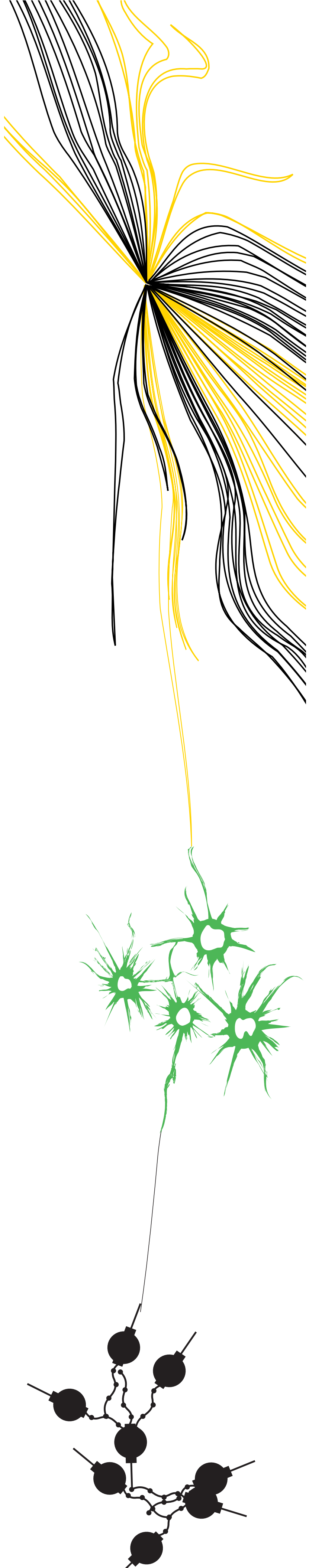

April 2021

MASTER THESIS

DEOBFUSCATING THIRD PARTY LIBRARIES IN ANDROID APPLICATIONS USING LIBRARY DETECTION TOOLS

A. S. Hoekstra

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE (EEMCS)
SERVICES AND CYBERSECURITY GROUP (SCS)

Exam Committee
Dr. Andreas Peter
Dr. Mannes Poel
MSc. Thijs van Ede

UNIVERSITY OF TWENTE.

Deobfuscating Third Party Libraries in Android Applications using Library Detection Tools

A. S. Hoekstra
University of Twente
Enschede, Netherlands
ae.s.hoekstra@student.utwente.nl

Abstract

Nowadays there is a mobile application for almost everything. Adversaries can create malicious applications and hide their intent by obfuscating them. Obfuscation makes applications hard to analyse. In this research we focus on deobfuscating third party libraries in Android applications. More specifically, we show how off-the-shelf library detection tools intended for identifying third party libraries, can be used to deobfuscate those libraries. We achieve this by modifying and extending the output of those tools such that the result can be applied as a valid deobfuscating transformation on Android apps. We compare this method against DeGuard, an existing deobfuscation method based on a statistical model. We create a novel data set of Android applications and third party libraries and use it to evaluate both approaches. We find that our method predicts fewer identifiers than the existing method. However, our approach is less computationally expensive and has a higher accuracy. This research focuses on tackling identifier obfuscation. Future research is needed to adapt and evaluate the developed method on more thorough obfuscation types such as class repackaging.

1 INTRODUCTION

In recent years there has been a surge in smartphone applications. These mobile applications have become an essential part of a business's success; they strengthen competitive advantage by increasing brand loyalty and increase customer spending by providing convenience [18]. It is therefore desirable to keep information about the inner workings of an application secret. An easy strategy to keep application information secret is to not publish the source code of the application. However, leaving the source code unpublished might not be enough as the binary code of applications can be reverse engineered. To make the reverse engineering process more difficult, it is almost standard practice to apply some form of obfuscation before publishing an application. Moreover, obfuscation has not only become standard practice in benign applications, adversaries increasingly use obfuscation techniques in order to hide activities in their malicious applications. As a result, malware analysts and law enforcement agencies demand better methods of reversing the obfuscation process. With the aim of assisting these parties, our work focuses on creating more advanced methods of automated deobfuscation.

The goal of binary reverse engineering is to make human readable source code from obfuscated machine code. Obfuscated machine code is made by first applying a compilation step on the source code, and then an obfuscation step on the compiled machine code. There are different types of obfuscation that can be used in the obfuscation step. To turn obfuscated machine code back into human readable source code, we first need to apply a deobfuscation step on the machine code, and then a decompilation step on the deobfuscated machine code. Decompilation tools are readily available for most programming languages and usually only one decompiler is needed to obtain source code for a program. Deobfuscation tools on the other hand, usually implement deobfuscation for only one type of obfuscation as there are too many variants to cover at once. As a consequence, to fully deobfuscate a program obfuscated with multiple obfuscation techniques, multiple deobfuscators need to be used.

For Android, creating obfuscated machine code follows almost the same steps as for any other compiled language. Because most Android applications are written in Java, we focus on Java in this paper. First, the Java source code is compiled using the Java compiler, thereafter, the compiled class files are obfuscated by an obfuscator. Additionally, the resulting class files are translated to the dex format which is used by the Android runtime. ProGuard has been the go-to obfuscation tool used for the obfuscation step in the Android build process since the beginning of Android [17]. ProGuard is an open source tool that can be used for layout obfuscation of Java bytecode. To this end, it renames identifiers in the Java bytecode to short meaningless strings. This technique is fundamental but highly effective. Apart from layout obfuscation, ProGuard also provides a whole range of bytecode optimisations such as dead code removal. In 2019, ProGuard has been replaced by R8 as the default obfuscation and optimisation tool but ProGuard is still widely used. R8 provides the same functionality as ProGuard but has tighter integration into the Android build process [4]. There exist other tools apart from the ones provided in the Android toolkit. One of these is DexGuard, which provides more thorough obfuscation such as control flow and data obfuscation.

In order to properly reverse engineer Android applications, the identifier obfuscation applied by ProGuard needs to be reversed. It has been shown that meaningful identifier names influence how quick source code can be understood [5, 10]. Good layout deobfuscation is therefore of great importance for reverse engineering. Reversing identifier deobfuscation is a non-trivial problem as app developers can use arbitrary names for identifiers. Once these identifier names are gone, they cannot be recovered as identifier obfuscation is a one-way transformation. However, it is possible to predict the original names. The state-of-the-art for predicting identifier names in Android applications is DeGuard [6]. DeGuard is a statistical model that makes identifier predictions using known identifier names from a large dataset of Android applications. It is important to note that DeGuard regards all identifiers in Android applications the same.

In reality, a distinction can be made between identifiers from third party libraries, and identifiers from code that is specifically written for one application. This distinction can be made because identifiers from third party libraries can often be found in public code repositories. Therefore, identifiers from third party libraries can be compared to the public ones and be identified. We think this distinction can be exploited to make better predictions for application specific code.

