Fingerprinting Android Obfuscation Tools Using Visualization

by

Ye Ning

Previous Degrees (Master of Physics, Institution of Applied Physics and Computational Mathematics in China, 2005)

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s): Natalia Stakhanova, PhD, Computer Science

Examining Board: Rongxing Lu, PhD, Computer Science, Chair
Suprio Ray, PhD, Computer Science
Erik Scheme, PhD, Electrical & Computer Engineering

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

November, 2017

©Ye Ning, 2018
Abstract

Android has become one of the most popular mobile device operating system. Indeed, its security issues have attracted a lot of attention. One of the major security concern is use of obfuscation strategies to evade anti-malware solutions. Android malware authors are increasingly using code obfuscation tools and techniques to hide malicious code. In this work, we introduce a new method to fingerprint the obfuscators used in Android apps by virtue of visualization analysis. First, binary files contained in Android apps are visualized into images. Then two types of statistical features are calculated upon those images. Those features are analyzed and synthesized into effective information to fingerprint the obfuscators used. The results show that our visualization-based method is able to reveal the information about the obfuscators used in the Android application development.
Dedication

This thesis work is dedicated to my wife, Xia Li, who has been a constant source of support and encouragement during the challenges of graduate school and life. I am truly thankful for having you in my life.

This work is also dedicated to my parents who always love me unconditionally. The good examples they set have taught me to work hard for the things that I aspire to achieve.
Acknowledgements

I would like to start by expressing my sincere gratitude to my supervisor Dr. Natalia Stakhanova for her continuous support, patience, motivation and immense knowledge. Her guidance helped me in all the time of research and writing of this thesis. I also want to take the chance to thank my colleagues Yan Li, Ratinder Kaur, Hugo Gonzalez. Yan gave help in establishing the first version of the dataset. Hugo contributed his ideas in the research work while Ratinder helped in writing this thesis and related paper.

Without help of those people, this work cannot even be in shape.
Table of Contents

Abstract ii

Dedication iii

Acknowledgments iv

Table of Contents vi

List of Tables viii

List of Figures x

1 Introduction 1

2 Background 6
   2.1 Android OS Overview ................................. 6
   2.2 Obfuscators .......................................... 8
   2.3 Visualization ...................................... 12
   2.4 Image Texture Analysis .............................. 13

3 Related Work 16
   3.1 Android Security .................................. 16
   3.2 Obfuscation and Visualization ...................... 19

4 The Proposed Solution 25
   4.1 Procedure .......................................... 25
4.2 Preprocessing .................................................. 26
4.3 Image Generation ............................................. 27
4.4 Features Extraction .......................................... 29
  4.4.0.1 Co-occurrence Matrix ................................. 33
  4.4.0.2 Haralick Vector ....................................... 37
4.5 Classification .................................................. 38

5 Evaluation Study ................................................ 40
  5.1 Experiment Setup .......................................... 40
  5.2 Dataset ........................................................ 41
  5.3 Evaluation of Extracted Features ......................... 44
    5.3.1 First-Order Statistical Features .................... 44
    5.3.2 Second-Order Statistical Features .................. 50
    5.3.3 Difference Matrix ..................................... 52
  5.4 Classification .............................................. 54

6 Discussion and Conclusion ..................................... 60

Bibliography .......................................................... 63

Vita
List of Tables

1.1 Market Share of Different Smart Phone Platform ........................................ 2
2.1 List of Obfuscators ....................................................................................... 10
3.1 Image Texture Analysis Method Used in Related Work ............................... 23
4.1 Components of Haralick Vector ................................................................. 37
5.1 Description of Dataset ................................................................................ 43
5.2 Results of Classification Using the First-Order Statistics ............................. 50
5.3 Confusion Matrix. a → Allatori; b → Jshrink; c → Klassmaster; d → ProGuard; e → DexGuard ................................................................. 50
5.4 Haralick Features of Original “monakhv” ................................................ 51
5.5 Difference Matrix Based on the Gray Scale Image of “a2dp”. 1 → Allatori with the option of obfuscating class names; 2 → Klassmaster obfuscating class names; 3 → Apkprotect; 4 → Bangcle; 5 → Original without using any obfuscators; 6 → Using ProGuard with all three functions on, they are obfuscation without using dictionary, shrinking and optimization; 7 → ProGuard without Optimization .............................. 52
5.6 Difference Matrix Based on Color Image of “a2dp” With Different Obfuscation Mode, with SimChi means simplified Chinese, Eng is short for English, default is obfuscation without any user defined dictionary 54
5.7 The Feature Group ...................................................................................... 55
5.8 A Binary Classification on Obfuscated and Non-obfuscated Apps ............ 55
5.9 Fingerprinting the Obfuscators .............................. 59
List of Figures

2.1 Structure of an APK file .......................................... 7
2.2 Use of Obfuscators .................................................. 11
2.3 Visualization Used in Malware Analysis [58] .................... 13
3.1 The Flow of Fingerprinting Process ............................... 21
4.1 The Flow of Fingerprinting Process ............................... 26
4.2 Generation of Gray Scale Images ................................. 28
4.3 Two Types of Original Images for “a2dp”: (a) Gray scale image, (b) Color image ......................................... 29
4.4 The spatial co-occurrence calculations [73] ....................... 35
4.5 The four directions of adjacency for calculating the co-occurrence matrix, where “X” is the pixel under investigation, with eight nearest-neighbor pixels labeled in the framework to describe pixel connectivity [73]. ........................................ 35
5.1 Comparison between the Gray Images originated from the app “com.cepmuvakkit.times”: (a) Non-obfuscated, (b) KlassMaster, (c) Allatori, (d) DexGuard, (e) Jshrink, (f) ProGuard ........................................ 43
5.2 Comparison between the Gray Image and its First-Order Statistics of the app “dendroid”: (a) Original Gray Tone Image, (b) Result of Four First-Order Statistics ........................................ 46
5.3 Shannon Entropy is calculated on different apps. (a) For “am.ed.exportcontacts”, Bangcle (Left), ProGuard (Right) (b) For “se.johanhil.clipboard”, Bangcle (Left), ProGuard (Right)  

5.4 3D graph for 5 obfuscators used on the same collection of 325 apps, using 3 first-order statistical quantities as coordinates.  

5.5 Gray Scale Image of “monakhv”.  

5.6 The impact of five different obfuscators on the same 325 samples.  

5.7 Classification Upon the 325 Samples Dealt By 5 Obfuscators: (a) Bagging, (b) K-NN, (c) SVM, (d) Regression.
Chapter 1

Introduction

Smart phones as well as other mobile devices are becoming increasingly important in our daily life. However, the assets of mobile phone/devices users, such as the contacts list, the log of social network access, browsing history or even the users’ confidential banking credentials have attracted intense attention from malware developers. This makes the security of the operating system used by the mobile smart devices an important issue.

The two mainstream mobile smart devices platforms are Android and iOS, taking up about 99% market share in total. By the end of the first quarter of the year 2017, Android has taken up more than 80% of the world-wide smart cellphone OS market share (see Table 1.1) [1]. Due to the increasing popularity, the security of this system attracts more and more attention. However, the Android platform has some inherent weaknesses such as its open nature and the lack of comprehensive measures for application markets protection. Those weaknesses have encouraged the cyber criminals to make Android a “priority” since it allows the existence of third-party apps markets along with the official market – Google Play. Even for Google Play itself, due to the huge amount of apps, no manual analysis is performed on the uploaded apps. Supported by Google’s machine learning systems, the newly
Table 1.1: Market Share of Different Smart Phone Platform

<table>
<thead>
<tr>
<th>Period</th>
<th>Android</th>
<th>iOS</th>
<th>Windows Phone</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016Q1</td>
<td>83.4%</td>
<td>15.4%</td>
<td>0.8%</td>
<td>0.4%</td>
</tr>
<tr>
<td>2016Q2</td>
<td>87.6%</td>
<td>11.7%</td>
<td>0.4%</td>
<td>0.3%</td>
</tr>
<tr>
<td>2016Q3</td>
<td>86.8%</td>
<td>12.5%</td>
<td>0.3%</td>
<td>0.4%</td>
</tr>
<tr>
<td>2016Q4</td>
<td>81.4%</td>
<td>18.2%</td>
<td>0.2%</td>
<td>0.2%</td>
</tr>
<tr>
<td>2017Q1</td>
<td>85.0%</td>
<td>14.7%</td>
<td>0.1%</td>
<td>0.1%</td>
</tr>
</tbody>
</table>

launched Play Protect, which can scan 50 billion apps every day, still lacks manual vulnerability analysis [5]. Furthermore, Android apps trust the end users with too much “permission” issues, which are heavy responsibilities to common users. After all, there are about 130 permissions that govern access to different resources.

For these reasons, more than 90% of malware on smart hand-held devices targets the Android system. The current situation is that there are about 8,400 new Android malware samples being reported every day. According to Kaspersky report, in 2016, they detected 8,526,221 malicious installation packages, 128,886 mobile banking Trojans and 261,214 mobile ransomware Trojans [8]. In the first quarter of 2017, 750,000 new Android malware apps were discovered [9].

Facing such an unprecedented amount of malware, it is urgent to develop efficient and scalable methods to facilitate malware analysis. However, most existing Android malware analyzing tools are based on signatures, i.e. rules describing unique characteristics of a malware sample. A generation of these signatures requires understanding of app functionality and complex, time-consuming analysis of its behavior. Consequently, the majority of malware strains (many of which are short lived, i.e., less than 24 hours) go undetected. Obviously, this is not efficient in the context of a large amount of malware. To solve this problem, we propose an alternative solution. Previous research studies show that apps’ development environment, especially various tools used in the process, tend to leave traces in the program structure [67]. It is also shown that those traces can be used to identify malware. Based on this
idea, in this thesis, we explore the traces of these tool artifacts to aid future malware analysis. A set of tools employed in the software development process is often referred to as a tool chain [4]. Particularly for the Android apps, this set of tools can help to optimize code size as well as power consumption to fit in the special needs of mobile devices [6]. To facilitate the development of software, some tools can be used to help apps developers pack their own libraries into the application, to protect their copyright. However, in practice, those tools can also be used by malware developers. For example, when the tools allow the developers to get access to the API headers, which is convenient in some cases, it may also potentially cause problems such as the Android OS’s absence of separation or isolation mechanism for third-party libraries like advertisement and analytic libraries [7]. One of the direct results is the malicious repacking tools for the purpose of malware injection, privacy collection and misusing the granted permission, etc. [10]. Under this circumstance, information about the tools used in apps development is very useful to the defenders. With this information, the analysts can achieve efficient malware incident triage which means they can have a quick assessment of a malware sample to determine how it was created and to evaluate its level of harm.

Obfuscators are frequently used tools mainly designed for the purpose of obfuscation and some other functionality. The initial goal of obfuscation is to transform original code to disguise its appearance and intent, and to protect it from reverse engineering and analysis. As a result, obfuscation is commonly used for legitimate purposes, i.e., software protection. In fact, the use of ProGuard – the most popular obfuscator – is encouraged in Android app development. Therefore, the use of obfuscators alone does not indicate the malicious nature of an app. However, as noted by Symantec, “Android malware is becoming stealthier. … malware authors started to obfuscate code bypass signature-based security software” [11]. Consequently, understanding whether an obfuscator is being used and what kind
of configuration is set is beneficial in facilitating malware triage and analysis. In the typical course of manual analysis, reverse engineers see patterns that can be attributed to the presence of obfuscation (and sometimes even specific tools). However, learning what represents a meaningful pattern requires years of experience. This raises two research questions. The first one is *Is it possible to identify these patterns automatically?* The reason for this question is, although many development tools, including obfuscators, tend to leave behind traces in the program structure, it still requires tedious manual analysis of binaries to recognize these traces [67]. The second question is *Do these traces have discriminatory power?* To be effective, these patterns have to remain stable among all binaries treated with a fixed tool, while varying the results from the same program obfuscated with different tools.

