



UMEÅ UNIVERSITY

UNVEILING OBFUSCATED ANDROID APPLICATIONS

Nils Nihlén

Bachelor Thesis, 15 hp/credits
BACHELORS PROGRAM IN COMPUTER SCIENCE

2025

Abstract

Obfuscation detection and de-obfuscation can guide security analysts in reverse engineering applications, as well as aid in automated detection of malware. Currently available tools are few, mainly focused on Android apps, and utilize classic machine learning solutions. However, they suffer from a lack of robust datasets for training. Through our evaluation of two detection, and two de-obfuscation tools, we found that detectors relying on supervised machine learning generalize poorly, unsupervised anomaly detection models perform adequately, and finally that de-obfuscators are narrow in scope and limited in their ability to retrieve semantically relevant names from obfuscated APKs. The multitude of ways of constructing an application coupled with the numerous types of obfuscation, and variations thereof, is challenging to accurately represent in a dataset. This highlights the need for better labeled data, more unsupervised learning solutions, such as anomaly detection systems, or alternatives to machine learning methods.

Contents

1	Introduction	1
1.1	Background	1
1.1.1	Android Package Structure	1
1.1.2	Obfuscation Techniques	2
1.1.3	Analysis approaches	3
2	Related Work	5
2.1	Obfuscation Detection	5
2.2	De-obfuscation	6
3	Methodology	7
3.1	Tools	7
3.1.1	APK Analysis	7
3.1.2	Obfuscation Detection	7
3.1.3	De-obfuscation	8
3.2	Datasets	9
3.3	Manual Analysis	9
3.4	Testing Environment	11
3.5	Evaluation	11
3.6	Limitations	12
4	Results	13
4.1	Detection Accuracy	13
4.1.1	DLHybrid	13
4.1.2	ObA	15
4.2	De-obfuscation performance	18
4.2.1	DeGuard	18
4.2.2	JADX	20
5	Discussion	23
5.1	Limitations	24
6	Conclusion	25
	References	27

1 Introduction

Code obfuscation means transforming code to reduce readability and obscure its purpose. Obfuscation is a helpful tool to protect the intellectual property of developers [1, 2], as it hinders reverse engineering. However, obfuscation is also a tactic commonly used by malware authors [1, 2, 3] to hide malicious code. These transformations can let malicious code avoid detection by commercial state-of-the-art malware detectors [3]. Obfuscation is commonly used on Android applications due to how simple it is to restore Java code from its compiled form [1]. With Android having a market-share of over 70%¹² this affects a very large user-base. With few available automated tools to aid malware analysts overcome this problem we provide a study on commonly employed obfuscation techniques and an evaluation of some available obfuscation detection and de-obfuscation tools, with our research question being: *Are the strategies employed by these tools more suited for specific types of applications?*. Our findings suggest that the lack of labeled and non-biased datasets hinders promising supervised learning tools like the one proposed by [4] from generalizing to data not transformed by the obfuscators they used to create their training datasets. Similarly the lack of good diverse sets of non-obfuscated applications hamper the performance of unsupervised learning tools using anomaly detection like ObA³ [5], though much less.

The remainder of this thesis is structured as follows; first some background about the Android Package structure and types of obfuscations. This is followed by a review of similar research divided into two topics. After the review, our methodology is outlined and described in terms of tools, datasets, testing environment, and evaluation. This is immediately followed by a presentation of our findings and a discussion. Lastly, we conclude the thesis and propose further research topics.

1.1 Background

The focus of this study is on obfuscation in Android applications, so the following section will explain the basic structure of an Android Package (APK) as well as go over commonly found obfuscation techniques.

1.1.1 Android Package Structure

An Android Package (APK) is a compressed zip directory containing all the parts of an Android application. The notable components of an APK are the following:

- **AndroidManifest.xml:** The Android manifest is a structured file containing information about the services, components and user-permissions that make up the application.

¹statcounter <https://gs.statcounter.com/os-market-share/mobile/worldwide>

²Statista <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/>

³NoahMauthe *ObA* (2023) <https://github.com/NoahMauthe/ObA>,

- **classes.dex**: holds the classes compiled into a Dalvik executable (.dex), this is where the applications code is and its the main target for obfuscation.
- **resources.arsc**: is a file of compiled resources (eg. images, UI elements) the application uses.
- **res/-directory**: This is a directory of non-compiled resources.
- **lib/-directory**: This directory holds the compiled native libraries used in the application.

1.1.2 Obfuscation Techniques

There are many types of obfuscation techniques and these can range from simple renaming of identifiers to encrypting classes or manipulating bytecode instructions at runtime. The following are the techniques we discuss, test, or found during this research:

- **Identifier renaming** is an obfuscation that replaces the names of variables, classes, methods etc. with incoherent or misleading, often short strings. By removing the semantic clues in an identifier the purpose of it becomes hard to deduce. Many commercially available obfuscators (eg. Proguard, DexGuard) have options to obfuscate an APK like this. Though it is a simple technique it can be very effective against manual analysts as well as static analyses focusing on identifying use of certain classes or methods. Renaming often takes the form of single letter identifiers (eg. 'a') but can also consist of strings such as "oO00OoOo0", or random HEX values. Excessive overloading also falls into this category, where for example multiple methods in the same class use the same identifier, only differing by their parameters.
- **Control-flow manipulation** is a broad type that consists of any manipulations that obscure the execution flow throughout an app at runtime. It can be code that is never executed or serves any purpose, (i.e. dead code), or removing loops and replacing them with multiple if-statements (i.e. code flattening). Another common way to achieve obfuscated control flow is to include multiple `goto` instructions, such as massive `switch`-statements jumping between cases without affecting the program state. Another technique is using multiple wrapper classes, or methods with the sole purpose of calling the object they wrap, introducing layers between an object and it's usage.
- **String encryption** obfuscates by encrypting constant string literals and decrypting them during runtime. This is a common strategy in malware to hide URLs that retrieve a malicious payload. Strings can be chopped up into smaller segments, encrypted, and hidden throughout the application. The decryption key is then either hidden in the same way somewhere in the application, or it is dynamically loaded at runtime. String encryption is well used in conjunction with the Java Reflection API to obscure the names of the reflected classes and methods.
- **Repacking** is the act of decompiling an APK and shuffling or renaming classes followed by repackaging it with a new developer key. This will change the signature of the app which some malware detection tools utilize to identify malicious applications. This is often used to either repackage already known malicious applications to avoid detection, or to repackage benign apps with malicious inserts of malware or ads. Repacking can also mean encrypting the original APK of an application and bundling it within a wrapper APK. This hides all of the original applications code effectively and requires decryption to access.

- **Native code stripping** is usually C/C++ code preemptively compiled into a `.so` (shared object) file and stored in the `lib/` directory. This shared object is accessed and executed by the Java Native Interface (JNI). The pre-compiled native code is stripped of high-level symbols and names, making it harder to reverse engineer.
- **Dynamic code loading** means loading classes or libraries during runtime, the loaded code does not have to be bundled in the APK, meaning it does not need to be declared in the `AndroidManifest.xml`. The loading makes use of the `DexClassLoader` API to execute code not installed as part of the APK. This way, the use of libraries can be hidden to obfuscate the control-flow and intent of an application. This is challenging to detect with static analysis (i.e. analyzing the app without executing it), especially when combined with reflection or encryption.
- **Reflection** is often used for backwards compatibility [3] in benign applications, however it is also a common tactic used by malware authors to hide the use of classes and methods from static analysis through their dynamic invocation. Use of the Reflection API allows access to private and hidden fields or methods in a class dynamically. The distinction between reflection as obfuscation or not can be made by examining the number of calls to the API and whether the reflective targets are obfuscated by eg. string encryption [5] where malicious applications are shown to utilize reflection more often and conceal the reflective targets.

1.1.3 Analysis approaches

There are two ways of analyzing an application: statically or dynamically. Static analysis refers to examining the code itself in an application, dynamic analysis instead examines an applications behavior during runtime. A static approach will have higher code-coverage and more efficiency w.r.t computational resources [6] in comparison to dynamic analysis as static analysis generally does not include executing any instructions, only analyzing the application components. This however makes static approaches more susceptible to obfuscation as dynamic class loading, encryption and control-flow manipulation among others may not be apparent until they are executed. Dynamic analysis is a more involved and costly process, with regards to time, computational effort, and scalability, but it allows detection of more complex obfuscation techniques. Due to dynamic analysis executing instructions and tracking the control flow throughout an application at runtime, it can only analyze the code being executed. This requires thoughtful planning and testing for any significant code coverage. There are detection and de-obfuscation frameworks that utilize hybrid solutions combining the two approaches, e.g. [7]. However they are not common.