2 RELATED WORK

2.1 Obfuscation

Collberg et al. have classified obfuscation techniques into four main categories; layout obfuscation, data obfuscation, control obfuscation, and preventive obfuscation [7]. Layout obfuscation is a fundamental form of obfuscation and is therefore used in almost every obfuscator. It renames program identifiers to meaningless strings and removes information that is unnecessary for running the program such as comments and debug information. Data obfuscation obscures the data items present in a program by changing how they are stored, encoded, or ordered. For instance, this can be done by choosing unusual data formats, or by reordering data in arrays using a mapping [19]. Control obfuscation applies transformations that alter the control flow of an application with the goal to make the control flow much harder to analyse. This can be done by, for example, breaking up a program into basic blocks and executing them indirectly via a switch statement or by directing control flow with exceptions [13, 16]. Preventive obfuscation exploits tool and platform specific weaknesses to prevent deobfuscation. An example of this is a trick that crashed the Mocha decompiler by inserting extra instructions after every JVM return instruction [12].

2.2 Deobfuscation

Rule-based

Early Java deobfuscation tools such as Java Deobfuscator and Enigma use rules to rename obfuscated identifiers [8, 11]. These rules are hand coded and can assign only generic names to program elements. For instance, the name `target` might be assigned to a method argument when that method modifies the argument. Because rule-based approaches can only assign generic names, they do not give much insight in code that contains a lot of context specific identifiers.

JSNice

V. Raychev et al. developed JSNice, a model that predicts the identifier names of obfuscated JavaScript programs using conditional random fields (CRF) and Maximum a posteriori estimation (MAP) [15]. JSNice takes all variables from an obfuscated JavaScript program and their relations and constructs a dependency network. The variable names are predicted simultaneously using MAP inference. MAP inference is used to get the most likely combinations of variable names. The CRF model is trained using dependency networks with known variable names and learning common combinations of variable names. JSNice was trained on a corpus of 324,501 JavaScript files and achieved an accuracy of 63.4% when predicting variable names.

DeGuard

Bichsel et al. developed DeGuard, a statistical model based on conditional random fields (CRF) that utilises MAP inference to predict the original names of identifiers in Android applications [6]. DeGuard builds upon JSNice and uses the same techniques but for Android applications instead of JavaScript. The main contributions of their research are a way of representing Android program code as a CRF and a probabilistic approach to predict identifiers using CRFs. In the model, identifiers and constants are represented as nodes in a CRF, and relations between the identifiers and constants are represented as edges. Bichsel et al. have evaluated DeGuard with the publicly available source code of 1784 Android applications from the F-Droid app store. It was found that DeGuard can retrieve 80.6% of the original identifier names when predicting unseen applications. A limitation of the approach is that CRF's can only predict combinations of values from the training data. They are also computationally expensive and do not scale well. This is especially a problem for DeGuard when using large amounts of training data.

A shortcoming of the research of Bichsel et al. is that DeGuard predicts all the identifiers in the same way. However, the identifiers in Android applications can be split into identifiers from third party libraries and identifiers specific for the application. There exist tools that can identify which libraries are used in Android applications. Our research builds further on the research Bichsel et al. and looks at predicting third party library identifiers that leverages these existing tools and in a way that is not limited by the amount of training data. This is further discusses in the Scientific Contribution.

Code2Vec

Alon et al. have researched a method of capturing program context by creating an encoding that uses paths within abstract syntax trees (AST) [2]. They show how to construct such AST paths and how they can be used as input for neural network models. An AST path is defined as a triple, containing two AST leaf nodes and a sequence of tokens along the path that has to be traversed to get from one leaf node to the other. Alon et al. have examined the usefulness of AST paths for three problems; method name prediction, variable name prediction, and type prediction. For each of these tasks they trained two models, a CRF model and a word2vec-based model. The evaluation shows promising results that AST paths can capture context well.

Alon et al. subsequently developed Code2Vec, a model that assigns different but close vectors to similar snippets of code, capturing subtle differences between snippets [3]. These vectors, called code embeddings, are continuous distributed vector representations for snippets of code. The vectors are generated by a neural network trained to combine AST paths for a snippet of code. Alon et al. showcase Code2Vec by training another model for suggesting function names that inputs code program entities and is therefore not directly applicable to deobfuscation tasks. However, this research shows a promising technique to encode program context which can also be useful for deobfuscation.

Allamanis

Allamanis et al. use a graph based neural network approach for two related tasks; variable naming and variable misuse detection [1]. They outline how to represent programs as graphs, where nodes represent program tokens and edges semantic and syntactic relations between those tokens. The outlined representation is suitable for use with gated graph neural networks (GGNN). GGNNs are an optimised version of graph neural networks (GNN) [14]. The resulting model does however perform relatively low on the variable naming task with an accuracy of 44.0%. This research is also not directly applicable to deobfuscation. Nevertheless, it shows an interesting method of representing program context, such as data flow, and how to learn it.

2.3 Library detection

Library detection is the detection of which third party libraries are included in applications. Library detection is made difficult by code obfuscation and shrinking. Identifier obfuscation makes it impossible for library detection tools to detect libraries based on identifier names. Therefore, library detection tools use features from the code structure to identify libraries. This includes features such as call graphs to standard library methods, and bytecode instructions. Optimisations and code shrinking make it hard to match features from the code inside applications with the original library code. For example, stripping unused methods and reordering instructions make it harder to make exact matches.

Library detection is commonly resilient against identifier obfuscation as this is a common obfuscation technique. Not all library detection tools are resilient against class repackaging. Class repackaging is a Java obfuscation technique that puts classes in the same package where possible. Library detection tools frequently use the package hierarchy in Java applications to boost the confidence in their predictions.

In this research we leverage library detection tools to deobfuscate third party libraries. For this research we selected LibPecker as a library detection tool [20]. LibPecker has as byproduct class mappings which makes using the tool suitable for use in deobfuscation. This tool has the additional advantage that it is resilient against identifier obfuscation, and in lesser extent also to class repackaging.

2.4 Scientific contribution

As discussed above, current research focuses predicting identifier names in Android applications. Alon et al. developed Code2Vec to predict names for identifiers, and Bichsel et al. built DeGuard to predict the most likely combination of identifier names for an obfuscated application. These state-of-the-art Android prediction tools make identifier predictions without regard to the type of code. In other words, these tools do not make a distinction between code from third party libraries and code that is specifically written for an application.

It is difficult to predict the names of obfuscated identifiers for code that is written specifically for an application because the combination of those identifiers is likely to be unique to that application. For third party libraries this is easier. Many third party libraries that are included in Android applications are publicly accessible online. Obfuscated code from third party libraries can therefore be matched with the unobfuscated public version in order to recover original identifier names.

The state-of-the-art Android prediction does not make a distinction between third party library code and application specific code. Furthermore, there has not been published research yet about the potential benefits of predicting third party library code separately. However, we expect that predicting library code separately could have benefits. This means that there is a scientific knowledge gap about the potential benefits of predicting third party library code separately. The goal of this research is to make a contribution to bridge this knowledge gap.