This thesis aims to address the questions raised above. To this end, we propose a novel Android obfuscator fingerprinting process, which identifies the obfuscation pattern of a specific obfuscator using the visualization method. We explore visual texture analysis of the images generated from the Android binary files. Furthermore, we want the classification task to be done automatically by computer instead of by a person manually. To address these problems, we employed a well-developed technique named image texture analysis [69]. After a survey of texture analysis methods, we decided to employ a statistical approach because of its efficiency, low computation cost and ease of use. Other popular texture analysis approaches like structural, model-based and transform methods will be described briefly in Section 2.4. By using the statistical method, we are able to extract first-order and second-order statistical features which are based on the spatial distribution of pixel values, to analyze the image texture. After that, information obtained in the previous steps can be synthesized into fingerprints of the obfuscators to judge which obfuscator has been used in the apps development. Then the experiments are performed on a large dataset generated from different Android obfuscation tools with various configuration
options. The results are promising and we achieve 73% accuracy in fingerprinting the obfuscators. The accuracy of binary classification of obfuscated and original apps is more than 86%.

The hypothesis of this study is that the use of the obfuscators on the Android apps leaves traces behind in the binary files. Those traces can be isolated from the specific obfuscated apps so that we can get app-independent and obfuscator-related features. Those features can be analyzed and extracted after those binary files are converted into images. Once those features are extracted, they can be further combined into different stable patterns to fingerprint each obfuscator.

Our main contribution can be listed as follows:

1. We propose an approach to fingerprinting the obfuscators used on Android apps based on visualization.

2. By using the statistical image analysis approaches, we are able to extract and test two types of statistical features – the first-order statistics and the second-order statistics. We confirmed the combination of the selected second-order statistics can be used to fingerprint each obfuscator.

3. A dataset of original apps as well as the apps handled by different obfuscators is established.

The thesis is organized as follows: Chapter 1 is the introduction; Chapter 2 gives the background of the problem; Chapter 3 describes the related work; Chapter 4 presents features we used; Chapter 5 discusses results obtained during experimental study; and Chapter 6 recaps the conclusion.
Chapter 2

Background

2.1 Android OS Overview

Since the first commercial version of Android 1.0 was released in 2008, this operating system has been developed by Google and the Open Handset Alliance continuously under Android Open Source Project (AOSP). As mentioned in Chapter 1, Android has become a leading mobile devices platform. Behind the achievement and development is the goal of Android’s owner, which is to “accelerate innovation in mobile and offer consumers a richer, less expensive, and better mobile experience”. This goal is also accurately reflected by the basic characteristics of the Android OS.

Firstly, Android is a truly open platform that can separate the hardware from the software it runs. This brings the developers and consumers a rich ecosystem to extend the system freely and even to include some third-party open source libraries under the business-friendly license: Apache/MIT [13]. Secondly, Android is initially designed for mobile devices, so the special requirements like battery and limited size of memory/speed are taken into account. Those requirements and considerations eventually make the whole Android OS look like a multiple-layer cake.

The base layer is the Linux kernel. The reason of such choice is that the designers
want to take advantage of Linux’s portability, security and its good management of memory, power and networking. On top of the Linux kernel, there is the native libraries layer containing libraries from the open source community which are mostly written in C/C++. The one above the native libraries layer is the application framework, which is the best-documented part, providing the developers with a universal, reusable software environment to deal with low-level details. The top layer is Applications. This layer faces the end users directly, bringing them the value of Android. Usually, an application is packed into an Application PacKage (APK) file. Basic structure of this package is shown in Figure 2.1. The classes.dex part is highlighted in red because it is the starting point of the visualization process. A .dex file is divided into several sections. Each section has a strict requirement of size and offset to guarantee its alignment which is an important property of .dex file. This property also implies that it is very easy to reverse engineer a regular .dex file. For instance,
a famous tool named Dex2jar can easily decompile a .dex back to a .jar. and then some other tools like JAD, can convert .jar to .java. This will surely lead to violation of copyright or even security problem. Consequently, some security measures have been taken to protect .dex files. Obfuscation is one of them (see Figure 2.2). Details of this issue will be further discussed later.

In addition to the classes.dex, the other components are still worthy of being mentioned since they are used in some related works [17, 25, 36]. AndroidManifest.xml, which is labeled in green, is an indispensable part of an APK file, where the metadata like package name is stored. “Permission” mechanics of Android is realized by this file. It defines one or more components such as Activities, Services, Broadcast Receivers or Content Providers, minimum and maximum version support, libraries to be linked etc. The different parts are glued together by AndroidManifest.xml, and it tells the system what to do with these top-level components. In an APK file, anything except code is called resources. Resources such as icons, images, string/numeric/color constants, UI layouts, menus, animation are contained in the folder res, and those resources will be compiled into the binary. Those that cannot be compiled into binary are contained in the folder assets.

Since the Android apps are written in Java, the three building tools – Apache Ant with Ivy [12], Apache Maven and Gradle – which are popular in developing Java project are also widely used in developing Android apps.

### 2.2 Obfuscators

Code obfuscation is gaining popularity. It is especially common on the Android platform, which employs Java programming language. Since Java byte-code executes in hardware-independent Java virtual machine, it retains most of the information of the original source code. As a result, it is easily decompiled and reverse engineered.
To prevent this, a corpus of obfuscators are designed. One type of obfuscators converts the code into an equivalent form that is hard to understand and reverse engineer. This type of obfuscator is also called code obfuscator including ProGuard, DexGuard, Allatori. In some cases, since these code obfuscators are based on code transformation, they are similar to the compiler optimizer. There is another type of obfuscators that is specifically designed for Android, which encrypts the classes.dex file. The encrypted classes.dex file will be decrypted in the memory at the run-time via DexClassLoader using reflection mechanism. Bangcle and APKProtect belong to this category. This type of obfuscators is also called the packers [83].

Collberg et al. [16] defined four types of code obfuscation: layout obfuscation that targets surface characteristics and includes methods such as source code formatting, variable renaming; data obfuscation that targets data structures and includes array/methods recording/splitting, change of variable encoding; control obfuscation, a more advanced transformation that aims to obscure the flow of the program control (e.g., redundant or junk code insertion, loops, statements reordering, code optimization) and preventive transformations to make known automatic deobfuscation techniques more difficult (e.g., reordering for-loop to run backwards, adding bogus data, inserting extra instructions to crash the deobfuscator).

In Figure 2.2, it is shown where the obfuscator can be used in the process of development. This figure shows the two types of obfuscators can be used either before or after dx, aiming at Java code and Dalvik file, respectively. All the obfuscators studied in this work, together with their functionality, are listed in Table 2.1.

Among those obfuscators, ProGuard is the most popular one supported by Ant and Gradle since it has been integrated into the Android Software Development Kit (SDK). Because it is free, open-source and contains many different options, its code is frequently used by other obfuscators’ developers, which makes it hard for us to distinguish it from some of the others. ProGuard also has a commercial edition.
Table 2.1: List of Obfuscators

<table>
<thead>
<tr>
<th>Functions &amp; Names</th>
<th>Obfuscate Names</th>
<th>Obfuscate Resources</th>
<th>Obfuscate Control Flow Obfuscation</th>
<th>Exception Obfuscation</th>
<th>Incremental Obfuscation</th>
<th>String Encryption</th>
<th>Custom Encryption</th>
<th>Remove Debug Info</th>
<th>Code Optimization</th>
<th>Watermarking</th>
</tr>
</thead>
<tbody>
<tr>
<td>DexGuard</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bangdle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProGuard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KlassMaster</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jshrinsk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alleatori</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- ✓: Available
- ✓: Non-available
the DexGuard, which is a strengthened version of ProGuard. Since they share most functionality, even code, it is very hard to tell them apart. As the extended commercial version of ProGuard, DexGuard not only focuses on code protection, but also has additional features like string and class encryption, obfuscation of the class and method names with non-ASCII characters. It is reported that the DexGuard-obfuscated samples are hard to reserve engineer [67].

Besides the ProGuard and DexGuard, Bangcle is the other one worthy of being mentioned. The provider of Bangcle distributes a Windows-based Graphic User Interface (GUI), so that the end users can upload their apps to be packed online. Once the work is done, Bangcle will send a confirm email to the users. This mechanism hinders us from incorporating Bangcle into script to execute automatically. APKProtect, as another GUI-based obfuscator shares the same problem.

Klassmaster, Jshrink and Allatori are Java obfuscators. Klassmaster can do the obfuscation of exception handling procedures in addition to traditional layout obfuscation. Jshrink primarily offers layout and control obfuscation. Allatori is a commercial Java obfuscator that includes a full range of obfuscation transformation.
2.3 Visualization

*Visualization* plays an important role in this work. Whenever we come to the topic of visualization, the famous proverb is always quoted: “A picture is worth a thousand words.” But the question is why we believe, at least in some cases, a picture is equal to or more important than a mass of words? Marty, the author of the book *Applied Security Visualization* [41], pointed out the reason lies in human’s instinct – we trust more what we see than what we hear. He further compared a human brain to a pattern-recognition tool which is good at detecting the change in size, color, shape, movement and texture. The point of view that the importance of visualization partly comes from human’s biological structure is widely accepted [57].

Besides, in the information explosion era, the importance of visualization also comes from the ever-growing amount of data generated from different areas. Most of the data are in textual form, such as dataset, documents, emails, URLs, etc. These data can be analyzed, stored and communicated by virtue of visualization very efficiently. In addition, we also have some extra reasons for applying visualization to the cyber security field. Firstly, a lot of intrusion detection systems are based on the analysis of event log data. Without using visualization, the huge amount of data cannot be probed efficiently. Efficiency brought by visualization is important to day-to-day regulatory work. Secondly, current cyber criminals have shifted from the traditional network-based attacks to the application layer attack, and the random attack has been developed into targeted attacks if necessary. These new situations call for in-depth analysis where visualization can play an important role.

Visualization applied in network/information security is surveyed by Shiravi et al. [42] and Liu et al. [43]. Both of them agree visualization can serve as an effective and efficient way of communicating information. It is further pointed out in [43], visualization can be applied to a variety of types of data like high-dimensional data, heterogeneous data, geographic data, narrative data and tables of counts, propor-
tions, and probabilities. In the Big Data era, data exploration and visualization systems also play a vital role in real time data changes and complex data processing results, as summarized by Raghav et al. [44].

Compared to the application of visualization to network/information security, direct usage of visualization to malware analysis is relatively rare. A good summary is presented by Wagner et al. [58], in which they pointed out the general process of using visualization in the malware analysis, as shown in Figure 2.3. From this figure, we can see, for most of the related work, visualization is used on the collected dataset to be analyzed and can bring us new knowledge hidden in the dataset leading to further actions. Since this branch is tightly related to current work, further discussion will be presented in Chapter 3.

2.4 Image Texture Analysis

The next thing is to analyze the results from visualization. In the current work, visualization is used to convert binary file into 2D images, which are actually rectangular surfaces containing various textures. Description of texture leads to basic features of the 2D images. Therefore, a method called Image Texture Analysis, which is developed and still used in other scientific areas like medical imaging, remote sensing, etc., is fit for current tasks.

Although image texture is important, there is no widely accepted definition for it.
Some qualitatively describe it as characteristic intensity (or colour) variations that typically originate from roughness of object surfaces [45]; some say it is analogous to the feeling when we touch some object with our finger [46]; some argue it is a description of images’ properties like smoothness, roughness, repetition of substructure, etc [73]. Among the different explanations, the idea used is that image texture is an attribute representing the spatial organization of the gray levels of the pixels in a region of a digital image, since it will lead to numeric description. Many numeric texture analysis approaches based on this idea have been invented, which can be broadly categorized into (1) structural method [73], (2) statistical method [73], (3) model-based method and (4) transform method [69]. For the structural technique, the fundamental texture elements, such as regularly spaced parallel lines, should be defined first. Upon those texture elements, a hierarchy of spatial arrangement can be set up. Then the choice of texture elements and the probability of the chosen texture elements, as a function of location, can be used to describe the image texture. Such a structural approach is good at giving symbolic descriptions of images fulfilling synthesis rather than analysis tasks.