2 Related Work

Obfuscation in Android applications is a well-researched topic. However there are few available tools for analysts to use when reverse engineering obfuscated applications. Even fewer are actively maintained or recent enough to combat the growing number of Android malware [3] and complexity of obfuscation techniques. There is also not one single tool capable of detecting a majority of the different types of morphing techniques. Our goal is to evaluate the strategies employed by some available obfuscation detection and de-obfuscation tools, and determine whether they are more suited for different types of applications. To the best of our knowledge, this has not been addressed in any relevant research. The following are studies relevant to the topic of obfuscation as a whole. Several studies have been performed discovering different obfuscation techniques. [1] conducted a study on obfuscation techniques encountered 'in the wild' and where they are commonly found. The study by [3] completes a thorough review of 511 research papers on obfuscation between 2010 and 2021 and propose a taxonomy of evasion techniques (i.e. obfuscations applied specifically to avoid detection by anti-virus and malware detection systems). The study also evaluates state-of-the-art malware detectors on their scope and level of testing, highlighting that most malware detection frameworks do not account for enough evasion techniques. [8] provide an overview of obfuscated malware detection techniques, as well as data- and feature-sets used in literature studying the topic. The study also evaluates the impact of obfuscation on malware analyses.

2.1 Obfuscation Detection

A promising obfuscation detection tool is proposed by [9] utilizing graph convolutional networks to detect obfuscation on a function level. However their proposed tool is unfortunately not publicly available. Another interesting tool is presented in a study by [10]. It uses a feature-based approach followed by a direct classification by a LSTM-RNN model (Long Short-Term Memory Recurrent Neural Network), this tool is also unavailable. The following studies were relevant to ours and available for evaluation. [5] proposes an obfuscation detection tool both extracting reflective calls and utilizing an anomaly detection model to reveal likely control-flow manipulations in a given APK. Their study also criticizes the current reflection obfuscation detection strategies, not distinguishing between normal use of the Reflection API and its use as an obfuscation technique. The study by [4] uses a multi-model system as a solution for detecting a wider range of techniques including identifier renaming, reflection, string- and class-encryption. Their proposed tool builds on AndrODet [11] and incorporates models utilizing natural language-processing and image recognition. The tool AndrODet proposed by [11] is an online learning system focusing on detecting obfuscation by renaming, string encryption, and control-flow manipulation. The study's evaluation of its string-encryption detection accuracy has been criticized for using biased datasets, but it remains an advanced detection model, hence its use by [4].

2.2 De-obfuscation

The use of de-obfuscation tools in conjunction with malware detectors is shown to improve the detection accuracy of the latter [6]. De-obfuscated code is also easier to read in manual analysis, making it an attractive resource. But, currently available de-obfuscation tools are mostly limited to reversing identifier renaming and control-flow manipulations [12, 13]. There are de-obfuscators meant to retrieve encrypted or encoded strings and restoring them eg. Katalina¹ but its efficacy is not documented or tested. [1] lifts the narrow scope of available de-obfuscators as an issue, mentioning that these tools focus on reversing singular obfuscation techniques, or even single techniques applied by specific off-the-shelf obfuscators, an example of the latter is [14]. What follows are some de-obfuscators we found relevant for our study. DeGuard is a statistical de-obfuscation tool proposed by [12]. It reverses renaming of classes, methods, fields and packages. The tool is only available online². The state-of-the-art DEX to Java decompiler tool JADX³ features a de-obfuscator which to our knowledge has not been studied or evaluated. The de-obfuscator included reverses identifier renaming. [7] propose a de-obfuscation tool using a hybrid static-dynamic strategy. It performs static analyses to locate anomalous areas in code, then feed the results to the dynamic analysis step, and iteratively repeat these steps until no further anomalous code can be located. The tool is however not available in its entirety, and focuses on renaming as well as more advanced obfuscation techniques they refer to as runtime-based obfuscation.

¹huuck *Katalina* (2023) <https://github.com/huuck/Katalina>

²*DeGuard* (2016) <http://apk-deguard.com/>

³Skylot *JADX - DEX to Java decompiler* (2024) <https://github.com/skylot/jadx>

3 Methodology

This section explains the different tools and datasets utilized in our research. As well as how the manual analysis of our dataset was conducted, the environment tests were performed in, and finally how the detection and de-obfuscation tools were evaluated.

3.1 Tools

What follows are the tools evaluated in this study, as well as the tools used to manually inspect applications when establishing the ground truth for the experiments.

3.1.1 APK Analysis

We used JADX¹ and apktool² to analyze the APKs in the datasets. JADX was chosen for this task as it has the best performance according to the empirical study by [15], we used apktool due to its availability and our previous experience with it. JADX graphical interface was used to decompile and navigate application code. Each of the applications were searched for the obfuscation techniques outlined in section 1.1.2. The use of an obfuscation technique by an application was rated as

- **-1** if there was no obfuscation of that type, or any use of classes, libraries, or methods associated with this technique.
- **0** if there was no obvious obfuscation of the corresponding type, but there was use of either obfuscated external libraries, classes, or methods associated with the obfuscation technique (eg. Use of reflective or encryption methods but not with the intent to obfuscate).
- **1** if there was obfuscation by the corresponding technique present and not just in external libraries.

apktool was used as a complementary tool to analyze parts of the APKs which were unsuccessfully decompiled by JADX or hidden. APKs are labeled as obfuscated by specific techniques if they were present in the DEX-layer. We excluded found obfuscation techniques, or highly optimized code which might appear as obfuscated in common libraries such as `android.support.v4.hardware.fingerprint` which frequently uses encryption when accessing and managing the devices stored fingerprints. Many common packages and libraries are obfuscated, but not by the developers who use them in their applications.

3.1.2 Obfuscation Detection

The first detection tool we evaluated was ObA³. The tool was developed as part of a study by [5]. ObA performs static analyses of given APKs, detecting methods whose control flow

¹Skylot *JADX - DEX to Java decompiler* (2024) <https://github.com/skylot/jadx>

²iBotPeaches *Apktool* (2025) <https://github.com/iBotPeaches/Apktool>

³NoahMauthe *ObA* (2023) <https://github.com/NoahMauthe/ObA>

is obfuscated, and obfuscation by reflection. The control flow manipulation detector uses an Isolation Forest (IF) model. IF systems are unsupervised, ensemble-of-trees machine learning models with little required hyperparameter tuning. These models are used for anomaly detection, which ObA utilizes by having trained their model on numerous open-source projects as they are generally not obfuscated. The anomaly detector analyzes an APKs `smali` files (decompiled DEX code) and flags methods that deviate from its established "normalcy" enough to fulfill a fixed threshold. To determine whether an application is obfuscated by reflection ObA counts the number of reflective method look-ups and how many of them have non-literal targets. Any reflective call with a hidden target (i.e. any reflective call without a string literal argument) is counted as obfuscated. The number of reflective method look-ups is stored in a database populated by ObA. The information was then extracted by us and used to calculate the ratio of obfuscated method look-ups to total number of method look-ups.

The second detection tool was proposed in [4]. The tool does not have a specific name and will in this study be referred to as DLHybrid. This obfuscation detector utilizes multiple models to classify an APK as obfuscated by Trivial (T) transformations (i.e. identifier renaming), String Encryption (SE), Class Encryption (CE) and Reflection (R). DLHybrid uses three models with different approaches and then combines their output along with an entropy feature to classify an APK. The models utilized are an image processing convolutional neural network (CNN), a natural language processing (NLP) model, and a custom implementation of the tool Androdet [11], referred to as new Androdet, or Androdet*. This custom implementation focuses only on detecting identifier renaming, since that is where it performed best. Before classifying any data, the APKs are preprocessed into the different representations the models expect. The image processing CNN model takes as input an image representation of an APK created from the Dalvik bytecode instructions, and outputs four independent probabilities, one for each type of obfuscation (T, SE, CE, R). The NLP model uses a TF-IDF vector of the most frequent opcode n-grams, essentially looking at how important a specific word (opcode in this case) is to a document. From the TF-IDF vector the model then outputs an array of four probabilities, the same as the CNN. Androdet* extracts features such as average identifier length, number of identifiers with length 1, 2, and 3 characters, and the average similarity score between a word and the next word found in a given application. These features are then used to output a probability of whether the APK is obfuscated by renaming. These three outputs are then fed into a hybrid model which fuses them and a single-value entropy feature calculated from the byte-level representation of an app, after which a final array of four probabilities are output.