Identifier prediction methods often use references to the Android standard libraries as features because these references cannot be obfuscated. When third party library code is first deobfuscated, references to that code can also be used as features. This could be useful for applying prediction methods that work only on unobfuscated code to obfuscated code. These third party library predictions could also be very beneficial if used in combination with existing tools such as DeGuard, especially when the library predictions have a high accuracy. When this is the case, they can be used as a starting point to make predictions about the application specific code.

This research focuses on leveraging existing tools that identify libraries, for deobfuscation and comparing this new method with existing state-of-the-art deobfuscation tools. This helps to fill part of the knowledge gap about predicting third party libraries separately from other application code and therefore helps to achieve an overall better understanding of deobfuscation.

2.5 Engineering contribution

All research on Android deobfuscation and library detection relies on a data set to validate the approach against. The usual approach in previous research was to download all apps from the F-Droid app store. The F-Droid app store was chosen often, because everyone is free to download the APK files of the listed apps. This is not allowed for other app stores such as Google Play and the Amazon Appstore.

A part of the applications on all app store is obfuscated. What researchers previously did, was filtering out the obfuscated applications and keeping only the unobfuscated applications. This has the disadvantage that the data set becomes much smaller. It also introduces some questions to validity, as newer and larger apps may be obfuscated more often than older and smaller apps. Obfuscation is in newer Android build tools enabled by default, and apps become smaller when applying obfuscation.

Developers publish their apps in the F-Droid app store by submitting a link to source code of the app. F-Droid will then built the app from the source code using a custom build environment. In this research we developed a generic build step that disables all obfuscation options.

Library detection tools can identify which third party libraries are included in an application. They do not provide an exact mapping which can be used to deobfuscate the application, i.e. replace all obfuscated identifiers with their original name. Although library detection tools do not create application mappings, they can be used to create them.

An application mapping contains for every identifier in an application another identifier. There are some challenges in creating application mappings using library detection tools. The main challenge is the validity of the application mapping. There are constraints to identifiers in Android applications. For instance, a simple constraint is that there cannot be two classes with the same qualified name. The challenge is to select the results from the library detection tools in such a way that all naming constraints are satisfied. In our research we line out how this can be achieved.

Library detection tools only give an indication which classes are used. This can be used to deobfuscate package and class identifiers. However, library detection tools cannot be used as input for deobfuscating method identifiers. Method identifiers can be deobfuscated by matching the methods between the original library class and the library class contained within the application. We show how this can be done using a signature based approach. This approach is further described in the Methodology section. As is the case with package and class identifiers, there are also naming constraints for methods. We identify these constraints and show how method identifiers can be added into a mapping without violating the constraints.

The method developed in this research can be used to gain a better understanding of how to improve deobfuscation tools.

3 METHODOLOGY

The state-of-the-art Android deobfuscation tools use statistical models to predict identifier names. DeGuard is a tool that predicts the names of obfuscated identifiers based on several features including their relation to known identifiers, namely identifiers from the Android standard library [6]. The goal of this research is to improve the state-of-the-art Android deobfuscation tools. More specifically, this research approaches Android deobfuscation by predicting third party libraries separate from the application specific code.

Third party libraries are publicly available reusable software components, that developers can use in their applications. Third party libraries are not easily recognisable anymore in obfuscated applications, because identifiers from third party libraries can be obfuscated (opposed to identifiers from the standard library). However, there are specialised tools that can identify which libraries are used in an obfuscated application. These tools often work with matching the application against a database of libraries using heuristics.

Although, there are tools that can detect libraries, some additional steps are required to create a valid mapping from obfuscated application identifiers to the their original names.

3.1 System overview

Figure 2 gives an overview of the system. We call this system Delibird (DEobfuscating LIBraries In anRDroid apps). Delibird uses an off-the-shelf library detection tool to detect libraries. A library detection tools can be used to identify which libraries are used in an application. For this research, we use LibPecker as library detection tool [20], see step (2) in Figure 2. LibPecker produces a class mapping. The class mappings are verified and merged into a single application mapping. In addition, methods are detected in the input APK using the class mapping. The produced application mapping is then applied to the input application.

3.2 Library deobfuscation

To deobfuscate the library part of an application we first detect libraries using LibPecker, an open-source library detection tool. Given an application and a library profile contain-

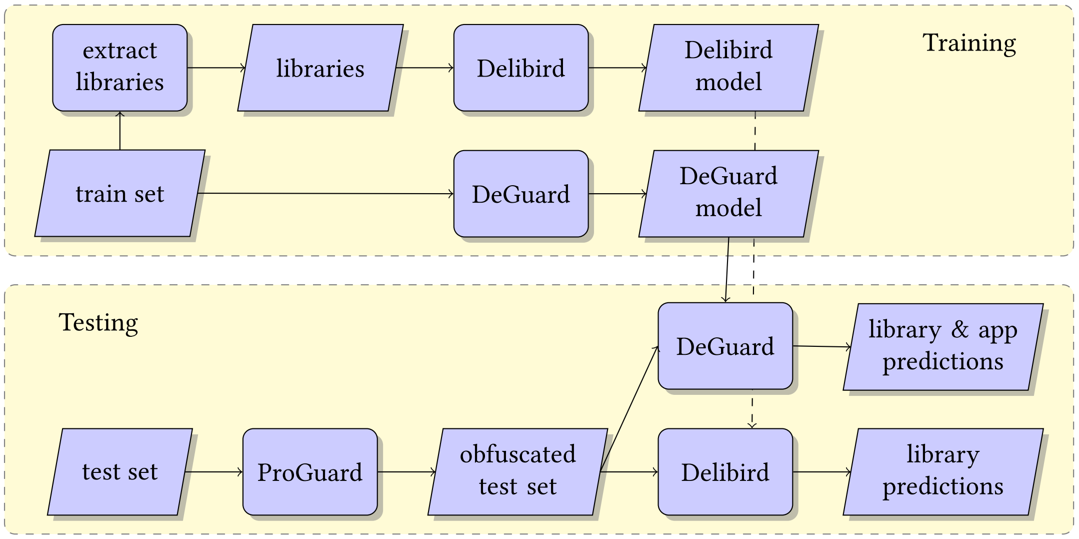

Figure 1: *Training* First third party libraries are extracted from the training applications. Then Delibird is trained using the resulting libraries. DeGuard is directly trained with the training applications. The resulting models are used in the testing phase. *Testing* The test set of unobfuscated applications is first obfuscated with ProGuard. The same obfuscated applications are then predicted with DeGuard and Delibird. DeGuard predicts both library identifiers and application specific identifiers. Delibird predicts only library identifiers. These predictions are compared with the original identifiers from the applications in the test set.

