Unmasking Android Obfuscation Tools Using Spatial Analysis

Ratinder Kaur, Ye Ning, Hugo Gonzalez, Natalia Stakhanova
Faculty of Computer Science
University of New Brunswick
Fredericton, Canada
Email: {rkaur1, ye.ning, hugo.gonzalez, natalia.stakhanova}@unb.ca

***Abstract*—Android has become one of the most popular mobile device operating systems. Indeed, its security issues have attracted a lot of attention. One of the major security concerns is the 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 novel fingerprinting approach for Android obfuscation tools based on spatial analysis. We investigate first-order and second-order statistical features to analyze spatial distribution of pixels representing Android binary images. With our approach, we are able to achieve nearly 90% accuracy in fingerprinting several obfuscation tools with specific configuration options.**

I. Introduction

Smartphones have become a pervasive part of everyday life. By 2017, Android had taken up nearly 80% of the world-wide smartphone operating system market share. The appearance of Android platform and its popularity has resulted in a sharp rise in the number of reported vulnerabilities and consequently in the number of threats. Being an attractive target for attackers, Android mobile malware variants increased by 40% as compared to 29% growth in previous year [1]. SophosLabs detected nearly 10 million suspicious Android apps by the end of 2017. More than half were either malware or unwanted applications including adware [2].

Traditional approaches predominantly based on recognition of well-documented threats (signatures), are struggling to cope with this rate of growth in malware numbers each year. Consequently, the majority of malware strains (many of which are short lived, i.e., less than 24 hours) go undetected. In spite of these unprecedented malware numbers, the majority of malware samples are variants of known malware. Leveraging the lack of security testing in Android app markets, attackers commonly employ a suite of widely available tools to facilitate the malware app development. To make detection of these variants more challenging and hide the presence of malicious content, the attackers commonly use code obfuscation[1]. 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 obfuscator is encouraged in Android app development. Therefore, the presence of obfuscation alone does not indicate malicious nature of an app. However, certain types of obfuscation are commonly associated with malicious use; therefore, understanding whether an obfuscation is being used and what kind of obfuscation is applied is beneficial in facilitating malware triage and analysis.

Typically, in the course of manual analysis, reverse engineers see patterns that can be attributed to a presence of obfuscation (and sometimes even specific tools). Learning what represents a meaningful pattern though requires years of experience. This raises two questions. First: *Is it possible to identify these patterns automatically?* Although many development tools and obfuscators tend to leave traces in the program structure [3], recognizing these traces typically requires tedious manual analysis of binaries. Automating this process will significantly speed up malware triage, allowing to recognize potentially malicious and most likely legitimate applications. Second: *Do these traces have discriminatory power?* To be effective, these patterns have to remain stable among all binaries treated with the same tool, while varying between the programs obfuscated with different tools. Several studies have previously noted that various development tools involved in production of malware tend to leave unique traces [4], [3], [5]. These traces are often visible through analysis of structural similarities of malware binaries.

Following this insight, we turn our attention to spatial analysis and propose a novel fingerprinting approach for Android obfuscation tools. Spatial analysis is widely applied in many fields for exploratory analysis. It utilizes statistical techniques to reveal non-obvious patterns analysing spatial relationships of features. Specifically, we analyse spatial properties of the images generated from Android binary files to derive patterns that can uniquely represent various obfuscation tools. We survey first-order and second-order statistical features based on the spatial distribution of pixel values to analyze the image texture for potential structural patterns. We refer to quantitative representation of these patterns as fingerprints. We validate our approach on a dataset generated from seven Android obfuscation tools with various configuration options. With our approach, we are able to achieve nearly 90% accuracy in fingerprinting several obfuscation tools with specific configuration options.

978-1-5386-7493-2/18/$31.00 2018 Crown

The organization of this paper is as follows: Section 2 highlights related work; Section 3 provides a short background on obfuscation tools used; Section 4 presents the proposed approach; Section 5 discusses results obtained during experimental study; and Section 6 concludes this paper.

II. Related Work

Up until now, the research on Android apps' obfuscation 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.

Zhang et al., proposed a system named ViewDroid to detect repackaged Android apps based on the analysis of *software birthmark*, a set of unique characteristics that an app possesses which can be used to identify the app [6]. 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 [7]. Consequently, the authors believed that, for most obfuscators, the app code structure cannot be changed. Based on these assumptions, they analysed statistical characteristics of the app code to perform Android malware detection. Although no direct discussion is included in the work of Kumar et al. [8], the detection of obfuscation in Java malware through analysis of metrics such as word count, identifier length, etc. can be potentially extended to Android binaries.

In the study performed by Rastogi et al., a system named DroidChameleon focused on evaluation of resistance of commercial Android anti-malware softwares against various obfuscation techniques [9]. Similar research was performed by Mercaldo and Visaggio [10]. A framework was developed to apply obfuscation on known malware smali code to investigate the rate of detection in a presence of obfuscation. The research on identification of various obfuscation is scarce. Wang and Rountev investigated potential for recognition of obfuscation techniques by extracting featured strings out of the data section in the .dex file [11].

Spatial analysis has been extensively studied and widely applied in many fields for exploratory analysis. In the mobile domain, in general and the malware analysis area, specifically, spatial exploration has seen very limited use and is typically employed through visualization of binaries. Those few studies that were published in this area focus on one of three objectives: *individual analysis of malware samples* to gain new insights of their behavior [12], [13]; a broader *malware detection* that includes bulk visualization for samples' comparison and classification [14]; and *malware systematization* to understand similarities and common behavior [15], [16].

DAVAST, a data-centric system level visualization utility, proposed by Wüchner et al. [12] focused on individual malware analysis by using system data flow graph. The system can visualize system activities as data flow graphs with nodes presenting operating system entities such as processes, files, and sockets; edges denoting data flows between the nodes. Pattern matching of the graphs can tell the difference between the benign and malicious behavior. In [13], a tree-based navigation visual interactive Hex editor is used. This mechanism helps a user to pinpoint the underlying structure of binary file quickly to enhance the efficiency of individual malware analysis and detection.

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. [14] presented a network behavior-based anomaly detection system for identification of malicious attacks, masquerading applications, 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. [17] 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 it also aims to identify anomalies and patterns, focuses primarily on internal binary structure to identify obfuscation presence and thus falls under the second sub-category: feature-based malware comparison. In the particular direction, the work done by Nataraj and his group is a good example [4]. The authors focused on several operating systems including Android and explored gray-scale images generated from binaries but also the signals as well. The experimental results show that, in terms of classification accuracy, their result is comparable to dynamic analysis while their efficiency is **4,000** times faster than dynamic analysis.

