Obfuscation-Resilient Code Detection Analyses for Android Apps

Dissertation

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Yan Wang
Graduate Program in Computer Science and Engineering

The Ohio State University

2018

Dissertation Committee:

Atanas Rountev, Advisor

Michael D. Bond

Zhiqiang Lin

ABSTRACT

Program analysis, both static and dynamic, is helpful in understanding and improving Android software. Many techniques have been developed for various purposes including test generation, bug detection, identification of third-party libraries, clone detection, and malware detection. However, Android developers commonly use app obfuscation to secure their apps and to protect their intellectual property. Such obfuscation presents an obstacle for a number of legitimate and important program analyses. Therefore, it is necessary to investigate and reduce the negative effects of Android obfuscation on program analysis algorithms and tools. This dissertation makes several contributions towards tackling this problem.

To understand Android obfuscation, it is useful to be able to identify the obfuscator used to process a given app. There are many popular obfuscators, including both free ones and commercial ones. Although the goals of these obfuscators are identical, their algorithms and implementations vary. Furthermore, the same tool may have different configurations, which will change its behavior and the resulting obfuscated code. As the first contribution of this dissertation, we define the obfuscator identification problem and propose a machine learning approach to identify the type of obfuscator and its concrete configurations used by an Android app. The approach achieves this identification by defining and solving a classification problem. We design the structure of the feature vector which is extracted from Android apps to represent

the characteristics of the obfuscated Dalvik bytecode. Then a classifier is applied to identify the obfuscator provenance information. To evaluate our approach, we studied hundreds of open-source Android apps and demonstrated the high accuracy of the proposed techniques. Moreover, we performed a study on thousands of Google Play Store apps, and found that obfuscation is widely used in popular apps.

Third-party library detection and clone detection are representative examples of program analysis problems that become more difficult in the presence of obfuscation. Third-party libraries are widely used in Android apps. Various program analyses need to identify or even remove these third-party libraries. In addition, many third-party libraries for Android have been shown to bring security and privacy hazards by adding security vulnerabilities to their host apps or by misusing inherited access rights. Correctly attributing improper app behavior to library code, as well as isolating libraries from their host apps, can be highly desirable in order to mitigate these problems. As for clone detection, attackers often generate repackaged apps to steal the revenue from original owners or to deceive the users in order to install. Several approaches have been proposed to detect third-party libraries and clones for Android, but none of them fully consider/handle the impact of obfuscation. The second contribution of this dissertation is a study of the limitations of prior work in this area, as well as the design of a novel approach for obfuscation-resilient detection of third-party libraries and app clones. Unlike earlier efforts, the approach employs interprocedural code features (for improved effectiveness) and similarity digests (for efficiency). Experimental evaluation using a number of applications and libraries shows substantial performance improvements compared to a state-of-art tool for detection of libraries and app clones.

The next contribution of this dissertation is at a much finer granularity: method-level detection. Identifying the existence of a certain API method of interest in a given obfuscated app is needed in many scenarios related to app analysis/transformation. Prior work on this topic does not successfully account for the effects of commonly-used obfuscation techniques for Android. To address this challenge, we propose an obfuscation-resilient approach for method detection. The key components of this approach are the design of a new type of fuzzy signature for methods, together with an efficient technique to compare these signatures against a given obfuscated app. We study three clients of our analysis and demonstrate the effectiveness and practicality of the developed method detector.

In conclusion, this dissertation presents several novel static analysis techniques to solve challenges created by Android obfuscation. These techniques exhibit good cost/precision performance on a large number of real-world apps, and advance the state of the art in analysis of obfuscated Android code.

To my family

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Dr. Atanas (Nasko) Rountev, for his continuous support, guidance and patience during my Ph.D. studies. I really appreciate his dedication in training and helping me with research and writing of this dissertation. Besides my advisor, I would like to thank Dr. Michael D. Bond and Dr. Zhiqiang Lin for serving on the dissertation committee and providing insightful comments, encouragement and questions. I would also like to thank all the members of the PRESTO group for the discussion, collaboration and fun we have together in the last five years. My special thanks to Dr. Xun Gong and Dr. Chih-hung Hsieh for their kindly mentoring on the intern projects at Google. Last but not least, I would like to thank my family: my parents and my wife for their unconditional support and understanding through my study and my life in general.

The material presented in this dissertation is based upon work supported by the U.S. National Science Foundation under grants CCF-1319695 and CCF-1526459, and by a Google Faculty Research Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VITA

August 2013 – present	Graduate Teaching/Research Associate, The Ohio State University
June 2013	B.Eng. Software Engineering, Tongji University
February 12, 1991	Born – Puyang, Henan, China

PUBLICATIONS

Research Publications

Shengqian Yang and Haowei Wu and Hailong Zhang and Yan Wang and Chandrasekar Swaminathan and Dacong Yan and Atanas Rountev. Static Window Transition Graphs for Android. In *International Journal of Automated Software Engineering (JASE'18)*, Jun 2018.

Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev.Orlis: Obfuscation-Resilient Library Detection for Android. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'18)*, May 2018.

Haowei Wu, Yan Wang, and Atanas Rountev. Sentinel: Generating GUI Tests for Android Sensor Leaks. In *IEEE/ACM International Workshop on Automation of Software Test (AST'18)*, May 2018.

Yan Wang and Atanas Rountev. Who Changed You? Obfuscator Identification for Android. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'17)*, May 2017.

Yan Wang, Hailong Zhang, and Atanas Rountev. On the Unsoundness of Static Analysis for Android GUIs. In *ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP'16)*, June 2016.

Yan Wang and Atanas Rountev. Profiling the Responsiveness of Android Applications via Automated Resource Amplification. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'16)*, May 2016.

Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. Static Window Transition Graphs for Android. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*, November 2015.

Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *International Conference on Software Engineering (ICSE'15)*, May 2015.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Programming Language and Software Engineering	Prof. Atanas Rountev
Databases/Analytics	Prof. S. Parthasarathy
High-Performance Computing	Prof. P. Sadayappan

TABLE OF CONTENTS

Page

Abstract ii

Dedication v

Acknowledgments vi

Vita vii

List of Figures xii

List of Tables xv

Chapters:

1. Introduction 1

    1.1 Challenges 2
    1.2 Obfuscation-Resilient Code Detection Analyses 3
        1.2.1 Android Obfuscator Identification 4
        1.2.2 Obfuscation-Resilient Detection of Third-party Libraries and App Clones 5
        1.2.3 Obfuscation-Resilient Detection of Methods 8
        1.2.4 Outline 9

2. Background 10

    2.1 General Software Obfuscation 10
    2.2 Android Obfuscation 12

3. Android Obfuscator Identification 18

3.1 Overview 19
3.1.1 Machine Learning Model 19
3.1.2 Obfuscators Used in Our Study 20
3.2 Machine Learning for Obfuscator Identification 20
3.2.1 Training Set 21
3.2.2 Feature Dictionary 22
3.2.3 Training a Classifier 25
3.3 Obfuscator Configuration 26
3.3.1 ProGuard Configuration 27
3.3.2 Allatori/DashO Configuration 29
3.4 Evaluation 30
3.4.1 Data Set 31
3.4.2 Methodology 31
3.4.3 Obfuscator Identification 31
3.4.4 ProGuard Configuration Identification 33
3.4.5 Allatori and DashO Configurations 36
3.4.6 Comparison of Classifiers 36
3.4.7 Case Study: Google Play Apps 38
3.4.8 Limitations 40
3.5 Summary 41

4. Obfuscation-Resilient Detection of Third-Party Libraries and App Clones 42

4.1 Background 42
4.1.1 Android Obfuscation Techniques 42
4.1.2 Existing Library Detection Tools for Android 44
4.1.3 Existing Clone Detection Tools for Android 48
4.2 Assumptions about Obfuscation 50
4.2.1 Method-Level Assumptions 51
4.2.2 Class-Level Assumptions 51
4.3 ORLIS: An Obfuscation-Resilient Approach for Library Detection . 52
4.3.1 Code Features and Detection Workflow 53
4.3.2 String Features for Methods 54
4.3.3 Stage 1: App-Library Similarity 56
4.3.4 Stage 2: Class Similarity 57
4.4 ORCIS: An Obfuscation-Resilient Approach for Clone Detection . . 57
4.4.1 Code Features and Detection Workflow 58
4.5 Tool Implementation 59
4.6 Evaluation 60

4.6.1 Evaluation of Orlis . . . 61
4.6.2 Evaluation of Orcis . . . 72
4.7 Summary . . . 80

5. Obfuscation-Resilient Detection of Methods . . . 81

5.1 Motivation and Challenges . . . 81
5.2 Obfuscation-Resilient Approach for Method Detection . . . 82
5.2.1 Invariant Properties . . . 83
5.2.2 Detection Workflow and Method Fuzzy Signature . . . 84
5.2.3 Stage 1: Checking the Set of Call Pairs . . . 88
5.2.4 Stage 2: Matching of Individual Methods . . . 88
5.2.5 Bloom Filter . . . 89
5.2.6 Implementation . . . 90
5.3 Evaluation . . . 91
5.3.1 Data Sets . . . 91
5.3.2 Evaluation on a Static Analysis of Android Wear Notifications 94
5.3.3 Evaluation on Data Leakage Analysis for Android Cloud Services . . . 97
5.3.4 Evaluation on Analysis of Analytics Libraries for Android Apps 101
5.3.5 Comparison of Method Fuzzy Signatures . . . 103
5.3.6 Comparison with Similarity Digests . . . 104
5.4 Limitations . . . 107
5.5 Summary . . . 107

6. Related Work . . . 108

6.1 General Software Obfuscation . . . 108
6.2 Android Obfuscation . . . 109
6.3 Impact of Android Obfuscation . . . 109
6.4 Provenance Analysis . . . 110
6.5 Similarity Digests . . . 111
6.6 General Software Clone Detection . . . 112

7. Conclusions . . . 115

7.1 Android Obfuscator Identification . . . 115
7.2 Obfuscation-Resilient Detection of Third-party Libraries and App Clones . . . 116
7.3 Obfuscation-Resilient Detection of Methods . . . 117

Bibliography . . . 118

LIST OF FIGURES

Figure **Page**

2.1 Example of Dalvik bytecode 13

2.2 Workflow of Android compilation 13

2.3 Workflow of an obfuscator 14

2.4 Workflow of a packer 14

2.5 Example of obfuscation performed by Allatori 15

2.6 Example of obfuscation performed by Legu 16

3.1 Workflow of obfuscator identification 21

3.2 Structure of feature dictionary 23

3.3 Workflow of configuration characterization 27

3.4 Performance of different classifiers 38

4.1 Workflow of library detection 54

4.2 String features used for matching 56

4.3 Workflow of clone detection 59

4.4 String feature of an app 60

4.5 Performance of similarity digests for *FDroidData* 64

4.6 Running time of *FDroidData* 67

4.7 Comparison with LibDetect 68

4.8 Impact of thresholds on *FDroidData* 71

4.9 Performance of similarity digests 74

4.10 Running time 75

4.11 Comparison with CodeMatch 76

4.12 Comparison with CodeMatch using ORLIS 77

4.13 Measurements with and without a library detection stage 78

4.14 Impact of thresholds for *AndroZooData* 79

5.1 Workflow of method detection 85

5.2 Class type fuzzy signature 86

5.3 Example of a method signature 86

5.4 Example of a method signature after obfuscation 87

5.5 Example of a method fuzzy signature 87

5.6 Android Wear API methods detection 95

5.7 Results of Wear API methods detection 95

5.8 Results of comparison 99

5.9 Results of cloud service API methods detection 100

5.10 Number of callees 101

5.11 Results of Google Analytics API methods detection 102

5.12 Results of Google Analytics API methods detection 102

5.13 Comparison of the two fuzzy signatures 103

5.14 Method size ratio 105

5.15 Results of comparison 106

LIST OF TABLES

Table **Page**

3.1 Accuracy and F_1 score for obfuscator identification 32

3.2 Precision and recall for obfuscator identification 33

3.3 Accuracy and F_1 for ProGuard configuration 33

3.4 Precision and recall for ProGuard configuration 34

3.5 Accuracy with never-seen custom names 35

3.6 Accuracy and F_1 for Allatori/DashO config 37

3.7 Precision and recall for Allatori/DashO config 37

3.8 Obfuscators for Google Play apps 40

3.9 Apps obfuscated with ProGuard 40

4.1 Obfuscator features 43

4.2 Apps in *FDroidData* 61

4.3 Comparison with LibDetect on *FDroidData* 62

4.4 Similarity score metrics 64

4.5 Number of APKs in each dataset 73

4.6 Number of clone and not-clone pairs in each dataset 73

5.1 Data Sets 92

5.2 APIs 93

5.3 Effects on model building for Android Wear notifications 97

CHAPTER 1: Introduction

The mobile app market has grown rapidly in the past decade. Android has become one of the largest mobile application platforms, with about 2.5 million apps in the Google Play Store by 2017 [7], generating revenue of about 31 USD billion [15]. Many apps contain sensitive private information or valuable business logic. Due to security considerations, as well as intellectual property concerns, developers often release closed-source application. To further hinder decompilation tools and manual reverse engineering, apps are often protected using *obfuscation*. An earlier study [115] estimates that about 15% of Google Play apps are obfuscated, and more recent studies indicate even wider use of obfuscation [119].

Although obfuscation provides desirable protection, it presents an obstacle for legitimate program analyses. Consider the *detection of app cloning and repackaging*. At least 25% of apps in Google Play are clones of other apps [115] and almost 80% of the top 50 free apps have fake versions because of repackaging [73]. Obfuscation may impair clone and repackage detection tools because an obfuscated clone app may appear to be different from the original app [52, 114]. Another consideration is *malware detection* using static analysis. Analysis of obfuscated malicious code presents a number of challenges [38, 133]. As yet another example, identifying the *third-party libraries* included in an Android app is a well-known problem important for general static analysis, security analysis, testing, and detection of performance bugs. According to a study [121], approximately 70% and 50% of vulnerabilities of

free and paid apps, respectively, are due to third-party libraries. Obfuscation hinders such library identification [9, 51, 74]. *API method detection* is another useful analysis that has many clients [128, 134]. Such an analysis will also be significantly affected by common obfuscation techniques. Due to the widespread use of Android obfuscation and the necessity of various analysis on Android apps, it is highly desirable to be able to perform these analyses even in the presence of app obfuscation.

1.1 Challenges

Limitations of static analysis. Static analysis of app code is needed in many scenarios, as outlined above. Such analysis relies on abstracted modeling of the code structure and control flow of the program. These code properties will be significantly affected by the obfuscation process. As a result, many static analysis techniques have to be carefully (re)designed to avoid the influence of obfuscation.

Lack of knowledge about obfuscators. There are numerous approaches for code obfuscation. Different obfuscator tools employ different groups of such approaches, but the specific techniques and the details of their implementation are not public. Furthermore, the behavior of an obfuscator could change due to various configuration options. At present, there are no systematic techniques to automatically characterize the obfuscator tool that was used to produce an obfuscated app of unknown provenance. Such knowledge can be beneficial for the development of obfuscation-resilient app analyses.

Challenges for detection of third-party libraries and app clones. One important program analysis for Android aims to detect the third-party libraries used in a given app. A related analysis problem is to decide whether one app is a clone

of another. Current approaches to solve these problems rely heavily on various code properties in order to perform similarity checking. However, obfuscation often leads to code removal (e.g., removing unused library methods and classes) and package flattening/repackaging (i.e., modifying the package hierarchy), which significantly affects the similarity measurements proposed in prior work. As a result, the state of the art does not provide satisfactory obfuscation-resilient detection of third-party libraries and app clones.

Challenges for method detection. Another example of a useful analysis is the detection of a known method (e.g., an API method from a popular library) in a given app. This problem has a range of uses in security vulnerability checking and construction of static app models. The naive approach would be to perform method signature matching. However, such signatures can be changed easily by obfuscation. In addition, unlike with the detection of entire libraries and apps, detecting a single method presents challenges because a method contains much less information than a library or an app. Further, both for method detection and for the coarser-grain library/app detection, the analysis algorithms should be able to run efficiently in order to allow app-market-scale analyses—that is, to feasibly allow thousands of apps to be analyzed.

1.2 Obfuscation-Resilient Code Detection Analyses

This dissertation develops enhanced program analysis techniques in order to solve a variety of problems due to Android obfuscation. Specifically, we propose novel approaches to (1) identify the obfuscator used to create a given app, (2) detect third-party libraries and app clones despite obfuscation, and (3) identify the presence of

API methods of interest in an obfuscated app. These techniques advance the state of the art in analysis of obfuscated Android apps, in terms of both conceptual strengths and experimentally-validated performance on real apps.

1.2.1 Android Obfuscator Identification

App developers could use a variety of free and commercial obfuscator tools. If the specific obfuscator used to create the final app can be identified, and if details of the obfuscation process can be inferred, subsequent app analyses (e.g., security analysis) can exploit this knowledge. Thus, our goal is to analyze a given app and determine (1) whether it was obfuscated, (2) which obfuscator was used, and (3) how the obfuscator was configured. In addition to enabling subsequent analyses tailored to the specific obfuscator, these questions are important for provenance analysis. The app is the result of a process that starts with the source code and produces the final distributed APK file. The provenance details of this process and the toolchain that implements it are relevant for a number of reasons. For example, such details are useful for digital forensics [90] and for reverse engineering of models used for test generation (e.g., [125, 129]) and code instrumentation for performance analysis (e.g., [118]). Knowledge of the obfuscator and the configuration options used by it reveal aspects of the app toolchain provenance, similarly to provenance analysis for binary code [90].

We have developed novel techniques to identify the obfuscator of an Android app for several widely-used obfuscation tools and for a number of their configuration options. The approach relies only on the app bytecode. The identification problem is formulated as a machine learning task, based on a model of various properties of the

bytecode (e.g., different categories of strings). This model reflects characteristics of the code that may be modified by the obfuscators—in particular, names of program entities as well as sequences of instructions. The main insight of our approach is that relatively simple code features are sufficient to infer precise provenance information. The specific contributions of this work are as follows. First, we define the obfuscator identification problem for Android and propose a solution based on machine learning techniques. To the best of our knowledge, this is the first work to formulate and solve this problem. Next, we identify a feature vector that represents the characteristics of the obfuscated code and implement a tool that extracts this feature vector from the app uses it to identify the obfuscator provenance information.

Using 282 apps and 5 popular obfuscators, we performed an experimental evaluation of the proposed approach. Our results demonstrate high accuracy for both obfuscator and configuration identification. A study of possible choices for the machine learning classifier was also performed. When applied to a group of apps from the Google Play app market, our techniques showed that obfuscation is widely used among the most popular apps. Chapter 3 describes details about our machine learning approach and its experimental evaluation. A description of this work also appeared in [119].

1.2.2 Obfuscation-Resilient Detection of Third-party Libraries and App Clones

Android apps often contain third-party libraries. Some usage statistics gathered commercially track 450 popular libraries [6] and show that many of them have significant usage in the Android ecosystem. For example, advertisement libraries, social networking libraries, and mobile analytics libraries are very popular. Some apps use

more than 20 third-party libraries [70]. Developers use these components to monetize their apps, integrate with social media, include single-sign-on services, or simply leverage the utility and convenience of libraries developed by others.

Program analysis of Android apps often requires detecting or removing third-party library code as a pre-processing step, since the libraries could introduce significant noise and could substantially affect the results of many analyses. Such library detection may be needed for security analysis [8, 10, 35, 43, 82, 86, 122, 124], clone and repackaging detection [26, 42, 132], and library removal and isolation [100, 126]. The use of libraries can also have security and privacy implications [9, 16, 34, 36, 45, 84, 86, 100, 111–113]—for example, malicious libraries or unintended security vulnerabilities in library code could lead to leaks of sensitive data.

Library detection is complicated significantly by app obfuscation. Attempts to match package/class names or method signatures with ones from known libraries are easily thwarted by the renaming typically done by Android app obfuscators. It is highly desirable to develop obfuscation-resilient third-party library detection. There have been several efforts and research tools aiming to solve this problem. However, as discussed in the Chapter 2, these tools have various limitations.

We performed an investigation of popular obfuscators and characterized their features that hinder library detection. We then studied how existing advanced publicly-available library detection tools perform in the presence of such features. Although some of the state-of-the-art tools are intended to be resilient to obfuscation, we argue that there is still room to improve recall, precision, and analysis cost.

Based on these observations, we propose a new approach to detect third-party libraries in obfuscated apps. The approach relies on obfuscation-resilient method-level

and class-level information. The richness of this information is significantly higher than what is commonly used by library detection tools—for example, it considers transitive calling relationships between methods, as well as the structural relationships in class hierarchies and methods defined in them. These features capture the *inter*-procedural structure and behavior of the app. This is in contrast with existing approaches, which conceptually consider analysis of each separate method but not the relationship between methods—that is, *intra*-procedural analysis. The design of our approach is informed by close examination of the code features preserved by typical Android obfuscators. The resulting library detection approach exhibits higher precision and recall than prior work of similar nature.

Library detection may have to be performed at large scale—for example, when app-market-scale analyses are applied for clone detection and security analysis. To reduce the cost of the analysis, we use the machinery of *similarity digests* [28,64,80,91]. Such digests are similar to cryptographic hashes such as MD5 and SHA, but with the additional property that a small change in the data object being hashed results in a small change to the hash itself. In our context, similarity digests provide an efficient mechanism for identifying a small number of likely matches. The use of such digests, rather than more traditional hashes, is critical: due to obfuscation, the library code that is preserved in the obfuscated app does not match precisely the code in the original library; thus, direct comparison of hash codes is not possible. We explore several similarity digests and demonstrate empirically that different ones are needed for different aspects of the library detection process.

We implemented this approach in the Orlis tool, which performs obfuscation-resilient library detection using interprocedural code features and similarity digests.

Our experimental evaluation studies the effects of various choices in the analysis implementation, and then compares precision/recall with the state-of-the-art LibDetect tool [42]. As demonstrated by our results, ORLIS presents an attractive choice for detection of third-party libraries in Android apps.

Similarly to library detection, app clone detection is also hindered by obfuscation. We extended our library detection approach to perform clone detection, and implemented this approach in the ORCIS tool, which performs obfuscation-resilient clone detection using interprocedural code features and similarity digests.

Our experimental evaluation studied various design choices for ORLIS and ORCIS, and compared these two tools with the state-of-the-art LibDetect library detection tool and CodeMatch clone detection tool. The results from this evaluation show that the proposed tools outperform the current state-of-art approaches.

Chapter 4 describes the design and evaluation of ORLIS and ORCIS. The library detection techniques were first presented in [120].

1.2.3 Obfuscation-Resilient Detection of Methods

ORLIS and ORCIS consider an entire library or app. However, for many static analyses, researchers are interested in identifying instances of certain methods of interest. For example, if some third-party library methods have security vulnerabilities, these specific methods have to be identified in an obfuscated app. As another example, in order to build static models for Android and Android Wear apps, it is often necessary to consider specific methods from certain libraries (e.g., Google libraries related to app analytics or wearable devices) and to identify these methods and their callers in a given app. In these cases, the naive approach to detect a method is signature

matching. However, as discussed earlier, obfuscation will significantly change method signatures. To address this problem, we propose an obfuscation-resilient approach for method detection. This approach is based on obfuscation-resilient properties, which are used to define "fuzzy" method signatures and code similarity analyses based on them. To achieve scalability, the approach employs Bloom filters, which allow fast set operations and contribute to practical running time.

In the evaluation of this approach, three client analyses are considered: (1) static analysis of notifications for Android wearable devices, (2) security vulnerability detection for Android cloud service APIs, and (3) analysis for Google Analytics APIs. The results from our evaluation show that the proposed technique is effective and efficient. Chapter 5 describes the details of this work.

1.2.4 Outline

The rest of this dissertation is organized as follows. Chapter 2 briefly introduces background on code obfuscation. Chapter 3, Chapter 4, and Chapter 5 present the novel program analysis techniques contributed by this dissertation. Related work is described in Chapter 6. Chapter 7 summarizes our contributions and conclusions.

CHAPTER 2: Background

2.1 General Software Obfuscation

Software obfuscation is a well-known technique for protecting software from reverse engineering attacks [23]. Obfuscation transforms the input code to generate new target code. In many cases, it is similar to the optimization pass of a compiler. One possible definition of obfuscation is as follows [23]:

Definition 2.1.1 *Let $P \rightarrow P'$ be a transformation of a source program P into a target program P'. $P \rightarrow P'$ is an obfuscating transformation if P and P' have the same observable behavior. More precisely, in order for $P \rightarrow P'$ to be a legal obfuscating transformation, the following conditions must hold:*

- *If P fails to terminate or terminates with an error condition, then P' may or may not terminate.*
- *Otherwise, P' must terminate and produce the same output as P.*

Obfuscation cannot achieve a "virtual black box" [13]—in other words, perfect obfuscation is impossible. Nevertheless, in practice it is still an effective approach to protect the code from humans and from code analysis and reverse engineering tools [24].

Evaluation of Obfuscation

There are metrics to evaluate the quality of obfuscation [23]. *Potency* measures how much more difficult the obfuscated code is to understand for a human than the original code. In other words, potency indicates the complexity of the obfuscated code. Many metrics from software engineering can be used to measure potency; examples include, program length, number of predicates, level of nesting conditionals, data flow complexity, data structure complexity, and class hierarchy complexity. *Resilience* measures how well an obfuscation resists an attack from an automatic deobfuscator. This aspect of obfuscation is important because it is easy to increase code complexity (i.e., potency) to confuse a human programmer, but many of these techniques cannot thwart automatic analyses. The resilience is a combination of two measurements: programmer effort to build an automatic deobfuscator, and deobfuscator execution time and space. Finally, another evaluation aspect is the *execution cost* for the obfuscated code.

Obfuscation Techniques

Numerous obfuscation techniques have been developed. Some popular techniques can be classified in four major categories [23].

The first category is *layout obfuscation*. This is the simplest technique in obfuscation. It will remove code formatting and rename the identifier names, method names, class names, and package names to some meaningless names. Such transformations have low potency, because they do not remove much semantic information. Their execution cost is zero.

Another example is *control obfuscation*. This is a large category with many existing techniques. Control transformations can introduce conditionals with only one possible run-time outcome, insert dead or irrelevant code, make loop conditions more complex, convert a reducible control flow graph to a irreducible one. etc. In general, control obfuscation has high potency and resilience with relatively low cost.

Aggregation obfuscation will obscure the program by destroying natural programming abstractions. For example, the code in one component (e.g., a block, a method, or a class) can be broken up and scattered over the program. Conversely, code in different places without obvious logical relationship can be aggregated into one component. Concrete examples in this category include inlining and outlining of methods, interleaving of methods, loop transformations, code ordering transformations, etc.

Finally, *data obfuscation* modifies the data structures used in the program code. Examples include encrypting constant strings, restructuring arrays, and splitting variables.

In addition to these four general categories, there are some techniques designed to prevent deobfuscation or static analysis. For example, to impair program slicing, some strategies such as adding parameter aliases and adding variable dependencies can be used. Additional examples of various obfuscation techniques can be found in taxonomy defined by Collberg at al. [23].

