modern analysis environments. Therefore, we have focused on the more challenging case where arguments are dynamically generated, and proposed the ratio of reflective method lookups with non-literal arguments as an indicator of this type of obfuscation. Our manual analysis suggests that a high such ratio is indeed a fairly strong indicator of reflection-based obfuscation. (Using our "99th percentile" threshold, around 60% of sampled apps from both the malware and Google Play datasets showed signs of reflection-based obfuscation.)

Our results also indicate that obfuscation based on dynamic reflection is significantly more common in malware. Moreover, as

shown in Figure 2, this difference cannot be explained by malware simply having fewer reflective calls than benign apps. Our findings are in agreement with earlier results [8], which indicated that the use of reflection for obfuscation purposes is more common in malware than in benign apps. Although determining absolute figures for the prevalence of reflection-based obfuscation in malware is not possible using our methodology, we concluded that around 7% of the malware apps in our dataset had a non-literal lookup ratio higher than the 99th percentile of the (presumably non-obfuscated) F-Droid apps. Out of the manually analyzed random sample of those, 57% turned out to use obfuscated reflective call targets. A reasonable ballpark figure for the prevalence of this kind of obfuscation in malware could therefore be around 5–10%, but more detailed studies would be needed to confirm that. Based on our findings, we recommend researchers applying static analysis on Android malware to first triage apps for dynamic reflection, since this obfuscation interferes with, e.g., static call-graph reconstruction

Our manual analysis showed that string encryption is by far the most common cause of statically unresolvable call targets. While it is difficult to say how frequently string encryption was applied with the deliberate *intent* to obfuscate call targets, from a practical point of view, our results imply that most cases of dynamic reflection could be detected by applying existing string encryption detectors [8, 20, 22]. Based on our comparison with APKiD output, however, it is important to use a *generic* string obfuscation detector, rather than relying on signature-based detection, since malware appear to use custom obfuscation much more frequently than benign apps.

Control-flow obfuscation. Our study showed that this kind of obfuscation was used both in Google Play and malware apps. To the best of our knowledge, we are the first to attempt to empirically study the use of control-flow obfuscation in the wild. Interestingly, while existing literature tend to discuss relatively simple control-flow obfuscation techniques, such as junk-code insertion, code reordering, call indirection, or insertion of fake method calls [4, 7, 15, 24], we observed several instances of advanced code obfuscation, which appeared to be variations of the control-flow flattening technique. While the use of such advanced obfuscation techniques does not appear to be very widespread, the results based on our random sample of 200 apps from each dataset indicated adoption rates of around 5–8%, with no marked difference between malicious and benign apps. Moreover, only two of the obfuscated apps matched an APKiD signature, indicating that custom or specialized obfuscation tools were used.

In future work, it would be interesting to study how the advanced control-flow obfuscation techniques that we observed are used by app developers, in order to better understand their practical impact on the reliability of static app analysis. For example, are there general-purpose obfuscation tools that can apply these techniques to arbitrary code, or are the obfuscated methods that we observed specialized implementations? As a potential example of the latter, it is possible that the obfuscated methods are decryption routines for custom string or class encryption implementations. Moreover, are the obfuscated methods part of the apps' main code, or are they included as part of an obfuscated third-party library? The latter case could explain why our observations were similar for both the Google Play and malware dataset, since malware is frequently created by repackaging of benign apps.

Another potential topic of future work would be to improve the accuracy of our anomaly detector. One concrete improvement that could be made, in order to reduce the number of false positives, is to include the Google Mobile Services classes in the training data.

6.2 Limitations and Threats to Validity

We have already discussed the fundamental limitations in evaluating the recall of our detectors in Sections 4.2.3 and 4.2.1, stemming from the "bootstrapping" problem that we face, and the resulting lack of ground-truth data.

Limitations of the analysis framework we use, i.e., Androguard, constitute a threat to internal validity. For example, the built-in decompiler failed on a total of 44,827 methods (distributed across 3,766 Google Play apps, 11,556 malware apps, and 37 F-Droid apps), preventing us from analyzing the AST for reflection use in those cases. (Out of the failed apps, the number of failures per app was not very large, however, with median of 3 for the Google Play and malware apps, and 2 for the F-Droid apps.)

As with any empirical study, the particular way in which datasets were compiled could pose a threat to its external validity. In particular, determining to what degree a malware dataset is representative of in-the-wild malware is very challenging. This limitation also applies to our set of AndroZoo malware. Moreover, the choice of VirusTotal detection threshold (20, in our study) also influences the selection of malware, where a high threshold might bias the dataset towards easy-to-detect malware, whereas a low threshold could lead to the inclusion of false positives (i.e., non-malware). Another potential risk related to using a high threshold is that detection rates might be lower for obfuscated malware, biasing the dataset towards fewer obfuscated samples. It is also possible that our selection strategy for Google Play apps influenced the results.