Similar work was done by Liu and Wang [18]. In this work, the authors focused on large scale Windows malware analysis 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 that 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. [19], the image analysis method is further divided into types. The first type uses features that describe the textures in an image such as the Haralick features [20], and the second type uses the Local Binary Patterns features. They also discussed other 12 feature extraction mechanisms. 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 in the third category: Malware Similarity. However, using visualization in this direction is rare. In the work of Han et al. [15], 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. [16] a method named **NCD** (**N**ormalized **C**ompression **D**istance) is employed to enumerate code similarity between malicious

Android apps and visualize their clusters.

III. Obfuscation

Code obfuscation of mobile programs is gaining popularity. It is especially common on the Android platform, which employs the Java programming language. Since Java bytecode executes in the hardware independent Java virtual machine, it retains most of the information of the original source code. As a result it is easy to decompile and reverse engineer. To prevent reverse engineering, Android apps are typically protected through encryption or code transformation known as obfuscation.

Collberg et. al. [21], 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 reordering/splitting, change of variable encoding; *control obfuscation*, a more advanced transformation that aim to obscure the flow of the program control (e.g., redundant or junk code insertion, loops, statements reordering, code optimization); and *preventative transformations* that focus on decompilers' weaknesses.

In practice Android code obfuscation is typically automated through the use of obfuscation tools that mostly apply layout obfuscation. In this work we focus on several popular obfuscators that are widely applied in practice: **ProGuard**[1], the most well known of all the Android obfuscators, is a free Java class file obfuscator that detects and removes unused classes, fields, methods, and attributes and also optimizes bytecode and removes unused instructions. ProGuard is integrated into the Android build framework. ProGuard also includes code shrinking (e.g., removal of unused classes, fields) and code optimization, a procedure that applies transformations similar to compiler optimization. **DexGuard**[2] is the extended commercial version of ProGuard which focuses on code protection, with additional features like string and class encryption, obfuscation of the class and method names with non-ASCII characters. DexGuard-obfuscated samples were reportedly difficult to reverse engineer. **APK Protect**[3] is another advanced off-the-shelf obfuscation and protection service specific to Android executable files. **Bangcle** (in Chinese) or SecNeo (in English)[4] uses an online service to restructure the APK decrypting the app at runtime, obfuscating native libraries, and using libraries with stack protection enabled. **Klassmaster**[5] is advanced Java obfuscator that in addition to traditional layout obfuscation includes obfuscation of exception handling procedures. **JShrink**[6] is free Java code obfuscator that primarily offers layout and control obfuscation. **Allatori**[7] is a commercial Java obfuscator that includes a full range of obfuscation transformations.

[1]https://www.guardsquare.com/en/proguard
[2]https://www.guardsquare.com/en/dexguard
[3]https://sourceforge.net/projects/apkprotect/
[4]http://www.secneo.com/
[5]http://www.zelix.com/klassmaster/
[6]http://www.e-t.com/jshrink.html
[7]https://www.yworks.com/products/yguard

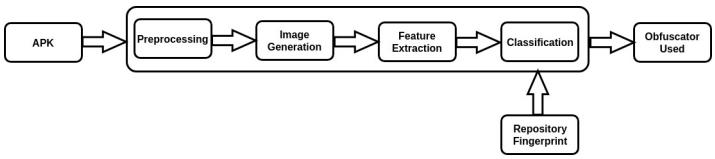


Fig. 1. The flow of fingerprinting process

IV. Fingerprinting obfuscation tools

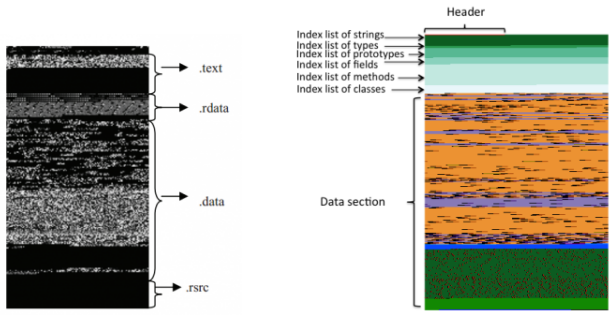
Our approach to determining unique patterns of obfuscation tools follows four main steps: preprocessing, image generation, feature extraction, and classification (Figure 1).

Given an Android apk file, the **Preprocessing** step ensures that the apk is valid for analysis, unpacks and extracts necessary executable .dex files.

The second step, **Image generation** produces a gray-scale image for each binary. The original .dex file is broken into 8-bit vectors and each vector is transformed into a decimal value which is then used to determine the gray-scale value of a pixel.

For each image, the **Feature extraction** step calculates first- and second-order statistical features. The first-order statistical features (Shannon entropy, Arithmetic mean, Chi square and Hamming weight) are extracted directly from image and thus show structural properties of .dex file. The second-order statistical features (Haralick Vector) give deeper textural analysis of the image, allowing the capture of small changes in adjacent bytes.

The last step **Classification** classifies images based on generated features and produces the most likely obfuscator employed in a given Android app.



(a) (b)

Fig. 2. Image generation approach: (a) Gray-scale malware image [22] , (b) Color malware image [3]

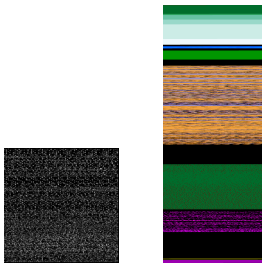
A. Generating Images

In this work we focus on analysis of spatial distribution of values derived from a binary file. To model the existing spatial variations, we employ image texture analysis. In malware binary analysis, two approaches were introduced to produce effective images representing binary structural information. Nataraj et al. [22] developed an approach for gray-scale binary images (Figure 2a). Each binary is treated as an 8-bit vector of integers organized in a 2D array, which is then mapped into

an image depending on byte's values (0 for black and 255 for white). A slightly different approach that maps each byte into corresponding color based on the nature/type of byte was considered by Jain et al.[3] (Figure 2b).

The comparative visual representation of both approaches on the example of "a2dp" Android app is given in Figure 3. 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 height of the color image is larger. The reason lies in the way the images are generated. The color image is generated based on the internal structure of the .dex file. First, the information contained in the header file is read out so that the remaining part can be determined by using the address pointers and offsets. Then for different contents, different colors are allocated.