The statistical method, as implied by the name, is to extract the statistical features out of an image, and use these statistics to describe the image texture. Usually, the statistics are performed on the gray scale level of each pixel contained in the image. We can perform the statistical calculation based on the frequency of particular gray level scale for each individual pixel, which is called first-order statistical method. This kind of statistical method does not take into account the correlation between the pixels. However, in the case of second-order statistical method, such a correlation between the pixels is considered by measuring the probability of finding a pair of grey-levels at selected distances and orientations over the entire image. Such a probability is stored in a matrix named co-occurrence matrix; that is why the method is also called gray-level co-occurrence matrix (GLCM). We can certainly extend this idea
to higher order statistical calculation with more variables involved, such as gray level run length and local binary patterns. But in most cases, extra effort such as deliberate cognition is needed [69]. For this reason, second-order statistics is the best trade-off between efficiency and effectiveness. Moreover, Materka [69] pointed out second-order statistics can achieve a higher discrimination rate than that of transform-based and structural methods, and [69] has also shown that GLCM can outperform wavelet packets (a transform-based technique) when applied to image texture classification.

The model-based method tries to interpret image texture in terms of fractal image model and stochastic model, respectively. The parameters of the model are used as textural feature descriptors. In practice, the stochastic model faces the problem of computational complexity. Although fractal model is good at describing some natural images, its lack of orientation selectivity prevents it from describing local image structures very well.

Transform methods is a collective name for a group of methods including Fourier [76], Gabor [77] and wavelet transforms [78]. The reason those three are grouped together is all of the methods make use of the spatial frequency properties of the pixel intensity variations to describe image texture. Fourier is less used in related work due to its lack of spatial localization. A descriptor of the images named GIST is frequently used in malware classification through visualization (see Table 3.1). The main idea of the method is to develop a low dimensional representation of the images like degree of openness, expansion, roughness etc. This method does not require any form of segmentation [70]. We can include Gabor filter or wavelet transform into the GIST dimensions [71].
Chapter 3

Related Work

3.1 Android Security

As introduced in the previous chapters, the spirit of open platform provides the users and the app developers with great convenience. However, from the angle of cybersecurity, this spirit causes the vulnerability of Android OS because it permits the existence of other third-party apps markets apart from the official market – Google Play. Compared to iOS, Android OS is less controllable at this point. The open platform also gives developers too much decision power which is usually not fully understood by the developers. This risk is likely to lead to privacy leakage. All these factors have added up to a severe Android OS security situation.

This situation has surely attracted attention from academia and industry. Various defense and malware analysis methods have been proposed. For the analysis issue, the main methods can be roughly categorized into 1) static, 2) dynamic, and 3) hybrid analysis.

- **Static Analysis:** a technique to figure out the features of a software without executing it, and the features can be further turned into the signature of the app. Usually those signatures are collected into a database to be used in a
signature-based classification with efficiency and simplicity. Because of this, static analysis is adopted by the existing commercial anti-malware companies to assess the security of Android apps, to detect apps clones and to automate test cases. Furthermore, this classification can be performed in an off-device cloud-based manner [15], which means a great saving of mobile phone’s battery and memory.

However, static analysis is likely to be circumvented by code obfuscation, and the signature database should be updated frequently; otherwise, it will result in malicious apps being missed.

- **Dynamic Analysis:** In this analysis process, apps are executed either in a sandbox or on an emulator, and their behavior is monitored. Under the controlled environment, more information of the apps, such as network traffic, use of memory and the user-machine interaction, can be obtained compared to static analysis. Obfuscation, which once blocked static analysis, can be easily overcome here. But the obvious shortcoming is that this approach is more expensive and less scalable compared to static analysis.

- **Hybrid analysis:** a combination of the previous two which takes advantage of their strengths. Usually, the static analysis is applied first to detect potential security issues, and then dynamic techniques are performed to improve their precision by eliminating the false warnings.

The general pros and cons of each analysis method are listed above. For each of them, there is a lot of work concerning different, more specific directions.

Take the static analysis as an example: the interaction between different components concerns some of the work. In the work of Lu et al., a tool named CHEX is established to analyze the Android component hijacking vulnerabilities [17]. Another component-based static analysis tool – SCanDroid [18] – has been invented
to extract security specifications from manifests that accompany the apps, and to check whether data flows through those apps are consistent with those specifications. Some of the static analysis methods concentrate on the dalvik-bytecode to check privacy leakage and/or telephony services misuses [19, 20], to figure out features from loops used in the code [21]. Data/Control flow analysis is also very popular in static analysis [22, 23]. Apart from these, there are some other tools available for other purposes, such as AppoScopy [24] for capturing syntactic or semantic patterns, AndroSimilar [25] used to generate statistical features so that zero-day variants of the already known malware can be detected in time.

As to dynamic analysis, some research work focuses on the abnormal profile caused by malware [26, 27]. For example, a Denial of Service (DoS) attack may be characterised by extremely high CPU usage, memory utilization statistics, network traffic, etc. Some studies examine malicious behavior directly [28, 29, 30], including sensitive data leakage, voice call without users’ consent, etc. Most of the dynamic analyses are performed on a Virtual Machine (VM) or an emulator; however, some perform dynamic analysis out of a VM or an emulator on purpose, for fear the VM or emulator itself may be susceptible [31].

A typical example of using hybrid analysis is the work done by Sounthiraraj et al [32], in which they use static analysis to identify potentially vulnerable apps to SSL/TLS man-in-the-middle attack first, and then use dynamic analysis to confirm the vulnerability by performing automatic UI exploration.

In addition to the general analyzing methods discussed above, permission-based analysis, as an important branch of static analysis, deserves special attention, for it plays an increasingly important role in defending against malware due to its efficiency. In addition, this method is designed upon Android peculiarities. Permissions are the mechanism by which app developers disclose how their apps will interact with users’ devices and personal information on devices. In this particular branch, most of the
researchers are interested in what kind of permissions are commonly used by legitimate apps, and use this as an important feature in classification [33, 34, 35]. For example, in the work of Feizollah et al., they listed and compared the top 10 frequently used permissions both in benign and malicious apps [35]. They found, in terms of occurrence frequency, 98% of apps from both sides have the intention of access to INTERNET, which is ranked as top one permission. Behind the top one, 89% of benign apps require permission of ACCESS_NETWORK_STATE while 89% of malicious apps need permission of READ_PHONE_STATE. It should be noted that READ_PHONE_STATE allows apps to get access to sensitive information like phone number, current cellular network information, the status of any ongoing calls, and a list of any phone accounts registered on the device. They also found that malicious apps tend to request more permissions than benign ones.

Some studies focus on the permissions themselves, trying to figure out the risk of each of them [36], or the appended permission in the updated apps since Android does not verify them [37], or even the permission residue data after an app is uninstalled [38]. Naturally, the end users’ awareness of the permissions they granted is also under investigation [39, 40].

### 3.2 Obfuscation and Visualization

Until now, the research on Android apps’ obfuscation has mainly focused on (1) detection of obfuscated apps, (2) evaluation of efficacy of obfuscation to mask the maliciousness against anti-virus detection, (3) identification of different obfuscators used in apps. To detect repackaged apps at a large scale, Zhang et al. proposed a system named ViewDroid based on the information of software birthmark, which is a unique characteristic that an app inherently possesses and can be used to identify the app [47]. Wang et al. asserted that even if obfuscation is performed, there are
still some important elements that cannot be changed; otherwise the correct execution of the app cannot be guaranteed [48]. They also believe, for most obfuscators, the apps’ code structure cannot be changed for the same reason. So, they calculated some statistical characteristics of the app code based on the two assertions to perform Android malicious code detection. Although, no direct discussion is included in the work of Kumar et al. [49], the detection of obfuscation in Java malware by virtue of metrics like word count, identifier length, etc., is still very interesting since the main part of Android apps is written in Java. In the study performed by Rastogi et al., a system named DroidChameleon is set up to evaluate resistance of some commercial Android anti-malware softwares against various common obfuscation techniques [50]. Similar research is performed by Mercaldo and Visaggio [51]. A framework is developed to apply obfuscation on some known malware’s smali code on purpose, then the two versions (obfuscated and non-obfuscated) are sent to an anti-virus system to check the rate of detection. This way the authors evaluated the efficacy of the applied obfuscation algorithm. Work on the identification of obfuscators is rare. Wang and Rountev investigated the topic by extracting featured strings out of the data section in the .dex file and used them to perform classification [52].

As described in Chapter 2, visualization has been extensively studied and widely applied in many fields for exploratory analysis. In the mobile domain in general, and malware analysis area in particular, visual exploration has seen very limited use. Those few studies that were published focus on one of three objectives: individual analysis of malware samples to gain new insights of their behavior; a broader malware detection that includes bulk visualization for samples’ comparison and classification; and malware systematization to understand similarities and common behavior.

For the first category – Malware Analysis – the work done by Wüchner et al. [53] and Donahue [54] can serve as representatives. In [53], DAVAST, a data-centric system level visualization utility is set up and applied in individual malware analysis
by using a system data flow graph approach. As shown in Figure 3.1, the system can visualize system activities as data flow graphs with nodes presenting operating system entities such as processes, files, and sockets; edges denote data flows between the nodes. Pattern matching of the graphs can tell the differences between benign and malicious behavior. In [54], a tree-based navigation visual interactive Hex editor is used. This mechanism can help user to pinpoint the underlying sections of a Portable Execution (PE) file quickly, thus enhancing the efficiency of realizing individual malware analysis and detection.

![Figure 3.1: The Flow of Fingerprinting Process](image)

The majority of work in the field falls into second category – Malware Comparison. This category can be further divided into feature-based and image-based approaches. Shabtai et al. [55] present a network behavior-based anomaly detection system for identification of malicious attacks, masquerading application, and injection of malicious code in the repackaged apps. This system has been tested on a broad range of apps and their different versions. Gove et al. [56] has developed a scalable visualization tool named SEEM for simultaneously comparing a large corpus of malware across multiple sets of attributes, so that the shared or reused attributes can be detected to reduce analysts’ workload.

Our visual exploration approach, although aiming to identify anomalies and pat-
terns, focuses primarily on internal binary structure to identify obfuscation presence and thus falls under the second sub-category – feature-based malware comparison. Related work in the particular direction is summarized in Table 3.1.

Similar work has been done by Liu and Wang [61]. In this work, the authors focus on Windows malware and the experiment is performed on a 2T database, using a selective ensemble learning method based on bagging and K-means. They compared the gray-scale image-based method to the n-gram and API call feature extraction method. The result shows the image-based method has an overall advantage over the other two no matter what classifier is used.

In the work of Ahmadi et al. [63], the image analysis method is further divided into two types. The first type uses features that describe the textures in an image such as the Haralick features [65], and the second type uses the Local Binary Patterns features. They also discussed 12 other feature extraction mechanisms, including 1-gram, metadata, entropy, string, different sections contained in a PE file, data packed, frequency of registers use, miscellaneous, operation code, frequencies of a set of symbols used in disassembled file and API. Their results show that, in terms of the importance of the features, the metadata about the size of the file and the address of the first bytes sequence is the most efficient one and the first type of the image analysis method is ranked as the sixth in the 14 mechanisms.