2.2 Android Obfuscation

Obfuscation is also pervasively used in the Android ecosystem. Obfuscation for Android is both simple to use (e.g., using the ProGuard tool available in Android Studio) and observed for many applications in the Google Play app store [115]. Android

```
1. new-instance v4, Landroid/content/ComponentName; // type@0033
2. move-result-object v1
3. if-nez v1, 000d // +0004
4. const/4 v2, #int 0 // #0
5. return-object v2
6. invoke-static {v3}, Landroid/support/v4/content/IntentCompat;
.makeMainActivity:(Landroid/content/ComponentName;)
Landroid/content/Intent; // method@0e03
7. invoke-direct {v4}, Landroid/content/Intent;.<init>:()V // method@013a
```

Figure 2.1: Example of Dalvik bytecode

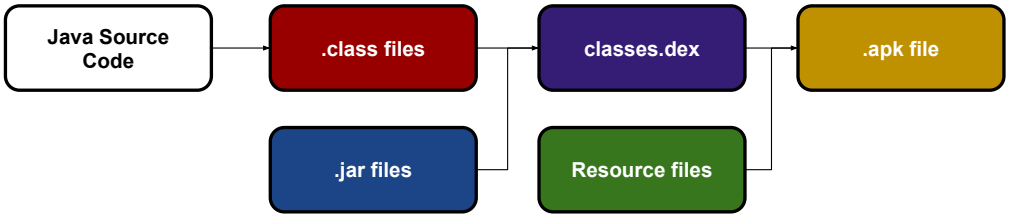


Figure 2.2: Workflow of Android compilation

apps are released as APK ("Application PacKage") files. The process of obtaining an APK file is shown in Figure 2.2. The developer builds the app using Java. The Java source code is compiled into *.class* files using some Java compiler. The *.class* files contains standard Java bytecode. Android uses its own format of bytecode called Dalvik bytecode. Figure 2.1 is a snippet of such bytecode. The Java bytecode is compiled into a file *classes.dex* ("Dalvik Executable"), which contains all the Dalvik bytecode together with other *.jar* files that may come from third-party libraries. In most cases, there is only one *classes.dex* file. However, if the code size exceeds the dex file limit, which requires the number of methods per file to be less than 65536, the app will be split into multiple dex files. In addition to the dex file(s), the apk also contains resource files (e.g., images). At installation time, the Android Runtime will

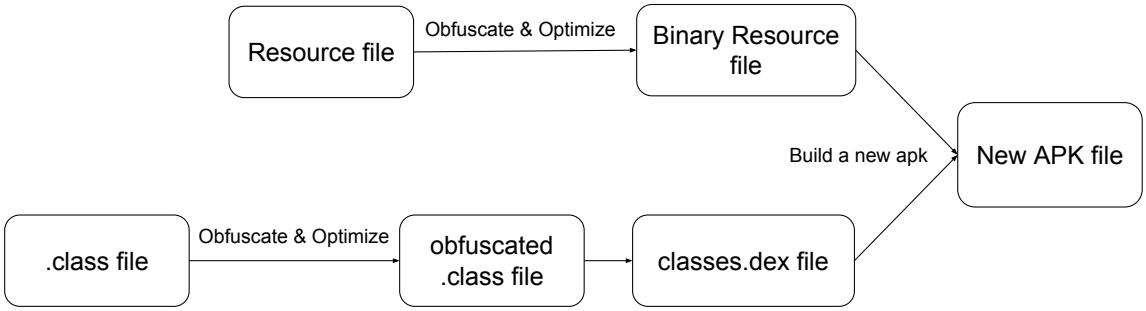


Figure 2.3: Workflow of an obfuscator

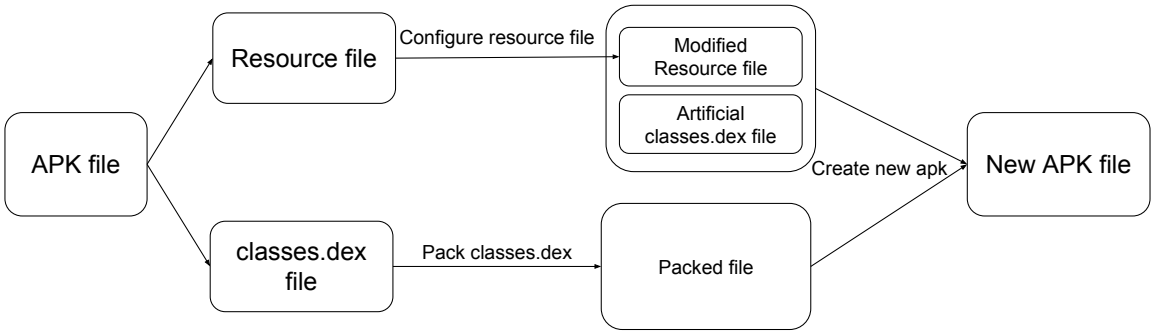


Figure 2.4: Workflow of a packer

further compile *classes.dex* using the *dex2oat* tool. The output will be executable code for target device, which achieves better performance than the Dalvik bytecode.

In Android, obfuscation always happens before releasing the apps by the developers and will modify the Dalvik bytecode. Without obfuscation, the Dalvik bytecode is relatively easy to reverse engineer, using tools such as Soot [107], Androguard [4], Baksmali [11], Apktool [5], Dex2jar [32], Dexdump [33], etc. With obfuscation, even when just using ProGuard (which only provides basic obfuscation), the code changes will affect these tools and hinder the reverse engineering process [49].

```
1  public static void initSetting(Editor editor){
2    editor.remove("deviceSalt");
3    editor.remove("encryptedKeyPass");
4    editor.commit();
5  }
6  private void goNextStage(){
7    gotoStage(this.stage + 1);
8  }
```

(a) Original classes.dex

```
1  public static void a(Editor arg0){
2    arg0.remove(R.a("o\u0018}\u0014h\u0018X\u001cg\t"));
3    arg0.remove(android.support.v7.appcompat.R.a("u,s0i2d't\tu;@#c1"));
4    arg0.commit();
5  }
6  private void a(){
7    a(this.l + 1);
8  }
```

(b) Obfuscated classes.dex

Figure 2.5: Example of obfuscation performed by Allatori

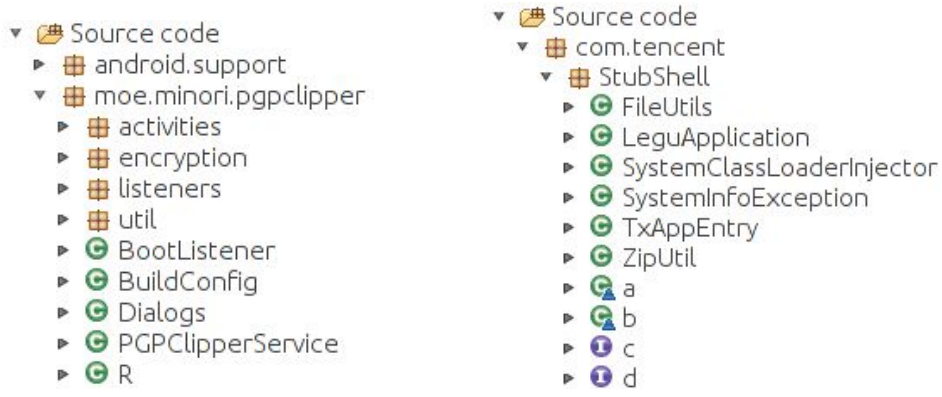
Two Categories of Obfuscators

For Android apps, few developers will manually obfuscate their application by themselves. Various obfuscators are available to Android developers. These tools can be broadly classified in two categories. An obfuscator in the first category performs code transformations. In most cases it will transform Java bytecode, and has almost the same behavior as a Java obfuscator except for considerations related to Android features. Examples of tools from this category are ProGuard, Allatori, DexGuard, and Shield4J. Figure 2.3 illustrates the workflow of this type of obfuscator.

One typical technique used in the Android obfuscation process is *layout obfuscation*, which replaces the names of packages, classes, methods, and fields with meaningless sequences of characters. Sometimes the package structure is also modified, which further obscures the names of packages and classes. Other frequently used techniques

include *control flow obfuscation*, which modifies code order or the control-flow graph, and *string encryption*, which encrypts the constant strings in the code. Some tools may go further and obfuscate the XML files in the resource part of the APK.

Example: Figure 2.5b is a representation of the obfuscation result of Allatori. Figure 2.5a shows the original code. For illustration purposes, the figures show the equivalent Java source code rather than the actual bytecode. Names are obfuscated and the constant strings are encrypted. For example, methods `initSetting` and `goNextStage` are renamed to `a`. The encrypted strings are decrypted using methods `R.a` and `android.support.v7.appcompat.R.a`. These methods are created by the obfuscator in the corresponding `R` resource classes. The tool uses multiple decryption methods because it employs several different encryption options and seems to arbitrarily pick one to encode a particular string.



(a) Original class structure (b) Legu class structure

Figure 2.6: Example of obfuscation performed by Legu

The second category of tools contains *packers*. A packer encrypts the original *classes.dex* file, then decrypts this file in memory at run time and executes it via

reflection using `DexClassLoader`. Tools such as APKProtect, Bangcle, and Legu belong to this category. Unlike the traditional obfuscations, they will not perform any bytecode modifications—rather, they hide the entire dex file. Figure 2.4 shows the corresponding process. The original *classes.dex* will be packed and a new artificial *classes.dex* file will be generated. The new file loads the original file in memory. The packed *classes.dex* is encrypted to prevent reverse-engineering analysis. Figure 2.6b shows the package and class structure after obfuscation with Legu. The original structure is shown in Figure 2.6a. New helper classes are created and the original classes are hidden.

The obfuscation process may provide some choices to the developers. For example, ProGuard, DashO and Allatori all have a configuration file to specify whether to perform different levels of obfuscation, or to provide a user-defined renaming style rather than the default one.

The next chapter describes an approach to automatically analyze a given obfuscated Android app with unknown provenance, and to determine which obfuscation tool/configuration was used to generate it.

CHAPTER 3: Android Obfuscator Identification

As discussed in Chapter 1, obfuscation can obstruct legitimate static analyses including malware detection, third-party library detection, clone detection, and security analysis. To tackle this problem, one could develop obfuscation-tailored variants of these techniques. Such analyses determine relevant properties of the obfuscation tool that was used to produce the analyzed app, and then use this knowledge inside specialized versions of analysis algorithms. The properties of the obfuscation tool are also useful for provenance analysis, digital forensics, and reverse engineering for test generation and performance analysis.

In this chapter we develop techniques to identify the obfuscator of an Android app for several widely-used obfuscation tools and for a number of their configuration options. The approach relies only on the Dalvik bytecode in the app APK. The identification problem is formulated as a machine learning task, based on a model of various properties of the bytecode (e.g., different categories of strings). This model reflects characteristics of the code that may be modified by the obfuscators—in particular, names of program entities as well as sequences of instructions. The main insight of our approach is that relatively simple code features are sufficient to infer precise provenance information. The specific contributions of this work are:

- We define the obfuscator identification problem for Android and propose a solution based on machine learning techniques. To the best of our knowledge, this is the first work to formulate and solve this problem.

- We identify a feature vector that represents the characteristics of the obfuscated code. We then implement a tool that extracts this feature vector from Dalvik bytecode and uses it to identify the obfuscator provenance information.

- We evaluate the proposed approach on Android apps obfuscated with different obfuscators, under several configurations. Our experiments indicate that the approach identifies the obfuscator with 97% accuracy and recognizes the configuration with more than 90% accuracy.

3.1 Overview

3.1.1 Machine Learning Model

The goal of our work is to identify the specific obfuscator that was used to create a given APK file. This information may enable the creation of obfuscation-tailored analysis, testing, and instrumentation, as well as provenance analysis for digital forensics. While the internal details of various obfuscation tools are not public, these tools are developed by unrelated development teams and are likely to differ substantially in low-level details. Our premise is that these differences manifest themselves in the obfuscated code. Thus, we aim to select suitable code features and to apply machine learning techniques based on them. In particular, we use labeled data to train a *classifier*. The classifier will map a vector of code features to a label representing one of several known obfuscation tools. Details of this approach are presented in Section 3.2.

3.1.2 Obfuscators Used in Our Study

There are many obfuscators and most of them are commercial. The cost of these tools is relatively high. As a result, in this study we selected free tools or free versions of paid tools. We used five different tools. ProGuard is provided by Google, Allatori is developed by a Russian company, and DashO is a product from an U.S. company. Legu and Bangcle are packers from China. ProGuard is integrated with Android Studio, the official IDE from Google. Allatori is a commercial tool, but it has a free educational version. Amazon, Fujitsu, Motorola, and hundreds of other companies worldwide have used Allatori to protect their software products from being reverse-engineered by business rivals [1]. The company developing DashO [30] has over $5,000$ corporate clients in over 100 countries. Legu is developed by Tencent, one of the largest IT companies in China. Currently there are more than 5 million developers using this company's platform and apps released by them will be protected by Legu [67]. Bangcle is developed by a company with the same name that provides security services to individuals, enterprises, and governments. The tool has served seventy thousand companies and has protected more than seven hundred thousand apps, which are installed on about seven hundred million devices [12].

3.2 Machine Learning for Obfuscator Identification

We use supervised learning in which a training set is used to create several classifiers. Later, a given APK (with unknown obfuscator provenance) is classified using these classifiers. The processing of an unknown APK is shown in Figure 3.1. In the first stage, the APK is classified in one of six categories based on obfuscator type. For the three obfuscators that allow customized configurations (ProGuard, Allatori,

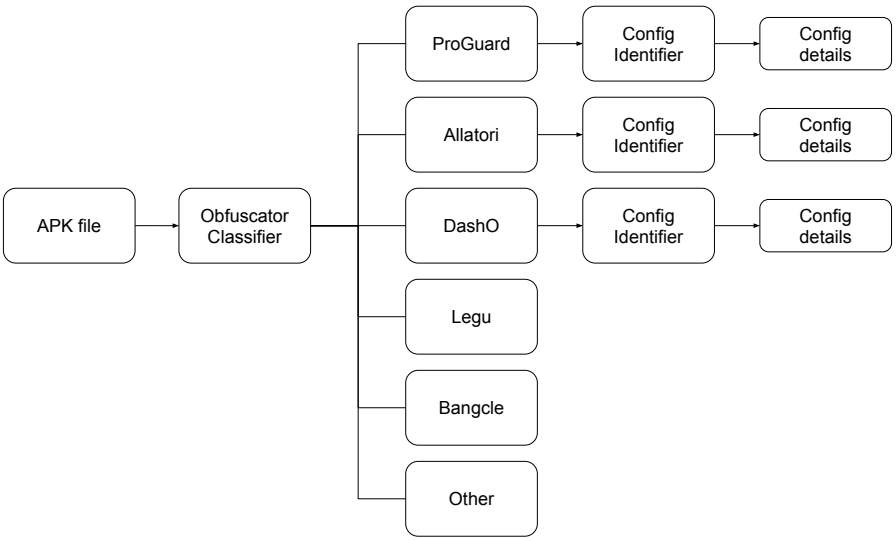


Figure 3.1: Workflow of obfuscator identification

and DashO), a second stage gathers information about the configuration under which the APK was obfuscated. This section describes how a classifier is generated for the first stage. The next section describes additional classifiers used in the second stage.

3.2.1 Training Set

For training, we obtained open-source apps from the F-Droid repository [37] and attempted to build them with Android Studio, which uses the Gradle build tool. A total of 282 apps were successfully built with Gradle. These apps were then obfuscated by us using various obfuscators and configurations. The source code was required because some obfuscators work on *.class* files. For each app, we created 6 *baseline APKs*: (1) without obfuscation, (2) obfuscated with ProGuard's default configuration, (3) obfuscated with Allatori's default configuration, (4) obfuscated with DashO's default configuration, (5) obfuscated with Legu, and (6) obfuscated with

Bangcle. Since ProGuard, Allatori, and DashO allow customization, we also created *customized APKs*. Using different configurations of ProGuard, 6 customized APKs were obtained per app. Similarly, 3 customized APKs per app were obtained using configurations of Allatori, and another 3 customized APKs were built using DashO. Details of the configurations used for training are presented in the next section.

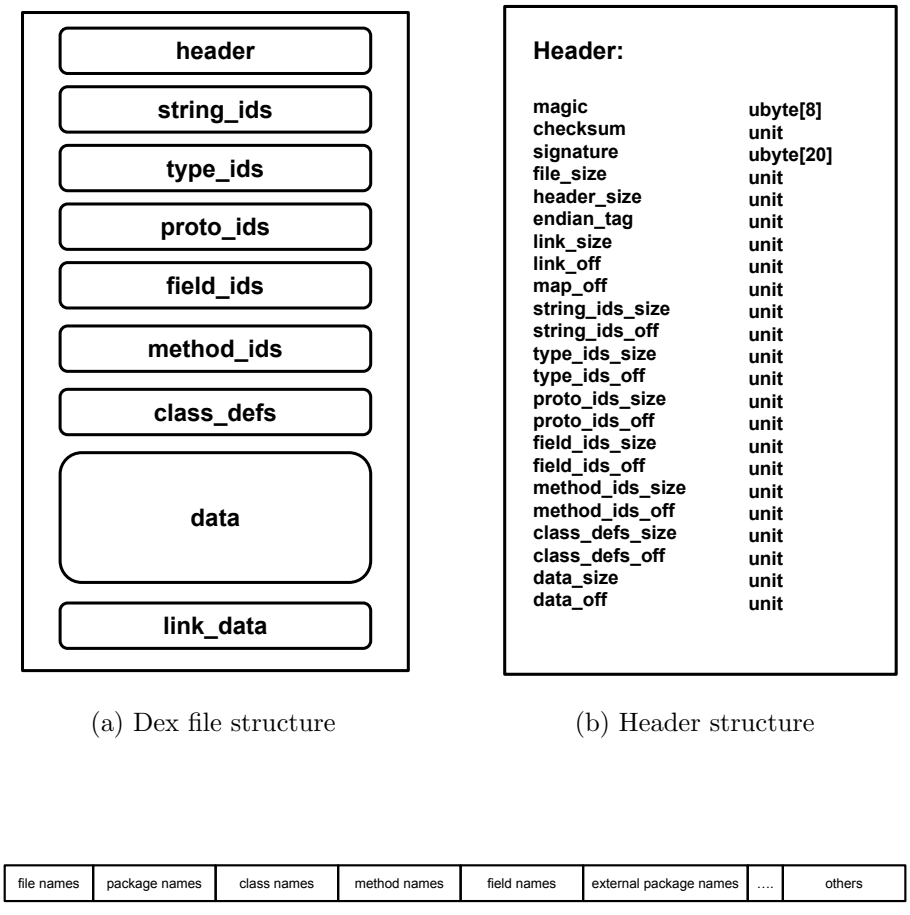
Each APK is labeled with a label from set $Labels = \{ProGuard, Allatori, DashO, Legu, Bangcle, Other\}$ where the last label was used for the unobfuscated APKs.[1] The features of each APK (described below) together with its label are used as input to the training phase.

3.2.2 Feature Dictionary

Features are obtained from the bytecode in *classes.dex*. The structure of this file is shown in Figure 3.2a. Many parts of the file are irrelevant for our purposes. For example, the structure of the header is shown Figure 3.2b; it contains the magic number, checksum number, file size, etc. This information varies across apps regardless of obfuscation.

In our approach we focus on the data section. We choose the *strings* in this section as candidates for features. These strings have different roles. For example, some are names of program entities such as classes and methods. For the code shown in Figure 2.5b, the set of class names is {`"Editor"`} and the set of method names is {`"a"`, `"remove"`, `"commit"`}. These distinctions between strings are informative.

[1]If the classifier produces *Other* (Figure 3.1), this means that the unknown APK was either unobfuscated or was obfuscated by an unknown tool.



(a) Dex file structure

(b) Header structure

Figure 3.2: Structure of feature dictionary

The feature dictionary is a collection of 10 sets of strings, as shown in Figure 3.2. All strings except file names are from the data section in the dex file. Four sets correspond to names of packages, classes, methods, and fields, respectively. These are program entities that are defined by the APK code. We also consider four corresponding sets, but for external entities defined by code outside of the dex file (e.g., by the Android framework). The reason for this separation is that an obfuscator may treat internal and external names differently. Finally, any other string occurring in the code

is considered to be in the set of "other" strings. For the code from Figure 2.5b, this "other" set is {`"this"`, `"arg0"`, `"o\u0018}\u0014h\u0018X\u001cg\t"`, `"u,s0i2d't\tu;@#c1"`}.

We developed a dex parser to collect this information from a given APK, based on the mapping relationship between ids and the strings in the dex file. The location and size of the ids can be obtained from the header. The file names are obtained from the structure of the APK archive, which can be examined using the standard `aapt` tool ("Android Asset Packaging Tool", used for Zip-compatible archives such as APK files). This information may be useful when the obfuscator adds its own helper library files.

For each obfuscated APK A_i from the training set, 10 sets $S_{i,j}$ of strings are extracted ($1 \leq j \leq 10$) based on the categories described above (illustrated in Figure 3.2). The following processing of these sets is performed to obtain the final feature dictionary. First, if several APKs $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$ are obtained from the same original app a using the same obfuscator l, their corresponding string sets are merged using set intersection. The result is a collection of 10 sets $S_j^{a,l}$ for $1 \leq j \leq 10$, where $S_j^{a,l} = \cap_{p=1}^{k} S_{i_p,j}$. The intent is to use strings that are configuration-independent. After this step, for each pair (a, l) where a is an app and l is an obfuscator, there is a collection of 10 string sets $S_j^{a,l}$.

The next step is to trim these string sets as follows. For each obfuscator l and each string category j, we consider the union of all $S_j^{a,l}$. The resulting set F_j^l can be thought of as a feature dictionary specific to this obfuscator l and this string category j (e.g., class names). Each such F_j^l is trimmed by removing rarely-occurring strings. Specifically, if the percentage of sets $S_j^{a,l}$ containing a string s is less than a particular

threshold, s is removed from F_j^l. Our experiments indicated that a threshold of 15% produces substantial reduction in the size of the feature vector without significant reduction in prediction accuracy.

To further reduce dimensionality, features that are covered by other ones will be ignored. Specifically, consider two string $s, s' \in F_j^l$. If the set of apps $\{a \mid s \in S_j^{a,l}\}$ is a proper subset of $\{a \mid s' \in S_j^{a,l}\}$, then s is less informative than s' and will be removed from F_j^l.

After these trimming techniques are applied, the feature dictionaries for all obfuscators are reduced from thousands of elements to hundreds of elements. Trimming also reduces the training time for the classifier by about a factor of two.

The final feature dictionary is a collection of 10 string sets $\hat{F}_j$ for $1 \leq j \leq 10$. Each set is computed as

$$\hat{F}_j = \bigcup_l F_j^l - \bigcap_l F_j^l$$

The second term removes strings that always appear regardless of the obfuscator l, since such strings are not useful for distinguishing among obfuscators.

The feature vector used for training and subsequent classification has 10 sub-vectors, each corresponding to one set $\hat{F}_j$. If a string $s \in \hat{F}_j$ occurs in category j inside an APK, the corresponding element of the j-th sub-vector of the feature vector is 1; otherwise the element is 0.

3.2.3 Training a Classifier

After constructing the feature vector for each APK in the training set, we build a corresponding classifier. In our scenario, we have a multi-class classification problem—that is, APKs are classified into one of several (more than two) classes. The classes

are defined by set *Labels* described earlier. We do not consider the case when more than one obfuscator is used for the same APK, because typically in practice developers will adopt one obfuscator tool. Therefore, a *one-vs-rest* classifier is appropriate for our purposes. This strategy trains a single classifier per class. Each class is fitted against all other classes. The concrete type of classifier we use is linear Support Vector Machine (SVM). A SVM finds a weight vector w that defines a decision boundary in the feature space which best separates two different classes. The distance from a particular example to that boundary is the *margin* and is defined as $w^T x$, where x is the feature vector. In such a binary classifier, each instance is assigned to class $+1$ or -1 depending on the sign of the margin.

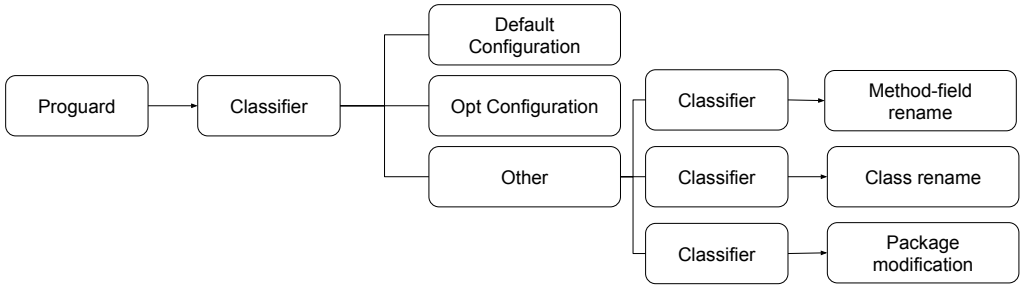
For one-vs-rest classification, a binary classifier can be extended to n classes. This standard approach determines n weight vectors $\{w_1, \ldots, w_n\}$ by partitioning the data into two groups, one for the current class and the other for everything else. Given these vectors and a new instance with feature vector x, we choose the class that maximizes the margin:

$$\text{argmax}_{1 \leq k \leq n} \, w_k^T x$$

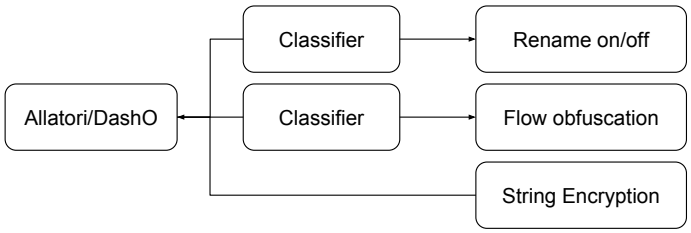
In addition to linear SVM, we also experimented with other kinds of classifiers; details are presented in the evaluation section. Our experience is that linear SVM provides a good balance between accuracy and running time.

3.3 Obfuscator Configuration

After obtaining the type of obfuscator, we determine characteristics of the obfuscator configuration, if applicable. Because the configuration may significantly change



(a) ProGuard config identification workflow



(b) Allatori and DashO config identification workflow

Figure 3.3: Workflow of configuration characterization

the behavior of an obfuscator, this knowledge may be useful for APK analysis and reverse engineering. In our case, ProGuard, Allatori, and DashO provide customization: the obfuscator user can define an XML file to modify the default tool behavior.

3.3.1 ProGuard Configuration

The configuration analysis for ProGuard is shown in Figure 3.3a. In the first step, three types of configurations are differentiated. The *default* one and the *default-opt* one are provided directly in Android Studio. The difference is that the opt configuration performs aggressive optimizations of the APK to reduce code size and to improve run-time performance. The third option *other* means the developer customized the configuration. There are four kinds of possible modifications: (1) customizing field

and method names, (2) changing class names, (3) package flattening, which moves all packages into a single user-specified parent package, and (4) repackaging classes, which moves all classes into a single user-specified package. The last two transformations will change the structure of the package hierarchy and we treat them as one category in the classification process. All modifications depend on user-defined values. For example, the tool user can provide a list of arbitrary strings as candidates for class names, or can specify a custom package name.