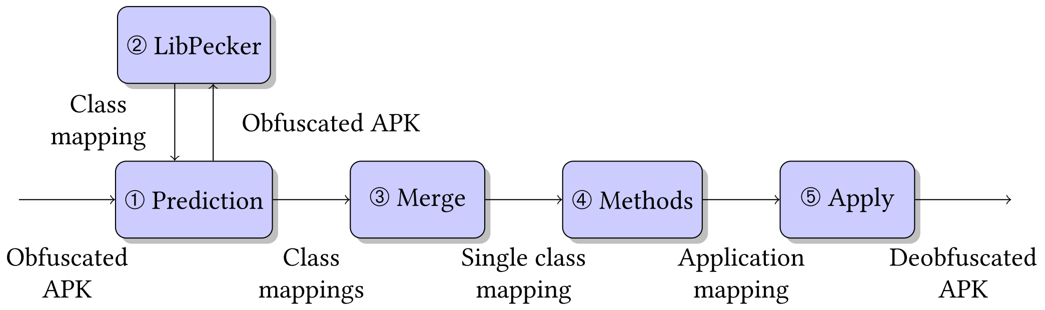

Figure 2: This figure gives an overview of the system. The first step (1) is a prediction step. This step takes an obfuscated APK and a set of library templates. These library templates are generated during the training phase of LibPecker. A library template contains feature of the library which can be used to detect that library in an APK. For each of the templates, LibPecker (2) is invoked and the class mappings produced by LibPecker are collected. In the next step (3), the class mappings are validated and merged into a single class mapping. Then using this class mapping, methods are detected in the matched classes and added to the mapping (4). Finally, the mapping is applied to the input application (5).

ing features of that library, LibPecker gives a similarity score and a mapping from classes in the app to classes from the detected library. Given a set of library profiles and an application, we get a set of class mappings per application. We use these mappings to rename classes. To rename methods in the classes, we use method signatures to match between the class methods with obfuscated names and the methods in the libraries with the original names. We will now elaborate on how exactly the classes and methods are matched and renamed.

3.3 Naming constraints

We cannot simply take the class mappings produced by LibPecker and apply them to our application because there are restrictions on what names identifiers are allowed to have. There are three constraints for renaming classes and packages; 1) there should not be subpackages with duplicate names in a package, 2) there should not be classes with duplicate names in a package, and 3) classes and subpackages should remain in the same package.

The first two naming constraints are needed to prevent invalid byte code. For example if we have two qualified classes “a.b.C” and “a.b.D” and their class names are both renamed to “SomeClass”, we end up with “a.b.SomeClass” and “a.b.SomeClass” respectively. This will confuse the Android runtime as classes have to be uniquely identifiable. This situation happens when two classes look a lot like each other in terms of functionality.

A similar situation arises when two subpackages are renamed to the same name. Suppose we have the qualified classes “a.b.C” and “a.d.C” and subpackages "b” and “d” are both renamed to “somepackage”. This results in the qualified classes “a.somepackage.C” and “a.somepackage.C” respectively. These kind of situations are prevented by constraint 2. The third constraint is needed to keep the package structure of the application intact.

3.4 Class deobfuscation

Because LibPecker is a library detection tool, it creates class mappings per detected library. The class mappings that are produced may not follow the naming constraints described in the previous section. For example, if two classes in the same package are predicted to be in packages from different libraries, constraint 3 is violated.

To merge the individual class mappings produced by LibPecker into a final mapping, the individual class mappings are first sorted by similarity score. Then the class mappings are added one by one to the final class mapping starting with the highest scoring ones. Before adding a class mapping to the final mapping, it is checked if adding this class mapping would cause a naming violation in the final mapping. If this is the case, the class mapping is ignored. The algorithm to merge the class mappings is listed in Algorithm 1.

Algorithm 1 Merging class mapping

```
1: function MERGE($mappings, classes$)
2:     result ← ∅
3:     packageMapping ← ∅
4:     outer:
5:     for all $(old, new) \in mappings$ do
6:         if $old$ already mapped $\vee$ $new$ already mapped to then
7:             continue
8:         end if
9:         $oldPackages \leftarrow Packages(old)$
10:        $newPackages \leftarrow Packages(new)$
11:        if length($oldPackages$) $\neq$ length($newPackages$) then
12:            continue
13:        end if
14:        for all $(os, ns) \in$ ... do
15:            if $os \in packageMapping$ then
16:                if $packageMapping(os) \neq ns$ then
17:                    continue outer
18:                end if
19:            else
20:                $packageMapping(os) \leftarrow ns$
21:            end if
22:        end for
23:        $result(old) \leftarrow new$
24:    end for
25:    for all $c \in classes$ do
26:        if $c \notin result$ then
27:            $newClass \leftarrow c$
28:            for all $p \in Packages(c)$ do
29:                if $p \notin packageMapping$ then
30:                    break
31:                end if
32:                $newClass \leftarrow packageMapping(p) \parallel$
                       $c.substring(length(p), length(c))$
33:            end for
34:            $result(c) \leftarrow newClass$
35:        end if
36:    end for
37:    return result
38: end function
```

The first check that is done to see if adding the class mapping would cause a naming constraint violation, is checking if the class is already mapped. This situation is likely to occur when there are multiple versions of the same library. In two subsequent minor versions of a library there are many similar identifiers because in subsequent versions often contain only small code changes. This can cause an identifier to be identified in two different versions of the same library. In that case the class is already mapped, the mapping is discarded. The second thing that is checked, is whether the mapping would rename a subpackage that is already renamed by an earlier class mapping to another name. In that case the mapping is also discarded.

After the class mappings produced by LibPecker have been merged into the final mapping, it is possible that a subpackage is renamed by class mapping. If that is the case, all classes that are contained in that subpackage, need to be mapped to that subpackage by renaming them. These class mappings are also added to the final mapping.

3.5 Method deobfuscation

We identify methods in the deobfuscated classes by matching them against methods from the library classes using the class mappings from LibPecker. We use a simple signature based approach to match the methods.

There are two constraints for renaming methods; 1) two obfuscated methods with the same parameter types should not be assigned the same name, and 2) overridden methods should stay overridden, and non-overriding methods should not become overriding. In Java, an overriding method is a public non-static method that has the same signature as a public non-static method in its parent class. When such a method is called, the method in the parent class is hidden, and the method in the child class is called [9].

Algorithm 2 Method signatures

```
1: function GetSignature($method$)
2:     signature ← ""
3:     types ← method.returnType $\cup$ method.parameterTypes
4:     for all type $\in$ types do
5:         baseType ← BaseType(type)
6:         arrayDimension ← ArrayDimension(type)
7:         if IsPrimitive(baseType) $\vee$
                   IsFromStandardLibrary(baseType) then
8:             signature ← signature || baseType || arrayDimension
9:         else
10:            signature ← signature || "*" || arrayDimension
11:        end if
12:    end for
13:    return signature
14: end function
```

The first step in matching the methods from a class in the application, with the methods from a class in the library, is to select all non-overriding methods from both classes. We do not select overriding methods because if we would rename such a method, we would also have to rename to original method in the parent class. This could interfere with renaming methods in the parent class. Renaming overriding methods in parent classes could potentially increase performance in case the overridden method could not be identified in the parent class. However, this significantly increases the complexity of identifying methods.

