Detection of Obfuscation Techniques in Android Applications

Alessandro Bacci*, Alberto Bartoli*, Fabio Martinelli†, Eric Medvet*, Francesco Mercaldo†
*Dipartimento di Ingegneria e Architettura, Università di Trieste, Trieste, Italy
{abacci, emedvet, bartoli.alberto}@units.it
†Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Pisa, Italy
{fabio.martinelli, francesco.mercaldo}@iit.cnr.it

***Abstract*—Current signature detection mechanisms can be easily evaded by malware writers by applying obfuscation techniques. Employing morphing code techniques, attackers are able to generate several variants of one malicious sample, making the corresponding signature obsolete. Considering that the signature definition is a laborious process manually performed by security analysts, in this paper we propose a method, exploiting static analysis and Machine Learning classification algorithms, to identify whether a mobile application is modified by means of one or more morphing techniques. We perform experiments on a real-world dataset of Android applications (morphed and original), obtaining encouraging results in the obfuscation technique(s) identification.**

I. Introduction

The widespread diffusion of mobile devices has induced a growing interest in malware writers, in particular for the Android operating system, that is the most widespread environment for such devices[1]. This trend has been increasing across years, from 2.3 million of new malware samples in 2015, to 3.2 millions in 2016, and 3.5 millions in 2017[2].

Conventional antimalware software are based on the *signature-based* paradigm: a threat may be recognized only if it is known [1, 2, 3]. Antimalware signatures are essentially code snippets gathered from real-world malicious samples and used by antimalware software by means of several pattern-matching heuristics applied to the code under analysis, that will be labelled as being either benign or malicious. The procedure for obtaining a signature from a malware sample is neither automatic nor instantaneous, though: security analysts need to obtain a sample of the malware, then they need to develop a suitable signature (or ascertain that the sample can be correctly labelled with the signatures already available) and finally push the new signature to all the antimalware tools, usually with an online update mechanism.

Unfortunately, once a suitable signature has been developed, malware writers can easily elude the detection by applying *obfuscation* techniques [4], that is, by modifying the malware code sufficiently enough to escape the pattern-matching heuristics of the antimalware tools, while still preserving the malicious capabilities of that code. As a matter of fact, when such techniques are able to alter to code fragments involved in the signature generation, the threat is no longer recognized as such [5, 6].

There is a proliferation of the so-called *variants* of a given malware, in particular in the Android landscape[3]: if an attacker distributes 1000 instances of a piece of malware and one of those instances is detected, then all the other instances may be detected with the same signature. But if the attacker distributes 1000 variants of a piece of malware, the likelihood that many of them will not be detected by a single signature is much higher. Strategies of this kind are attractive to attackers, because obfuscating a malware sample is much cheaper and quicker than developing and distributing a signature capable of detecting all those variants. The security analyst processes to inspect malicious behaviour usually involve a fair amount of time consuming and laborious work [7], while attackers are able to generate (often using automatic tools) a plethora of variants of a given malicious payload [8].

The main effort provided by research community in last years was aimed to malicious behaviour identification in mobile environment [9, 10, 11, 12], while the code obfuscation techniques identification is poorly investigated in literature. In this paper we propose a method to detect automatically whether a sample of code has been modified by means of one or more obfuscation techniques, focusing on code for mobile environments (Android). Our proposal is based on static analysis (i.e., the proposed method does not need to execute the application, thus the analysis does not require infecting a device) and machine learning techniques.

The paper proceeds as follows: the next section discusses the current state-of-the-art related to code obfuscation identification; Section III describes the proposed method; results of the experiments evaluation are provided in Section IV; finally, conclusions are drawn in Section V.

II. Related Work

In this section, we review the recent literature related to code obfuscation techniques identification in mobile malware landscape.

A framework able to inject a set of morphing techniques has been proposed by researchers in [1] with the aim of evaluating

[1]https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/

[2]https://nakedsecurity.sophos.com/2017/11/07/2018-malware-forecast-the-onward-march-of-android-malware/

[3]https://www.csoonline.com/article/3027598/cyber-attacks-espionage/27-of-all-malware-variants-in-history-were-created-in-2015.html

the current antimalware technologies against morphed variants of malware. The main outcome of the paper is that all the studied antimalware software are vulnerable to trivial code transformations.

Authors in [2] evaluate 10 antimalware tools using 6 original and morphed mobile malware belonging to 6 different families. The authors conclude that the antimalware are susceptible to common widespread evasion techniques. A similar analysis has been conducted in [6], considering recent Android malware detection approaches based on Machine Learning, instead of existing antimalware tools: the authors show that methods based on dynamic analysis are much more robust to obfuscation than those based on static analysis, which are, in general, more accurate on non-obfuscated samples.

Alterdroid [13] is a malware analysis framework consisting in the analysis of the behavioral differences between the original application and a set of automatically generated versions of it, where a number of modifications have been carefully injected (the so-called variants). In addition, Alterdroid performs a dynamic analysis (i.e., every app is executed over a time span equal to 120 s) to identify the malware.

Researchers in [14] investigate the optimal set of instructions being executed to identify obfuscated Android malware using the SVM classifier. They find a set of instructions that are good indicators of malware and determine how long the program needs to run in order to obtain an accurate classification. They obtain an average accuracy equal to 84.4%.

An interesting framework for the automatic generation of variants of Android applications is presented in [15], with the aim of assessing the effectiveness of current Android antimalware tools.

Beyond works that explicitly deal with obfuscated malware, there is a number of approaches for the detection of Android malware which are stated to be robust to obfuscation. Most of them are based on features extracted by means of dynamic analysis which is designed to capture the actual behavior of an application, rather than its statically viewed code. Significant examples in these respects include [16, 17, 18].