During training we can only observe specific instances of these user-defined names. The resulting classifiers can subsequently determine, for a given (never-seen-before) ProGuard-obfuscated APK, which settings have been customized. As shown in Figure 3.3a, the approach first identifies whether the configuration is one of the two default ones or is user-customized. In the third case, we determine which specific settings have been changed.

The training process for both stages is similar to what was described in the previous section, but with different feature vectors. For each app a, we constructed 6 APKs using ProGuard: default, default-opt, customized field/method names, customized class names, package flattening, and repackaging. The labels used to label these APKs are from set $\{$*default*, *default-opt*, *field&method*, *class*, *package*$\}$. The last label is used for both package-related transformations. In our training set, APKs with the last 3 labels are constructed with arbitrary new names defined by us. The resulting classifiers are used to process future APKs in which the user-specified names are different from the ones used during training.

We consider a feature dictionary which is a combination of the features of APKs with labels *default* and *default-opt*. The remaining APKs are not used because they

are dependent on customizable user-defined names. For defining the first classifier in Figure 3.3a, the APK labels are mapped to the simplified label set {*default*, *default-opt*, *other*}. For the remaining three classifiers shown in Figure 3.3a, the APK labels are mapped to the two labels *true* and *false*, based on whether the particular setting is the default one or not. As before, linear SVMs are used for the classifiers.

3.3.2 Allatori/DashO Configuration

The analyses of Allatori and DashO configurations are shown in Figure 3.3b. The two tools are very similar in terms of configuration options and are analyzed using the same approach; below we describe only Allatori processing.

We identified three categories of customization settings. The *rename* switch controls whether the tool will obfuscate the names of packages, classes, etc. This setting is different from the one used in ProGuard as it does not require used-defined names. If turned on, the identifiers will be obfuscated using logic internal to the tool and the package hierarchy will also be changed. The process of identifying this property is the same as with ProGuard. *String encryption* is a setting to determine whether the constant strings in the code are encrypted. If turned on, the tool will identify all string data and encode it as illustrated in the earlier example from Figure 2.5b. The tool also adds code to decode the strings at run time. Our learning cannot handle this case well, because the approach does not model the "naturalness" of constant strings. One simple approach to detect this setting could be to examine the code for calls to tool-specific string decoding methods.

Another customization category is related to *flow obfuscation*. If set, the tool will change control-flow features of loops and branches. Here we need a different

Algorithm 1: FeatureDictionaryForAllatoriOrDashO

Input: $APKs = \{apk_i\}$: APKs obfuscated by Allatori/DashO with and without flow obfuscation
Input: N

```
1 feature_InstSequences ← ∅
2 foreach apk ∈ APKs do
3     foreach class ∈ apk do
4         foreach method ∈ class do
5             feature_InstSequences ← feature_InstSequences ∪ all instruction sequences of length N
```

feature dictionary, constructed as described in Algorithm 1. The process is similar to considering N-grams in natural language processing.

For each obfuscated APK we compute the set of consecutive instruction sequences with length N. Since the obfuscation may affect the names of instruction operands, only the operator is considered. The sequences are obtained using the Androguard tool. Consider an app a, its baseline obfuscated APK, and its 3 APKs obtained from customized configurations. Each APK defines a set of sequences. We consider the union of these 5 sets and remove from it the intersection of the 5 sets. The result is a set S^a of configuration-dependent sequences. Next, rarely-occurring sequences are filtered. If, over the entire set of apps a, the percentage of sets S^a containing a sequence s is less than the threshold from Section 3.2.1, s is removed from all S^a. The union of the final S^a is the feature dictionary.

3.4 Evaluation

We evaluated our approach on a number of Android apps. The evaluation considers the following questions: (1) Do the selected features represent the characteristics of these five obfuscators so that the classifiers can predict the correct label with high

accuracy? (2) Are linear SVMs suitable in terms of accuracy and time cost, compared to other possible classifiers?

3.4.1 Data Set

We obtained 282 apps from F-Droid [37] and created several APKs for each app, as described in Section 3.2.1. In total, we successfully generated about 2600 APK files, because in some cases the obfuscator(s) failed.

3.4.2 Methodology

The data set has to be split into training set and testing set. The training set is used to build the parameters of the model, while the testing set is chosen for evaluating the performance. Using standard 10-fold cross validation, we randomly divides the apps into 10 equally-sized subsets $Apps_i$ for $1 \leq i \leq 10$. The experiment is executed 10 times, once for each i. In each run, set $Apps_i$ is used as the testing set and the remaining apps are used as the training set. The predicted labels are then compared with the actual ones. The machine learning framework used in our evaluation is *scikit-learn* [104]. This toolkit provides various classifiers, including linear SVMs.

3.4.3 Obfuscator Identification

Table 3.1 shows the accuracy and F_1 score measurements obtained using 10-fold cross validation. The table shows the mean values and standard deviations of the 10 runs. The accuracy, for one run, is the ratio of number of testing APKs with correctly-predicted labels to the total number of testing APKs. For multi-class classification, two types of F_1 scores are typically computed: *micro* and *macro* [83]. The micro F_1 score considers the total number of true positives, false negatives, and false positives.

	Accuracy	F_1 micro	F_1 macro
mean	0.975	0.975	0.968
SD	0.007	0.007	0.009

Table 3.1: Accuracy and F_1 score for obfuscator identification

The macro metric computes a value for each label and uses the unweighted mean. The corresponding equations are shown below.

$$precision = \frac{TP}{TP + FP} \quad recall = \frac{TP}{TP + FN}$$

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

$$F_{1macro} = \frac{\sum_{i=1}^{N} F_{1i}}{N}$$

$$precision_{micro} = \frac{\sum_{i=1}^{N} TP_i}{\sum_{i=1}^{N} TP_i + \sum_{i=1}^{N} FP_i}$$

$$recall_{micro} = \frac{\sum_{i=1}^{N} TP_i}{\sum_{i=1}^{N} TP_i + \sum_{i=1}^{N} FN_i}$$

$$F_{1micro} = 2 * \frac{precision_{micro} * recall_{micro}}{precision_{micro} + recall_{micro}}$$

where TP is the number of true positives, FP is the number of false positives, FN is the number of false negatives, and N is the number of labels. The results indicate that our approach has high accuracy and F_1 score.

To evaluate the performance for each individual obfuscator, we also collected data about precision and recall for each label. The results are shown in Table 3.2 and can be summarized as follows. Apps that are not obfuscated are detected correctly. Legu and Bangcle seem to have unique characteristics and our approach identifies them correctly. For ProGuard, Allatori, and DashO the model achieves very high

Precision						
	ProGuard	Allatori	DashO	Legu	Bangcle	Other
mean	0.977	0.979	0.993	1.000	1.000	1.000
SD	0.005	0.003	0.008	0.000	0.000	0.000
Recall						
	ProGuard	Allatori	DashO	Legu	Bangcle	Other
mean	0.976	0.977	0.996	1.000	1.000	1.000
SD	0.005	0.004	0,007	0.000	0.000	0.000

Table 3.2: Precision and recall for obfuscator identification

ProGuard stage I						
	Acc.		F_1 micro		F_1 macro	
mean	0.938		0.938		0.917	
SD	0.022		0.022		0.031	
ProGuard stage II						
	field&method		class		package	
	Acc.	F_1	Acc.	F_1	Acc.	F_1
mean	0.988	0.963	0.989	0.967	0.940	0.909
SD	0.012	0.039	0.006	0.020	0.030	0.045

Table 3.3: Accuracy and F_1 for ProGuard configuration

accuracy. Since these tools have some similar obfuscation strategies, in rare cases the classifier may be unable to distinguish them correctly. Nevertheless, the overall measurements indicate that our model has a very high chance to correctly identify the type of obfuscator, for the five types covered by our implementation.

3.4.4 ProGuard Configuration Identification

If the obfuscator is identified as ProGuard, the next step is to characterize its configuration as illustrated in Figure 3.3a. In the first stage, we determine how the

Precision						
	ProGuard stage I			ProGuard stage II		
	default	default-opt	other	field&method	class	package
mean	0.984	0.939	0.936	0.995	0.998	0.905
SD	0.048	0.069	0.028	0.014	0.015	0.032
Recall						
	ProGuard stage I			ProGuard stage II		
	default	default-opt	other	field&method	class	package
mean	0.940	0.783	0.979	0.942	0.935	0.923
SD	0.073	0.088	0.016	0.056	0.064	0.040

Table 3.4: Precision and recall for ProGuard configuration

configuration is mapped to a label from {*default*, *default-opt*, *other*}. For "other", a second stage determines a *true*/*false* label in each of the following 3 categories: field&method, class, and package. Here *true* means that the category was customized by the ProGuard user. The accuracy and F_1 scores for both stages are shown in Table 3.3. Since there are three labels in the first stage, we consider both micro and macro F_1 score. The labels in the second stage are binary and there is only one F_1 score. The results shown in the table indicate that our approach characterizes the ProGuard configuration with high accuracy.

The precision and recall of each stage are shown in Table 3.4. The recall of *default-opt* is relatively low. By comparing the ProGuard XML configuration files of *default* and *default-opt*, we observed that the difference between them is not very significant. Configuration *default-opt* will perform further performance optimizations and some related small code changes. In some scenarios, the optimization may not be available or it cannot cause an apparent difference. This may be the reason why for some apps this configuration is not identified. In addition, some apps are

	field&method	class	package
mean	0.957	0.960	0.945
SD	0.006	0.002	0.005

Table 3.5: Accuracy with never-seen custom names

very simple and have few classes. For correctness, ProGuard will preserve some classes. For example, the app entry class defined in the *manifest.xml* file will not be obfuscated. As another example, subclasses of framework class `android.view.View` require some class members to be unchanged. If the app is simple, the differences between *default* and *default-opt* may be insignificant and unrecognizable, which may cause some misclassification.

ProGuard allows the user to specify custom strings for class names, field names, method names, or package names. In the training process, we use a specific set of random strings (defined by us) for these values. To ensure that the classification can be successfully applied to APKs in which different user-defined obfuscated names were used, we performed the following experiment. We created another set of random strings, different from the string set used for training. During each run of the cross validation, the model is also tested against APKs obfuscated with this second set of string (i.e., with strings that have not been observed during training). The mean and standard deviation of the accuracy are shown in Table 3.5. The accuracy is high for all three categories of names, indicating that our model is able to detect the change of these configuration settings regardless of the specific string values being used.

3.4.5 Allatori and DashO Configurations

The configuration characterization for Allatori and DashO has only one stage. As mentioned earlier, we do not consider string encryption, but focus on renaming and control-flow modifications. For both of these categories, a *true*/*false* label is determined to indicate that the configuration setting has been changed by the user. To compute the feature vector for the control-flow configuration, we define a feature dictionary containing all instruction subsequences of length 2, 3, and 4. The accuracy and F_1 scores are shown in Table 3.6. The results indicate that with high confidence we can identify whether these two Allatori and DashO configuration options have been modified.

The precision and recall are shown in Table 3.7. The high values indicate that model typically correctly identifies the Allatori and DashO configuration settings. Misclassifications have the same root causes as with ProGuard. For example, for correctness, the tool has to preserve some elements of the original code. As another example, the tool will change the order of instructions inside a loop; if there are no loops in the code, no control-flow changes will be observed.

3.4.6 Comparison of Classifiers

Linear SVMs are only one of many available approaches that can be used for classification. To compare different classifiers, we considered 8 popular options: k-nearest neighbors, decision trees, linear SVMs, random forests, multi-layer perception, AdaBoost, Gaussian naive Bayes, and logistic regression. All are available as part of the *scikit-learn* machine learning toolkit used in our experiments. The experimental machine uses Ubuntu 16.04 with Intel Core i7-4770 CPU 3.40GHz and 16GB RAM.

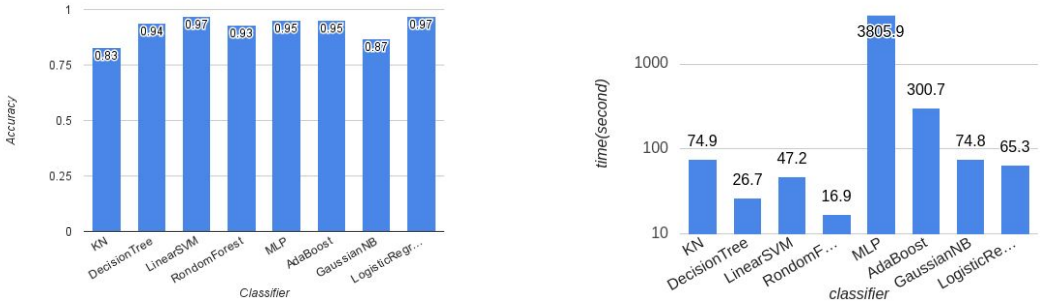
Allatori				
	Rename		Flow	
	Accuracy	F1	Accuracy	F1
mean	0.994	0.987	0.933	0.931
SD	0.007	0.014	0.003	0.033
DashO				
	Rename		Flow	
	Accuracy	F1	Accuracy	F1
mean	0.981	0.981	0.979	0.984
SD	0.014	0.013	0.015	0.015

Table 3.6: Accuracy and F_1 for Allatori/DashO config

Allatori				
	Precision		Recall	
	Rename	Flow	Rename	Flow
mean	0.993	0.936	0.989	0.929
SD	0.015	0.053	0.017	0.035
DashO				
	Precision		Recall	
	Rename	Flow	Rename	Flow
mean	0.977	0.974	0.986	0.976
SD	0.023	0.034	0.021	0.024

Table 3.7: Precision and recall for Allatori/DashO config

The accuracy for different classifiers is shown in Figure 3.4a. The y-axis is the mean of accuracy for each stage including both obfuscator and configuration identification. The x-axis is the type of classifier. The results indicate that the proposed feature vector works well for most classifiers. Among them, linear SVMs and logistic regression have the best performance. Figure 3.4b illustrates the total running time of 10-fold cross validation for both obfuscator and configuration identification. This includes



(a) Accuracy of classifiers

(b) Running time of classifiers

Figure 3.4: Performance of different classifiers

the time to create the classifier (in training) and to apply it (in testing), given the feature vectors. The y-axis is the time in seconds. Based on these measurements, linear SVMs has the best accuracy while still exhibiting good efficiency. The time to extract the feature dictionary and feature vectors from the APKs (total for all 10 runs in 10-fold cross validation) is about 12 hours.

3.4.7 Case Study: Google Play Apps

We performed a case study of applying our classifiers to a set of popular apps from the Google Play store, which is the largest and most influential app store. We do not have ground truth to measure the quality of our classification, since we do not know the obfuscator provenance of apps from the store. Nevertheless, under the assumption that the classification has high accuracy, this study presents insights into the use of obfuscation techniques in widely-used apps. The apps were crawled from the Play store from May 2016 through August 2016. We collected the top 100 free

apps for each store category (e.g., education, finance, etc.) and obtained a total of 2389 APKs; some apps appear in multiple categories.

The analysis of these APKs is summarized in Table 3.8. The number in each column is the number of apps classified as being obfuscated by the corresponding tool. More than 55% of the APKs (1330) are identified as being obfuscated by one of the tools we consider. The remaining 1059 APKs may be obfuscated by other obfuscators or not obfuscated at all. The number of obfuscated apps is relatively high. One reason may be that most of the top popular apps are developed by companies instead of individuals. Companies pay more attention to security and intellectual property. For the relative usage of obfuscators, ProGuard is the most widely used tool (about 88% of the APKs). This is not surprising, since ProGuard is already integrated with Android Studio and requires little extra effort to configure and use. Moreover, Google provides an official tutorial on ProGuard and recommends the developers to use it before releasing their apps. The two Chinese tools Legu and Bangcle are rarely used in the Google Play store. The reason may be that most of their users are from China since both tools only have interfaces written in Chinese. The audiences of these Chinese developers are mainly from China. However, app users in China cannot access Google Play.

For apps using ProGuard and Allatori, we further analyzed their configurations. The results for ProGuard are shown in Table 3.9. About 70% of the apps use the default or default-opt configuration. The default configuration is the most frequently used. This may be because the default-opt setting is well known to be unsafe due to code optimizations that are not correct in all situations. Table 3.9 also shows the number of apps that customize the corresponding settings. About 65% of apps

	ProGuard	Allatori	DashO	Legu	Bangcle	Other
Num.	1170	145	0	15	0	1059

Table 3.8: Obfuscators for Google Play apps

ProGuard stage I			
	default	default-opt	other
Num.	555	236	379
ProGuard stage II			
	package	class	field&method
Num.	246	41	54

Table 3.9: Apps obfuscated with ProGuard

that do not use default/default-opt employ package modifications. This setting is the most convenient one to use, because it does not require a full list of candidate values, but only a customized package name. The other two settings are not frequently customized. From the 145 apps obfuscated with Allatori, 26 change the renaming and 18 use the flow option.

3.4.8 Limitations

The generality of the experimental observations should be interpreted in the context of several limitations of the proposed techniques. First, we only consider five obfuscators. There are other tools that may use different obfuscation techniques. Second, our training data may have limited generality that does not allow some characteristics of the obfuscators to emerge. Although the open-source F-Droid apps used for training cover a wide range of target domains and developers, they may differ from closed-source apps in some characteristics that affect the accuracy of the approach.

Finally, because we focus on features from the string tables of the dex files, other relevant program properties are not represented. For example, obfuscation features such as function merging and string encryption cannot be captured by the proposed approach. Despite these limitations, we consider these experimental results to be promising initial evidence that automated and precise obfuscator provenance analysis for Android is feasible and efficient.

3.5 Summary

In this chapter, we propose a simple, efficient, and effective approach for provenance analysis of obfuscated Android applications. Using features extracted from APKs, we construct several classifiers that determine which obfuscator was used and how it was configured. Although the approach has some limitations, it exhibits high accuracy on apps from F-Droid, and provides insights about the use of obfuscation in popular Google Play apps. The information extracted with the proposed techniques can be potentially used to improve and refine a variety of existing analyses and tools for Android.

CHAPTER 4: Obfuscation-Resilient Detection of Third-Party Libraries and App Clones

Third-party libraries are heavily used in Android apps. As described in Chapter 1, there are several reasons why it is important to identify instances of third-party libraries in a given app. Such detection is significantly more complicated if the analyzed app was obfuscated. A related problem is clone detection—deciding whether a given app is a cloned version of some other app. Clone detection is a valuable tool for identifying repackaged apps that violate the intellectual property of the original app's developer, or apps that introduce malicious additional functionality inside a legitimate app. Techniques for clone detection are very similar to ones for detection of third-party libraries. In both cases, obfuscation-resilient detection techniques are essential. As we argue later in this chapter, existing library detection and clone detection tools do not have sufficient resilience to obfuscation. This conclusion is based on a comprehensive study of several popular obfuscators and the techniques they use.

4.1 Background

4.1.1 Android Obfuscation Techniques

In our study of obfuscator properties, we investigated techniques used in three tools: ProGuard [87], Allatori [1] and DashO [30]. As discussed in the previous chapter, these tools are widely used in Android development.

	ProGuard	Allatori	DashO
Identifier renaming	Yes	Yes	Yes
Method renaming	Yes	Yes	Yes
Class renaming	Yes	Yes	Yes
Package renaming	Yes	Yes	Yes
Code addition	Yes	Yes	Yes
Code removal	Yes	Yes	Yes
String encryption	No	Yes	Yes
Repackaging/Package flattening	Yes	Yes	Yes
Control flow modifications	No	Yes	Yes
Customized configuration	Yes	Yes	Yes

Table 4.1: Obfuscator features

The features of each obfuscator is summarized in Table 4.1. All three can obscure identifier names, method names, class names, and package names. Moreover, the values of the new names can be customized, which means that developers can have their own unique renaming schemes. In addition, the tools can modify the package hierarchy, in particular (1) repackaging classes from several packages into a new, different package and (2) flattening the package hierarchy. Both Allatori and DashO can modify the code's control flow and encrypt the constant strings in the program. Finally, the tools can remove unused code ("Code removal" in the table) and add their own utility methods in the new code ("Code addition").

Note that all three tools can be easily configured to merge the app code with the code of included third-party libraries, and obfuscate this combined code. Such a setup obfuscates the interfaces between the app and its libraries, hides the library features being used by the app, and reduces code size (e.g., unused library classes/methods can be removed). However, this obfuscation makes library detection significantly harder, as described below.

4.1.2 Existing Library Detection Tools for Android

Given the APK for an (obfuscated) Android app, the goal of library detection is to determine which components of the app belong to some third-party library that was used by the app developers. The granularity of the answer is typically considered at two levels: package level and class level. In package-level matching, a package in the package tree of the app is determined to be from some library. In class-level matching, a class in the app code is detected to be a library class. Package-level matching has a number of limitations, as described below. For the purposes of subsequent analyses (e.g., app clone detection [42]), class-level matching is both more general and more useful.

There are currently two approaches for recognizing third-party libraries in Android apps. The first is to identify a component that occurs in many apps, and to consider it to be an instance of some unknown library. The specific library that was the original source of the component remains undetermined. The second approach is based on a database of known libraries. In this case, given an never-before-seen app, the analysis determines which libraries from the database are present in the app.

Detection of Unknown Libraries

Several approaches perform library detection without having a database of known libraries. Such approaches divide an app into components which are regarded as library candidates. Then a similarity metric or a feature-based hashing algorithm is used to classify these candidates. If a group of similar candidates exists in many different apps, components in that group are considered instances of the same unknown library.

An advantage of this approach is that there is no need to maintain a library database. One example from this category is the approach used by Chen et al. [20], which mined libraries from a large number of apps. However, when obfuscation is considered, this approach cannot perform well because several key assumptions are violated. The first assumption is that all instances of a library included by different apps have the same package name. This assumption is the basis of clustering algorithms used in similarity-based library identification. However, using off-the-shelf obfuscators, it is easy to violate this assumption.

Some researchers have considered the possibility that using package names makes library identification less robust. A recent tool called LibRadar [74] uses an algorithm that takes obfuscated package names into consideration. LibRadar classifies library candidates through feature hashing and therefore does not need package-name-based clustering. However, LibRadar recognizes library candidates according to the directory structures of packages. In particular, it requires a library candidate to be a subtree in the package hierarchy. This is another assumption that is easily violated through standard transformations available in obfuscation tools (as illustrated in Table 4.1). LibD [69] is another tool that does not rely on the hierarchy of the package tree, but instead uses the relationships among packages—e.g., inclusion and the inheritance relationships between classes across different packages. However, these relationships can be dramatically changed by obfuscation because classes from several packages may be merged into one single package.

Tools from this category (e.g., WuKong [116] and AnDarwin [27]) may also use features such as the number of different types of instructions, a method's control-flow graph, or the number of API calls, in order to define hashes for methods or

classes. These hashes are compared during matching, with hash equality used as evidence of a precise match. This is problematic because obfuscation may (1) modify a method's control-flow graph, (2) remove unused methods and classes, and (3) add new utility methods. All three of these changes are illustrated in Table 4.1. Such changes modify method hashes and class hashes. To summarize, existing work on detection of unknown libraries is not resilient to obfuscation because it lacks prior knowledge of possible libraries and the detection is entirely based on obfuscation-sensitive similarity metrics.

Detection Using a Repository of Known Libraries

The alternative approach, which relies on knowledge of existing libraries (using some pre-assembled repository of such libraries), is more promising when dealing with obfuscation. A representative example of a tool from this category is LibScout [9]. This tool's matching is based on the package structure and hashes of classes in the package. Class hashes themselves are based on hashes of simplified ("fuzzy") forms of method signatures. Although robust against control flow modification and package/-class/identifier renaming, LibScout is not fully resilient to obfuscation. For example, flattening the package hierarchy would still defeat this approach because the package hierarchy is modified and the boundaries between app and library code become blurred. Another problem is the removal of unused library code, or the addition of obfuscation-tool-specific utility methods. As a result of these changes, it is common for an app to contain a strict subset of the set of classes and methods that are observed in the library code inside the database. This happens when the obfuscation tool removes library methods and classes that are not used, directly or transitively, by the app code. This causes mismatches for class hashes and the ultimate failure of

library detection. Our experience with various obfuscation tools [119] confirms that such code changes are common in practice.

Currently, the most sophisticated tool in this category is LibDetect, which is a component of the CodeMatch tool for app clone and repackage detection [42]. The evaluation of LibDetect indicates that it outperforms alternative approaches [68, 74]. This approach uses five different abstract representations of a method's bytecode to match app methods against library methods. Then a mapping between an app class and a library class is established according to method matches. The approach is resilient to many obfuscation techniques, including package renaming and removal of unused library code. However, the bytecode in a method's body can the changed in a variety of ways during obfuscation. A few examples of such changes are adding dead code (unreachable statements as well as spurious conditionals with dead branches), replacing statements with equivalent constructs (e.g., changing an if-then to a try-catch), and modifying expressions to use different operators (e.g., replacing multiplication with bitshift). These bytecode changes can affect the code representations used by LibDetect and the matching based on them. Section 4.6 presents a detailed comparison of the library detection performance of LibDetect and Orlis.

There are other studies that use pre-computed information about known libraries to detect libraries in Android apps. For example, AdRisk [45] proposes to identify potential risks posed by advertisement libraries; they use a whitelist of such libraries. Book et al. [16] also use a whitelist of advertisement libraries to investigate library changes over time. For Android app clone detection, Chen et al. [19] use a library whitelist to filter third-party libraries when detecting app clones. Because such approaches only compare package names, they cannot handle aggressive obfuscation.

4.1.3 Existing Clone Detection Tools for Android

A significant body of work exists on the problem of detecting cloned/repackaged Android apps. In general, these techniques can be divided into two categories.

One approach is to consider the app GUI. Zhang et al. [127] propose a user-interface-based technique for app repackaging detection. The detection is based on a feature graph which captures users navigation behavior across app views. This approach performs sub-graph similarity measurements, which could be computationally expensive. Soh et al. [106] define a method to detect Android app clones based on the analysis of user interface information collected at run time. However, running a large number of apps in devices or emulators is not scalable. Crussell et al. [27] develop a tool to detect clones based on the edit distance of layout trees, which is generated from the layout files of activities. This method as well as all other GUI-based approaches do not filter out the effects of third-party libraries, which may impede their effectiveness.

Another approach is to use code features. Similarly to traditional software clone detection, the program control-flow graph or dependence graph can be used for Android clone detection. Chen et al. [19] consider the centroid of the control-flow graph; clone detection is based on comparison of centroids. Crussell et al. [26] detect clones by comparing program dependency graphs between methods in candidate apps. However, the control flow may be modified by obfuscation, which in turn could affect control dependencies in the PDG.

