Mirzaei, O., Fuentes, J.M., Tapiador, J. González-Manzano, L. (2019). AndrODet: An adaptive Android obfuscation detector. *Future Generation Computer Systems*, 90, pp. 240-261.

DOI: 10.1016/j.future.2018.07.066

Future Generation Computer Systems 90 (2019) 240–261

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

AndrODet: An adaptive Android obfuscation detector

O. Mirzaei *, J.M. de Fuentes, J. Tapiador, L. Gonzalez-Manzano

Computer Security Lab (COSEC), Universidad Carlos III de Madrid, Av. Universidad, 30. ES-28911 Leganes, Spain

HIGHLIGHTS

- An online learning system to detect 3 types of obfuscation in Android applications.
- ID-renaming detection module identifies obfuscated apps after observing few samples.
- String encryption detection module improves its accuracy by observing few apps.
- Control flow obfuscation detection module reaches a good accuracy from few seen apps.
- The proposed system is compared with a batch-learning equivalent by time and memory.

ARTICLE INFO

Article history:
Received 17 April 2018
Received in revised form 21 June 2018
Accepted 28 July 2018
Available online 6 August 2018

Keywords:
Obfuscation detection
Android
Machine learning
Malware

ABSTRACT

Obfuscation techniques modify an app's source (or machine) code in order to make it more difficult to analyze. This is typically applied to protect intellectual property in benign apps, or to hinder the process of extracting actionable information in the case malware. Since malware analysis often requires considerable resource investment, detecting the particular obfuscation technique used may contribute to apply the right analysis tools, thus leading to some savings.

In this paper, we propose AndrODet, a mechanism to detect three popular types of obfuscation in Android applications, namely identifier renaming, string encryption, and control flow obfuscation. AndrODet leverages online learning techniques, thus being suitable for resource-limited environments that need to operate in a continuous manner. We compare our results with a batch learning algorithm using a dataset of 34,962 apps from both malware and benign apps. Experimental results show that online learning approaches are not only able to compete with batch learning methods in terms of accuracy, but they also save significant amount of time and computational resources. Particularly, AndrODet achieves an accuracy of 92.02% for identifier renaming detection, 81.41% for string encryption detection, and 68.32% for control flow obfuscation detection, on average. Also, the overall accuracy of the system when apps might be obfuscated with more than one technique is around 80.66%.

1. Introduction

The widespread usage of smartphones in various security-sensitive operations in recent years, such as bank transactions and online payments [1], requires that the security of these platforms must be improved. This affects particularly to smartphones hosting Android applications, as they have the biggest world-wide market share [2]. More specifically, in recent years where re-packaging popular smartphone banking applications has raised in number [3], hardening apps against reverse engineering has become increasingly important.

Source code is an important intellectual property for both legitimate software developers and malware writers; specifically, in Android operating system where the applications can be easily decompiled for automated code analysis or visual inspection. In the legitimate context, obfuscation prevents the competitors from cloning or copying the source code with little effort and just by adding very few extra features, while in a non-legitimate context, it hides the apps' semantics from analysts by increasing the cost of reverse engineering and decompilation.

Obfuscation has been vastly applied to both malware and benign Android applications in the last years [4]. In particular, three types of obfuscation have been used, including identifier renaming, string encryption, and control flow obfuscation mainly because they are either available in free obfuscators or in the trial versions of commercial obfuscators. Also, they create a satisfactory level of confusion in the app's source code. Based on previous researches [4], malware writers prefer to make use of more complex renaming policies than legitimate software developers. Also, string encryption is more popular in malware than in benign apps. Finally, although control-flow obfuscation is only offered by few

* Corresponding author.
E-mail addresses: omid.mirzaei@uc3m.es (O. Mirzaei), jfuentes@inf.uc3m.es (J.M. Fuentes), jestevez@inf.uc3m.es (J. Tapiador), lgmanzan@inf.uc3m.es (L. Gonzalez-Manzano).

https://doi.org/10.1016/j.future.2018.07.066

commercial obfuscators, its prevalence and detection has not been studied before.

Prevalent usage of obfuscation in Android malware has also cast doubt on the reliability of most Android malware analysis tools [5,6], and, in particular, static ones. The majority of these tools rely upon some static features which are obtained from the source code and are severely impacted by little transformations in the source code [6]. Consequently, they are not resilient to transformation attacks. Also, obfuscation has turned out to be a new barrier to protect Android users [7], and, therefore, detecting obfuscation is critical in understanding the underlying semantics of malware specimens.

Previous works leverage on batch learning systems to detect obfuscation. Thus, after extracting a set of features from the apps pooled as training set, a system is trained to detect one or more types of obfuscation [4,8]. While these systems offer promising accuracy rates, they do have a major drawback. Systems which work based on batch learning do not necessarily remain effective over time — when new applications appear or when novel obfuscation techniques are proposed. Thus, they must be eventually re-trained with the updated dataset. This task is not feasible in a setting where apps are developed and introduced constantly (as it currently happens in both Android malware and benign apps). Also, most of the recent works have tried to detect trivial types of obfuscation on a small dataset of apps. Finally, advanced obfuscation techniques such as control flow obfuscation has not been addressed based on a representative recent malware dataset [8].

To overcome these limitations, in this paper, we explore the use of online learning algorithms through Data Stream Mining (henceforth DSM) [9]. DSM can be seen as an adaptation of traditional machine learning methods so as to be suitable for streams of elements. Remarkably, DSM approaches do not need to be re-trained, as they continuously learn from the input samples. Leveraging DSM, we aim to detect basic forms of obfuscation (particularly, identifier renaming and string encryption), as well as the non-trivial control flow obfuscation. To assess our approach, we consider a dataset of 34,962 samples from both malware and benign applications.

Overview of our system In this work, we propose AndrODet, an online learning system to detect three common types of obfuscation techniques in Android applications, known as identifier renaming, string encryption, and control flow obfuscation. All of these obfuscation techniques are detected based on some static, quick-to-obtain features extracted from the Dalvik executable bytecode of applications. AndrODet is modular, meaning that there is a separate embedded module within the system to detect each type of obfuscation, and each of these modules are trained separately.

AndrODet has been implemented in python and tested on a combination of malware and benign samples. The former set of apps are collected from a recently released and carefully-labeled malware dataset, called *AMD* [10], while the latter are obtained by crawling the popular open-source repository of benign apps known as F-Droid [11]. We have also compared our results with state-of-the-art batch learning algorithms by leveraging Auto Tune Models (ATM) [12], a system developed for hyper-parameter tuning of batch learning algorithms and classification using a variety of algorithms from this kind.

Experimental results show that online learning algorithms can detect three popular types of obfuscation techniques in Android applications with high accuracy. In addition, they can save significant amount of time and memory as compared to batch learning algorithms.

Contributions In short, the main contributions of this paper are as follows:

- We propose AndrODet, a modular online learning mechanism to detect identifier renaming, string encryption, and control flow obfuscation in Android applications. To allow future works benefit from this research, we make our tool publicly available at: https://github.com/OMirzaei/AndrODet
- As AndrODet is based on DSM techniques, there is no need to re-train the system from scratch. Thus, we compare the effectiveness of our system with machine learning algorithms working based on batch learning. To do this, we leverage *MOA* [9] and add some extra features to this tool for hyper-parameter tuning which will be used later for classification. This enables us to have a fair comparison between the results obtained from online learning algorithms using *MOA* and the ones which are obtained from batch learning methods using *ATM*.
- AndrODet is able to deal with multidex Android applications. Our system looks for all classes.dex files in different directories and extracts its features from all of them.
- We assess the efficiency of our tool with *AMD* [10] and PraGuard [13]. Both datasets, with more than 24 k apps in total, contain ground truth for apps which are obfuscated by identifier renaming and string encryption techniques. Moreover, to create ground truth for control flow obfuscated apps which was previously lacking, we have leveraged a well-known obfuscator known as Allatori [14] and have obfuscated all the samples of F-Droid [11], a free and open source Android applications repository. We aim at publicly releasing the latter set of apps to foster further research in this direction.

Organization The remainder of this paper is as follows. Section 2 introduces some basic concepts as background. Section 3 describes the proposed system. Evaluation results are presented in Section 4 followed by a discussion in Section 5. Section 6 surveys some related works, and, finally, Section 7 concludes the paper and presents future research directions.

2. Background

In this Section, we introduce the main concepts and techniques related to our work, namely the Dalvik bytecode (Section 2.1), common types of obfuscation in Android (Section 2.2), and relevant details to data mining and machine learning (Section 2.3).

2.1. Dalvik bytecode

Android programs are written mostly in Java, although they can contain calls to binaries and other shared libraries known as native components [15]. Once written, they are compiled to Java bytecode and, then, to Dalvik bytecode. The final result is a Dalvik EXecutable (DEX) file with a .dex format or an optimized version of it with an .odex format.

The Dalvik Virtual Machine (DVM) is a register-based machine which executes Dalvik bytecode instructions (through a shared library, called libdvm.so) and provides a Java-level abstraction for the Java components of applications [16], while Java Native Interface (JNI) supports the use of native components. DVM is based on Just-in-Time (JIT) compilation and is replaced by Android RunTime (ART) after Android version 4.4, which works based on Ahead-Of-Time (AOT) compilation and has led to significant improvements in performance and memory consumption [17].

Analyzing Dalvik bytecode is simpler than machine code, it has a better readability for human analysts, and it provides better semantic information. Also, it is easy to be reverse engineered using tools like Dexdump [18], Dex2jar [19], Androguard [20], and Apktool [21] to name a few. Thus, many static malware analysis

tools [22], deobfuscators [4,23], and unpackers [24,25] have been proposed which extract their features directly from Dalvik bytecode. For instance, key program features such as method names, class names, field names, variables, and strings are very quick to obtain from the .dex file and give useful preliminary information. Fig. 1 provides a Dalvik bytecode snapshot from a malicious app which belongs to the FakeInstaller family. As it can be seen, constant strings and some useful information about identifiers are easily obtainable by parsing this bytecode.

2.2. Obfuscation in Android

Obfuscation is commonly used to protect software against reverse engineering, thus making the software harder to understand [26]. There are multiple obfuscation techniques [27]. In this work we focus on three well-known obfuscation techniques that are commonly applied to Android applications, namely identifier renaming, string encryption, and control flow obfuscation [27,28].

A common practice in programming is to choose meaningful names for identifiers (i.e, variables, class and method names, etc.) to increase the code readability. This will helps in identifying and fixing bugs or adding extra features later, as understanding the semantics of code with meaningful identifiers is much simpler. However, malware writers try to choose either meaningless names for their identifiers or else use obfuscators in order to garble the key identifiers used in their source code. Obfuscators use a variety of methods to rename key identifiers of an application either at the source code level or directly in the .dex files. An obfuscated identifier can be often told apart visually from a non-obfuscated one because its name is meaningless. For example, a common renaming strategy is to choose random short strings in lexicographic order, e.g., 'a', 'b', 'aa', 'ab', 'ac', etc., usually with lengths less than 3 depending on the number of identifiers. A second strategy is to leverage the overloading feature of Java through excessive overloading and map irrelevant identifier names to the original ones.

By doing so, reverse engineers need to put much more effort into understanding the hidden semantics of code when critical information such as method names are obscured. Based on a recent study [4], the prevalence of identifier renaming is slightly less in malware than in benign apps from third-party markets. Also, malware authors tend to use more complex renaming policies, such as using special characters (e.g., encoded in Unicode), which creates challenges for systems which are developed to detect this type of obfuscation.

Constant strings can also leak sensitive and private source code information. Thus, they are encrypted in different ways to prevent a convenient reverse analysis of applications. The most simplest way to encrypt encryption is through an XOR operation. However, standard cryptographic algorithms can be applied, including AES or DES [29]. Also, secret keys can be defined (or either changed) dynamically to apply more advanced types of string obfuscation, which is almost impossible to be handled by static analysis tools. Studies show that string encryption is more popular in malware and nearly all benign apps do not make use of this type of obfuscation.

Control flow obfuscation hinders static analysis by changing the logical flow of the program through modifications in its Control Flow Graph (CFG). Typical techniques from this category try to expand or flatten the CFG in order to increase the cost of reverse engineering of applications. Common ways to do this include injecting dead (or irrelevant) code, extending loop conditions, adding redundant operations, parallelizing code, re-ordering statements, loops, and inserting opaque predicates. The majority of these approaches affect the some properties of he CFG, such as the number of nodes and branches. Based on recent observations, control flow obfuscation is not widely used, and it is only offered by a few number of commercial obfuscators such as Allatori [14] and DashO [30].

2.3. Data mining and machine learning

Although data mining and machine learning share some concepts, they are different in a few major aspects. Generally, data mining is defined as the process of discovering hidden patterns from a big amount of data, or, in other words, getting some insights about the data stored in databases [31]. The data can be stored electronically, and the search for patterns is commonly automated by computer. On the other hand, machine learning is usually defined as the process of learning from previous observations [32]. In most cases, new information is learned after exploring meaningful patterns from previous seen data obtained by trying various methods of data mining. Data mining has been used in a variety of domains, including many areas in cybersecurity [31], and, specifically, in malware detection [33].

Traditional data mining algorithms need to have the whole set of past observations (referred to as the training set) to discover interesting patterns and will be used later by machine learning algorithms to predict future observations. Thus, to explore new patterns from a new set of observations, they need to be re-run. However, with the emergence of new devices and technologies, and the amount and frequency of data generated by them such as smartphones and the Internet-of-Things (IoT), traditional data mining algorithms cannot be applied efficiently as they need to be repeated in short intervals that is not feasible at a low cost.

Continuous and fast streams of data introduce big challenges to traditional data mining algorithms in particular, and machine learning methods working based on them in general. Some of these challenges include but not limited to concept drift [34], feature drift [35], temporal dependencies [36], and restricted resources requirements, both in time and memory. In addition, typical issues known in traditional data mining and machine learning algorithms, including non-representativeness of training dataset, missed feature values, underfitting, overfitting, and irrelevant features may be found here. Thus, several attempts have been made in recent years to introduce new methods for handling data streams.

DSM is a variation of traditional mining techniques which tries to explore patterns from continuously and rapidly evolving data. The two approaches are similar in terms of predicting a label for new upcoming instances represented by a number of features known as feature vector. However, DSM methods build their models from an incrementally growing pool of training instances in contrast with a large static training dataset which is commonly used by traditional data mining algorithms [37]. Therefore, all machine learning methods which are based on traditional data mining are known as batch learning algorithms, and the ones which make use of data stream mining are referred to as online learning algorithms. Due to the extensive application areas of DSM, several tools have been developed, including Massive Online Analysis (*MOA*) [38], Scalable Advanced Massive Online Analysis (SAMOA) [39], Advanced Data Mining and Machine Learning System (ADAMS) [40], JUBATUS [41], Vowpal Wabbit [42], StreamDM [43].