These two tools were chosen due to their availability, relatively wide coverage of obfuscation techniques, and differing strategies employed (i.e. both supervised and unsupervised machine learning as well as heuristics).

3.1.3 De-obfuscation

DeGuard⁴ [12] analyzes a given APK and builds a dependency graph over how the applications components relate to each other. The relationships between two components represent different feature functions, which combined with the dependency graph specifies a probabilistic graphical model called a Conditional Random Field (CRF). The graph, features, and model are then used to predict names, under a set of constraints. The constraints ensure that the resulting APK is syntactically well-formed, i.e. the names adhere to the rules for identifiers, such as fields in the same class having unique names.

⁴DeGuard (2016) <http://apk-deguard.com>

JADX⁵ is first and foremost a DEX to Java decompilation tool. But, it features a built-in de-obfuscator that reverses renamed identifiers. There is little to no documentation on how the de-obfuscator works, and to the best of our knowledge it has not been evaluated or used in any relevant studies.

We chose these tools based on their availability and focus on reversing identifier renaming. Renaming transformations are among the most common types of obfuscations and can be applied by several different obfuscators. This motivated the choice of tools as we could accurately apply and observe the renaming operations as well as effectively compare the de-obfuscated names to their original identifiers.

3.2 Datasets

The datasets used to evaluate the tools are randomly chosen subsets from the Androzoo [16] dataset. The subsets were chosen randomly to represent actual 'in-the-wild' applications, as opposed to apps obfuscated by specific obfuscators. We utilized two different subsets (I) A set of 30 benign applications with reported VirusTotal (VT) scores of 0. (II) The second dataset consists of 20 randomly chosen applications with a VT score ≥ 20 . All of the applications chosen were filtered by their dex date between 2018-01 and 2024-12 to capture a variety of applications and test how more recent apps might affect the tools. The applications were also filtered by their size being between 2MB and 50MB. The choice to use multiple datasets was made to differentiate between the types of obfuscations encountered in different applications. And to evaluate the tools performance on applications with multiple obfuscations. Malicious applications are more likely to be obfuscated, and more likely to apply multiple or custom obfuscations as observed by [1], our findings support this as well.

3.3 Manual Analysis

To measure the accuracy of the tools, a manual analysis was conducted on both the datasets. This pre-processing step consisted of looking at the following to determine if corresponding obfuscations were present:

- **Renaming:** meant looking at naming conventions. We encountered multiple types of renaming, most common was single character names eg. 'a', 'b'. Other types included randomized strings of characters, both consisting of randomly chosen characters from the set of all legal identifier characters (i.e. a-z, A-Z, 0-9, _, \$), and randomly chosen from a subset of the legal characters (eg. "OoO00Ooo0", "Ii1IiI1"). Lastly we encountered identifiers consisting of two random words (eg. "taxihealth", "trophyghost"). APKs containing any of these types of identifiers (outside of external packages or libraries) were classified as renamed.
- **Reflection:** First we located use of methods that reflectively access classes or methods, including `Class.forName`, `Class.getMethod`, `Class.getDeclaredMethod`, and `Method.invoke`. Locating these calls was followed by looking at how they are used. Applications containing reflective calls with multiple wrapper methods or encrypted string arguments were labeled as obfuscated by reflection.
- **Encryption:** Searches for "crypto", "cipher", "secretkey" together with searches for common encryption libraries such as `javax.crypto` and `BouncyCastle` located

⁵Skylot, *JADX - DEX to Java decompiler* (2024) <https://github.com/skylot/jadx>

their use. This was coupled with searching for byte-array operations, as some of the applications we analyzed utilized simple XOR-operations to encode strings. Determining whether these were used for obfuscation or not was done by examining their targets. Encryption and decryption of strings or byte-arrays used in reflective lookups or method arguments, as well as decryption of classes or native files (i.e. .so or .jar files) were classified as obfuscated by encryption.

- **Control-flow manipulation:** to determine whether an application had an obfuscated control flow we tracked the execution flow, starting from the `MAIN` activity declared in the `AndroidManifest`. Any impossible `if`-conditions (eg. `if 0 != 0`), multiple nested `try-catch`-blocks, chains of wrapper classes, or wrapper methods (eg. a method with the same name as the intended target method, but with a different set of accepted arguments, that then discards the differing input and calls the intended function) were classified as obfuscated.
- **Native code:** Since many applications use native libraries (.so files) to load for example parts of an app written in C/C++, or maps and textures for Unity3d game apps, we first looked for use of the Java Native Interface (JNI) and its `loadLibrary` method, then at the loaded target. If the shared object was encrypted, or its name hidden when loading, it was classified as obfuscated.
- **Dynamic Code Loading:** Similarly to the native code, it is not uncommon to dynamically load external packages or libraries. So, we labeled applications hiding the target package by encrypting and decrypting the name, or dynamically assembling the name at runtime as obfuscated.
- **Repacking:** We classified any APK as repacked where the decompiled DEX layer of the application was a simple wrapper that decrypts and loads the actual application. Most of the repacked apps we encountered followed similar formats produced by either the Jiagu or Tencents Legu packers.

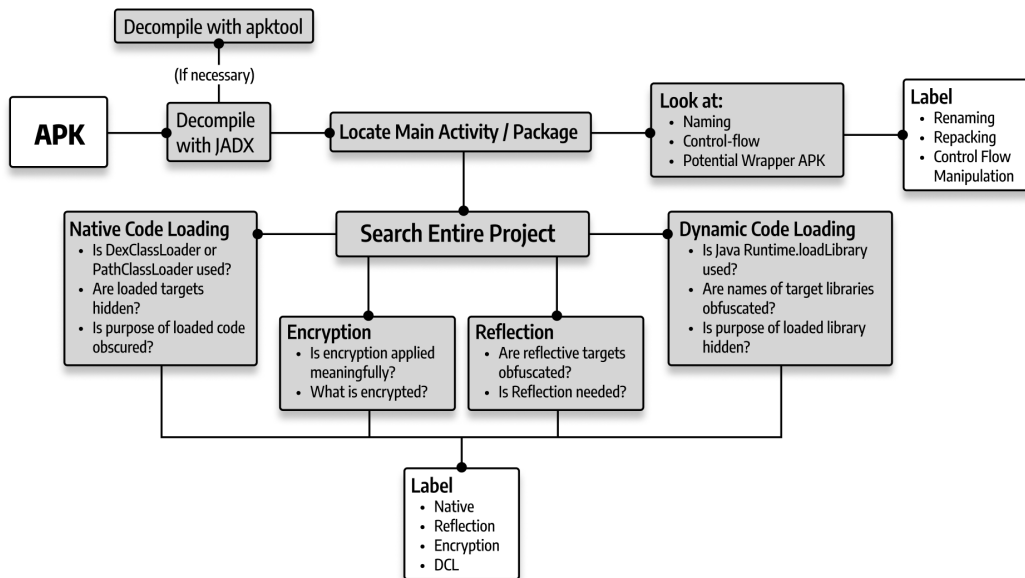


Figure 1: General workflow of manual analysis.

Figure 1 describes the general workflow of how the manual analyses were conducted. After the analyses we adapted ObA to run on local data, following their thorough document-

tation on how to do so. This included writing a few custom classes and adapting the local dataset as well as the database storing the results. DLHybrid also required some adjustments to set up, namely fixing minor errors, labeling the data, and running pre-processing scripts. After the setup, the two tools were run separately on both datasets, one at a time.

3.4 Testing Environment

The tools were evaluated separately and in different virtual environments. The host device runs on Ubuntu 24.04.2 LTS, with a AMD Ryzen 7 2700X Eight-Core CPU, and 16GB of RAM. The virtual environment running ObA used Python 3.12.3, while the one running DLHybrid used Python 3.9.18 to match the tools components versions. The entire list of installed packages and their versions can be found in the appendix.

3.5 Evaluation

This section covers how the detectors and de-obfuscators were evaluated. This generally consisted of comparing their produced output to our labels and the classified applications code directly.

Detectors: DLHybrids results were interpreted by analyzing the terminal output, comparing individual predictions to our labels and looking through the classified applications code in the JADX GUI to attempt to find the reasons for the produced results. This included looking at naming conventions, use of reflection, and encryption, as well as where these were used. The ObA results were extracted from the database populated by the tool and interpreted. Methods reported as anomalous by ObA were inspected to ascertain as to why, then compared to our labels and notes on specific cases of control flow manipulation. The authors of ObA [5] used the ratio of reflective method look-ups with non-literal targets to total number of reflective method look-ups, and a threshold of 0.4 where applications with an equal or higher ratio were deemed to be obfuscated by reflection. We used the same criteria, extracting the results and calculating the ratio, followed by classifying the samples accordingly.