For all selected methods in both classes we generate a signature that we later use to create mapping from the methods in the app class to the methods in the library class. How this signature is generated, is specified in Algorithm 2. The signature is based on return type and the parameter types of a method. For every type it is checked if it is a primitive type or a type from the standard. If this is the case, the type is added to the signature. Otherwise, a wildcard is added to the signature. The reason for adding a wildcard for the other types is that those can be obfuscated. All of the types and wildcards plus their possible array dimensions are concatenated as the signature.

After generating the signatures, all signatures that have a one-to-one mapping from the app class to a signature from the library class methods, are selected. The other methods cannot be uniquely mapped and are therefore discarded. If there are two methods from the app class that map to two library functions that have the same name, constraint 1 is violated. The mappings of all such combination of methods are therefore discarded. For example, if there is an application method `int a(int)` which could map to either library method `int b(int)` or `int c(int)`, it is discarded.

All mappings that assign an app method the same name as a public method with same signature in one of the parent classes are discarded, because this would violate constraint 2. For example, if there is a class A with the method `int a()`, the method `int b()` in class B that extends class A, cannot be renamed to "a" because this would override a method that was not overridden before.

The remaining methods are renamed according to the mappings. All methods that override one of the mapped methods are renamed as well to the new name of the method they override.

4 DATASET

4.1 Apps

To train our system we needed a dataset of unobfuscated Android applications and the third party libraries they contain. To our best knowledge, no such dataset exists. Common Android application datasets contain applications from sources such as Google Play and the Amazon App store. However, applications in these app stores are often obfuscated, besides, it is difficult to determine exactly which third party libraries an application uses without looking at the application source code.

We therefore used the F-Droid app store as a source for our dataset. All of the applications in the F-Droid app store are open source as required by the F-Droid policy. Although the apps from the F-Droid source are open source, the compiled apps that are published in the store can still be obfuscated. This can be resolved by compiling the apps while disabling obfuscation in the build process. The apps from the F-Droid app store can be built from source code using the F-Droid build tools. To remove any obfuscation applied during the build process, we patched the F-Droid build server. We added code that scans for ProGuard configurations and adds a rule that disables the obfuscation functionality of ProGuard. We added support for modifying Ant, Gradle, and Maven configurations. Our modification to the F-Droid build server does not disable ProGuard entirely as this would also disable all bytecode optimisations, which are used frequently. Disabling ProGuard entirely would therefore result in a dataset that is not representative for apps from other app stores. Our dataset contains 1511 applications.

4.2 Obfuscated apps

To test the library deobfuscation and DeGuard modules, we need obfuscated versions of the apps. Android applications are compiled in two steps; first the Java source code is compiled into Java bytecode, and then the Java bytecode is converted to Dex bytecode. Dex is a bytecode format for the Android Runtime. Usually, the obfuscation is done on the Java bytecode right before converting it to dex because ProGuard only takes Java bytecode as input. How an application is exactly obfuscated is specified in the build configuration of the application. To obfuscate the apps in our dataset in the usual way, we would have to manually modify all build configurations. We therefore decided to separate the obfuscation of the apps from the compilation of the apps by obfuscating the apps after compiling them.

We achieved this by converting the dex files inside the apps to jar files using dex2jar. We then ran ProGuard on the jar files with a custom ProGuard configuration that enabled only obfuscation and no bytecode optimisations. Finally, we converted the obfuscated jar files back to dex using the standard dex tool provided by the Android toolchain.

4.3 Third party libraries

We needed a set of libraries to train the library detection module of our system. By far the most common way of obtaining libraries is by downloading them from public source code repositories. Downloading all libraries from several online code repositories is not very efficient, because not all libraries are used frequently or are useful for Android apps. Instead, we use all libraries included in the apps as a representative dataset. It is very likely that commonly used libraries are both in the training data as in the test data. This is the underlying assumption of this study. To be able to test if this assumption holds, it is important that only third party libraries corresponding to the apps in the training set are used during the training phase later on. It is important to not use the identified libraries from the test set for training, as otherwise this assumption cannot be verified.

To get the libraries that are used in an app, we inspect the build configurations in the source code of the apps, and download the libraries listed there from online code repositories. A build configuration specifies how an app should be build and among others what libraries the app requires. For the Gradle and Maven build configurations, we use the Gradle and Maven dependency tools respectively to list the libraries and also the library dependencies of those libraries. Because Ant has no dependency management system, and not all libraries are available in source code repositories, there are some apps with hardcoded libraries in their source code. So in addition, we extract all hardcoded libraries from the source files. We achieve this by extracting all jar files containing Java bytecode from the apps.

We cross reference the class names from all the libraries with the class names present in the apps to filter out libraries that are not actually used in the apps such as testing frameworks. We keep a reference of which app classes are originally from a library to be able to make a separation in the prediction results between library classes and non-library classes later on in the evaluation.

4.4 Dataset validation

It is important that our dataset does not contain any obfuscated apps. Any obfuscated identifiers left in the apps add noise to the dataset which can impact the performance of identifier prediction systems. To verify that there were no obfuscated identifiers left in our created dataset, we developed a heuristic that detects obfuscated identifiers.

The heuristic we developed exploits the naming artifacts introduced by ProGuard during obfuscation. When ProGuard obfuscates identifiers, it replaces them by default with a short name according to a predictable naming scheme. The default naming scheme that ProGuard uses is alphabetic; the first identifier that is being obfuscated is renamed to ‘a’, the second to ‘b’ and so on. Because this naming scheme generates identifier names that are used rarely in source code, it is easy to distinguish them from non-obfuscated identifiers.

The heuristics first checks if there is a root package in an application called ‘a’. If there is such as package, we directly assume that the application is (partly) obfuscated. We have found no evidence in the source code of the apps in our dataset of root packages that have such a name. The second thing that is checked is the presence of at least the classes named ‘a’, ’b’, ‘c’, ‘d’, and ‘c’ in any package. If this is the case we also assume the app is obfuscated. The last thing the heuristic checks is if there is a class that contains either methods or fields named ‘a’, ‘b’, ‘c’, ‘d’, and ‘e’. If at least one of these checks passes, we label the application as obfuscated, and unobfuscated otherwise.

We designed this heuristic to minimize the number of false negatives, apps that are wrongly labeled as unobfuscated, while keeping the number of false positives relatively low. Because this heuristic is quite strict, only apps with very little obfuscation remain undetected. This is not really a problem because apps with only a few obfuscated classes have little impact on the quality of the DeGuard model. The number of false positives this heuristic creates is low, as developers rarely use the same names that are also part of the naming scheme of ProGuard.

We found that this heuristic works well within our dataset, we selected a set of 100 apps downloaded from F-droid and labelled the apps as obfuscated or not by hand and tested the heuristic. This resulted in only 3 false positives and no false negatives.