Online learning algorithms update their models over time (incremental learning) based on new coming instances compared to batch learning methods that keep their built model static once it is extracted. Therefore, online learning can save a significant amount of computational resources, and, also, the time which is taken for extracting the models. Furthermore, online learning algorithms do not require to decide on the number of instances to be used for training which is critical in the performance of batch learning algorithms. In return, they split the stream into disjoint chunks of

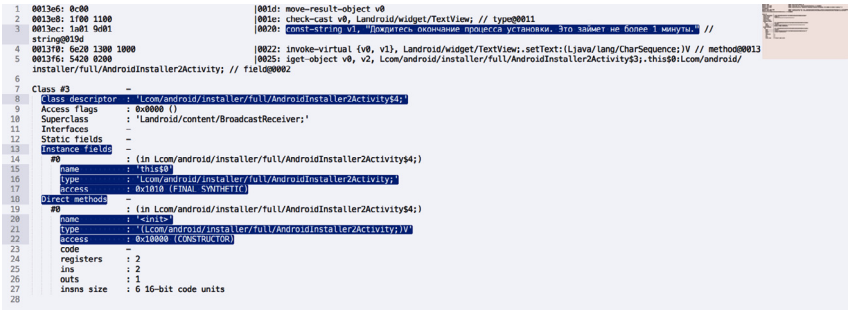


Fig. 1. A snapshot of Dalvik bytecode for an app from the FakeInstaller family.

data known as landmark windows. A landmark can be defined as the number of observed instances up to the moment. Thus, once a new landmark is reached all past instances are discarded. Another strategy is to discard one instance at a time which is done by sliding windows.

3. Approach

This section presents our approach to detect three types of obfuscation techniques in Android applications. A general overview of the system is proposed in Section 3.1. Then, primary goals are clearly defined in Section 3.2. In Section 3.3, we describe all the details related to the datasets which are used in this work. The set of all features considered for our detectors and possible feature selection algorithms are discussed in Section 3.4. Finally, classification algorithms chosen for our online learning system and their hyper parameter tuning are presented in Section 3.5.

3.1. Overview

AndrODet is an online learning system which is developed to detect three main types of obfuscation in Android applications, namely identifier renaming, string encryption, and control flow obfuscation. Also, it can detect obfuscation in Multidex Android applications. Android RunTime (ART) which is used in Android 5.0 (API level 21) and higher supports loading multiple Dalvik EXecutable (DEX) files from APK files. It then performs pre-compilation at install time and scans for all classes.dex files to compile them into a single .oat file. This feature enables applications to distribute their code into several .dex files. Specific Android malware variants have also been observed which load their malicious .dex file from a secondary directory (e.g. assets directory) [44,45]. AndrODet searches for all classes.dex files in different directories and extracts its features from all of them.

The proposed system is modular, i.e., there is an embedded module (binary classifier) to detect each type of obfuscation as shown in Fig. 2(a). Using a modular architecture has three main advantages. First, it reduces feature overlap, and, thus, improves the precision accuracy. Second, the system can be easily updated with a new set of features for each module based on variations in obfuscation techniques. Third, different learning algorithms can be used for each module based on the nature of the input data.

To label new unseen apps, all required features are extracted by each module and a feature vector is created at the first step, as depicted in Fig. 2(b). A binary classifier is then chosen to decide whether or not the app is obfuscated. These classifiers are trained incrementally using online learning algorithms while labeling new applications.

3.2. Goals

AndrODet is intended to achieve the following main goals:

- **Rapidity**. The system must be able to work in a reduced amount of time.
- **Readiness**. The system must be ready to work with moderate training requirements.
- **Accuracy**. The system must accurately identify the type of obfuscation that has been applied.
- **Scalability**. The system must be able to cope with a large number of applications using a moderate amount of resources.

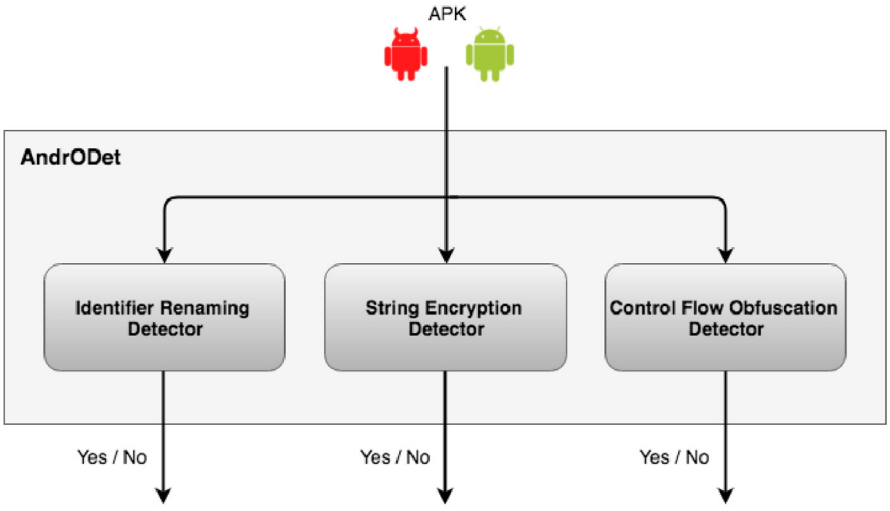
3.3. Dataset description

Our dataset is formed by both malware and benign applications, and contains ground truth for all of the obfuscation techniques considered in this work. We have built up the ground truth for identifier renaming and string encryption obfuscation techniques by leveraging the AMD dataset [10], a recently released Android malware dataset with apps from 71 families ranging from 2010 to 2016 (Table 1). This dataset is formed by 24,553 applications that are labeled based on a number of behavioral criteria, including the presence of different anti-analysis techniques (e.g., identifier renaming or string encryption) in the apps of each family of one particular variety. To have a fair and balanced ratio of obfuscated and non-obfuscated samples, we have selected the same number of apps for each type, some of which were obfuscated using more than one technique.

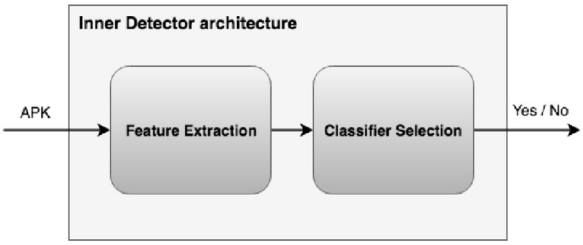
In order to create a dataset of Android apps for control flow obfuscation technique, 1,380 applications were downloaded from the F-Droid market [11]. Both the compiled app package (APK file) and their Java source code are available in the market. Therefore, they are used as the ground truth for non-obfuscated apps. Also, to gather the same number of control flow obfuscated apps, we apply Allatori [14] over 1380 apps selected randomly from AMD dataset. These apps are control flow obfuscated to the maximum level.[1] According to Allatori documentation, this level of obfuscation makes the apps bigger in size and a little bit slower as it uses all types of control flow obfuscation techniques. We finally choose 80% of this repository (2208 apps) to assess the accuracy of control flow obfuscation detection module, and we leave the remaining 20% (552 apps) to test its efficiency over unseen applications. The ratio of obfuscated and non-obfuscated samples is again equal in both portions.

Finally, we have used an additional released dataset, known as PraGuard [13] to evaluate the performances of our identifier renaming and string encryption detector modules over unseen applications. This dataset is composed of 10,479 samples, obtained by obfuscating the MalGenome [46] and the Contagio Minidump [47] datasets with seven different obfuscation techniques. It is worth mentioning that during our feature extraction process, we found

[1] http://www.allatori.com/doc.html



(a) AndrODet global structure.



(b) Inner structure of each detector module.

Fig. 2. AndrODet architecture.

Table 1
Number of apps per obfuscation technique.

Dataset	Identifier Renaming		String Encryption		Control Flow Obf.		Global	
	Obf	Non-obf	Obf	Non-obf	Obf	Non-obf	Obf	Non-obf
F-Droid	0	0	0	0	0	1380	0	1380
AMD	5992	5992	7119	7119	1380	0	14,491	13,111
PraGuard	1495	1495	1495	1495	0	0	2990	2990
Total	**7999**	**7999**	**8614**	**8614**	**1380**	**1380**	**17,481**	**17,481**

that some apps cannot be disassembled properly with dexdump, and, thus, we have discarded them from our datasets.

3.4. Feature extraction and feature selection

The first important decision to make in learning-based systems is to choose the set of features that will be used to label (or predict) new unseen instances. Once they are defined, analysts may decide to apply feature selection algorithms to discard those features that are not relevant despite the initial assumption, or those with a low variance among all instances. In our case, we aim to identify a set of features that, apart from being useful for the prediction task, can be rapidly extracted from the applications. Thus, we simply parse the Dalvik bytecode (recall Section 2.1) of each app using dexdump [18] to find the majority of features. Table 2 shows the set of all features considered. In addition, the distributions of all features extracted from all apps in our dataset are included in the Appendix A for further analysis. In what follows, we describe them per module in more detail.

Table 2
Set of all features considered for each detector module.

Identifier Renaming	String Encryption	Control Flow Obfuscation
Avg_Wordsize_Flds	Avg_Entropy	Num_Nodes
Avg_Distances_Flds	Avg_Wordsize	Num_Sinks
Num_Flds_L1	Avg_Length	Num_Edges
Num_Flds_L2	Avg_Num_Equals	Num_Goto/LOC
Num_Flds_L3	Avg_Num_Dashes	Num_NOP/LOC
Avg_Wordsize_Mtds	Avg_Num_Slashes	LOC
Avg_Distances_Mtds	Avg_Num_Pluses	File_Size
Num_Mtds_L1	Avg_Sum_RepChars	
Num_Mtds_L2		
Num_Mtds_L3		
Avg_Wordsize_Cls		
Avg_Distances_Cls		
Num_Cls_L1		
Num_Cls_L2		
Num_Cls_L3		

3.4.1. Features for identifier renaming detection

To detect identifier renaming, we extract 5 different features from the key identifiers of Dalvik bytecode, including fields, methods, and classes. The set of features considered here are the average

Table 3
Examples of identifiers extracted from an obfuscated malware sample in the Obad family.

App MD5 = f7be25e4f19a3a82d2e206de8ac979c8

List of fields	List of methods	List of classes
cOlcOOo	ocCCIII	oIoICCOc
lOocoOI	onOpen	AdminReceiver
loOoOlOl	onUpgrade	cOoOlCO
oclCIII	OoCOocII	lOocoOI
OoCOocII	OOIIIcCc	OICCcII
OoclOClo	onCreate	OclcoOlc
OIICCCco	cClccOlc	OoCOocII
ClCCCcCl	CcOCoIcO	OOIIIcCc
occcclc	oIOocIIo	OoclOClo
oOCCOOl	CoOOoOo	ClOIIoIc
oCOIIOO	oIIclclc	ClCCCcCl
ClOIIoIc	lCclCcoC	olcCIIC

Table 4
Examples of identifiers extracted from a non-obfuscated malware sample in the Univert family.

App MD5 = dadba61b42e3129dcbb2c37ba7177290

List of fields	List of methods	List of classes
mBigLargeIcon	getItemId	KeyEventCompatEclair
mParentFragment	isSingleShare	ViewPager
mSetIndicatorInfo	performPause	ContextCompat
EDGE_ALL	makeMainSelectorActivity	NotificationCompat
mPendingBroadcasts	setDrawerShadow	ParcelableCompat
TRANSIT_NONE	getCallingPackage	ScrollerCompat
mHandler	getConstantState	TransportPerformer
mTaskInvoked	setUserVisibleHint	PagerTitleStrip
mNumOp	setMenuVisibility	TimeUtils
PRIORITY_DEFAULT	setOverScrollMode	BackStackRecord
ACTIVITY_CREATED	dismissAllowingStateLoss	FileProvider
children	dataToString	SupportMenu

wordsize (in bytes), the average distance of consecutive extracted identifiers, and the number of identifiers with length 1, 2 and 3. To compute the distance between two identifiers, we first represent each string as a vector of natural numbers, where each component is given by the corresponding byte in the string. If they are not of the same length, the shorter identifier is right-padded by blank spaces. After this, the d_1 distance between both vectors is computed:

$$d_1(A, B) = \sum_{i=1}^{n} |a_i - b_i|, \tag{1}$$

where $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$ are the byte-level representations of both strings. Since we operate at the byte level, we refer to this as the ASCII distance of the two identifiers.

The rationale for the ASCII distance is the following. When using renaming, identifiers are normally replaced by repetitive or random sequences of characters in the English alphabet in Android benign apps, and special characters (encoded in Unicode) in malware samples [4]. Thus, consecutive extracted identifiers in ID-renamed malware samples usually have a small ASCII distance compared to the ones in benign apps, as shown in Tables 3 and 4. Moreover, based on our observations (Fig. 3), the number of identifiers with lengths lower than 3 were much more frequent in obfuscated samples than in benign apps, which provides additional support to our logic to choose this set of features for identifier renaming detection.

3.4.2. Features for string encryption detection

For string encryption detection we considered 29 different features at the beginning, all of which were obtained from the app bytecode. The set of initial features we considered included: the average entropy, the average wordsize, the average length, the average number of equals ('='), the average number of dashes ('−'), the average number of slashes ('/'), the average number of pluses ('+'), the average sum of repetitive characters which are appear more than once in a string, and the frequency of 21 different special characters, including underlines and spaces. However, we were left with only 8 features after applying feature engineering techniques, namely the average entropy, the average wordsize, the average length of strings, the average number of equals, dashes, slashes, and pluses, and, finally, the average sum of repetitive characters. This was done using a tree-based feature selection algorithm which scores features based on their importance and discards irrelevant ones [48].

We chose this set of features by visually analyzing a number of strings from both obfuscated and non-obfuscated samples. Critical constant strings in Android malware are normally encrypted by either AES or DES encryption algorithms [49]. Also, they are commonly encoded using Base64 scheme. These block cipher algorithms, depending on the mode which is adopted, require the input string to be an exact multiple of the block size. If the string to be encrypted is not an exact multiple, it is padded before encrypting by adding a padding string (or a pad byte). In our studies, we observed many strings in obfuscated samples which were padded by using '=' or '==' strings (Table 5). Furthermore, equal signs, dashes, slashes, and plus signs are observed mostly in obfuscated strings than in non-obfuscated ones.

Table 5
A snapshot of constant strings extracted from obfuscated malware in the Kyview and Triada families.