De-obfuscators: To assess the performance of JADX and DeGuard, we chose an application that through our manual analysis was confirmed to be without obfuscation, excluding obfuscated external packages. We then used the obfuscator tool Obfuscapk⁶ proposed in [17] to apply renaming on classes, methods, and fields. Additionally, we obfuscated another application using ProGuard to see if the renaming operations applied by either obfuscator would affect the performance. As ProGuard is a tool integrated in the APK building process, and requires specifying each entry-point in a project to be obfuscated, we used an application made as an assignment for a course⁷ on Android app development, with which we are familiar. This application was obfuscated by the standard 'Minify' optimization and obfuscation step that ProGuard performs when generating a signed APK through Android Studio. Similarly to Obfuscapk the ProGuard renaming operations are applied to classes, methods and fields, as well as the majority of package names. The names that were generated by Obfuscapk all started with 'p' for renamed packages and classes, 'm' for methods, and 'f' for fields followed by a random HEX value of 8 characters. A portion of the renamed fields also appended an extra 8 random characters from the set $\{0-9\} \cup \{a-z\} \cup \{A-Z\}$. The names generated by ProGuard

⁶ClaudiuGeorgiu *Obfuscapk* (2022) <https://github.com/ClaudiuGeorgiu/Obfuscapk>

⁷Umeå University, *Utveckling av mobila applikationer* <https://www.umu.se/utbildning/kurser/utveckling-av-mobila-applikationer2/>

are all either single characters from the set $\{a-z\}$, or two characters beginning with a letter from $\{a-z\}$ followed by a number from $\{0-9\}$. The obfuscation process was followed by running the de-obfuscators on the obfuscated APKs and comparing their output to the original applications and its obfuscated counterpart to find whether the renamed identifiers were successfully reversed or a semantically comparable name suggestion was made. Both DeGuard and JADX deobfuscator provide mapping files containing every identifiers obfuscated and deobfuscated name, this file was used to further analyze the de-obfuscated results.

3.6 Limitations

Due to our choice of building our own datasets, the required manual analyses limited the number of applications we were able to include. While this avoided the issue of bias resulting from using a single obfuscator when building a dataset, some types of applications might instead be overrepresented. As could the frequency of obfuscation techniques and combinations thereof. Another potential limitation is the assumption that a *semantically comparable name*, in our evaluation of the de-obfuscators, is a name that concisely and intuitively describes the purpose or intended use of a class, method, or variable.

4 Results

This section contains the results of our experiments divided into two general categories. First the obfuscation detectors test results, second the de-obfuscators.

4.1 Detection Accuracy

The obfuscation detectors results are divided by their targeted obfuscation technique. DLHybrid is discussed by its detection of renaming, string encryption, and reflection, while ObA's results cover control-flow manipulation and reflection. DLHybrids results were not significantly impacted by the differences in the datasets. However, it did not generalize well to our data. Our experiments resulted in a 2% (1/50) accuracy for detecting renaming, 38% (19/50) for string encryption, and 32% (16/50) for reflection, across all applications. ObA's control-flow obfuscation detectors performance was affected by the presence of several common external libraries in the benign dataset, producing multiple false positives. The reflection detection demonstrated a good performance correctly labeling 94% (47/50) of all applications.

4.1.1 DLHybrid

The output produced by DLHybrid consists of an array of prediction probabilities for each classified APK, and at the end, three arrays showing the precision, recall, and F1-score when comparing the predictions to the true labels. Precision is the percentage of true positives out of all positive predictions, recall is the percentage of true positives among all actual positives, and the F1-score is the harmonic mean of precision and recall. The output arrays of values corresponds to [T, SE, R, CE] i.e. [Renaming, String Encryption, Reflection, Class Encryption]. As we could not confirm any cases of class encryption in our samples we have omitted analysis of DLHybrids performance regarding it.

Malicious application results

Renaming: Evaluating the set of malicious applications, DLHybrid did not predict any application as obfuscated by renaming (hence the 0 F1-score in Table 1), despite 85% (17/20) using some form of it. DLHybrid uses a threshold of 0.5, only labeling apps as obfuscated, whose predicted probability is greater than or equal to this. The predicted probabilities ranged from 0.002 to 0.230, indicating very low confidence in finding any renaming. The set of malicious applications utilized much more excessive overloading, giving multiple classes, packages and methods the same name, and in some cases used names with Chinese, Arabic, or unprintable characters (eg. zero-width spaces). This coupled with the fact that most of the applications do not rename everything, only select packages, classes, methods, and fields are the likely cause of the low detection rate.

Table 1: Precision, Recall, and F1-Score by category on malicious apps

	T	SE	R
Precision	0.0	0.5	0.4
Recall	0.0	0.4286	1.0
F1-Score	0.0	0.4615	0.5714

String Encryption: The detection accuracy for string encryption was higher. DLHybrid classified 30% (6/20) of the applications as using string encryption. However, only three of these were True Positives (TP). In total the set contained seven samples which were obfuscated by string encryption. The False Positives (FP) all utilized many different encryption, and encoding operations, though not for encrypting strings, instead they were mostly used to encrypt network operations (i.e. requests, responses, connections, URLs etc.). There were 4 False Negative (FN) samples, including two applications using a custom XOR-encoding on strings but little to no use of any encryption libraries. One application used encryption similarly to the FP samples, mostly for network operations, but with a handful of encrypted log statements and strings. The last FN sample was repacked by way of encrypting the original application and wrapping it in a stub app. The wrapper application encrypts all its strings as part of the APK encryption. This suggests that the string encryption detection accurately finds heavy use of encryption operations, but is limited when differentiating between encryption as obfuscation and benign use.

Reflection: DLHybrid seemed to perform best when detecting reflection obfuscation, but upon inspecting the output probabilities it had classified every malicious application as obfuscated. The high precision, recall, and F1-score (as seen in Table 1) come from the fact that 40% (8/20) of applications used reflection to obfuscate. Every app contained reflective look-ups, which suggests DLHybrid was unable to differentiate between ordinary use of reflection and obfuscation by it.

Benign application results

Renaming: All types of obfuscation are more infrequent in our sets of benign apps compared to the malicious set. 30% (9/30) of applications had renamed identifiers. The set of TP in classifying renaming consisted of one sample with a predicted probability of 0.557. This sample had applied renaming to most classes, methods, and fields, likely similar to the data DLHybrid is trained on. The probabilities corresponding to the set of FN classifications of renaming ranged from 0.012 to 0.114, again showing low detection rates. The set of True Negative (TN) probabilities ranged from 0.013 to 0.310, seemingly at random. Three applications developed through MIT App Inventor had higher probabilities (0.178, 0.239, and 0.251), as these consist of automatically generated code, identifiers might falsely appear obfuscated. However, another TN sample containing automatically generated code following similar naming conventions had a significantly lower predicted probability (0.019). This difference could either be due to different naming conventions or the amount of unobfuscated core framework classes in use, we were unable to find a clear cause.

Table 2: Precision, Recall, and F1-Score by category on benign apps

	T	SE	R
Precision	1.0	0.0833	0.0
Recall	0.1111	0.6667	0.0
F1-Score	0.1999	0.1481	0.0

String encryption was only present in 10% (3/30) of applications in this set. DLHybrid predicted that 80% (24/30) were obfuscated by string encryption. Two of the 24 positives were TP, and among the six negatives, one was FN. The sample with the highest predicted probability (0.983) was the only application without any form of encryption or encoding operations. And the application with the lowest prediction (0.004) was the only FN sample. The remaining samples predictions ranged between 0.05 and 0.93, seemingly at random, suggesting poor generalization to unseen data.

Reflection Similarly to the set of malicious applications, DLHybrid predicted that 90% (27/30) applications used reflection to obfuscate their code. However, despite all of the samples making use of the Reflection API, none did so to obfuscate. An argument can be made that a repackaged, encrypted app using reflection to invoke classes in a native object uses reflection as a light layer of obfuscation (like two of our samples), yet these do not obscure the reflective targets in any significant manner, and the model predicted lower probabilities (0.065 and 0.713) for these two samples compared to most others. The set of FP ranged from 0.713 to 0.996, excluding the sample from the previous example the range is 0.866 to 0.996. Again suggesting limited ability to differentiate between obfuscation and ordinary use of reflection. The set of TN samples consisted of three applications, two of which were developed through MIT App Inventor, and one which, as mentioned, was repacked. Since the repacked sample is a lean wrapper application, it does not make as many reflective look-ups or invocations as the other apps in the set, this is a possible reason for the low predicted probability. The two other samples in the TN set did however make heavy use of the Reflection API, similarly to the third application made through MIT App Inventor (whose prediction was 0.977), we have yet to find a reason for this difference in DLHybrids predictions.