A variety of other features have also been considered. Shao et al. [101] examine app resources to detect app repackaging. The resources along with their statistical features and structural features are extracted from major packages and the manifest

file. The features are based on resources such as activities, permissions, etc. Structural features, which are more expensive to compute, contains the layouts and event handlers of resources. Wang et al. [116] propose a two-phase detection approach that includes a coarse-grained detection phase to identify suspicious apps by comparing light-weight static semantic features such as the API calls, followed by a fine-grained phase to compare more detailed features including the number of variables in different contexts. Third-party libraries are filtered out before the analysis. Zhou et al. [132] design a detection tool using fuzzy hashing of the operators in Dalvik bytecode. The same authors [131] propose a module decoupling technique to partition an app's code into primary and non-primary modules. They developed feature fingerprinting to extract various semantic features such as API calls, permissions, etc. (from primary modules) and convert them into feature vectors. The feature vectors are compared using Jaccard distance. Hanna et al. [48] represent Android apps by hashing the k-gram features of code sequences and the hash similarity is measured using Jaccard similarity. Zhauniarovich et al. [130] compute hash values for all files referenced in the manifest of an APK, and compare them with the extracted hashes of a potentially repackaged app. The features and metrics described above are vulnerable to common obfuscation techniques—for example, by the addition and removal of code and by the transformation of control flow.

Guan et al. [46] introduce an approach to detect repackaged apps by comparing the input and output states of core functions in the app. This technique tries different permutations of input parameters and checks the equivalence of outputs generated by symbolic execution. This approach is time consuming and may be impractical for large-scale apps analysis.

Currently, the-state-of-the-art clone detection tool for Android is CodeMatch [42]. The tool employs LibDetect to filter out third-party libraries before the clone detection. CodeMatch represents each app class by combining its own Android-API type list, the type list of all its fields, and an abstract code representation of all its methods. Fuzzy hashes are created based on this information. The evaluation of CodeMatch shows that it achieves better performance than several other approaches [19, 127, 130, 132]. However, similarly to LibDetect, some obfuscation techniques that impact the code representation of a method body can thwart the matching of CodeMatch. A detailed comparison on the performance between CodeMatch and and our proposed clone detection approach is performed in Section 4.6.

To summarize, existing tools for library detection and clone detection are not highly resilient against the code modification features that are easily available in popular obfuscators. This motivates our work to develop new library detection and clone detection that better handles a wide range of obfuscation transformations.

4.2 Assumptions about Obfuscation

Our approaches for library detection and clone detection are based on several assumptions derived from our studies of Android obfuscators (Section 4.1). Prior work on library detection and clone detection typically does not define or validate its assumptions on the properties of obfuscation. As a result, it is hard to reason about the generality of various techniques and to perform systematic comparisons between them.

4.2.1 Method-Level Assumptions

We assume that a method is either included in the app as a separate entity, or not included at all. This means that during obfuscation, a method is neither split into several new methods, nor merged with another method. We also assume that the number and types of method formal parameters are preserved, as well as the return type. Of course, we need to assume that the name of the method could have been modified by the obfuscator. Similarly, if a method parameter/return type is defined by a class, it is possible that this class was renamed by the obfuscator—both in the class definition and in the method signature.

Our approach also assumes that regardless of changes to the method body (e.g., modifications to the method's control-flow graph, changes to statements and expressions, addition of unused code), the *calling relationships between two methods are preserved*. Specifically, we assume that if method m_1 contains a call site that may call m_2 at run time, and the obfuscator includes m_1 in the app, it will also include m_2 and will preserve the corresponding call site inside m_1. We have not observed obfuscators that violate this assumption; one reason may be that removal of such call sites (e.g., via inlining) is complex to implement and could introduce semantic errors due to polymorphic calls, reflection, and dynamic class loading.

4.2.2 Class-Level Assumptions

Another group of assumptions relates to the properties of classes. As with methods, we assume that a class is either fully excluded from the app, or included as a separate entity and not split/merged. We also assume that classes may have been renamed, but the class hierarchy is preserved—that is, if a class C is included in the

app, so is the chain of C's transitive superclasses. In addition, we assume that some methods from C may have been removed by the obfuscator, but the ones that are preserved are still members of C after obfuscation. Note that the obfuscator may have added new methods to C—for example, we have seen cases where an obfuscator adds utility methods for string decryption.

4.3 Orlis: An Obfuscation-Resilient Approach for Library Detection

We developed a library detection approach that belongs to the second category described in the Chapter 2: a given obfuscated app is compared against a repository of known libraries, and high-likelihood matches are reported to the user. However, unlike existing tools, we do not consider the app package structure and the detailed method body bytecode, because they can be modified significantly by obfuscation. The output of our approach is class-level mapping between application code and library code: for each app class, the analysis reports (1) whether this class likely originated from one of the libraries in the repository, and (2) the specific library class that is the most likely match for the app class. As discussed earlier, class-level matching provides essential information for subsequent analyses such as app clone/repackaging detection. Furthermore, the reported app-to-library matching is injective: at most one app class is matched with a particular library class. The tool will be referred to as Orlis, since it performs obfuscation-resilient library detection using interprocedural code features and similarity digests.

4.3.1 Code Features and Detection Workflow

Based on the assumptions described earlier, we define a set of code features used for library detection. First, for each method in a library or in an analyzed app, we use a *fuzzy signature* as defined by others [9]. These signatures are used to build a fuzzy call graph for each method, which is then mapped to a *string feature* for the method (Section 4.3.2). The method features for a library are used to create a *library digest*; a similar *app digest* is built from the features of app methods (Section 4.3.3). Method features are also used to compute a *class digest* for each class, based on the methods appearing in the class and in its transitive superclasses. In all cases, similarity digests are used (Section 4.3.4).

Using this approach, for each library from the repository we can pre-compute a library digest as well as class digests for its classes. Given an unknown obfuscated app, its app digest and class digests are computed. Based on this information, the library detection workflow proceeds in two stages, as shown in Figure 4.1. Stage 1 compares the app digest with each library digest. Libraries that are highly unlikely to be included in the app are removed from further consideration. This decision is based on similarity scores derived from the digests. After this stage, only a small number of libraries from the repository are left as candidates. In Stage 2, an injective mapping from app classes to library classes is determined, based on the similarity of class digests. Only classes from the candidate libraries are considered for this mapping.

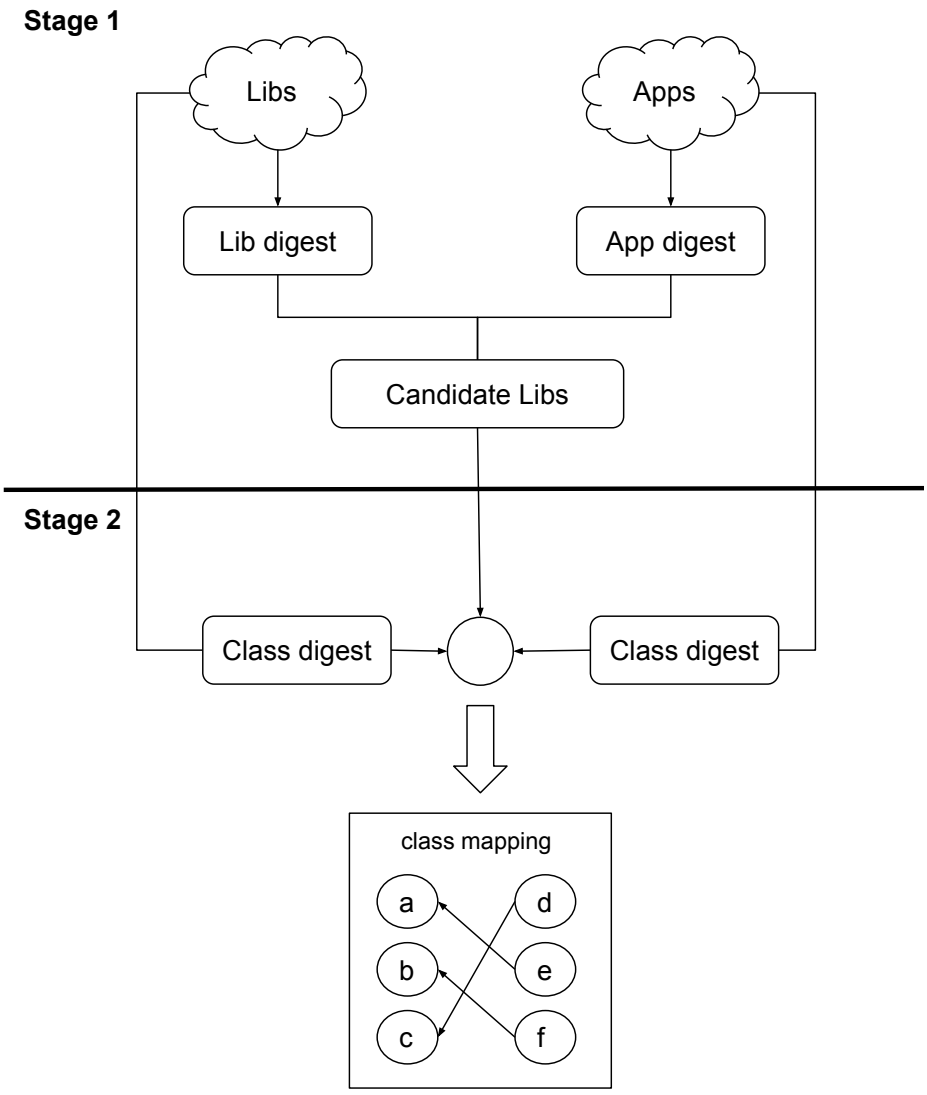


Figure 4.1: Workflow of library detection

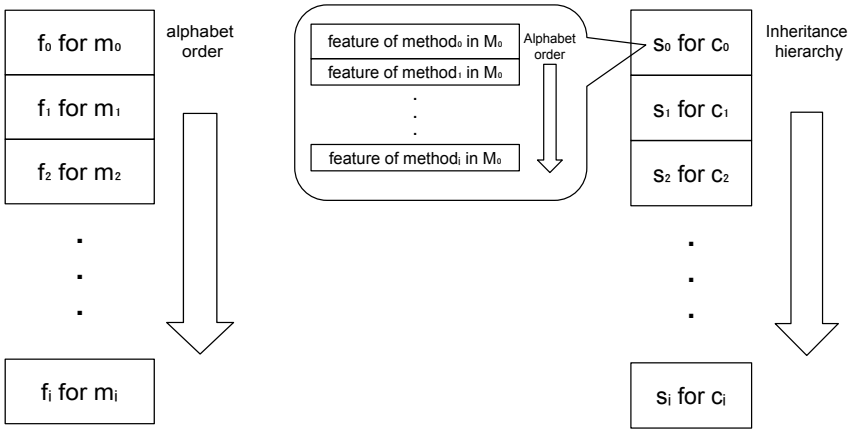
4.3.2 String Features for Methods

Fuzzy signatures are computed as follows. Given a method signature containing a name, parameter types, and return type, a simplified signature is created by (1) removing the method name and (2) replacing all classes defined in the library/app with

a single placeholder name. Names of Android framework classes and Java standard library classes remain unchanged. For example, if we consider a library method with signature `int methodA(android.view.View, ClassA,int,java.lang.String)`, the corresponding fuzzy signature is `int(android.view.View,X,int,java.lang.String)`. Here `View` and `String` are standard library classes that would not be renamed by an obfuscator, while library class `ClassA` is replaced with a placeholder `X` because obfuscation may change its name when the library is included in an app and that app is obfuscated.

Recall the assumption that (transitive) calling relationships between methods would not be affected by obfuscation. If a library method m is included in an obfuscated app, m's call graph in the library (i.e., m, its transitive callees, and the call edges between them) would be a subgraph of m' call graph in the app. Note that we cannot claim that the two call graphs are identical: it is possible that there are extra callees of m in the app, due to (1) app subclasses that override methods defined in library superclasses, and (2) calls to utility methods inserted by the obfuscator. One can try to determine whether m is present in a given obfuscated app by (1) constructing m's call graph in the library, (2) constructing the app's call graph, (3) replacing each method in the graphs with its fuzzy signature, and (4) searching for an isomorphic subgraph. Unfortunately, subgraph isomorphism is a classic NP-complete problem.

Instead, we use a string representation of the fuzzy call graph of each method in a library or an app to define features that are then used to create similarity digests. Specifically, consider a library with a set of methods $\{m_1, m_2, \ldots, m_n\}$. For each method m_i, we compute a feature f_i: a string containing the fuzzy signatures of all



(a) String feature of an app or a library

(b) String feature of a class C_0

Figure 4.2: String features used for matching

methods reachable from m_i in the library call graph, including m_i itself. To make f_i deterministic, the fuzzy signatures are sorted alphabetically and then concatenated to create the final string feature f_i. The same approach is used to define a feature f_j for a method m_j in a given obfuscated app.

4.3.3 Stage 1: App-Library Similarity

As described earlier, the purpose of this stage is to determine the similarity between an app and a library in order to filter out libraries that are unlikely to be included in the app. For each app (or library), a string feature f_{app} (or f_{lib}) is computed based on the method string features described above. The string features for the methods are sorted alphabetically and then concatenated, as illustrated in Figure 4.2a. A digest is computed from this feature. The pre-computed digests in the library repository are compared for similarity with the digest of the given obfuscated

app. Experimental results presented in Section 4.6 show that this approach is very effective in filtering out a large number of libraries.

4.3.4 Stage 2: Class Similarity

Stage 2 determines a mapping from app classes to library classes. Suppose an app has a set of classes $S_{app} = \{ac_1, ac_2, \ldots, ac_n\}$ and a set of candidate libraries $\{lib_1, lib_2, \ldots, lib_k\}$ as determined by Stage 1. Let $S_{lib} = \{lc_1, lc_2 \ldots, lc_m\}$ be the union of the set of classes in lib_i over all i.

For each class in S_{app} or S_{lib}, a string feature is computed as illustrated in Figure 4.2b. Consider a class C_0 and the chain of its superclasses $C_1, C_2, \ldots, C_k$, accounting only for classes that are defined in the app (or the library)—that is, excluding classes such as `java.lang.Object`. As discussed earlier, we assume that if C_0 is included in the app, so are all C_i in this list. Let M_i be the set of methods defined in C_i. We consider the string features of all methods in M_i, sort them alphabetically, and concatenate them to obtain a string s_i $(0 \leq i \leq k)$. Then the concatenation of $s_0, s_1, \ldots, s_k$ (in that order) defines the string feature for C_0. Finally, a digest of this string is computed.

A pair-wise similarity score s_{ij} is computed for classes ac_i and lc_j by comparing their class digests. Then all pairs $\langle ac_i, lc_j \rangle$ are sorted based on s_{ij} and the sorted list is processed using Algorithm 2. The resulting class-level mapping is clearly injective.

4.4 Orcis: An Obfuscation-Resilient Approach for Clone Detection

Using similar ideas, we also developed the Orcis tool, which performs obfuscation-resilient library detection using interprocedural code features and similarity digests.

Algorithm 2: Class-level mapping

```
Input: SortedPairs: list of ⟨ac_i, lc_j⟩ sorted by s_ij
Result ← {}
AppClasses ← {}
LibClasses ← {}
foreach ⟨ac_i, lc_j⟩ ∈ SortedPairs in order do
    if s_ij < threshold then
        break
    if ac_i ∈ AppClasses then
        continue
    if lc_j ∈ LibClasses then
        continue
    AppClasses ← AppClasses ∪ {ac_i}
    LibClasses ← LibClasses ∪ {lc_j}
    Result ← Result ∪ {⟨ac_i, lc_j⟩}
```

The library detection of ORLIS is used as a pre-process to filter out third-party library classes. After this filtering, one naive approach is to compute the similarity of all pairs of application classes between two candidate apps. Clearly, this would be very expensive. Instead, we designed a feature string for each app and used the similarity digest to accelerate the process as well as increase the resilience to obfuscation transformations.

4.4.1 Code Features and Detection Workflow

The overall workflow of ORCIS is shown in Figure 4.3. Library detection is used to filter out all library classes. This can be done by using ORLIS or some other tools.

We use code features similar to the ones used for library detection. For each (non-library) class in the app, a string feature is computed as described in Section 4.3.4. After that, the feature of entire app is generated based on the class features. Note that this differs from the computation of the app digest in ORLIS, in which library methods are included. All feature strings of classes are sorted and concatenated into

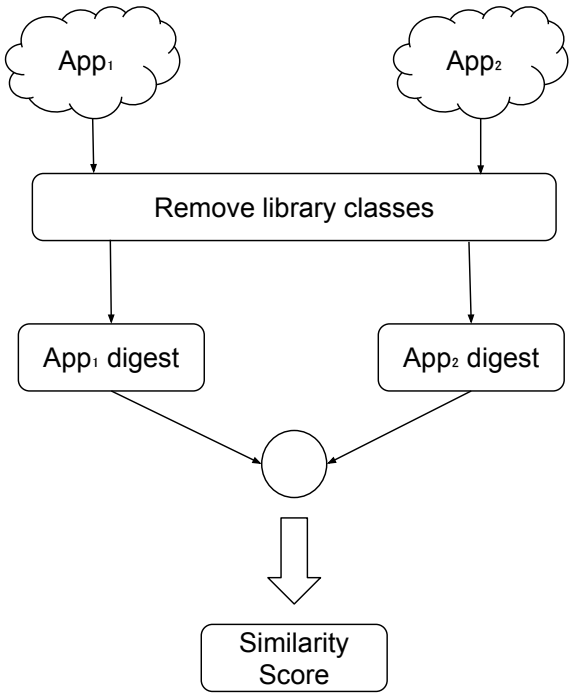


Figure 4.3: Workflow of clone detection

one single string feature. Figure 4.4 illustrates this construction. To tolerate code removal and addition, a similarity score will be calculated and a threshold can be used to determine whether a clone is reported. To improve efficiency, we do not compare the string features. Rather, as with library detection, we compute and compare similarity digests for the string features.

4.5 Tool Implementation

To evaluate our approaches, we implemented the ORLIS and ORCIS tools. Both tools are available at https://presto-osu.github.io/orlis-orcis together with a description of the experimental subjects used in their evaluation. The library/app

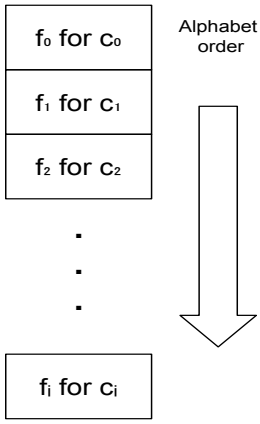


Figure 4.4: String feature of an app

bytecode is analyzed using the Soot analysis framework [107]. The call graph is computed using class hierarchy analysis to resolve polymorphic call sites. For library detection, digest `sdhash` [99] is used in Stage 1, while digest `ssdeep` [108] is used in Stage 2. This choice of digests is discussed in Section 4.6. For clone detection, `sdhash` is used in the matching, which is discussed in Section 4.6.

To determine the set of transitive callees for each method, we first identify the strongly-connected components (SCC) in the call graph. They are used to create the SCC-DAG, in which nodes are SCCs and an edge represents the existence of a call graph edge that connects two SCCs. Reverse topological sort order traversal of the SCC-DAG is then used to compute, for each SCC, the SCCs transitively reachable from it. The method features are derived from this information.

4.6 Evaluation

Our experimental evaluation had two main goals: (1) to study the effects of design choices in the performance of ORLIS and ORCIS, and (2) to compare ORLIS and ORCIS

	ProGuard	DashO	Allatori
# of apps	203	215	241

Table 4.2: Apps in *FDroidData*

with the state-of-the-art LibDetect tool and CodeMatch tools. Evaluation considered precision, recall, and analysis running time. All experiments were performed on a machine with an Intel Core i7-4770 3.40GHz CPU and 16GB RAM. The reported running time is the sum of user CPU time and system CPU time.

4.6.1 Evaluation of Orlis

Two different data sets were used in our evaluation of Orlis. The first one, denoted by *FDroidData*, is obtained from the F-Droid repository [37]. The second data set, denoted by *LibDetectData*, is based on the data used in the evaluation of the LibDetect tool [42]; details of this data set and its uses are presented in Section 4.6.1.

Evaluation with Open-Source Apps

The apps in *FDroidData* are open-source apps. To collect *FDroidData*, we gathered apps with available source code from a variety of app categories and tried to build them with the Gradle tool; 282 apps were built and used to construct our data set. A total of 453 unique library jars were included in this set of apps. These jars formed the library repository for this data set. The apps were then obfuscated by us. The app source code is necessary because (1) the obfuscators are applied to Java bytecode constructed from the source code, and (2) we can determine which third-party library jar files are included in the build process.

We attempted to obfuscate each app using the tools described in Section 4.1: ProGuard, DashO, and Allatori. The number of resulting apps is shown in Table 4.2. The numbers differ across obfuscators because sometimes an obfuscator may fail to obfuscate an app. The obfuscators were executed in a configuration that includes the third-party libraries in the scope of obfuscation, and performs package flattening and renaming. As discussed in Section 4.1, such obfuscation is not handled by many library detection tools. LibDetect [42], which according to its evaluation represents the most advanced current approach in terms of precision and recall, is designed to handle this issue. We used *FDroidData* (which contains the total of 659 apps from Table 4.2) to compare the performance of Orlis and LibDetect. In addition, this data set was used to study the effects of various similarity digests and thresholds in the design of Orlis.

	recall	precision	F1
LibDetect	0.10	0.63	0.17
Orlis	0.63	0.71	0.67

Table 4.3: Comparison with LibDetect on *FDroidData*

Comparison with LibDetect

For each obfuscated app in this set, we established the ground-truth mapping from app-included library classes to the library classes in the library repository, using the logs provided by the obfuscation tools. We then considered the sets of pairs (app class,library class) reported by Orlis. For each app, the precision for this app was computed as the ratio of the number of reported true pairs to the total number

of reported pairs. Recall for an app was computed as the ratio of the number of reported true pairs to the total number of true pairs. The F1 score for each app was also computed (2 times the product of recall and precision, divided by the sum of recall and precision). Similar metrics were computed for LibDetect. That tool outputs information at the package level (i.e., pairs of packages). We modified the tool to print the set of class-level pairs used to compute the package-level pairs, and used those class-level pairs to compute recall and precision.

The results of this experiment are presented in Table 4.3. Each cell in the table shows the average value across all apps in the data set. In these experiments ORLIS was configured with `sdhash` for Stage 1 and `ssdeep` for Stage 2, as described below. We examined manually the output of LibDetect to determine the reasons for the reported low recall. We observed that the tool computes a relatively small number of matching class pairs; a possible explanation is that the approach does not consider methods whose size is below a certain threshold. When mapped to package-level information, the recall increases significantly. This can be observed in the results presented in Section 4.6.1, which describes another comparison with LibDetect.

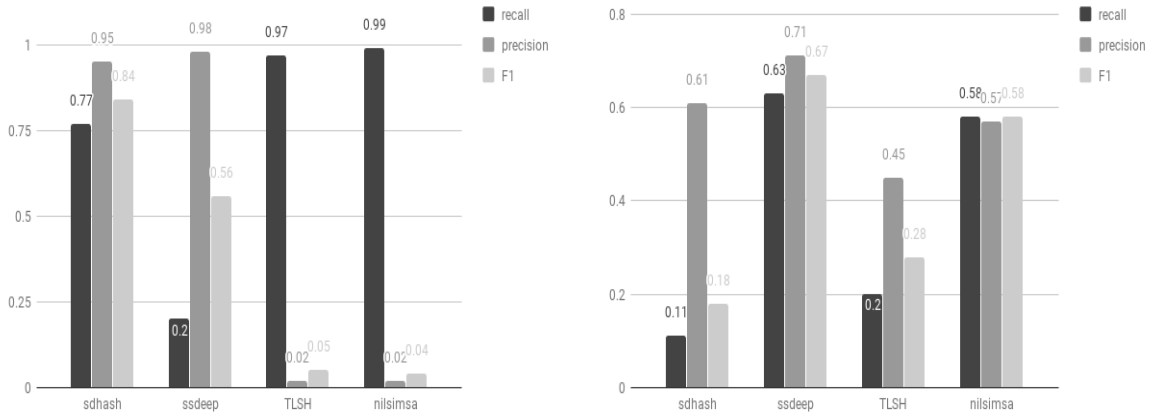
Selection of Similarity Digests

At present, there are four popular public implementations of similarity digests that could be used for our purposes: `sdhash` [91], `ssdeep` [64] , `TLSH` [81], and `nilsimsa` [28]. All of them are able to (1) generate a digest for a given byte array, and (2) compute a similarity score between two digests to indicate the similarity of the represented byte data. Their similarity score metrics are shown in Table 4.4. The scores for `sdhash`, `ssdeep` and `nilsimsa` are in a fixed range. For data without any similarity, the similarity score is 0. `TLSH` uses a similarity score of 0 to indicate

that the compared digests are identical. As the digest difference increases, so does the similarity score, without any pre-defined bound.

Similarity digest	`sdhash`	`ssdeep`	`TLSH`	`nilsimsa`
Score range	0–100	0–100	0–∞	0–128
No similarity	0	0	N/A	0
Identical	100	100	0	128

Table 4.4: Similarity score metrics



(a) Precision, recall, and F1 score for stage I

(b) Precision, recall and F1 score for stage II

Figure 4.5: Performance of similarity digests for *FDroidData*

Figure 4.5a shows how Stage 1 filtering performs when using these digests. Recall that this stage compares an app digest with each library digest in the repository, in order to determine which libraries are likely to have been included in the app. Conservatively, the tool reports a library as a possible candidate if there is any similarity

between the digests. For `sdhash`, `ssdeep`, and `nilsimsa` this means a similarity score greater than zero. `TLSH` does not have a pre-defined value to represent no similarity; we used 300 because this is the value used in the tool's own evaluation [80]. Given the reported candidate library jars, we can compute precision, recall, and F1 score for each app. The averages of these measurements over the apps in the data set are presented in Figure 4.5a. When gathering and analyzing these measurements, we found that the results were skewed if the repository contained several versions of the same library, and each was treated as a separate library for computing these metrics. Thus, for Figure 4.5a, we computed the metrics by treating different versions of the same library as being the same entity (in the numerator and denominator of recall and precision). This produced metrics that faithfully represent the relevant properties of the similarity digests.

Among the four choices, `sdhash` exhibits the highest F1 value and the best trade-off between recall and precision. Thus, we used this choices for the remaining experiments described in this section. The average number of candidate library jars per app is around 22, which is only 5% of the total number of libraries in the repository. The results are similar for the *LibDetectData* dataset described later: the average number of candidate library jars per app is also around 22, for a repository containing 7519 library jars. This is a promising indication that the number of candidate libraries produced by Stage 1 is independent of the total number of libraries. Note that depending on the desired trade-offs of the tool, another choice of a similarity digest could be made. For example, recall can be increased with `TLSH` and `nilsimsa`, at the expense of precision; as a result, fewer libraries would be filtered and the cost of subsequent Stage 2 matching would increase.

Figure 4.5b shows the performance of Stage 2. This stage compares app classes with classes in the candidate library jars. Recall from Algorithm 2 that class pairs are processed in sorted order of their similarity scores: decreasing order for `sdhash`, `ssdeep`, and `nilsimsa` and increasing order for `TLSH`. In addition, a pair is considered only if its similarity score exceeds a pre-defined threshold (or is below that threshold, for `TLSH`). The default thresholds are the same as the ones for Stage 1: 0 for `sdhash`, `ssdeep`, and `nilsimsa` and 300 for `TLSH`. The effect of different threshold values is studied further in Section 4.6.1. For each app in *FDroidData*, we computed precision and recall as described earlier. The results for `ssdeep` in the figure match those in Table 4.3. As the measurements show, `ssdeep` has the best recall, precision, and F1 score.