App MD5: 9f973194e1d2db2c8d37571b1b8afa49, Family: Kyview

AES
AES/CBC/PKCS5Padding
ARuhFl7nBw/97YxsDjOCIqF0d9D2SpkzcWN42U/KR6Q=
KXbn1K9Cz2ZgeOTJa+Veo9TtqgqFQ49etShsU9z+UAP37syBIxS/qy9gK8yB2kKw
cbSAmn5ZqTUlLC/bgOZkEzXGEOY21uWifgdKJs9yk7A=
XONjIhr7f5+v7VYE2sRnrybwgpe9YIOqpcEHDUiel7EzNqAyI0RSFuWdEz2ratN+
LbZjxcpsz6RheqLbO48YwKTUVh9wQrFoY7gJK2jAZFl=
/XHxH5XHwv8SxKlJV4XyYOIB7MuqmSwqMacPj1bbgbS8lA8tETEArriXswHCehFP
Jil+B/2MHKx+6dpy/2xm493DojzmiB3wB5+eGz7hPDU=

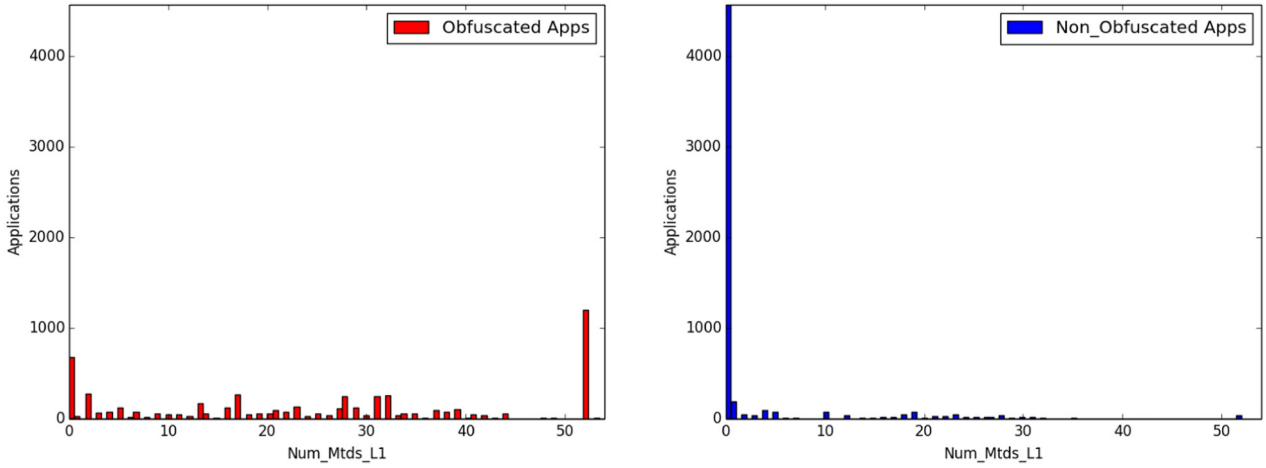
App MD5: a19f784807c3249837135de9b1a43fdf, Family: Triada

Sw4QQ1hFGFJJF1UWDwN1dnYKVQQGJAJDWwMUYkZVEUYHQg==
Wg4WQ2hRRkNySV8BOUNVX1U=
UQ4IGU5EGEZYF1UWDwN1dnYKVQQGJAJDWwMUYkZVEUYHQg==
VxkRaFZCUWxdS1sWOUtdXIU8QwAPAw==

3.4.3. Features for control flow obfuscation detection

Finally, to detect control flow obfuscation, features are extracted from both Dalvik bytecode and the CFG of applications. Seven different features are extracted here: the number of nodes; the number of sinks (i.e. nodes with an outdegree = 0); the number of edges from the CFG of each application; the number of goto instructions per line of code; the number of NOP instructions per line of code; and the total number of lines of code from the app's bytecode. Additionally, the app's file size is considered because some advanced types of Android malware pack their native code in the resource or assets directories and decrypt them at runtime using a decryption stub [25,50]. So, this feature compensates for the limitations of dexdump in accurately measuring the lines of code from sophisticated Android malware specimens.

Although features for control flow obfuscation detection are extracted from both bytecode and the CFG of apps, we had the intuition that some code features may overlap with others extracted from CFG. For instance, goto instructions simply add more branches to the CFG, and, thus, increase the in-degree or out-degree of some nodes. However, extracting features from both



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. 3. Distribution of methods with length 1 in obfuscated (a) and non-obfuscated (b) apps.

bytecode and the CFG guarantees that no features will be missed due to the limitations that may exist in Android reverse engineering tools.

3.5. Classification algorithms and hyper-parameter tuning

The second critical decision in learning-based systems is to choose an appropriate classifier to label unseen samples. Additionally, most of these classifiers have various parameters which have significant impacts on their performance. They are commonly known as classifiers' hyper-parameters which need to be set wisely based on the application context. One simple example is the number of neighbors (k) in the famous k-Nearest Neighbor (or kNN) learning algorithm [51].

Three strategies are usually adopted to tune classifiers' hyper-parameters [52]. In the first approach, all combinations of hyper-parameter values are tried in a greedy way to find the best possible set of combinations. In the second approach, all combinations are explored again but in a random fashion. The advantage of this method is that it may find the optimal solution faster than a greedy search. The third strategy is to use a random search but with a limited number of trials, which will make the algorithm even faster but does not guarantee finding the optimal set of combinations.

In AndroDet, all classifiers update their models while observing new applications based on online learning algorithms. To do this, we have used a wide variety of algorithms provided by *MOA*, including Hoeffding Tree [53], Weighted Majority Algorithm [54], Leveraging Bag [55], LearnNSE [56], Stochastic Gradient Descent (SGD) [32], and Naive Bayes [57]. Moreover, we have extended this tool to enable us choosing the best possible hyper-parameters for the classifiers by developing a hyper-parameter tuning procedure. From the three discussed strategies, we have chosen limited random search, which gave us a satisfactory classification performance in a reasonable period of time.

4. Evaluation

This section presents the evaluation results. We first present the experimental settings. Then we evaluate the performance of each AndroDet's detection module separately (Sections 4.2–4.4). Finally, we consider cases in which apps may be obfuscated with more than one technique (Section 4.5).

Additionally, we test the accuracy of our system on unseen apps (as discussed in Section 3.3) and compare the results with a similar system based on batch learning algorithms. We adopt the same strategy here, i.e., we initially test the performance of each module on unseen apps (Section 4.6.1, 4.6.2 and 4.6.3), and, then, we present the accuracy of system when apps may use a combination of obfuscation techniques (Section 4.6.4). We finally compare the performances of both systems in terms of time and memory usage in Section 4.7.

4.1. Experimental settings

Experiments were carried out on an Ubuntu server with 15 processors and 24 GB of RAM. We use Massive Online Analysis (MOA) in its version as of February 2018 [38] to analyze the accuracy of AndroDet. Also, to compare its efficiency with a similar system based on batch learning algorithms, we leverage the Auto-Tuned Models (ATM) tool [12], a recently proposed tool for machine learning and hyper-parameter tuning. We have selected various learning algorithms from this tool, namely kNN, Support Vector Machines (SVMs) [58], decision trees [59], and random forests [60].

For online learning algorithms, we have used leveraging bag, and, from batch learning ones, we have finally selected SVM after observing the performances of classifiers. Moreover, to have a fair comparison, we first tune the hyper-parameters of classifiers (Fig. 4) in both MOA and ATM following a limited random search strategy with 200 trials (known as budget in *ATM*). This helps us to obtain fairly well combination of parameters for each learning algorithm.

4.2. Identifier renaming detection

We use the full AMD dataset in order to inspect how the accuracy of identifier renaming module evolves over time using the *EvaluatePrequential* class of *MOA* [61]. This class evaluates a classifier on a stream by testing, and, then, training with each sample in the sequence. Experimental results show that AndroDet identifier renaming detection module is able to predict whether an app is obfuscated or not with a high accuracy immediately after observing few samples. As it is shown in Fig. 5(a), the accuracy reaches around 71% after observing only 25 samples. Also, it improves step by step by observing more samples from the dataset. Our module to detect

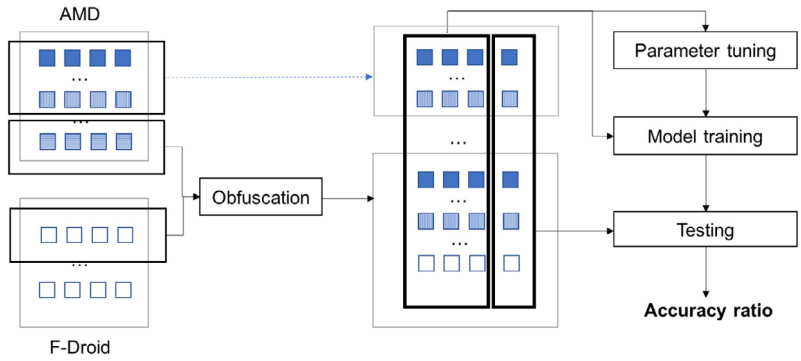


Fig. 4. Data preparation (left) and the overall architecture of classification process (right), including parameter tuning, model training and testing. White squares: non-obfuscated apps; dark blue squares: apps with string encryption obfuscation; dashed blue squares: apps with ID renaming obfuscation. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

identifier renaming obfuscation could achieve an average accuracy of 92.02% over the whole AMD dataset.

The number of samples correctly classified (TP) as obfuscated is 5758, and 631 samples were incorrectly classified (FP) as obfuscated (Table 6). One reason is that some obfuscators use a different strategy to rename key identifiers of malware samples such as using non-ASCII characters. The second reason is that non-obfuscated malware specimens do contain obfuscated identifiers as well in the majority of cases mainly because they import some classes from Android or Google libraries which are already obfuscated.

4.3. String encryption detection

As malware samples use a wide range of cryptographic functions, classifying apps as either obfuscated or non-obfuscated is not straightforward even if a fine set of features is considered. Also, advanced malware pack the original .dex file of applications and decrypt them at run-time by using a wrapper; therefore, they put a big challenge ahead of systems which rely heavily on features extracted before runtime.

Similar to identifier renaming detection, we use the full AMD dataset to evaluate the accuracy of our string encryption detection module when new apps are fed into the system over time. Our module for string encryption detection could achieve an average accuracy of 81.41% as shown in Fig. 5(b). It improves soon after observing a few samples and increases up to 87.4% at maximum.

In total, 5499 samples were correctly classified (TP) as obfuscated, and 906 apps were mistakenly classified (FP) as obfuscated. In our studies, we found that malware samples make use of a wide range of cryptographic functions and encryption strategies which makes it challenging to consider a proper set of features in order to detect this particular type of obfuscation. A very simple way to do this is to simply extract some features from encrypted strings. Another advanced way is to extract features from encryption/decryption functions which are not always easily extractable as they are sometimes hidden in resource directory and are dynamically exercised at run-time.

4.4. Control flow obfuscation detection

Due to the limited number of samples we had for this type of obfuscation, we assess the average accuracy of our control flow obfuscation detection module over time only based on 80% of the applications collected here, and we keep the 20% remaining apps to test our system over unseen apps (recall Section 3.3) in the next sections (Sections 4.6.3 and 4.6.4). Experimental results show that the corresponding AndroDet module for control flow obfuscation detection is able to identify obfuscated apps with an average accuracy of 68.32% and a maximum accuracy of 73.4% on the final samples (Fig. 5(c)). This seems to be reasonable due to the limited number of samples we could feed into this module. Also, maximum accuracy percentage shows that this module would probably be able to have a better performance if it is fed with more training samples with a proper distribution of features.

Control flow obfuscation detection module could correctly label 898 samples as obfuscated (TP). Also, 429 samples were wrongly classified as obfuscated (FP). The main important reason for these relatively smaller values comparing with the ones achieved for identifier renaming and string encryption detection modules is the small amount of apps we had as ground truth for this type of obfuscation.

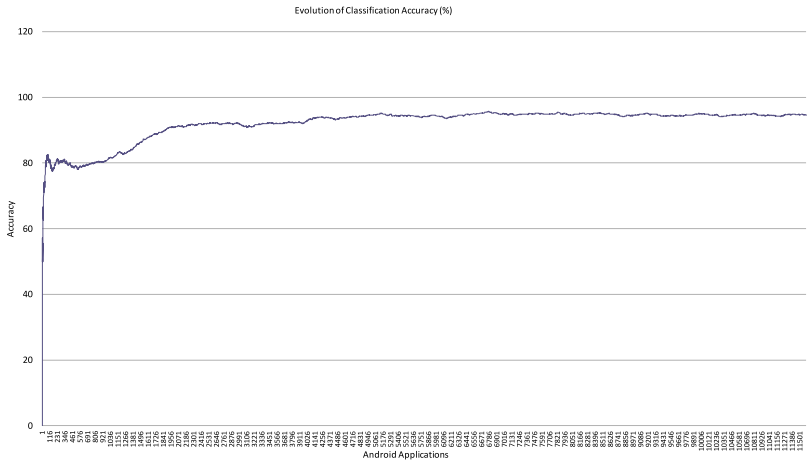
Generally speaking, accuracy plots for each of the obfuscation detection modules demonstrate the improvement of online learning algorithms over time when they observe more and more samples considering the fact that they do not need to be re-trained.

4.5. Performance evaluation for combined techniques

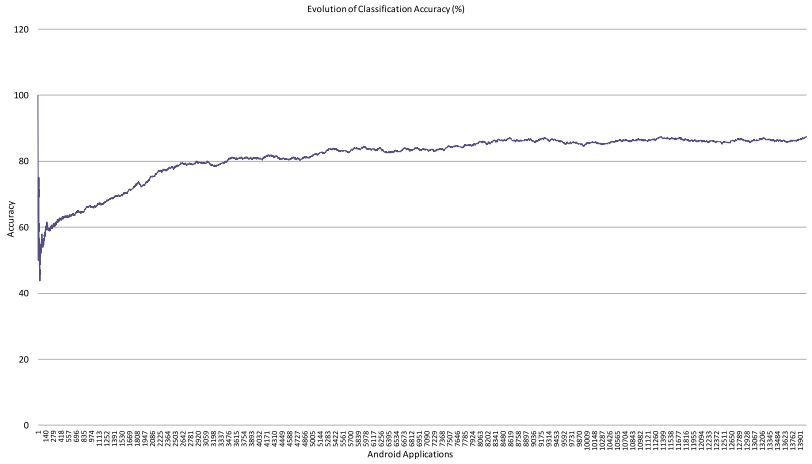
To measure the performance of our system when apps are obfuscated using a combination of techniques, we extend the binary classification problem of each module to a multi-label classification problem and calculate the global accuracy using the same strategy we adopted for individual modules. Here, each detector module is tested and trained separately using the *EvaluatePrequential* class.

To achieve our goal and to be able to create a multi-label confusion matrix, we consider the presented encoding in Fig. 6. Thus, total number of combinations is 8 each of which is a binary representation of techniques used to obfuscate an application. For instance, 6 ('110') is a label which shows that an app is obfuscated using both identifier renaming and string encryption techniques, and 0 ('000') demonstrates that the app is not obfuscated with any of these three techniques. However, we have excluded those labels for which we did not have any ground truth in our datasets.