Study of similarity between the apps is an interesting topic in this domain and falls into the third category – Malware Similarity. However, visualization used in this direction is rare. In the work of Han et al. [64], image matrices are generated, using the opcode sequences from malware samples, then the similarities are evaluated based on the RGB-colored pixels in the image matrices. In the study of Paturi et al. [66], a method named Normalized Compression Distance (NCD) is employed to enumerate code similarity between malicious Android apps and visualize their clusters.
In Section 2.4, we mentioned a couple of texture feature extraction methods that have been used in malware detection and classification. Related work in this direction is categorized in Table 3.1, according to which texture feature extraction approach has been used. In the work of Han et al. [79], GLCM feature and efficient incremental

<table>
<thead>
<tr>
<th>Related Work</th>
<th>Texture Analysis Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Han et al. [79]</td>
<td>GLCM</td>
</tr>
<tr>
<td>Conti et al. [90]</td>
<td>First-Order Statistics</td>
</tr>
<tr>
<td>K. Kancherla and S. Mukkamala [80, 84]</td>
<td>Wavelet, Gabor, Intensity-based (average intensity, variance, mode, skewness, kurtosis)</td>
</tr>
<tr>
<td>Nataraj et al. [81, 82, 60], Kirat et al. [86], Yang and Wen [87]</td>
<td>GIST based on Gabor filter bank</td>
</tr>
<tr>
<td>A. Makandar and A. Patrot [85]</td>
<td>Gabor wavelet with GIST</td>
</tr>
</tbody>
</table>

clustering algorithm based on the P-Stable locality sensitive hashing are used to obtain the depth of annotations to malicious code families. The binary mapping metaphor is proposed in the work of Conti [90]. He is the only one who uses the first-order statistics to a variety of binary files including .txt, .jpg and .zip files. In the series work of Nataraj and his group [81, 82, 60, 86], GIST is used in different manners for different problem. In [81], a given image is divided into 20 sub-bands and each band provides 16 features so that 320 GIST features are obtained. Their experiment shows that although accuracy of GIST classification is a little smaller than dynamic features, the efficiency of static GIST analysis is 4,000 times faster than dynamic analysis. In [82], a system named SARVAM is established for content-based Search And RetrieVAL of Malware. Features used in this system are still GIST based on bank of Gabor filter. Since May 2012, this system has received close to 212,000 samples of which nearly 45% were possible variants of already existing malware from their database. In [60], they were not only concerned with gray-scale images
generated from the samples but also with the signals as well. Apart from Windows OS, other OS such as Linux and Android are under investigation. In the work of Kirat [86], which is also from the Nataraj’s group, uses the same image analysis method to build a fast and precise malware detection framework named SigMal. In the series work of Kancherla [80, 84], both Gabor and wavelet features are employed. In [84], they also included intensity based features like skewness and kurtosis. The studies of Kancherla and Mukkamala only focus on Windows PE file analysis. In the work of Yang and Wen [87], they use the wavelet transform based GIST to classify the Windows and Android malware and achieve an overall accuracy of 95.42%.

From the summary of Table 3.1, it is interesting to see Gabor filter or Gabor filter based GIST is more popular than GLCM in this particular area. This does not mean Gabor filter is more powerful than GLCM (see Chapter 13 in the book Handbook of Texture Analysis [45]). One reason is a series of influential studies is from the same group ([81, 82, 60, 86] are from Nataraj’s group), using the same Gabor filter based GIST method. So, in this particular research area, Gabor filter and Gabor based GIST are better tested in more scenarios than GLCM. Under this circumstance, we prefer GLCM so that this work can be done in a different way to find something new. One basic difference between this work and the related work is we have different purpose, although similar image analysis method is employed. Our ultimate purpose is to find out how the obfuscators have been used on the application while previous works were designed to detect or classify the malware. This subtle difference gives rise to profound impact on the final results, which will be discussed later.
Chapter 4

The Proposed Solution

The obfuscation techniques, which are usually realized by the tools – the obfuscators – were initially designed for good purposes such as protection of developers’ copyright from reverse engineering. Gradually, those techniques and tools are used for malicious purpose. Therefore, information about the usage of the obfuscators provides useful reference in malware detection.

To extract the information effectively, the solution proposed is to analyze the grayscale image generated from the .dex file contained in an Android app. Ultimately, after such analysis, usage of different obfuscator can be related to different group of features for the images. This group of features can be used as the fingerprint of each obfuscator to facilitate further malware detection.

4.1 Procedure

The whole system is composed of five relatively independent subsystems: (1) the generation of the apps, (2) Preprocessing, (3) Images generation process, (4) Features extraction and (5) Classification.

The complete flow of the proposed solution is presented in Figure 4.1. In the APK generation process, APK files are compiled from the source code. In this stage,
compilation can be done without using any obfuscators, which will lead to original apps. Most of the obfuscators under investigation, except for Bangcle, are involved in the compilation process as shown in Figure 2.2. Bangcle needs to be used on the non-obfuscated apps. Preprocessing is for extracting classes.dex out of the given APK and renaming the .dex file properly. Image generation, as implied by its name, is the procedure to convert the binary .dex files into images for further investigation. In the step of feature extraction, four first-order statistics as well as the Haralick vector are calculated [73]. Classification is performed on a dataset whose entries are the obfuscated and original apps. This procedure is designed to test the efficacy of the features obtained in the previous step. Another important outcome of classification is each of the obfuscators can be uniquely related to certain group of features. Such a group of features can be used to label the obfuscators so that once an obfuscated APK is given, we are able to tell which obfuscator has been used.

### 4.2 Preprocessing

The whole preprocessing stage includes unpacking the APK file, extracting binary classes.dex files out of the apps and renaming those classes.dex files so that each of them will have a unique name. In essence, an APK file is a compressed file. However, the regular decompression tools cannot recognize the file with the extension .apk. The very first step is to convert the extension into .zip. In this step, we read in the names of the apps and store them into a string array. Each string is parsed so that useful information like name and version number can be taken out and stored.
for future usage. Then those ZIP files can be expanded in a temporary folder one by one. Each time only the classes.dex is copied to certain directory in the form of “new name”.dex. In this step, we will add information about the obfuscators so that after this step, we can read from the “new name” which obfuscator and what kind of configuration of such obfuscator have been used. By doing so, each of the sample is uniquely labeled in an informative manner. This eventually facilitates a “supervised” classification. For example, an original app named “An.stop_10.apk” may be turned into “An.stop_10.proguardNoObfOpt1.release.dex”. This new name tells us ProGuard is used (proguard) with the first level of optimization (Opt1) and obfuscation is switched off (NoObf). Obviously, in this example, this new name means that, in this specific investigation, we want to know how the first level of optimization alone will affect the final result. Also in this example, “10” is the version number. If in the future, we encounter a newer version such as “11”, this old one will be discarded. The reason is that we found, in most of the cases, the differences between the new and the old version are trivial. The apps which only differ in version will give us similar images. Including those similar images in the same dataset is misleading, since a set of similar features can be extracted from those images, which makes us believe a stable profile of obfuscator has been identified, while in fact the stable profile is only a reflection of repeated samples. Therefore, selecting and keeping the newest version app is an important task of preprocessing.

4.3 Image Generation

We have different ways of generating images from the binary .dex file. One is proposed by Jain et al. [67]. In that work, a color image is generated based on the internal structure of the .dex file. The procedure is briefly described here. First, the information contained in the header file is read out (see Section 2.1) so that the
remaining part can be determined by using the address pointers and offsets. Then for different contents, different colors are allocated.

The other way of image generation results in gray-scale images. The basic process is shown in Figure 4.2. From this figure, we can see the binary digits in .dex file are grouped into an 8 bit vector, then each entry in the vector is further mapped into the decimal gray-scale values of each pixel; thus an image is formed based on these pixels. Suppose the grouped 8 binary digits are 00000000; it is converted into decimal number 0, used to denote “black”, and the other extreme is the group of 11111111 being mapped to decimal number 255, corresponding to a “white” pixel. That is why we say the range of gray-scale value of each pixel is 0∼255.

![Figure 4.2: Generation of Gray Scale Images](image)

For the purpose of comparison, two types of images generated from the same app, “a2dp” are The main difference between these two images is the height. In spite of the fact that the original file size is the same (514 KB), the resulting color image is taller. The reason for this lies in the way the images are generated. Unlike the gray-scale image, the color image has incorporated information that does not carry any functional value, such as debugging information and padding. It is well-known that a code compiled in the debug format usually has bigger size than that of release format due to the extra debugging information. Padding bytes are commonly used in a .dex file to acquire the required alignment.

Despite these differences between representations the gray-scale image contains the same information as color image. In our experiments we calculated first- and second-
order statistics for both types of the images. These experiments produced similar results. We thus employed gray-scale visualization approach for the image generation.

4.4 Features Extraction

As discussed in the Section 2.4. Feature extraction is done by virtue of image texture analysis in this case. In the past, a variety of image texture analysis approaches have been developed. Among them, statistical analysis method has been used and fully tested in the scientific area such as remote sensing, medicine, agriculture, etc. It has been proved to be a powerful image analysis method. However, this method is seldom used in the mobile security field. For these reasons, in this work, we use two types of statistical features: first-order and second-order statistics.
The First Order Statistics

Once the gray-scale image is generated, we can assume the image is a function \( X(x, y) \), of two space variables \( x \) and \( y \), where \( x \) ranges from 0 to \( N - 1 \) while \( y = 0 \ldots M - 1 \). \( N \) is related to the width of the image and \( M \) is related to the height of the image. The function \( X(x, y) \) ranges in 0~255, which is the gray-scale value of each pixel. First-order statistics are simple to calculate because they are based on the histogram of gray-scale value. The histogram can be obtained through

\[
h(i) = \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} \delta(X(x, y), i). \tag{4.1}
\]

where \( \delta(i, j) \) is the well-known Kronecker delta function. When \( i = j \), the value of this function is 1; otherwise 0. Such a gray-scale value \( X(x, y) \) can be regarded as the random value as used in Equation 4.3. The whole gray-scale image is in fact a 2D matrix of the random values. First-order statistics are based on the distribution of the random values contained in the 2D matrix, which can be defined as \( h(i) \) being divided by the total number of the pixels in the image, like:

\[
p(i) = h(i)/NM. \tag{4.2}
\]

Among basic statistical tests that are available for analysis of data, we choose Shannon entropy, Arithmetic mean, Chi square and Hamming weight. These tests were initially explored by Conti in their analysis of statistical characteristics of primitive structures in binary files [90]. As the study showed these statistical measurements were sufficient to differentiate between different binary file types. These selected first-order statistics are defined as follows:

- **Shannon entropy** \((H)\): an established technique for measuring uncertainty
is calculated as follows:

\[ H = -\sum_{i=0}^{n-1} p(X_i) \log_{10} p(X_i) \]  \hspace{1cm} (4.3)

where \( X \) is a random variable ranging from 0 to 255 to denote gray level of the pixels. \( p(X_i) \) is the probability density function defined in Equation 4.2. Usually, compressed or encrypted files have higher entropy than those who have repeating values, for the former have higher randomness than the latter.

- **Arithmetic mean**: the sum of byte values in a given fragment divided by the fragment size.

\[ H = \frac{\sum_{x=0}^{N-1} \sum_{y=0}^{M-1} X(x, y)}{NM}, \]  \hspace{1cm} (4.4)

where \( N \) and \( M \) are the width and height of the image, respectively.

- **Chi square \((X^2)\)**: an effective means of measuring randomness and is sensitive to difference in random, pseudo random, and compressed data. Its mathematical definition is

\[ X^2 = \sum_{i=0}^{n-1} \frac{(\text{observed} - \text{expected})^2}{\text{expected}}. \]  \hspace{1cm} (4.5)

In this formula, ‘observed’ means the observed distribution of byte values while ‘expected’ is only a uniform random distribution value used here. Therefore it can tell us to what extent a random variable deviates from a uniform random distribution.

- **Hamming weight**: a counting of the numbers of non-zero symbols in a given
Obvious horizontal structures can be observed in the gray-scale images. To study those horizontal structures and to investigate how those quantities can depict those structures effectively, those images are divided horizontally into stripes to see how those quantities vary along the height of the images. After that, a graph about statistics versus percentage of the height is drawn. The other way is to calculate the statistics on the whole image without any division. One of the advantages of the second approach is that its result is rotation and translation invariant, and we will show in Chapter 5 that values obtained in this way are used in constructing 3D plots to visually determine if a pair of obfuscators are distinguishable or not.

**The Second-Order Statistics**

These first-order statistics can be easily computed, given the image. However, they only give a coarse description of the random values without considering the correlation of the pixels. That is to say, if the random values of different pixels are not independent of each other, the valuable relationship between the pixels that cannot be reflected by the first-order statistics due to this limitation. Fortunately, the deeper insight into relationships between individual pixels that can be described by the co-occurrence matrix, which can represent the joint probability distributions of pairs of pixels. Therefore, the effectiveness of second-order statistics is much higher than the first-order statistics in image discrimination, like Julesz once commented: “no texture pair can be discriminated if they agree in their second-order statistics” \[75\].