The MalDozer [16] tool considers API calls, a common kind of outcome deriving from dynamic analysis, and analyzes them with deep learning to identify Android malware. In the evaluation authors obtain an F1-Score ranging from 96% to 99% on real-world Android applications.

The RevealDroid tool [17] is stated to be obfuscation resilient thanks to a set of features including sensitive APIs and intents usage and information flows. The effectiveness of the selected features is evaluated using two different simple classifiers, which obtain an accuracy ranging between 93% and 96% in malware detection. Authors evaluate RevealDroid also in malware family identification obtaining an accuracy equal to 87% and 95% for all the combinations of features and classifiers.

Finally, the authors of [18] propose to use a wide range of metrics concerning the device resources consumption in order to discriminate between malware and benign applications, the metrics being stated to be robust to camouflage. They show that, when capturing those metrics in a precise experimental setting, very good detection effectiveness (greater than 99% on 2000 applications) can be achieved.

III. Application representation and obfuscation detection

A. The obfuscation techniques

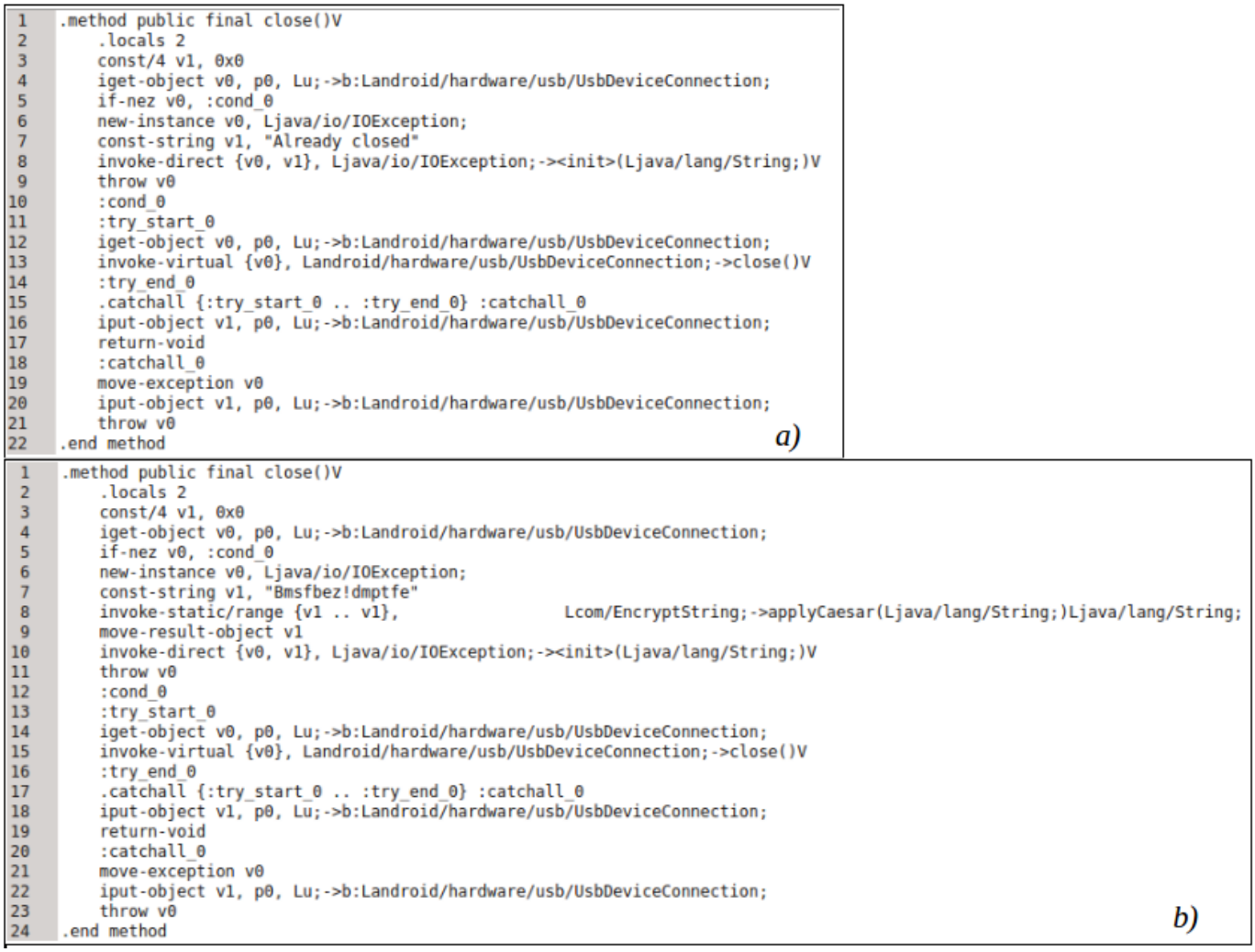
Android runs compiled Java code stored in `.dex` files targeting the *Dalvik* virtual machine. We obtained a human-readable Dalvik bytecode (the *smali* representation) from the `.dex` file of the application under analysis, using *apktool*[4], a tool for reverse engineering which allows to decompile and recompile Android applications. apktool is able to decode resources to nearly original form and rebuild them after making some modifications.

We designed, implemented, and publicly released[5] a Java tool able to apply a set of code modifications (i.e., obfuscations) to a smali representation in an automated way.