5 EVALUATION

In this section we will evaluate how the research is implemented. The goal of this research is to see how library detection based deobfuscation compares to DeGuard in predicting identifiers from third party libraries. For this research we implemented both a library detection based deobfuscation system and the DeGuard deobfuscation system which is based on statistical learning. Figure 1 contains an overview of the evaluation setup.

5.1 Dataset

To compare both systems, we have evaluated them on the same dataset of obfuscated applications. This was required to test DeGuard on a new dataset as the original paper of DeGuard did not publish their dataset and did not release raw results. Therefore, it was not possible to use the results from the DeGuard paper for this research. The dataset was split into a training and a test set using a 90/10 split. Our training set contains 1361 samples, and our test set contains 150 samples. We did not use 10-fold cross validation, similar to DeGuard, as the running time of DeGuard is too long to perform within the time constraints of this research.

5.2 Training

Delibird was trained on the libraries from the apps in the training set. Using our implementation of DeGuard we extracted the features from the apps in the training set. DeGuard uses the Nice2Predict tool to train the model from the features [15]. Training with Nice2Predict was done on the University of Twente HPC cluster using 16 CPU cores and 480Gb of memory (which was the minimum requirement). Nice2Predict was setup with default parameters, the same as the original paper where DeGuard is described. Delibird was trained with a single CPU core and 8Gb of memory.

5.3 Testing

DeGuard also uses Nice2Predict for prediction. Given, a trained Nice2Predict model and an obfuscated application, Nice2Predict predicts the names of identifiers. Delibird uses library templates from the training step and uses those to predict obfuscated application identifiers.

	Correct	Incorrect	Unpredicted
method	0.362	0.231	0.408
class	0.506	0.439	0.055
package	0.359	0.444	0.197

Table 1: This table contains the exact values for the graph in Figure 4.

5.4 Metrics

There are different types of identifiers in Android applications; method names, class names, and package names. We split the evaluation on these identifier types as they occur in different frequencies. For each of these identifier types, we collected whether they were correctly predicted, incorrectly predicted, or unpredicted. DeGuard gives an unpredicted result when it has not identified enough neighbouring identifiers to make a prediction. Delibird gives an unpredicted result when there is no match between the identifier and an identifier from a library.

To see if an identifier is correctly predicted, we first use the original mapping created by ProGuard to what the obfuscated identifier is. Then we use the mapping created by DeGuard to get the prediction for the obfuscated identifier. The identifier is correctly predicted when it is an exact case-sensitive match with the original identifier. This is quite a strict requirement, but doing a case insensitive match would likely not make a significant impact on the results because third party libraries are included in apps without making changes to the identifiers.

5.5 Results

We have evaluated the results of DeGuard for all identifiers, the combination of identifiers from third party libraries and those not from third party libraries. These results can be compared to the original DeGuard paper to give an indication of correctness. These results can be found in Figure 4.

Because in this research we want to compare the third party library deobfuscation capabilities of DeGuard with those of Delibird, we created a baseline from the results of Figure 4 that contains only the results for third party library identifiers. This DeGuard baseline for third party libraries can be found in Figure 5. A substantial amount of identifiers found in Android applications are from third party libraries as can be seen Figure 3.

The Delibird results can be found in Figure 6. This figure contains the results split by identifier types. As can be seen in Figure 6, the Delibird method results in a lot unpredicted identifiers. This is caused by the use of LibPecker.

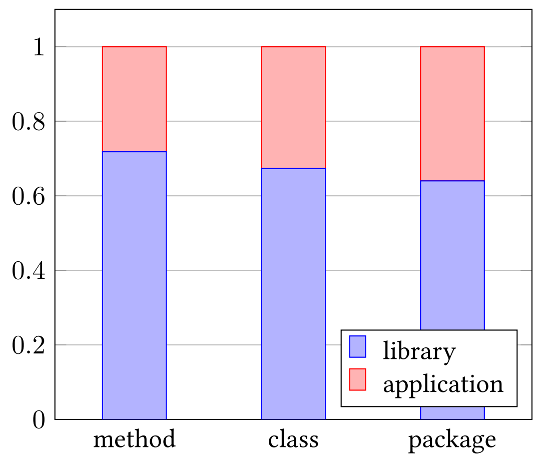

Figure 3: A substantial amount of identifiers found in Android applications are from third party libraries. 72% of method identifiers found in Android applications are from third party libraries, compared to 67% of class identifiers, and 64% of package identifiers.

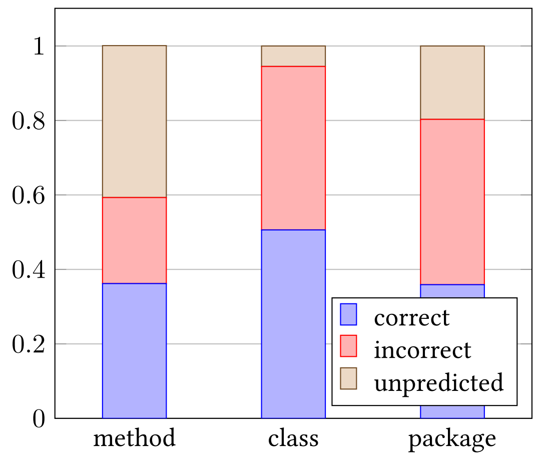

Figure 4: For this research we implemented the DeGuard model. The performance of this implementation on our data set is shown in this figure. This includes results on all identifiers including both third party library identifiers and the application specific identifiers. For each identifier type, this figure shows the percentage of correct (blue), incorrect (red), and unpredicted (beige) identifiers. Exact numbers can be found in Table 1.

	Correct	Incorrect	Unpredicted
method	0.445	0.185	0.371
class	0.645	0.303	0.053
package	0.425	0.412	0.163

Table 2: This table contains the exact values for the graph in Figure 5.

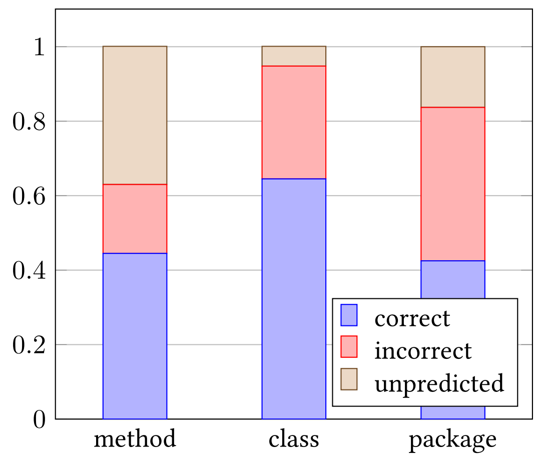

Figure 5: For our research we want to compare against the library deobfuscation capabilities of DeGuard. This figure shows the performance of our implementation of DeGuard only the library identifiers from our data set. For each identifier type, this figure shows the percentage of correct (blue), incorrect (red), and unpredicted (beige) identifiers. Exact numbers can be found in Table 2.