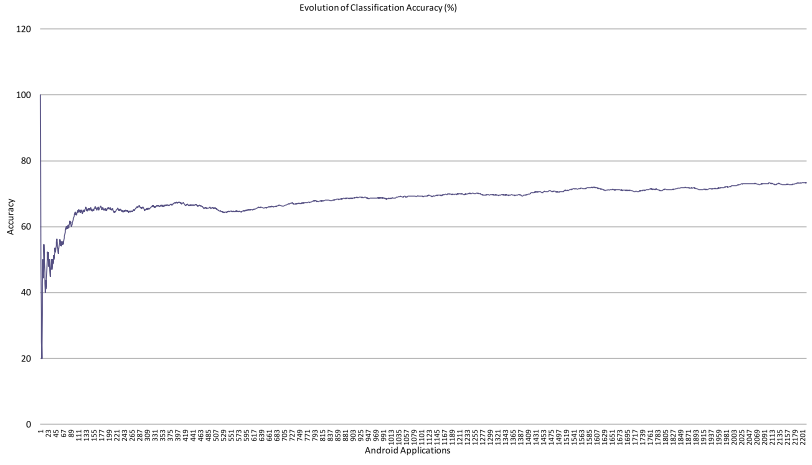
As it is clear from the confusion matrix (Table 7), the performance of each module obtained by dividing the true positive by false negative for that obfuscation technique is close to the values we separately evaluate on the previous sections. Also, the global accuracy of AndroDet is approximately 80.66% considering the fact that some apps could be obfuscated with more than one technique. The prediction accuracy for apps which are obfuscated by identifier renaming and string encryption at the same time is 76.68% which stems in the fact that we had limited samples obfuscated with both techniques as ground truth.



(a) Identifier renaming detection



(b) String encryption detection



(c) Control flow obfuscation detection

Fig. 5. Evolution of detector modules' accuracies over time.

4.6. Comparison against batch learning algorithms

This section compares the accuracy of our system to detect each type of obfuscation with a similar system based on batch learning algorithms over unseen applications. To do so, we make use of a new dataset, known as PraGuard (recall Section 3.3). Also, we present and discuss the performance of both systems when a combination of techniques are used to obfuscate Android apps. Table 8 summarizes the results.

Table 6
Performance metrics for each detection module.

Detector	TPR (Recall)	FPR (Inverse Recall)	Precision	F1 Score
Identifier Renaming	0.91	0.02	0.95	0.92
String Encryption	0.80	0.08	0.78	0.79
Control Flow Obfuscation	0.66	0.1	0.7	0.67

Table 7
Confusion matrix for multi-label classification with MOA (real classes on rows and predicted classes on columns).

	N	CF	SE	IR	IR+SE
N	10,313	0	715	368	719
CF	210	758	0	0	142
SE	392	0	5784	242	701
IR	103	0	99	5513	277
IR+SE	309	0	300	213	1224

N: No Obfuscation, CF: Control Flow Obfuscation. SE: String Encryption, IR: Identifier Renaming.

Table 8
Comparison of the accuracy between two systems for Android obfuscation detection based on online and batch learning algorithms (maximum accuracies).

Identifier Renaming		String Encryption		Control Flow Obfuscation	
MOA	ATM	MOA	ATM	MOA	ATM
95.1%	91.5%	85.6%	81.2%	73.7%	87.9%

4.6.1. Identifier renaming detection

To compare the performance of AndroDet's identifier renaming detection module with a similar system based on batch learning algorithms, we do the following experiment. We first feed our online learning module with a combined dataset of apps from AMD and PraGuard to measure its average accuracy using MOA. Then, we train another module based on batch learning algorithms with AMD to test it later over the PraGuard dataset using ATM tool.

Our results show that the online learning module improves its accuracy to 95.1% by observing further samples from PraGuard dataset. One the other hand, the module based on batch learning could achieve an accuracy of 91.5% (Table 8). The results obtained here highlights the adaptability power of online learning systems versus batch learning ones when new samples appear over time.

4.6.2. String encryption detection

We adopt a Similar strategy to compare the performance of our online learning based module with another module which makes use of batch learning for string encryption detection on unseen applications, i.e., we observe how the accuracy of our learning module evolves over time when the new dataset (PraGuard) is fed into the system. We then train the batch learning based module with the AMD dataset and test it over PraGuard dataset to compare their accuracies.

Results confirm that the online module is able to update its model incrementally by observing new samples, and, thus, could reach an accuracy of 85.6% compared to the batch learning module with a lower accuracy. Although the difference is not big, this result bolds the advantage of online learning algorithms over batch learning ones in improving their built model without the need of time consuming training procedure.

4.6.3. Control flow obfuscation detection

Due to the limited available ground truth for control flow obfuscated apps, 80% (2220 apps) of the apps (1387 obfuscated apps from AMD and 1387 non-obfuscated apps from F-Droid) is used to evaluate our online learning module (as performed in Section 4.4), and 20% (554 apps) is used to inspect how our system's accuracy evolves when new apps appear, and, also, to compare its performance with a similar module based on batch learning algorithms.

Table 9
Confusion matrix for multi-label classification with MOA on unseen applications (real classes on rows and predicted classes on columns).

	N	CF	SE	IR	IR+SE
N	11,913	0	715	374	887
CF	145	1018	0	0	224
SE	459	0	7196	368	591
IR	97	0	166	7049	175
IR+SE	229	0	216	267	2829

N: No Obfuscation, IR: Identifier Renaming, SE: String Encryption, CF: Control Flow Obfuscation.

Table 10
Confusion matrix for multi-label classification with ATM on unseen applications (real classes on rows and predicted classes on columns).

	N	CF	SE	IR	IR+SE
N	12,021	0	709	369	790
CF	45	1216	0	0	126
SE	459	0	7146	368	641
IR	92	0	149	6877	369
IR+SE	254	0	216	267	2804

N: No Obfuscation, IR: Identifier Renaming, SE: String Encryption, CF: Control Flow Obfuscation.

The accuracies obtained from both systems show that the batch learning based module can predict the label of unseen apps with a higher accuracy. However, there is a major difference between our test samples used for this module with the other two modules (Sections 4.6.1 and 4.6.2). The difference is that unseen apps are fed into the system from the same datasets (AMD and F-Droid) which were used for evaluating our online module, and, thus, are expected to have similar features. In other words, unseen apps do not add much information to the previously built model of our module.

4.6.4. Combined obfuscation techniques

In a final assessment, we repeat the same experiment as we did in Section 4.5, but on unseen applications. Thus, we use the PraGuard dataset as ground truth for identifier renaming and string encryption techniques, and the remaining 20% of apps from AMD and F-Droid as ground truth for control flow obfuscation. We compare our results with another system based on batch learning algorithms. For AndroDet, we inspect how our system can extend its built model when new apps are fed into the system and when they might use a variety of obfuscation techniques.

As it is clear from the confusion matrices of the two detection systems (Tables 9 and 10), the global accuracy of AndroDet when it is fed with more unseen applications and is tested at the same time is around 83.34% which shows a minor improvement comparing with the one obtained in Section 4.5. On the contrary, the global accuracy of a similar system based on batch learning algorithms is around 85.64%. Also, accuracies of detector modules which can be obtained from these matrices are aligned with the results we achieved before (Table 8).

In particular, the individual accuracy of the control flow obfuscation detection module on unseen applications using batch learning algorithms is slightly higher than the accuracy of the same

IR	SE	CF
2	1	0
0/1	0/1	0/1

Fig. 6. Multi-label encoding of obfuscation techniques.

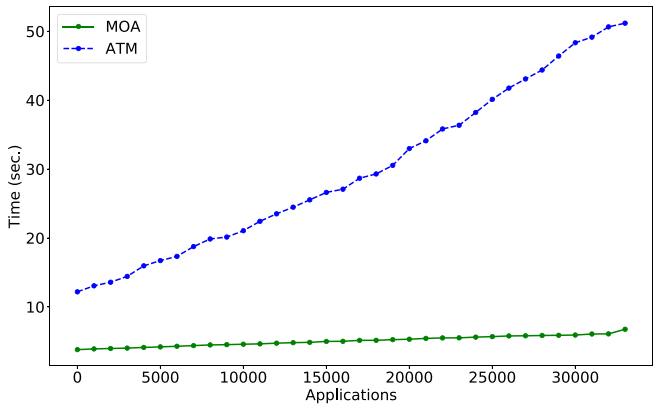
module based on online learning algorithms. This is vice versa for the other two obfuscation techniques, namely identifier renaming and string encryption, i.e. the accuracies of detector modules which make use of online learning algorithms are higher than the same modules which are based on batch learning algorithms. Also, the system which works based on batch learning algorithms outperforms AndrODet when it comes to apps that are obfuscated by both identifier renaming and string encryption techniques.

4.7. Performance comparison: time and memory

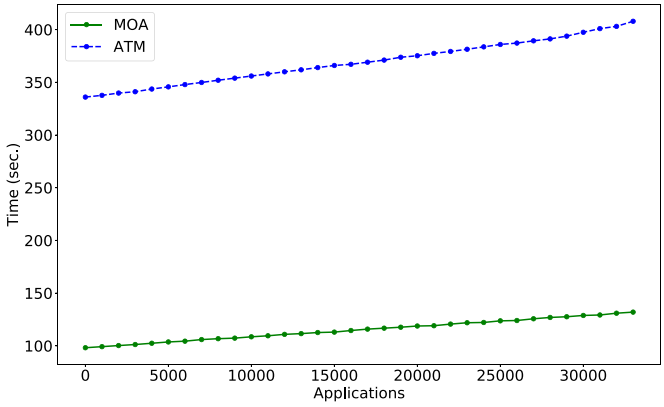
One key advantage of using online learning algorithms in classification is their ability to update their model upon observing new samples opposite to batch learning algorithms which do need to be re-trained after specific intervals in order to preserve their accuracies over time. Re-training process needs a considerable amount of memory as well. Thus, to compare AndrODet with a similar system based on batch learning algorithms (the systems discussed in Section 4.6.4) in terms of time and memory we conduct the following experiment.

For time analysis, we assume that the batch learning system needs to be re-trained after classifying every 1000 samples (1000 epochs). With this assumption, we start classifying the whole applications (recall Table 1); but, here, the system is re-trained after classifying every 1000 samples. Thus, the time for each epoch is calculated by summing up the time which is needed to train, and, then, test the system over next 1000 samples. And, the final cumulative time is the sum of time spent in all epochs until it classifies all applications. For AndrODet, each epoch's time is obtained by only measuring the time which is used for classification. We analyze memory usage based on the same assumption as shown Fig. 7. Here, we exclude the amount of time and memory which is used for hyper-parameter tuning in both systems. However, we consider the time which is needed to train both systems at the beginning.

As it is clear, AndrODet outperforms a similar system based on batch learning algorithms in both time and memory consumption on a medium size dataset. If the dataset size increases time by time, and if the built model is needed to be updated in shorter intervals, this difference will most probably be higher between online learning systems and batch learning ones. Another important aspect is to inspect the amount of memory which is consumed as the dataset grows in size over time. Based on our observations, AndrODet consumed 33.79 MB at maximum as the dataset increased to around 34 K apps. In contrary, the system based on batch learning algorithms consumed 71.89 MB of RAM memory as more samples were added to the training set over time.



(a) Computation Time.



(b) Memory Consumption.

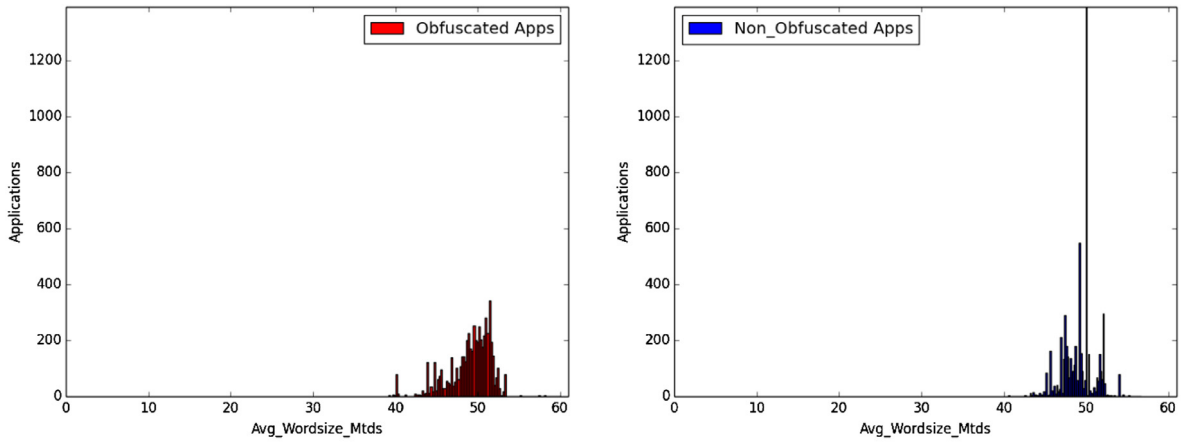
Fig. 7. Comparison of time and memory consumption between online learning algorithms using MOA (a) and batch learning algorithms using ATM (b) for Android obfuscation detection.

5. Threats to validity

This section discusses a number of potential limitations we encountered in our work. Our datasets contain two main issues that could impact the validity of our results. On the one hand, they do not contain an uniform distribution for all combinations of obfuscation techniques. For example, there is not a sample in our datasets in which string encryption and control flow obfuscation have been jointly applied. To the best of our knowledge, there is no dataset that contains such a type of application. Therefore, the analysis on the effectiveness of this approach for these types is left for future work. On the other hand, our datasets contain apps which are control flow obfuscated using a single tool (i.e., Allatori). As a consequence, apps which are obfuscated with other tools may evade detection by AndrODet if the techniques they employ are quite different.

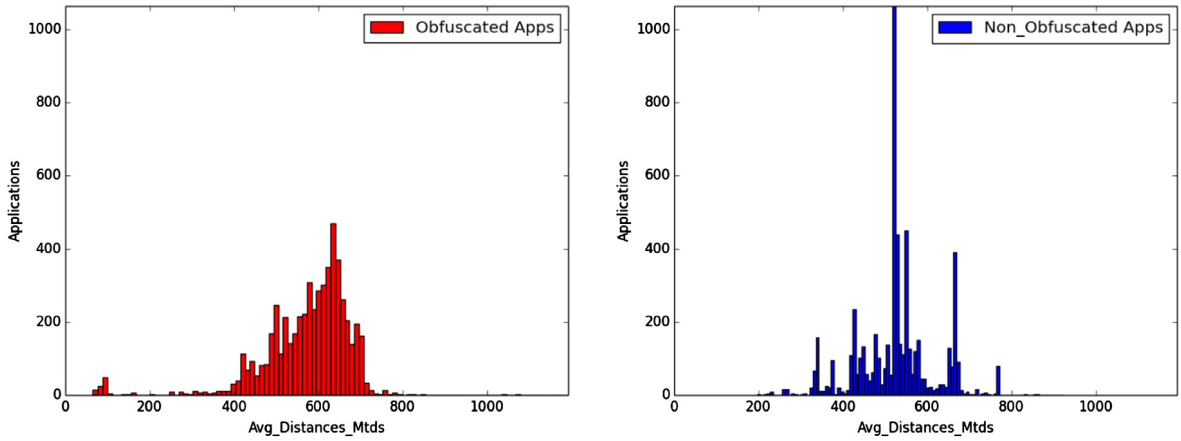
State-of-the-art Android reverse engineering tools are shown not to work properly in all cases. Thus, systems that make use of features extracted by these tools are prone to errors. For instance, disassemblers may make mistakes which could in turn hide information to the systems that use the result of disassembly. Also, tools which extract control flow graphs are not perfect, specially when apps adopt advanced anti-analysis techniques.