Finally, using the F-Droid apps for training the anomaly detector could also pose a threat to external validity, if the open-source apps are not sufficiently representative of Android apps in general. (The false positives on GMS methods appear to be one example of such lacking generality.)

7 CONCLUSION

In this work, we have characterized the use of code obfuscation in both benign and malicious Android apps, by means of a large-scale empirical study. In addition to a new method for detecting suspected reflection-based obfuscation, we have proposed a novel approach to triaging apps for the presence of control-flow obfuscation, based on identifying anomalous control flow. Our approach differs from existing works by not making assumptions on the types of obfuscation used in the wild, and has allowed us to perform what is, to the best of our knowledge, the first empirical study of control-flow obfuscation use in Android apps.

Our results indicate that reflection-based obfuscation is significantly more common in malware, whereas control-flow obfuscation is encountered with relatively low frequency in both benign and malicious apps. An important finding in our study is that we identified several apps that used advanced control-flow obfuscation techniques, which have hardly received any attention in the existing literature on Android obfuscation detection.

REFERENCES

[1] 2022. *APKiD*. https://github.com/rednaga/APKiD

[2] Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery* 29, 3 (2015), 626–688.

[3] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 468–471.

[4] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, and Francesco Mercaldo. 2018. Detection of Obfuscation Techniques in Android Applications. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*. Association for Computing Machinery, 9 pages. https://doi.org/10.1145/3230833.3232823

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

[6] H Bunke and G Allermann. 1983. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters* 1, 4 (1983), 245–253. https://doi.org/10.1016/0167-8655(83)90033-8

[7] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. 2020. Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection. *IEEE Transactions on Information Forensics and Security* 15 (2020), 987–1001. https://doi.org/10.1109/TIFS.2019.2932228

[8] 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 *Security and Privacy in Communication Networks*. Springer International Publishing, 172–192.

[9] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, 896–899. https://doi.org/10.1145/3238147.3240480

[10] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini. 2020. Hidden in Plain Sight: Obfuscated Strings Threatening Your Privacy. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. Association for Computing Machinery, 694–707. https://doi.org/10.1145/3320269.3384745

[11] Ratinder Kaur, Ye Ning, Hugo Gonzalez, and Natalia Stakhanova. 2018. Unmasking Android Obfuscation Tools Using Spatial Analysis. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)*. 1–10. https://doi.org/10.1109/PST.2018.8514207

[12] Marian Kühnel, Manfred Smieschek, and Ulrike Meyer. 2015. Fast Identification of Obfuscation and Mobile Advertising in Mobile Malware. In *2015 IEEE Trustcom/BigDataSE/ISPA*. 214–221. https://doi.org/10.1109/Trustcom.2015.377

[13] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. Association for Computing Machinery, 318–329. https://doi.org/10.1145/2931037.2931044

[14] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. In *Proceedings of the 36th International Conference on Machine Learning*, Vol. 97. PMLR, 3835–3845.

[15] Zhiqiang Li, Jun Sun, Qiben Yan, Witawas Srisa-an, and Yutaka Tsutano. 2019. Obfusifier: Obfuscation-Resistant Android Malware Detection System. In *Security and Privacy in Communication Networks*. 214–234.

[16] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation Forest. In *2008 Eighth IEEE International Conference on Data Mining*. 413–422. https://doi.org/10.1109/ICDM.2008.17

[17] Xiaoxiao Ma, Jia Wu, Shan Xue, Jian Yang, Chuan Zhou, Quan Z. Sheng, Hui Xiong, and Leman Akoglu. 2021. A Comprehensive Survey on Graph Anomaly Detection with Deep Learning. *IEEE Transactions on Knowledge and Data Engineering* (2021). https://doi.org/10.1109/TKDE.2021.3118815

[18] 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

[19] Noah Mauthe, Ulf Kargén, and Nahid Shahmehri. 2021. A Large-Scale Empirical Study of Android App Decompilation. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 400–410. https://doi.org/10.1109/SANER50967.2021.00044

[20] O. Mirzaei, J.M. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano. 2019. AndrODet: An adaptive Android obfuscation detector. *Future Generation Computer Systems* 90 (2019), 240–261. https://doi.org/10.1016/j.future.2018.07.066

[21] Alireza Mohammadinodooshan, Ulf Kargén, and Nahid Shahmehri. 2019. Comment on "AndrODet: An adaptive Android obfuscation detector". *CoRR* abs/1910.06192 (2019). arXiv:1910.06192 http://arxiv.org/abs/1910.06192

Table 2: Temporal distribution of malware samples in our dataset, based on the year the sample was added to AndroZoo.