In the following we describe the considered obfuscation techniques which have been shown to be widely used by malware creators [19, 4]:

1) **Disassembling & Reassembling.** We disassemble and then reassemble with *apktool* the compiled Dalvik bytecode. This procedure modifies the ordering with the `.dex` file of several items. Signatures relying on the order of different items in the `.dex` file will likely be ineffective with this transformation.
2) **Repacking.** Every Android application contains a developer signature key that will be lost after disassembling the application and then reassembling it. In order to create a new key, we use the *signapk*[6] tool to embed a new signature key in the reassembled app to avoid detection by signatures that match the developer keys.
3) **Changing package name.** Each Android application is identified by a unique package name. This transformation is focused at renaming the application package name, using a random string generator, in both the Android Manifest and all the application classes, to elude detection by signatures based on package name.
4) **Identifier renaming.** To avoid detection signatures relying on identifier names, this transformation renames each package name and class name by using a random string generator, in both Android Manifest and smali classes, handling renamed classes invocations.
5) **Data encoding.** The `.dex` files contain all the strings and arrays used in the code. Strings could be used to create detection signatures to identify malware. To elude such signatures, this transformation encodes strings with a *Caesar cipher* with a fixed key equal to 3. This technique has also been applied to the code of the so-called metamorphic malware [20, 21]. The original string

[4] http://ibotpeaches.github.io/Apktool/
[5] Anonymized for double-blind review.
[6] https://code.google.com/p/signapk/



```
.method public final close()V
    .locals 2
    const/4 v1, 0x0
    iget-object v0, p0, Lu;->b:Landroid/hardware/usb/UsbDeviceConnection;
    if-nez v0, :cond_0
    new-instance v0, Ljava/io/IOException;
    const-string v1, "Already closed"
    invoke-direct {v0, v1}, Ljava/io/IOException;-><init>(Ljava/lang/String;)V
    throw v0
    :cond_0
    :try_start_0
    iget-object v0, p0, Lu;->b:Landroid/hardware/usb/UsbDeviceConnection;
    invoke-virtual {v0}, Landroid/hardware/usb/UsbDeviceConnection;->close()V
    :try_end_0
    .catchall {:try_start_0 .. :try_end_0} :catchall_0
    iput-object v1, p0, Lu;->b:Landroid/hardware/usb/UsbDeviceConnection;
    return-void
    :catchall_0
    move-exception v0
    iput-object v1, p0, Lu;->b:Landroid/hardware/usb/UsbDeviceConnection;
    throw v0
.end method
```

a)

```
.method public final close()V
    .locals 2
    const/4 v1, 0x0
    iget-object v0, p0, Lu;->b:Landroid/hardware/usb/UsbDeviceConnection;
    if-nez v0, :cond_0
    new-instance v0, Ljava/io/IOException;
    const-string v1, "Bmsfbez!dmptfe"
    invoke-static/range {v1 .. v1},                Lcom/EncryptString;->applyCaesar(Ljava/lang/String;)Ljava/lang/String;
    move-result-object v1
    invoke-direct {v0, v1}, Ljava/io/IOException;-><init>(Ljava/lang/String;)V
    throw v0
    :cond_0
    :try_start_0
    iget-object v0, p0, Lu;->b:Landroid/hardware/usb/UsbDeviceConnection;
    invoke-virtual {v0}, Landroid/hardware/usb/UsbDeviceConnection;->close()V
    :try_end_0
    .catchall {:try_start_0 .. :try_end_0} :catchall_0
    iput-object v1, p0, Lu;->b:Landroid/hardware/usb/UsbDeviceConnection;
    return-void
    :catchall_0
    move-exception v0
    iput-object v1, p0, Lu;->b:Landroid/hardware/usb/UsbDeviceConnection;
    throw v0
.end method
```

b)

Figure 1. An example of the application of the data encoding obfuscation technique. The box above shows a snippet of the original smali code gathered by the `.dex` of the application under analysis, while the box below shows the corresponding code snippet after the application of the data encoding technique.

will be restored during application run-time, with a call to a *smali* method that knows the *Caesar key*.

6) **Call indirections.** Some detection signatures could exploit the call graph of the application. To evade such signatures, we designed and implemented a transformation which mutates the original call graph, by modifying every method invocation in the smali code with a call to a new method inserted by the transformation which simply invokes the original method.
7) **Code Reordering.** This transformation is aimed at modifying the instructions order in smali methods. A random reordering of instructions has been accomplished by inserting `goto` instructions with the aim of preserving the original runtime execution trace. Considering that the reordering is random, this is considered the strongest obfuscation technique able to alter the signature provided by current antimalware technologies [4]. The transformation was applied only to methods that do not contain any type of jumps (i.e., `if`, `switch`, recursive calls).
8) **Junk Code Insertion.** These transformations introduce code sequences that have no effect on the business logic of applications. This is considered a weak technique: for this reason, antimalware technologies are usually able to identify samples obfuscated only with this technique [22]. The transformation provides three different types junk code insertions: (a) insertion of `nop` instructions into each method, (b) insertion of unconditional jumps into each method, and (c) allocation of three additional registers on which garbage operations are performed.

Figures 1 and 2 show two examples of the application of the Data encoding and Code Reordering obfuscation techniques, respectively.