4.1.2 ObA

ObA’s control flow manipulation detection found anomalous methods in 65% (13/20) of the malicious samples, and 93.34% (28/30) of benign samples. However, a large portion (300/561) of the detected anomalies across both datasets are from a common external library `com.google.android.gms`. This particular library’s impact on both the anomaly detector and use of non-literal reflective method look-ups is outlined in the study [5] associated with ObA. Therefore we will exclude anomalies raised due to this library in our analysis, as well as leaving its internal reflective method look-ups out of the ratio-calculation when classifying reflection. Reflective method look-ups in the GMS API if included, would’ve been the cause for one FP in the set of malicious apps, and 2 FP in the benign set. A potential limitation we found with the anomaly detector was that to save space during runtime ObA prunes small methods, which might be an issue as a common form of control flow manipulation is the use of wrapper methods or classes. Single line methods that only call the actual target would likely be discarded and not counted as anomalous, as would wrapper classes without any actual implementation, only extending the true target class, we encountered both of these techniques in our samples. The number of methods actually analyzed compared to the reported total number of methods indicate the overwhelming majority of methods were

dismissed. On average across both datasets, 0.5% of all methods were analyzed.

Malicious application results

ObA’s anomaly detector is used as an exploratory tool by its authors [5], guiding analysis by reporting anomalous methods, as opposed to classifying an APK or method as obfuscated. For the purpose of our study we chose to consider an application as obfuscated by control flow manipulation if it contained any anomalous methods. The observed results for the set of malicious applications can be seen in the confusion matrix of Table 3.

Table 3: Confusion matrix classifying control flow obfuscation in malicious apps

		Predicted	
		Positive	Negative
Actual	Positive	9	5
	Negative	3	3

As ObA only logs the methods falling under its threshold anomaly score the main limitation with this model is the false negative samples. Without an entry in the database created by ObA, no guidance is provided for manual analysis. Of the five FN samples three were packed, two by Jiagu and one by ApkEncryptor. The wrapper applications created by these packers (at least the versions used for the FN samples) contain methods with excessively nested try-catch blocks, impossible if-conditions and seemingly flattened code i.e. clear examples of control flow manipulation. One of the two FN samples that are not repacked has obfuscated control flow by multiple wrapper classes and methods, the other uses dead code in the form of numerous mathematical operations on extraneous local variables, along with impossible if-conditions. This supports the findings by [5] that ObA is limited in its ability to detect simpler forms of control flow obfuscation (e.g. dead code or unreachable branches), but better at detecting more complex techniques (e.g. flattened control flow or excessive branching), with the exception of those produced by the packers encountered in our samples. As for the FP samples, the overwhelming majority of anomalies were from external libraries similar to `com.google.android.gms`. The falsely labeled anomalous methods not in external libraries were from automatically generated code in apps developed through MIT App Inventor or Andromo.

The tool performed satisfactory in detecting reflection, as seen in the low false positives and negatives in Table 4. According to the classification outlined in subsection 3.5 we found that in the set of malicious applications this yielded only one sample which was classified incorrectly.

Table 4: Confusion matrix classifying reflection obfuscation in malicious apps

		Predicted	
		Positive	Negative
Actual	Positive	7	1
	Negative	0	12

The FN sample was the application with the highest number of reflective method look-ups across both datasets, 16.9% of which were without literal arguments. Relying solely on the ratio was not enough to capture this sample, consideration of the total number of reflective

methods, or non-literal look-ups in conjunction with the ratio could possibly aid in detecting cases like these. Alternatively incorporating a filter that excludes common libraries causing false positives, together with an adjustment to the ratio threshold is worth exploring.

Benign application results

The anomaly detector is heavily influenced by external libraries other than GMS. Sampling individual methods that were flagged as anomalous, they all contain either a) switch statements with numerous cases, or b) large wrapping if- or while-blocks with nested if-, while-, or try-catch-blocks. One of the TP samples we had labeled as using control flow manipulation was due to one specific large method utilizing this format, and because of multiple wrapper methods being used to obscure the execution of a target method, e.g. some method a() calls b(String) with an empty string that is discarded, followed by b(String) calling c() the true target. These wrapper methods were not detected, likely due to the pruning of small methods. However the previously mentioned single method was flagged. The other detected anomalies for this sample were due to external libraries for analytics and cross-platform functionality. Another TP sample had flagged only methods in an external CrashReporter class containing switch-cases for exception handling, but had not flagged the methods we had labeled as anomalous. One of the methods we had labeled contained one wrapping try-catch statement with 23 internal try-catches, nested in varying levels, as well as multiple always true or always false if-conditions (eg. `if 0 == 0`, or `if 0 != 0`). The only structural difference we could find, which could have caused this method to go undetected, while other similar ones did not was that the outer-most wrapping statement was a try-catch statement, where the other similar ones had if- or while-statements as wrappers. Similarly to the last example, another TP sample was flagged due to external libraries, these ones relating to JSON and XML parsers, which contain multiple large switch-statements. But the chain of wrapper classes extending each other to hide the invocation of another class, which were the cause of our true label, had not been detected.

Table 5: Confusion matrix classifying control flow obfuscation in benign apps

		Predicted	
		Positive	Negative
Actual	Positive	4	1
	Negative	18	7

The confusion matrix in Table 6 shows that the majority of applications contained false positives. However, the overwhelming majority of anomalous methods flagged were in common external libraries. Examples include `org.apache.commons.httpclient` and `android.support.constraint`, where the former contains methods for error-handling, using nested try-catch blocks and switch-statements to manage HTTP responses, and the latter contains methods for parsing XML attributes efficiently by mapping them to small integers as cases in a switch-statement. The FP methods that weren't part of such common external libraries, were either automatically generated methods from app development tools such as MIT App Inventor or Andromo, or app-specific implementations of data parsers or HTTP client management. The single FN sample was packed by Tencent's Legu packer. Four methods were deemed obfuscated by control flow manipulation during our manual analysis due to their excessive length and nested structure of while-, try-catch and if-blocks, as well as numerous seemingly redundant operations. Only two methods in the application were analyzed by ObA, and neither was flagged by the anomaly detector. These methods also used try-catch

blocks as their outer-most layer, and not while- or if-statements like those flagged in other applications.

As previously mentioned, none of our benign samples were obfuscated by reflection. But, two samples contain a reflection utility class, which makes heavy use of non-literal reflective look-ups. We deemed these to not be obfuscated, instead following that structure due to modularity since the same class is used to reflectively invoke multiple packed native objects. These two samples are the false positives observed in Table 6.

Table 6: Confusion matrix classifying reflection obfuscation in benign apps

		Predicted	
		Positive	Negative
Actual	Positive	0	0
	Negative	2	28

The two samples in question use Tencent’s Legu packer whose resulting wrapper application contains the reflection utility class used to invoke classes in the packed application. An argument can be made that these employ light obfuscation by reflection, but our assumption is it follows that format due to its re-usability and modularity. The wrapper stub apps are very lean, only featuring classes and methods for unpacking and running the wrapped app, hence their high ratios of non-literal reflective method look-ups to total number of reflective method look-ups.

4.2 De-obfuscation performance

Neither DeGuard¹ nor the JADX² de-obfuscator could restore any of the original identifiers. DeGuard was able to suggest semantically relevant names for only a small portion of identifiers. JADX was able to produce names of classes whose corresponding source files had intact original names, but unable to provide any context for methods and fields names. However its use of numbered prefixes when deobfuscating identifiers does reduce the effect of overloading, effectively differentiating between classes and methods with identical names only in different packages. DeGuard was unable to process the APK obfuscated by ProGuard, and without any meaningful errors or messages we were unable to determine the cause, as such the analysis of DeGuards performance is only in relation to the APK obfuscated by Obfuscapk.