Year	Count	Fraction
2012	113	0.05%
2013	6,035	2.55%
2014	29,548	12.47%
2015	36,868	15.56%
2016	71,269	30.09%
2017	10,412	4.40%
2018	31,065	13.11%
2019	39,671	16.75%
2020	7,809	3.30%
2021	4,098	1.73%

[22] Alireza Mohammadinodooshan, Ulf Kargén, and Nahid Shahmehri. 2019. Robust Detection of Obfuscated Strings in Android Apps. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*. Association for Computing Machinery, 25–35. https://doi.org/10.1145/3338501.3357373

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[24] Mila Dalla Preda and Federico Maggi. 2017. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques* 13, 3 (2017), 209–232.

[25] Zhengyang Qu, Shahid Alam, Yan Chen, Xiaoyong Zhou, Wangjun Hong, and Ryan Riley. 2017. DyDroid: Measuring Dynamic Code Loading and Its Security Implications in Android Applications. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 415–426. https://doi.org/10.1109/DSN.2017.14

[26] S. V. N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt. 2010. Graph Kernels. *J. Mach. Learn. Res.* 11 (aug 2010), 1201–1242.

[27] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. *Software tamper resistance: Obstructing static analysis of programs*. Technical Report CS-2000-12. University of Virginia.

[28] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. 2018. Software Protection on the Go: A Large-Scale Empirical Study on Mobile App Obfuscation. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 26–36. https://doi.org/10.1145/3180155.3180169

[29] Yan Wang and Atanas Rountev. 2017. Who Changed You? Obfuscator Identification for Android. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 154–164. https://doi.org/10.1109/MOBILESoft.2017.18

[30] 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*. Association for Computing Machinery, 222–235. https://doi.org/10.1145/3274694.3274726

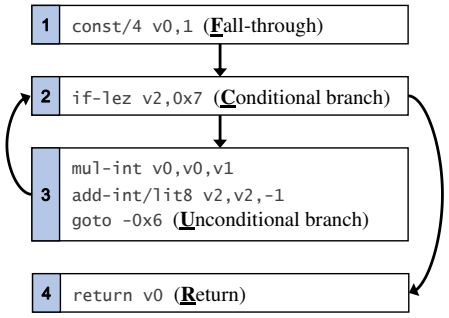
[31] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 363–376. https://doi.org/10.1145/3133956.3134018

[32] Lei Xue, Hao Zhou, Xiapu Luo, Le Yu, Dinghao Wu, Yajin Zhou, and Xiaobo Ma. 2020. PackerGrind: An Adaptive Unpacking System for Android Apps. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2020.2996433

[33] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 01 (2020), 1145–1152. https://doi.org/10.1609/aaai.v34i01.5466

A TEMPORAL DISTRIBUTION OF MALWARE SAMPLES

Table 2 shows the distribution of the AndroZoo malware of apps based on the year of addition.



Per-node n-grams (full paths in boldface)

1	**FCR**, **FCU**, CR, CU, 2xFC, 2xC, 2xF, R, U
2	**CUC**, **CR**, CU, UC, 3xC, R, U
3	**UCR**, **UCU**, CR, CU, 2xUC, 2xC, R, 3xU
4	**R**

Final method signature

{1xCUC, 1xFCR, 1xFCU, 1xUCR, 1xUCU,
3xCR, 3xCU, 2xFC, 3xUC, 7xC, 2xF, 4xR, 5xU}

Figure 4: Example of our custom vector-embedding approach.

B GRAPH EMBEDDING EXAMPLE

Figure 4 illustrates the process of applying our custom vector embedding approach to a CFG, when using $n = 3$. If we use node 2 in the figure as an example, it can be seen that two paths of maximum length 3 are possible: $2 \rightarrow 3 \rightarrow 2$, and $2 \rightarrow 4$, corresponding to the subsequences "CUC" and "CR", respectively. The total n-gram counts for each node is shown in the table below the CFG in the figure. In addition to the n-grams corresponding to the two complete paths, we get the 2-grams "CU" and "UC", which are part of the sequence "CUC", as well as several 1-grams. For example, since there are two occurrences of "C" in the first path ("CUC"), and one in the second path ("CR"), we get a total count of 3 "C" 1-grams. The total n-gram counts across all nodes are shown in the bottom of the figure. To construct the final vector, these would first be normalized by dividing by the number of nodes (4 in this case), and then inserted as elements at their respective positions in the vector. (The frequency counts for all other valid 1, 2, and 3-grams would be set to 0 in the vector.)