It is worth noting that `sdhash` and `TLSH` require certain amount of data: in order to compute a digest, the data size should exceed 512 bytes and 256 bytes, respectively. In *FDroidData*, only about 40% of library class features exceed the 512 byte threshold and about 51% exceed the 256 byte threshold. Classes for which this threshold is not met do not have digests and do not appear in the reported mapping. This causes the low recall for `sdhash` and `TLSH`. We further examined various missing and mis-mapped classes. Most of them have very simple call graphs; some even have no callees at all. As a result, their string features are very short and likely to be similar. This leads to a high chance of digest similarity (or no digest at all) and makes them indistinguishable.

We also measured the running time of the four digest functions. There are two components to this cost. The first one is digest building, which is the time to calculate the digest value based on the given string feature. The second component is digest comparison, which computes a similarity score based on the digest values.

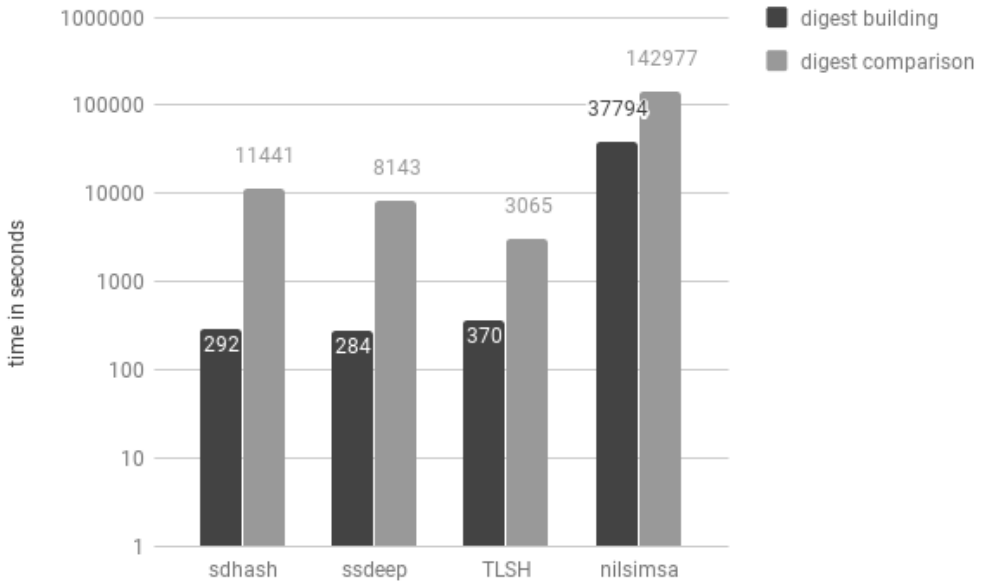
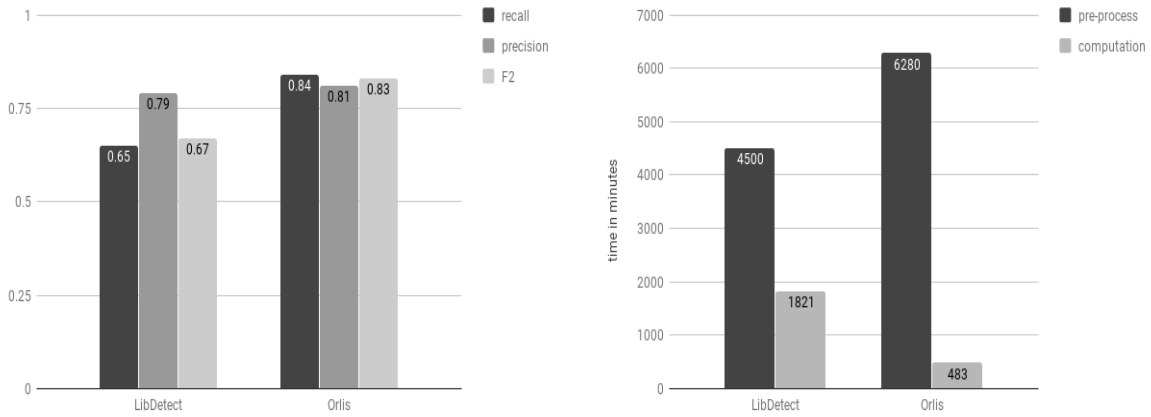


Figure 4.6: Running time of *FDroidData*

Figure 4.6 shows measurements for both components. "Digest building" represents the time to calculate digests for the 659 apps and 453 libraries in *FDroidData*, while "digest comparison" is the time to compute the similarity scores between apps and libraries in Stage 1 and between app classes and library classes in Stage 2. The results show that `nilsimsa` is significantly more expensive than the remaining digests. For digest building, the other three choices do not show substantial differences: all three complete in around 5–6 minutes. For digest comparison there are more significant differences in running time. Although, slower than `TLSH`, `sdhash` and `ssdeep` can complete the digest comparison in less than 200 minutes. Taking into account the recall, precision, and cost of each digest function, we decided to use `sdhash` in Stage 1 and `ssdeep` in Stage 2 of Orlis.



(a) Recall, precision and F2

(b) Running time

Figure 4.7: Comparison with LibDetect

Evaluation with Closed-Source Apps

The second dataset we used, denoted by *LibDetectData*, is based on the data set used for the evaluation of the LibDetect tool [42]. As mentioned earlier, this tool presents the state of art in Android library detection. In order to compare the performance of Orlis with LibDetect, we used data made publicly available by the authors of this work.[2] We tried to collect the same data used in the evaluation of LibDetect and to use the exact same metrics. In this prior work, 1000 apps were collected from five different app stores. Then, the ground truth for those apps was constructed manually. Since the authors of LibDetect were unable to distribute these apps, we searched for and downloaded the apps with the same names. To ensure that the app is the same as the one used in the evaluation of LibDetect, we matched all app packages with ones in the ground truth from LibDetect's web site [21]. If any

[2]We sincerely thank the authors of LibDetect for their valuable help with providing information about their experiments and benchmarks.

package name did not match, we considered this as evidence that we may have a different version of the app, and removed that app from the data set. As a result, we were able to obtain 712 apps that matched ones used in this prior work. To gather the libraries used in the repository, we attempted to download all libraries used in LibDetect's experiments, based on the library URL from the same web site. We successfully obtained 7519 libraries out of the 8000 listed at the web site. Some libraries are missing because the corresponding URLs are no longer available.

The evaluation of LibDetect uses F2 scores, and we also computed the F2 scores for our evaluation.

The metrics presented in the evaluation of LibDetect are at the package level: the ground truth is a set of library packages for each app. The class-level pairs computed by the analysis are used to construct the corresponding package sets; we did the same for the output of Orlis. Recall and precision are computed based on these package sets. Following the metrics used in this prior work, both precision and recall were computed by using the average number of true positives, false positives, and false negatives across all apps. For the i-th analyzed app, let the number of true positives, false positives, and false negatives be tp_i, fp_i, and fn_i respectively. The average number of true positives $\overline{tp}$ is the average of all tp_i. The average number of false positives $\overline{fp}$ and the average number of false negatives $\overline{fn}$ are defined similarly. The overall precision, recall, and F2 score are computed as follows:

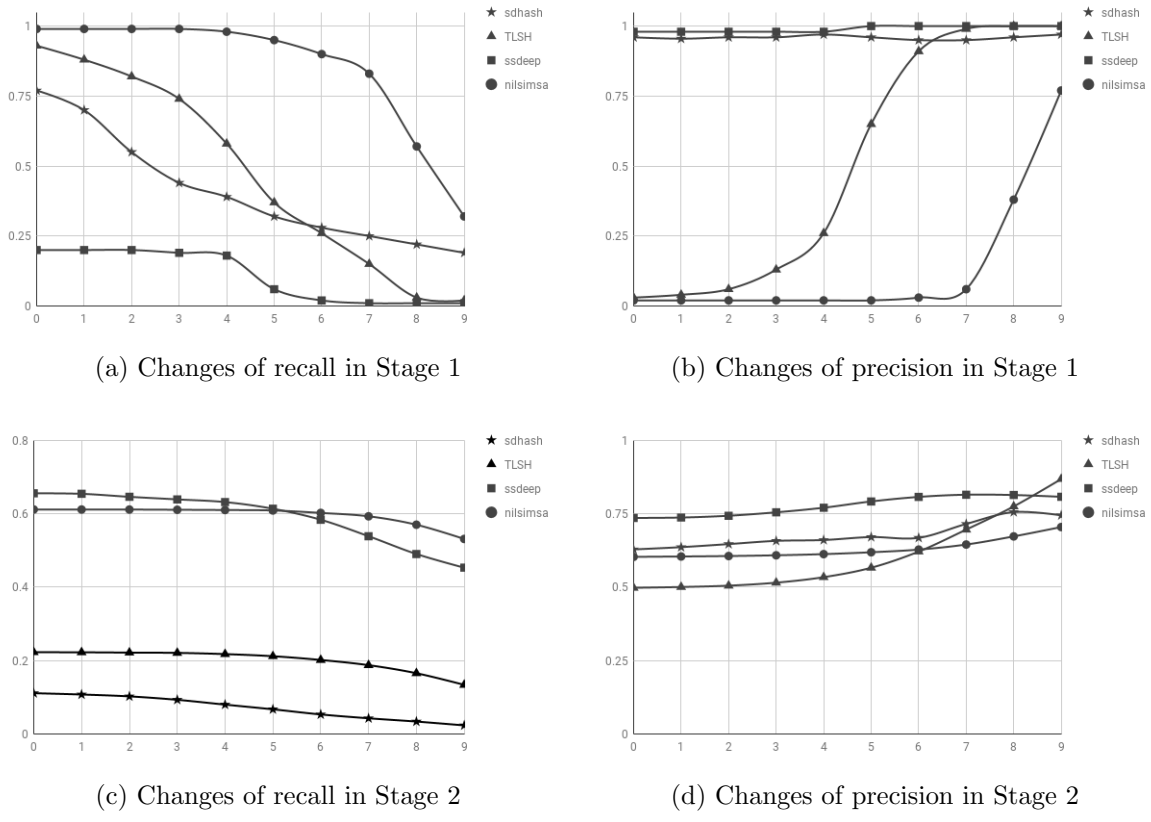
$$
\begin{aligned}
recall &= \frac{\overline{tp}}{\overline{tp} + \overline{fn}} \\
precision &= \frac{\overline{tp}}{\overline{tp} + \overline{fp}} \\
F2 &= \frac{5 \times precision \times recall}{4 \times precision + recall}
\end{aligned}
$$

We computed the same metrics in our evaluation. Since the ground truth provided at LibDetect's web site does not contain class-level mappings, we were unable to compute precision and recall at the class level (unlike for *FDroidData*, where we know the class-level ground truth). The results from this experiment are shown in Figure 4.7a. Overall, ORLIS exhibits higher precision and recall. This indicates that using interprocedural features such as call graphs and class hierarchies is a viable approach for library detection.

We also measured the running time for both tools. LibDetect needs to parse each known library and store the necessary information in a database. Similarly, ORLIS processes each known library, builds call graphs, derives string features and digests, and stores them in its repository. Repository building is a one-time cost; it is also highly parallelizable. Typically, the number of known libraries is smaller than the number of apps that would be analyzed against the repository. Figure 4.7b shows the time for both repository building ("pre-process") and app analysis ("computation"). ORLIS takes more time to construct its repository, but the cost of app analysis is lower. Besides the time cost, both LibDetect and ORLIS have to store the library information to disk. For the libraries in the *LibDetectData* data set, LibDetect uses 8812 MB space in a MySQL database, while ORLIS uses 460 MB of files on disk. These measurements do not include the space to store the library jars.

Effect of Thresholds

In our approach, it is possible to set a cut-off threshold in Stage 1 or Stage 2. Pairs whose similarity score does not exceed the threshold (or, for `TLSH`, is not smaller than the threshold) are considered non-matching by default. The experiments described so far use the threshold of 300 for `TLSH` and 0 for the remaining digest



(a) Changes of recall in Stage 1

(b) Changes of precision in Stage 1

(c) Changes of recall in Stage 2

(d) Changes of precision in Stage 2

Figure 4.8: Impact of thresholds on *FDroidData*

functions. It is interesting to see how different threshold values affect both stages. We selected 10 thresholds, equally spaced, from the "least restrictive" end to the "most restrictive" end of the similarity score range. Figure 4.8 shows the changes of the recall and precision for Stage 1 and Stage 2 using *FDroidData*. As expected, recall in Figure 4.8a and Figure 4.8c decreases while the precision in Figure 4.8b and Figure 4.8d increases when the threshold becomes more restrictive. Depending on the use case, the desired trade-off may be selected by choosing the appropriate threshold. For example, when using library detection as a pre-processing step in

clone/repackage detection (to remove library classes from further consideration), a relatively loose threshold is more appropriate in order to detect as many as library classes as possible.

4.6.2 Evaluation of Orcis

To evaluate Orcis, we found three public available datasets for clone detection. The first one, denoted by *PiggybackedData*, is from the work of Li et al. [85] on detection of piggybacked apps (piggybacking is a type of repackaging). The second one, denoted by *AndroZooData*, is from a large Android apps dataset created by Allix et al. [2]. The third one, denoted by *CodeMatchData*, is the dataset used in the evaluation of CodeMatch [22]. The number of apps we successfully downloaded for each dataset is shown in Table 4.5. The ground truth is provided in the website of each dataset. For *PiggybackedData* and *AndroZooData*, the number of app pairs that are not clones of each other is too large. Therefore, we randomly sampled a subset of app pairs. The number of clone pairs and not-clone pairs used in our experiment is shown in Table 4.6. When examining the ground truth of *CodeMatchData*, we found some errors: 41 pairs of not-clone pairs were actually clones, because there was a one-to-one mapping between their classes. To establish the existence of such a mapping, we decompiled the apps using Jadx [55], a decompiler that transforms dex code to Java source code. After decompilation, the Java source code for both apps has the same textual format. Then we tried to match the source code of all application classes by checking whether (1) the two apps have the same number of application classes and (2) there is an injective mapping between those classes, where matched classes have the exact same source code text. As a result of this comparison,

	PiggybackedData	AndroZooData	CodeMatchData
# of apps	2754	18010	1589

Table 4.5: Number of APKs in each dataset

	# of clone pairs	# of not-clone pairs	total
PiggybackedData	1497	1000	2497
AndroZooData	15297	10000	25297
CodeMatchData	361	546	907

Table 4.6: Number of clone and not-clone pairs in each dataset

we found 41 app pairs that were labeled as not clones in the ground truth used in the evaluation of *CodeMatchData*, but in reality were clones. In our experiments, we treated those pairs as clone pairs, and we adjusted the ground truth accordingly for all measurements. Note that this comparison approach was used to study a small number of mislabeled app pairs, and is too expensive to apply at a large scale.

Evaluation of Similarity Digests

We used the same similarity digests as in Section 4.6.1 and measured their performance on the three datasets. As for library detection before, the threshold was chosen to be as low as possible: the worst-possible similarity score was treated as "no match", and any other score was "match". The performance was evaluated for precision, recall, and F1 score. Precision is the ratio of the number of reported true clone pairs and the total number of reported clone pairs. Recall is computed as the ratio of the number of reported true clone pairs and the total number of true clone pairs. F1 score is calculated as described in Section 4.6.1.

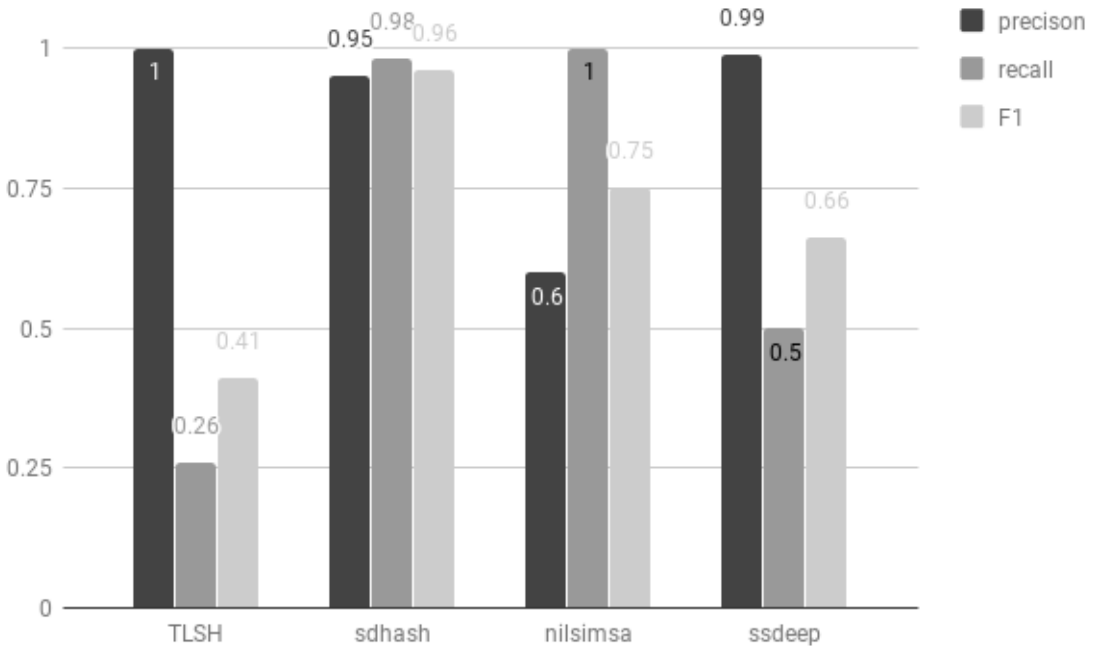


Figure 4.9: Performance of similarity digests

Figure 4.9 shows these measurements. In general, `sdhash` has the best performance for Orcis. Although the precision is slightly lower than that of `TLSH` and `nilsimsa`, the recall and F1 are much better compared with other methods.

We also measured the running time of the analysis for each of the four digest functions. Figure 4.10 shows these results. The time measurements shown in the figure include digest building and digest comparison for the entire dataset. The results show that `nilsimsa` is significantly more expensive than the remaining digests. The other three choices do not show substantial differences. Taking into account of the performance and cost of each digest function, we decided to use `sdhash` as the similarity digest in Orcis. As mentioned earlier, `sdhash` can only work on data that

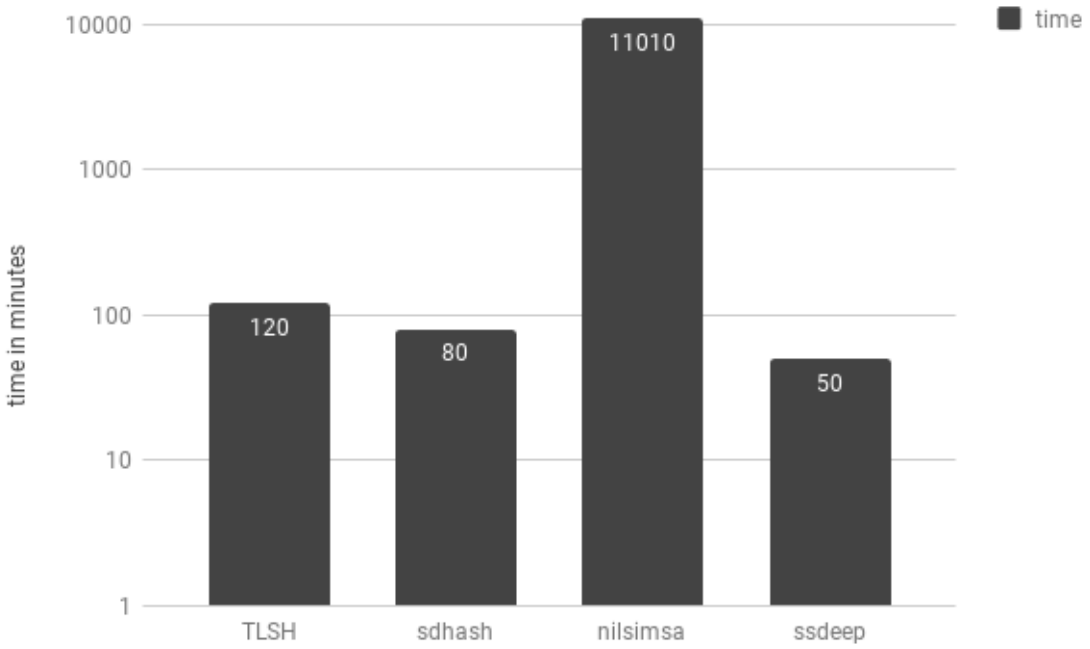
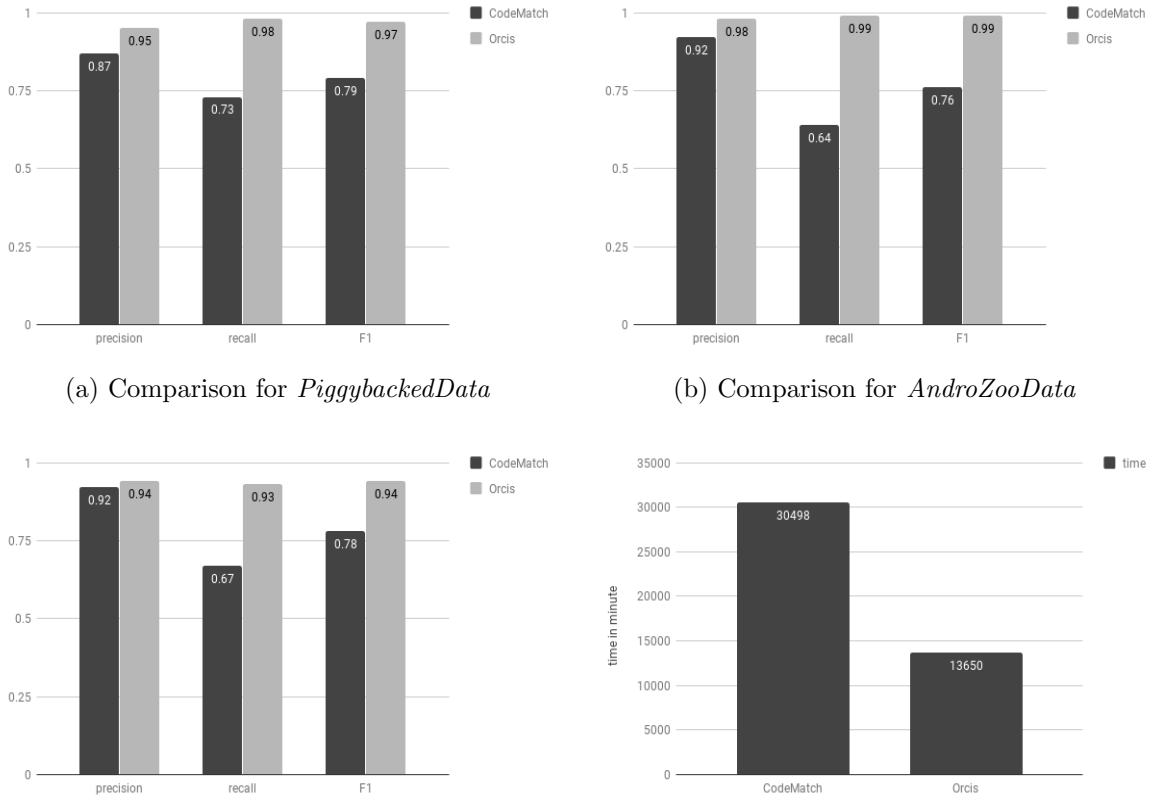


Figure 4.10: Running time

is larger than 512 bytes. In the clone detection scenario, the feature of each app satisfies this requirement.

Comparison with CodeMatch

We compared ORCIS with the state-of-the-art CodeMatch clone detection tool. The results of this comparison are shown in Figure 4.11. For all three datasets, ORCIS has better precision, recall and F1 score. The total running time for these three datasets is shown in Figure 4.11d. Overall, ORCIS takes less than half of the time taken by CodeMatch.



(a) Comparison for *PiggybackedData*

(b) Comparison for *AndroZooData*

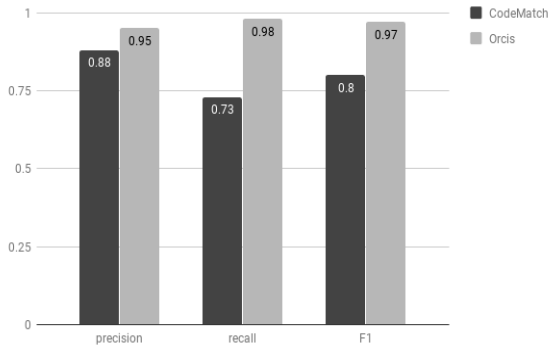
(c) Comparison for *CodeMatchData*

(d) Comparison of running times

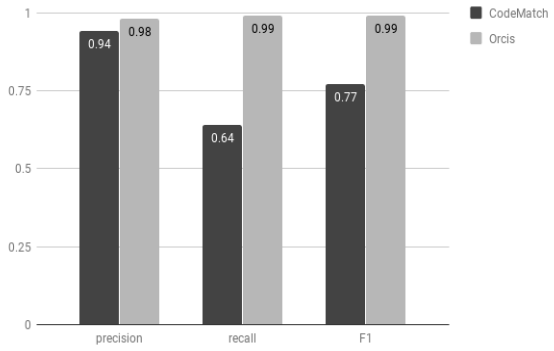
Figure 4.11: Comparison with CodeMatch

Effects Of Library Detection

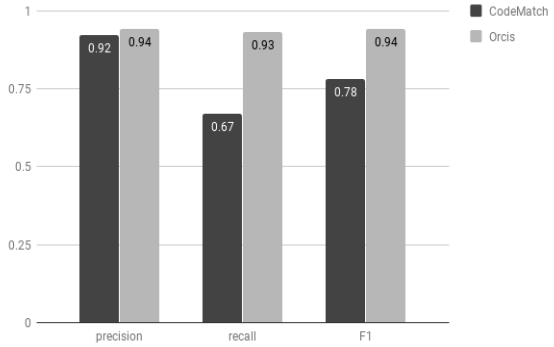
Both CodeMatch and Orcis have a library detection stage to filter out library classes. By default, Orcis uses Orlis, while CodeMatch uses LibDetect. The better performance of Orcis may be caused by the library detection stage. To further investigate the impact of library detection, we replaced the library detection of CodeMatch with Orlis. The results are shown in Figure 4.12. After using Orlis, the



(a) Comparison for *PiggybackedData*



(b) Comparison for *AndroZooData*



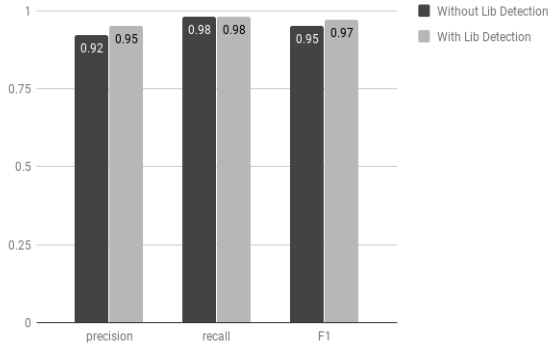
(c) Comparison for *CodeMatchData*

Figure 4.12: Comparison with CodeMatch using ORLIS

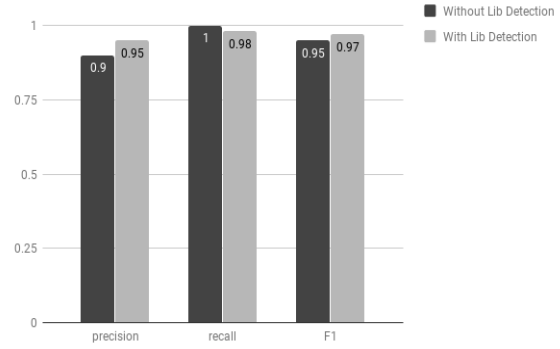
performance of CodeMatch becomes slightly better. However, ORCIS still shows better precision, recall and F1 score. As a result, we can conclude that the improved performance is not caused only by the difference in library detection techniques.