B. Obfuscation detection method

We consider the problem of detecting code obfuscation on both trusted and malware Android apps. In particular, given an app a, we aim at identifying which of the code obfuscation

```
.method public constructor <init>(Lcom/hoho/android/usbserial/driver/FtdiSerialDriver;Landroid/hardware/usb/UsbDevice;I)V
    .locals 1
    iput-object p1, p0, Lu;->i:Lcom/hoho/android/usbserial/driver/FtdiSerialDriver;
    const/4 v0, 0x0
    invoke-direct {p0, p2, v0}, Ls;-><init>(Landroid/hardware/usb/UsbDevice;I)V
    const-class v0, Lcom/hoho/android/usbserial/driver/FtdiSerialDriver;
    invoke-virtual {v0}, Ljava/lang/Class;->getSimpleName()Ljava/lang/String;
    move-result-object v0
    iput-object v0, p0, Lu;->g:Ljava/lang/String;
    return-void
.end method
```

a)

```
.method public constructor <init>(Lcom/hoho/android/usbserial/driver/FtdiSerialDriver;Landroid/hardware/usb/UsbDevice;I)V
    .locals 1
    goto :insn_0
    :insn_1
    const/4 v0, 0x0
    goto :insn_2
    :insn_5
    move-result-object v0
    :insn_6
    iput-object v0, p0, Lu;->g:Ljava/lang/String;
    goto :insn_7
    :insn_3
    const-class v0, Lcom/hoho/android/usbserial/driver/FtdiSerialDriver;
    goto :insn_4
    :insn_2
    invoke-direct {p0, p2, v0}, Ls;-><init>(Landroid/hardware/usb/UsbDevice;I)V
    goto :insn_3
    :insn_0
    iput-object p1, p0, Lu;->i:Lcom/hoho/android/usbserial/driver/FtdiSerialDriver;
    goto :insn_1
    :insn_4
    invoke-virtual {v0}, Ljava/lang/Class;->getSimpleName()Ljava/lang/String;
    goto :insn_5
    :insn_7
    return-void
    :insn_8
    nop
.end method
```

b)

Figure 2. An example of the application of the code reordering obfuscation technique. The box above shows a snippet of the original smali code gathered by the `.dex` of the application under analysis, while the box below shows the corresponding code snippet after the application of the code reordering technique.

techniques mentioned in the previous section (if any) have been applied to a.

We adopt two different strategies to detect obfuscation: a feature-based approach, composed by a features extraction phase and subsequent application of established supervised binary classification techniques, and a direct classification approach based on Long Short-Term Memory Recurrent Neural Networks (LSTM-RNN). Both approaches are preceded by a common initial phase of pre-processing during which a is converted to sequences of opcodes through decompilation. Each phase is described in detail below.

In the *app conversion phase*, we start extracting the `.dex` file from the original `.apk` file of a. Decompiling the `.dex` file, we obtain the sequence of machine level instructions of a, each formed by an opcode and relative parameters. We discard the parameters and then segment the sequence of opcodes into subsequences, based on the Java method in a to which they belong. Each Java method thus corresponds to a subsequence of opcodes and the final representation of a is a unordered list (a bag) of sequences of opcodes.

Then, we process the resulting bag of sequences according to one of the two following approaches. In both cases, we assume that a number of samples of obfuscated and non-obfuscated apps are available.

1) Feature-based approach: We base this approach on the method proposed in [10], a supervised binary classification method which has been shown to lead to good discriminating properties for what concerns opcodes sequences, slightly modifying it to better fit the problem addressed in this work.

Given the target obfuscation technique τ that we want to detect, we say that a is positive if a has been obfuscated with τ, otherwise we say that a is negative.

In a *learning phase*, we denote the set of *positive apps* available for learning (i.e., the set of apps obfuscated with τ) as A_P and the set of *negative apps* available for learning (i.e., the set of apps not obfuscated with τ) as A_N. First, we compute the frequency $f(a, o)$ for each $n$gram of opcodes o in a, n being a parameter of the method. Each such frequency is a candidate feature. Since the number of features may be remarkably large, we perform a feature selection computing for each o its global frequencies relatively to A_P and to A_N, through the following formulas:

$$\bar{f}_P(o) = \frac{1}{|A_P|} \sum_{a \in A_P} f(a, o) \tag{1}$$

$$\bar{f}_N(o) = \frac{1}{|A_N|} \sum_{a \in A_N} f(a, o) \tag{2}$$

and then calculating the relative difference using:

$$d(o) = \frac{\text{abs}(\bar{f}_P(o) - \bar{f}_N(o))}{\max(\bar{f}_P(o), \bar{f}_N(o))} \quad (3)$$

We discard those o for which $d(o) = 1$ (i.e., those $n$grams which occur only in A_P and never occur in A_N or viceversa) to avoid obtaining a classifier that fails to generalize. Then we create the set O of selected features choosing the k $n$grams with the highest value of $d(o)$ among the remaining ones, where k is a parameter of the method, ensuring that O does not contain $n$grams that are subsequences of another $n$gram also present in O. After this feature extraction phase, we we train a binary classifier based on the frequencies in a of the features determined by O. We experimented with three popular binary classification methods, namely Support Vector Machines (SVM), Random Forest (RF), and Multi-Layer Perceptron (MLP). The outcome for this phase is hence, for each obfuscation technique τ, (a) the set O of relevant $n$grams and (b) the trained classifier.

In the *prediction phase*, we consider the bag of sequences of the application a being analyzed and, for each obfuscation technique τ, (i) we extract the features (i.e., frequences of the $n$grams determined by O), and (ii) we apply the learned classifier obtaining a binary prediction, meaning that a is deemed to be (positive) or not to be (negative) obfuscated with the technique τ.

2) Direct classification approach: In this approach, a further preprocessing step is performed on each app both in the learning and the prediction phase, the preprocessing consisting in obtaining a single opcode sequence by concatenating all the sequences in the app bag of sequences (in the order resulting from the app conversion phase).