Unlike the gray-scale image, the color image incorporates 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. And padding bytes are commonly used in a .dex file to create the required alignment. Aside these differences both representations contain the same information. In our experiments we performed preliminary analysis for both types of the images. Since these experiments produced similar results, we employed a slightly simpler gray-scale method for the image generation.



(a) (b)

Fig. 3. Two types of original images for "a2dp" app: (a) Original gray tone image, (b) Original color image

B. Features

Once the images are generated, we proceed to image texture analysis. Since an image is a numerical representation of byte values, image texture represents the spatial organization of the gray levels of the pixels in a binary. Although many numeric texture analysis approaches were introduced in the past research, statistical method is seen as the most powerful image analysis method [23], [24].

We therefore focus on this method and extract the statistical features that describe the binary image texture. Usually, the statistics are performed on the gray-scale level of each pixel contained in the image.

We leverage first-order and second-order statistical analysis at this stage. First-order statistics are simple to calculate and effective in revealing straightforward differences in images. We calculate Shannon entropy, Arithmetic mean, Chi square, and Hamming weight. These first-order statistics give us a coarse view of potential nuances in the images.

The deeper insight into relationships between individual pixels can be derived through second-order statistics that look at correlations between pixels. A well known approach to statistical analysis of image texture is the gray-level co-occurrence matrix (GLCM) introduced by Haralick et al. in 1973 [25]. GLCM characterizes image by analysing frequency of neighbouring pixels in a specific spatial relationship. For second-order statistical analysis, we employ so-called Haralick features that represent texture features of GLCM.

The statistical features we utilized in this work are defined as follows:

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

$$H(X) = -\sum_{i=0}^{n=1} p(X_i) \log_{10} p(X_i) \tag{1}$$

where X is a random variable ranging from 0 to 255 to denote gray level of the pixels. $p(X_i)$ is the probability mass function. Usually, compressed or encrypted files have higher entropy than those that 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.

$$\bar{X} = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{2}$$

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

$$X^2 = \sum_{i=0}^{n=1} \frac{(\text{observed} - \text{expected})^2}{\text{expected}}. \tag{3}$$

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

Hamming weight: a count of non-zero symbols in a given alphabet. In our experiment, the alphabet contains only binary values. Therefore, the Hamming weight is the fraction of the total number of bits equal to 1 divided by the total number of bits.

$$\sum_{x=0}^{N-1} \sum_{y=0}^{M-1} (1 - \delta(X(x,y), 0)). \tag{4}$$

Haralick vector: Let i and j be gray-scale values, the entry of co-occurrence matrix is the occurrences (frequencies) of i and j separated by vector distance $\vec{d}$, which can be further expressed in terms of absolute distant d and the direction defined by the angle θ [25]. In this work, we set $d = 1$; and θ can be $0^o, 45^o, 90^o$ and 135^o. In other words, the neighbouring pixels are analyzed at various rotation angles (e.g. 0, 45, 90,

and 135 degrees) as depicted in the Figure 4. By dividing total number of neighbouring pixels $R(d, \theta)$, the co-occurrence matrix is converted into *point probability matrix* whose entry is denoted by $p_{\vec{d}}(i, j)$, the normalized probability, which is the base of other statistics.

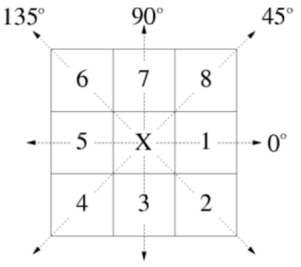


Fig. 4. The four directions of adjacency for calculating the co-occurrence matrix, where "X" is the pixel under investigation, with eight nearest-neighbor pixels labeled in the framework to describe pixel connectivity [23]

We derive all 13 Haralick features and use them as a vector, which we refer to as a Haralick Vector. Mathematical definition and its components are listed in the Table I. Since the Haralick vector can be calculated along four directions, the final analysis is performed on $4 \times 13 = 52$ Haralick features. In the rest of this work, any feature from the 0° is labelled by **H** since it is the horizontal direction; **V** (vertical) stands for 90°; **D** (diagonal) marks 45° direction; **SD** (secondary-diagonal) is for 135°. According to this convention, the label "H13" can be explained as the 13th component of the Haralick vector calculated in the 0° (horizontal) direction.

TABLE I
COMPONENTS OF HARALICK VECTOR

Name	Formula
Angular Second Moment (**energy**, **Uniformity**)	$f_1 = \sum_{i=1}^{N_g} \sum_{j=1}^{N_g} p_i j^2$
Contrast **(Inertia)**	$f_2 = \sum_{n=0}^{N_g-1} n^2 \sum_{i=1}^{N_g} \sum_{j=1}^{N_g} p_{ij}$ with $\lvert i-j \rvert = n$
Correlation	$f_3 = \frac{\sum_i \sum_j (ij) p(i,j) - \mu_x \mu_y}{\sigma_x \sigma_y}$
Sum of Squares: Variance	$f_4 = \sum_i \sum_j (i-\mu)^2 p(i,j)$
Inverse Difference Moment (**Homogeneity**)	$f_5 = \sum_i \sum_j \frac{1}{1+(i-j)^2} p(i,j)$
Sum Average	$f_6 = \sum_{i=2}^{2N_g} i p_{x+y}(i)$
Sum Variance	$f_7 = \sum_{i=2}^{2N_g} (i - f_8)^2 p_{x+y}(i)$
Sum Entropy	$f_8 = -\sum_{i=2}^{2N_g} p_{x+y} \log[p_{x+y}(i)]$
Entropy	$f_9 = -\sum_i \sum_j p(i,j) \log(p(i,j))$
Difference Entropy	$f_{10} = -\sum_{i=0}^{N_g-1} p_{x-y}(i) \log(p_{x-y}(i))$
Difference Variance	$f_{11} =$ variance of p_{x-y}
Information Measure of Correlation 1	$f_{12} = \frac{HXY - HXY1}{\max HX, HY}$
Information Measure of Correlation 2	$f_{13} = (1 - \exp[-2.0(HXY2 - HXY)])^{1/2}$

V. EXPERIMENTAL STUDY

The goal of the experimental study is to validate the proposed approach and to test effectiveness of features in classifying the obfuscators. The classifiers are evaluated with a 10-fold cross-validation.

TABLE II
DESCRIPTION OF DATASET

Name	Obfuscation Options	Total Number	Used in Classification
Allatori	Default	325	298
	Layout		305
	Data		305
	Control		305
Klassmaster	Default	325	298
	Layout		305
	Data		305
	Control		305
DexGuard	Default	325	298
	Layout		305
	Data		298
	Control		298
Jshrink	Default (Layout)	325	325
ProGuard	Default (Layout)	325	325
Bangcle		50	0
Non-obfuscated		325	284
Total Number Used in Classification: 4559			

A. Dataset

The dataset comprises both obfuscated and non-obfuscated Android apps. A total of 1399 unique, as defined by MD5 hash, Android app's source code is collected from F-droid[8] market. The source code is compiled by Ant with Ivy[9]. Then the obfuscation is applied in two ways depending on the obfuscator tool being used. Some obfuscation tools are used during compilation time by including corresponding .jar file, such as Allatori and DexGuard. Here the obfuscation is done automatically by executing C-shell scripts. Some Windows-based GUI obfuscation tools can be directly used on the .apk files after compilation step e.g., Bangcle. In this case, obfuscation is done manually.

Out of 1399 collected apps many could not be obfuscated due to various reasons such as version collision, failure to download important external files, etc. We successfully compiled 325 apps and obfuscated each of them by 5 different obfuscators: Allatori, DexGuard, Jshrink, Klassmaster, and ProGuard. For each of the obfuscators, we experimented with 4 different obfuscation types: Default, Layout, Data, and Control. We were not able to successfully obfuscate all 325 apps with each obfuscation type due to various reasons including incompatible version of the Android plugin, SDK version collision, improper environment parameters set in the configuration files, etc., resulting in 305 or 298 apps in some cases.

For classification, we used obfuscators with all the different obfuscation options to find the most efficient app-independent features. The default configuration for Allatori, DexGuard, and Klassmaster includes layout obfuscation, data obfuscation, and control obfuscation. Whereas, Jshrink and ProGuard by default only apply layout obfuscation. We also manually generated 50 samples of Bangcle that are only used in testing first-order statistics. Information about this dataset is presented in Table II.

B. Evaluation of Extracted Features

[8]https://f-droid.org/
[9]http://ant.apache.org/

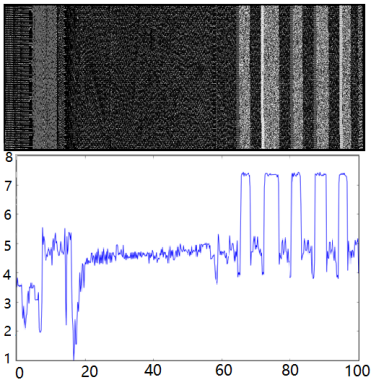


Fig. 5. Gray-scale image and its corresponding value of Shannon entropy

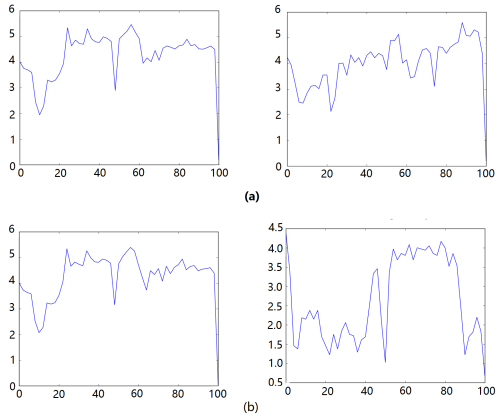


Fig. 6. Shannon entropy is calculated on different apps.(a) For "am.ed.exportcontacts", Bangcle (Left), ProGuard (Right) (b) For "se.johanhil.clipboard", Bangcle (Left), ProGuard (Right)

1) First-order Statistical Features: The first test is to check if the first-order statistics can describe the internal structure of the images, for we want to know if the obfuscation can be reflected by certain image structures directly.

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 motivation for this method is from the observation of the images. We generated a large number of gray-scale images under different circumstances and found a common image structure with horizontal stripes varying in location and contrast. Consequently, to check the efficacy of the first-order statistics, we sliced the images along the horizontal stripes and extracted numerical feature values. 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 Fig. 5. All the four first-order statistics are calculated on those segments. The results are plotted in a graph where the x-axis stands for percentage of the image's height and the y-axis is the statistical value obtained. Fig. 5 depicts how the internal structure of a gray image can be disclosed by the first-order statistical features. For the convenience of comparison, the gray-scale image has been rotated 90^o counter-clockwise. Since all the four statistics lead to equivalent results, only Shannon entropy is shown for brevity.

The experiments similar to the one described by Fig. 5 have been performed repeatedly on a group of randomly selected apps to confirm that the internal structure can be reflected by the first-order statistics effectively. However, this is not the goal we are content with. We want to find if there is a stable relation between the image internal structure and the obfuscator's behavior. Further experiments show that the first-order statistics can easily reveal the usage of Bangcle. No matter what app has been obfuscated by Bangcle, the graph always shows the same profile. For some other obfuscators such as ProGuard, it is more challenging to capture a stable profile from the curves (Fig 6). 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, Bangcle always relocates most of the content in the .dex file to some other resource file, leaving behind a .dex file of fixed length with small variation in the structure.

Thus, this profile is unique to Bangcle. From the results it is evident that the curve drawn by calculating first-order statistical quantities not only depends 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, Default dataset for all obfuscators is utilized. Three of the first order statistics, Chi square, mean value and Shannon Entropy, are calculated for each sample and then plotted in a 3D graph as shown in Fig.7. This is the second approach in which the calculation is performed on the whole image, where 3 of the statistics are selected as the coordinates to decide its position in a 3D graph.

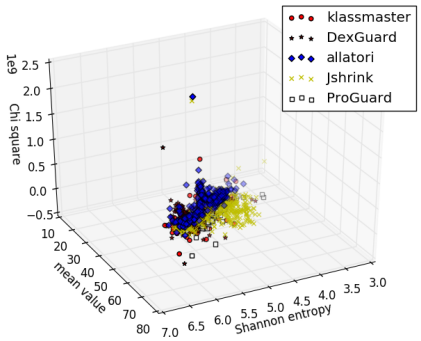


Fig. 7. 3D graph for 5 obfuscators used on the Default dataset, using 3 first-order statistical quantities as coordinates.

Figure 7 implies the possibility of using clustering method. It shows the dots from different obfuscators have the tendency to be clustered in different groups. 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 close to be distinguished. A similar conclusion was drawn with another combination of first-order statistics (Hamming weight, Chi square and Shannon entropy) because mean and Hamming weight resulted in same values. The reason for this is that while counting bits equal to 1 for Hamming weight, we are actually calculating a sum of 1s and diving it with the total number of bits, and this is equivalent to the mean value.

To clarify the effectiveness of first-order statistics quantitatively, classification is performed on the entire dataset – 5 obfuscators with Default, Layout, Data and Control options. The results are summarized in Table III. In Table III, SMO

TABLE III
RESULTS OF CLASSIFICATION USING THE FIRST-ORDER STATISTICS

Algorithm / Obfuscation:	Bagging	Regression	Kstar	SMO
Default:	54.86%	55.31%	27.97%	41.12%
Layout:	43%	47.15%	18.78%	40.19%
Data:	52.09%	54.07%	30.62%	45.48%
Control:	50.55%	55.94%	30.50%	35.90%

and Kstar are implementations of SVM and k-NN algorithms provided by weka [26]. However, the highest accuracy generated by Regression is still less than 56%. In the worst case, the accuracy from k-NN is no more than 19%. This means, only for certain obfuscators such as Bangcle, the first-order statistics are useful; but they are not sufficient for other obfuscators like ProGuard, Klassmaster, etc.

2) Second-order Statistical Features: As an example, a gray-scale image generated from an app named "monakhv" is shown in Figure 8 and the corresponding Haralick features are listed in Table IV. Unlike the gray-scale image contained in Figure 5, this image does not contain obvious horizontal stripes.

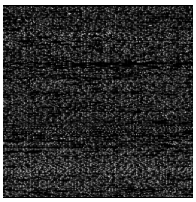


Fig. 8. Gray Scale Image of "monakhv".

It is clear from the results that the values along 45° and 135° are almost identical up to 2 decimal places. This numerical feature shows the symmetry between the two angles, which is also visible in Figure 8. In contrast, the values along 0° and 90° are quite different which is reflected by the vague horizontal stripes in the image.

C. Classification

First of all, a binary classification is performed on a dataset containing 284 non-obfuscated apps and 305 obfuscated apps with the default option. Only default option is considered for

TABLE IV
HARALICK FEATURES OF NON-OBFUSCATED "MONAKHV"

0°	45°	90°	135°
1.59472e-01	1.20405e-01	1.20380e-01	1.20239e-01
5.13845e+00	1.24506e+01	1.24553e+01	1.24919e+01
9.08526e-01	7.78173e-01	7.78099e-01	7.77438e-01
2.80871e+01	2.80638e+01	2.80651e+01	2.80639e+01
9.53298e-01	8.11906e-01	8.11913e-01	8.11495e-01
1.62506e+01	1.62493e+01	1.62496e+01	1.62493e+01
1.07210e+02	9.98048e+01	9.98054e+01	9.97638e+01
3.22172e+00	3.58547e+00	3.58623e+00	3.58744e+00
3.38706e+00	4.08070e+00	4.08066e+00	4.08186e+00
3.899647e-02	2.576184e-02	2.576488e-02	2.573827e-02
4.34044e-01	1.45525e+00	1.45525e+00	1.45745e+00
-9.02657e-01	-6.77462e-01	-6.77486e-01	-6.77085e-01
9.98097e-01	9.92326e-01	9.92327e-01	9.92308e-01

this experiment as it contains features from all different types of obfuscation including layout, data, and control. In the non-obfuscated dataset, some apps are generated from the source code different from those used to generate the obfuscated apps, since in the real world not all the samples are from the same original code. The classification results are shown in Table V. 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 default features so that the smallest accuracy value of the classification is above 61%.

TABLE V
BINARY CLASSIFICATION OF OBFUSCATED AND NON-OBFUSCATED APPS

Algorithm:	Bagging	KSTAR	SMO	Regression
Accuracy:	65.53%	67.06%	61.29%	64.01%
Precision:	64.80%	64.90%	72.60%	63.60%

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 experiments, all 52 Haralick features are used. For each dataset the same feature selection experiment was performed. The required features are selected according to their information gain value and the overall classification accuracy. We thus selected the top 11 for Default, top 14 for Layout, top 16 for Data, and top 23 for the Control dataset. The features selected for each obfuscation type, ranked according to their information gain values, are listed in Table VI.

Figure 9 presents classification results for four different types of dataset: Default, Layout, Data, and Control. The x-axis and y-axis denote number of features and accuracy, respectively. With each iteration of the experiment the features with the least information gain are taken away. The results shows that the highest accuracy obtained with Default dataset is 56.92%. With Layout, Data, and Control the accuracies are 51.02%, 65.52% and 60.22%, respectively. In all the cases, accuracy is less than 66%, which is very low compared to the similar work done on malware family classification. This is because, compared to the tangible malware families, our purpose of pursuing obfuscator's characteristic behavior is more abstract. The characteristic behavior cannot have observable

TABLE VI
FEATURE GROUP SELECTED FROM THE HARALICK VECTOR

Default		Layout		Data		Control	
IG	Features	IG	Features	IG	Features	IG	Features
0.196	D13	0.1739	V3	0.0959	H3	0.0964	SD7
0.1836	SD3	0.1585	SD3	0.0709	D3	0.0961	D7
0.1777	D3	0.1503	SD13	0.0677	SD3	0.0873	D13
0.1619	V3	0.1359	H3	0.0581	H13	0.087	SD13
0.1596	SD13	0.1181	D13	0.0544	V3	0.0817	SD3
0.154	H13	0.1098	H13	0.0523	H2	0.075	D3
0.145	H3	0.1064	D3	0.048	H4	0.0707	V13
0.1283	D12	0.1039	V12	0.0479	V7	0.0672	H12
0.1071	V10	0.0877	V13	0.0479	SD7	0.0654	H13
0.1055	V12	0.0764	V2	0.0467	V4	0.062	H7
0.1042	V13	0.0692	H2	0.0467	D4	0.0588	H4
-	-	0.0682	D12	0.0467	SD4	0.0568	H2
-	-	0.068	SD12	0.0466	D7	0.0564	SD4
-	-	0.0669	D6	0.0458	H7	0.0564	D4
-	-	-	-	0.0384	SD2	0.0556	V4
-	-	-	-	0.0384	D2	0.0512	V12
-	-	-	-	-	-	0.046	D12
-	-	-	-	-	-	0.042	V7
-	-	-	-	-	-	0.0367	V6
-	-	-	-	-	-	0.0367	D6
-	-	-	-	-	-	0.0367	H6
-	-	-	-	-	-	0.0367	SD6
-	-	-	-	-	-	0.0309	H3

independent existence without applying the obfuscators on the apps. Therefore, it is impossible to figure out completely app-independent features.

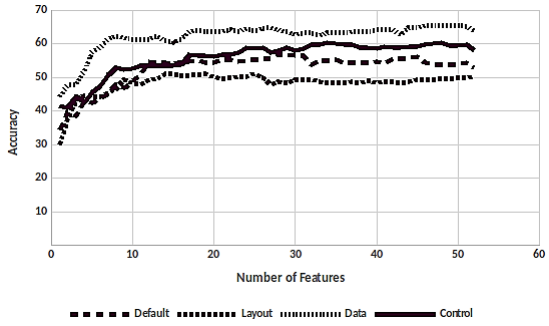


Fig. 9. Haralick feature selection

TABLE VII
FEATURE GROUP SELECTED WITH MIX FEATURES

Default		Layout		Data		Control	
IG	Features	IG	Features	IG	Features	IG	Features
0.3155	Shannon entropy	0.3366	Shannon entropy	0.1445	Hamming weight	0.0964	SD7
0.3055	Hamming weight	0.2966	Hamming weight	0.0959	H3	0.0961	D7
0.196	D13	0.1739	V3	0.0709	D3	0.0873	D13
0.1836	SD3	0.1585	SD3	0.0677	SD3	0.087	SD13
0.1777	D3	0.1503	SD13	0.0581	H13	0.0837	Hamming weight
0.1619	V3	0.1359	H3	0.0544	V3	0.0817	SD3
0.1596	SD13	0.1181	D13	0.0523	H2	0.075	D3
0.154	H13	0.1132	Mean value	0.048	H4	0.0707	V13
0.145	H3	0.1125	Chi square	0.0479	SD7	0.0672	H12
0.1402	Mean value	0.1098	H13	0.0479	V7	0.0654	H13
0.1343	Chi square	0.1064	D3	0.0467	V4	0.062	H7
0.1283	D12	0.1039	V12	0.0467	SD4	0.0588	H4
0.1071	V10	-	-	0.0467	D4	0.0568	H2
-	-	-	-	0.0466	D7	0.0564	D4
-	-	-	-	0.0458	H7	0.0564	SD4
-	-	-	-	0.0444	Shannon entropy	0.0556	V4
-	-	-	-	0.0384	D2	0.0512	V12
-	-	-	-	0.0384	SD2	0.046	D12
-	-	-	-	-	-	0.042	V7
-	-	-	-	-	-	0.0367	D6
-	-	-	-	-	-	0.0367	SD6
-	-	-	-	-	-	0.0367	H6
-	-	-	-	-	-	0.0367	V6
-	-	-	-	-	-	0.0309	H3

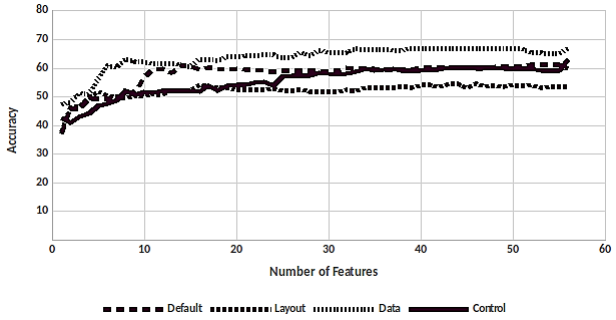


Fig. 10. Mix feature selection

With the aim to improve the overall accuracy, experiments are also performed on mixing the first and second order features. In the mix case, all 56 features are used. Here also the required features are selected according to their information gain value and the overall classification accuracy. For Default the top 13, for Layout the top 12, for Data the top 18 and for Control the top 24 features are selected. The features selected for each obfuscation type, ranked according to their information gain values, are listed in Table VII. It can be seen that some of the first-order features with higher information gain value appear in the mix feature list.

Figure 10 presents similar classification results for four different types of dataset with mixed features. The results show that the highest accuracy obtained with Default dataset is 61.27%. With Layout, Data, and Control the accuracies are 54.49%, 66.96% and 63.43%, respectively. These results are slightly better than when only Haralick features are used, the reason being the selection of some first-order features contributes to the overall accuracy. But still the accuracy obtained is less than 68%.

1) Fingerprinting Obfuscators: The final set of experiments were performed to point out by what group of the features and to what extent an obfuscator can be fingerprinted. This experiment was performed on all datasets: Default, Layout, Data, and Control for both Haralick features and Mix features. In the series of classification, the starting feature group contains all the selected features mentioned before in Table VI and Table VII. Then those features are taken away one by one according to their information gain value. The final set of features that reaches the highest accuracy is regarded as the fingerprint for a given obfuscator. These fingerprint features are summarized in Table VIII to Table XI for Haralick features and Table XII to Table XV for Mix features, respectively. The features selected in the group are ranked according to their decreasing information gain value.

As our results show, with Haralick features we are able to distinguish the obfuscators using default and layout configurations with at least 80% accuracy. Overall, Jshrink and ProGuard have the highest accuracy and precision as compared to other obfuscators. We can detect the presence of Jshrink with 88% accuracy in default configuration (82% ProGuard) and 89% accuracy in layout configuration (90% ProGuard). The reason is that they both mainly apply layout obfuscation while the default obfuscation option contains layout obfuscation plus

shrinking and some optimization. The accuracy for data and control dataset ranges between 68% and 84%, respectively. In both configurations, DexGuard achieves the highest accuracy among other obfuscators (84%).

With Mix features, as expected the overall accuracies in fingerprinting obfuscators increased slightly. Similar to experiments with Haralick features, Jshrink has the highest accuracy of 91% with Mix features for both Default and Layout dataset. Other obfuscators also show accuracy that ranges between 80% and 90% for these configurations. For data and control configurations we are able to achieve over 80% for DexGuard, although we are less accurate for other obfuscation tools (between 66% and 71%).