The basis of those second-order statistics is so-called co-occurrence matrix, upon which different version of statistical methods have been developed. Among the dif-

\[
\sum_{x=0}^{N-1} \sum_{y=0}^{M-1} (1 - \delta(X(x,y), 0)).
\]
ferred methods, Haralick Vector is selected and used in this work. The Haralick Vector was once used as the “standard” image texture analysis approach during the 1970s. Since then, this set of features has been widely and successfully used in many different scientific research areas [45]. To understand the principles of the second-order statistics, the two key points – co-occurrence matrix and Haralick Vector – are explained as follows.

4.4.0.1 Co-occurrence Matrix

Same as the establishment of the first-order statistics, a given 2D image can be seen as a 2D matrix with each entry denoting the discrete gray-scale level value of each pixels. The gray-scale value of each pixel can be defined as a function of position of the pixel like \( f(x, y) = i \), with \( i = 0 \ldots G \), where \( i \) is the gray-scale level value of the pixel located at the position of \((x, y)\); \( G \) is the maximum value of gray-scale levels. In practice, \( G = 255 \). To facilitate the understanding of this scheme, we use a simplified image that has only four gray-scale level in the upper left of Figure 4.4, in which \( G = 3 \). For example, for the entry at the first column and first row, its gray-level value is 0, so we have \( f(1, 1) = 0 \) (labeled in red). Apparently, \( f(4, 4) = 3 \), according to the convention, means on the fourth column and fourth row (labeled in green), the gray-level value is 3. In essence, the co-occurrence matrix is a second-order histogram, which is denoted by \( h_{d,\theta}(i, j) \) [69, 72]. The definition of \( h_{d,\theta}(i, j) \) is based on the 2D matrix image. One thing that needs to be noted here is that the value of \( i \) or \( j \) does not denote the position as in the image array but the value of the gray-level value. In our example, it is easy to see \( 0 \leq i \leq 3 \) and \( 0 \leq j \leq 3 \). This implies the co-occurrence matrix to be established is a \( 4 \times 4 \) square matrix, as we see in the upper right of Figure 4.4. The symbol \((i, j)\) means, to set up the co-occurrence matrix, we need the gray-scale value information of a pair of pixels. Suppose the position of one pixel is \((x_1, y_1)\), the other is \((x_2, y_2)\), then the pixels to
be paired should fulfill the condition

\[(x_2, y_2) = (x_1, y_1) + (d \cos \theta, d \sin \theta), \quad (4.7)\]

where \(d\) is the distance between the two pixels to be paired, and usually \(d = 1\) or 2, depending on computational requirement. \(\theta\) is the angle of the line between the two pixels with respect to horizontal direction. It can be 0°, 45°, 90° and 135° as demonstrated in Figure 4.5. For the future reference, it is a good opportunity to make a convention according to this figure. Since 0° is for the horizontal direction, the features extracted this way are labeled with \(H\). Features from the 90° are labeled by \(V\), because they are from the vertical direction. The angle 45° is the diagonal direction, which means the features from this direction are labeled by \(D\). 135° is called the secondary-diagonal direction. Naturally, the features from this direction are labeled by \(SD\). For example, if a given feature is named ‘H13’, that means this feature is extracted from the horizontal direction, and it is the 13th component of the Haralick Vector.

Each entry in a co-occurrence matrix is the \textit{number of the times} of the occurrence such that

\[f(x_1, y_1) = i \text{ and } f(x_2, y_2) = j, \quad (4.8)\]

where the pair of coordinates \((x_1, y_1)\) and \((x_2, y_2)\) fulfill the condition described by Equation 4.7.

For example, if we choose 0° as our observing angle and let \(d = 1\), we can see \(f(0, 0) = 0\) while \(f(0, 1) = 0\) is a pair fulfilling all the conditions. The other pair that can be found in the image is \(f(0, 1) = 0\) and \(f(1, 1) = 0\) is the other pair. So \#(0, 0) = 2. If we further take into the symmetry in this case that \(i = j = 0\), this number need to be doubled, hence the result \#(0, 0) = 4. Under the same
Figure 4.4: The spatial co-occurrence calculations [73]

Figure 4.5: The four directions of adjacency for calculating the co-occurrence matrix, where “X” is the pixel under investigation, with eight nearest-neighbor pixels labeled in the framework to describe pixel connectivity [73].
condition of $d$ and $\theta$, $\#(0, 1) = \#(1, 0) = 2$. All these have been reflected in the matrix in bottom left of Figure 4.4. We can also check the bottom right co-occurrence matrix is established in the direction of $90^\circ$ with $d = 1$.

Once the co-occurrence matrix $h_{d,\theta}(i,j)$ is set up, we can achieve point probability matrix $p_{d,\theta}(i,j)$ by dividing the co-occurrence matrix with the total number of neighboring pixels $R(d, \theta)$ in the gray-scale image. $R(d, \theta)$ is defined as the number of pair of the pixels that can be found in the image fulfilling the following condition:

$$|x_1 - x_2| = d \cos \theta, \text{ and } |y_1 - y_2| = d \sin \theta. \quad (4.9)$$

All the following Haralick features are calculated based on this probability matrix $p_{d,\theta}(i,j)$. Before we list and discuss all the features to be used, the notations should be defined in advance.

- $p(i,j)$: $(i,j)$th entry in the co-occurrence matrix, $= h_{d,\theta}(i,j)/R$
- $p_x(i)$: $i$th entry in the marginal-probability matrix obtained by summing the rows of $p(i,j) = \sum_{j=1}^{N_g} p_{d,\theta}(i,j)$.
- $N_g$: Number of distinct gray levels in the quantified image.
- $p_{x+y}(k): \sum_{i=1}^{N_g} \sum_{j=1}^{N_g} p(i,j)$ with $i + j = k$ and $k = 2, 3, ..., 2N_g$.
- $p_{x-y}(k): \sum_{i=1}^{N_g} \sum_{j=1}^{N_g} p(i,j)$ with $|i - j| = k$ and $k = 0, 1, ..., N_g - 1$.
- $\mu_x$, $\mu_y$ and $\sigma_x$, $\sigma_y$: the mean and standard deviations of the row and column sums of the matrix.
- $HX$ and $HY$: entropies of $p_x$ and $p_y$.
- $HXY$: $- \sum_i \sum_j p(i,j) \log(p(i,j))$.
- $HXY1$: $- \sum_i \sum_j p(i,j) \log[p_x(i)p_y(j)]$. 

36
\[ HXY 2 = - \sum_i \sum_j p_x(i)p_y(j) \log[p_x(i)p_y(j)]. \]

### 4.4.0.2 Haralick Vector

Once the relevant notation is defined, we can further define 13 Haralick features and use them as a vector, which is called Haralick Vector. Mathematical definition and their names are listed in the Table 4.1. Among the 13 components, the components

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angular Second Moment (energy, Uniformity)</td>
<td>( f_1 = \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} p_i j^2 )</td>
</tr>
<tr>
<td>Contrast (Inertia)</td>
<td>( f_2 = \sum_{n=0}^{N_y-1} n^2 \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} p_{ij} ) with (</td>
</tr>
<tr>
<td>Correlation</td>
<td>( f_3 = \sum_i \sum_j \frac{(i - \mu_x)(j - \mu_y)}{\sigma_x \sigma_y} p_{ij} )</td>
</tr>
<tr>
<td>Sum of Squares: Variance</td>
<td>( f_4 = \sum_i \sum_j (i - \mu)^2 p(i, j) )</td>
</tr>
<tr>
<td>Inverse Difference Moment (Homogeneity)</td>
<td>( f_5 = \sum_i \sum_j \frac{1}{1+(i-j)^2} p(i, j) )</td>
</tr>
<tr>
<td>Sum Average</td>
<td>( f_6 = \sum_{i=2}^{2N_y} i p_{x+y}(i) )</td>
</tr>
<tr>
<td>Sum Variance</td>
<td>( f_7 = \sum_{i=2}^{2N_y} (i - f_5)^2 p_{x+y}(i) )</td>
</tr>
<tr>
<td>Sum Entropy</td>
<td>( f_8 = - \sum_{i=2}^{2N_y} p_{x+y}(i) )</td>
</tr>
<tr>
<td>Entropy</td>
<td>( f_9 = - \sum_i \sum_j p(i, j) \log(p(i, j)) )</td>
</tr>
<tr>
<td>Difference Entropy</td>
<td>( f_{10} = - \sum_{i=0}^{N_y-1} p_{x-y}(i) \log[p_{x-y}(i)] )</td>
</tr>
<tr>
<td>Difference Variance</td>
<td>( f_{11} = \text{variance of } p_{x-y} )</td>
</tr>
<tr>
<td>Information Measure of Correlation 1</td>
<td>( f_{12} = \frac{HXY - HXY^1}{\max HX, HY} )</td>
</tr>
<tr>
<td>Information Measure of Correlation 2</td>
<td>( f_{13} = (1 - \exp[-2.0(HXY 2 - HXY)])^{1/2} )</td>
</tr>
</tbody>
</table>

like homogeneity, contrast, entropy and energy are sensitive to the choice of the direction. The homogeneity and entropy supplies the indication on the dominant values of the main diagonal based on the distribution of \( p(i, j) \). As an example, it is easy to see from the formula of homogeneity, when \( i = j \), if the \( p(i, j) \) is a large value, the whole item will have a large value, which means that it is more possible to find the neighboring pixel in the certain direction of the same gray-scale value, hence an obvious visual homogeneity. Some other properties of an image can
also be reflected by some other Haralick Vector components. One of the example is the energy supplies the information on the randomness of the spatial distribution. Since those image texture properties are usually not independent with each other, this implies some of the Haralick Vector components are more or less related to each other, and it is not necessary to include all the features in practice. Thus a Principal Component Analysis (PCA) procedure is indispensable. Since Haralick features can be expressed as a vector, if there is more than one vector, it is natural for us to define “distance” between a pair of vectors. One of the straightforward definitions can be:

\[ d = \sum_{i=1}^{13} (x_i - y_i)^2, \]

(4.10)

where \( x_i \) and \( y_i \) are the components of two vectors. This distance, which is actually the square of Euclidean, can roughly tell us how two vectors are close to each other in a high dimensional space, which reflects the degree of similarity between two images.

4.5 Classification

The classification is conducted on the dataset whose entries are from the same group of original APK files and dealt by different obfuscators. The reason we design the dataset in this way is that we want to see, when the target app is fixed, what kind of difference will be brought by different obfuscators. Features established on such differences will be mostly dependent on the obfuscators and independent from the specific original app used.

Many classification algorithms have been developed. According to the survey by Alba [74], in the particular Android malware detection area, the following classification algorithms are most effective for malware analysis. They are also commonly applied in mobile malware detection field. They are listed and briefly explained as
below:

- **Bagging**: also known as bootstrap aggregating, is an ensemble algorithm used to improve stability and to reduce variance by employing a bag of base learner [91].

- **KNN**: is short for the k-nearest neighbor, as known as a lazy learner, which is a non-parametric algorithm labelling the new data according to the vote of its k neighbors without learning anything from the training data and simply using the training data itself for classification [92].

- **SVM**: is short for support vector machines. Given a set of training data, SVM can classify the new data into different categories. If proper kernel function is employed, SVM can be used in high dimensional feature spaces [93].

- **Regression**: a classification algorithm borrowed from the field of statistics, which can predict future values based on the given dataset. The algorithm can be regularized to avoid overfitting [94].

In this chapter, the solution proposed in this thesis is introduced. The whole solution is based on the assumption that the use of obfuscators can be featured by the first- and second-statistical quantities obtained from the gray-scale images generated from the binary container of an Android app. The effectiveness of the group of features used to label each obfuscator is tested and confirmed by the classification performed on certain dataset.
Chapter 5

Evaluation Study

As discussed in Chapter 4, two kinds of statistics are calculated from the images to study and extract the trace of obfuscators left in the binary .dex file. In this Chapter, the effectiveness of those two types of statistics will be tested and discussed. After that, classification performed on different dataset will further show us how the features can be combined to form a unique fingerprint for each of the obfuscators.