4.2.1 DeGuard

The `mapping.txt` file produced by DeGuard contained 6732 mappings of the form $X \rightarrow Y$, where X is the original obfuscated identifier and Y is the de-obfuscated name. However, only 2017 of these had a change in name ($X \neq Y$). Manually parsing these 2017 mappings, removing any references to renamed identifiers outside the packages obfuscated by Obfuscapk and irrelevant suggested names (eg. $a() \rightarrow f()$), the resulting list of possibly relevant suggested names consisted of 66 mappings.

Ten of these mappings were variables, none of which had semantically relevant names, e.g. 'proxy', 'name', 'parser' in a heap implementation containing none of those terms. 24 of the mappings were renamed methods whose names were never obfuscated, these included

¹*DegGuard* (2016) <http://apk-deguard.com>

²Skylot *JADX - DEX to Java decompiler* (2024) <https://github.com/skylot/jadx>

method names such as 'add', 'all', 'any', 'get', 'map', and 'set', which in turn were renamed to 'cast', 'lift', 'multiply', 'getSongs', 'setPlaylist', 'getPlainText', etc. all but one produced misleading names, sometimes opposite to their purpose. The method whose name was not misleading added little context however, a method 'add' was renamed 'addElement'. Examples of suggested names opposite to the purpose of a method includes two methods originally named 'add' which were renamed 'removeAt' and a third method named 'add' was renamed to 'removeAll'. These add-methods are thread-safe shared list and array insertion methods, any element removal operations are performed on copies of the shared collection and are not the methods purpose. Another eight methods that were not obfuscated, yet renamed by DeGuard were the internal 'as', and 'to' methods in `io.reactivex` classes such as `Observable` and `Flowable`. Every 'as' method was renamed 'checkConstraints', and every 'to' method was renamed 'parseJson'. The 'as' methods simply convert an input object to a caller-specified generic type and returns it. The return case does include a null-check which might have inspired the suggested name. As for the 'to' methods, they are similar, difference is their input is a Function instead of a class object. Neither the 'to' methods purpose or use indicate any correlation to JSON data or parsing. We have not found any indication as to why this name was suggested.

Of the remaining 24 mappings, two were methods named 'a' in an external library, that we did not obfuscate, they were renamed 'requestList' and 'openFile'. The former is a bridging method, wrapping an item in a list as the target method expects one, the list returned from that target is then discarded only extracting the now modified, wrapped element. The 'openFile' method performs an HTML to PDF conversion, but the first action within this method is to open a file, so while the name reflects a small part of the methods purpose, it does not accurately represent its entirety. The remaining 22 mappings were all declared `/*synthetic*/` in JADX, meaning they are not actual methods or code written, they are instead an artifact of compiling Java code containing eg. a Lambda expression or method reference. For example:

```
Runnable r = () -> System.out.println("Hello");
```

creates a Runnable class object 'r' which will execute the lambda expression. Since the expression does not have an identifier, the compiler creates a method so the expression can be referenced. As such they have no original names which we can compare the results to. Instead we will compare the DeGuard-suggested names to the methods purpose, similarly to the obfuscated method names in external packages. Three of these synthetic methods were named 'getFirstLeaf' by DeGuard, one saves a `PDFDocumentCheckpoint`, the other two load or convert HTMLs to PDFs, nothing tree- or leaf-related is found in either method or how they are called. Another example is a method renamed to 'setConnected' which simply sets an `AtomicBoolean` named 'saving' to True, indicating that a document is being saved. 18 of these synthetic methods suggested names were not semantically relevant. Three suggested names were relevant but too generic to accurately represent the purpose of the methods, eg. 'getObjects' for a method which returns an `Observable` for a list of `Bookmark` objects. One suggested name was adequate, 'saveBitmap' for a block that compresses and writes a `Bitmap` to a file and returns a `URI` to it, the name could be more specific, but it does capture most of the methods purpose.

In summary, none of the identifiers generated by Obfuscapk were de-obfuscated, the overwhelming majority of names suggested by DeGuard were semantically irrelevant, a small portion of names indicated the opposite of a methods purpose, and the minimal amount of

relevant names were too generic, with only a single adequate name. The unobfuscated method names (add, all, any, map, get, set) and the synthetic method names generated during compilation (a, b, c, etc.) were likely mistaken for ProGuard-created identifiers, as those usually consist of minimal names of 1-3 characters. The study in which DeGuard [12] was proposed only evaluated their tool on applications obfuscated by ProGuard, this, in combination with our findings that DeGuard was unable to rename a single identifier created by Obfuscapk, suggests that it is limited in its ability to de-obfuscate longer identifiers, or that it is specifically geared towards reversing ProGuard obfuscation. Extracting semantic clues from a method or variable and predicting a relevant name is a complex task which DeGuard was unable to perform in our experiment.

4.2.2 JADX

The JADX de-obfuscator is used by setting an upper and lower bound for identifier length, where any identifier whose length is outside those bounds is renamed. The default setting is minimum length of 3 and maximum length of 64, meaning any names consisting of 1, 2, or 65+ characters are renamed. This indicates that the intended use is for projects utilizing simple ProGuard renaming operations, or similar tools producing single or double character names. As the identifiers generated by Obfuscapk were all either 9 or 17 characters long, this causes an issue with how the range is specified. Either we set the range to a minimum of 2 and maximum of 8 to protect shorter names, but renaming all identifiers with nine characters and longer, obfuscated or not, or the opposite, a minimum of 18 and maximum of 64, protecting only the longer names. We tested both and decided to evaluate the results where the shorter names were protected, as there were more identifiers that fell into that group, meaning fewer unobfuscated identifiers being renamed. This bounds issue can be avoided to some extent by whitelisting packages or classes from being touched by the de-obfuscator. However, several obfuscated packages and classes in our Obfuscapk sample contained both original and obfuscated method names, as such, individual methods would have to be whitelisted to sidestep the issue completely, which is unreasonable to perform manually on a project of this size.

JADX has a built-in feature that handles renaming identifiers if they are unprintable, invalid, cause case sensitivity issues or if the true name of a class could be extracted from the source file name (eg. a class renamed 'p87c74b8a' extracted from the file `LogoutDialog.java` would be named 'LogoutDialog'). Using the tool `dexdump` we found that there are 5067 classes contained in the application obfuscated by Obfuscapk. 2600 of these had been renamed during the obfuscation process, and 1218 of those were successfully restored by JADX through their source file names. While this is a useful feature, obfuscating source file names is not outside the scope of common off-the-shelf obfuscators, using Obfuscapk's 'DebugRemoval' obfuscator would have removed this information.

The file containing the name mappings generated by JADX deobfuscator showed that 55424 methods had been renamed, 22031 of these were from obfuscated packages. With the exception of 141 methods who had been numbered if there were multiple methods with the same name in a class (eg. `call() : String` and `call() : Url` would become `call() : String` and `call2() : Url`), the remaining 21890 renamed methods names did not contain any semantic relevance. The de-obfuscated names generated by JADX follow a pattern of first a character 'm', 'f', 'c', or 'p' for method, field, class, or package respectively, followed by a unique number in the range from 0 to n, where n is the total number of renamed identifiers for that type (i.e. methods have a prefix of 'm0', 'm1', ..., 'm50000', classes have 'c0', ..., 'c5067'). This type identifier and number is followed by an 'x' used as a delimiter separating the prefix and

suffix. The suffix consists of a unique random HEX-value of 8 characters. Similarly to the methods, every field, and package of those we obfuscated also followed this structure. The classes however did manage to retain some relevant information. Any Interfaces, enum, and abstract classes were given names with a prefix reflecting this (eg. class 'p46a2a41c' is an abstract enumerator, renamed 'EnumC4180xc8b1ba68', 'p02b123cb' is an interface renamed 'InterfaceC4129x57257fe1'). Aside from the class names extracted from source files, and those containing 'Interface', 'Abstract', etc. no original names were restored, nor were any semantically relevant ones suggested.

When de-obfuscating the application obfuscated by ProGuard we set the name length bounds to the default, minimum of 3, maximum of 64, since all the identifiers that we obfuscated in this application had names of one or two characters. The obfuscated classes in this APK did not retain their source file names and could not be restored. Similarly to the previous application, classes were renamed according to a format, but not the same one, instead of the delimiter 'x', and the random Hex suffix, the obfuscated class name was used as a suffix, (i.e. class names begin with a 'C' followed by a unique number and then the obfuscated name) As an example, the class 'e' in package 'a' was renamed to 'C0007e'. And just like the previous application, interfaces, abstract classes, or classes implementing or extending core framework types (eg. Runnable, Executor, View.OnClickListener, etc.) kept that information as a prefix to the name (eg. class 'd' implements Runnable, and was renamed to 'RunnableC0006d'). Every method and field follows this same pattern, the new names consisting of a numbered prefix and the obfuscated name as a suffix.

In summary, JADX could provide original class names if the source file names were intact, though this is likely rarely the case as common obfuscators (e.g. Obfuscapk, ProGuard) can easily remove this information. Additionally, JADX does incorporate relevant information about class types or inherited classes in names if they are present. However, no original or semantically relevant names could be provided for methods, fields, or packages. The identifier length bounds used to filter which names are de-obfuscated or not are an issue when obfuscated names are longer than 1 or 2 characters as without excluding individual, unobfuscated methods in otherwise obfuscated packages will remove relevant names. But, the numbered prefixes added during de-obfuscation are helpful since a side-effect of ProGuards renaming structure is excessive overloading, multiple classes and methods with the same name (eg. both packages 'a' and 'a0' contain a class named 'a', which in turn contain methods named 'a'). These numbered prefixes on methods and fields, does make differentiating between same-name objects easier.

5 Discussion

Our original intention was to determine whether specific detection strategies employed by tools like ObA and DLHybrid were more or less suited for certain types of applications. But, with no meaningful difference in DLHybrid's results when classifying benign and malicious apps, the fundamental issue of lacking robust data becomes apparent. Absence of diverse labeled sets of obfuscated apps is a recurring issue in studies on this topic. A common approach (used by [4, 7, 9]) is to build datasets by obfuscating open-source projects with off-the-shelf obfuscators (eg. ProGuard, Obfuscapk). This creates a bias as these tools can only generate so many techniques and ways of applying them. DLHybrid was originally trained on the PRAGuard dataset which consists of 1497 original applications, obfuscated by four transformations in seven different combinations, yielding a total of 10479 samples. The obfuscations applied to these samples encompass the entirety of the applications. An APK transformed by renaming has every source file, package, class, method, and field renamed to random letters. An app obfuscated by string encryption replaces each string literal inside a class with a byte array where each byte is encoded with an XOR operation then decrypted at runtime. This does not necessarily represent how applications are obfuscated 'in-the-wild' according to our random sampling, where mostly only a select few important packages or classes are obfuscated heavily. These differences are likely the reason DLHybrid had trouble generalizing to our data, and would likely display the same issues on similar applications using custom or partial obfuscation or packing. ObA addressed the issues with lack of good data, and its results were satisfactory. The difference between the two datasets did not meaningfully impact ObA's performance. However, the set of benign apps contained far more use of common external libraries, and the amount of false positives produced by the anomaly detector shows that ObA is sensitive to these. This sensitivity could be addressed by either excluding the libraries in question when analyzing methods, or training the anomaly detector on samples utilizing more of them. With the prevalence of machine-learning strategies in obfuscation detection, any supervised learning must take these problems into consideration. Unsupervised learning strategies using anomaly detection would only have the same issues when measuring accuracy, but they sidestep the issue of diversity and bias in labeled datasets of obfuscated APKs for training. Given good, representative sets of un-obfuscated applications, the main drawbacks of anomaly detection models would be lack of explainability and granularity. Anomaly detectors simply detect outliers instead of 'obfuscated by reflection' or 'obfuscated by renaming'. An ensemble of anomaly detectors each with their specific technique in focus could avoid the latter issue, but would require determining which features represent each technique. ObA's anomaly detector did guide the analysis during evaluation of its results, which is valuable when manually investigating an application.

Another issue that arose during this study is the management of packed APKs, or applications obfuscated by native code stripping. Multiple samples in our datasets were obfuscated this way, by using common APK protectors or packers such as Jiagu, Tencents Legu, or Apkencryptor. This thesis restricts itself to the DEX-layer in APKs, i.e. we did not make any attempts at unpacking native objects in our samples. The analysis of the packed samples were only in reference to the wrapper application. Many apps utilize native code or man-

aged assemblies eg. game applications developed with Unity3d. But disassembling the native layer and analyzing its state of obfuscation is to the best of our knowledge outside the current state-of-the-art de-obfuscation and detection tools capabilities, at least those intended for use on Android applications. Tools like Ghidra¹ can decompile native code in applications. Which could be incorporated in the obfuscation detection or de-obfuscation process, but to our knowledge no such tools make use of this. Reverse engineering code written in C/C#/C++ is outside the scope of this study, but the prevalence of native code in the APKs we analyzed suggests a future need to include this functionality in application analysis.

5.1 Limitations

Possible threats to the validity of this thesis include the small datasets used to evaluate these tools, assumptions made based on this data, the necessary custom implementation of the two obfuscation detectors, and the limited number of samples used to evaluate the de-obfuscators.

As we have previously mentioned and discussed, the complete lack of robust bias-free labeled datasets of obfuscated applications left us with the option of manually labeling a small dataset, or collecting unobfuscated applications and obfuscating them ourselves, similarly to what other studies have done. We chose the former as biased data is an issue in the topic of obfuscation. But as the chosen data had to be manually analyzed, the number of samples might be too small to accurately represent the majority of types of applications, as well as the obfuscation techniques found in them. As such, conclusions drawn about the evaluated tools might be based on patterns only present in our dataset, and not representative of 'in-the-wild' applications as a whole.

When adapting the obfuscation detection tools to run on our data, custom implementations of classes and methods were necessary. ObA specifically required more custom implementation, which, while in part documented, some assumptions were made as to how internal components communicated. DLHybrid did not require custom implementation, however due to the lack of documentation, especially regarding component versions, setting up the environment in which the tool was evaluated required making uninformed decisions. The modifications made to ObA, as with any implementation, comes with the risk of human error, a factor that could affect the validity of the findings. DLHybrid made use of several functions and notations which are deprecated in- or removed from- current versions of Python3.12. To avoid completely rewriting the tool, and evaluating it 'as-is' we downgraded the environment versions to what we deemed most likely was originally used. This might have had an effect on the results if some functionality was absent due to version incompatibilities.

Lastly, as DeGuard was unable to process the ProGuard obfuscated application, we were limited to analyzing its performance on a single sample. The performance analyzed could be influenced by things such as structure or size of the APK, or the specific type of renaming performed by Obfuscapk, i.e. it might not be representative of DeGuards performance. Future studies would do well in evaluating similar tools on many more samples.

¹National Security Agency *Ghidra* (2025) <https://github.com/NationalSecurityAgency/ghidra>

6 Conclusion

We have found that the main limitation with obfuscation detection tools is the lack of diverse, bias-free, and labeled training data. Furthermore, circumventing the issue requires either (a) use of unsupervised learning strategies such as anomaly detectors trained on good diverse sets of applications without obfuscation, (b) manually labeling a large number of 'in-the-wild' applications to create a representative dataset which can train supervised learning models, or (c) distilling the specific features that indicate the presence of obfuscation, and differentiates them from 'normal' code. Our findings suggest that (a) and (c) are both effective, as originally proposed by [5].

With the multitude of ways of implementing an application for Android, comes as many different structures and designs of these apps. Some implementation approaches can lead to code appearing to be obfuscated when they are in fact not. For example code generated by the visual development tool MIT App Inventor might look deliberately obfuscated due to every literal being named Lit1, Lit2, ..., LitN, and every instantiation of these being enclosed in separate if-else-blocks leading to a structure similar to flattened code, a form of control flow manipulation. But these are just a side-effect of the code being automatically generated. Similarly, packed applications using modular encryption and reflection classes, or Unity3D apps loading maps and textures from native objects might seem obfuscated, but are structured this way for much different purposes. Detection and de-obfuscation tools need to take the different application structures and multitude of ways of applying obfuscation into account to generalize on applications found 'in-the-wild' or otherwise outside their training datasets. Obfuscation resilient malware detection is a related, and necessary topic with more available tools and comparable amount of research to obfuscation detection. However the detection of malware is usually just the first step, analyzing and understanding it must be done to prevent further evolutions or variants. With how easy it is to repackage an Android application, it is possible to generate multiple variations of pirated apps with malicious inserts [3]. Due to this, further research on obfuscation detection is necessary, specifically detection strategies that consider the issues with lack of robust labeled data and takes steps to actively address them. Furthermore, as the positive effects of de-obfuscation on malware detection have been shown [6], we propose that more research on the topic is needed and that the narrow scope of current de-obfuscators is an issue worth addressing.