In the learning phase, we build a set A of labeled apps where each app a is associated with n_τ binary labels $l_1, \ldots, l_{n_\tau}$, where $l_i = 1$ if a has been obfuscated (at least) with the ith technique τ_i, and $l_i = 0$, otherwise. Then, we build a larger set A' starting from A such that for each element $\langle a, (l_1, \ldots, l_{n_\tau})\rangle \in A$, there are one or more elements $\langle a_j, (l_1, \ldots, l_{n_\tau})\rangle \in A$, where a_j is the jth chunk of the concatenated opcode sequence of a, with chunks length equal to l, l being a parameter of the method—the binary labels do not change. Then, we train a LSTM-RNN on A': the network has an output layer of n_τ neurons, each representing an obfuscation technique. In other words, just one classifier (the trained LSTM-RNN) can detect the obfuscation by means of zero or more of the n_τ techniques. Moreover, since the LSTM-RNN operates directly on the opcode sequence, there is no need for feature selection.

In the prediction phase, after the same preprocessing described above, we apply the trained LSTM-RNN to the app concatenated opcode sequence, thus obtaining n_τ predictions corresponding to the n_τ obfuscation techniques.

IV. Experimental evaluation

A. Data

We performed an experimental evaluation of the proposed approaches using a dataset consisting of 6600 apps. The used dataset is a subset of the one used in [10] and contains 3300 Android malware apps and 3300 trusted (i.e., non malware) apps. We included also malware apps in the dataset because we wanted to assess the obfuscation detection ability of the proposed approaches, regardless of the fact that the analyzed app is malware or not.

We separately applied several obfuscation techniques (through the tool introduced in the previous section) to each app, obtaining one version of the app per obfuscation technique. The obfuscation techniques we actually used are a subset of 6 techniques from those presented in Section III-A, in particular:

1) Changing package name
2) Identifier renaming
3) Data encoding
4) Call indirections
5) Code reordering
6) Junk code insertion

The remaining two techniques (Disassembling & reassembling and Repacking) do not modify the code itself and are implicitly performed during the application of the other techniques, so we did not include them as target techniques in our dataset.

The resulting dataset consists of 52 800 apps, and includes, for each of the 6600 original apps, the non-obfuscated version of the app, a version obfuscated by applying all of the 6 techniques listed above, and 6 other obfuscated versions, one for each obfuscation technique.

B. Experimental procedure

In this section, we present methodology and results of our experiments in identifying obfuscation techniques on Android apps for the proposed approaches.

In our feature-based approach, we experimented with three different classifiers: SVM with Gaussian kernel and $c = 1$, RF with $n_{\text{tree}} = 500$, and MLP with 4 hidden layers. Concerning the feature extraction, we set the opcode length to $n = 3$ and the number of features $k = 5000$, for SVM and RF; for MLP, we experimented varying the value of k in $[250, 5000]$.

In the direct classification approach, we used a LSTM-RNN with a single LSTM layer of 60 neurons followed by two fully-connected hidden layers and the output layer; we trained the network for one epoch using a batch size of 1024. We set the length of each chunk to $l = 200$ to keep a reasonably long sequence of opcodes without exceeding the memory capability of the LSTM cells.

We assessed our approaches detection ability separately for each technique. In each case, we performed a 10-fold cross-validation, that is, we split our dataset in 10 segments (evenly distributing positive and negative apps in each segment) and repeated our experiments 10 times, choosing each time a different segment as testing set and the union of the remaining

Table I
EFFECTIVENESS OF THE PROPOSED APPROACHES IN TERM OF AUC AND EER (AVERAGE AND STANDARD DEVIATION ACROSS THE 10 FOLDS) IN IDENTIFYING DIFFERENT OBFUSCATION TECHNIQUES, CONSIDERING THE CASE OF LEARNING SET WITH APPS OBFUSCATED WITH *All techniques* (LEFT) OR WITHOUT THOSE APPS (RIGHT).

		w/ *All techniques*		w/o *All techniques*	
Classifier	Technique	AUC	EER	AUC	EER
SVM	Any technique	0.69±0.01	0.35±0.01	0.63±0.04	0.40±0.04
	Junk code	0.94±0.05	0.13±0.11	0.90±0.10	0.16±0.14
	Identifier renaming	0.72±0.11	0.36±0.07	0.68±0.12	0.38±0.08
	Data encoding	0.98±0.01	0.06±0.01	0.94±0.03	0.09±0.03
	Code reordering	0.93±0.05	0.14±0.09	0.85±0.10	0.20±0.12
	Change package name	0.71±0.11	0.36±0.07	0.51±0.09	0.48±0.04
	Call indirection	0.89±0.10	0.16±0.14	0.86±0.12	0.21±0.15
RF	Any technique	0.84±0.02	0.24±0.02	0.84±0.02	0.24±0.02
	Junk code	0.97±0.02	0.07±0.06	0.89±0.10	0.15±0.14
	Identifier renaming	0.82±0.08	0.27±0.06	0.61±0.02	0.41±0.01
	Data encoding	1.00±0.00	0.01±0.01	0.96±0.02	0.07±0.02
	Code reordering	0.97±0.02	0.08±0.05	0.87±0.11	0.18±0.14
	Change package name	0.82±0.08	0.27±0.06	0.61±0.02	0.40±0.02
	Call indirection	0.94±0.06	0.11±0.10	0.84±0.10	0.22±0.12
MLP	Any technique	0.83±0.01	0.25±0.01	0.84±0.01	0.25±0.01
	Junk code	0.97±0.02	0.07±0.06	0.90±0.10	0.16±0.14
	Identifier renaming	0.80±0.06	0.30±0.04	0.59±0.07	0.44±0.05
	Data encoding	0.99±0.01	0.03±0.01	0.78±0.16	0.27±0.19
	Code reordering	0.97±0.03	0.09±0.05	0.77±0.10	0.30±0.11
	Change package name	0.80±0.06	0.30±0.04	0.57±0.09	0.44±0.05
	Call indirection	0.92±0.08	0.13±0.12	0.70±0.07	0.39±0.05
LSTM-RNN	Any technique	0.83±0.07	0.23±0.07	0.85±0.08	0.20±0.09
	Junk code	0.97±0.01	0.06±0.01	0.95±0.02	0.08±0.02
	Identifier renaming	0.72±0.03	0.34±0.01	0.52±0.14	0.49±0.11
	Data encoding	0.74±0.04	0.33±0.03	0.56±0.15	0.48±0.11
	Code reordering	0.79±0.06	0.28±0.06	0.65±0.09	0.41±0.09
	Change packet name	0.72±0.03	0.34±0.02	0.52±0.14	0.49±0.12
	Call indirection	0.73±0.03	0.33±0.01	0.56±0.12	0.48±0.12

ones as training set. We measured the effectiveness in terms of global *Area Under the ROC Curve* (AUC) and *Equal Error Rate* (EER). The ROC (Receiver Operating Characteristic) curve is created by varying the threshold of a binary classifier and plotting the resulting true positive rate (TPR) against the false positive rate (FPR). Starting from the ROC curve, AUC is calculated simply computing the area under the curve. EER is obtained equalizing the false positive rate and false negative rate (FNR), where FNR $= 1 -$ TPR.

In order to gain more insights, we also introduced an addition classification task, denoted by *Any technique*, in which the positives are the apps obfuscated with at least one technique.

We executed two full suites of experiments, with different compositions of the training set. A first suite with the training set constructed with the full dataset and a second suite in which we removed from the training set all the apps obfuscated with all the obfuscation techniques. This choice allowed us to explore a more specific classifier trained on a single technique and therefore without any knowledge of the other techniques.

For what concerns learning time, we measured a total learning time per fold in the order of minutes for MLP and RF, of several hours for SVM, and of a few days for LSTM-RNN. Concerning the prediction time, MLP and RF took less then $1\,\mathrm{s}$ to classify the entire testing set, while SVM and LSTM-RNN took tens of seconds. We conducted our experiments on a machine with 8 core running at $2.4\,\mathrm{GHz}$ and $32\,\mathrm{GB}$ of RAM.

C. Results and discussion

Table I shows the results achieved by the various classifiers for each obfuscation technique. The table shows the average value of the index (AUC or EER) and its standard deviation across the 10 folds.

There are three main observations which can be made based on the results in Table I. First, there are some obfuscation techniques which our approaches, in general, are more effective in detecting than others. In particular, the Data encoding technique appears to be easily detected by all the approaches, with RF trained including All techniques samples reaching an AUC of 1.0, which corresponds to an EER of 0.01: MLP and SVM reach AUC $= 0.99$ (EER $= 0.03$) and AUC $= 0.98$ (EER $= 0.06$), respectively. On the other hand, Identifier renaming seems to be the obfuscation technique which is harder to detect.

Second, figures show that the inclusion in the learning data of sample of applications obfuscated with all the techniques is in general beneficial. In particular, the AUC obtained with All techniques is always greater than without All techniques, for all the approaches and all the techniques with the exception of

Any technique. We think that this finding suggests that, despite the fact that including samples which are obfuscated with more than one technique may confuse the classifier, the number of different samples in the learning data plays a crucial role: by including All techniques samples, we essentially double the number of positive samples, hence making the learning data larger and, likely, more representative of the unknown data to be classified in the testing phase. Accordingly, this improvement does not manifest in the case of Any technique, for which the learning set is large enough also w/o including the All technique samples.

Third, Table I shows that there are differences among the 4 considered approaches. RF appears to be the best performing one; yet, the difference with MLP is negligible. A larger difference in both AUC and EER is instead observable w.r.t. LSTM-RNN. We argue that a better effectiveness detection could be obtained with a finer tuning of the parameters, including the number of epochs of training: indeed, however, we observed in our experiments that only small improvements were obtained after the batches corresponding to the first epoch.

In Figure 4, we show the box plots for the AUC values which are summarized in Table I. Besides confirming the previous finding, Figure 4 shows that there are some techniques for which all the approaches exhibit a rather wide interval of AUC values (e.g., Identifier renaming). This finding may corroborate our previous reasoning about the importance of the amount of data available for learning, at least for some technique: more or less fortunate conditions result in greater or lower detection effectiveness in the testing phase.

Finally, in order to better investigate the effectiveness of the feature-based approaches, we performed a further suites of experiments by varying the value for the parameter k (i.e., the number of $n$grams actually considered, see Section III-B1) when coupled with the MLP classifier. The results are shown in Figure 4, which plots the AUC vs. the value of k.

It can be seen that k has a different impact on the effectiveness of detection of the various obfuscation techniques. In general, the greater k, the better. Yet, it can be seen that for Junk code and Code reordering, negligible improvements are obtained with $k > 1000$. Interestingly, the former technique is the one for which LSTM-RNN is the closer to the best approach (RF): this may suggest that different approaches are appropriate for different techniques.

V. Concluding remarks and future work

Generating a signature from malicious samples is a laborious and time-consuming process for malware analysts. Furthermore, even once a threat is identified, malicious writers can construct many variants of a given malware quickly and cheaply by applying several obfuscation techniques. Detecting all such variants with a single signature is extremely hard.