5.1 Experiment Setup

The apps compilation process is performed on a machine installed with a Ubuntu 16.04 LTS operating system with a 4 core 3.10 GHz Intel i5-2400 CPU and 7.7 GB memory. A set of C-shell scripts are developed for this purpose. The procedure of preprocessing, image generation, texture analysis and classification are done on a Windows 10 operating system with 4 core 1.80 GHz AMD A10-8700 Radeon R6 CPU and 8 GB installed memory. All the work is organized and run by a series of Windows Power-shell scripts.

For the classification, the packages used are from the famous machine learning software – weka [95]. In our experiments, we are using the following implementations
of classification algorithms, K-NN is called Kstar [92] while SVM is SMO using a sequential minimal optimization algorithm invented by John Platt [93]. As to the Bagging, the basic classifier is fast decision tree learner with pruning [91]. For the Regression, class is binarized and one regression is built for each class value. This classification via regression method is based on models trees, which are a type of decision trees with linear regression as their leaves [94].

5.2 Dataset

To establish a dataset, we collect Android apps’ source code from F-droid market [88]. In total, 3956 unique apps (i.e. unique MD5 hash value) were downloaded. Those samples can be compiled by either Gradle or Ant. As mentioned at the end of Section 2.1, to make sure all the apps are generated from the same environment, only Ant is used in this work. It is easy to find out that one of typical property of apps developed by Ant is, because in the root directory, there must be a file named build.xml used for setting some basic configuration. With this criteria, we managed to select 2027 apps in total that could be developed by Ant. After running the C-shell script to compile those files, we found only 1399 APKs can be compiled successfully without any obfuscator involved. 628 apps failed to be compiled for a variety of reasons. Some of the failure are caused by the version collision, some are due to the failure of downloading some external files.

Once the non-obfuscated apps were obtained, we proceeded to use those obfuscators displayed in Table 2.1 on the 1399 apps one by one. According to the requirement of development environment, the obfuscators can be roughly divided into two types. One is used in Windows-based GUI system, the other is presented in the form of .jar files and can be conveniently incorporated in C-shell script in Linux. Bangcle and ApkProtect belong to the first type. To be obfuscated by Bangcle, an app
should be submitted through service GUI to an online remote server, where the app is obfuscated. All these procedures should be done manually. Later on, we will see in Section 5.3.1, it is easy to distinguish Bangcle from the others. For these reasons, we only generate 50 samples for testing.

Allatori, DexGuard, Jshrink, Klassmaster and ProGuard belong to the second type, which can be run through CLI. A separated C-shell script is written for each of them to take into account different configuration and user-defined options of the obfuscators.

For ProGuard and DexGuard, this information is stored in `proguard-project.txt`/`dexguard-project.txt` and `local.properties`, respectively. As to the rest, most of the information is in `custom_rules.xml`, and extra information for Allatori and Klassmaster is stored in `allatori.xml` and `myZKMScript.txt`, respectively.

Eventually, with the default option, we successfully generated 1371 apps obfuscated by Allatori, 1387 for DexGuard, 1030 for Jshrink, 1032 for KlassMaster and 1063 for ProGuard. Not all the selected 1399 apps are handled by those obfuscators successfully. Several reasons led to most of the failure, such as incompatible version of the Android plugin, SDK version collision, improper environment parameters set in the configuration files, etc.

All the successfully obfuscated apps are generated from the default configuration of each tool. By default, Allatori, DexGuard, and Klassmaster do layout obfuscation, data obfuscation and control obfuscation. Jshrink and ProGuard by default only do layout obfuscation. ProGuard, as the most widely used obfuscator, is studied in more detail by generating more apps with different combinations of options. Therefore, we also have 1046 apps with optimization but without obfuscation, 835 with obfuscation but without optimization and 99 more with only shrinking. The dataset is summarized in Table 5.1.

Repeated experiments show images obtained from the visualization are actually
synthesized results under the influence of the apps themselves as well as the obfuscator used. As an example, the gray images originated from the same app –

![Gray Images](image)

**Figure 5.1:** Comparison between the Gray Images originated from the app “com.cepmuvakkit.times”: (a) Non-obfuscated, (b) KlassMaster, (c) Allatori, (d) DexGuard, (e) Jshrink, (f) ProGuard

“com.cepmuvakkit.times” – are displayed in Figure 5.1. Among those images, the first one, Figure 5.1a is the one without using any obfuscation. The rest of the images, from Figure 5.1d to 5.1f, are from the obfuscated ones.

For this particular sample, since Figure 5.1a is identical to Figure 5.1b, we can conclude the KlassMaster does nothing to this sample. Meanwhile, comparison between
the Figure 5.1a and the other images shows all the other obfuscators changed the original app in different ways. Especially for Jshrink, it shrinks the app’s size as implied by its name. This is related to one of the tasks of this study – to figure out how each obfuscator changes the app. The way they change the app can be used as the feature to label the obfuscators, provided it is stable to different samples.

The app-related information contained in the results prevents us from figuring out pure obfuscator-related features. In order to compare different behavior between the obfuscators, it is good to use all the obfuscators on a fixed sample, then we can tell the difference between the apps generated by using different obfuscators. In other words, by repeating such comparisons on a group of fixed samples, we can figure out which app-independent features belong to each obfuscators. To this end, we need to select a group of apps that can be obfuscated by different tools. Such a group is found from the overlapped part of the dataset established for each obfuscator mentioned in the previous paragraph. Eventually, we selected 325 apps that can be obfuscated by the 5 tools. After the old versions from the same app family are deleted, 601 original apps are selected.

5.3 Evaluation of Extracted Features

After the dataset of apps is set up, classes.dex of each app is extracted and renamed in the proper way. This is done automatically by running a Window PowerShell script. This script can also invoke Python scripts to convert those .dex files into gray-scale images.

5.3.1 First-Order Statistical Features

In this section, the experimental results of the first-order statistical features obtained from the gray-scale images are discussed. As mentioned before, we tested the first-
order statistics in two ways. The first way is to divide an image horizontally into small segments so that the image can be scanned to disclose valuable details by using the first-order statistics. The number of the segmentation ranges from 50 to 1000, depending on the behavior of the curves. If the division is enough to disclose a stable pattern, we just stop there for the sake of efficiency. Our experiments show, for most of the apps, 500 segmentation is enough. The motivation for this method is from the observation of the images. After large amount of gray-scale images are generated under different circumstances, one thing in common for those images we noticed is that most of them contain obvious horizontal stripes structure varying in location and contrast. Consequently, to check the efficacy of those first-order statistics, the natural way is to use them to scan through the image in slices. It turned out to be an efficient analyzing method to disclose the internal structures of those images.

As an example, the image obtained from the app “dendroid” is presented in Figure 5.2. All the four first-order statistics are calculated on those segments. The result is plotted in graphs where x-axis stands for percentage of the image’s height and y-axis is the statistical value obtained (see Figure 5.2). The second approach is the calculation performed on the whole image, so that three of the statistics can be selected as the coordinates to decide its position in a 3D graph. Obviously, statistics obtained in this way disclose less detailed information compared to the previous one, however, results obtained this way are rotational symmetric.

Figure 5.2 depicts how the internal structure of a gray image can be disclosed by the first-order statistical features. The gray image in Figure 5.2a is set horizontally on purpose for a convenient comparison. In Figure 5.2b, all the x-axes denote the percentage of the height of image starting from the top to the bottom. All the y-axes denote the four different first-order statistics, respectively. In this figure, we can see all the first-order statistics show the same five distinct plateau structures start at about 65 and end at around 95 on x-axis, corresponding to the five light strips in
Figure 5.2: Comparison between the Gray Image and its First-Order Statistics of the app “dendroid”: (a) Original Gray Tone Image, (b) Result of Four First-Order Statistics

the gray image. The reason is when a pixel’s decimal gray-level value is 0, it is a black one; and the decimal gray-scale value of 255 means a white pixel in the image. Therefore, all the statistics reach higher value in the brighter strip area than the
In addition to the Shannon Value, for the rest of the first-order statistics described in Figure 5.2b, we can also see spikes located on the top and left side of the plateau structures. The explanation again lies in the gray-scale value of the pixels. Mean Value shows the two obvious spikes which can be visually detected from the gray image, while Chi Square gives more spikes, which means this quantity is more sensitive than Mean Value. This is understandable, as the definition of Chi Square contains the square function which magnify the black-white contrast.

In short, Figure 5.2 shows those first-order statistics are effective in describing internal structure of the images. Such an effectiveness is not unique to the example in the Figure 5.2. Repeated experiments show it can be found by using any other samples. This explains why they can be used effectively in classifying malware family, since malware samples from the same family can give the similar images [90]. However, malware family classification is not the ultimate purpose of this work; instead, we want to find out which obfuscator has been used on the samples. Only a good description of the images is not enough for us; we also need a good description of an obfuscator’s behavior which is supposed to leave behind traceable features in the images. For this reason, further investigation is performed on a group of images derived from the same obfuscator. It shows that the first-order statistics can easily reveal the usage of Bangcle. No matter what app has been obfuscated by Bangcle, the graphs always show the same profile. However, for other obfuscators such as ProGuard, it is not easy to capture stable profile from the curves. (See Figure 5.3).

In this figure, two apps named “am.ed.exportcontacts” and “se.johanhil.clipboard” are obfuscated by Bangcle and ProGuard. It is clear that the effect of ProGuard varies from one app to another, which is in contrast to that of Bangcle. The reason is, as a packer, the Bangcle always relocate most of the content in the .dex file to a resource file, leaving behind a .dex file of fixed length in the structure of small
Figure 5.3: Shannon Entropy is calculated on different apps. (a) For “am.ed.exportcontacts”, Bangcle (Left), ProGuard (Right) (b) For “se.johanhil.clipboard”, Bangcle (Left), ProGuard (Right)

variation. Thus, this profile is only unique to Bangcle. However, the results confirm it is evident that the curve drawn by calculating the first-order statistics not only depend on what obfuscator has been used but also on what app has been handled, i.e., the behavior of curves is app-dependent.

To find out app-independent features, the small dataset based on 325 samples mentioned earlier is utilized. For each obfuscator, default configuration is used to guarantee consistency. Three out of the four first-order statistics can be calculated and used as coordinates of a 3D graph. We tested different combinations and did not notice any essential difference. As an example, Chi Square, Mean Value and Shannon
Entropy, are calculated for each sample and then plotted in a 3D graph as shown in Figure 5.4.

Figure 5.4: 3D graph for 5 obfuscators used on the same collection of 325 apps, using 3 first-order statistical quantities as coordinates.

Figure 5.4 implies the potency of using clustering method. It shows the dots of different obfuscators have the tendency to be clustered in different groups. For example, this graph shows dots of Allatori and Jshrink can be separated easily. However, if we compare the obfuscators in pairs, the comparison shows that some of them are far away from each other, while some of them are too similar to be distinguished.

To further clarify the effectiveness of the first-order statistics quantitatively, a classification is performed on the same dataset used to plot the 3D graphs, containing 1625 entities obfuscated by 5 different obfuscators. The results are summarized in Table 5.2.
Table 5.2: Results of Classification Using the First-Order Statistics

<table>
<thead>
<tr>
<th>Algorithm:</th>
<th>Bagging</th>
<th>Regression</th>
<th>K-NN</th>
<th>SVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy:</td>
<td>62.58%</td>
<td>60.00%</td>
<td>23.32%</td>
<td>47.88%</td>
</tr>
</tbody>
</table>

In this table, even for the highest accuracy generated by Bagging, it is still less than 63%. In the worst case, the accuracy from k-NN is no more than 24%. This means, only for certain obfuscators such as Bangcle, the first-order statistics are useful, but they are not enough for other obfuscators like ProGuard, Klassmaster, etc.