Advanced code obfuscation techniques in Android may use a combination of transformations [62]. Although AndrODet is modular and can detect if a malware is obfuscated using more than one technique, it does not consider all possible combinations which might exist in the wild. However, there is not a comprehensive and systematic study to report the prevalence of adopting various combinations of Android obfuscation techniques at the moment. Moreover, advanced malware specimens use a wide range of techniques to evade malware analysis systems which can affect our system.



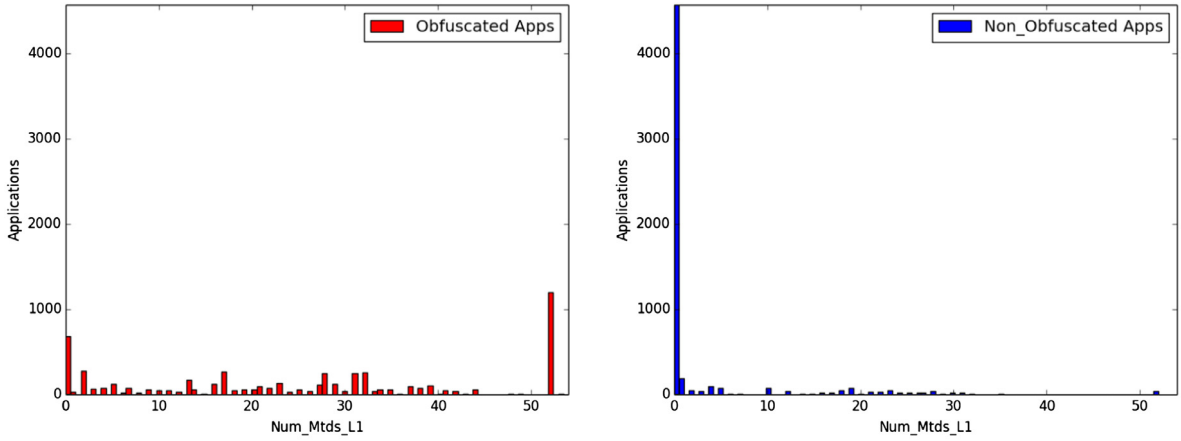
(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. A.8. Distribution of the average wordsize of methods in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. A.9. Distribution of the average ASCII distances between consecutive extracted methods in (a) obfuscated and (b) non-obfuscated apps.

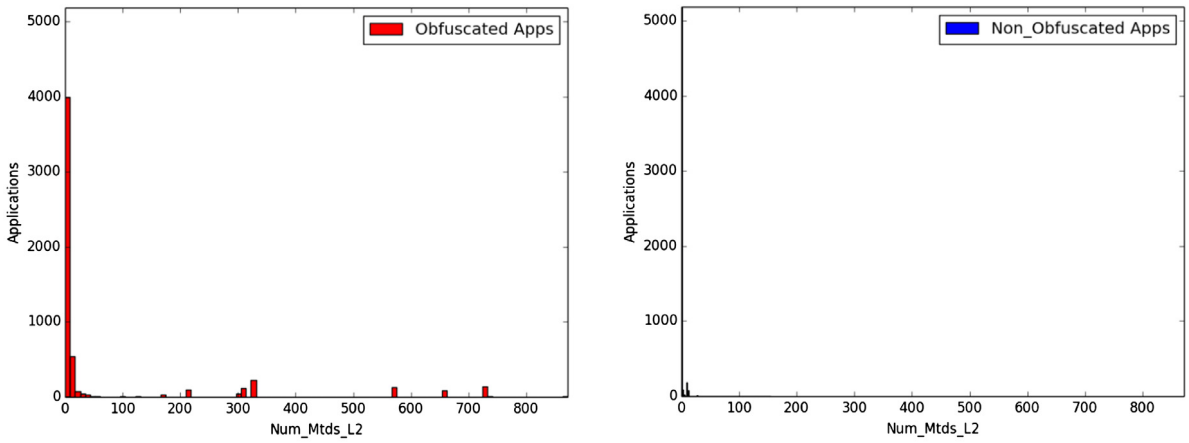


(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. A.10. Distribution of methods with length 1 in (a) obfuscated and (b) non-obfuscated apps.

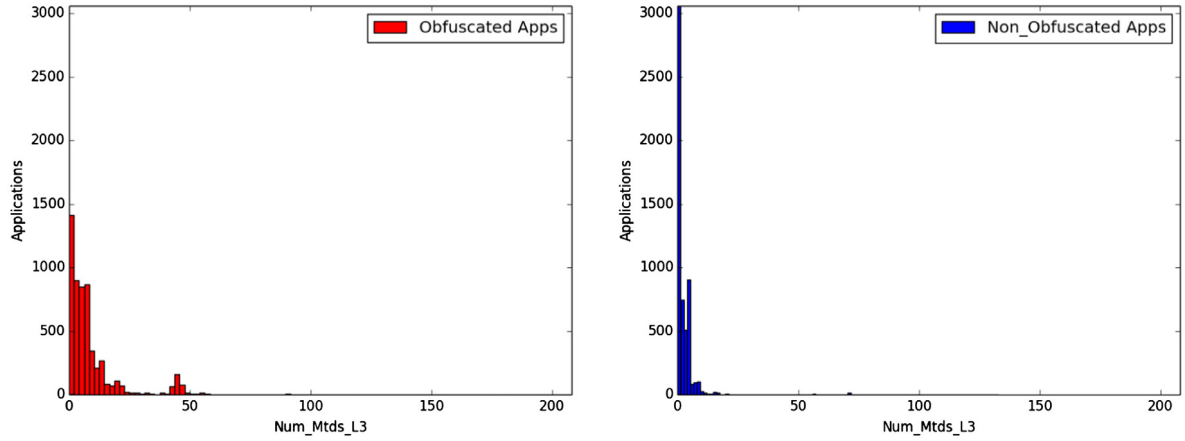
6. Related work

Many prior works have attempted to address the problem of handling obfuscation in Android. On the one hand, the goal of several works is to carry out a process without any impact despite of obfuscation. Particularly, a matter of interest is malware analysis. In this regard, [63] propose RevealDroid, a system for malware detection and family identification in an obfuscation-resilient manner. On the other hand, Zhang et al. aim to detect



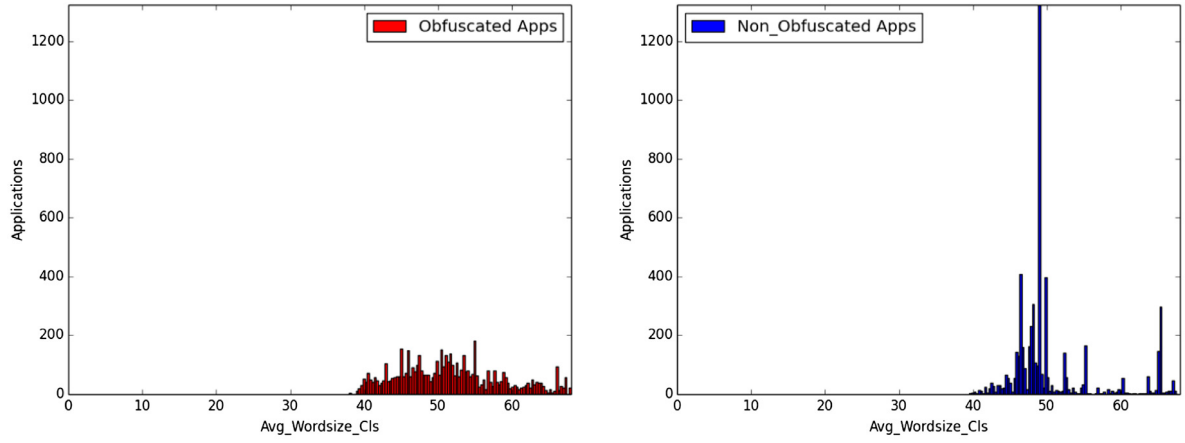
(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. A.11. Distribution of methods with length 2 in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. A.12. Distribution of methods with length 3 in (a) obfuscated and (b) non-obfuscated apps.

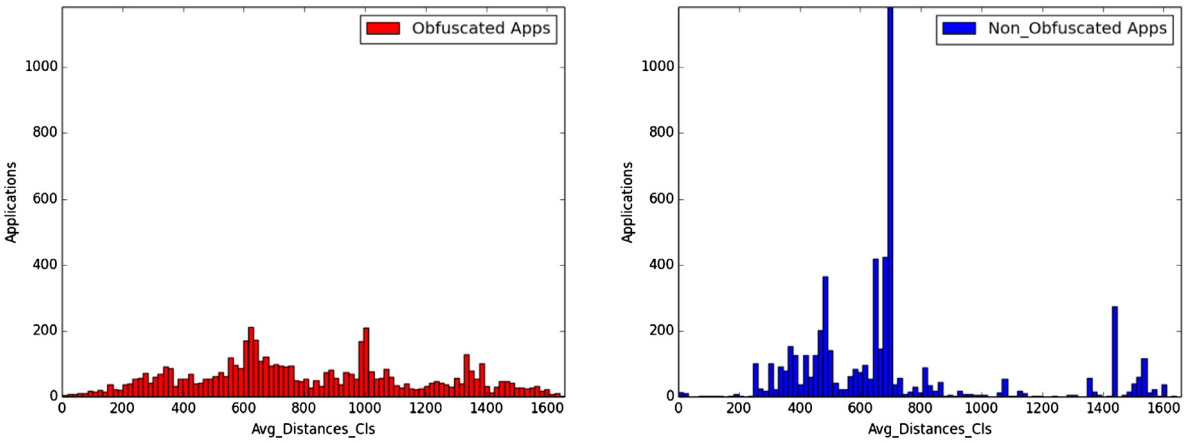


(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. A.13. Distribution of the average wordsize of classes in (a) obfuscated and (b) non-obfuscated apps.

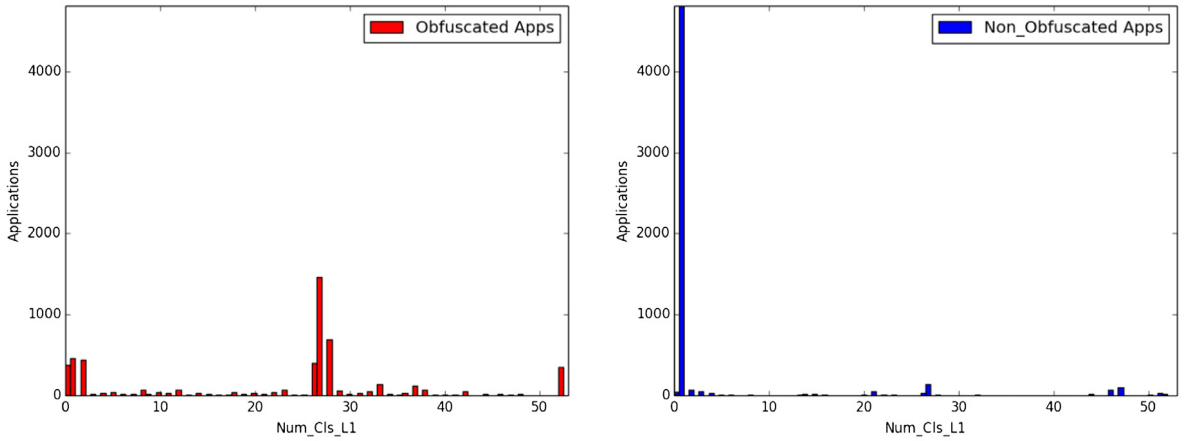
repackaged applications by inspecting the user interactions in the graphical interface [64]. The same problem is addressed by CodeMatch, which is able to deal with other types of obfuscation such as code slicing [65].

The works described so far consider obfuscation as an obstacle to be saved to achieve a goal of a different nature. In this work, the detection of obfuscation is indeed the target of the approach. In this regard, two actions have been considered in other works,



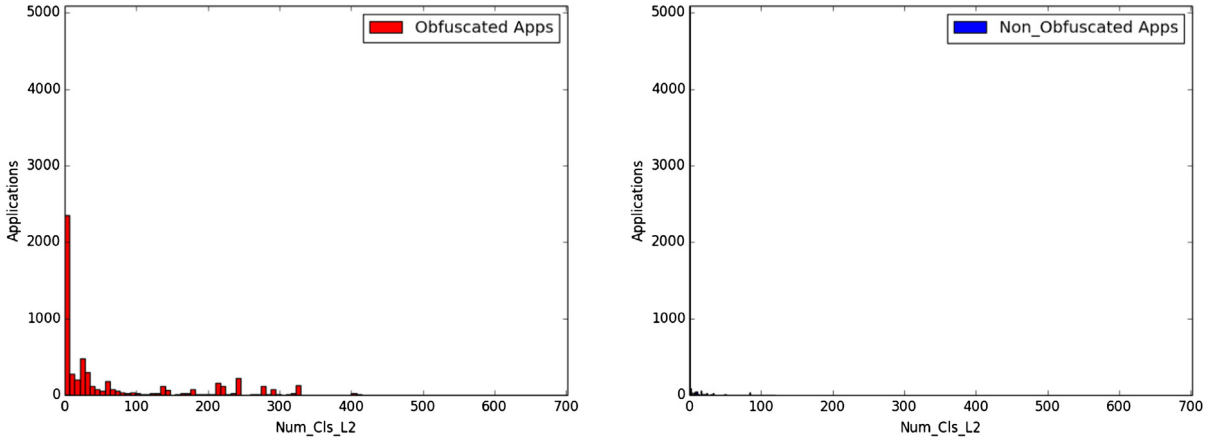
(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. A.14. Distribution of the average ASCII distances between consecutive extracted classes in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. A.15. Distribution of classes with length 1 in (a) obfuscated and (b) non-obfuscated apps.