TABLE VIII
FINGERPRINTING WITH HARALICK FEATURES & DEFAULT DATASET

Obfuscator	Feature Group	Accuracy	Precision
Jshrink	SD3, V3, D3, V12, H3, D12, V10, H13, D13	88.24%	83.3%
Klassmaster	D13, SD13, V13, V3	80.77%	52.4%
ProGuard	SD3, D13, SD13, D3, V13, H3, V3, H13, D12, V12	82.72%	64.2%
DexGuard	D13, SD13, H13, SD3, D3, H3, V13, V10, V12, D12	81.81%	56%
Allatori	SD13, D13, H3, V3, H13, V12, V13, SD3, D12, V10, D3	81.61%	55.8%

TABLE IX
FINGERPRINTING WITH HARALICK FEATURES & LAYOUT DATASET

Obfuscator	Feature Group	Accuracy	Precision
Jshrink	V3, SD3, V12, D3, H3, V2, H2, D12, SD12, V13, H13	88.72%	83.6%
Klassmaster	D13, SD13, V3, SD3, V2, V12, D3, H3, H13, D12	80.38%	37.5%
ProGuard	D13, SD13, D3, SD3, D12, V13, SD12, H13, H3, V3, V12	89.81%	82.4%
DexGuard	SD13, D13, V3, SD3, V2, V12, D3, H3, H13, D12	80.45%	40%
Allatori	V3, H3, V12, SD3, D3, H2	85.19%	84.8%

TABLE X
FINGERPRINTING WITH HARALICK FEATURES & DATA DATASET

Obfuscator	Feature Group	Accuracy	Precision
Allatori	V3, D3, SD7, H13	72.71%	65.4%
Klassmaster	H3	68.84%	62.2%
DexGuard	H3, V7, SD3, D3, V3, H13	83.87%	80.3%

TABLE XI
FINGERPRINTING WITH HARALICK FEATURES & CONTROL DATASET

Obfuscator	Feature Group	Accuracy	Precision
Allatori	SD13, D3, D4, H12, H13, H7, D7, H3, H4, H6	67.29%	53.2%
Klassmaster	SD13, D3, D4, H12	65.08%	35.7%
DexGuard	SD13, D7, SD7, D13, H12, SD3, D3, V13, H13, H7, H4, H2, SD4, D4, SD6, D6, V6, H6, V4, V12, D12, V7	84.31%	80.2%

D. Validation

In this section, we validate our approach with the results of an existing technique. Wang and Rountev [11] proposed an approach that uses string-based features of the .dex file to identify Android obfuscators. First, the size of our dataset is roughly double that of their dataset (2600 vs 4559). Both generated apps with similar obfuscators except for some exceptions. Second, they used limited configurations options, default and custom, to train the classifier. In the case of ProGuard for custom options, focus was on optimization, renaming, and repackaging, while in the case of Allatori/Dasho they focus only on renaming and control-flow modifications.

TABLE XII
FINGERPRINTING WITH MIX FEATURES & DEFAULT DATASET

Obfuscator	Feature Group	Accuracy	Precision
Jshrink	Shannon_entropy, SD3, V3, D3, H3, Hamming_weight, D12, V10, H13, D13, SD13, Chi_square	90.90%	84.6%
Klassmaster	Hamming_weight, Shannon_entropy, Mean_value, D13, SD13, Chi_square, V3, V10, SD3, D3, H13	81.29%	54.4%
ProGuard	Hamming_weight, Shannon_entropy, SD3, D13, SD13, D3, H3, V3, H13, Chi_square, Mean_value	84.73%	69.4%
DexGuard	Hamming_weight, D13, SD13, H13, SD3, D3, H3, Mean_value, Chi_square, Shannon_entropy, V10	86.22%	70.9%
Allatori	Shannon_entropy, SD13	80.44%	25%

TABLE XIII
FINGERPRINTING WITH MIX FEATURES & LAYOUT DATASET

Obfuscator	Feature Group	Accuracy	Precision
Jshrink	Shannon_entropy, V3, SD3, V12, D3, H3, Hamming_weight, Chi_square, Mean_value, H13, SD13	91.09%	84.1%
Klassmaster	Shannon_entropy, Hamming_weight, Mean_value, D13, SD13, V3, Chi_square, SD3, V12, D3, H3, H13	80.26%	38.9%
ProGuard	Hamming_weight, D13, SD13, Shannon_entropy, D3, SD3, H13, Chi_square, H3, Mean_value, V3, V12	90.38%	81.1%
DexGuard	Shannon_entropy, Hamming_weight, Mean_value, SD13, D13, V3, Chi_square, SD3, V12, D3, H3, H13	80.64%	55.6%
Allatori	Hamming_weight, Shannon_entropy, V3, H3, V12, SD3, D3, Chi_square, Mean_value	82.63%	65.4%

In our proposed approach, we considered nearly all types of code obfuscation options available with an obfuscator tool separately, only excluding user-custom options (for e.g., where a user provides a custom encryption algorithm). Finally, the underlying technique chosen is different. They mainly focus on string-based features of the .dex file but for control-flow modification they extracted instruction sequence n-grams, i.e., different types of features for different configurations. Moreover, the existing approach could not tackle other common obfuscation options like string encryption, merging and splitting functions, as the string-based model failed to represent other relevant properties from the .dex file. On the other hand, we utilize spatial analysis to reveal obfuscation patterns on the .dex file. The main advantage of using spatial analysis is that it will always exhibit distinctive traces varying in intensities that will uniquely fingerprint any obfuscator. While we are not as accurate as Wang and Rountev [11] (97% vs 90%), the result shows that the same type of statistical features have the potential to fingerprint obfuscation tools even when they are employed for simple or advanced level of code modification.

VI. CONCLUSION

In this paper, we have presented an approach for analysing the presence of obfuscation in Android binary files through the use of spatial texture analysis. Advantageously, this technique can be implemented in real-time to classify an Android app by the fingerprints generated from the repository (Figure 1). Our results show that obfuscation tools leave behind traces that form unique patterns distinct for different tools. The straightforward analysis of the first-order statistics showed that we can easily distinguish the Bangcle obfuscator. However the more advanced obfuscation techniques require deeper insight into binary files which is achieved with second-order statistical analysis. In our experiments, we observed that the proposed technique can indicate the degree of similarity between the obfuscators, and show the subtle changes caused by different

TABLE XIV
FINGERPRINTING WITH MIX FEATURES & DATA DATASET

Obfuscator	Feature Group	Accuracy	Precision
Allatori	V3, H4, SD7, H7, H3, D2, H2, Shannon_entropy, H13, D3, SD4, D4, SD3, SD2, V7, V4, D7, Hamming_weight	71.82%	64.5%
Klassmaster	Hamming_weight, H3, SD3, D3, H2, SD2, H4, V7, V4, Shannon_entropy, H7, H13, D2, SD4, V3, D7, D4, SD7	70.28%	61.3%
DexGuard	Hamming_weight, H3, V7, SD3, D3, V3, H13, H2, H4, SD7, V4, SD4, D4, D7, H7	84.97%	84%