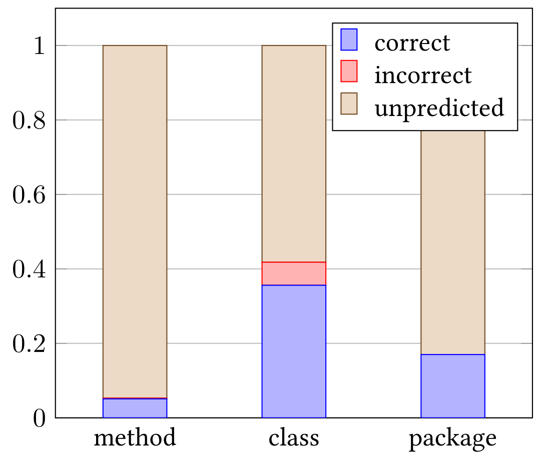

Figure 6: The results of our Delibird method on our data set are shown in this figure. These results can be compared against the results in Figure 5. For each identifier type, this figure shows the percentage of correct (blue), incorrect (red), and unpredicted (beige) identifiers. Exact numbers can be found in Table 3.

	Correct	Incorrect	Unpredicted
method	0.051	0.002	0.947
class	0.356	0.062	0.582
package	0.170	0.000	0.830

Table 3: This table contains the exact values for the graph in Figure 6.

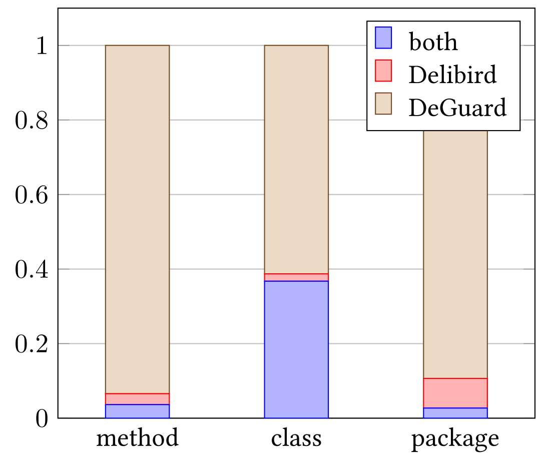

Figure 7: This figure shows for all identifier predictions, including correct and incorrect predictions, if they they are predicted by Delibird, DeGuard, or by both Delibird and DeGuard. The largest part of all identifiers are only predicted by DeGuard. This is the case because DeGuard leaves fewer identifiers unpredicted as can be concluded from Figure 5 and Figure 6. The fraction of predictions made by both Delibird and DeGuard is the largest for the class identifiers. Delibird makes a lot of method and package predictions that DeGuard does not when compared to the total amount of Delibird method and package predictions.

LibPecker predicts whether an application contains a certain third party library and is designed to have a high recall and accuracy. Because LibPecker is designed to have a high accuracy, it makes library predictions on only a few very confident class matches. More class matches are not needed to have a high accuracy and recall on predicting libraries. A few class matches can already make a confident prediction of whether a library is included in an application. Because LibPecker only identifies a few classes, it leaves a lot of classes unidentified. This is not a problem for LibPecker, because it only needs a couple of identified classes. However, it is a problem Delibird because the goal of Delibird is to predict all classes. This is the reason why so many classes are left unpredicted by Delibird.

Figure 7 shows that Delibird and DeGuard predict the same class identifiers. There is only a small portion of the identifiers that is only predicted by Delibird.

Figure 8 shows the fraction of correct predictions for the identifiers that are predicted by by both Delibird and DeGuard. The identifiers that are predicted by both Delibird and DeGuard correspond to the both category from Figure 7.

Delibird predicts the same methods identifiers better than DeGuard does. However, DeGuard does still do good job at predicting that fraction of method identifiers. A similar pattern can be seen for the class predictions, although, both have fewer correct class predictions than method predictions.

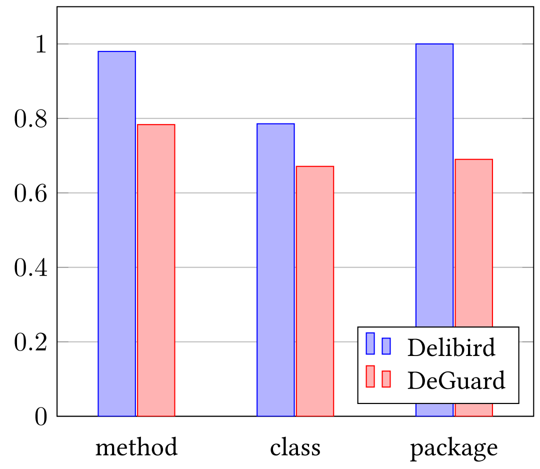

Figure 8: Fraction of correct predictions for identifiers predicted by both Delibird and DeGuard.

6 DISCUSSION

In this section we will discuss the performance of our implementation of DeGuard and the differences in the results from Delibird and DeGuard.

6.1 DeGuard

When comparing the performance of our implementation of DeGuard, see Figure 4, and the one in the original DeGuard paper, there are some differences. The most noticeable is that our implementation has more incorrectly predicted identifiers.

This can have several reasons. In the DeGuard paper the test applications were randomly selected from the F-Droid applications that are build with the Gradle build system. We randomly chose the applications to evaluate on from all applications, including other build systems. Gradle is the most recent build system. Our training set contains therefore apps with older build systems and these older build systems may indicate that those are old applications. Older applications could be harder to predict than newer applications.

Furthermore, the original DeGuard paper does not specify the details of the algorithms very precisely. It could be that some things are implemented slightly different. For example some features are implemented slightly different.

Finally, a difference in the size and composition of the dataset could also explain some differences. For example, the size of the training set is important. The size of our training set is smaller than the one used in the DeGuard paper. The training set size used in the original DeGuard paper was 1684. The test set size was 100. Our training set contains 1361 samples, and our test set contains 150 samples. Besides, if our training set would have been larger, we could also expect a better performance from Delibird.

6.2 Delibird

When comparing the results of DeGuard and Delibird, there is a clear difference. Delibird has a lot more unpredicted identifiers than DeGuard, the recall is lower. Delibird has less incorrect predictions than DeGuard, but has also less correct predictions.

Although delibird is overall not better, it can still be useful because the precision of Delibird is better than DeGuard. Because the ratio of correct and incorrect prediction is better in Delibird than in Deguard. The identifiers that Delibird predicts are predicted with high precision. Moreover, the most important improvement of Delibird over DeGuard is that Delibird requirements a lot less memory for training than DeGuard.

Library detection can be use to deobfuscate libraries with good precision. However, the recall of Delibird is low. This can be explained by the fact that library detection is measured on library level. A few confident predictions of classes are enough to identify a library. Thus, a good recall on library detection systems does not necessarily translate to a good recall in library deobfuscation. However, because the library that is used is already identified, it should be possible to match all other classes from the library as well because the search is very small. This could be added as an extension to Delibird.