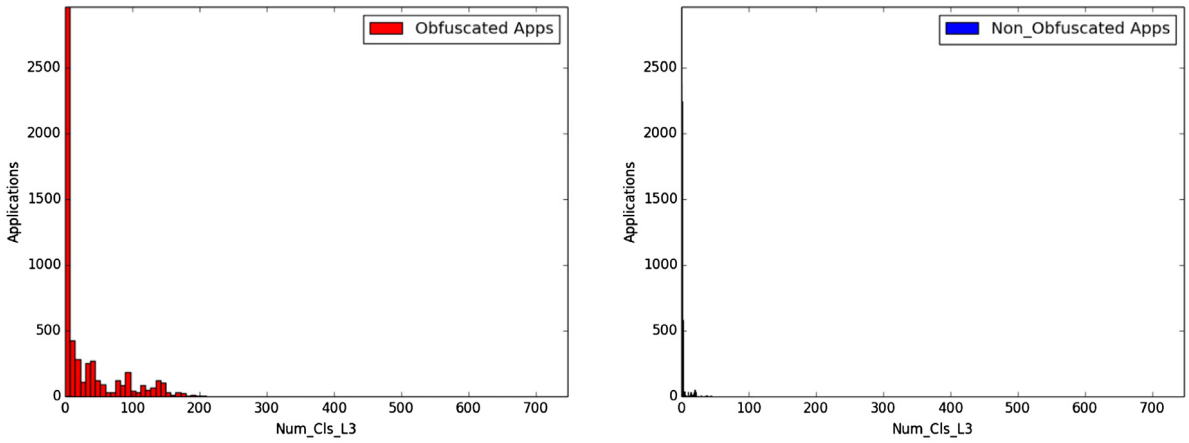


(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. A.16. Distribution of classes with length 2 in (a) obfuscated and (b) non-obfuscated apps.

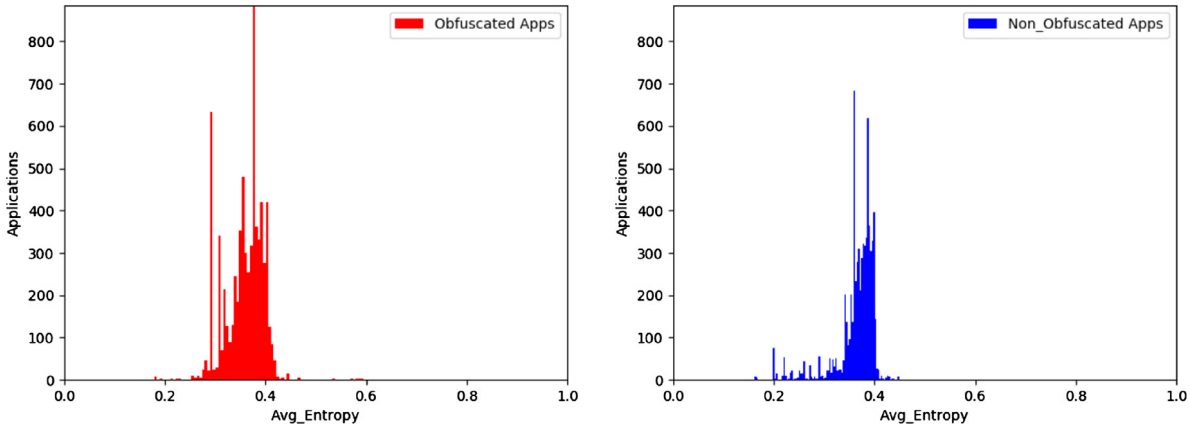
either detecting obfuscation or even attempting to deobfuscate the app. Each one is described in the following.

With respect to obfuscation detection, in 2018 Dong et al. have carried out a large-scale investigation. They focus on four types of obfuscation, namely identifier renaming, string encryption, Java reflection and packing. For each of them, they propose a lightweight detector that leverages signatures and machine learning techniques. Their approach is assessed using a dataset formed



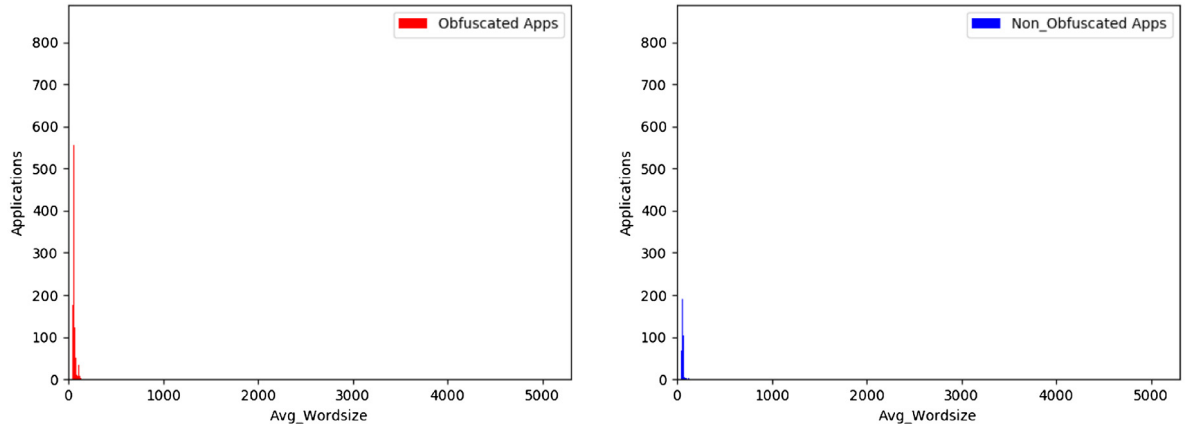
(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. A.17. Distribution of classes with length 3 in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

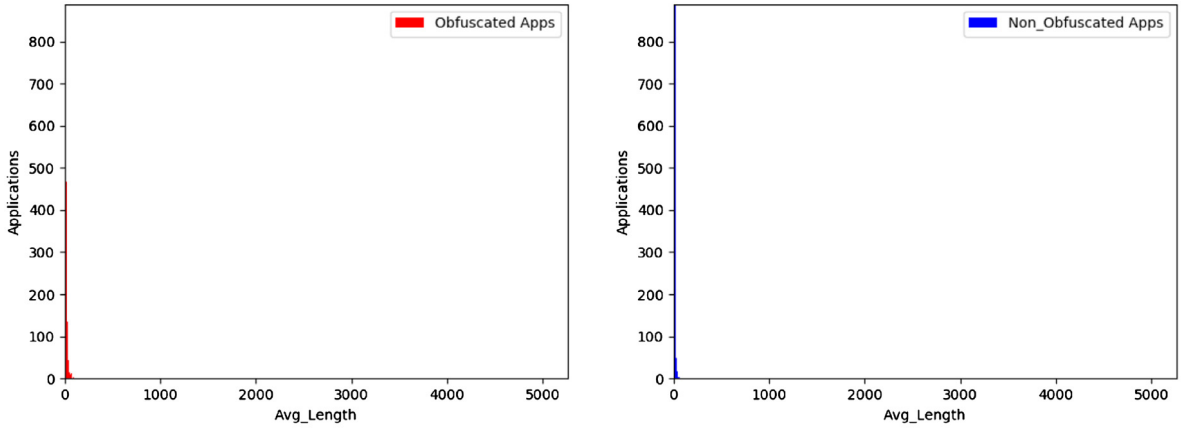
Fig. B.18. Distribution of the average entropy of strings in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

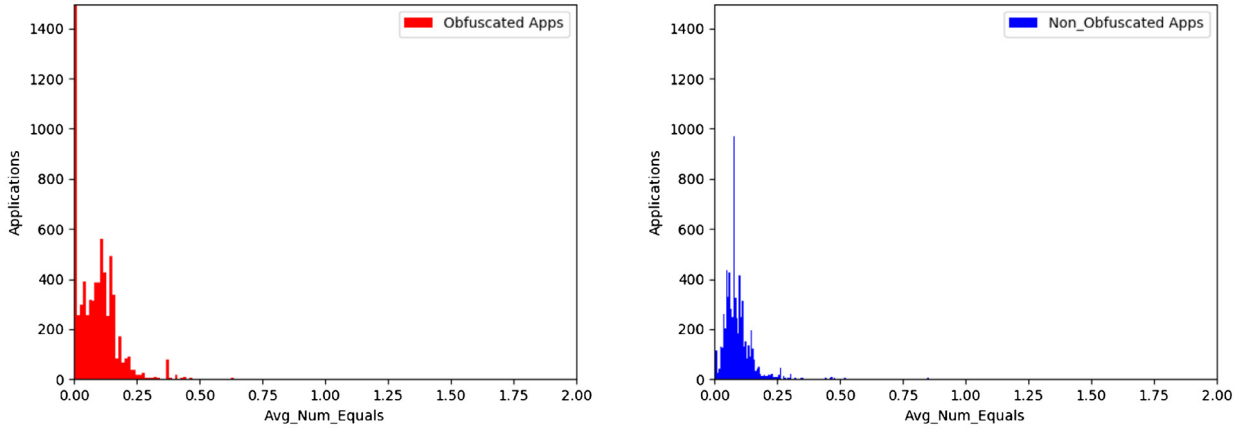
Fig. B.19. Distribution of the average wordsize of strings in (a) obfuscated and (b) non-obfuscated apps.

by 114,560 apps from both goodware and malware. To detect identifier renaming and string encryption, they use Support Vector Machine (SVM) as technique and 3-grams as features. To date, their work is the most similar to ours. In a similar vein, Wang and Rountev attempted to detect the tool that has been applied. For this purpose, they take 282 apps from F-Droid and obfuscate them using different tools using several configurations. These configurations indicate the type of obfuscation applied. Interestingly, these configurations involve identifier renaming, string encryption, package modification and control flow obfuscation. Using



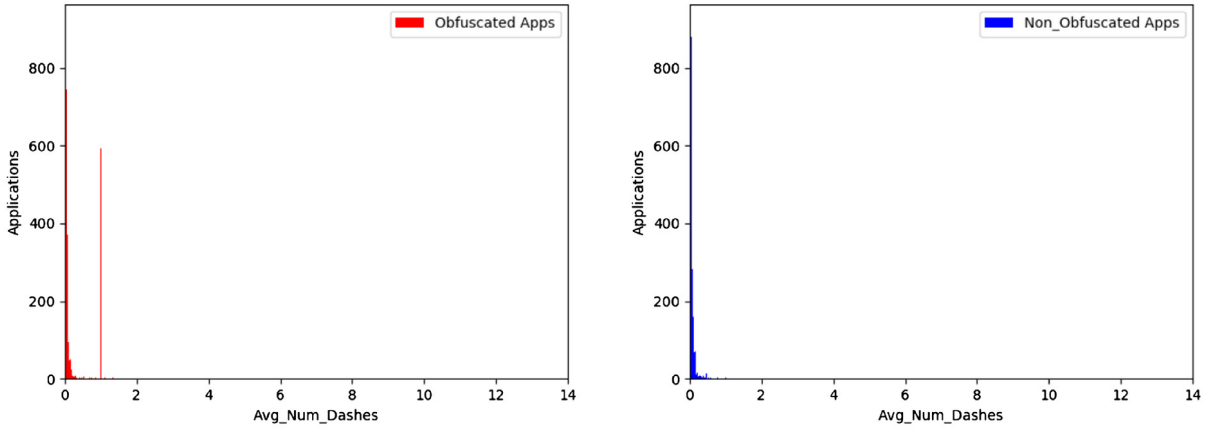
(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. B.20. Distribution of the average length of strings in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. B.21. Distribution of the average number of '=' characters in (a) obfuscated and (b) non-obfuscated apps.

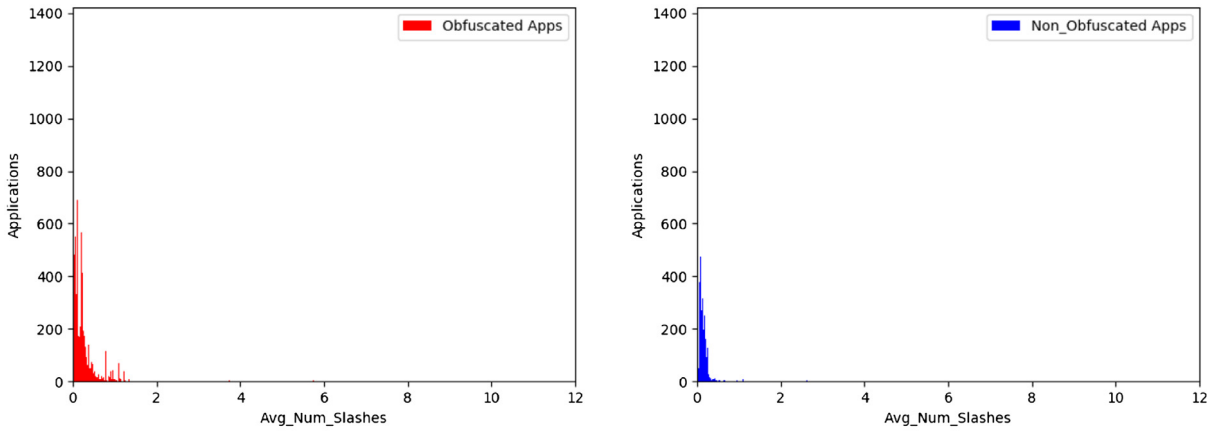


(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. B.22. Distribution of the average number of '-' characters in (a) obfuscated and (b) non-obfuscated apps.

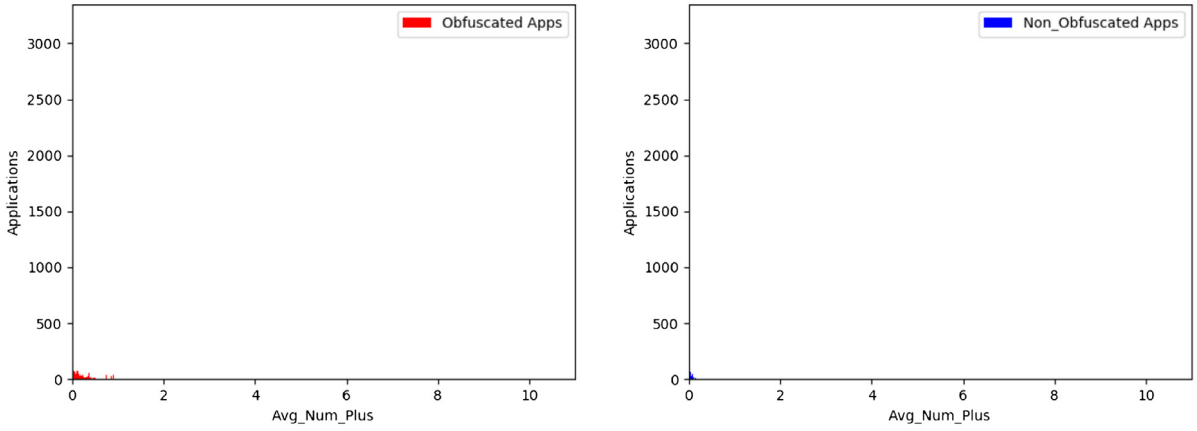
10 sets of strings (e.g. method names, package names, etc.), their approach also relies upon SVMs [8]. In their approach, they reach 97.5% of accuracy for obfuscator detector, and similar rates when it comes to detect which configuration has been applied in each tool. As compared to this work, their dataset is significantly smaller. Moreover, they do not deal with the re-training aspect.

Concerning deobfuscation attempts, [66] presents early results on deobfuscation against ProGuard tool. Their approach is based



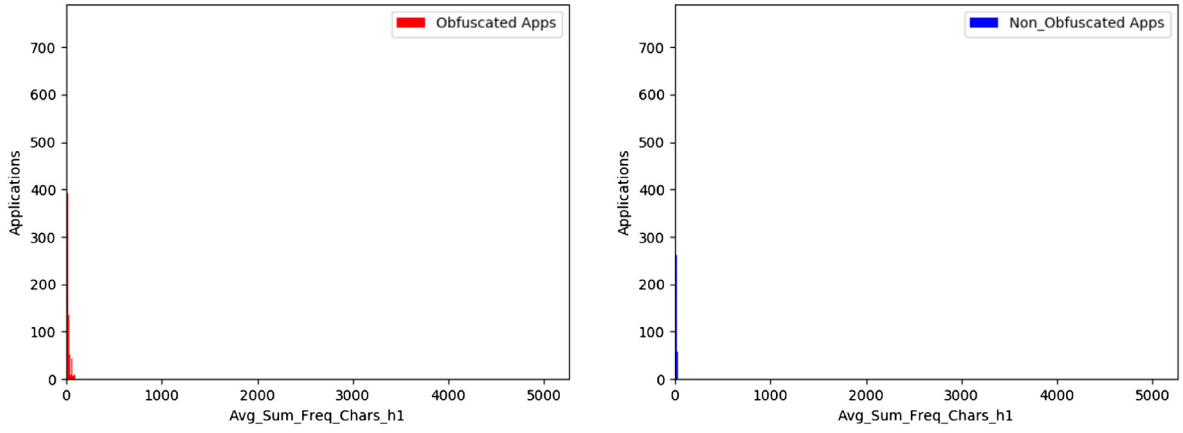
(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. B.23. Distribution of the average number of '/' characters in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

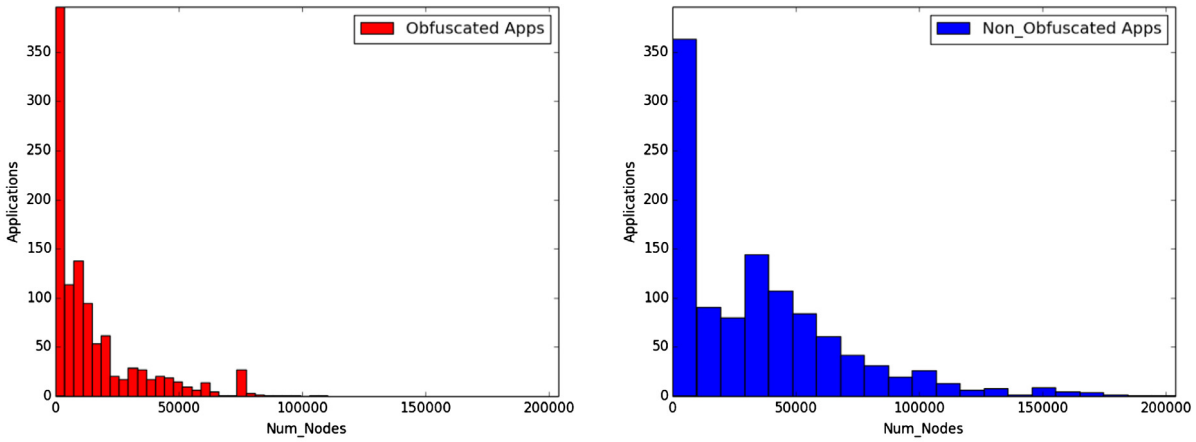
Fig. B.24. Distribution of the average number of '+' characters in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

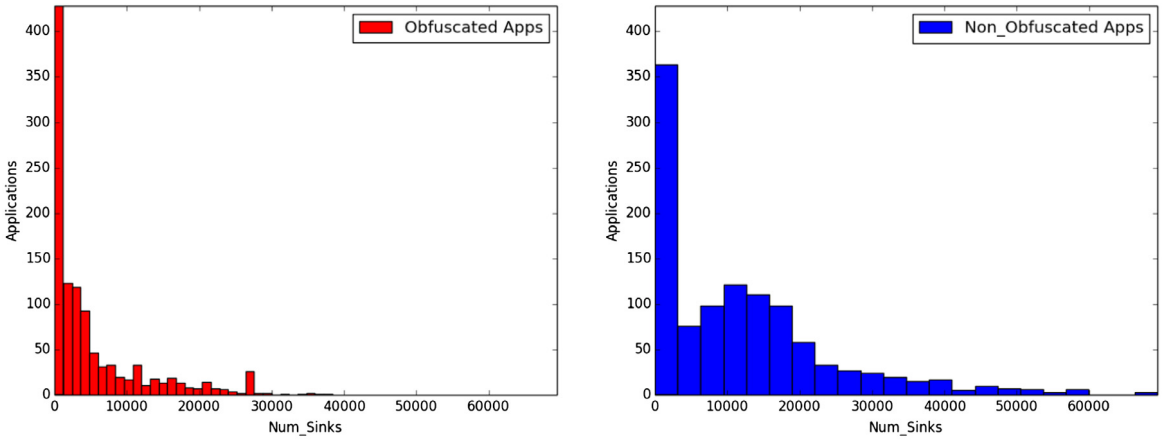
Fig. B.25. Distribution of the average sum of repetitive characters in (a) obfuscated and (b) non-obfuscated apps.

on comparing the similarity of some portions of the code against a database filled up with unobfuscated code. On the other hand, Yoo et al. propose a string deobfuscation technique to improve malware detection ratios [67]. This technique is based on running the app, intercepting all results coming from functions returning strings, and, then, repackaging the app replacing the original strings with these intercepted results. In this way, no matter what kind of encryption is applied, the tool is able to get the decrypted



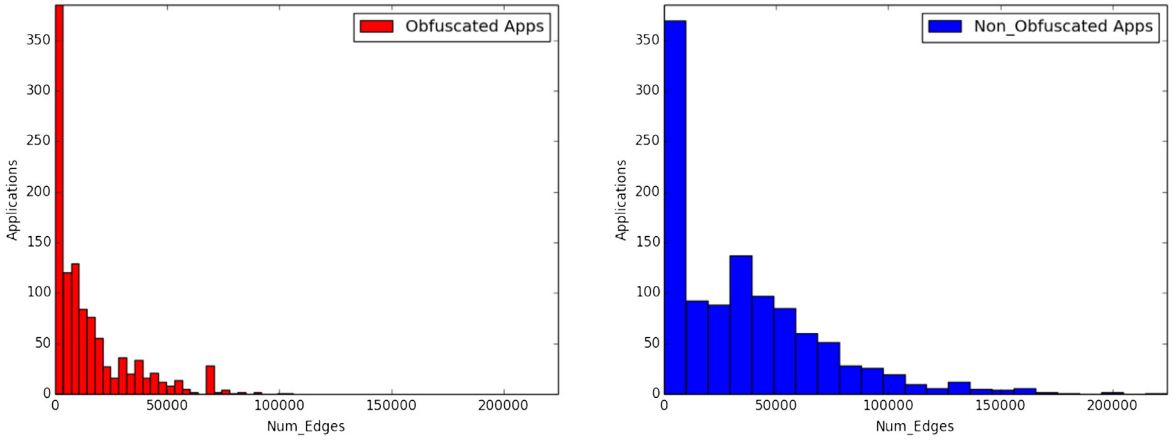
(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. C.26. Distribution of the number of nodes in the CFG of (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. C.27. Distribution of the number of sinks in the CFG of (a) obfuscated and (b) non-obfuscated apps.

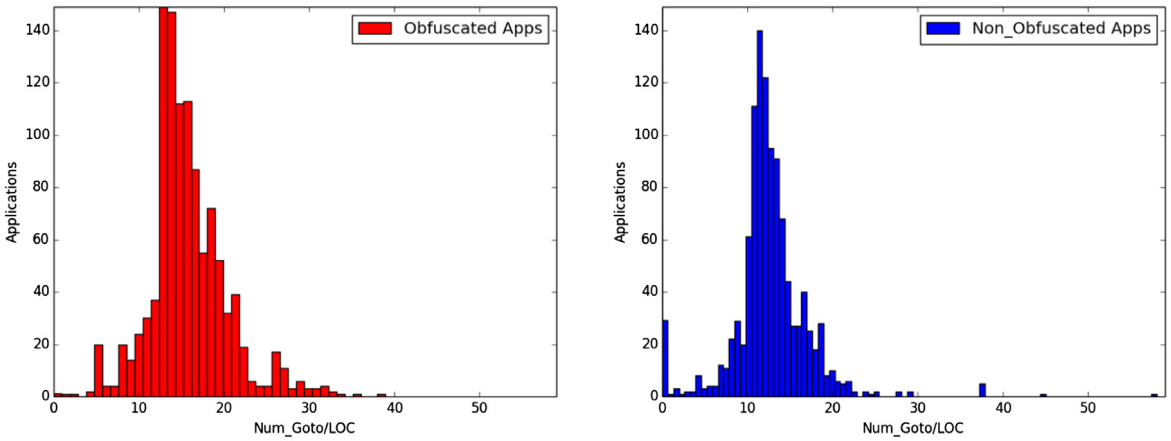


(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. C.28. Distribution of the number of edges in the CFG of (a) obfuscated and (b) non-obfuscated apps.

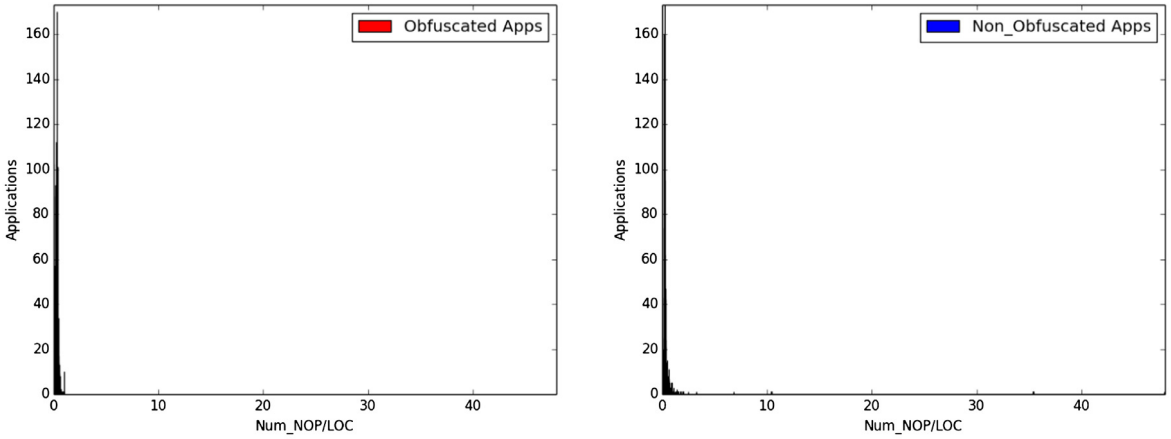
value. Their method outperforms other tool-specific mechanisms such as dex-oracle.[2] Another deobfuscation work is presented by Bischel et al. [68]. Their focus is on identifier renaming obfuscation, and their approach bases on comparing a given identifier with a large database of non-obfuscated ones. As compared to these attempts, our proposal does not aim to deobfuscate, but can

2 https://github.com/CalebFenton/dex-oracle , last accessed March 2018



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. C.29. Distribution of the number of Goto instructions per line of code in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. C.30. Distribution of the number of NOP instructions per line of code in (a) obfuscated and (b) non-obfuscated apps.

serve as starting point to address this in future. In particular, the output of AndroDet is useful to spot the type of obfuscation at stake, which can be considered to apply focused deobfuscation techniques. Moreover, our approach considers several types of obfuscation.

7. Conclusion

Obfuscation is one of the main obstacles when it comes to Android app analysis. Thus, having a mechanism to detect the type of existing obfuscation (if any) can contribute saving resources for analysis. Indeed, particular analysis techniques may be applied once this detection has been done. To contribute in this direction, in this work AndroDet has been proposed. AndroDet shows promising accuracy ratios for detecting identifier renaming, string encryption, and control flow obfuscation. Moreover, it requires moderate training needs and can be configured to work in online basis, that is, with incremental training. To foster further research in this area, both AndroDet sources and the experimental dataset are freely available.

Several issues are devised as future research directions. First, addressing other types of obfuscation. Second, refining the feature set to improve the current accuracy of modules. Last but not least, extracting features by directly parsing the header of Dex files which will save more time and will compensate the limitations of Android reverse engineering tools.

Acknowledgments

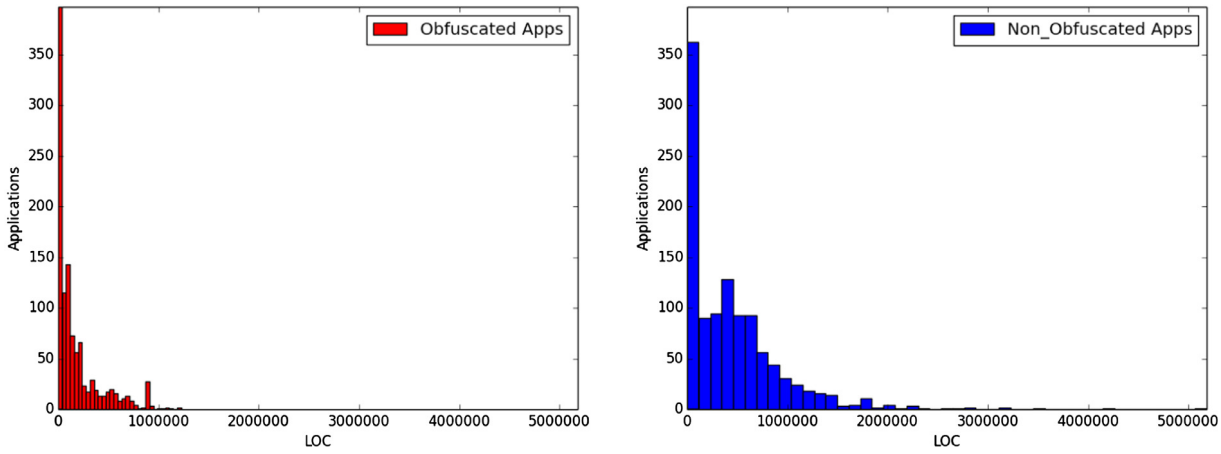
This work has been partially supported by MINECO grant TIN2016-79095-C2-2-R (SMOG-DEV) and CAM grant S2013/ICE-3095 (CIBERDINE), co-funded with European FEDER funds. Furthermore, it has been partially supported by the UC3M's grant Programa de Ayudas para la Movilidad. The authors would like to thank the Allatori technical team for its valuable assistance, and, also, the authors of the AMD and PraGuard datasets which made their repositories available to us. Finally, we would like to thank the anonymous reviewers for their comments.