The Need for Library Detection

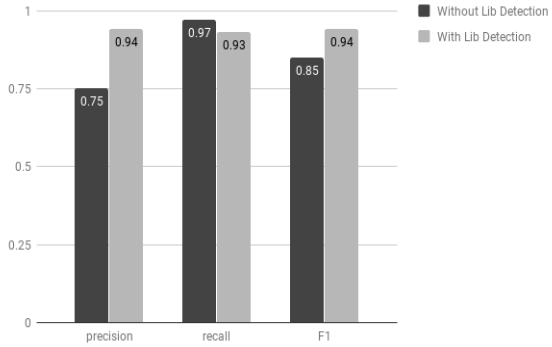
ORCIS performs library detection before the clone detection analysis. To examine the necessity for this pre-processing, we compared the measurements of tool output with and without the library detection stage. Figure 4.13 shows the results of this



(a) Comparison for *PiggybackedData*



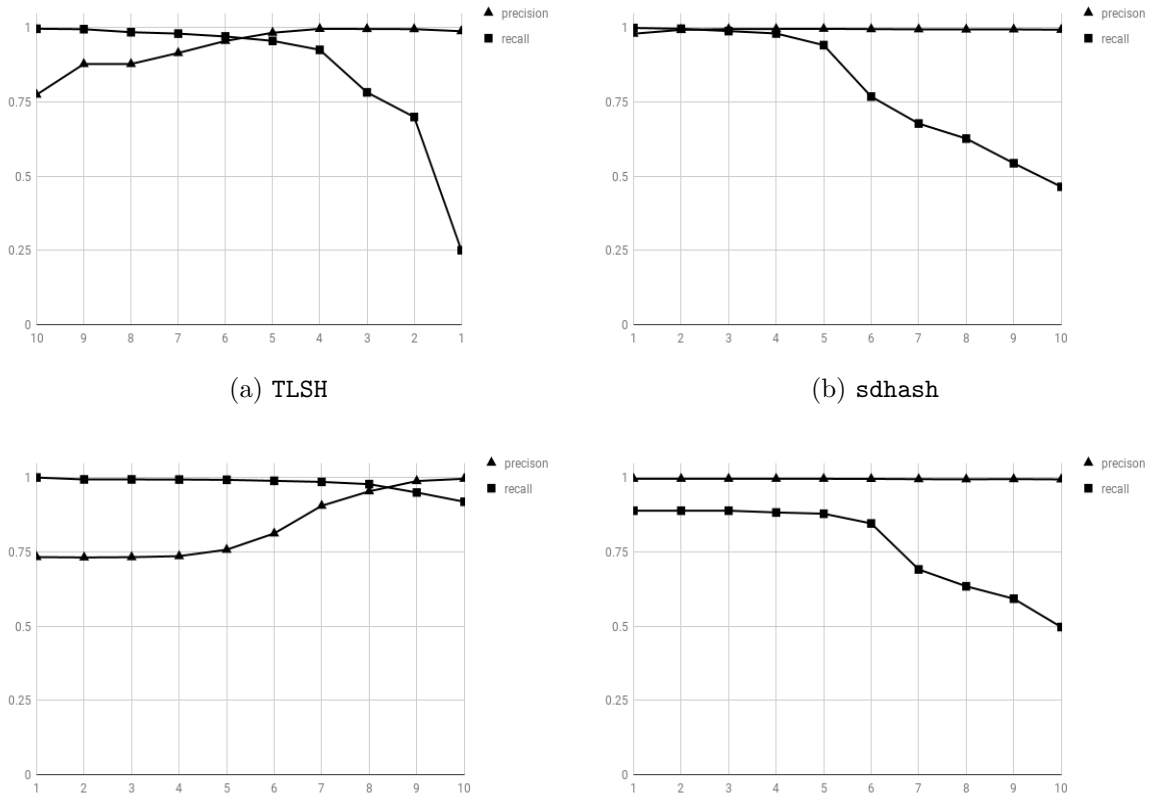
(b) Comparison for *AndroZooData*



(c) Comparison for *CodeMatchData*

Figure 4.13: Measurements with and without a library detection stage

comparison. With library detection, ORCIS achieves better precision as well as F1 score. The recall is slightly worse for *PiggybackedData* and *AndroZooData*. In general, the performance with library detection is better. In general, if library classes are included in the scope of clone detection, this will increase the similarity between apps, especially when the library code is a large portion of the overall app code. As a result, precision could decrease.



(a) `TLSH` (b) `sdhash`

(c) `nilsimsa` (d) `ssdeep`

Figure 4.14: Impact of thresholds for *AndroZooData*

Effect of Thresholds

In our evaluation, we choose the smallest possible similarity as a threshold to report a match. Pairs whose similarity score does not exceed the threshold (or, for `TLSH`, is not smaller than the threshold) are treated as not being clones. Of course, the threshold selection will affect the performance of ORCIS. To investigate these effects, we selected 10 thresholds, equally spaced, from the "least restrictive" end to the "most restrictive" end of the similarity score range. Figure 4.8 shows the changes of the recall and precision for dataset *AndroZooData*; this dataset was used because

it contains the largest number of apps. The general trend is the same for the other two datasets. As expected, the precision increases as the threshold becomes stricter, while the recall decreases.

4.7 Summary

In this chapter, we propose to use call graphs and class hierarchies as the basis for library matching and clone detection for obfuscated Android apps, together with similarity digests for efficiency. Our experimental results indicate that the proposed Orlis and Orcis tools advance the state of the art in obfuscation-resilient library detection and clone detection for Android.

CHAPTER 5: Obfuscation-Resilient Detection of Methods

The approach described in the previous chapter can be used to detect a match with an entire library or app. However, for many static analyses, researchers are more interested in identifying instances of certain methods of interest. For example, if some library methods have security vulnerabilities, these specific methods have to be identified in an obfuscated app. As another example, in order to build static models for Android and Android Wear apps, it is often necessary to consider specific methods from certain libraries (e.g., Google libraries related to app analytics or wearable devices) and to identify these methods and their callers in a given app. In these cases, the naive approach would be to detect a certain method of interest via signature matching. However, as discussed in previous chapters, obfuscation will significantly change method signatures. To address this problem, we propose an obfuscation-resilient approach for method detection.

5.1 Motivation and Challenges

Method detection appears to be similar to library detection at a finer granularity. In the previous chapter we proposed an approach for library detection under obfuscation. One natural question is whether this approach works for method detection. Unfortunately, the answer is negative. The reason is that, in general, a method contains much less information than a library or an app. The similarity digests used in the approach from Chapter 4 cannot work well for such a small amount of data.

Furthermore, the number of methods is far larger than the number of classes; thus, comparing each pair of methods could be rather expensive. One solution is to only consider the pairs with the same fuzzy signature (as defined in Chapter 4). Even though this approach would be helpful in reducing the number of candidate pairs, it still leaves a large number of candidates. Later in this chapter we present experimental evidence that the approach from the previous chapter is not effective for detection of individual methods. This motivates our design of a new approach for obfuscation-resilient method detection.

Even though Chapter 4 summarized commonly-used obfuscation techniques for Android, it is helpful to further highlight the ones that will impact method detection. First, renaming will certainly influence such detection—for example, changing the names of methods and classes will affect signature matching. The second relevant technique is package obfuscation. Some researchers have proposed detection based on package relationships. However, package hierarchy modifications and package flattening will disable such approaches. Third, control flow obfuscation will change the control flow of the method body. Approaches that depend on this control flow will be affected. Finally, the obfuscator may insert new callees inside the method's code. One example is the insertion calls to decryption methods before the use of constant strings that have been encrypted during obfuscation. Matching the call graph of a method may be affected by this transformation.

5.2 Obfuscation-Resilient Approach for Method Detection

Similarly to the library detection problem considered in the previous chapter, we assume that a given, previously-unknown, obfuscated app is compared against a

pre-assembled repository of known libraries. The goal is to detect whether certain "methods of interest" from the repository are present in the app. To solve this problem, we propose a new approach using inter-procedural code features and a new type of method fuzzy signature that contains more information than the one used in Chapter 4. The input to the approach is the set of library *.jar* files, the APK of the app, and the set of relevant library methods. The output is the matching from these library methods to their corresponding "copies" in the app.

5.2.1 Invariant Properties

Our approach is based on five properties we assume to be preserved during obfuscation. These properties are refined from the assumptions used in Chapter 4. Specifically, we assume that the obfuscation defines a mapping μ from library classes/methods to app classes/methods with the following properties. First, for a library method m and its corresponding app method $\mu(m)$, either (1) both methods have the same primitive return type, or (2) the return type of m is some library class c and the return type of $\mu(m)$ is the the corresponding app class $\mu(c)$. We assume a similar property for each pair of corresponding formal parameters of m and $\mu(m)$. Further, if m is in class c, then $\mu(m)$ should be in class $\mu(c)$. Mapping μ should also preserve the class hierarchy: if c_1 has a direct superclass or superinterface c_2, the same relationship should hold for $\mu(c_1)$ and $\mu(c_2)$. Finally, μ should preserve a method's callees: if m_1 contains a call site that may invoke m_2 at run time, then $\mu(m_1)$ contains a call site that may invoke $\mu(m_2)$. As a result, the set of transitive callees of $\mu(m)$ in the app is a superset of the set of transitive callees of m in the library.

The properties described above are derived from typical behaviors we have observed in real-world code obfuscators. For primitive types, obfuscation has no effect. For a class type, the name of the class may be changed, but the actual class identity remains the same. For example, suppose the return type of a method is class *com.amazon.aws.Client* and this class is renamed to $x.a$ during obfuscation. Then the return type of the method becomes $x.a$. The class membership invariant means that methods will not be moved between classes. The preservation of the inheritance hierarchy indicates that superclass and superinterface relationships are not modified. Further, if method m_2 is a transitive callee of method m_1 before obfuscation, it will also be a transitive callee after obfuscation. This definition allows for more callees to be added by the obfuscator, as long as all existing callees are preserved.

5.2.2 Detection Workflow and Method Fuzzy Signature

The overall workflow of our approach is shown in Figure 5.1. There are two stages of processing. Consider a library method of interest m. Instead of comparing m with each method in the app directly, in the first stage we perform a quick check to decide whether there is a possibility that m exists in the app. If the answer is positive, the second stage checks m against individual candidate methods in the app. This processing is done separately for each method of interest m.

In both stages, we use a method fuzzy signature to represent a method. The traditional signature has four components: class to which the method belongs, return type, method name, and parameter types. Figure 5.3 shows one example of such a signature. Method *deleteVersion* is a member method of class *AmazonS3Client* with *void* return type and three parameter types. One possible form of this signature after

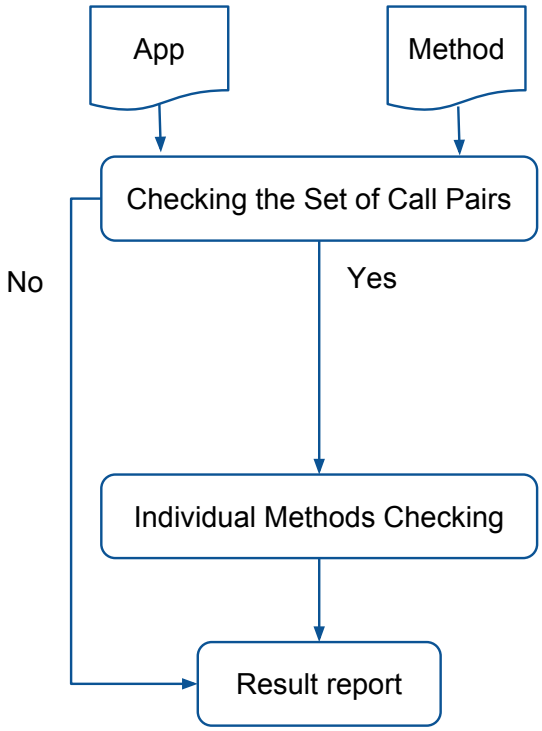


Figure 5.1: Workflow of method detection

obfuscation is shown in Figure 5.4. In Chapter 4, we use a simple fuzzy signature that (1) removes the method name and (2) replaces all classes defined in the library/app with a single placeholder name X. This kind of signature is very weak and could cause hundreds of methods to be mapped to the same fuzzy representation. Based on the invariant properties, we define a new obfuscation-resilient fuzzy signature which contains more information.

First, a fuzzy class type is built for each class except for ones from the Java/Android standard libraries. The fuzzy type includes information about the inheritance

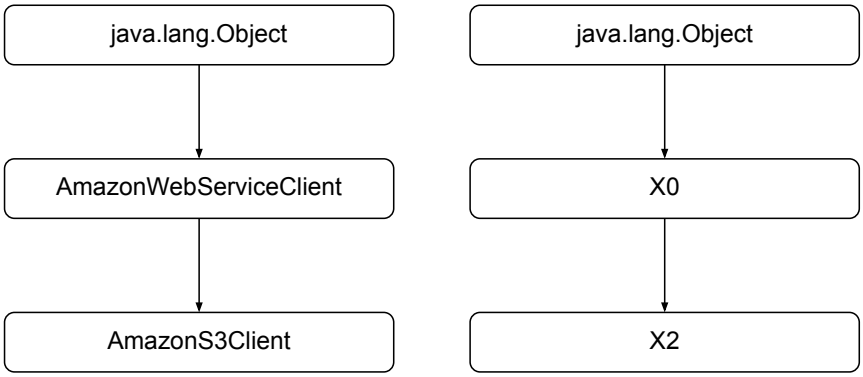


Figure 5.2: Class type fuzzy signature

```
AmazonS3Client: void deleteVersion (java.lang.String, AmazonS3Client, ObjectMetadata)
```

Figure 5.3: Example of a method signature

hierarchy of the class. For example, suppose class *AmazonS3Client* has a superclass chain with *AmazonWebServiceClient* and *java.lang.Object*, as shown in the left side of Figure 5.2. Class *java.lang.Object* will remain as is since it is a Java standard library class. Class *AmazonWebServiceClient* will be mapped to the placeholder name *X0*, where 0 means the number of interfaces this class implements directly. Similarly, *AmazonS3Client* is mapped to placeholder name *X2* because it has two direct superinterfaces (not shown in the example). Then all components in the inheritance chain will be combined to create the fuzzy type *java.lang.Object/X0/X2* for class *AmazonS3Client*. Under the assumptions described earlier, both a library class c and its version $\mu(c)$ in the obfuscated app will have the same fuzzy representation.

```
wy.blg: void a (java.lang.String, wy.blg, wy.bla)
```

Figure 5.4: Example of a method signature after obfuscation

```
java.lang.Object/X0/X2$1: void (java.lang.String, java.lang.Object/X0/X2$1, java.lang.Object/X0/X2$2)
```

Figure 5.5: Example of a method fuzzy signature

A method's fuzzy signature is built using the fuzzy class types. It is possible that two different class types in the same method signature may have the same fuzzy representation. In this case, an additional numeric suffix is added to the fuzzy class type, in the order in which the types appear in the signature. For example, suppose *AmazonS3Client* and *ObjectMetadata* have the same fuzzy class type *java.lang.Object/X0/X2*. Since *AmazonS3Client* occurs before *ObjectMetadata* in the method signature, its fuzzy class type will become *java.lang.Object/X0/X2$1*, while the fuzzy class type of *ObjectMetadata* changes to *java.lang.Object/X0/X2$2*. Figure 5.5 shows the final fuzzy signature for the method in Figure 5.3. This representation is more informative than the one used in Chapter 4, but it is still obfuscation-resilient: a library method m and its counterpart $\mu(m)$ in the obfuscated app will have the same fuzzy signature. In our experience, when using this more precise representation, the average number of methods that have the same fuzzy signature reduces from hundreds to tens or even ones.

5.2.3 Stage 1: Checking the Set of Call Pairs

Given a library method of interest m, the first stage of our approach performs a check to decide whether m is likely to exist in the app. This is done by constructing and comparing two sets of *call pairs*. A call pair is a string of the form (f_{m_1}, f_{m_2}), where method m_1 can invoke method m_2 directly or indirectly, and f_{m_1} and f_{m_2} are the corresponding fuzzy signatures. For the library method m, we can compute the set S_m of all call pairs (f_m, f_{m_2}). For the obfuscated app, let S_{app} be the union of the set of call pair sets of all methods in the app. When constructing either set, if there are multiple instances of the same call pair (due to the imprecise nature of fuzzy signatures), an incremental number is appended to each instance in order to create a unique version of it in the set.

According to our invariant properties, all transitive callees of a method will be preserved after obfuscation. Thus, if a library method m exists in the app, S_m will be a subset of S_{app}. If this property does not hold, there is no need to perform method-level checking of m against individual app methods.

5.2.4 Stage 2: Matching of Individual Methods

If the set of call pairs of method of interest m is present in the app's set of call pairs, the next step of our approach will attempt to identify m in the obfuscated app. All methods that have the same fuzzy signature as m will be considered. If there are several such methods, the best match has to be determined. This is done by comparing S_m with the corresponding set S_{m_i} for any candidate app method m_i. We first check if S_m is a subset of S_{m_i}. If this is the case, a similarity score is computed as $\frac{|S_m|}{|S_{m_i}|}$. The similarity score is between 0 and 1. Larger scores indicate more similarity.

The method m_i with the largest score will be reported. If there are multiple methods with the same highest similarity score, all of them will be reported.

5.2.5 Bloom Filter

If this analysis is performed on a large number of Android apps (e.g., as part of app-market-size studies), efficiency is an important concern. The bottleneck in our approach are the set-related operations. We have to check subset relationships in both stages of the analysis. Normally, a hashset is fast for item traversal. However, to check a subset relationship, each item in the hashset has to be visited. If the size of the set is large, it is still slow. Instead, we adopt Bloom filters, which is a powerful technique to represent sets. A Bloom filter only contains an array of bits and several hash functions. In order to insert an item, the hash values of all hash functions are computed for this item. These values indicate positions in the bit array. Then the corresponding bits in those positions are set if not already set. To determine the existence of an item, all corresponding bits in the positions computed by all hash functions have to be checked. The time complexity of both insertion and checking is $O(1)$. For subset checking of two Bloom filters, we only need to compare the two bit arrays. For example, suppose we have two bit arrays a and b. If every bit set in a is also set in b, then a is considered to be a subset of b. In most cases, bit array checking is much faster than the item traversal in hashset. The size of the bit array depends on performance requirements. A Bloom filter has no false negatives, but may have false positives, both for checking set membership and for checking subset relationships. The false positive rate is determined by the size of the bit array, the number of hash functions, and the maximum number of items in the set. For example,

if the number of hash functions is 10, the maximum number of items in the set is 100000, and the size of the bit array is 170KB, the false positive rate is about 0.001. The false positives are because of the collision of the hash functions: even though an item is not in the set, the corresponding positions in the bit array may still have a chance to be set when inserting other items. The size of the bit array will affect both the time cost and the false positive rate. This trade-off has to be considered when using Bloom filters.

5.2.6 Implementation

To evaluate our approach, we implemented a method detection tool, denoted by ORMIB, which performs obfuscation-resilient method detection using interprocedural features and Bloom filters. The library/app bytecode is analyzed using the Soot analysis framework [107]. The call graph is computed using Class Hierarchy Analysis. To determine the set of transitive callees for each method, we used the same approach as in Chapter 4.

For Bloom filters, we used the implementation in the Guava library from Google. The original library does not provide the API to check subsets. We implemented it using the bit checking approach mentioned above. The hash functions are *Murmur3*, which are the current state-of-the-art hash functions for Bloom filters in terms of speed and false positive rate. Our desirable false positive rate is around 0.001. Therefore, if we know the maximum size of the set, the size of the bit array in the Bloom filter is determined. One way to decide the maximum size of the set is to use the size of the set of call pairs of a method or an app directly. However, this will result in a large number of Bloom filters with different bit array sizes. Unfortunately, Bloom filters

with different bit array sizes are incompatible: they cannot be compared to check the subset relationship. Another choice is to use the maximum size of the sets of call pairs for all methods and apps we have observed. But this is still not desirable because the gap between the minimum and maximum set size is very large. A large app may easily have about $100,000$ call pairs, while for a simple method this number is often less than 5. As a compromise, we design a multi-layer scheme in our implementation. To be specific, we have five intervals for set sizes: $[0, 10]$, $(10, 100]$, $(100, 500]$, $(500, 1000]$, and $(1000, 5000]$. The largest number of each interval will be used as the maximum size of the set. For example, if a method has no greater than 10 call pairs, 10 is used as the maximum size of the set when building the Bloom filter. In our experiment, no method has more than 5000 call pairs. Most of the methods will fall in the first three intervals. The same design is used for apps, but with five different intervals because apps have more call pairs than a single method.

5.3 Evaluation

We examined Ormib on three clients: (1) a static analysis of notifications in Android Wear applications, (2) data leakage detection for Android cloud service APIs, and (3) analysis of Google Analytics API calls in Android apps. The evaluation considered precision, recall, and analysis running time. All experiments were performed on a machine with an Intel Core i7-4770 3.40GHz CPU and 16GB RAM. The reported running time is the sum of user CPU time and system CPU time.

5.3.1 Data Sets

The goal of Ormib is to detect certain methods in a given obfuscated app. For the evaluation, a set of obfuscated Android apps and a list of library methods of

	# of API methods	# of apps
WearData	11	12
CloudData	32	3860
GoogleAnalyticsData	15	2454

Table 5.1: Data Sets

interest are needed. Three different data sets were used in our evaluation. Table 5.1 shows characteristics of these three datasets. The first one, denoted by *WearData*, is obtained from the work of Zhang et al. [128]. In this work, 12 apps are used in the evaluation and 11 API methods from the Android support library are needed to construct a model of Android Wear notifications. All apps in *WearData* are not obfuscated. The second data set is denoted by *CloudData*. The API methods are from the work of Zuo et al. [134] in which security-related defects are analyzed for Android cloud service APIs. There are 32 cloud service APIs from three cloud libraries: Azure, Amazon AWS, and Firebase. The apps in *CloudData* are downloaded from Google Play. We crawled 17, 345 APK files by April 22, 2018. These apps are the union of the top 600 free apps (based on popularity) in all app categories except game and family. Among these apps, we gathered 3860 apps that contain at least one unobfuscated cloud service API method. The third data set, denoted by *GoogleAnalyticsData*, is based on ongoing work by Zhang et al. This work considers 15 API methods from the Google Analytics library and 2454 apps containing unobfuscated calls to these methods. The signatures of all API methods from these datasets are shown in Table 5.2.

All apps we collected contain unobfuscated calls to the relevant API methods. We attempted to obfuscate each app using ProGuard with all available obfuscation

Client	Method Signature
Android Wear API	NotificationCompat$Builder: NotificationCompat$Builder addAction(NotificationCompat$Action) NotificationCompat$Builder: NotificationCompat$Builder addAction(int,CharSequence,PendingIntent) NotificationCompat$WearableExtender: NotificationCompat$WearableExtender addAction(NotificationCompat$Action) NotificationCompat$WearableExtender: NotificationCompat$WearableExtender addActions(List) NotificationCompat$Builder: NotificationCompat$Builder extend(NotificationCompat$Extender) NotificationCompat$Builder: Notification build() NotificationCompat$Action$Builder: NotificationCompat$Action build() NotificationManagerCompat: void notify(int,Notification) NotificationManagerCompat: void notify(String,int,Notification) NotificationCompat$WearableExtender: NotificationCompat$WearableExtender addPages(List) NotificationCompat$WearableExtender: NotificationCompat$WearableExtender addPage(Notification)
Cloud Service APIs	TransferUtility: TransferObserver download(String, String, File) TransferUtility: TransferObserver download(String, String, File, TransferListener) TransferUtility: TransferObserver upload(String, String, File) TransferUtility: TransferObserver upload(String, String, File, ObjectMetadata) TransferUtility: TransferObserver upload(String, String, File, CannedAccessControlList) TransferUtility: TransferObserver upload(String, String, ObjectMetadata, CannedAccessControlList) TransferUtility: TransferObserver upload(String, String, ObjectMetadata, CannedAccessControlList, TransferListener) AmazonS3Client: void deleteObject(String, String) AmazonS3Client: void deleteVersion(String, String, String) AmazonS3Client: boolean doesObjectExist(String, String) AmazonS3Client: String getBucketLocation(String) AmazonS3Client: S3Object getObject(String, String) AmazonS3Client: String getObjectAsString(String, String) AmazonS3Client: ObjectMetadata getObjectMetadata(String, String) AmazonS3Client: String getResourceUrl(String, String) AmazonS3Client: URL getUrl(String, String) AmazonS3Client: ObjectListing listObjects(String) AmazonS3Client: ObjectListing listObjects(String, String) AmazonS3Client: ListObjectsV2Result listObjectsV2(String) AmazonS3Client: ListObjectsV2Result listObjectsV2(String, String) AmazonS3Client: PutObjectResult putObject(String, String, File) AmazonS3Client: PutObjectResult putObject(String, String, InputStream, ObjectMetadata) AmazonS3Client: PutObjectResult putObject(String, String, String) AmazonS3Client: void restoreObject(String, String, int) CognitoCredentialsProvider: void ⟨init⟩(String,String,String,String) BasicAWSCredentials: void ⟨init⟩(String,String) MobileServiceClient: void ⟨init⟩(String,Context) MobileServiceClient: void ⟨init⟩(String,String,Context) NotificationHub: void ⟨init⟩(String,String,Context) CloudStorageAccount: CloudStorageAccount parse(String) FirebaseOptions: void ⟨init⟩(String,String,String,String,String,String,String) FirebaseOptions: void ⟨init⟩(String,String,String,String,String,String)
Google Analytics APIs	HitBuilders$HitBuilder: HitBuilders$HitBuilder setCustomMetric(int,float) HitBuilders$HitBuilder: HitBuilders$HitBuilder setCustomDimension(int,String) HitBuilders$HitBuilder: HitBuilders$HitBuilder addProduct(com.google.android.gms.analytics.ecommerce.Product) HitBuilders$HitBuilder: HitBuilders$HitBuilder addPromotion(Promotion) HitBuilders$HitBuilder: HitBuilders$HitBuilder addImpression(Product,String) HitBuilders$HitBuilder: HitBuilders$HitBuilder setProductAction(ProductAction) HitBuilders$HitBuilder: HitBuilders$HitBuilder setPromotionAction(String) HitBuilders$HitBuilder: HitBuilders$HitBuilder setCampaignParamsFromUrl(String) HitBuilders$ScreenViewBuilder: Map build() Tracker: void enableAutoActivityTracking(boolean) GoogleAnalytics: com.google.android.gms.analytics.Tracker newTracker(int) GoogleAnalytics: com.google.android.gms.analytics.Tracker newTracker(String) Tracker: void send(Map) Tracker: void setScreenName(String) GoogleAnalytics: com.google.android.gms.analytics.GoogleAnalytics getInstance(android.content.Context)

Table 5.2: APIs

techniques including renaming, repackaging, dead code removal, etc. All apps in our datasets were obfuscated successfully. ProGuard generates a mapping file between the method signatures before and after obfuscation. Based on this mapping, we could build the ground truth for the corresponding obfuscated app.