References

- [1] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild, 2018.
- [2] Xiaolu Zhang, Frank Breiting, Engelbert Luechinger, and Stephen O’Shaughnessy. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Science International: Digital Investigation*, 39:301285, 2021.
- [3] Wael Elmersy, Ali Feizollah, and Nor Badrul Anuar. The rise of obfuscated Android malware and impacts on detection methods. *PeerJ Computer Science*, 8:e907, 2022.
- [4] Mauro Conti, Vinod Parameswaran Pillai, and Alessio Vitella. Obfuscation detection in android applications using deep learning. *Journal of Information Security and Applications*, 70:103311, 2022.
- [5] Ulf Kargén, Noah Mauthe, and Nahid Shahmehri. Characterizing the use of code obfuscation in malicious and benign android apps. In *The 18th International Conference on Availability, Reliability and Security (ARES 2023)*, page 12, New York, NY, USA, August 2023. ACM.
- [6] Yun-Chung Chen, Hong-Yen Chen, Takeshi Takahashi, Bo Sun, and Tsung-Nan Lin. Impact of code deobfuscation and feature interaction in android malware detection. *IEEE Access*, 9:123208–123219, 2021.
- [7] Michelle Wong and David Lie. Tackling runtime-based obfuscation in android with TIRO. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1247–1262, Baltimore, MD, August 2018. USENIX Association.
- [8] Sidra Siddiqui and Tamim Ahmed Khan. An overview of techniques for obfuscated android malware detection. *SN Computer Science*, 5:328, 2024.
- [9] Shuai Jiang, Yao Hong, Cai Fu, Yekui Qian, and Lansheng Han. Function-level obfuscation detection method based on graph convolutional networks. *Journal of Information Security and Applications*, 61:102953, 2021.
- [10] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, and Francesco Mercaudo. Detection of obfuscation techniques in android applications. In *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES ’18*, New York, NY, USA, 2018. Association for Computing Machinery.
- [11] Omid Mirzaei, Jose Maria de Fuentes, Juan Tapiador, and Lorena Gonzalez-Manzano. Androdet: An adaptive android obfuscation detector. *Future Generation Computer Systems*, 90:240–261, 2019.

- [12] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 343–355, New York, NY, USA, 2016. Association for Computing Machinery.
- [13] Geunha You, Gyoosik Kim, Sangchul Han, Minkyu Park, and Seong-je Cho. Deoptfuscator: Defeating advanced control-flow obfuscation using android runtime (art). *IEEE Access*, 10:61426–61440, 2022.
- [14] Richard Baumann, Mykolai Protsenko, and Tilo Müller. Anti-proguard: Towards automated deobfuscation of android apps. In *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems, SHCIS '17*, page 7–12, New York, NY, USA, 2017. Association for Computing Machinery.
- [15] Noah Mauthe, Ulf Kargén, and Nahid Shahmehri. A large-scale empirical study of android app decompilation. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 400–410, 2021.
- [16] Kevin Allix, Tegawendé Francois Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16*, pages 468–471, New York, NY, USA, 2016. ACM.
- [17] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. Obfuscapk: An open-source black-box obfuscation tool for android apps. *SoftwareX*, 11:100403, 2020.

Appendix A

ObA Virtual environment:

```
androguard @ git+https://github.com/androguard/androguard.  
  git@5ddede4ef991eb1614b69c82aa205332451d7697  
API @ git+https://github.com/NoahMauthe/APIs.  
  git@6dc8a4d535ebdcb37b9706b51ccb6a4d9f53a07f  
  
absl-py==2.2.2  
apkid==2.1.5  
asn1crypto==1.5.1  
asttokens==3.0.0  
astunparse==1.6.3  
certifi==2025.1.31  
cffi==1.17.1  
charset==5.2.0  
charset-normalizer==3.4.1  
click==8.1.8  
colorama==0.4.6  
contourpy==1.3.2  
cryptography==44.0.2  
cyclor==0.12.1  
decorator==5.2.1  
executing==2.2.0  
ezodf==0.3.2  
flatbuffers==25.2.10  
fonttools==4.57.0  
gast==0.6.0  
google-pasta==0.2.0  
grpcio==1.71.0  
h5py==3.13.0  
idna==3.10  
ipython==9.1.0  
ipython_pygments_lexers==1.1.1  
jedi==0.19.2  
joblib==1.4.2  
keras==2.11.0  
kiwisolver==1.4.8  
libclang==18.1.1  
lxml==5.3.2  
Markdown==3.8  
markdown-it-py==3.0.0  
MarkupSafe==3.0.2  
matplotlib==3.10.1  
matplotlib-inline==0.1.7  
mdurl==0.1.2  
ml_dtypes==0.5.1  
namex==0.0.9  
networkx==3.4.2  
numpy==1.26.4  
opt_einsum==3.4.0  
optree==0.15.0  
packaging==25.0  
pandas==2.2.3  
parso==0.8.4  
pexpect==4.9.0  
pillow==11.2.1  
prompt_toolkit==3.0.51  
protobuf==5.29.4  
psycopg2-binary==2.9.10  
ptyprocess==0.7.0  
pure_eval==0.2.3  
pycparser==2.22  
pydot==3.0.4  
Pygments==2.19.1  
pyparsing==3.2.3  
python-dateutil==2.9.0.post0  
python-magic==0.4.27  
pytz==2025.2  
requests==2.32.3  
rich==14.0.0  
scikit-learn==1.3.2  
scipy==1.15.2  
setuptools==79.0.0  
six==1.17.0  
stack-data==0.6.3  
tensorboard==2.19.0  
tensorboard-data-server==0.7.2  
termcolor==3.0.1  
threadpoolctl==3.6.0  
toml==0.10.2  
tqdm==4.66.6  
traitlets==5.14.3  
typing_extensions==4.13.2  
tzdata==2025.2  
urllib3==2.4.0  
wcwidth==0.2.13  
Werkzeug==3.1.3  
wheel==0.45.1  
wrapt==1.17.2  
yara-python-dex==1.0.7
```

DLHybrid Virtual environment:

```
absl-py==2.2.2
alembic==1.15.2
androguard==4.1.3
apkInspector==1.3.3
asn1crypto==1.5.1
asttokens==3.0.0
astunparse==1.6.3
banal==1.0.6
cachetools==5.5.2
certifi==2025.4.26
cffi==1.17.1
charset-normalizer==3.4.1
click==8.1.8
colorama==0.4.6
contourpy==1.3.0
cryptography==44.0.2
cyclers==0.12.1
dataset==1.6.2
decorator==5.2.1
exceptiongroup==1.2.2
executing==2.2.0
flatbuffers==25.2.10
fonttools==4.57.0
frida==16.7.14
gast==0.4.0
google-auth==2.39.0
google-auth-oauthlib==0.4.6
google-pasta==0.2.0
greenlet==3.2.1
grpcio==1.71.0
h5py==3.13.0
idna==3.10
importlib_metadata==8.6.1
importlib_resources==6.5.2
ipython==8.18.1
jedi==0.19.2
joblib==1.4.2
keras==2.11.0
kiwisolver==1.4.7
libclang==18.1.1
loguru==0.7.3
lxml==5.4.0
Mako==1.3.10
Markdown==3.8
MarkupSafe==3.0.2
matplotlib==3.9.4
matplotlib-inline==0.1.7
mutf8==1.0.6
networkx==3.2.1
numpy==1.23.0
oauthlib==3.2.2
opt_einsum==3.4.0
packaging==25.0
pandas==2.2.3
parso==0.8.4
pexpect==4.9.0
pillow==11.2.1
prompt_toolkit==3.0.51
protobuf==3.19.6
ptyprocess==0.7.0
pure_eval==0.2.3
pyasn1==0.6.1
pyasn1_modules==0.4.2
pyparser==2.22
pydot==3.0.4
Pymments==2.19.1
pyparsing==3.2.3
python-dateutil==2.9.0.post0
pytz==2025.2
PyYAML==6.0.2
requests==2.32.3
requests-oauthlib==2.0.0
rsa==4.9.1
scikit-learn==1.6.1
scipy==1.13.1
six==1.17.0
SQLAlchemy==1.4.54
stack-data==0.6.3
tensorboard==2.11.2
tensorboard-data-server==0.6.1
tensorboard-plugin-wit==1.8.1
tensorflow==2.11.0
tensorflow-estimator==2.11.0
tensorflow-io-gcs-filesystem
  ==0.37.1
termcolor==3.0.1
threadpoolctl==3.6.0
tqdm==4.67.1
traitlets==5.14.3
typing_extensions==4.13.2
tzdata==2025.2
urllib3==2.4.0
wcwidth==0.2.13
Werkzeug==3.1.3
wrapt==1.17.2
zipp==3.21.0
```




UMEÅ UNIVERSITY