Appendix A. Distribution of features for identifier renaming detection

This section presents the distribution of attributes in the methods and classes which were extracted from obfuscated and non-obfuscated samples of AMD dataset. (See Figs. A.8–A.17.)

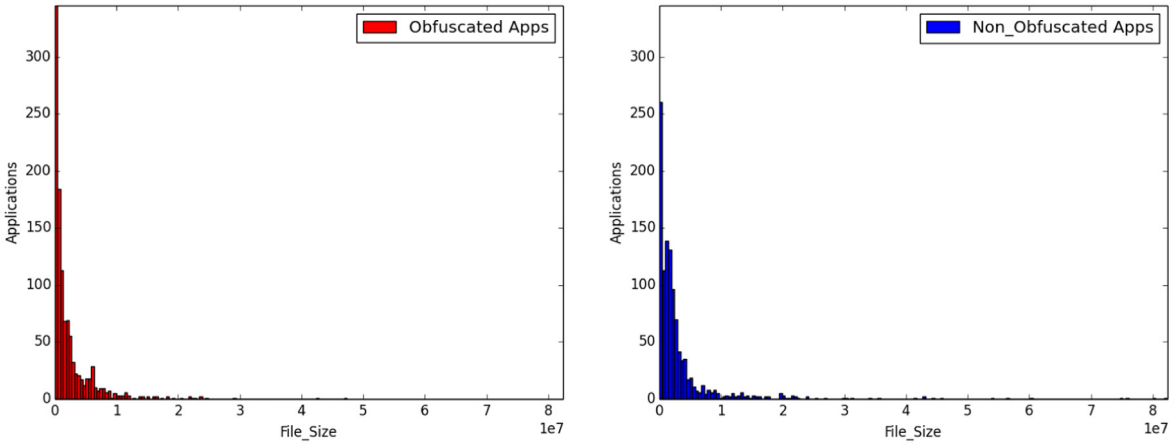
Appendix B. Distribution of features for string encryption detection

This section presents the distribution of attributes in the strings which were extracted from obfuscated and non-obfuscated samples of AMD dataset. (See Figs. B.18–B.25)



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. C.31. Distribution of the total number of lines of code in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Fig. C.32. Distribution of the total number of lines of code in (a) obfuscated and (b) non-obfuscated apps.

Appendix C. Distribution of features for control flow obfuscation detection

This section presents the distribution of attributes extracted from the CFG and Dalvik bytecode of obfuscated (from AMD dataset) and non-obfuscated (from F-Droid dataset) samples. (See Figs. C.26–C.32).

References

[1] A. Bianchi, Y. Fratantonio, A. Machiry, C. Kruegel, G. Vigna, S.P.H. Chung, W. Lee, Broken fingers: On the usage of the fingerprint API in android, in: NDSS'18, 2018.
[2] Smartphone os market share. https://www.idc.com/promo/smartphone-market-share/os. (Accessed 19 February 2018).
[3] Mobile malware evolution. https://securelist.com/mobile-malware-review-2017/84139/. (Accessed 14 March 2018).
[4] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, K. Zhang, Understanding android obfuscation techniques: A large-scale investigation in the wild, 2018. ArXiv preprint arXiv:1801.01633.
[5] V. Rastogi, Y. Chen, 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.
[6] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, F. Mercaldo, 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] Y. Duan, M. Zhang, A.V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, X. Wang, Things you may not know about android (un) packers: A systematic study based on whole-system emulation, in: NDSS'18, 2018.
[8] Y. Wang, A. Rountev, Who changed you?: obfuscator identification for android, in: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, IEEE Press, 2017, pp. 154–164.
[9] A. Bifet, R. Kirkby, Data Stream Mining a Practical Approach, Citeseer, 2009.
[10] F. Wei, Y. Li, S. Roy, X. Ou, W. Zhou, Deep ground truth analysis of current android malware, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA'17, Springer, Bonn, Germany, 2017, pp. 252–276.
[11] F-droid. https://f-droid.org. (Accessed 10 February 2018).
[12] T. Swearingen, W. Drevo, B. Cyphers, A. Cuesta-infante, A. Ross, K. Veeramachaneni, ATM: A distributed, collaborative, scalable system for automated machine learning, 2017.
[13] D. Maiorca, D. Ariu, I. Corona, M. Aresu, G. Giacinto, Stealth attacks: An extended insight into the obfuscation effects on android malware, Comput. Secur. 51 (2015) 16–31.
[14] Allatori. http://www.allatori.com/. (Accessed 10 February 2018).
[15] L.-K. Yan, H. Yin, Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis, in: USENIX Security Symposium, 2012, pp. 569–584.
[16] A. Desnos, G. Gueguen, Android: From reversing to decompilation, Proc. Black Hat Abu Dhabi (2011) 77–101.

[17] H. Meng, V.L. Thing, Y. Cheng, Z. Dai, L. Zhang, A survey of android exploits in the wild, Comput. Secur. (2018).
[18] Dexdump. http://googlesource.com/platform/dalvik/+/eclairrelease/dexdump/DexDump.c.(Accessed 10 February 2018).
[19] Dex2jar. https://bitbucket.org/pxb1988/dex2jar. (Accessed 10 February 2018).
[20] Androguard. http://github.com/androguard/androguard. (Accessed 10 February 2018).
[21] Apktool. https://ibotpeaches.github.io/Apktool. (Accessed 10 February 2018).
[22] K. Tam, A. Feizollah, N.B. Anuar, R. Salleh, L. Cavallaro, The evolution of android malware and android analysis techniques, ACM Comput. Surv. 49 (4) (2017) 76.
[23] Y. Wang, A. Rountev, Who changed you ? Obfuscator identification for android, 2017.
[24] R. Yu, Android packers: facing the challenges, building solutions, in: Proceedings of the 24th Virus Bulletin International Conference, 2014.
[25] B. Li, Y. Zhang, J. Li, W. Yang, D. Gu, Appspear: Automating the hidden-code extraction and reassembling of packed android malware, J. Syst. Softw. (2018).
[26] C. Collberg, C. Thomborson, D. Low, A Taxonomy of Obfuscating Transformations, Technical Report, 1997.
[27] S. Banescu, A. Pretschner, A tutorial on software obfuscation, Adv. Comput. (2018).
[28] V. Balachandran, D.J. Tan, V.L. Thing, et al., Control flow obfuscation for android applications, Comput. Secur. 61 (2016) 72–93.
[29] J. Li, D. Gu, Y. Luo, Android malware forensics: Reconstruction of malicious events, in: Distributed Computing Systems Workshops, ICDCSW, 2012 32nd International Conference on, IEEE, 2012, pp. 552–558.
[30] Dasho. https://www.preemptive.com/products/dasho/overview. (Accessed 10 February 2018).
[31] S. Dua, X. Du, Data Mining and Machine Learning in Cybersecurity, CRC press, 2016.
[32] I.H. Witten, E. Frank, M.A. Hall, C.J. Pal, Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann, 2016.
[33] Y. Ye, T. Li, D. Adjeroh, S. Iyengar, A survey on malware detection using data mining techniques, ACM Comput. Surv. 50 (3) (2017) 41.
[34] A. Tsymbal, The Problem of Concept Drift: Definitions and Related Work, vol. 106, Computer Science Department, Trinity College Dublin, 2004.
[35] J.P. Barddal, H.M. Gomes, F. Enembreck, B. Pfahringer, A survey on feature drift adaptation: Definition, benchmark, challenges and future directions, J. Syst. Softw. 127 (2017) 278–294.
[36] I. Žliobaitė, A. Bifet, J. Read, B. Pfahringer, G. Holmes, Evaluation methods and decision theory for classification of streaming data with temporal dependence, Mach. Learn. 98 (3) (2015) 455–482.
[37] H.M. Gomes, J.P. Barddal, F. Enembreck, A. Bifet, A survey on ensemble learning for data stream classification, ACM Comput. Surv. 50 (2) (2017) 23.
[38] A. Bifet, G. Holmes, R. Kirkby, B. Pfahringer, Moa: Massive online analysis, J. Mach. Learn. Res. 11 (May) (2010) 1601–1604.
[39] G.D.F. Morales, A. Bifet, SAMOA: scalable advanced massive online analysis, J. Mach. Learn. Res. 16 (1) (2015) 149–153.
[40] P. Reutemann, J. Vanschoren, Scientific workflow management with ADAMS, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2012, pp. 833–837.
[41] S. Hido, S. Tokui, S. Oda, Jubatus: An open source platform for distributed online machine learning, in: NIPS 2013 Workshop on Big Learning, Lake Tahoe, 2013.
[42] Vowpal. https://github.com/JohnLangford/vowpal_wabbit. (Accessed 12 February 2018).
[43] Streamdm. http://huawei-noah.github.io/streamDM. (Accessed 12 February 2018).
[44] N.Y. Kim, J. Shim, S.-j. Cho, M. Park, S. Han, Android application protection against static reverse engineering based on multidexing, J. Internet Serv. Inf. Secur. 6 (4) (2016) 54–64.
[45] H. Choi, Y. Kim, Large-scale analysis of remote code injection attacks in android apps, Secur. Commun. Netw. 2018 (2018).
[46] Y. Zhou, X. Jiang, Dissecting android malware: Characterization and evolution, in: Security and Privacy, SP, 2012 IEEE Symposium on, IEEE, 2012, pp. 95–109.
[47] Mobile malware mini dump. http://contagiominidump.blogspot.com. (Accessed 19 February 2018).
[48] Tree-based feature selection. http://scikit-learn.org/stable/modules/feature_selection.html. (Accessed 17 March 2018).
[49] F. Wei, Y. Li, S. Roy, X. Ou, W. Zhou, Deep ground truth analysis of current android malware, 2015, pp. 1–22.
[50] M. Grace, Y. Zhou, Q. Zhang, S. Zou, X. Jiang, Riskranker: scalable and accurate zero-day android malware detection, in: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, ACM, 2012, pp. 281–294.
[51] F.A. Narudin, A. Feizollah, N.B. Anuar, A. Gani, Evaluation of machine learning classifiers for mobile malware detection, Soft Comput. 20 (1) (2016) 343–357.
[52] J.S. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, Algorithms for hyper-parameter optimization, in: Advances in Neural Information Processing Systems, 2011, pp. 2546–2554.
[53] P. Domingos, G. Hulten, Mining high-speed data streams, in: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2000, pp. 71–80.
[54] N. Littlestone, M.K. Warmuth, The weighted majority algorithm, Inf. Comput. 108 (2) (1994) 212–261.
[55] A. Bifet, G. Holmes, B. Pfahringer, Leveraging bagging for evolving data streams, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2010, pp. 135–150.
[56] M.A. Thalor, S. Patil, Ensemble for non stationary data stream: Performance improvement over learn++. NSE, in: Information Processing, ICIP, 2015 International Conference on, IEEE, 2015, pp. 225–228.
[57] C. Salperwyck, V. Lemaire, C. Hue, Incremental weighted naive bays classifiers for data stream, in: Data Science, Learning by Latent Structures, and Knowledge Discovery, Springer, 2015, pp. 179–190.
[58] I. Steinwart, A. Christmann, Support Vector Machines, Springer Science & Business Media, 2008.
[59] J.R. Quinlan, Induction of decision trees, Mach. Learn. 1 (1) (1986) 81–106.
[60] L. Breiman, Random forests, Mach. Learn. 45 (1) (2001) 5–32.
[61] Evaluateprequential. https://www.cs.waikato.ac.nz/~abifet/MOA/API/classmoame_1_1tasks_1_1_evaluate_prequential.html.(Accessed 12 March 2018).
[62] M. Dalla Preda, F. Maggi, Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology, J. Comput. Virol. Hacking Tech. 13 (3) (2017) 209–232.
[63] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-khaligh, S. Malek, Department of computer science obfuscation-resilient, efficient, and accurate detection and family identification of android malware, 2015, pp. 1–15.
[64] F. Zhang, H. Huang, S. Zhu, D. Wu, 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, ACM, 2014, pp. 25–36.
[65] L. Glanz, S. Amann, M. Eichberg, M. Reif, B. Hermann, J. Lerch, M. Mezini, CodeMatch: obfuscation won't conceal your repackaged app, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 638–648.
[66] R. Baumann, M. Protsenko, T. Müller, Anti-ProGuard: Towards automated deobfuscation of android apps, in: Proceedings of the 4th Workshop on Security in Highly Connected IT Systems, ACM, 2017, pp. 7–12.
[67] W. Yoo, M. Ji, M. Kang, J.H. Yi, String deobfuscation scheme based on dynamic code extraction for mobile malwares, 2 (2016) 1–8.
[68] B. Bichsel, V. Raychev, P. Tsankov, M. Vechev, Statistical deobfuscation of android applications, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2016, pp. 343–355.

Omid Mirzaei is a Ph.D. candidate in the Computer Security Lab (COSEC) at the Department of Computer Science and Engineering of Universidad Carlos III de Madrid. His Ph.D. is funded by the Community of Madrid and European Union for the research project CIBERDINE. His main area of research is computer security. However, he is particularly interested in reverse engineering, malware analysis, and the study of protocols for secure communication using artificial intelligence tools and techniques. In addition, he is eager to tackle security issues from a multi-objective perspective, i.e. trying to deal with such problems by consuming the least possible amount of in hand resources. Currently, he is working on security analysis, malware analysis and risk management with a special focus on smartphone devices and is supervised by Dr. Juan Tapiador and Dr. Jose M. de Fuentes.

Dr. Jose Maria de Fuentes is visiting lecturer with the Computer Science and Engineering Department at Universidad Carlos III de Madrid, Spain. He is Computer Scientist Engineer and Ph.D. in Computer Science by Universidad Carlos III de Madrid. He has published +30 articles in international conferences and journals, all of them related to applied cryptography and privacy preservation. He is member of the Editorial board of Wireless Networks journal, as well as member of the TPC of +30 international conferences and workshops. He has participated in 6 national R+D projects and contracts. Since 2015 he has been appointed National Secretary for the Spanish mirror of ISO/IEC JTC 1/SC 27.

Dr. Juan Tapiador is Associate Professor in the Computer Security (COSEC) Lab at Universidad Carlos III de Madrid, Spain. His research focuses on engineering secure software and systems. His main research areas include malware analysis, reverse engineering, anomaly and intrusion detection, and automating defense and analysis techniques. He holds a M.Sc. in Computer Science from the University of Granada (2000), and a Ph.D. in Computer Science (2004) from the same university.

Dr. Lorena Gonzalez-Manzano is visiting lecturer with the Computer Science and Engineering Department at Universidad Carlos III de Madrid, Spain. She is Computer Scientist Engineer and Ph.D. in Computer Science by Universidad Carlos III de Madrid. Her research interests are on Internet of Things and cloud computing security. She has published +20 papers in national and international conferences and journals and she is also involved in national R+D projects. She is member of the TPC of +15 international conferences and workshops as well as member of Editorial board of Future Generation Computer Systems journal.