TABLE XV
FINGERPRINTING WITH MIX FEATURES & CONTROL DATASET

Obfuscator	Feature Group	Accuracy	Precision
Allatori	SD13, SD7, H13, D3, D4, H12, H7, H6, H2, H3, H4, D6, D7, D12, SD4, SD6, V13, SD3, V12, D13, V4, V6, V7	67.85%	54.7%
Klassmaster	Hamming_weight, SD7, H13, D3, D4, H12, H7, H6, H2, H3, H4	66.96%	52.5%
DexGuard	SD13, D7, SD7, D13, H12, SD3, Hamming_weight, D3, V13, H13, H7, H4, H2, SD4, D4, SD6, D6, V6, H6, V4, V12, D12, V7, H3	84.09%	82.8%

configurations of the same obfuscator. To show the potential of fingerprinting the obfuscators, a series of binary classifications has been performed for different obfuscators with different obfuscation types. The result shows, with selected group of Haralick features, by using the recommended classifications algorithm, the accuracy of distinguishing an obfuscator is at least more than 80% for the Default and Layout dataset and not less than 65% for the Data and Control dataset. We further mixed the first and second order features to see if overall results are improved. In that case, we saw a slight increase in the accuracy of distinguishing an obfuscator.

Our future work involves investigation in several directions. We believe that the first-order statistics should be fully utilized in a way to extract more detailed information about the obfuscators. Further a larger dataset should be established so that all the features can be put to a stricter test. Last but not least, this technique can be further extended to fingerprint specific algorithms used on a particular file.

REFERENCES

[1] Symantec, "Internet security threat report," April 2016. [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf

[2] SophosLabs, "Looking ahead: Sophoslabs 2018 malware forecast," February 2018. [Online]. Available: https://media.scmagazine.com/documents/321/sophos_2018_malware_forecast,__80124.pdf

[3] A. Jain, H. Gonzalez, and N. Stakhanova, "Enriching reverse engineering through visual exploration of android binaries," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, ser. PPREW-5. New York, NY, USA: ACM, December 2015, pp. 1–9.

[4] L. Nataraj and B. S. Manjunath, "Spam: Signal processing to analyze malware," *IEEE Signal Processing Magazine*, vol. 33, no. 2, pp. 105–117, March 2016.

[5] R. Chouchane, N. Stakhanova, A. Walenstein, and A. Lakhotia, "Detecting machine-morphed malware variants via engine attribution," *J. Comput. Virol.*, vol. 9, no. 3, pp. 137–157, Aug. 2013.

[6] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, ser. WiSec'14. New York, NY, USA: ACM, July 2014, pp. 25–36.

[7] C. Wang, Z. Li, L. Gong, X. Mo, H. Yang, and Y. Zhao, "An android malicious code detection method based on improved dca algorithm," *Entropy*, vol. 19, no. 2, pp. 65–80, February 2017.

[8] R. Kumar and A. R. E. Vaishakh, "Detection of obfuscation in java malware," *Procedia Computer Science*, vol. 78, pp. 521–529, January 2016.

[9] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: Evaluating android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, ser. ASIA CCS'13. New York, NY, USA: ACM, May 2013, pp. 329–334.

[10] F. Mercaldo and C. A. Visaggio, "Evaluating malwares obfuscation techniques against antimalware detection algorithms," Technical Report, March 2015. [Online]. Available: https://www.researchgate.net/publication/274249693_Evaluating_malwares_obfuscation_techniques_against_antimalware_detection_algorithms

[11] Y. Wang and A. Rountev, "Who changed you? obfuscator identification for android," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '17. Piscataway, NY, USA: IEEE Press, May 2017, pp. 154–164.

[12] T. Wüchner, A. Pretschner, and M. Ochoa, "Davast: Data-centric system level activity visualization," in *Proceedings of the Eleventh Workshop on Visualization for Cyber Security*, ser. VizSec '14. New York, NY, USA: ACM, November 2014, pp. 25–32.

[13] J. Donahue, A. Paturi, and S. Mukkamala, "Visualization techniques for efficient malware detection," in *Proceedings of the IEEE International Conference on Intelligence and Security Informatics*, ser. VizSec'14. Seattle, WA, USA: IEEE Press, June 2013, pp. 289–291.

[14] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, "Mobile malware detection through analysis of deviations in application network behavior," *Computers & Security*, vol. 43, pp. 1–18, June 2014.

[15] K. Han, B. Kang, and E. G. Im, "Malware analysis using visualized image matrices," *The Scientific World Journal*, vol. 2014, pp. 1–15, July 2014.

[16] A. Paturi, M. Cherukuri, J. Donahue, and S. Mukkamala, "Mobile malware visual analytics and similarities of attack toolkits," in *Proceedings of the International Conference on Collaboration Technologies and Systems*, ser. CTS'13. San Diego, CA, USA: IEEE, May 2013, pp. 149–154.

[17] R. Gove, J. Saxe, S. Gold, A. Long, and G. Bergamo, "Seem: A scalable visualization for comparing multiple large sets of attributes for malware analysis," in *Proceedings of the Eleventh Workshop on Visualization for Cyber Security*, ser. VizSec '14. New York, NY, USA: ACM, November 2014, pp. 72–79.

[18] L. Liu and B. Wang, "Malware classification using gray-scale images and ensemble learning," in *Proceedings of the 3rd International Conference on Systems and Informatics*, ser. ICSAI'16. Shanghai, China: IEEE, January 2016, pp. 1018–1022.

[19] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, "Novel feature extraction, selection and fusion for effective malware family classification," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '16. New York, NY, USA: ACM, March 2016, pp. 183–194.

[20] L. P. Coelho, "Mahotas features," February 2015. [Online]. Available: http://mahotas.readthedocs.org/en/latest/features.html

[21] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Sciences, The University of Auckland, Tech. Rep. 148, July 1997. [Online]. Available: http://www.cs.auckland.ac.nz/$\sim$collberg/Research/Publications/CollbergThomborsonLow97a/index.html

[22] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: Visualization and automatic classification," in *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, ser. VizSec'11. New York, NY, USA: ACM, July 2011, pp. 1–7.

[23] R. M. Haralick, "Statistical and structural approaches to texture," *Proceedings of the IEEE*, vol. 67, no. 5, pp. 786–804, 1979.

[24] A. Materka and M. Strzelecki, "Texture analysis methods - a review," Technical University of Lodz, Institute of Electronics ul. Stefanowskiego 18, 90-924 Lodz, Poland, Tech. Rep., 1998. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.4968&rep=rep1&type=pdf

[25] R. M. Haralick, K. Shanmugam, and I. Dinstein, "Textural features for image classification," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-3, no. 6, pp. 610–621, Nov 1973.

[26] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, June 2009.