5.3.2 Evaluation on a Static Analysis of Android Wear Notifications

Prior work by others [128] developed a static analysis to model the structure and behavior of notifications in Android Wear applications. Android Wear defines in a set of additional Android APIs that are used for wearable devices such as smartwatches. The static analysis mentioned above identifies call sites that invoke these APIs and models their semantics by constructing a so-called constraint graph. Based on this graph, a static model of Android Wear notification is built. This model contains four categories of nodes: (1) static abstractions that represent the run-time notification objects, (2) the *notify* calls each such object reaches, (3) the *screens* it could trigger on the wearable, and (4) the *actions* that could be triggered by a user on these screens. An automated testing framework is further developed based on the model.

The building of the constraint graph relies entirely on signatures of Android Wear APIs from the Android support library, which can be easily obfuscated by free and commercial tools. If such obfuscation is performed, the constraint graph analysis cannot identify the relevant API methods and cannot proceed further. To help recover the correct model, one can detect the corresponding API methods after obfuscation (using our Ormib tool) and then replace the original API method signatures with the corresponding obfuscated ones.

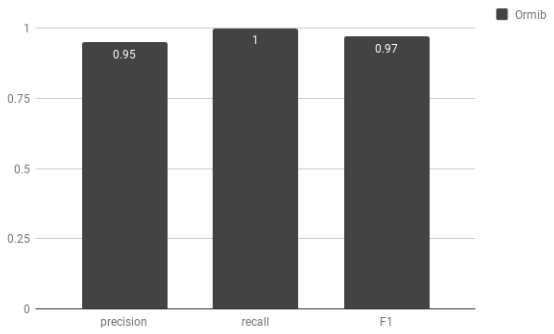
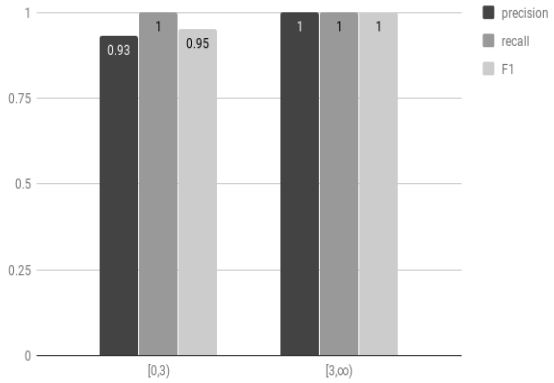
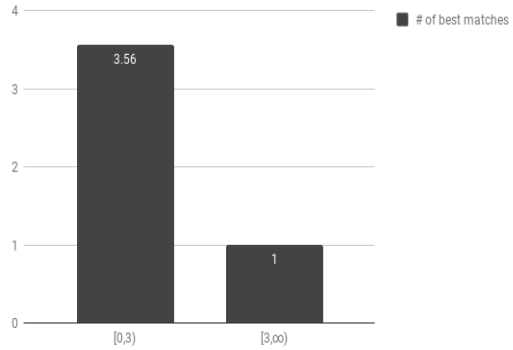


Figure 5.6: Android Wear API methods detection



(a) Performance for Wear API methods detection



(b) Average number of best matches

Figure 5.7: Results of Wear API methods detection

We use "interesting methods" to denote the API methods needed to build the constraint graph. These methods are listed in the first part of Table 5.2. The output of Ormib is a set of best matches for each interesting method. If the actual obfuscated method from the ground truth is in this reported set, we report that the method is detected correctly. Figure 5.6 shows the overall performance of Ormib. Recall is the ratio of the number of correctly reported interesting methods to the total number

interesting methods in the app. Precision is computed as the ratio between the number of correctly reported interesting methods and the total number of reported interesting methods. The F1 score is twice the product of recall and precision, divided by the sum of recall and precision. To provide additional characterization of the results, Figure 5.7a shows the recall, precision, and F1 score classified by the number of transitive callees the methods have. We divided the methods into two categories: one for methods with fewer than 3 transitive callees, and another for methods that have no less than 3 transitive callees. For the second category, the recall and precision are both 1.0. For the first category, the precision is slightly lower. Figure 5.7b shows another important metric: the average number of best matches. For methods with no less than 3 transitive callees, there is only one best match. For the remaining methods, more than one method has the same highest similarity score and our approach cannot tell which is the correct match.

Overall, a small number of transitive callees has a negative effect on the precision and the number of best matches, because more methods will appear to be matches due to the small amount of information that can be obtained from the calling relationships. In the extreme case, a method may have no callees at all; then the detection is based only on the fuzzy signature comparison. When the API methods have multiple best matching methods, each one of them will be tested and the one with the best performance in the static analysis of Wear notifications will be chosen to recover the model.

The running time of our approach has two separate components: (1) building the Bloom filters using call pairs, and (2) detecting a specific method using the Bloom filters. Bloom filters can be persisted and their building incurs one-time cost. This

Before Obfuscation												
	QuickLyric	Whatsapp	QKSMS	Loop	Silence	Tasks	Telegram	Toulibre	ArcusWeather	GroupMe	Slack	Signal
—NO—	2	1	7	1	3	1	1	1	2	2	5	3
#notify	2	1	6	1	3	1	1	1	1	1	3	3
#screens	0	0	6	0	0	0	0	0	3	1	2	0
#actions	2	1	18	3	7	3	2	2	1	5	3	7
After Obfuscation												
	QuickLyric	WhatsappUp	QKSMS	Loop	Silence	Tasks	Telegram	Toulibre	ArcusWeather	GroupMe	Slack	Signal
—NO—	0	0	0	0	0	0	0	0	0	0	0	0
#notify	0	0	0	0	0	0	0	0	0	0	0	0
#screens	0	0	0	0	0	0	0	0	0	0	0	0
#actions	0	0	0	0	0	0	0	0	0	0	0	0
After Replacement												
	QuickLyric	Whatsapp	QKSMS	Loop	Silence	Tasks	Telegram	Toulibre	ArcusWeather	GroupMe	Slack	Signal
—NO—	2	1	7	1	3	1	1	1	2	2	5	3
#notify	2	1	6	1	3	1	1	1	1	1	3	3
#screens	0	0	6	0	0	0	0	0	3	1	2	0
#actions	2	1	18	3	7	3	2	2	1	5	3	7

Table 5.3: Effects on model building for Android Wear notifications

process takes about half an hour per app in our evaluation. The time for matching a certain API method against a given app is less than 0.01 seconds.

Table 5.3 shows how the model changes when using Ormib. We computed the number of items in each category of the model before obfuscation, after obfuscation, and after incorporating the obfuscated signatures Ormib found. It is clear from the results that obfuscation will render the static model construction useless: no relevant APIs are detected and all categories of items in the model become empty. Using our method detection, the model building becomes obfuscation-resilient and produces the same model as for the unobfuscated apps.

5.3.3 Evaluation on Data Leakage Analysis for Android Cloud Services

Another client we considered is defined in work by Zuo et al. [134] on data leakage analysis. As Android apps become more complex, the cloud is used as the back-end in a growing number of apps. However, this could lead to increase in data leaks from

cloud service API methods. The work by Zuo et al. develops the LeakScanner approach which aims to understand such leaks and to design techniques to automatically identify them.

The cloud back-end needs service authentication from the apps to get information about which app issues the request and the resources need to be accessed. Misuse of user authentication and misconfiguration of user permissions are the root causes to data leaks related to cloud services. When using the cloud service API methods, it is necessary to pass the corresponding keys. Based on the keys, the cloud providers can identify the privilege of the user as superuser or normal user. Providing the wrong key could cause vulnerabilities. To study this issue, Zuo et al. inspected several cloud service API methods used in Android apps and determined the relevant key string values passed into the APIs. Then a further test on the type of the keys was performed.

This approach assumes knowledge of call sites to the appropriate cloud-access-related libraries (Azure, Amazon AWS, and Firebase). The list of relevant APIs methods is shown in Table 5.2. However, these libraries can be obfuscated as part of the obfuscation of an app that includes them. Because obfuscation can change the API method signatures, LeakScanner employs an obfuscation-resilient API method detection approach. The package structure, transitive callees, and some additional information for a method are used to build a hash value. The method name is not considered. Methods that have the same hash value as the cloud service API methods are used in following constant string propagation.

This hash comparison approach has certain limitations. First, the package structure can be changed significantly during obfuscation. As mentioned earlier, classes

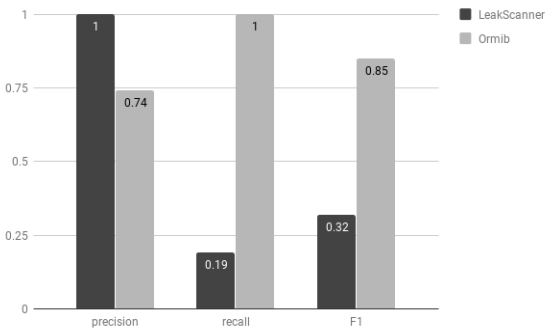
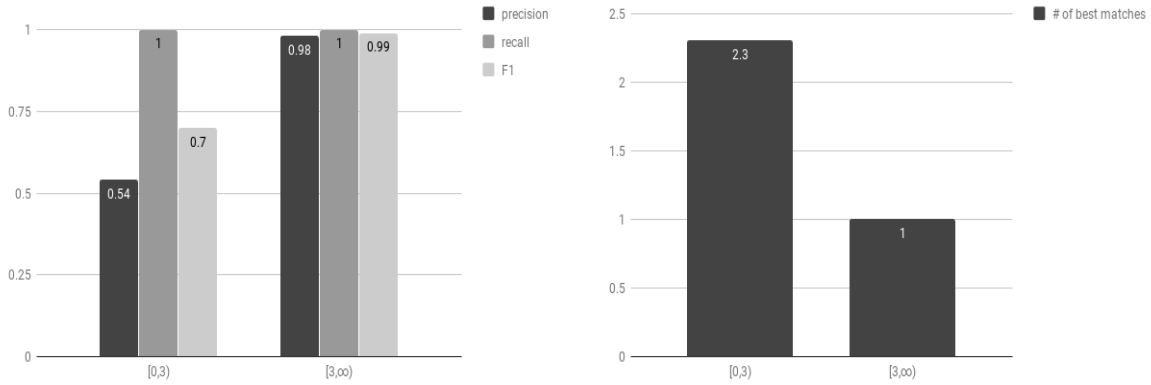


Figure 5.8: Results of comparison

from different packages may be merged into an artificial package, or the existing package structure can also be flattened. This kind of package transformations will change the hash value and render LeakScanner's detection ineffective. Second, additional callees can be inserted by the obfuscator, such as inserting string decryption methods when string encryption is enabled. The hash value will also be different because of these additional calls.

We compare the performance of Ormib with LeakScanner on *CloudData* using the same metrics as in Section 5.3.2. Figure 5.8 shows the comparison of precision, recall, and F1 score. Overall, Ormib has better recall and F1 score and does not miss any API method. LeakScanner can achieve a high precision because the hash value matching is a very strong requirement for reporting a match. However, due to the more complex obfuscation, it can only detect about 19% of the API methods. We further examined the performance of Ormib on *CloudData*. Figure 5.9a and Figure 5.9b show the performance based on the number of callees for a method. Without missing a single interesting method, methods with equal or greater than 3



(a) Performance of cloud service API methods detection

(b) Average number of best matches

Figure 5.9: Results of cloud service API methods detection

transitive callees also exhibit a high precision, about 98%. Furthermore, their average number of best matches is almost one. This means that ORMIB can identify the obfuscated method accurately. For the "small" methods—the ones with fewer than 3 transitive callees—the precision decays and the number of possible candidates increases. For example, one interesting cloud service API method that has the following signature "$BasicAWSCredentials : void\langle init\rangle(java.lang.String, java.lang.String)$" has no callees. Without this method, the overall precision increases to about 98% from 74% and the average number of best matches decreases from 2.3 to around 1.0. Figure 5.10 shows the number of callees for all considered cloud service API methods. Only one method has no callees at all and it affects the performance significantly. The average number of transitive callees for these cloud service API methods is about 84.

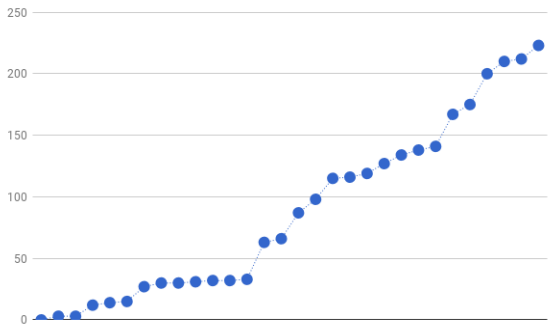


Figure 5.10: Number of callees

5.3.4 Evaluation on Analysis of Analytics Libraries for Android Apps

Analytics libraries are widely used by Android and Android Wear developers for targeted advertising and behavioral analysis, as well as for crash and performance reports. Ongoing work by Hailong Zhang et al. in the PRESTO research group, which is currently not described in any public document, performs analysis of API calls to the popular Google Analytics library. As a client, we consider one aspect of this analysis, which aims to statically determine and analyze the parameter values for the APIs methods described in the last part of Table 5.2. However, similarly to the other two analyses described earlier, the API methods may be concealed by obfuscation. In this experiment, we attempted to detect these Google Analytics API methods in the apps from *GoogleAnalyticsData*. Figure 5.11 shows the overall performance of ORMIB for this client analysis. Generally, ORMIB achieves a high precision, recall, and F1 score. Figure 5.12a and Figure 5.12b indicate the performance and the average number of best matches, classified by the number of transitive callees. As with the other two clients, the performance is relatively worse when a method has few callees.

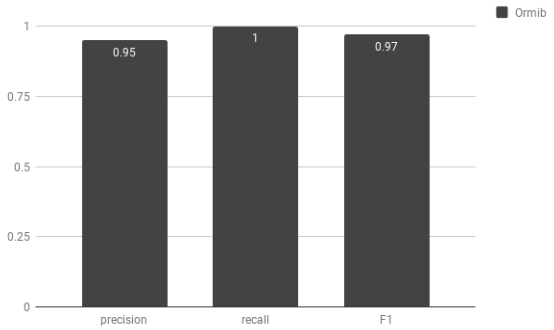
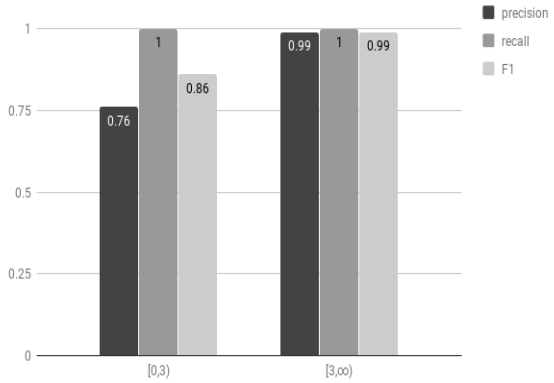
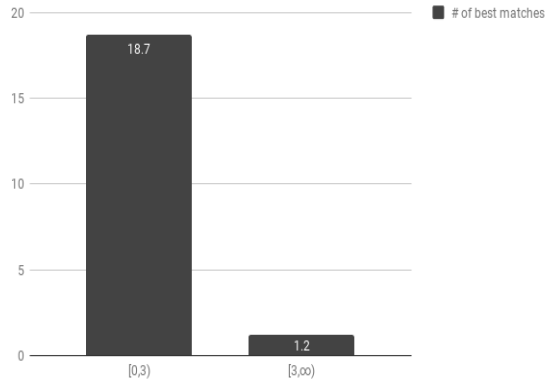


Figure 5.11: Results of Google Analytics API methods detection



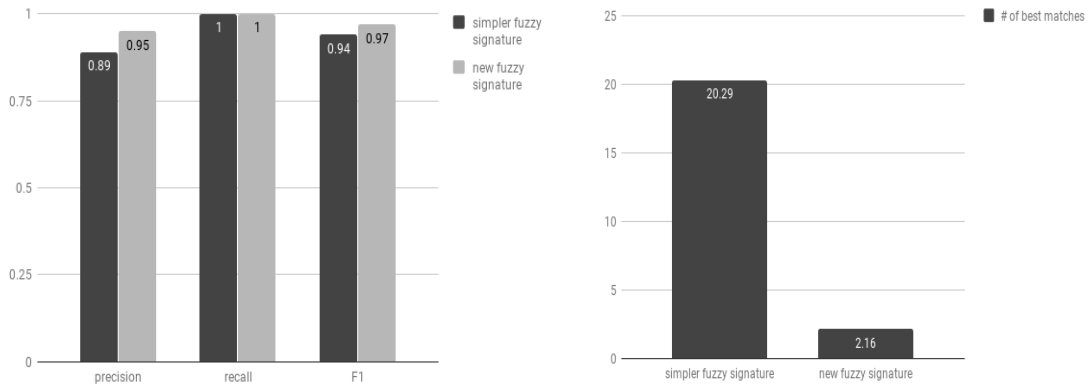
(a) Performance of Google Analytics API methods detection



(b) Average number of best matches

Figure 5.12: Results of Google Analytics API methods detection

Before obfuscation, this client analysis detects a total of 430,698 constant string values that flow to the relevant Google Analytics APIs (when applied to all apps from *GoogleAnalyticsData*). Identifying such constants is an essential component of this client analysis. After obfuscation, 0 such constants can be found and as a result the client analysis is fully disabled. We were able to fix this problem by replacing the



(a) Comparison of the performance

(b) Comparison of the average number of best matches

Figure 5.13: Comparison of the two fuzzy signatures

original method signatures with the obfuscated ones, as detected by ORMIB. After this substitution, all 430,698 constant string values could be recovered.

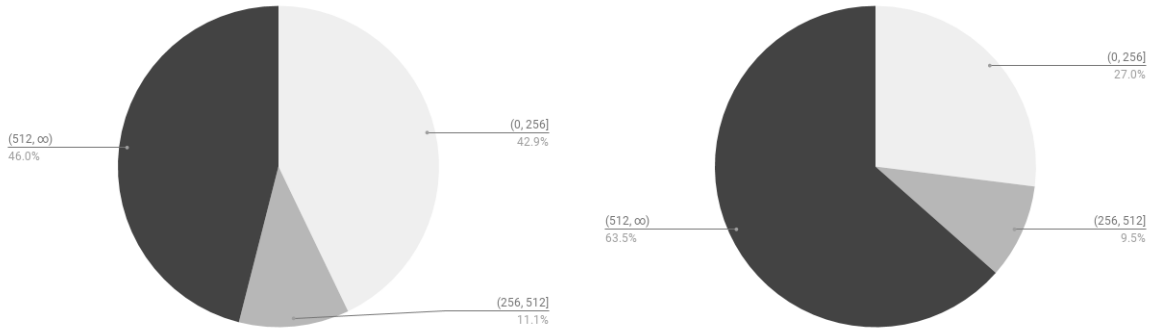
5.3.5 Comparison of Method Fuzzy Signatures

Our approach defines a new method fuzzy signature. We have argued that this signatures is more powerful than the simpler one used in Chapter 4. To support our argument, we compared the results of using these two different fuzzy signatures. Figure 5.13a shows this comparison for *WearData*; the results are similar for the other two datasets. Clearly, the new method fuzzy signature has much better precision and F1 score. Figure 5.13b shows the difference in the average number of best matches. Using the simpler fuzzy signatures will cause an increase in the number of matches because it captures fewer features for a method. To summarize, the new method fuzzy signature achieves a better false positive rate as well as a smaller number of best matches.

5.3.6 Comparison with Similarity Digests

In Section 5.1, we claimed that the approach in Chapter 4 does not perform well and a new approach should be designed for method detection. To support this claim, we compared the performance of ORMIB with the approach using similarity digests as defined in Chapter 4. Among the four similarity digests, `sdhash` and `TLSH` are not suitable because they have requirements on the minimum data size: the minimum size is 512 bytes for `sdhash` and 256 bytes for `TLSH`. The string feature for a single method is significantly smaller than that a class, an app, or a library. For all API methods in Table 5.2, the ratio of methods that do not satisfy this minimum-data requirement is substantial. Figure 5.14a shows the ratio of methods that have a string feature with size greater than 512 bytes, greater than 256 bytes and less or equal to 512 bytes, and less or equal to 256 bytes. Figure 5.14b shows the same information but using the new method fuzzy signature to build the string features. The sizes of new method fuzzy signatures are larger than those of the old ones due to their increased complexity. But even when using the new method fuzzy signatures, only about 64% of the methods can be handled by `TLSH` and 73% can be hashed using `sdhash`. Similar measurements for the simpler fuzzy signatures are 46% and 57% for `TLSH` and `sdhash`, respectively. Thus, `sdhash` and `TLSH` are not appropriate for method detection.

The other two similarity digests, `ssdeep` and `nilsimsa`, have no limitations on the data size. We investigated their performance. First, we built the string feature of each method using the approach from Chapter 4 and generated the similarity hashes using `ssdeep` and `nilsimsa`. Then we attempted to detect a given method by computing the similarity scores between it and each method in the app. The methods with the



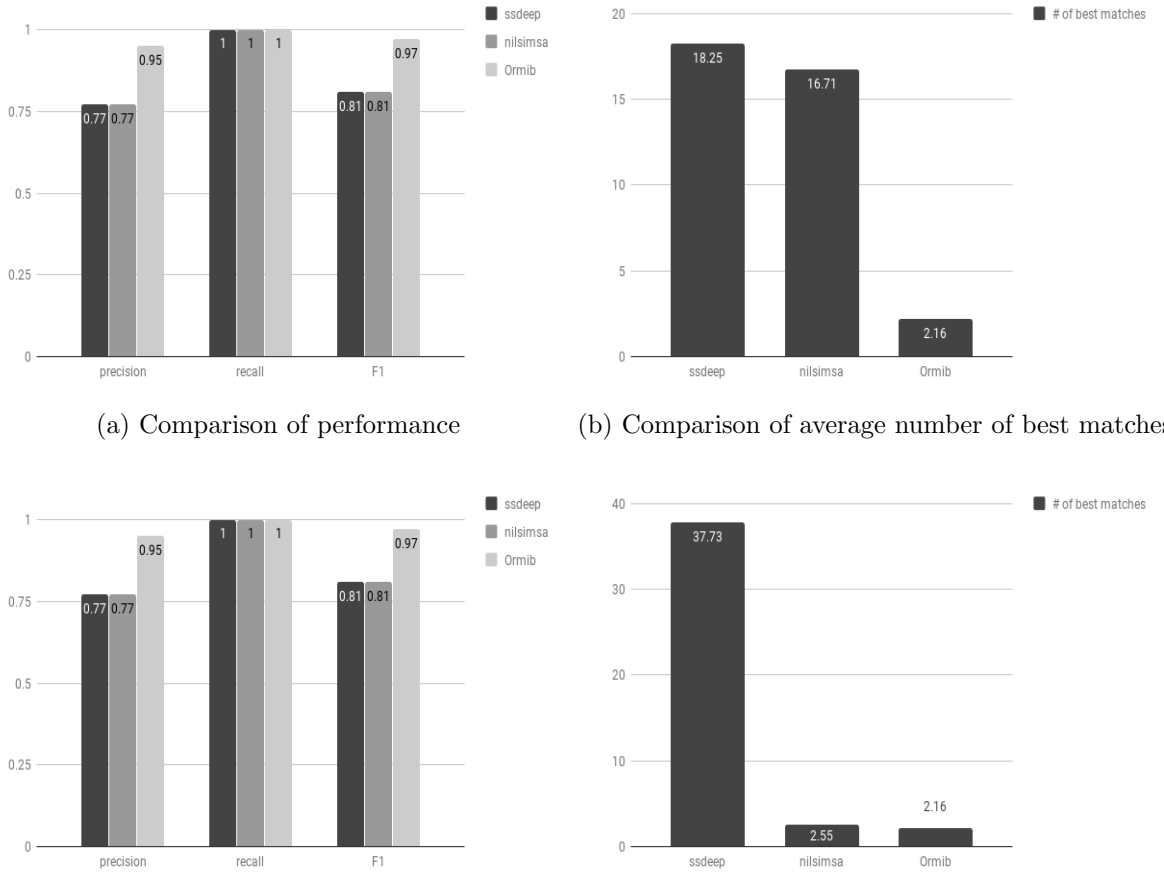
(a) Ratio of method sizes using simpler fuzzy signa-(b) Ratio of method sizes using new fuzzy signatures
tures

Figure 5.14: Method size ratio

best similarity scores were reported. The thresholds were set similarly to the ones from Chapter 4. For this experiment, we used data set *WearData*.

Figure 5.15a shows the result of this comparison. ORMIB has much better precision as well as F1 score than the approaches using `ssdeep` and `nilsimsa`. Figure 5.15b shows the average number of best matches. In some cases, more than one method will have the same best similarity score computed by `ssdeep` and `nilsimsa`. In such cases, all methods will be reported as best matches. ORMIB exhibits the smallest average number of best matches.

However, the difference may be caused by ORMIB's use of the new fuzzy signatures. To further investigate this possibility, we also measured the performance of the similarity digests using the new fuzzy signature in the process of building the method string features. Figure 5.15c and Figure 5.15d show the performance and the average amount of best matches after using the new fuzzy signatures. This change does not affect the performance of `ssdeep` and `nilsimsa` much, but the average number of



(a) Comparison of performance

(b) Comparison of average number of best matches

(c) Comparison of performance using new fuzzy signature

(d) Comparison of average number of best matches using new fuzzy signature

Figure 5.15: Results of comparison

best matches is affected. Matching based on `nilsimsa` becomes able to identify the best matching method, while the matching based on `ssdeep` becomes worse. In this experiment, Ormib still exhibits the best performance. In summary, for detecting a single method in an obfuscated Android apps, Ormib outperforms all approaches based on the similarity digests from Chapter 4.

5.4 Limitations

In our approach, we did not consider reflection obfuscation. This technique will change a method call to its corresponding reflection version. However, popular obfuscators typically do not provide this functionality. Another technique we do not consider is obfuscation via a packer. A packer will encrypt the implementation of the app and will only expose some artificial wrapper code. To handle this case, preprocessing to extract the actual code of the app is needed. Several approaches [77,94,102] are available for this purpose. Finally, we focus on the Java code in an Android app. Native code obfuscation is not considered.

5.5 Summary

In this chapter, we propose a new approach for method detection using interprocedural information and class-hierarchy-based fuzzy signatures, together with Bloom filters for efficiency. Our experimental results indicate that the proposed Ormib tool can achieve good performance for several client static analyses.