In this paper we proposed a method aimed at identifying automatically whether a sample under analysis has been modified by means of obfuscation techniques. The method is based on established supervised binary classification techniques (SVM, Random Forest, and Multi-Layer Perceptron) operating on ad hoc features deriving from static analysis, which have already been shown to be suitable for classification of Android malware; as a second option, we also explored the usage of Long Short-Term Memory Recurrent Neural Networks, which can operate directly on the sequences of opcodes.

We evaluated the proposed method on a dataset of real-world Android applications modified by means of the following obfuscation techniques: changing package name, identifier renaming, data encoding, code reordering and junk code insertion. The results are promising: for some techniques, almost perfect detection is possible (e.g., Equal Error Rate ≤ 0.01 with Random Forest for detecting the Data encoding technique), whereas for others the detection looks harder.

References

[1] 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*. ACM, 2013, pp. 329–334.

[2] ——, "Catch me if you can: Evaluating android anti-malware against transformation attacks," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 1, pp. 99–108, 2014.

[3] G. Canfora, F. Mercaldo, G. Moriano, and C. A. Visaggio, "Composition-malware: building android malware at run time," in *Availability, Reliability and Security (ARES), 2015 10th International Conference on*. IEEE, 2015, pp. 318–326.

[4] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*. IEEE, 2010, pp. 297–300.

[5] G. Canfora, A. Di Sorbo, F. Mercaldo, and C. A. Visaggio, "Obfuscation techniques against signature-based detection: a case study," in *Mobile Systems Technologies Workshop (MST), 2015*. IEEE, 2015, pp. 21–26.

[6] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Impact of code obfuscation on android malware detection based on static and dynamic analysis," in *4th International Conference on Information Systems Security and Privacy*. Scitepress, 2018, pp. 379–385.

[7] F. Mercaldo, V. Nardone, and A. Santone, "Ransomware inside out," in *Availability, Reliability and Security (ARES), 2016 11th International Conference on*. IEEE, 2016, pp. 628–637.

[8] A. Cimitile, F. Martinelli, F. Mercaldo, V. Nardone, and A. Santone, "Formal methods meet mobile code obfuscation identification of code reordering technique," in *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2017 IEEE 26th International Conference on*. IEEE, 2017, pp. 263–268.

[9] S. Wang, Q. Yan, Z. Chen, B. Yang, C. Zhao, and M. Conti, "Detecting android malware leveraging text

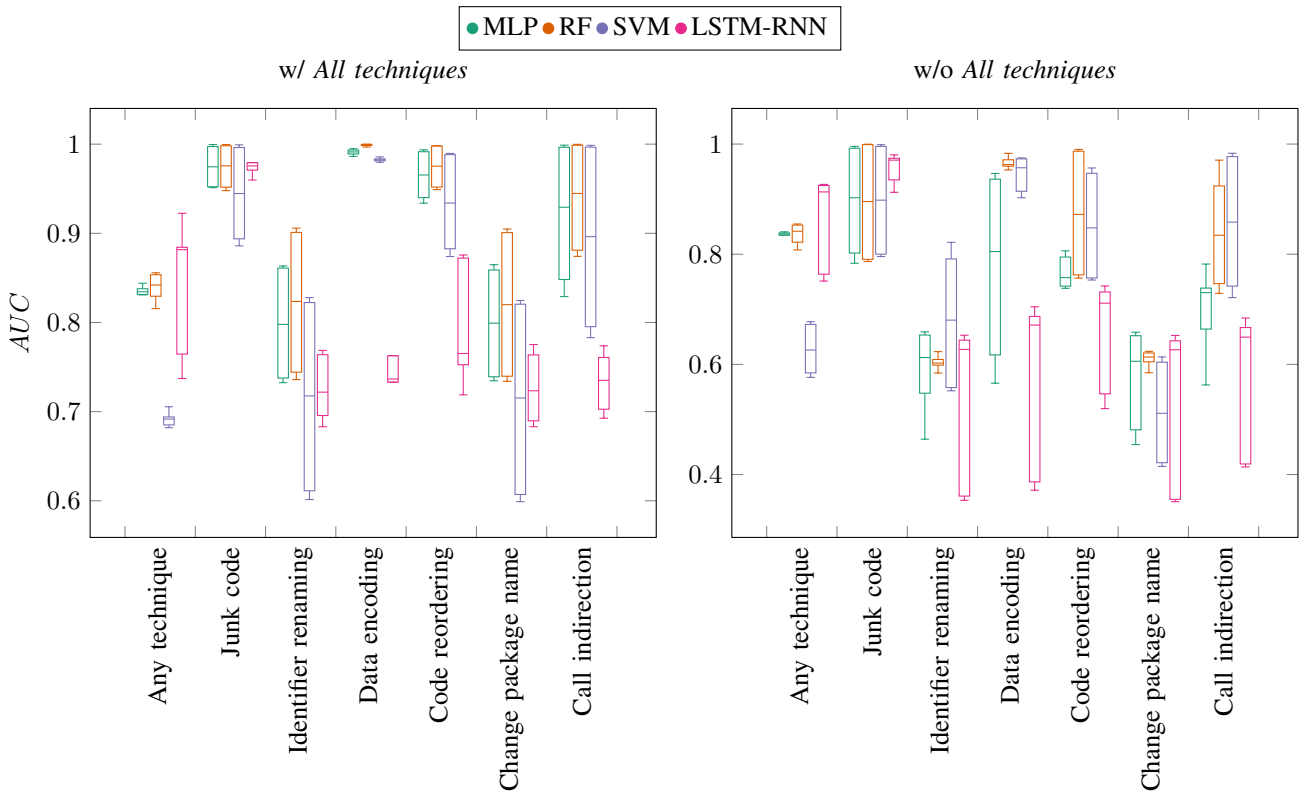


Figure 3. Box-plots of effectiveness per fold and target technique of Multilayer Perceptron, RF, and Support Vector Machine, with (left) and without (right) *All Techniques* apps in training set

