Obfusifier: Obfuscation-Resistant Android Malware Detection System

Zhiqiang Li[1], Jun Sun[1], Qiben Yan[1,2], Witawas Srisa-an[1] and Yutaka Tsutano[1]

[1] Department of Computer Science and Engineering, University of Nebraska–Lincoln
Lincoln, NE 68588, USA
[2] Department of Computer Science and Engineering, Michigan State University
East Lansing, MI 48824, USA
{zli,jsun,qyan,witty,ytsutano}@cse.unl.edu

Abstract. The structure-changing obfuscation has become an effective means for malware authors to create malicious apps that can evade the machine learning-based detection systems. Generally, a highly effective detection system for detecting unobfuscated malware samples can lose its effectiveness when encountering the same samples that have been obfuscated. In this paper, we introduce Obfusifier, a highly effective machine-learning based malware detection system that can sustain its effectiveness even when malware samples are obfuscated using complex and composite techniques. The training of our system is based on obfuscation-resistant features extracted from unobfuscated apps, while the classifier retains high effectiveness for detecting obfuscated malware. Our experimental evaluation shows that Obfusifier can achieve the precision, recall, and F-measure that exceed 95% for detecting obfuscated Android malware, well surpassing any of the previous approaches.

Keywords: Malware detection · Android · Obfuscation.

1 Introduction

Code obfuscation is a common approach used by developers to help protect the intellectual properties of their software. The goal of obfuscation is to make code and data unreadable or hard to understand [16]. This, in effect, makes reverse-engineering of their applications more difficult. Typically, there are three major types of obfuscation methods: (1) trivial obfuscations, which most tools can easily handle; (2) data-flow and control-flow obfuscations, which can be Detectable by Static Analysis (DSA); and (3) encryption-based obfuscations, which often involve some forms of encryption to hide the actual code and data.

Recently, various obfuscation techniques have been applied on malicious apps to evade the security analysis. These techniques are especially effective in defeating existing malware and virus scanners, which often rely on signature matching or program analysis. As will be shown in Section 3, we apply DSA based obfuscation techniques to known malware samples and evaluate them by VirusTotal [7].

The analysis results indicate that many existing techniques deployed by VirusTotal cannot detect obfuscated malware samples and would indicate them as benign.

These DSA based obfuscation techniques are effective in defeating virus scanners, because they change the flow of the program by adding (e.g., junk code insertion), reordering (e.g., code reordering, function inlining, function outlining), or redirecting code (e.g., method indirection). These code manipulations can change the signatures of a program and complicate program analysis. In addition, these techniques also change method, variable, and class names so that static analysis techniques that look for previously known values would fail to locate them. Also, note that encryption-based obfuscation techniques are effective in defeating malware detectors because they "hide" the entire code-base and data through encryption. Prior to running, these encrypted applications must be decrypted to reveal the real codes (that may or may not have been obfuscated using DSA techniques) and data for execution. Encryption-based obfuscation is beyond the scope of this work.

Recently, machine learning has become widely used for Android malware detection in the state-of-the-art systems [20, 9, 12, 24, 27, 23]. These existing systems extract features from benign and malware Android samples to build classifiers to detect malware. Currently, the samples used in building classifiers are not obfuscated. However, one recent work [25] as well as Section 3 have shown that when obfuscated Android malware samples are submitted to these classifiers, they can be miscategorized since the features used by these classifiers are now more ambiguous due to obfuscation [19]. In this paper, we propose Obfusifier, a machine-learning based malware detector that is constructed using features from unobfuscated samples but can provide accurate and robust detection results when obfuscated samples are submitted for detection.

Our key insight is that there are portions of codes that cannot be obfuscated, because the obfuscation of them will break the functionality. One of these portions is *the API invocations into the Android framework.* As a result, our feature selection focuses mainly on the usage of Android APIs. Our approach then extracts features that are related to such usage. In total, we extract 28 features to build our classifier using 4,300 benign apps and 4,300 malware samples obtained from VirusShare; these apps are not obfuscated. We then test our system using 568 obfuscated malware. The result indicates that our system can achieve 95% precision, recall, and F-measure, corroborating the obfuscation resilience of Obfusifier.

2 Background on Code Obfuscation

In this work, we use Alan, a Java-based code obfuscation tool for Android. Next, we describe the obfuscation features supported by Alan. As will be discussed in Section 5, we employed all techniques in a composite fashion to obfuscate our malware samples to make them as challenging as possible to be detected by Obfusifier.

Disassembling & Reassembling. The Dalvik bytecode in the DEX file of the Android app can be disassembled and reassembled. The arrangement of classes, strings, methods in the DEX files can be changed in different ways. In other words, the architecture or the arrangement of the DEX files can be modified, and this transformation creates changes that significantly alter the structures of the program, rendering signature-based detector ineffective.
Repackaging. Developers must sign their Android app before it is released to the market. Cybercriminals can unzip the released Android app and repack it via tools in the Android SDK. After repacking, hackers must sign the repackaged app with their own keys, because they do not have the developers original keys, this newly signed app does not have the same checksum with the original app. This process neutralizes the effectiveness of malware detectors that compare checksums primarily for detection.
Data Encoding. The strings and arrays in the DEX files can be used as signatures to identify malicious behaviors. Encryption of strings and arrays can make signature-based detection ineffective [14].
Code Reordering. This feature aims to change the order of the instructions randomly, and the original execution order is preserved by inserting `goto` instructions. Because this reordering is random, the signature generated by this malware would be significantly different from the signature of the original malware. This is by far the strongest obfuscation technique for evading the signature-based detectors [32].
Junk Code Insertion. This technique does not change the programming logic of the code. As such, compared with other transformations, its impact towards the detector is less significant, and malware only obfuscated with Junk Code Insertion are very likely to be detected [17]. Three types of junk codes are inserted including `nop` instructions, unconditional jumps, and additional registers for garbage operations.
Identifier Renaming. This transformation modifies package and class names with random strings. It can be used to evade the signature-based detection.
Call Indirection. Some malware scanners take advantage of the structure of the method graphs to generate signatures. The original method call can be modified by inserting a newly and randomly generated method before calling the original method. This transformation can insert many irrelevant nodes into the method call graph of an obfuscated app. If a detector is relying on a signature based on a method call graph, this obfuscation technique can be effective in evading the detection. Furthermore, a machine learning detector based on method call graph features would also likely fail to detect malware samples employing this obfuscation technique.

3 Effects of Obfuscation on Malware Detection

Obfuscation techniques that can transform the structure of an application has the potential to allow malware to evade detection of many anti-virus scanners. To elaborate and quantify the magnitude of this phenomenon, we investigated

the effects of obfuscations on the effectiveness of existing virus scanners. The data collection process to conduct our experiments (described next) and the subsequent evaluation of our proposed system is described in Section 5.

In the first experiment, we assessed the effect of obfuscation on the accuracy of detection by about 60 scanners deployed by VirusTotal [7]. The experiment involved randomly selecting 30 malware samples from VirusShare (we downloaded them in June, 2018). We then applied obfuscation using ALAN, a Java-based code obfuscation tool which is capable of applying several types of structure-altering transformations including code reordering, junk code insertion and call indirection directly on DEX code of an Android app [13, 25]. Once these apps have been obfuscated, we resubmitted them for scanning again on VirusTotal. We report the scanning result in Figure 1.

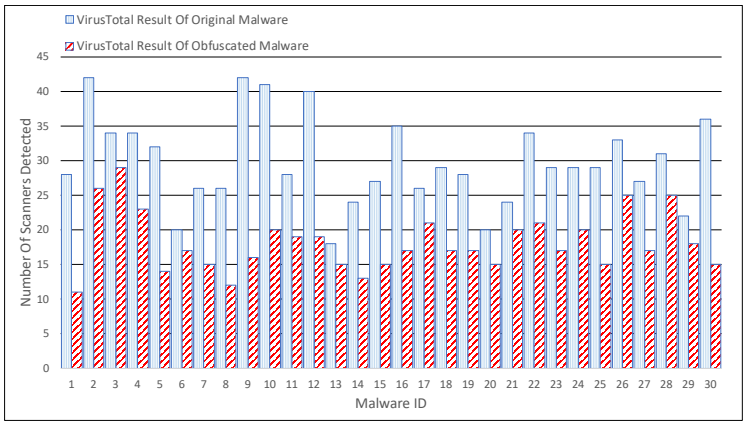

Fig. 1. The Difference in Detection Rate of Original and Obfuscated Malware

In the figure, the horizontal axis lists malware ID (from 1 to 30). The vertical axis presents how many anti-virus scanners identify an app as malware. The light blue bar is the detection number for the original app, and the red twilled bar is the result for the obfuscated app. The number of scanners that can accurately identify each obfuscated app as malicious decreases dramatically. The biggest drop occurs in App 9 as its unobfuscated version was detected by 42 scanners and its obfuscated version is only detected by 16 scanners, a reduction of 62%.

In the second experiment, we focused on the accuracy of 14 popular scanners in detecting obfuscated malware. We randomly obfuscated 1,540 apps using ALAN. Table 1 shows the detection difference between these 1,540 unobfuscated malicious apps and their obfuscated versions. The scanner Antiy-AVL can identify 1,427 as malware before obfuscation, but can only identify 260 obfuscated versions. The difference for McAfee and Symantec is 641 before and after obfuscation, which is surprisingly high. Ad-Aware and Baidu cannot detect obfuscated malware at all. We checked 60 scanners, and the number of scanners which could still identify the obfuscated apps as malicious decreased by 34.4% on average. Prior work called DROIDCHAMELEON [25] has shown that 10 popular anti-virus

products such as Kaspersky, AVG and Symantec, lose their detection effectiveness when the malware samples are obfuscated.

Scanner	Number of detected (original)	Number of detected (obfuscated)	Difference
Antiy-AVL	1427	260	1167
MAX	1429	463	966
Comodo	999	122	877
F-Prot	830	54	776
Alibaba	975	291	684
K7GW	1348	679	669
McAfee	1446	805	641
Symantec	763	122	641
McAfee-GW-Edition	1265	669	596
DrWeb	1119	607	512
BitDefender	464	20	444
eScan	434	2	432
Ad-Aware	430	0	430
Baidu	308	0	308

Table 1. The Difference of Detection Rate By Scanners

We conducted the third experiment to better understand the effects of obfuscation on malware detection effectiveness of existing scanners. To do so, we focused our analysis on a malware sample that belongs to Adware:android/dowgin [1] family, which is an advertising module displaying advertisement while leaking or harvesting information such as its IMEI number, location, and contact information from the device.

We then obfuscated this malware sample using ALAN [6]. Before it was obfuscated, 20 scanners from the VirusTotal [7] were able to identify it as malware. However, after obfuscation, only 8 scanners could detect it. We statically analyzed this app and its obfuscated counterpart. We checked its method call graph, and found that there were 4,948 methods, 7,244 function calls prior to obfuscation. After obfuscation, the number of methods increased to 6,387 and the number of function calls increased to 8,683. Obviously, some methods were inserted as part of the obfuscation process.

Fig. 2. Obfuscation Process

Figure 2 illustrates this obfuscation process. we have A→B as the original function call, but in the obfuscated graph, we have A→C, C→B instead. This is called Call Indirection. The structure of the original method graph is modified, and scanners which is based on the signatures from the call graph would not be able to detect such changes.

We also compared their DEX codes. There were 93,077 lines in the original DEX file, but there were 148,819 lines after obfuscation. Scanners which rely on the order of the instruction as signatures would be ineffective by such changes. Prior work called RevealDroid [19] has shown that even for machine learning-based detectors, obfuscation is still problematic.

Clearly, there is a need to create a malware detector that maintains its effectiveness in spite of obfuscation. Our approach, Obfusifier applies static analysis to identify code that cannot be obfuscated and then efficiently extracts effective features to build a machine learning-based detection system. In the next section, we introduce our proposed system.

4 Introducing Obfusifier

The main goal of the Obfusifier is to identify Android malware which has been transformed via different obfuscation techniques and difficult to detect via common antivirus scanners. Thus, the selected features must satisfy the following four policies. First, these features must give a good representation of the difference between malware and benign apps. Second, a very high detection accuracy must be achieved when handling malware without obfuscation. Third, the detection time must be sufficiently short for real-world application scenarios. Fourth, the system must be resilient when used to detect obfuscated malware.

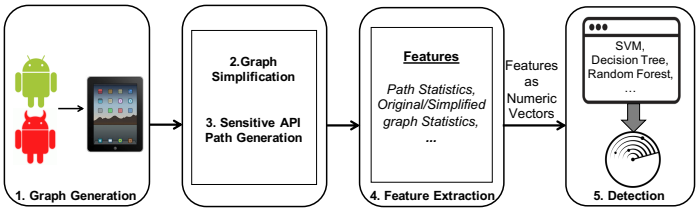

Fig. 3. System Architecture

In this section, we describe the architectural overview of our proposed system, which operates in five phases: Graph Generation, Graph Simplification, Sensitive API Path Generation, Feature Extraction, and Malware Detection, as shown in Figure 3. Next, we will describe each phase in turn.

4.1 Graph Generation

Method graph can be used as a good representation of the malware structure. It represents the calling relationship between different methods and subroutines. Each node in the graph represents a method, and a directed edge from one node to the other shows their calling relationship. We implement Obfusifier based on Jitana [28], a high-performance hybrid program analysis tool to perform static and dynamic program analysis. Jitana can analyze DEX file, which includes the user-defined code, third party library code, framework code (including

implementations of various Android APIs), and underlying system code. JITANA analyzes the classes to uncover all methods and generates the method graph for the app. OBFUSIFIER takes advantage of the calling relationship of methods to detect malware. As shown in Figure 4, blocks represent methods, and directed edges indicate calling relationship among methods. Each block contains the name of the method, its modifiers and the class name which this method belongs to. OBFUSIFIER captures the interactions of these methods, and understands the semantic information that can help detect malware.

There are three types of methods in the graph: Android API method, system-level method and user-defined method. All of these methods can be exploited by malware writers to conduct malicious behaviors. In terms of code obfuscation, APIs and system-level methods cannot be transformed by code obfuscation techniques, otherwise the app will fail to run. The user-defined methods and the classes the methods belong to can be renamed, so that the malware can evade the antivirus scanners. As a result, only relying on the original method graph may not be enough to build a obfuscation-resistant detector due to the negative impact of code obfuscation. Lightweight features can be extracted from these method graphs to build the malware detection system.

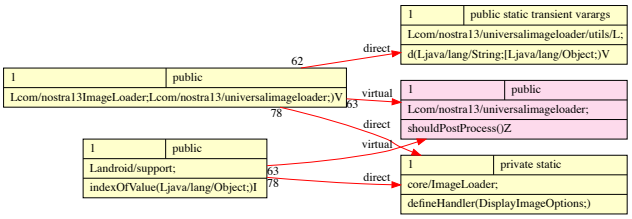

Fig. 4. Method Graph

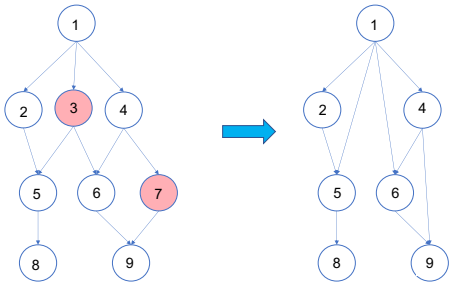

Fig. 5. Graph Simplification Process

4.2 Graph Simplification

Our key insight is that *the Android APIs and system-level methods cannot be transformed by code obfuscation*, and this characteristics can be exploited to extract obfuscation-resistant features. Android APIs are published by Google, so we can easily create a list of these APIs. System-level methods include the Android OS source code and the Linux kernel source code, so it is not as convenient to gather all these methods and therefore, we do not collect them. We simply rely on the list of Android APIs that we collected.

In order to generate obfuscation-resistant graphs, we only keep the Android APIs in the original method graph, and ignore the system-level methods, user-defined methods and those from third party libraries. For example, as shown in the Figure 5, nodes 1, 2, 4, 5, 6, 8 and 9 are Android APIs, and node 3 and node 7 are system-level or user-defined method. In this situation, our system simply ignores nodes 3 and 7, and generates a new call edge from node 1 to node 5 and another edge from node 4 to node 7. By doing so, we remove two nodes and combine four method calls into two.

By performing graph simplification, we are able to reconstruct a graph that is obfuscation-resistant while preserving the structural and semantic information with respect to Android API usage of the original graph. In addition, the API-only graph contain as much as an order of magnitude less information than the original graph, allowing feature extraction to be much faster especially during the path traversal phase.

4.3 Sensitive API Path (SAP) Generation

SAP is the program execution path from one node to the other in the API-only graph. An SAP can be used to differentiate between the malicious and benign behaviors. In order to generate SAPs, we need to select the critical APIs which are used for path generation, since these APIs reflect the semantic information about the behaviors of apps. We analyze the call frequency of APIs, and keep APIs which are used only by malware because they can directly reflect the malicious behaviors. We also extract some frequently used APIs by both malware and benign apps in common, because even though they are used by both, the additional program context (e.g., method call characteristics) can still represent the difference between malware and benign apps. In the API-only graph, all nodes whose in-degree are zero are considered as sources, and nodes whose out-degree are zero are considered as destinations. Obfusifier generates SAPs from sources to destinations via depth first search (DFS).

Figure 6 illustrates the process to generate SAPs. In the figure, there are two sources (Node 1 and Node 2, marked as green) and two destinations (Node 4 and Node 10, marked as red) in the graph. Node 1 and Node 2 are sources (in-degree is zero) as there are no edges flowing into them. Node 4 and Node 10 are destinations as they are selected and frequently used APIs. Starting from Node 1, Node 2, and ending with Node 4, Node 10, four SAPs can be identified: 1→4, 1→5→9→10, 2→5→9→10 and 2→6→9→10.

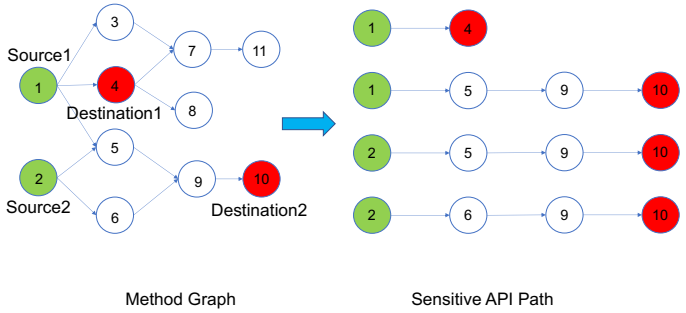

Fig. 6. Sensitive API Path (SAP) Generation

SAP reflects the running behaviors of apps, whose patterns can be useful in distinguishing between malicious apps and benign ones.

4.4 Feature Extraction

We now describe the features, which our system extracts from the original graphs, API-only graphs, and SAPs.

Path Statistic Feature (F1). We collect seven statistic features from Sensitive API Path. These features include the the lengths of the longest and short paths, the number of paths, the sum of lengths of all paths, the average length per path, the number of methods in all paths, and the average number of methods per path. These statistical features can indicate path characteristics and represent malicious behaviors. For example, malware which conducts malicious behaviors tends to generate shorter and less paths than benign apps. Since the paths in API-only graphs only consist of APIs, this feature set is not affected by code obfuscation. These features are concatenated to construct a numeric vector to reflect the unique characteristics of app behaviors that can further preserve rich information in these paths.

Simplified Graph Statistic Feature (F2). We select eight features from the simplified graph. They are the number of methods, the number of classes and the number of edges in the graph, graph density, the average in-degree and out-degree of the graph, the number of sources (nodes of which in-degree are zero) and destinations (nodes of which out-degree are zero). Compared with the original graph, the simplified graph is much less obfuscated, because it does not include the renamed user-defined classes and methods.

Original Graph Statistic Feature (F3). We also collect eight features from the original graph. These features are the same as F2. Even though some of the methods in the original graph are obfuscated, we still think these graphs can reflect the malicious behaviors. Keeping features from the original graph might still be useful to identify malware, whether it is obfuscated or not.

Other Statistic Feature (F4). We save the original graph, simplified graph, and SAP in separate files, and use the sizes of these files to form three new numeric features. We assume that the size of file can reflect the amount of generated information which indicates the complexity of these graphs and paths. Besides, we also calculate the ratio of the number of methods in original graph to the number in the simplified graph, and the ratio of the number of classes in original graph to the number in the simplified graph. The ratio can reflect the level of obfuscation accurately, and we hypothesize this will also contribute to the malware detection. Finally, we form F4 as a vector of five features.

4.5 Detection

In the Detection phase, we apply three well-recognized machine learning algorithms to determine if an Android app is malicious or benign. Our proposed system utilizes four different features (F1 - F4) as previously mentioned. Intuitively, we consider that each of the four feature sets can reflect malicious behaviors in some specific patterns. For API-only Graph Statistic Feature, because we remove all the user-defined classes and methods, which are usually transformed by obfuscation techniques, to generate simplified graph, these features are less likely affected by obfuscation. Besides, these features also reflect the structural difference between malware and benign apps. For example, we find that benign apps usually have more sources, destinations, classes and methods than malware. Thus, we need the Original Graph Statistic Feature because the graph simplification process also eliminates some of the original structural characteristics of graphs. The size of files where we store graphs and paths can also help build our detection system, for example, we observe that the size of the file storing graph and paths from malware is usually smaller than benign apps. We also notice that the graph density from malware is greater than benign apps, so we gather these file size and graph density information to form Other Statistic Feature.

We evaluate the performance of our system by using different feature sets individually. In addition, we also concatenate different feature sets to construct the combined new feature set and assess its impact on the detection result. In terms of the classification policy, we apply three popular algorithms: Decision Tree, Random Forest, and Support Vector Machine (SVM) [26, 22, 15]. These machine learning algorithms have been shown to achieve superior performance in addressing classification problems, which are integrated into Obfusifier.

5 Empirical Evaluation

To evaluate Obfusifier, we show its detection performance in terms of accuracy, precision, recall, and F-measure. we also illustrate its resistance against obfuscation, and ultimately its runtime performance. We first present the process of collecting our experimental apps, both benign and malicious, and explain

how to transform malware using obfuscation techniques. Next, we show our detection results based on different sets of features. We also compare our system with several related approaches. Finally, we present the runtime performance of Obfusifier.

5.1 Experimental Objects

To evaluate the performance of our proposed system, we collected a dataset containing both malware and benign apps. We downloaded 24,317 malware samples from VirusShare [5]. Compared to Android Genome Project [34], which is often used by many researchers [24, 8], we included more malware samples and they are also newer. However, they also include many of the samples in the Android Genome Project. For benign apps, we collect 20,795 apps from APKPure [4], a third party website providing Android apps. Note that we also used these collected apps to conduct experiment in Section 3.

In order to avoid polluting our benign dataset with malware samples, we cross-checked all apps downloaded from APKPure with VirusTotal, and remove those apps identified as malware by VirusTotal from the benign dataset. After we completed the cross-checking process, there are only 11,238 apps left for the benign dataset. This checking process took 29 days. Also note that all the samples in the malware dataset are also identified by VirusTotal as malicious.

5.2 Experimental Methodology

To evaluate the performance of our system, and guarantee the balance of the data, we randomly chose 4,300 malicious samples, 4,300 benign apps as training/testing samples from our dataset. We also applied 10-fold cross validation.

In order to verify Obfusifier's ability to resist to the code obfuscation, we randomly choose another 568 benign apps and 568 malware as additional testing set. We transform the additional 568 malicious samples using Alan by applying all its transformations mentioned in Section 2.

As previously mentioned, our system utilizes four sets of features (F1, F2, F3, F4) to construct our classifier and perform detection. To evaluate the classification performance of the system, four metrics are calculated. They are Accuracy, Precision, Recall and F-measure. We also assess the performance of our system by combinations of different sets of features. For example, by concatenating F1 and F2 (F1 U F2), we form a new feature vector. Besides, we also compare our system with several popular approaches based on similar datasets.

Our runtime evaluation was conducted using Macbook Pro with a dual-core 2.8 GHz Intel Core i7 running OS-X High Sierra and 16 GB of 1.33 GHz main memory.

5.3 Detection Result

We discuss two usage scenarios in this section. The first scenario is when we evaluate our classification system by 10-fold cross validation. All samples in the

dataset are the original (non-obfuscated) apps. This experiment is conducted to show that the classifier is effective and can detect unobfuscated malware with high accuracy. In a typical application, we imagine that security analysts would use obtainable, unobfuscated malware and benign samples for training and testing. Table 2 reports our result.

In the second scenario, we continued to use the original unobfuscated samples as in the first scenario for training; i.e., we used the same classifier built in the first scenario. However, we expand the testing dataset to include 568 more benign apps and then 568 more obfuscated malware samples (using ALAN) so that we can evaluate the ability of our system to maintain accurate detection in spite of obfuscation. Note that we applied all obfuscation methods supported by ALAN to make detection more challenging and our testing dataset also includes the same number of unobfuscated benign apps to maintain balance.

Table 3 shows the result of the second scenario (with obfuscation), in which all the testing malware samples are obfuscated. But above all, in both cases, there are not obfuscated apps in the training dataset, which means we do not need obfuscated apps in the training phase, and this characteristic guarantees that our system is robust and able to resist to obfuscated malware. The most powerful strength of our system is to identify obfuscated malware without training with obfuscated apps. Next, we discuss the results based on each feature.

Result Based on F1. Based on the the Path Statistic Feature (F1), we implement and evaluate our learning-based system. Table 2–F1 shows the detection result without obfuscation in terms of three approaches: SVM, Decision Tree and Random Forest. F1 is constructed by seven statistic features from Sensitive API Path (SAP). All the SAPs are generated from the API-only graphs, in which only methods that cannot be obfuscated are kept. Because obfuscation has little or no effects on this graph, this feature set is important to build the proposed obfuscation-resistant malware detection system.

In the first scenario (without obfuscation), we calculate the four metrics as shown in Table 2–F1 for each classification technique based on F1. The Random Forest and Decision Tree achieve the F-measure of 87.9% and 85.3% respectively. On the other hand, SVM only yields the F-measure of 60.2%. The Random Forest also has the accuracy of 87.6%, which outperforms the SVM and Decision Tree. This result indicates that our system can incorrectly detect malware if we only rely on F1.

In the second scenario, we assess our system with obfuscated apps. As shown in Table 3–F1. Similar to the case without obfuscation, Random Forest performs better than SVM and Decision Tree. It has the accuracy of 89.7% and the F-measure of 89.8%. This result shows that our system is somewhat effective when identifying obfuscated Android malware. Interestingly, by checking accuracy and F-measure for F1, the result with obfuscation in Table 3–F1 is slightly better than the one without obfuscation in Table 2–F1. This is because the impact of these transformations on the SAP feature is minor, so the system trained using SAP can resist obfuscation naturally. However, the SAP is from the simplified graph, which removes many user-defined methods from the orig-

inal graph. Because of this, some contexts of the program which is helpful to recognize the non-obfuscated malware are missing. As such, F1 performs better when handling obfuscated malware.

Result Based on F2. API-only Graph Statistic Feature (F2) is the feature vector directly form the the simplified graph. This feature is significant because it reflects the structural difference between malware and benign apps, and the influence of obfuscation on F2 is very small due to the elimination of all the newly added methods or renamed methods (Junk Code and Call Indirection) in the obfuscated and original graph.

Table 2–F2 shows the evaluation result without obfuscation. For F2, Random Forest achieves the best accuracy of 92.9%. It also attains the highest F-measure of 93.1% with 91.0% precision and 95.2% recall. Obviously, the result based on F2 is better than F1. it indicates that features directly from the graph are more effective than features from paths.

Table 3–F2 shows that our system is very effective even when dealing with obfuscated malware. In terms of Random Forest, we can achieve the very high accuracy 94.3% and F-measure 94.6%. This result validates out assumption that features from these simplified API-only graphs, in which obfuscated methods are removed, are very effective in identifying malware and resisting the negative impact of code obfuscation. As such, system trained based on F2 is more obfuscation-resistant.

	F1			F2			F3			F4			F1∪F2∪F3∪F4		
	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)
I. Accuracy	71.3	85.3	87.6	70.2	91.0	92.9	69.6	92.3	94.0	63.9	90.8	92.6	63.9	94.0	95.5
II. Precision	97.8	82.8	85.7	99.6	89.9	91.0	99.9	90.2	92.2	99.9	88.9	91.3	99.9	92.4	93.9
III. Recall	43.5	89.1	90.3	40.5	92.3	95.2	39.2	95.0	96.2	27.7	93.2	94.3	27.9	95.9	97.3
IV. F-Measure	60.2	85.3	87.9	57.6	91.1	93.1	56.3	92.5	94.1	43.4	91.0	92.7	43.6	94.1	95.5

Table 2. The performance of Obfusifier on non-obfuscated apps using five different features (F1 – F4, F1UF2UF3UF4) and three different Machine Learning algorithms: Support Vector Machine (SVM), Decision Tree (DT) and Random Forest (RF).

	F1			F2			F3			F4			F1∪F2∪F3∪F4		
	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)
I. Accuracy	73.6	89.2	89.7	71.3	92.9	94.3	51.5	90.9	80.6	50.1	89.3	91.5	50.0	93.3	90.2
II. Precision	99.3	87.9	89.1	100.0	90.5	91.2	100.0	91.7	90.4	0	88.9	92.3	0	92.5	92.7
III. Recall	47.4	91.0	90.5	42.7	95.8	98.2	1.9	89.9	68.4	0	89.9	90.7	0	94.2	87.3
IV. F-Measure	64.2	89.4	89.8	59.8	93.1	94.6	3.8	90.8	77.9	0	89.4	91.5	0	93.4	89.9

Table 3. The performance of Obfusifier with obfuscated apps as testing set

Result Based on F3. F1 and F2 are created based on API-only graphs, in order to reduce the impact of code obfuscation on malware detection. Based on our reported results, these two feature sets not only help to identify non-obfuscated

apps, but also show a remarkable efficacy when dealing with obfuscated malware. However, when graphs are simplified, some structural information which is beneficial to distinguish non-obfuscated malware might be lost. In case that original malware samples are also available, there is a potential to improve effectiveness by extracting features from the original method graph to form a feature set called Original Graph Statistic Feature (F3). The meaning of each feature in F3 is exactly the same as F2.

As illustrated in Table 2–F3, F3 achieves higher performance than F1 and F2 in all three classification techniques. Random Forest performs better than Decision Tree and SVM for F3 as it attains F-measure of 94.1% while the other two approaches (SVM and Decision Tree) achieve 56.3% and 92.5%, respectively.

For obfuscated malware, the performance of F3 is not as good as F1 and F2. As illustrated in Table 3–F3, most of the metrics show F3 cannot handle the obfuscated apps as good as F1 and F2. For example, in terms of Random Forest, F3 only has F-measure of 77.9%, which is lower than F1 and F2, which achieve 89.8% and 94.6% respectively. As such, F3 alone is not a sufficient feature set to achieve obfuscation-resistant capability.

Result Based on F4. We transform the sizes of several files, the ratio of the number of methods in original graph to the number in the simplified graph, and the ratio of the number of classes in original graph to the number in the simplified graph into a new feature, referred to Other Statistic Feature (F4). We assume that these file sizes and the ratios are also efficient features for distinguishing between malware and benign apps.

Table 2–F4 shows the detection result on non-obfuscated malware. Random Forest achieves the highest accuracy of 92.6% and F-measure of 92.7%, outperforming SVM and Decision Tree. Table 3–F4 illustrates the result with obfuscation. Random Forest also performs best yielding F-measure of 91.5%. Results based on F4 verify our assumption, and these sizes of files and ratios can provide another efficient feature to build the malware detection system.

Result Based on F1 U F2 U F3 U F4. By aggregating all our feature sets, as shown in Table 2, using Random Forest, we achieve the accuracy of 95.5% and F-measure of 95.5%. Table 3 shows that the combination of all feature sets also works well for obfuscated malware.

5.4 Comparison with Related Approaches

Next, we compare the performance of Obfusifier with other research efforts including RevealDroid [19], MUDFLOW [12], Adagio [20] and Drebin [9]. More information about these systems are available in Section 7.

In this work, we relied on the data provide in the RevealDroid paper as a base for comparison. They conducted an investigation that compared the detection performance of RevealDroid with the other three systems. Thus, we simply compared our system's performance against the reported performance.

Another noticeable difference is that RevealDroid obfuscated their malware using DroidChameleon [25]. RevealDroid applies four sets of transforma-

tions on there dataset including call indirection, rename classes, encrypt arrays, and encrypt strings. We, on the other hand, obfuscated our dataset with ALAN, and enabled all transformations described in Section 2. In our approach, "Data Encoding" technique includes the "Encrypt Arrays and Encrypt Strings" by DroidChameleon, and our "Identifier Renaming" includes "Rename Classes" by DroidChameleon. The level of obfuscation in our dataset is higher than REVEALDROID, so our transformed malware should be more difficult to detect.

The malicious apps used to investigate REVEALDROID are from Android Malware Genome [34], the DREBIN dataset [9] and VirusShare [5]. Our malicious dataset is only from VirusShare. However, the samples on VirusShare contain similar apps in Android Malware Genome and DREBIN dataset. The similarity of the dataset ensures the fairness of comparisons.

When comparing with the other four systems, we consider two scenarios. The first scenario is testing the non-obfuscated malware (without obfuscation). The second scenario is testing the obfuscated malware (with obfuscation). In the first scenario, REVEALDROID splits a dataset including 1,742 benign apps and 7,989 malicious ones into two parts evenly. One part is the training dataset, the other one is for testing. The training dataset has half of the benign apps and half of the malicious apps. For this case, we also split our dataset consisting 4,300 benign apps and 4,300 malicious apps randomly into two parts evenly, one part for training, other part for testing. In the second scenario, REVEALDROID has 7,995 malicious apps and 878 benign apps in the training set, and 1,188 obfuscated malicious apps and 869 benign apps for testing. Similar to their dataset, there are 4,300 benign apps and 4,300 malicious ones in our training set, and we form a testing set with 568 benign apps and 568 obfuscated malicious ones.

Note that all of our samples are chosen and split randomly. Compared with the imbalanced dataset from REVEALDROID, our dataset is very balanced. When training imbalanced data, which the number of malware is greater than benign apps, the classifier often favor the majority class and form a biased prediction model. The imbalance in the testing set will also cause notable inaccuracy.

Table 4 shows the comparison result without obfuscation. Table 5 presents the comparison result with obfuscated malware. Without obfuscation, as illustrated in Table 4, DREBIN shows the best performance with the average precision, recall and F-measure reaching 99%, we think this is because DREBIN gathers all types of features, such as permission, API call, intents and the diversity of the feature set plays a significant role to detect malware. OBFUSIFIER has the average F-measure of 96%, which is the same as RevealDroid. Even though the performance is not as good as DREBIN, both OBFUSIFIER and REVEALDROID outperform ADAGIO, of which average F-measure is 90%. MUDFLOW has the worst result, with only average 71% F-measure and 66% recall.

With obfuscation, as illustrated in Table 5, OBFUSIFIER outperforms all other four systems. This result is from feature combinations of F1 U F2 U F4. Note that F3 is a feature set extracted from the original method graph so the F3 feature set is not obfuscation resistant. It achieves surprisingly high metrics, with an average of 95% precision, recall, and F-measure. Note that the F-measure of

MUDFLOW with obfuscation is only 74%. This result is very close to the result without obfuscation (F-measure of 71%).

We suspect that this is because its feature sets are not influenced by the obfuscation techniques. DREBIN shows poor performance with obfuscation, the average precision, recall and F-measure are 0%. This is because all of DREBIN's feature sets are negatively influenced by obfuscation, and this result indicates that DREBIN is not resilient against obfuscation. ADAGIO achieves the average F-measure 62% with obfuscation, but this is not as good as its result (F-measure 90%) without obfuscation. Still it shows the ability to detect obfuscated malware. The average F-measure and recall of REVEALDROID is 85%, which is not as high as OBFUSIFIER.

	MUDFLOW (%)			RevealDroid (%)			Adagio (%)			Drebin (%)			Obfusifier (%)		
	Pr	Re	Fm	Pr	Re	Fm	Pr	Re	Fm	Pr	Re	Fm	Pr	Re	Fm
Ben	85	34	49	90	88	89	90	76	83	97	100	98	97	94	95
Mal	87	99	93	97	98	98	95	98	96	100	99	100	94	97	96
AVG	86	66	71	96	96	96	92	87	90	99	99	99	96	96	96

Table 4. Comparison Without Obfuscation (Pr = Precision, Re = Recall, and Fm = F-measure)

	MUDFLOW (%)			RevealDroid (%)			Adagio (%)			Drebin (%)			Obfusifier (%)		
	Pr	Re	Fm	Pr	Re	Fm	Pr	Re	Fm	Pr	Re	Fm	Pr	Re	Fm
Ben	98	47	64	91	72	80	54	73	62	42	100	59	97	92	95
Mal	72	99	84	82	95	88	73	54	62	0	0	0	93	98	95
AVG	88	73	74	86	85	85	63	63	62	18	42	25	95	95	95

Table 5. Comparison With Other Methods (Pr = Precision, Re = Recall, and Fm = F-measure)

5.5 Runtime Performance

For real-world applications, a malware detector must be both effective and efficient. To evaluate the efficiency of OBFUSIFIER, we measured the time taken to analyze and detect a malware sample. As part of analysis, one critical factor that can affect efficiency is the time to train the classification model and the time needed to test each app. The training time is the time to build the prediction model. The testing time is the average number to test each app. Another key factor is the time we spend to statically analyze apps and extract its features. We also list the average time needed to analyze each app in different phases: Graph Generation, Graph Simplification, SAP Generation and Feature Extraction. We measured the time of 100 apps (50 benign and 50 malicious apps, respectively)

and calculate the average execution time for each app in each phase. We found the system took the average total of 35.06 seconds to analyze each app, generate graphs, simplify paths, and extract features. This runtime result should be acceptable for detecting obfuscated and complex malware in real-world settings.

6 Discussion

Our evaluations have shown OBFUSIFIER's robustness, and its ability to handle obfuscated Android malware with high efficiency and accuracy. However, there are still some limitations of our system.

First, we only obfuscate malicious apps using ALAN. According to the results from VirusTotal, ALAN provides several very effective obfuscation techniques which help malware evade many existing anti-virus scanners. However, in order to verify OBFUSIFIER's ability to deal with different obfuscation techniques, we plan to experiment with more Android obfuscation tools, such as DashO [3], DexGuard [2] to transform malware codes.

Second, our system cannot handle the malware transformed by the obfuscation on the native code. Malware authors can take advantage of this loop-hole to encrypt the strings and arrays in the native code, and then decrypt them during runtime to hide the malicious behaviors. One notable tool that can close this loop-hole is OBFUSCATOR-LLVM [21], which targets the native code obfuscation. We plan to experiment with this tool and attempt to integrate it into our workflow.

Third, our system is based on static analysis of the DEX code, but if the DEX code is encrypted and then decrypted at runtime, we cannot capture its method graph and malicious behaviors. A special obfuscation technique called packing [18], which is used to protect Android apps being reverse engineered. It creates a wrapper application, and hide the original DEX code so that the original app cannot be reverse engineered. This wrapper app loads necessary libraries to unpack the original code at runtime. In future, we will consider the incorporation of dynamic analyzer in OBFUSIFIER.

7 Related Work

In this section, we describe works that are closely related to ours, including the four baseline systems used in Section 5 and other prior works about malware detection.

Garcia et al. [19] introduced REVEALDROID as a lightweight machine learning-based system to detect Android malware and identify Android malware families. It constructs features from the Android API usage, reflection characteristic and native binaries of the app. The evaluation shows that REVEALDROID can detect malware (both non-obfuscated and obfuscated malware) and identify malware family with high accuracy. MUDFLOW [12] is built on the static analysis tool FLOWDROID [10]. It extracts the normal data flow from benign apps as patterns, mines these benign patterns, and use these pattern to automatically

identify malicious behaviors. The novelty of their work is that they only use information from benign apps to train their system, and identify abnormal flows in malicious apps. Our evaluations indicate that MUDFLOW has some ability to detect obfuscated malware, with the precision of 88% and F-measure of 74%. ADAGIO [20] extracts the function call from Android apps and map these function calls to features, and build a machine learning system based on these features. As shown, the proposed system loses its accuracy when the malware samples are obfuscated.

DREBIN [9] is a machine learning-based malware detector, which performs broad static analysis on Android apps and collect many features such as permission, API calls, intents from app's code and the Manifest file and embedded them in a vector space that can be used to discover patterns of malware. These patterns are then used to build a machine learning detection system. The system is accurate but it requires running on a rooted device. As shown in our experiment, DREBIN is not able to detect obfuscated malware. APPCONTEXT [31] is another machine learning-based malware detector which focusing on the context difference between malware and benign apps. It leverages SOOT [29] as the static analysis engine and uses the permission mappings offered by PScout [11] to extracts the contexts based on Android components, Android permissions and Intent. They achieve 87.7% precision and 95% recall, which are lower than our system. The average analysis time of APPCONTEXT for each app is about 5 minutes [31], while we only need 35 seconds. This behavior based approach might be able to resist obfuscation, and we hope we can get its source code and assess its performance over obfuscated malware in future.

DROIDMINER [30] is a system that mines the program logic from Android malware, extracts this logic to modalities, which are ordered sequence of APIs, and constructs malicious patterns for malware detection. It builds a method call graph for each app, control flow graph, and generates modalities (API paths and subpaths) from sensitive methods. A feature vector based on the existence of modalities is formed for classification. They replace user-defined methods with framework API functions. We, on the other hand, remove the user-defined methods for efficiency. DROIDSIFT [33] is also a machine learning-based detector based on static analysis. They generate weighted contextual API dependency graphs, build graph databases, and construct a graph-based feature vector by performing graph similarity queries. Their features represent program behaviors at the semantic level. Note that the average detection time is about 176.8 seconds [33], while we only need 35 seconds.

8 Conclusion

We introduce OBFUSIFIER, a machine learning based malware detection system that is highly resistant to code obfuscation. The key insight is that obfuscation cannot be applied to portions of codes that include calls to Android APIs, kernel functions, and third party library APIs. Our system mainly extracts features based on these portions of codes unaffected by obfuscation. In total, we use four

feature sets consisting of 28 features. Our results showed that the effectiveness of the system is not affected by obfuscation. The system can achieve an average F-measure of 96% for detecting non-obfuscated malware. More importantly, the system can achieve an average F-measure of 95% in detecting obfuscated malware, suffering only a 1% drop in performance.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments and feedback. This work was supported in part by the NSF grants CNS-1566388 and CNS-1717898.

References

1. f-secure.com. https://www.f-secure.com/sw-desc/adware_android_dowgin.shtml.
2. guardsquare.com. https://www.guardsquare.com/en/products/dexguard.
3. preemptive.com. https://www.preemptive.com/products/dasho/overview.
4. Apkpure.com. https://apkpure.com/, December 2017.
5. Virusshare.com. https://virusshare.com/, December 2017.
6. Alan-android-malware.com. http://seclist.us/alan-android-malware-evaluating-tools-released.html, April 2019.
7. Virustotal.com. https://www.virustotal.com/#/home/upload, April 2019.
8. Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013.
9. D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.
10. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.
11. K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.
12. V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 426–436, 2015.
13. 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 *ICISSP*, pages 379–385, 2018.
14. J.-M. Borello and L. Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.
15. L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, Oct. 2001.
16. J. Cannell. Obfuscation: Malware's best friend. https://blog.malwarebytes.com/threat-analysis/2013/03/obfuscation-malwares-best-friend/, March 2016.
17. 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.

18. Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang. Things you may not know about android (un) packers: a systematic study based on whole-system emulation. In *25th Annual Network and Distributed System Security Symposium, NDSS*, pages 18–21, 2018.
19. J. Garcia, M. Hammad, and S. Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. *ACM Transactions of Software Engineering and Methodology*, 9(4), June 2017.
20. H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.
21. P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM – software protection for the masses. In B. Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.
22. N. Landwehr, M. Hall, and E. Frank. Logistic model trees. 95(1-2):161–205, 2005.
23. Z. Li, J. Sun, Q. Yan, W. Srisa-an, and S. Bachala. Grandroid: Graph-based detection of malicious network behaviors in android applications. In *International Conference on Security and Privacy in Communication Systems*, pages 264–280. Springer, 2018.
24. Z. Li, L. Sun, Q. Yan, W. Srisa-an, and Z. Chen. Droidclassifier: Efficient adaptive mining of application-layer header for classifying android malware. In *Proc. of Securecomm*, pages 597–616. Springer, 2016.
25. V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, 2014.
26. I. Steinwart and A. Christmann. *Support Vector Machines*. Springer Publishing Company, Incorporated, 1st edition, 2008.
27. L. Sun, Z. Li, Q. Yan, W. Srisa-an, and Y. Pan. Sigpid: significant permission identification for android malware detection. In *Proc. of MALWARE*, pages 1–8. IEEE, 2016.
28. Y. Tsutano, S. Bachala, W. Srisa-an, G. Rothermel, and J. Dinh. An efficient, robust, and scalable approach for analyzing interacting android apps. In *Proc. of ICSE*, Buenos Aires, Argentina, May 2017.
29. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.
30. C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. Technical report, Texas A&M, 2014.
31. W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 303–313. IEEE Press, 2015.
32. I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.
33. M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proc. of CCS*, pages 1105–1116, 2014.
34. Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of IEEE S&P*, pages 95–109, 2012.