CHAPTER 6: Related Work

6.1 General Software Obfuscation

Researchers have explored various questions related to obfuscation. Schrittwieser et al. [97] measured the performance of different program analyses against obfuscation. They analyzed the resilience of obfuscation techniques for static analysis, dynamic analysis, and humans. Barak et al. [13] theoretically proved that a general perfect "virtual black box" obfuscation is impossible by creating a family of unobfuscatable functions. Collberg et al. [23] summarized the existing popular techniques used in Java obfuscation. The follow-up work of Collberg et al. [24] detailed how obfuscation was used in protecting general software programs, and discussed the differences among obfuscation, watermarking, and tamper-proofing. Wang et al. [117] conducted an empirical study of iOS app obfuscation. Some work has been done on evaluating the quality of obfuscation. Ceccato et al. [17] proposed an approach to assess the difficulty attackers had in understanding and modifying obfuscated code through controlled experiments involving human subjects. Anckaert et al. [3] developed a framework based on software complexity metrics measuring four program properties including code, control flow, data, and data flow. A number of obfuscating and deobfuscating transformations were evaluated based upon their impact on a set of these metrics, which was used to quantitatively evaluate the strength of the (de)obfuscating transformations. Some studies pay more attention to deobfuscation.

Coogan et al. [25] focused on analysis of instructions that affect the observable behavior of the obfuscated code. Madou et al. [75] proposed a graphic and interactive framework for code obfuscation and deobfuscation.

6.2 Android Obfuscation

There is a growing body of work related to Android obfuscation. One representative example is a new obfuscation approach that integrates several techniques including native code, variable packing, and dead code insertionc̆itekovacheva-iait13. Another study considered several obfuscation techniques and their properties (e.g., monotonicity) [39]. De-obfuscation approaches have also been proposed. One recent example is an approach based on a probabilistic model [14]. ProGuard is the only obfuscator considered in that work; further, the technique considers only the renaming of program elements (e.g., methods and classes), but not repackaging or control-flow modifications. Garcia et al. [41] proposed a machine learning approach to identify the family of Android malware even in the presence of obfuscation, by using Android APIs, reflection, and native calls as features. This study focused on anti-malware analysis and considered obfuscation-resilient properties different from the ones used in our work. We also investigate popular obfuscators and the techniques they employ, but our goal is to identify code features that can be used to identify third-party libraries and app clones.

6.3 Impact of Android Obfuscation

As described earlier, obfuscation of Android apps will affect many legitimate program analyses. Work on clone/repackage detection [19, 26, 27, 46, 48, 52, 78, 101, 106,

109, 114, 116, 127, 131, 132] found that obfuscation impaired detection results. Studies of malware detection [38, 44, 76, 88, 89, 103, 133] also showed that obfuscation was an obstacle to malware analysis. Besides these studies, Hammad et al. [47] examined the impact of obfuscation on Android anti-malware products by investigating 7 obfuscators and 29 obfuscation techniques. Additionally, there is a body of work on library detection for Android [9, 16, 19, 20, 27, 42, 45, 69, 74, 116]. Some of these techniques have obfuscation-resilient aspects, but as described in Section 4.1, several assumptions used in these tools are violated by commonly-used Android obfuscators. Our approach is specifically designed to exploit obfuscation-resilient features. AdDetect [79] and PEDAL [71] used machine learning to detect advertisement libraries in Android apps. However, these approaches can only handle this particular category of libraries, while our approach is applicable to any library. Li et al. [68] investigated the common libraries used in Android apps by mining libraries and refining packages among a large number of apps. However, the purpose of this work is to study characteristics of these Android libraries, e.g., their popularity. The resulting library package list is not aimed at detecting libraries in a given obfuscated app. To summarize, although some of these tools claim to still work well in the presence of obfuscation, none could completely eliminate the obfuscation effects, either in their assumptions or in their experimental evaluations. This highlights the importance of understanding obfuscation and designing obfuscator-specific analysis techniques.

6.4 Provenance Analysis

As far as we know, there is no prior work on techniques to identify the provenance of obfuscated Android apps. There is some work on identifying the toolchain

used in compiling a given program. Rosenblum et al. [90] proposed a machine learning approach to discover the type of compiler used to create a given binary. They formulated the provenance problem as a classification problem and tried to discover characteristics of binary code that are associated with particular toolchain components and development models. We also make use of a machine learning method, but for the purpose of obfuscator identification for Android apps. Due to these different targets, the details of the two techniques are substantially different.

6.5 Similarity Digests

Similarity digests are somewhat similar to standard hashes, but allow one to measure the similarity between two data objects based on comparison of their digests. Several different techniques have been developed in this area. One scheme is based on feature extraction (e.g., `sdhash` [91]), which is a statistical approach for selecting fingerprinting features that are most likely to be unique to a data object. Another scheme is fuzzy hashing (e.g., `ssddep` [64]), which uses rolling hashes and produces a pseudo-random value where each part of the digest only depends on a fragment of the input. The result is treated as a string and is compared with other digests on the basis of edit distance. Roussev et al. [92] proposed a similarity approach that uses partial knowledge of the internal object structure and Bloom filters. Follow-up work [93] attempts to balance performance and accuracy by maintaining hash values at several resolutions, but requires understanding of the syntactic structure of data objects, which affects its generality. Locality sensitive hashing is also a type of similarity digest. Locality-sensitive hashing [54] was originally used for a randomized

hashing framework for efficiently approximating nearest neighbor search in high dimensional space. Since then, it has become one of the most popular solutions for this type of problem. In general, there are two approaches: (1) approximating the distance between data objects by comparing their hashes, and (2) mapping similar objects to the same bucket in a hash table, after which search is performed within a bucket. The hashing functions `TLSH` [81] and `nilsimsa` [28] discussed in Section 4.6 belong to the former category. For the second category, many hash functions (e.g., [18,29,31]) have been developed for Euclidean distance, angle-based distance, Hamming distance, etc. However, such distance metrics cannot be applied directly to our data.

6.6 General Software Clone Detection

Significant body of work exists on software clone detection. Clone detection based on semantic similarity has employed program dependence graphs (PDG), which contain control dependence and data dependence information. Komondoor et al. [63] adopted program slicing to detect isomorphic PDG subgraphs and code clones. Krinke [65] designed an approach to finds maximal similar PDG subgraphs. Liu et al. [72] developed an approach to detect software plagiarism by mining and comparing PDGs. Higo et al. [50] proposed a PDG-based incremental approach to detect clones. Code clone detection results or their intermediate products were persisted using databases, and could be used in the next code clone detection. Hummel et al. [53] introduced a clone detection method based on data-flow graph isomorphism. Jia et al. [56] proposed a clone detection algorithm that incorporated a combination of lexical and local dependence analysis. Gabel et al. [40] designed a clone detection method using PDG

graph isomorphism and improved scalability by reducing the PDG into trees, by selecting PDG subgraphs related to structured syntax. Kamalpriya et al. [59] enhanced PDG-based clone detection by using approximate subgraph matching.

Besides using PDGs, tree-based and metric-based detection are also widely used in clone detection. Jiang et al. [57] compared the parse trees of two programs to detect clones. Smith et al. [105] proposed a technique to detect clones based on block similarity. The similarity was computed using a fingerprint of statements. Lavoie et al. [66] introduced a method to construct clone oracles based on Levenshtein metric. This approach could also be used to measure other clone detection techniques. Sæbjørnsen et al. [95] presented a clone detection algorithm for binary code using tree similarity. White et al. [123] introduced a learning-based detection technique which links patterns mined at the lexical level with patterns mined at the syntactic level. Some researchers have employed semantic web techniques to assist with clone detection. Keivanloo et al. [61] provided a two-step approach using pattern matching via semantic web queries and content matching based on set intersection for similarity measurements. Schügerl [98] proposed large-scale clone detection using description logic and a semantic web reasoner.

Clone detection methods depending on run-time information have also been developed. Jiang et al. [58] defined functional equivalence as generating the same output given the same input. Their approach automatically provides random inputs to an extracted code fragment and compares the output values. Khan et al. [62] explored the run-time effects of clones that were executed under various system usage scenarios. For functional programming languages, Kamiya [60] presented a code clone detection approach using run-time information.

Some researchers have focused on improving the efficiency of clone detection. Svajlenko et al. [110] explored the performance and applicability of one clone detection framework. Sajnani et al. [96] proposed a technique to use the MapReduce framework to horizontally scale clone detection across multiple machines.

CHAPTER 7: Conclusions

The introduction of Android has significantly influenced the mobile computing industry. The large market and the potential revenues from it have resulted in widespread use of obfuscation to prevent reverse engineering and to protect intellectual property. However, various legitimate program analyses are necessary to keep Android apps robust and secure, and obfuscation impedes many of these analyses. This dissertation makes several contributions toward strengthening program analyses and tools for Android apps in the presence of obfuscation.

7.1 Android Obfuscator Identification

Identifying an obfuscator and its configuration provides useful knowledge on how a specific obfuscated Android app was constructed. This information is beneficial for various clients. For example, if it is determined that the package structure has been modified, any assumptions that rely on package boundaries can be invalidated. As another example, if an app is packed, preprocessing is needed to determine that actual program code before any further analyses can be performed. In Chapter 3 we proposed an efficient and effective approach for provenance analysis of obfuscated Android apps. Using features extracted from APKs, we construct several classifiers that determine which obfuscator was used and how it was configured. Although the approach is based on various assumptions about the obfuscation process, in practice it exhibits high accuracy on real-world open-source apps and provides insights about

the use of obfuscation in popular Google Play apps. The information extracted with the proposed techniques can be potentially used to improve and refine a variety of analyses and tools for Android.

7.2 Obfuscation-Resilient Detection of Third-party Libraries and App Clones

Two important analyses for Android apps are third-party library detection and app clone detection. Libraries are essential components of Android apps and are used pervasively. They facilitate the development of rich app functionality but also introduce security/performance concerns and complicate various app analyses (e.g., malware analysis and clone detection). App cloning has become a serious threat to the Android app market as it facilitates plagiarism and malware.

Library detection and clone detection are useful techniques to tackle these problems. Unfortunately, obfuscation present significant difficulties for both categories of techniques. As the second contribution of this dissertation, we perform an investigation of obfuscation techniques used by popular Android obfuscators and characterize their impact on library/clone detection analyses. Our conclusion is that even state-of-the-art approaches for library/clone detection can benefit from increased resilience to obfuscation. In Chapter 4 we show that calling relationships and class hierarchy relationships can be used to create effective obfuscation-resilient library/clone detection, while similarity digests can ensure practical analysis cost. Using apps from several datasets and a large number of third-party libraries, we demonstrate that the resulting library detection tool Orlis and app clone detection tool Orcis outperform the current state-of-the-art detection tools.

7.3 Obfuscation-Resilient Detection of Methods

The third contribution of this dissertation, described in Chapter 5, focuses on the detection of a single method. Such detection is necessary for various static analyses including security analysis, static model building, test generation, and code instrumentation. Our study on several existing program analyses shows that naive signature-based method detection renders these analyses useless for obfuscated apps. We propose a method detection approach based on a new type of fuzzy signature and on interprocedural information about methods. Efficiency is addressed with the help of Bloom filters. Experimental evaluation on three clients shows that our Ormib method detection tool is precise, obfuscation-resilient, and scalable for real-world static analyses and apps.

BIBLIOGRAPHY

[1] *Allatori*, 2017. *www.allatori.com*.

[2] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon. Androzoo: Collecting millions of android apps for the research community. In *MSR*, pages 468–471, 2016.

[3] B. Anckaert, M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel. Program obfuscation: A quantitative approach. In *QoP*, pages 15–20, 2007.

[4] *Androguard*, 2017. *github.com/androguard/androguard*.

[5] *Apktool*. *ibotpeaches.github.io/Apktool/*.

[6] *Appbrain: Android library statistics*, Jan. 2018. *www.appbrain.com/stats/ libraries*.

[7] *Appbrain: Number of Android applications*, Jan. 2018. *www.appbrain.com/ stats/number-of-android-apps*.

[8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, pages 259–269, 2014.

[9] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in Android and its security applications. In *CCS*, pages 356–367, 2016.

[10] M. Backes, S. Bugiel, E. Derr, S. Gerling, and C. Hammer. R-Droid: Leveraging Android app analysis with static slice optimization. In *ASIACCS*, pages 129–140, 2016.

[11] *Baksmali*. *github.com/JesusFreke/smali*.

[12] *Bangcle*, 2017. *www.bangcle.com/*.

[13] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59:4, 2012.

[14] B. Bichsel, V. Raychev, P. Tsankov, and M. T. Vechev. Statistical deobfuscation of Android applications. In *CCS*, pages 343–355, 2016.

[15] *Google's Android generates 31 billion revenue*, 2017. *www.bloomberg.com/news/articles/2016-01-21/google-s-android-generates-31-billion-revenue-oracle-says-ijor8hvt*.

[16] T. Book, A. Pridgen, and D. S. Wallach. Longitudinal analysis of Android ad library permissions. *CoRR*, abs/1303.0857, 2013.

[17] M. Ceccato, M. D. Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. Towards experimental evaluation of code obfuscation techniques. In *QoP*, pages 39–46, 2008.

[18] M. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388, 2002.

[19] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *ICSE*, pages 175–186, 2014.

[20] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on Android and iOS. In *IEEE S&P*, pages 357–376, 2016.

[21] *CodeMatch*, 2017. *www.st.informatik.tu-darmstadt.de/artifacts/code match*.

[22] *CodeMatchData*. *www.st.informatik.tu-darmstadt.de/artifacts/codemat ch*.

[23] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report TR-148, U. Auckland, 1997.

[24] C. S. Collberg and C. Thomborson. Watermarking, tamper-proffing, and obfuscation: Tools for software protection. *IEEE Trans. Software Engineering*, 28:735–746, 2002.

[25] K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *CCS*, pages 275–284, 2011.

[26] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on Android markets. In *ESORICS*, pages 37–54, 2012.

[27] J. Crussell, C. Gibler, and H. Chen. AnDarwin: Scalable detection of Android application clones based on semantics. *IEEE Trans. Mobile Computing*, 14:2007–2019, 2015.

[28] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. An open digest-based technique for spam detection. In *ISCA*, pages 559–564, 2004.

[29] A. Dasgupta, R. Kumar, and T. Sarlós. Fast locality-sensitive hashing. In *KDD*, pages 1073–1081, 2011.

[30] *DashO*, 2017. *www.preemptive.com/company.*

[31] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, pages 253–262, 2004.

[32] *Dex2jar. github.com/pxb1988/dex2jar.*

[33] *Dexdump source code. android.googlesource.com/platform/dalvik/+/eclair-release/dexdump/DexDump.c.*

[34] *Security bug resolved in the Dropbox SDKs for Android*, 2017. *blogs.dropbox.com/developers/2015/03/security-bug-resolved-in-the-dropbox-sdks-for-android.*

[35] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in Android applications. In *CCS*, pages 73–84, 2013.

[36] W. Enck, D. Octeau, P. D. McDaniel, and S. Chaudhuri. A study of Android application security. In *USENIX Security*, 2011.

[37] *F-Droid Repository*, 2017. *f-droid.org.*

[38] P. Faruki, A. Bharmal, V. Laxmi, M. S. Gaur, M. Conti, and M. Rajarajan. Evaluation of Android anti malware techniques against Dalvik bytecode obfuscation. In *TrustCom*, pages 414–421, 2015.

[39] F. C. Freiling, M. Protsenko, and Y. Zhuang. An empirical evaluation of software obfuscation techniques applied to Android APKs. In *ICST*, pages 315–328, 2014.

[40] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE*, pages 321–330, 2008.

[41] J. Garcia, M. Hammad, and S. Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. *TOSEM*, 26(3):11:1–11:29, 2018.

[42] L. Glanz, S. Amann, M. Eichberg, M. Reif, B. Hermann, J. Lerch, and M. Mezini. CodeMatch: Obfuscation won't conceal your repackaged app. In *FSE*, pages 638–648, 2017.

[43] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of Android applications in DroidSafe. In *NDSS*, 2015.

[44] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. R. Cavalli. Protection against code obfuscation attacks based on control dependencies in Android systems. In *SERE*, pages 149–157, 2014.

[45] M. C. Grace, W. Zhou, X. Jiang, and A. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WiSec*, pages 101–112, 2012.

[46] Q. Guan, H. Huang, W. Luo, and S. Zhu. Semantics-based repackaging detection for mobile apps. In *ESSoS*, pages 89–105, 2016.

[47] M. Hammad, J. Garcia, and S. Malek. A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In *ICSE*, pages 421–431, 2018.

[48] S. Hanna, L. Huang, E. X. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among Android applications. In *DIMVA*, pages 62–81, 2012.

[49] R. Harrison. Investigating the effectiveness of obfuscation against Android application reverse engineering. Technical Report RHUL-MA-2015-7, Royal Holloway University of London, 2015.

[50] Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto. Incremental code clone detection: A PDG-based approach. In *WCRE*, pages 3–12, 2011.

[51] W. Hu, D. Octeau, and P. Liu. Duet: Library integrity verification for Android applications. In *WiSec*, pages 141–152, 2014.

[52] H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithms. In *TRUST*, pages 169–186, 2013.

[53] B. Hummel, E. Jürgens, and D. Steidl. Index-based model clone detection. In *IWSC*, pages 21–27, 2011.

[54] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[55] *Jadx. github.com/skylot/jadx.*

[56] Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsushita. KClone: A proposed approach to fast precise code clone detection. In *IWSC*, 2009.

[57] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[58] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA*, pages 81–92, 2009.

[59] C. M. Kamalpriya and P. Singh. Enhancing program dependency graph based clone detection using approximate subgraph matching. In *IWSC*, pages 61–67, 2017.

[60] T. Kamiya. An execution-semantic and content-and-context-based code-clone detection and analysis. In *IWSC*, pages 1–7, 2015.

[61] I. Keivanloo, C. K. Roy, and J. Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *IWSC*, pages 36–42, 2012.

[62] M. A. A. Khan, K. A. Schneider, and C. K. Roy. Active clones: Source code clones at runtime. *ECEASST*, 63, 2014.

[63] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Static Analysis Symposium*, pages 40–56, 2001.

[64] J. D. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97, 2006.

[65] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, University of Passau, 2003.

[66] T. Lavoie and E. Merlo. Automated type-3 clone oracle using levenshtein metric. In *IWSC*, pages 34–40, 2011.

[67] *Legu*, 2017. *legu.qcloud.com/*.

[68] L. Li, T. F. Bissyandé, J. Klein, and Y. L. Traon. An investigation into the use of common libraries in Android apps. In *SANER*, pages 403–414, 2016.

[69] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo. LibD: Scalable and precise third-party library detection in Android markets. In *ICSE*, pages 357–376, 2017.

[70] *Apps with most 3rd party libraries*, 2017. *www.privacygrade.org/stats*.

[71] B. Liu, B. Liu, H. Jin, and R. Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. In *MobiSys*, pages 89–103, 2015.

[72] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *KDD*, pages 872–881, 2006.

[73] S. Luo and P. Yan. Fake apps feigning legitimacy. Technical report, Trend Micro, 2014.

[74] Z. Ma, H. Wang, Y. Guo, and X. Chen. LibRadar: Detecting third-party libraries in Android apps. In *ICSE*, pages 641–644, 2016.

[75] M. Madou, L. V. Put, and K. D. Bosschere. LOCO: An interactive code (de)obfuscation tool. In *PEPM*, pages 140–144, 2006.

[76] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on Android malware. *Computers & Security*, 51:16–31, 2015.

[77] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *ACSAC*, pages 431–441, 2007.

[78] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu. Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection. *IEEE Trans. Reliability*, 65(4):1647–1664, 2016.

[79] A. Narayanan, L. Chen, and C. K. Chan. AdDetect: Automated detection of Android ad libraries using semantic analysis. In *ISSNIP*, pages 1–6, 2014.

[80] J. Oliver, C. Cheng, and Y. Chen. TLSH–A locality sensitive hash. In *CTC*, pages 7–13, 2013.

[81] J. Oliver, S. Forman, and C. Cheng. Using randomization to attack similarity digests. In *ATIS*, pages 199–210, 2014.

[82] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl. To pin or not to pin—helping app developers bullet proof their TLS connections. In *USENIX Security*, pages 239–254, 2015.

[83] A. Özgür, L. Özgür, and T. Güngör. Text categorization with class-based and corpus-based keyword selection. In *ISCIS*, pages 606–615, 2005.

[84] *Discovering a major security hole in Facebook's Android SDK*, 2017. *blog.parse.com/learn/engineering/discovering-a-major-security-hole-in-facebooks-android-sdk/*.

[85] *PiggyBackedData*. *github.com/serval-snt-uni-lu/Piggybacking*.

[86] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications. In *NDSS*, 2014.

[87] *ProGuard*, 2017. *developer.android.com/studio/build/shrink-code.html*.

[88] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: Evaluating Android anti-malware against transformation attacks. In *CCS*, pages 329–334, 2013.

[89] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: evaluating Android anti-malware against transformation attacks. *TIFS*, 9(1):99–108, 2014.

[90] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. In *ISSTA*, pages 100–110, 2011.

[91] V. Roussev. Hashing and data fingerprinting in digital forensics. *IEEE S&P*, 7:49–55, 2009.

[92] V. Roussev, Y. Chen, T. Bourg, and G. G. R. III. md5bloom: Forensic filesystem hashing revisited. *Digital Investigation*, 3:82–90, 2006.

[93] V. Roussev, G. G. Richard, and L. Marziale. Multi-resolution similarity hashing. *Digital Investigation*, 4:105–113, 2007.

[94] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *ACSAC*, pages 289–300, 2006.

[95] A. Sæbjørnsen, J. Willcock, T. Panas, D. J. Quinlan, and Z. Su. Detecting code clones in binary executables. In *ISSTA*, pages 117–128, 2009.

[96] H. Sajnani and C. V. Lopes. A parallel and efficient approach to large scale clone detection. In *IWSC*, pages 46–52, 2013.

[97] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl. Protecting software through obfuscation: can it keep pace with progress in code analysis? *ACM Computing Surveys*, 49, 2016.

[98] P. Schügerl. Scalable clone detection using description logic. In *IWSC*, pages 47–53, 2011.

[99] *sdhash*, 2017. *roussev.net/sdhash/sdhash.html*.

[100] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim. FlexDroid: Enforcing in-app privilege separation in Android. In *NDSS*, 2016.

[101] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang. Towards a scalable resource-driven approach for detecting repackaged Android applications. In *ACSAC*, pages 56–65, 2014.

[102] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee. Eureka: A framework for enabling static malware analysis. In *ESORICS*, pages 481–500, 2008.

[103] T. Shen, Y. Zhongyang, Z. Xin, B. Mao, and H. Huang. Detect Android malware variants using component based topology graph. In *TrustCom*, pages 406–413, 2014.

[104] *Scikit-learn framework*, 2017. *scikit-learn.org/stable/*.

[105] R. Smith and S. Horwitz. Detecting and measuring similarity in code clones. In *IWSC*, 2009.

[106] C. Soh, H. B. K. Tan, Y. L. Arnatovich, and L. Wang. Detecting clones in Android applications through analyzing user interfaces. In *ICPC*, pages 163–173, 2015.

[107] *Soot Analysis Framework*, 2017. *www.sable.mcgill.ca/soot*.

[108] *ssdeep*, 2017. *ssdeep-project.github.io/ssdeep/index.html*.

[109] M. Sun, M. Li, and J. C. Lui. DroidEagle: Seamless detection of visually similar Android apps. In *WiSec*, pages 9:1–9:12, 2015.

[110] J. Svajlenko, I. Keivanloo, and C. K. Roy. Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *IWSC*, pages 16–22, 2013.

[111] *Backdoor in Baidu Android SDK puts 100 million devices at risk*, 2017. *www.thehackernews.com/2015/11/android-malware-backdoor.html*.

[112] *Facebook SDK vulnerability puts millions of smartphone users' accounts at risk*, 2017. *www.thehackernews.com/2014/07/facebook-sdk-vulnerability-puts.html*.

[113] *Warning: 18,000 Android apps contains code that spy on your text messages*, 2017. *www.thehackernews.com/2015/10/android-apps-steal-sms.html*.

[114] M. L. Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk. Revisiting Android reuse studies in the context of code obfuscation and library usages. In *MSR*, pages 242–251, 2014.

[115] N. Viennot, E. Garcia, and J. Nieh. A measurement study of Google Play. In *SIGMETRICS*, pages 221–233, 2014.

[116] H. Wang, Y. Guo, Z. Ma, and X. Chen. Wukong: A scalable and accurate two-phase approach to Android app clone detection. In *ISSTA*, pages 71–82, 2015.

[117] P. Wang, Q. Bao, L. Wang, S. Wang, Z. Chen, T. Wei, and D. Wu. Software protection on the go: A large-scale empirical study on mobile app obfuscation. In *ICSE*, pages 26–36, 2018.

[118] Y. Wang and A. Rountev. Profiling the responsiveness of Android applications via automated resource amplification. In *MobileSoft*, pages 48–58, 2016.

[119] Y. Wang and A. Rountev. Who changed you? Obfuscator identification for Android. In *MobileSoft*, pages 154–164, 2017.

[120] Y. Wang, H. Wu, H. Zhang, and A. Rountev. Orlis: Obfuscation-resilient library detection for Android. In *MobileSoft*, pages 13–23, 2018.

[121] T. Watanabe, M. Akiyama, F. Kanei, E. Shioji, Y. Takata, B. Sun, Y. Ishi, T. Shibahara, T. Yagi, and T. Mori. A study on the vulnerabilities of mobiles apps associated with software modules. *arXiv preprint arXiv:1702.03112*, 2017.

[122] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *CCS*, pages 1329–1341, 2014.

[123] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *ASE*, pages 87–98, 2016.

[124] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *USENIX Security*, pages 499–514, 2015.

[125] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static window transition graphs for Android. In *ASE*, pages 658–668, 2015.

[126] W. Yang, J. Li, Y. Zhang, Y. Li, J. Shu, and D. Gu. APKLancet: Tumor payload diagnosis and purification for Android applications. In *ASIACCS*, pages 483–494, 2014.

[127] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *WiSec*, pages 25–36, 2014.

[128] H. Zhang and A. Rountev. Analysis and testing of notifications in Android Wear applications. In *ICSE*, pages 347–357, 2017.

[129] H. Zhang, H. Wu, and A. Rountev. Automated test generation for detection of leaks in Android applications. In *AST*, pages 64–70, 2016.

[130] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. L. Spina, and E. Moser. FSquaDRA: Fast detection of repackaged applications. In *CODASPY*, pages 130–145, 2014.

[131] W. Zhou, Y. Zhou, M. C. Grace, X. Jiang, and S. Zou. Fast, scalable detection of "Piggybacked" mobile applications. In *CODASPY*, pages 185–196, 2013.

[132] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *CODASPY*, pages 317–326, 2012.

[133] Y. Zhou and X. Jiang. Dissecting Android malware: characterization and evolution. In *IEEE S&P*, pages 95–109, 2012.

[134] C. Zuo, Z. Lin, and Y. Zhang. Why does your data leak? Uncovering the data leakage in cloud from mobile apps. In *IEEE S&P*, 2019.