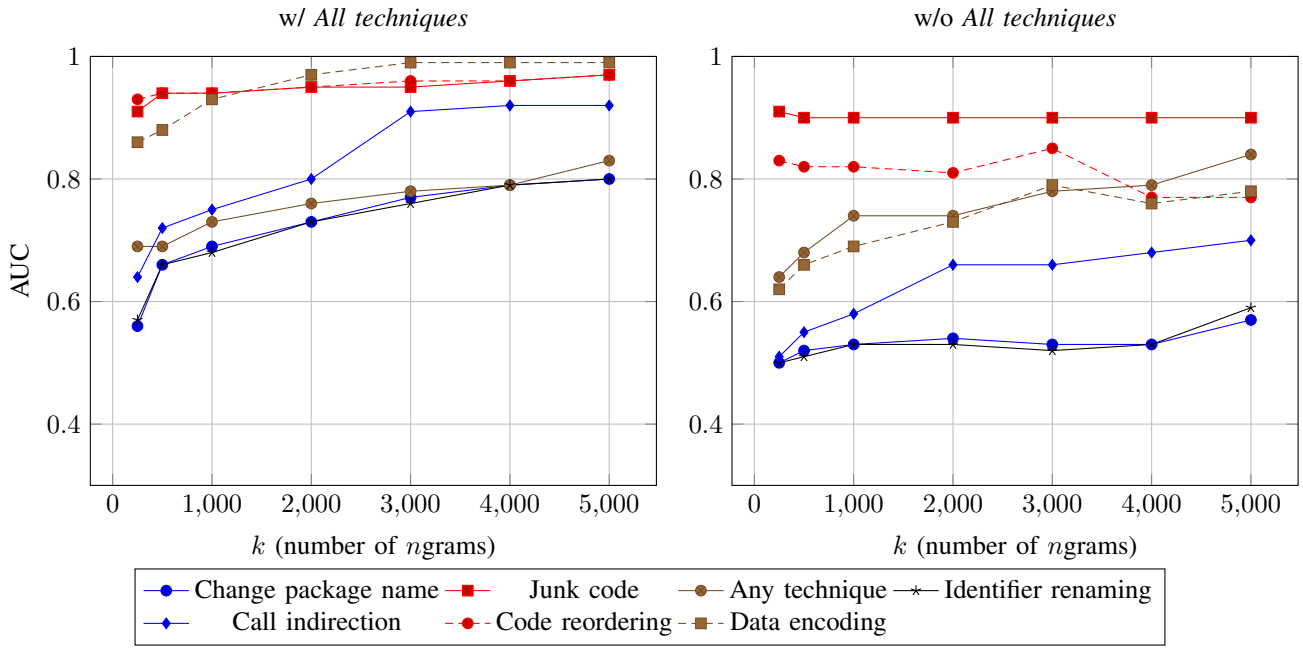


Figure 4. Variation of AUC scores of the Multilayers Perceptron classifiers in based on the total length of the feature vector

semantics of network flows," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1096–1109, 2018.

[10] G. Canfora, A. De Lorenzo, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Effectiveness of opcode ngrams for detection of multi family android malware," in *Availability, Reliability and Security (ARES), 2015 10th International Conference on*. IEEE, 2015, pp. 333–340.

[11] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye, "Android malware detection," *IEEE Transactions on Industrial Informatics*, 2018.

[12] D. Maiorca, F. Mercaldo, G. Giacinto, C. A. Visaggio, and F. Martinelli, "R-packdroid: Api package-based characterization and detection of mobile ransomware," in *Proceedings of the Symposium on Applied Computing*. ACM, 2017, pp. 1718–1723.

[13] G. Suarez-Tangil, J. E. Tapiador, F. Lombardi, and R. Di Pietro, "Alterdroid: differential fault analysis of obfuscated smartphone malware," *IEEE Transactions on Mobile Computing*, vol. 15, no. 4, pp. 789–802, 2016.

[14] P. O'kane, S. Sezer, and K. McLaughlin, "Detecting obfuscated malware using reduced opcode set and optimised runtime trace," *Security Informatics*, vol. 5, no. 1, pp. 1–12, 2016.

[15] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang, "Auditing anti-malware tools by evolving android malware and dynamic loading technique," *IEEE Transactions on Information Forensics and Security*, 2017.

[16] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Maldozer: Automatic framework for android malware detection using deep learning," *Digital Investigation*, vol. 24, pp. S48–S59, 2018.

[17] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek, "Obfuscation-resilient, efficient, and accurate detection and family identification of android malware," *Department of Computer Science, George Mason University, Tech. Rep*, 2015.

[18] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Acquiring and analyzing app metrics for effective mobile malware detection," in *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*. ACM, 2016, pp. 50–57.

[19] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 290–299.

[20] J.-M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *Journal in Computer Virology*, vol. 4, no. 3, pp. 211–220, 2008.

[21] G. Canfora, F. Mercaldo, C. A. Visaggio, and P. Di Notte, "Metamorphic malware detection using code metrics," *Information Security Journal: A Global Perspective*, vol. 23, no. 3, pp. 57–67, 2014.

[22] C. S. Collberg, C. D. Thomborson, and D. W. K. Low, "Obfuscation techniques for enhancing software security," Dec. 23 2003, uS Patent 6,668,325.