Table 5.3: Confusion Matrix. a → Allatori; b → Jshrink; c → Klassmaster; d → ProGuard; e → DexGuard

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>Bagging</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>220</td>
<td>14</td>
<td>5</td>
<td>25</td>
<td>61</td>
</tr>
<tr>
<td>b</td>
<td>31</td>
<td>176</td>
<td>24</td>
<td>78</td>
<td>16</td>
</tr>
<tr>
<td>c</td>
<td>15</td>
<td>20</td>
<td>198</td>
<td>26</td>
<td>66</td>
</tr>
<tr>
<td>d</td>
<td>22</td>
<td>41</td>
<td>30</td>
<td>214</td>
<td>18</td>
</tr>
<tr>
<td>e</td>
<td>17</td>
<td>10</td>
<td>70</td>
<td>19</td>
<td>209</td>
</tr>
<tr>
<td>(b)</td>
<td>Regression</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>214</td>
<td>7</td>
<td>11</td>
<td>37</td>
<td>56</td>
</tr>
<tr>
<td>b</td>
<td>29</td>
<td>166</td>
<td>26</td>
<td>95</td>
<td>9</td>
</tr>
<tr>
<td>c</td>
<td>9</td>
<td>25</td>
<td>190</td>
<td>36</td>
<td>65</td>
</tr>
<tr>
<td>d</td>
<td>28</td>
<td>36</td>
<td>24</td>
<td>216</td>
<td>21</td>
</tr>
<tr>
<td>e</td>
<td>20</td>
<td>8</td>
<td>80</td>
<td>28</td>
<td>189</td>
</tr>
<tr>
<td>(c)</td>
<td>Kstar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>151</td>
<td>29</td>
<td>49</td>
<td>49</td>
<td>47</td>
</tr>
<tr>
<td>b</td>
<td>92</td>
<td>66</td>
<td>71</td>
<td>42</td>
<td>54</td>
</tr>
<tr>
<td>c</td>
<td>85</td>
<td>48</td>
<td>67</td>
<td>54</td>
<td>71</td>
</tr>
<tr>
<td>d</td>
<td>139</td>
<td>33</td>
<td>51</td>
<td>56</td>
<td>46</td>
</tr>
<tr>
<td>e</td>
<td>140</td>
<td>35</td>
<td>72</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>(d)</td>
<td>SMO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>221</td>
<td>2</td>
<td>26</td>
<td>16</td>
<td>60</td>
</tr>
<tr>
<td>b</td>
<td>74</td>
<td>148</td>
<td>55</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>c</td>
<td>24</td>
<td>16</td>
<td>277</td>
<td>1</td>
<td>57</td>
</tr>
<tr>
<td>d</td>
<td>85</td>
<td>18</td>
<td>145</td>
<td>17</td>
<td>60</td>
</tr>
<tr>
<td>e</td>
<td>23</td>
<td>3</td>
<td>134</td>
<td>0</td>
<td>165</td>
</tr>
</tbody>
</table>

The confusion matrices related to Table 5.2 are shown in Table 5.3. Those confusion matrices further indicate in detail why the Kstar fails and which obfuscator is more likely to be confused to another due to the similarity.

5.3.2 Second-Order Statistical Features

Haralick vector, which is used as a representation of second-order statistical features, is an efficient statistical approach to describe image texture. A grayscale image generated from an app named “monakhv” is shown in Figure 5.5 and the corresponding
Haralick features are listed in Table 5.4.

![Gray Scale Image of “monakhv”](image)

Figure 5.5: Gray Scale Image of “monakhv”.

<table>
<thead>
<tr>
<th></th>
<th>0°</th>
<th>45°</th>
<th>90°</th>
<th>135°</th>
</tr>
</thead>
<tbody>
<tr>
<td>0°</td>
<td>1.59472e-01</td>
<td>1.20405e-01</td>
<td>1.20380e-01</td>
<td>1.20239e-01</td>
</tr>
<tr>
<td>45°</td>
<td>5.13845e+00</td>
<td>1.24506e+01</td>
<td>1.24553e+01</td>
<td>1.24919e+01</td>
</tr>
<tr>
<td>90°</td>
<td>9.08526e-01</td>
<td>7.78173e-01</td>
<td>7.78099e-01</td>
<td>7.77438e-01</td>
</tr>
<tr>
<td>135°</td>
<td>2.80871e+01</td>
<td>2.80638e+01</td>
<td>2.80651e+01</td>
<td>2.80639e+01</td>
</tr>
<tr>
<td></td>
<td>9.53298e-01</td>
<td>8.11906e-01</td>
<td>8.11913e-01</td>
<td>8.11495e-01</td>
</tr>
<tr>
<td></td>
<td>1.62506e+01</td>
<td>1.62493e+01</td>
<td>1.62496e+01</td>
<td>1.62493e+01</td>
</tr>
<tr>
<td></td>
<td>1.07210e+02</td>
<td>9.98048e+01</td>
<td>9.98054e+01</td>
<td>9.97638e+01</td>
</tr>
<tr>
<td></td>
<td>3.22172e+00</td>
<td>3.58547e+00</td>
<td>3.58623e+00</td>
<td>3.58744e+00</td>
</tr>
<tr>
<td></td>
<td>3.38706e+00</td>
<td>4.08070e+00</td>
<td>4.08066e+00</td>
<td>4.08186e+00</td>
</tr>
<tr>
<td></td>
<td>3.899647e-02</td>
<td>2.576184e-02</td>
<td>2.576488e-02</td>
<td>2.573827e-02</td>
</tr>
<tr>
<td></td>
<td>4.34044e-01</td>
<td>1.45525e+00</td>
<td>1.45525e+00</td>
<td>1.45745e+00</td>
</tr>
<tr>
<td></td>
<td>-9.02657e-01</td>
<td>-6.77463e-01</td>
<td>-6.77486e-01</td>
<td>-6.77085e-01</td>
</tr>
<tr>
<td></td>
<td>9.98097e-01</td>
<td>9.92326e+01</td>
<td>9.92327e+01</td>
<td>9.92308e+01</td>
</tr>
</tbody>
</table>

It is visible from the table that the values along 45° and 135° are almost identical up to the second decimal place. This numerical feature shows the symmetry between the two angles, which is confirmed from Figure 5.5. In contrast, the values along 0° and 90° are quite different which reflects the vague horizontal stripes in the image.
5.3.3 Difference Matrix

As discussed before, some of the obfuscators are similar to each other while some are quite different from each other. This phenomenon can be reflected by the “distance” between the clusters plotted in the 3D graph. In Chapter 4, we mentioned such a “distance” can be calculated via Haralick vector (see Equation 4.10). If the obfuscators are compared in pairs to get the values of distance, we can obtain a matrix containing those values, which is named Difference Matrix. An example is shown in Table 5.5 (only the integer part is kept for brevity).

Table 5.5: Difference Matrix Based on the Gray Scale Image of “a2dp”. 1 → Allatori with the option of obfuscating class names; 2 → Klassmaster obfuscating class names; 3 → Apkprotect; 4 → Bangle; 5 → Original without using any obfuscators; 6 → Using ProGuard with all three functions on, they are obfuscation without using dictionary, shrinking and optimization; 7 → ProGuard without Optimization

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>153</td>
<td>342</td>
<td>7633</td>
<td>66</td>
<td>9838</td>
<td>8839</td>
</tr>
<tr>
<td>2</td>
<td>153</td>
<td>0</td>
<td>474</td>
<td>7772</td>
<td>155</td>
<td>9982</td>
<td>8983</td>
</tr>
<tr>
<td>3</td>
<td>342</td>
<td>474</td>
<td>0</td>
<td>7302</td>
<td>319</td>
<td>9519</td>
<td>8510</td>
</tr>
<tr>
<td>4</td>
<td>7633</td>
<td>7772</td>
<td>7302</td>
<td>0</td>
<td>7618</td>
<td>3559</td>
<td>1837</td>
</tr>
<tr>
<td>5</td>
<td>66</td>
<td>155</td>
<td>319</td>
<td>7618</td>
<td>0</td>
<td>9830</td>
<td>8828</td>
</tr>
<tr>
<td>6</td>
<td>9838</td>
<td>9982</td>
<td>9519</td>
<td>3559</td>
<td>9830</td>
<td>0</td>
<td>1757</td>
</tr>
<tr>
<td>7</td>
<td>8839</td>
<td>8983</td>
<td>8510</td>
<td>1837</td>
<td>8828</td>
<td>1757</td>
<td>0</td>
</tr>
</tbody>
</table>

The big number represents significant behavioral difference between the obfuscators, while the small number shows similarity. In order to tell to what extent an obfuscator has changed the original non-obfuscated app, the original non-obfuscated app is also included in the comparison. The smallest number – 66 – gives the distance between the original app and its Allatori obfuscated version. This shows that among all the obfuscators, Allatori brings the least change to the sample. On the other side, ProGuard with all the options has changed most of the original file. The maximum value, 9982 can be seen for Klassmaster and ProGuard (with all options). That means it is easy to distinguish between these two obfuscators. It is worth noting
that the same conclusion is drawn from the color image generated from “a2dp” though the numbers are different, which verifies the assumption that the two different visualization methods are actually supportive to each other, as long as they are from the same .dex file.

A similar calculation of difference matrix is performed on the 325 samples obfuscated with default options. The result is shown in Figure 5.6. We also counted the maximum and minimum values occurring within the pair of obfuscators in the difference matrix. Occurrence of maximum value is presented by black column and occurrence of minimum value is denoted by gray column in the image. The two columns of the first pair A-D in this image show the occurrence of maximum value between Allatori and DexGuard which is less than 10, while the occurrence of minimum value is more than 110. It shows from all the samples, Jshrink is quite different from Klassmaster, while Allatori is close to DexGuard.

![Distance Between the Obfuscators](image)

Figure 5.6: The impact of five different obfuscators on the same 325 samples.

Actually, difference matrix not only can be used to investigate the overall behavior of those obfuscators, it also detect the alteration caused by different options of the same obfuscator. Table 5.6 is used to show a certain app obfuscated by ProGuard with different dictionaries. As we know, ProGuard provides the options to perform obfuscation with or without user-defined dictionaries. The result shows when the
Table 5.6: Difference Matrix Based on Color Image of “a2dp” With Different Obfuscation Mode, with SimChi means simplified Chinese, Eng is short for English, default is obfuscation without any user defined dictionary

<table>
<thead>
<tr>
<th></th>
<th>Eng</th>
<th>Arabic</th>
<th>SimChi</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eng</td>
<td>0</td>
<td>137</td>
<td>199</td>
<td>25</td>
</tr>
<tr>
<td>Arabic</td>
<td>137</td>
<td>0</td>
<td>63</td>
<td>116</td>
</tr>
<tr>
<td>SimChi</td>
<td>199</td>
<td>63</td>
<td>0</td>
<td>179</td>
</tr>
<tr>
<td>Default</td>
<td>25</td>
<td>116</td>
<td>179</td>
<td>0</td>
</tr>
</tbody>
</table>

user is using an English dictionary, the result is close to that of default mode, since in this table, the minimum value appears between English and Default. This is understandable since the default language is English. Meanwhile, since the words in the default English dictionary are different from those of the user-defined English dictionary, we still see a slight difference reflected by the minimum “distance” – 25. When the Chinese or Arabic dictionary is used, the ASCII code designed for Latin alphabetic is not enough. Both of the languages need special Unicode sets, which are encoded in different sets of digital numbers. That means using those dictionaries to obfuscate class names or method names, etc., the final binary files will be quite different from the result of the ASCII code.

5.4 Classification

Again, to further check the effectiveness of the Haralick features, and to investigate what kind of useful information can be determined from the dataset, classification is done.

In the following experiments, all 52 Haralick features are not used. The required features are selected according to their information gain value calculated by Weka. The initial number of features in the group is 12, which are derived from the four angles (see Figure 4.5), with 3 features from each angle. This group includes $H_3$, $H_{12}$, $H_{13}$, $V_3$, $V_{10}$, $V_{12}$, $D_3$, $D_{12}$, $D_{13}$, $SD_3$, $SD_{12}$ and $SD_{13}$. The explanation
about these abbreviation of the features together with their information gain values are presented in Table 5.7.