The high precision can be explained by the fact that the library detection method identifies a few very confident class predictions to make a library prediction. Because these class predictions are used by Delibird, the deobfuscation of Delibird has also a high precision.

The method predictions depend on the prediction of the classes by LibPecker. Delibird matches the methods in the predicted classes with the methods in the predicted library classes. This has the effect that only methods in predicted classes are attempted to be predicted. The methods that are predicted, have a high accuracy as a signature based approach is used. By definition, this signature based approach to match methods is 100% accurate in correctly predicted classes. In incorrectly predicted classes, there will not be many predictions as only very few method signatures can be matched between the library and application class. This results in an overall high accuracy.

In this research we did not use K-Fold cross validation. Using cross-validation would result in a higher confidence interval of the results. However, due to time constraints K-Fold cross validation is not used. A single evaluation run of DeGuard and Delibird consists out of training Delibird, training DeGuard, prediction with DeGuard and prediction with Delibird. Such a run takes three days of uninterrupted running time to complete because the Nice2Predict model that DeGuard uses is slow for large inputs. Meaning that if we used 10-fold cross-validation, this would take a month in the ideal scenario. However the cluster we trained on is not always available. The original DeGuard paper also did not use k-fold cross validation, probably for the same reasons.

The main bottleneck in the evaluation run is training DeGuard. DeGuard uses Nice2Predict internally as its prediction model Nice2Predict is a toolkit for training conditional random fields (CRFs) and making predictions. CRFs are computationally expensive models. To speed up training, Nice2Predict loads the entire dataset into memory. This explains the high memory usage of DeGuard. Another speed up technique that Nice2Predict uses is parallel training. This optimisation also increases memory usage. This means that Nice2Predict has a trade-off between memory and running time. Predicting with Nice2Predict is also quite slow with large datasets as inference for CRFs is a difficult problem. Exact inference, meaning finding the optimal combination of predictions, is intractable general types of CRFs. Nice2Predict uses an approximate inference method, however, this is still quite slow. Training and predicting with Delibird is not a bottleneck as both tasks can be parallelised easily without needing as many resources as DeGuard.

7 CONCLUSION

In this research we evaluated the possibility of deobfuscating third party Libraries in Android applications by using existing library detection tools. In this paper we evaluated this approach and compared it to an existing deobfuscation approach based on statistical learning.

We found that library detection tools can be successfully used for deobfuscating third party libraries with high accuracy. However, compared to existing techniques based on statistical learning, the recall is significantly lower.

A major advantage of our approach is that it does not require a large amount of computational resources as is the case with DeGuard. This is the case because library detection tools are based on simpler matching techniques as opposed to the computationally expensive CRF model used in DeGuard. Another advantage of our approach is that extra training samples can be added easily. This is not the case with DeGuard. DeGuard needs to be retrained for all changes in the data set.

8 FUTURE WORK

For this research we used a basic type of obfuscation, namely identifier deobfuscation. This is a basic kind of obfuscation that does not change the structure of programs. DeGuard relies on the structure of a program to predict classes and packages. There are more thorough obfuscation options. One such an option provided by ProGuard is repackaging. This obfuscation technique puts all classes in the same root package. It is expected that this technique influences the prediction capabilities of DeGuard and Delibird. DeGuard uses the package hierarchy to quickly identify classes in the same package using package structure features. Library detection tools use package hierarchies in a similar manner to detect libraries. Java classes in the same package usually have strong interdependence and Java classes in different packages are often more loosely coupled. It might me interesting to see if graph clustering techniques can be used to reconstruct the original package hierarchy from repackaged classes. This would give DeGuard and Delibird extra features to make predictions.

In future work it would be interesting to see if the amount of unpredicted identifiers can be reduced. This could be achieved by improving the used library detection tool or selecting another library detection tool. The library detection tool we used (LibPecker) is not the only tool that has library prediction capabilities. We chose this tools as it has an additional class mapping output. Other library detection tools with this capabilities can be evaluated as well.

For this research, we used the libraries included in apps from the F-Droid appstore as a training set for the library detection tool. This has the advantage that the training set contains libraries that are likely to be used in Android applications. There are also other sources to consider for a library data set such as MavenCentral. These repositories contain many Java libraries which can be used to increase the library training set. Although, using these libraries results in a larger training set, these repositories might also contain libraries that are not frequently used inside Android applications. It is interesting to look at what the trade-offs are in using different library data sets.

Third party libraries are a common practice in other programming languages, such as JavaScript and Python. For these platforms obfuscation is also a common practice. For JavaScript uglifyjs exists as a obfuscator and for Python pyminifier. What makes Java suitable for deobfuscation, is the clear structure of compiled programs. Structure of programs in preserved in Java bytecode. Therefore identifier deobfuscation is possible. The same holds for interpreted languages such as JavaScript and Python. Therefore, it might be interesting to explore if this research could also be applied to those other languages.

This research can be used a starting point for other deobfuscation system. First use Delibird as a pre-processing step, this gives extra known identifiers in addition to the known identifiers from the standard library.

References

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *ACM SIGPLAN Notices*, volume 53, pages 404–419. ACM, 2018.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3:40, 2019.

[4] Android studio release notes 3.4. `https://developer.android.com/studio/releases#3-4-0`, 2019.

[5] Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G Feitelson. Meaningful identifier names: the case of single-letter variables. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 45–54. IEEE, 2017.

[6] 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*, pages 343–355. ACM, 2016.

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

[8] Enigma. Last accessed: 14-03-2021, `https://www.cuchazinteractive.com/enigma/`.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java TM Language Specification, Third Edition.* Sun Microsystems, 2005.

[10] Johannes Hofmeister, Janet Siegmund, and Daniel V Holt. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER)*, pages 217–227. IEEE, 2017.

[11] Java deobfuscator. Last accessed: 14-03-2021, `https://javadeobfuscator.com/`.

[12] Mark D LaDue. Hosemocha. *URL: http://www. cigital. com/hostile-applets/HoseMocha. java*, 1997.

[13] Tımea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30:3–19, 2009.

[14] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[15] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*, pages 111–124. ACM, 2015.

[16] Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. Java obfuscation with a theoretical basis for building secure mobile agents. In *IFIP International Conference on Communications and Multimedia Security*, pages 89–103. Springer, 2003.

[17] The source optimizer for Java bytecode. `https://www.guardsquare.com/en/products/proguard`. Accessed: 2019-04-04.

[18] Rebecca Jen-Hui Wang, Edward C Malthouse, and Lakshman Krishnamurthi. How mobile shopping affects customer purchase behavior: A retailer's perspective. In *Let's Get Engaged! Crossing the Threshold of Marketing's Engagement Era*, pages 703–704. Springer, 2016.

[19] Joanna Witkowska. The quality of obfuscation and obfuscation techniques. In *Biometrics, Computer Security Systems and Artificial Intelligence Applications*, pages 175–182. Springer, 2006.

[20] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. Detecting third-party libraries in android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 141–152. IEEE, 2018.