Table 5.7: The Feature Group

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Explanation of the Abbreviation</th>
<th>Inf. Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>V3</td>
<td>The third component of Haralick Vector from the vertical direction</td>
<td>0.2426</td>
</tr>
<tr>
<td>SD3</td>
<td>The third component of Haralick Vector from the secondary-diagonal direction</td>
<td>0.2228</td>
</tr>
<tr>
<td>D3</td>
<td>The third component of Haralick Vector from the diagonal direction</td>
<td>0.2174</td>
</tr>
<tr>
<td>H3</td>
<td>The third component of Haralick Vector from the horizontal direction</td>
<td>0.1972</td>
</tr>
<tr>
<td>V12</td>
<td>The twelfth component of Haralick Vector from the vertical direction</td>
<td>0.167</td>
</tr>
<tr>
<td>D12</td>
<td>The twelfth component of Haralick Vector from the diagonal direction</td>
<td>0.1306</td>
</tr>
<tr>
<td>SD12</td>
<td>The twelfth component of Haralick Vector from the secondary-diagonal direction</td>
<td>0.1298</td>
</tr>
<tr>
<td>D13</td>
<td>The thirteenth component of Haralick Vector from the diagonal direction</td>
<td>0.1238</td>
</tr>
<tr>
<td>H12</td>
<td>The twelfth component of Haralick Vector from the horizontal direction</td>
<td>0.1207</td>
</tr>
<tr>
<td>SD13</td>
<td>The thirteenth component of Haralick Vector from the secondary-diagonal direction</td>
<td>0.0949</td>
</tr>
<tr>
<td>H13</td>
<td>The thirteenth component of Haralick Vector from the horizontal direction</td>
<td>0.0715</td>
</tr>
<tr>
<td>V10</td>
<td>The tenth component of Haralick Vector from the vertical direction</td>
<td>0.0706</td>
</tr>
</tbody>
</table>

First of all, a binary classification is performed on a dataset containing 601 non-obfuscated apps and 661 obfuscated apps. Except for 80 apps, all the other non-obfuscated apps are generated from the source code not being used to generate the obfuscated apps, since in the real world not all the samples are from the same original code. The results are shown in Table 5.8. These results indicate that the non-obfuscated apps and the obfuscated apps are quite distinct. The differences can be well depicted by the selected Haralick features so that even the smallest accuracy value of the classification is above 83%.

Table 5.8: A Binary Classification on Obfuscated and Non-obfuscated Apps

<table>
<thead>
<tr>
<th>Algorithm:</th>
<th>Bagging</th>
<th>K-NN</th>
<th>SVM</th>
<th>Regression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy:</td>
<td>85.02%</td>
<td>87.00%</td>
<td>83.12%</td>
<td>85.34%</td>
</tr>
<tr>
<td>Precision:</td>
<td>85.10%</td>
<td>87.05%</td>
<td>83.30%</td>
<td>85.30%</td>
</tr>
</tbody>
</table>

The second classification is performed on the dataset containing $325 \times 5 = 1625$ samples in total. Those samples are from 5 different obfuscators; each of them contributes 325 samples. The classification is not only performed on the whole dataset, but also on different partial dataset generated from the complete one, to check the similarities between the obfuscators.
Considering DexGuard is a commercial edition of ProGuard, we believe they share a lot in common such as functionality and code. Therefore, we perform experiments on the dataset without DexGuard, dataset with DexGuard and ProGuard being merged together, and even the dataset without both of them. Besides, as implied in Figure 5.6, Allatori and DexGuard also share a lot in common. To confirm this observation, dataset without Allatori and DexGuard is also studied.

Figure 5.7 presents the classification results. The x-axis and y-axis denote number of features and accuracy, respectively. In this figure, \textbullet{-}---\textbullet{-}WithAll denotes the whole dataset, \textbullet{-}---\textbullet{-}WithoutDex is the dataset without DexGuard; \textbullet{-}---\textbullet{-}MergeDexPro means in the dataset data from DexGuard and ProGuard have been merged; \textbullet{-}---\textbullet{-}WithoutDex&Pro stands for the dataset without both DexGuard and ProGuard; \textbullet{-}---\textbullet{-}WithoutDex&All is for the dataset without both DexGuard and Allatori. The experiments have been performed iteratively. In each iteration, the feature with the least information gain value is selected and eliminated one by one. The results show that the accuracy obtained in all the cases is less than 73%, which is very low compared to the similar work done on malware classification. The reason is, compared to the tangible malware families, our purpose of pursuing obfuscator’s characteristic behavior is more abstract. Those characteristic behavior cannot become an observable independent existence without applying the obfuscators on the apps. Therefore, it is impossible to figure out completely app-independent features.

As can be seen, with the decreasing number of features used, the accuracy increases slowly to the maximum value and then drops to minimum gradually. As expected, after removing DexGuard data or merging DexGuard into ProGuard, the overall performance of classification is enhanced. It is understandable that without both of them, the result is even better since they are similar to each other. Classification results from Bagging and SVM show that removing DexGuard data is better than merging it into ProGuard, while K-NN shows there is no big difference between the
Figure 5.7: Classification Upon the 325 Samples Dealt By 5 Obfuscators: (a) Bagging, (b) K-NN, (c) SVM, (d) Regression
two schemes. Regression is superior to other methods in all the different cases in terms of the accuracy.

With complete features, removing both Allatori and DexGuard has the same effect as that of eliminating both ProGuard and DexGuard, which confirms the conclusion of Figure 5.6. Moreover, the dataset with both Allatori and DexGuard is more sensitive to the number of features used for classification. Also, after removing the Allatori and DexGuard, there are still some tools that are similar to ProGuard. This reflects the fact that ProGuard is a successful free obfuscator and has been inherited by other obfuscators. The other group is formed by K-NN and Regression. Both of them clearly support that Allatori is similar to DexGuard and the similarity degree even surpass that between ProGuard and DexGuard by showing higher accuracy from the former dataset. Despite this small divergence, both of the two groups confirm the conclusion drawn from the Figure 5.6. However, we also see that Figure 5.6 fails to describe the similarity between ProGuard and DexGuard. This indicates that the method based on the calculation of distance is only a rough approximation which is less accurate than the classification techniques. The reason is implied in the definition of distance which is the Equation 4.10. From there, we can see the summation of the square of the Haralick Vector components blur out some useful information.

The previous experiments, once again, shows the potential of fingerprinting the obfuscators. However, to turn the possibility into reality, we still need to point out by what group of the features and to what extent one obfuscator can be fingerprinted. To this end, a series of binary classifications has been performed, aiming at each of the obfuscators. Structure of the dataset used in each of the classification is the same – 325 samples handled from the target obfuscator to be fingerprinted plus 650 samples from the other obfuscators and the non-obfuscated ones. The results are summarized in Table 5.9.
Table 5.9: Fingerprinting the Obfuscators

<table>
<thead>
<tr>
<th>Obfuscator</th>
<th>Feature Group</th>
<th>Accuracy</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jshrink</td>
<td>SD3, V3, D3, V12, H3, SD12</td>
<td>78.77%</td>
<td>78.19%</td>
</tr>
<tr>
<td>Klassmaster</td>
<td>V3, SD12, D12, SD3, H12, SD13</td>
<td>73.33%</td>
<td>72.23%</td>
</tr>
<tr>
<td>ProGuard</td>
<td>D3, SD3, V3, H3, SD13, D13, H13, V12</td>
<td>80.72%</td>
<td>80.30%</td>
</tr>
<tr>
<td>DexGuard</td>
<td>H12, SD3, V12, D12, SD12, V3, D3</td>
<td>75.28%</td>
<td>74.22%</td>
</tr>
<tr>
<td>Allatori</td>
<td>H3, V3, SD3, D3, SD12, V12</td>
<td>80.92%</td>
<td>80.68%</td>
</tr>
</tbody>
</table>

In the series of classification, the starting feature group contains all the 12 features mentioned before. Then those features are taken away one by one according to its information gain value. After those procedures, the features left in the group can reach the highest accuracy value are regarded as the fingerprint of the obfuscator to be labeled and summarized in the table. The features selected in the group are ranked from left to right with decreasing information gain value.

To use these fingerprints in the detection of obfuscators, the given apps should be handled according to the process described in Figure 4.1 and classified by the fingerprints generated from our repository. If an given sample cannot be labeled by any of the studied obfuscators, it will be labeled as an unknown sample to be treated and analyzed.
Chapter 6

Discussion and Conclusion

In this thesis, we have presented a visualization method to fingerprint the usage of obfuscators on Android binary files. Our investigation shows that obfuscators will leave behind certain pattern of traces, forming their unique marks. We have also found out those tools’ fingerprints can be extracted from the binary .dex file contained in the Android app.

Visualization method has been used to convert this binary file into an image, from which statistical features can be analyzed and extracted. Two types of images, the color and gray-scale images, can be generated. Despite the difference between the two types of images in appearance, our preliminary experiments showed their equivalence in analyzing .dex files. So, the majority of experimental work was performed on gray-scale images.

First-order and second-order statistical features have been calculated and studied. Compared to second-order statistics, the first-order statistics are easier to get and use. They can be used to disclose the internal structures of the .dex file and potential features. We have two approaches of using the first-order statistics. The first approach is to evenly divide the image into stripes and then calculate those first-order statistics on those segments. Results obtained in this way can be expressed in
graphs. By this means, the first-order statistics are able to distinguish Bangcle from other obfuscators easily. Meanwhile, further investigation has shown that first-order statistics used in this way are not enough to identify complex behavior of other obfuscators. First-order statistics are further investigated on a small group of apps, in which each sample of app has been handled by five different obfuscators. This time, first-order statistics are calculated on the whole images. Three of the four quantities are selected as coordinates of a 3D graph. By doing so, we are able to confirm, for some obfuscators, the generated dots tend to cluster in different areas. This implies first-order statistics obtained in this way can be potentially used in clustering. To confirm the potency of the first-order statistics, a classification is performed on this dataset. The low value of accuracy shows the first-order statistics are not enough to classify complicated obfuscators such as ProGuard, Allatori, etc.

Second-order statistics – the Haralick Vector – have been calculated for they are good at analyzing 2D image texture, generating essential features to distinguish those images. In the experiments based on the second-order statistics, we observed that the proposed difference matrix technique is useful to tell the degree of similarity between the obfuscators, and it can even tell the subtle change caused by different configurations of the same obfuscator. A binary classification has also been performed on a dataset composed of obfuscated and non-obfuscated apps. The highest accuracy value, 87%, indicates the efficiency of this method in distinguishing the obfuscated apps from the original apps. Direct classification toward the obfuscators has been performed on a small dataset composed of the samples obfuscated by five different obfuscators. Although the highest accuracy is only 73%, our preliminary results still disclose some valuable clue related to the obfuscators’ behavior such as the similarity between DexGuard, ProGuard and Allatori. To show the potential of fingerprinting the obfuscators, a series of binary classifications have been performed for different obfuscators on different dataset. Those datasets have the same structure – composed
of 325 samples handled by the target obfuscator to be fingerprinted and the 650 samples either obfuscated by other obfuscators or the non-obfuscated apps. The result shows, with properly selected algorithm and the group of features, those obfuscators can be correctly fingerprinted at the accuracy of more than 72% at least.

Our future work involves investigation in these directions. Firstly, to make full use of first-order statistics, instead of dividing the image into stripes, we need to divide it into grids. Calculation will be performed on each of the grid. We believe that this will give rise to more detailed information. The first-order statistics will be combined with second-order statistics to form a bigger bag of features. Furthermore, a larger dataset should be established so that all the features can be put to a stricter test. In the future, there will be more new obfuscators to be developed. Those new objects need to be included in the future investigation. Last but not the least, this technique can be further extended to fingerprint specific compression algorithms used on a particular file.
Bibliography


Vita

Candidate’s full name: Ye Ning
University attended (with dates and degrees obtained): University of New Brunswick, 2018, Master of Computer Science
Publications:
Conference Presentations: