Evaluating Android Malware Detection System Against Obfuscation and Stealth Techniques

Shikha Badhani[1] and Sunil K. Muttoo[2]

[1]Department of Computer Science,
Maitreyi College,
University of Delhi
shikhamalik2@gmail.com

[2]Department of Computer Science,
University of Delhi,
drskmuttoo@gmail.com

***Abstract*:**
Malicious apps targeting the Android OS have grown significantly due to their popularity. To stay ahead of malware creators, malware detection systems need to be evaluated against detection evading techniques. Malware authors now frequently use obfuscation and other stealth techniques, to construct malicious applications that can avoid being detected by malware detection systems. When confronted with the same malware samples that have been obfuscated, a highly successful detection system for identifying unobfuscated malware samples might lose its efficiency. Thus, it becomes important to analyze the malware detection systems against obfuscation and other stealth techniques. In this paper, we evaluate CENDroid – an Android malware detection system that uses static analysis and combines clustering and ensemble methods to develop a classifier, against various techniques of evading malware at the feature-level as well as classifier-level. Experimental results show that the features used in CENDroid - API tags and permissions, displayed strong robustness against code obfuscation and app hiding techniques. A comparison of obfuscation resilience of API tags and permissions features with code graphs and Androguard is also presented. At the classifier-level, CENDroid could detect all the malware in four test sets created by using obfuscation and app hiding techniques. Thus, CENDroid, being syntax-based, is immune to semantic-level changes that preserve the syntax of the app. This study can also be used as a framework to evaluate any Android malware detection system against obfuscation and stealth techniques, gaining insights into its strengths and weaknesses and informing potential improvements to enhance its effectiveness.

Keywords: Obfuscation Resilience, Android Malware Detection, API Tags, Permissions, Static Analysis.

(Article history: Received: June 20,2023 and accepted Aug.16, 2023)

I. INTRODUCTION

Android is not only the most popular mobile OS but also leads in the market share of all the operating systems worldwide. According to StatCounter [1], nearly 40% of the operating system market share belongs to Android followed by Windows and then iOS, etc. in the period from Jan 2021 to Jan 2022. Due to its popularity, it has also become an attractive target for malware authors to explore. Detecting these malwares on time is of utmost importance as they spread like wildfire. Recently Kaspersky [2] researchers discovered a malware named “Fleckpe” on the official Google Play Store that has infected over 620,000 Android devices. Malware creators always look for vulnerabilities in the new techniques developed so that they can create novel malware that can evade the detection techniques developed. Thus, it becomes important to evaluate the malware detection systems against the techniques that try to evade detection to present a complete picture of their capability to detect malware. Various Android malware detection systems have been proposed in the past that can detect malware with high accuracy [3]–[7]. In the literature, we can find various studies that evaluate Android malware detection systems against various techniques such as code obfuscation, steganography, cryptography, etc.[4], [8], [9].

Code obfuscation techniques make the code difficult to understand, and thus used to protect intellectual property but it is also used by malware creators to avoid detection as detection systems usually check for similarity patterns [10].

Most malware developers use various code obfuscations to conceal the harmful code inside the original Android apps to avoid detection [11]. To find malware, the majority of anti-malware tools rely on signature-based techniques [11]. Using various obfuscation methods, a signature can be easily evaded [12], [13]. Thus, testing Android malware detection systems against code obfuscations becomes an essential evaluation step to prove robust resilience against obfuscated malwares. In the literature, we found that many Android malware detection systems developed have not been evaluated against code obfuscations [14]–[16]. Merely reporting performance metrics on datasets of malicious and benign Android apps does not provide a complete analysis of the detection system that might fail to detect the obfuscated versions of malwares. In this paper, we extend and evaluate the existing Android malware detection system – CENDroid [3] – a Cluster-ENsemble classifier for detecting malicious Android apps, against code obfuscation and app hiding techniques. CENDroid is a static Android malware detection system that uses permissions and API (Application Programming Interface) tags as features along with cluster-ensemble classifier (k-mode clustering combined with ensemble learning – majority vote, weighted majority vote, and stacking) to detect whether an Android app is malicious or benign with high accuracy. For creating obfuscated malware, three tools namely ADAM [17], AAMO [11], and DroidChameleon [12] have been used and for creating malwares using app hiding techniques we have used the methods as described in [13]. The resilience of CENDroid against obfuscated malware has been performed at the feature-level as well as at the classifier-level to provide insights about how resilient the features and the classification algorithm used in CENDroid, are against code obfuscations.

The main contributions of our study are as follows:

- Analyzing the resilience of CENDroid against obfuscation and app hiding techniques at the feature-level as well as at the classifier-level. At the feature-level, the resilience of features extracted in CENDroid (API tags and permissions) is evaluated against various obfuscation and app hiding techniques. The methodology used in this study for feature-level analysis is based on the approach presented in [8]. At the classifier-level, we train CENDroid using a training set that comprises various benign and malicious Android apps and then observe their prediction accuracy on a special test set comprising obfuscated and hidden malware apps.
- A comparison of API tags and permissions feature sets with the feature-level analysis of code graphs and Androguard [8] is also presented.

The novelty of this work lies in presenting a framework comprising of feature-level as well as classifier-level analysis for evaluating malware detection systems. This study can also be used as a framework to evaluate any Android malware detection system against obfuscation and stealth techniques, gaining insights into its strengths and weaknesses and informing potential improvements to enhance its effectiveness.

The remainder of this paper is organized as follows. Related work and preliminaries are presented in Sect.II. In Sect.III, we present our methodology of feature- and classifier-level evaluation of CENDroid. The experimental results are presented in Sect. IV along with a discussion. Finally, we conclude in Sect.V.

II. Related work and preliminaries

In this section, we describe works that are closely related to ours, including the code obfuscation tools and app hiding techniques used in our study.

Android malware detection systems developed are based on static, dynamic, or hybrid approaches. The static approach involves extracting features from Android apps by analyzing the code of the app and the dynamic approach involves extracting features by executing the Android apps in an isolated environment. The static approach has the advantage of over dynamic analysis as the overhead of executing every app for extracting features can be computationally more expensive. However static analysis is unable to detect malware that has no trace of malware in the code of the app but performs malicious activity during runtime. The hybrid approach involves both – static as well as dynamic, thus coupling the advantages of static and dynamic approaches.

In [18], Control-Flow Graphs (CFG) are constructed for Android apps to obtain API datasets, and then ensemble learning is performed for Android malware detection. In [19], code graphs are extracted from Android apps and then machine learning is applied for classifying Android apps as benign or malicious. Code graphs are high-level control-flow graphs that contain package-level information in the nodes of the graphs. RevealDroid [9] extracts static features such as usage of APIs, reflection-based features, and native binaries of apps, and uses machine learning to perform malware detection and family identification. To assess RevealDroid against obfuscation, DroidChameleon [12] is used. DroidEvolver [20] uses static analysis to extract Android APIs to detect Android malware over time. DroidEvolver constructs a model pool with a set of online machine learning models. DroidEvolver could detect 96% of the obfuscated malware created by using DroidChameleon. DL-Droid [6] uses deep learning with a state-based input generation approach to detect malicious Android applications. It is an automated dynamic analysis system. DroidClone [21] exposes code clones (segments of code that are similar) in Android applications to help detect malware. The resistance of DroidClone was also tested by using the DroidChameleon tool to generate obfuscated apps. KTSODroid [22] extracts features from Android apps that are independent of most obfuscation methods and reflect the behavior of the application. The kernel task structure of the process running in memory is used to create the profile for an application which is further used in the classification task. Further, they also evaluate the effectiveness of proposed features against four obfuscation techniques; Class encryption, Reflection, String Encryption, and Junk Code insertion. MGOPDroid [23] is an efficient anti-obfuscation Android malware detection system that extracts opcodes features and uses a novel feature weight calculation method that combines TFIDF and the variation in opcode features before and after application obfuscation to select effective anti-obfuscation features of Android malware. To create the obfuscated dataset from benign and malicious apps, they use the AVPASS tool [24]. A highly effective malware detection system based on machine learning called OBFUSIFIER [25] can maintain its efficacy even when malware samples are obfuscated by extracting features based on these portions of codes that are not affected by obfuscation. SeqDroid [26] and DANdroid [27] use deep learning to detect obfuscated Android malware.

In [8], graph-based features – code graphs [19]are evaluated against code obfuscation and app hiding techniques [13], and a comparison is also presented between the similarity computation of code graphs and Androguard [28], [29]. Code graphs [19] are directed graphs that represent the semantic properties of Android apps. The code graph is formed when the decompiled code of the Android app is traversed and an API is encountered. Each node of the code graph contains the package to which an API belongs. Androguard is an open-source Python tool [29] used to extract static features from Android apps. Androguard uses the similarity distance computation based on Normalized Compression Distance (NCD) [28]. The dataset used [8] consists of malware from 33 malware families. To create obfuscated malware, three obfuscation tools (ADAM [17], AAMO [11], and DroidChameleon [12]) were employed, and taxonomy was defined for the obfuscations [8]. A fast-to-compute similarity metric [30] was used to compare the graph-based features of malicious apps with the corresponding obfuscated malicious apps.

Following are the commonly used obfuscation techniques as discussed in ADAM, AAMO, and DroidChameleon, mentioned in the preceding para:

- Repackaging - Android apps are packaged as compressed archive files. Un-compressing the APK files, adding empty resources, and re-compressing the file create a functionally identical APK with a different hash.
- Reassembly - Disassembling and assembling the DEX code (i.e., classes.dex) contained in an APK archive has the result of creating a functionally equivalent app yet with a different digest.
- Re-aligning - For efficient memory mapping, APK files are often aligned to 4-byte boundaries. This produces a functionally identical yet slightly different APK.
- Re-signing - Every APK must be digitally signed before it can be published and run on a device. An APK can be re-signed multiple times with different certificates and private keys. A different signature is generated and attached to APK.
- Junk code insertion/insert dead code- This technique includes inserting junk code such as no-operation (NOP) instructions, unused random methods, and classes. This results in a change of size or digest of the APK.
- Debug code deletion/debug code stripping - Debug statements such as .line, .source, .param, etc. are often used for debugging. Removing them would not change the functionality of the app but would impact the hash.
- Unconditional jump insertion – Forward and backward unconditional jumps are added in the code that preserves the semantics of the code.
- Call indirection - Methods calls are redirected to proxy methods that share the same prototype as the original method and call the original methods.
- Code reordering - Instructions are grouped into labeled blocks. These blocks are then rearranged and unconditional jumps are inserted to preserve the order of these blocks just like in the original code.

- Reflection - Static method calls are converted into reflection calls which are resolved at runtime.
- Opaque predicate insertion - An opaque predicate is a conditional expression whose value is known to the obfuscator a priori but still needs to be evaluated at run time and thus is unknown to static analyzers.
- Renaming - This obfuscator renames fields, methods, class names, or even resources. The corresponding references are also updated to keep the code error-free.
- Encryption – Objects such as strings, native codes, or resources can also be encrypted. The decryption is performed every time an object is accessed during runtime.

In the study presented here, we have extended the methodology, taxonomy of obfuscations, and similarity metric used in [8] for feature-level analysis of CENDroid. We observed that not only at the feature-level, analysis of obfuscated malware detection should also be done at the classifier-level since the model employed also has a big role in classification apart from the properties of features alone. Thus, in this study, we have also performed the classifier-level analysis of CENDroid along with a feature-level analysis.

Various techniques have also been developed to hide Android apps in benign apps to evade detection. In [13], malwares were created by hiding malicious app inside benign app using techniques such as concatenation, steganography, cryptography, and/or obfuscation. The malwares created could easily evade detection by popular anti-malware tools. Thus, in our work we have evaluated the capability of CENDroid at the feature and classifier-level, to detect such stealth malwares.

III. METHODOLOGY

In this section, we present the methodology of evaluating CENDroid against code obfuscation and app hiding techniques. This methodology is generic and can be applied to any other malware detection system as well.

Research question

In this paper, we propose the solution to the following:

- How resilient are the features extracted in CENDroid towards code obfuscation and app-hiding techniques?
- How resilient is the classifier model of CENDroid towards code obfuscation and app-hiding techniques?

The methodology for evaluating CENDroid at the feature-level as well as classifier-level is shown in Fig. 1. CENDroid extracts API tags and permissions from Android apps and then uses a combination of cluster-ensemble learning to classify apps as benign or malicious [3]. In the cluster-ensemble approach, clustering (k-mode) is performed on the training set to divide it into an optimal number of clusters, and then ensemble learning is performed on each cluster [3]. It has been demonstrated that the method of using numerous clusters within classified data and learning the boundaries inside these clusters by using a series of base classifiers achieves diversified learning [31]. In the case of CENDroid also, the cluster-ensemble approach has shown improvement in performance metrics over the ensemble approach [3]. To analyze CENDroid, this study extracts the API tags and permissions features from the dataset of Android apps and performs feature-level analysis to evaluate the resilience of these features against code obfuscations and app hiding techniques. Another evaluation is performed at the classifier-level to analyze how resilient the cluster-ensemble classification algorithm proposed in CENDroid is against code obfuscations and app hiding techniques. For obfuscations, three tools – ADAM, DroidChameleon, and AAMO are used.

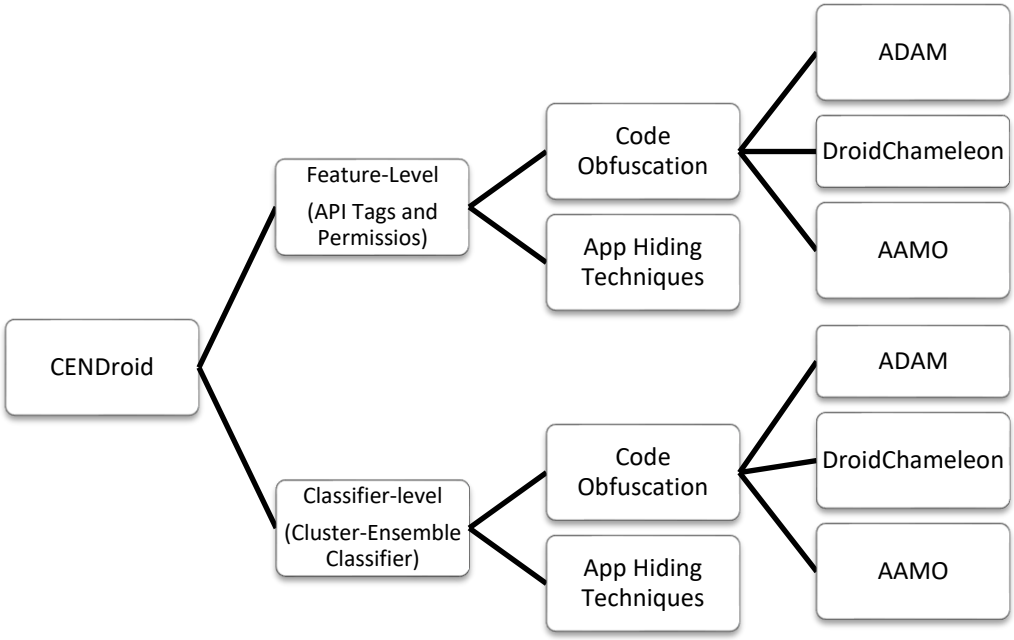


Fig. 1 Methodology for evaluating CENDroid

Feature-level analysis

At the feature-level, API tags and permissions are extracted from the original malware and its obfuscated apps. For creating obfuscated apps of malware, ADAM, DroidChameleon, and AAMO obfuscating tools are used, and for creating apps with hidden malwares, the techniques proposed in [13] namely concatenation, obfuscation, cryptography, and/or steganography have been employed. Single-level obfuscations - Level A (obfuscations that do not change the source code), Level B (obfuscations that change the source code), Level C (obfuscations that perform renaming), and Level D (obfuscations that perform encryption) as well as multiple obfuscations (Level AB, AC, AD, ABC, and ABCD) were performed as done in [8]. The features of the original malware app and its obfuscated versions are then compared using a similarity metric. A similar experiment is performed for app hiding techniques in which malwares were created by using a cover benign app in which an image in which the malicious app code is hidden, was stored. Using reflection and dynamic code loading (using DexClassLoader), the malicious app code was then extracted from the images and loaded dynamically to perform malicious behavior [13]. Since, in these hiding techniques, a benign app is converted into malware, a comparison of the malware created is made with the cover benign app instead of the original malicious app to analyze whether API tags and permission features can detect these hidden malwares or not. Thus, API tags and permissions are extracted from the cover benign app and also from the malwares created by hiding malicious code in images and hiding them in the cover benign app using various techniques, and then a similarity metric is used to analyze the difference.

The similarity metric used in [8] for comparing graph-based features can even be used for the API tags and permissions features. The API tags and permissions features are binary vectors. Hence, the same similarity metric can also be used for them. The interpretation is as follows.

Similarity metric between two binary vectors A and B computed as [30]:

$$\boldsymbol{sim}(\boldsymbol{A},\boldsymbol{B}) = \frac{\mathbf{2} \times |\boldsymbol{A} \cap \boldsymbol{B}|}{|\boldsymbol{A}| + |\boldsymbol{B}|} \quad (\mathbf{1})$$

The similarity metric can take values between 0 and 1. The time complexity of this similarity computation is $O(c)$, where c is a constant, as the features are binary vectors of fixed length, and the computation involves only intersection, addition, multiplication, and division operations. It takes the value 1 when the two binary vectors are similar and 0 when there is no similarity. We have used the same threshold for similarity calculation i.e., 0.8, as used in the case of graph-based features [8]. Using Eq. 1, a similarity score ≥ 0.8 implies that an app is highly similar to another app. For obfuscation, a high similarity score is required between malware and its obfuscated version to prove resilience towards obfuscation. However, in the case of app hiding techniques, a low similarity score is required between the benign app and its malicious version created by hiding inside an image, to prove that the features reflect the introduction of app hiding techniques. Further, we also present the comparison of API tags and permissions features evaluation against obfuscation and app hiding techniques with that of code graphs [19] and Androguard [29] which has been reported in [8].

Classifier-level analysis

At the classifier-level, CENDroid is first trained with a dataset consisting of the original malwares [8]. Then four test sets comprising of obfuscated malware created using the three tools mentioned above [8], and malware created via app hiding techniques described in [13] are used, namely:

Test set 1 – This test set comprises all the obfuscated apps generated by the ADAM tool.

Test set 2 – This test set comprises all the obfuscated apps generated by the DroidChameleon tool.

Test set 3 – This test set comprises all the obfuscated apps generated by the AAMO tool.

Test set 4 – This test set comprises all the obfuscated apps generated by the app hiding techniques.

Then predictions are made on these test sets by CENDroid to evaluate how accurately these malicious test sets can be detected by CENDroid.

IV. Experimental Results and Discussion

In this section, we present the results obtained on evaluating the resilience of CENDroid against obfuscation and app hiding techniques. We have used the dataset of 33 malware families as used in [8] for performing code obfuscation. For evaluation against app hiding techniques, the dataset of 16 malwares (8 for malwares created using PNG image + 8 for malwares created using JPEG image) developed in [13] was used.

Feature-level resilience

We perform a comparison of malware apps and their obfuscated versions created by using the three obfuscators and also benign app and its malicious versions created by using app hiding techniques.

Resilience of features against the ADAM tool

The dataset of 330 obfuscated malicious apps (10 obfuscations performed for each of the 33 malware families using the ADAM tool) [8] was used. ADAM tool can perform level A, B, C, D, AB, ABC, and ABCD obfuscations. API tags and permissions were extracted from the original APK and the obfuscated version and compared. The API tags and permissions features displayed high immunity with a perfect similarity score of 1 against all the obfuscations (single as well as multiple) as performed by the ADAM tool. The reason for such high similarity scores is attributed to the level of abstraction these features offer. Being syntax-related, they are not dependent on the semantics of the code. Obfuscations such as inserting defunct code, changing the CFG, encrypting strings, etc. have no effect on the usage of API tags and permissions used in an app since obfuscations retain the functionality an app serves. Whatever API calls and permissions are present in an app, the same shall also be present in its obfuscated version. As compared to the average similarity scores of code graphs and Androguard, API tags, and permissions displayed better resilience to obfuscations performed using the ADAM tool as shown in Table I. As compared to code graphs and Androguard, API tags, and permissions feature sets performed extremely well especially when multiple obfuscations are applied. The resilience of code graphs and Androguard drops for obfuscations involving level C (renaming) and level D (encryption).

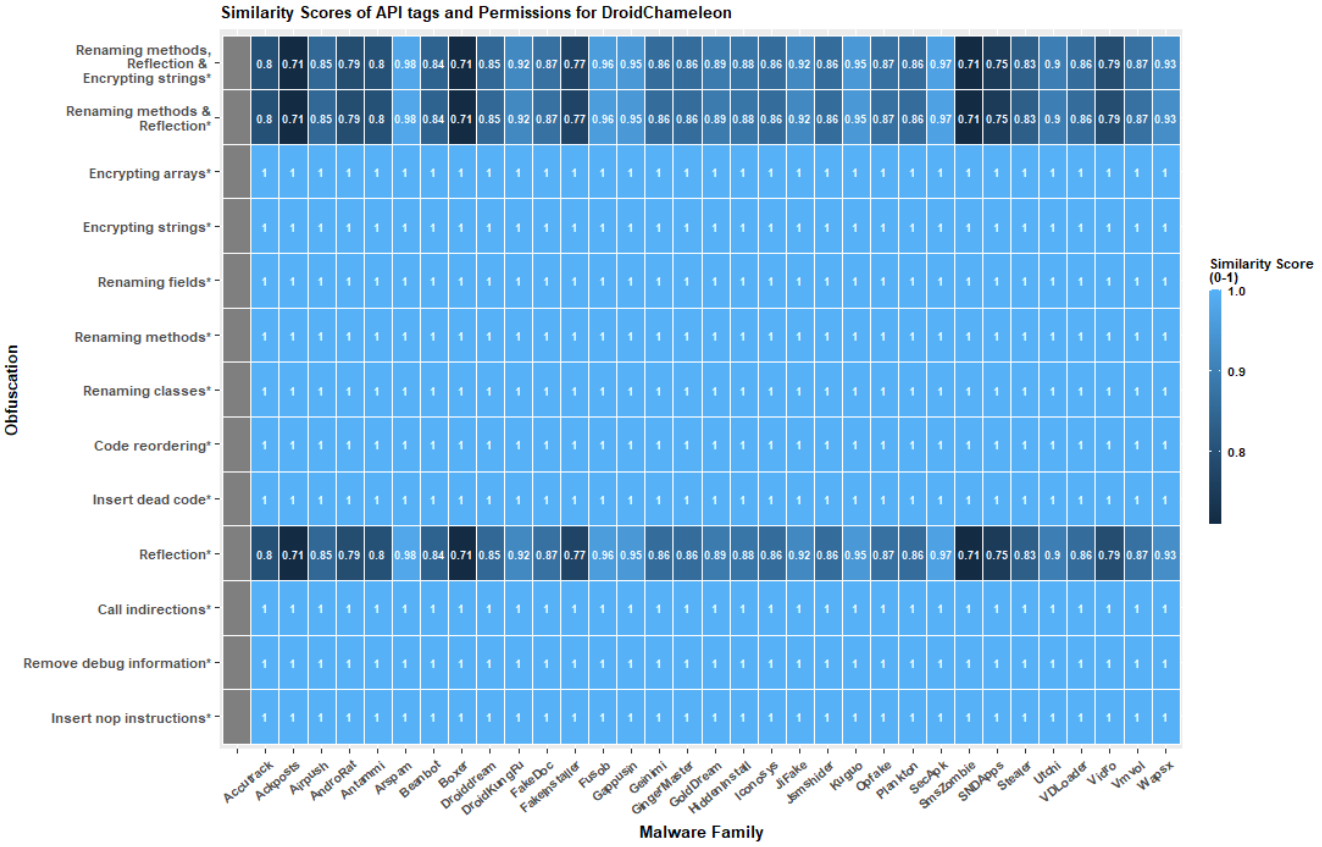
Resilience of features against the DroidChameleon tool

In this experiment, a dataset of 429 obfuscated malicious apps (for each of the 33 malware families, 13 obfuscations performed by the DroidChameleon tool) [8] was used. DroidChameleon tool can perform level AB, AC, AD, ABC, and ABCD obfuscations. In Fig 2, the heatmap representation of similarity scores for obfuscations performed using the DroidChameleon tool are displayed for each malware family. In the case of DroidChameleon, the API tags and permissions feature set again scored 1 for the similarity between the original app and all the single level obfuscated apps for a similar reason as given for ADAM. For multiple obfuscations, the average similarity scores were above the threshold but not 1. For obfuscations that involved reflection, the API tags and permissions feature set did get reduced a little but the impact was still above the threshold. As compared to code graphs and Androguard, API tags, and permissions feature set again outperformed the obfuscations performed using the DroidChameleon tool as shown in Table II. Code graphs also got impacted due to reflection obfuscation and the impact was large enough to cause the value to drop below the threshold. Androguard got impacted a lot not only by reflection but many other obfuscations. The average similarity scores of code graphs and Androguard were below the threshold for many obfuscations.

TABLE I. COMPARISON OF AVERAGE SIMILARITY SCORES FOR ADAM OBFUSCATIONS

Level	**A (Obfuscations that do not change the source code)**			**B (Obfuscations that change the source code)**	**C (Obfuscations that perform renaming)**		**D (Obfuscations that perform encryption)**	**AB**	**ABC**	**ABCD**
Average Similarity Scores	***Rebuild***	***Re-sign***	***Alignment***	***Insert dead code***	***Renaming***	***Modify CFG***	***Encryption***	***Rebuild + Insert dead code***	***Rebuild + Insert dead code + Modify CFG***	***Rebuild + Insert dead code + Modify CFG + Encryption***
AP	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
CG [8]	1.00	1.00	1.00	1.00	0.98	0.94	0.86	1.00	0.94	0.82
AG [8]	1.00	1.00	1.00	0.99	1.00	0.80	0.84	0.99	0.80	0.70

AP – API tags and Permissions, CG – Code Graphs, AG – Androguard



* Includes level A obfuscations – Re-sign and Rebuild

Fig 2. Similarity scores of API tags and permissions for DroidChameleon obfuscations

TABLE II.COMPARISON OF AVERAGE SIMILARITY SCORES FOR DROIDCHAMELEON OBFUSCATIONS

Level	**AB (Obfuscations that do not change the source code and those that change the source code)**						**AC (Obfuscations that do not change the source code and those that perform renaming)**			**AD (Obfuscations that do not change the source code and those that perform encryption)**		**ABC**	**ABCD**
Average Similarity Scores	***Insert nop instruction****	***Remove debug statements****	***Call indirections****	***Reflection****	***Insert dead code****	***Code reordering****	***Renaming classes****	***Renaming methods****	***Renaming fields****	***Encrypting strings****	***Encrypting arrays****	***Renaming methods & Reflection****	***Renaming methods, Reflection & Encrypting strings****
AP	1.00	1.00	1.00	0.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.85	0.85
CG [8]	1.00	1.00	0.94	0.62	0.86	0.78	0.98	0.96	1.00	0.95	1.00	0.61	0.60
AG [8]	0.71	1.00	0.96	0.44	0.16	0.31	1.00	0.99	1.00	0.82	1.00	0.44	0.38

* Includes level A obfuscations – Re-sign and Rebuild
AP – API tags and Permissions, CG – Code Graphs, AG – Androguard

Resilience of features against the AAMO tool

In this experiment, a dataset of 726 obfuscated malicious apps (for each of the 33 malware families, 22 obfuscations performed by the AAMO tool) [8] was used. AAMO tool can perform Level A, B, C, D, AB, ABC, and ABCD obfuscations. Fig. 3 presents the AAMO results in a heatmap representation of similarity scores for obfuscations performed using the AAMO tool for each malware family. The API tags and permissions feature set confronted all the types of obfuscations performed using the AAMO tool. It also performed much better than the code graphs and Androguard as shown in Table III. The minimum average similarity score of API tags and permissions was 0.99 while that of code graphs and Androguard was 0.64 and 0.18 respectively for the obfuscations performed using the AAMO tool.

Thus, for all three obfuscation tools, API tags and permissions feature set was highly robust in being resilient to changes due to obfuscations. The experimental results show that API tags and permissions feature sets are not affected by single-level obfuscations. However, they do get affected slightly when multiple obfuscations are applied. For all three obfuscation tools, all the average similarity scores of the API tags and permissions feature sets were > 0.85.

API tags contain the package-level information of the sensitive APIs that are being used in an app (Badhani&Muttoo, 2019c). Since the idea behind obfuscation is to make the code difficult to understand yet retain the behavior of the app that's why the sensitive APIs also remain the same in the obfuscated apps. This generalization of API tags gives them an edge to confront obfuscation strongly. The permissions requested by an app also remain the same even after obfuscation to maintain the functionality. Thus, even permissions are not much affected by obfuscations.

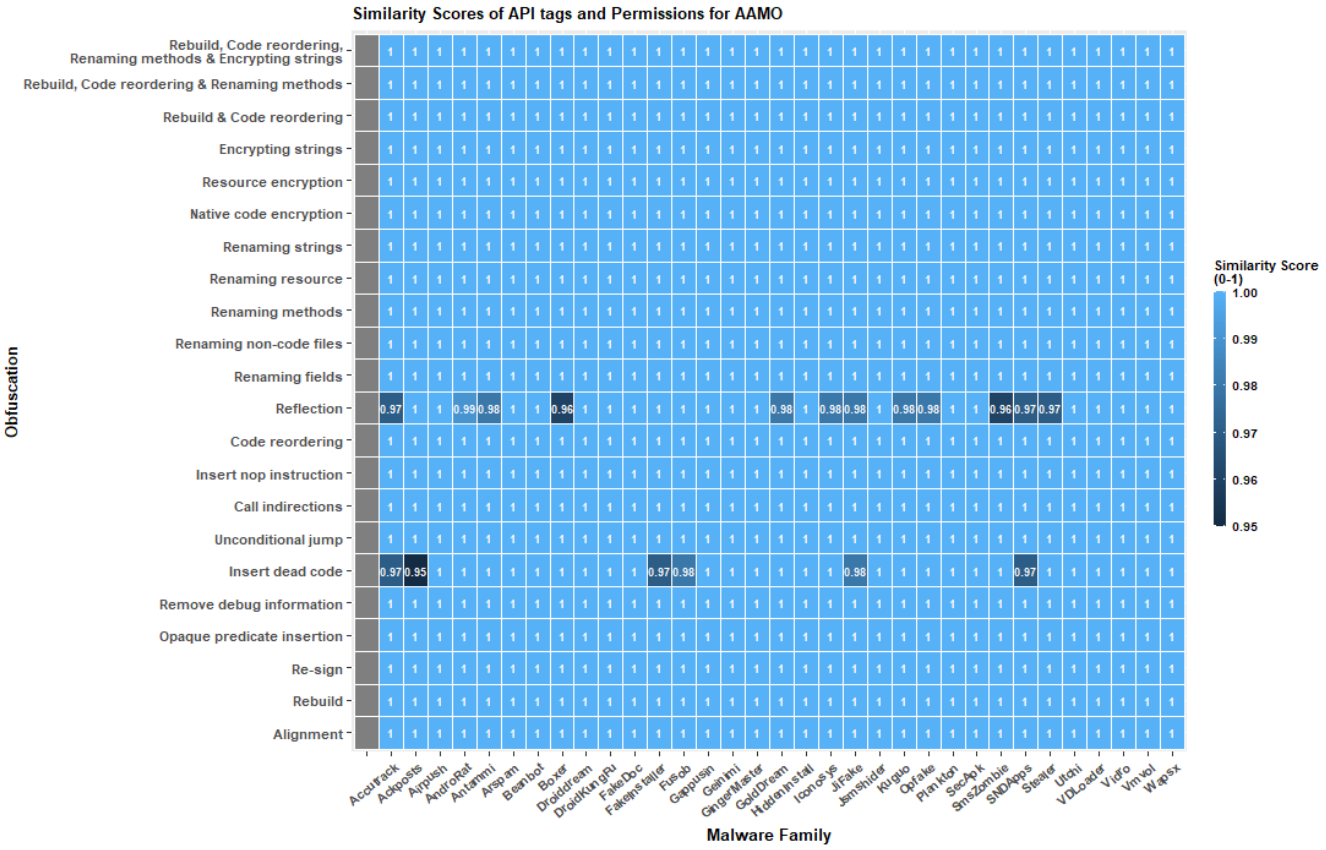


Fig. 3 Similarity scores of API tags and permissions for AAMO obfuscations

TABLE III COMPARISON OF AVERAGE SIMILARITY SCORES FOR AAMO OBFUSCATIONS

Level	**A (Obfuscations that do not change the source code)**			**B (Obfuscations that change the source code)**								**C (Obfuscations that perform renaming)**					**D (Obfuscations that perform encryption)**			**AB**	**ABC**	**ABCD**
Average Similarity Scores	***Alignment***	***Rebuild***	***Re-sign***	***Opaque predicate insertion***	***Remove debug information***	***Insert dead code***	***Unconditional jump***	***Call indirections***	***Insert nop instruction***	***Code reordering***	***Reflection***	***Renaming fields***	***Renaming non-code files***	***Renaming methods***	***Renaming resource***	***Renaming strings***	***Native code encryption***	***Resource encryption***	***Encrypting strings***	***Rebuild & Code reordering***	***Rebuild, Code reordering & Renaming methods***	***Rebuild, Code reordering, Renaming methods & Encrypting strings***
AP	1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.0
CG [8]	1.00	1.00	1.00	0.97	1.00	0.90	0.90	0.86	1.00	0.77	0.85	1.00	0.99	0.97	1.00	1.00	1.00	0.95	0.84	0.77	0.75	0.64
AG[8]	1.00	1.00	1.00	0.74	1.00	0.87	0.66	0.33	0.62	0.20	0.29	1.00	0.94	1.00	0.94	0.94	1.00	0.94	0.81	0.20	0.20	0.18

AP – API tags and Permissions, CG – Code Graphs, AG – Androguard

Resilience of features against app hiding techniques

Table IV shows the similarity scores of the API tags and permissions feature set of the benign app with its malware variants created in [13] by hiding a malicious app inside an image using 8 techniques. Both PNG and JPEG images were used to hide the malicious app.

Features extracted in CENDroid (API tags and permissions) also confront the app hiding techniques with great strength with all the similarity values < 0.8. This signifies that there is less similarity between the benign app and its malicious versions since various API packages and permissions have been added in building these malicious versions which have been reflected in these features also. The threshold value was below 0.8 even for code graphs and Androguard.

Classifier-level resilience

For training CENDroid, we have used those datasets which contain the original samples of the malicious family that were obfuscated using the obfuscation tools [8]. The test results obtained on the four test sets as described in created are shown in Table V for CENDroid. All the cluster-ensemble methods of CENDroid (majority vote, weighted majority vote, and stacking) were able to detect all the obfuscated apps (100%) for all four test sets as shown in Table V. Thus, such high detection rates prove that not only at feature-level but also at classifier-level, CENDroid has displayed high resilience against obfuscation as well as app hiding techniques. The reason can be attributed to the strength of API tags and permissions feature set against obfuscation and app hiding techniques coupled with the highly accurate classifier of CENDroid. Thus, we propose that the resilience of features against obfuscations and other malware evading techniques must be evaluated as the resilience is propagated to the classifier as well thus resulting in accurate malware detection systems.

TABLE IV COMPARISON OF SIMILARITY SCORES OF APP HIDING TECHNIQUES

Features	**Image**	**Technique for hiding malicious app inside an image**							
		Concatenate	***Concatenate &Obfuscate***	***Concatenate & Cryptography***	***Concatenate, Cryptography, & Obfuscate***	***Steganography***	***Obfuscate & Steganography***	***Cryptography & Steganography***	***Cryptography, Steganography, &Obfuscate***
AP	PNG	0.69	0.69	0.69	0.69	0.67	0.67	0.67	0.67
	JPEG	0.64	0.64	0.64	0.64	0.64	0.64	0.64	0.64
CG [8]	PNG	0.69	0.69	0.69	0.69	0.62	0.62	0.62	0.62
	JPEG	0.53	0.53	0.53	0.53	0.62	0.62	0.62	0.62
AG [8]	PNG	0.70	0.70	0.70	0.70	0.58	0.58	0.58	0.58
	JPEG	0.56	0.56	0.56	0.56	0.17	0.17	0.17	0.17

AP – API tags and Permissions, CG – Code Graphs, AG – Androguard

TABLE V PREDICTION OF CENDROID

Test Set	**Number of malicious apps**	**Number of malicious apps detected by the Cluster-ENsemble Classifier (CENDroid)**		
		Majority Voting	***Weighted Majority Voting***	***Stacking***
Test Set 1 (ADAM)	330	330	330	330
Test Set 2 (DroidChameleon)	429	429	429	429
Test Set 3 (AAMO)	726	726	726	726
Test Set 4 (App Hiding Techniques)	16	16	16	16

Thus, in this study, we were able to evaluate the resilience of CENDroid against code obfuscations and app hiding techniques and a comparison with other approaches (Androguard and Code graphs) is also presented. The framework for evaluating Android malware detection systems presented in this study has the advantage of providing a holistic view of how the features behave and how the performance metrics of the classification algorithm change when presented with obfuscations and app hiding techniques. The approach mentioned in [8] is limited to feature-level analysis. Thus, this research can help in identifying areas (at the feature-level or classifier-level or both) where more focus is required to improvise the Android malware detection systems against obfuscations and app hiding techniques.

V. CONCLUSIONS

In this paper, an Android malware detection system – CENDroid is evaluated against code obfuscation and app hiding techniques at the feature-level as well as at the classifier-level. At the feature-level, the features used in CENDroid (API tags and permissions) are extracted from the malware and their obfuscated variants. Experimental results show that API tags and permissions feature sets confronted all the single as well as multiple obfuscations by reporting a high degree of similarity between the original malware and its obfuscated variants. Low similarity scores between a benign app and its malicious variants were reported by the permissions and API tags feature set for the app hiding techniques and this was desirable as well. As compared to code graphs and Androguard, API tags, and permissions feature sets performed better, especially for multiple obfuscations for all three obfuscation tools. At the classifier-level, three test sets of obfuscated apps (created by using ADAM, DroidChameleon, and AAMO) and a fourth test set of stealth malwares (created by using app hiding techniques) were used to test the models trained by CENDroid. CENDroid could detect all the malware in each of the four test sets. Thus, CENDroid, being syntax-based, is immune to semantic-level changes that preserve the syntax of the app. Robust obfuscation resilience of features propagates to the classifier as well and results in highly accurate malware detection systems.

ACKNOWLEDGMENT

This paper is derived in part from the Ph.D. thesis of the first author who received her doctorate from the Department of Computer Science, University of Delhi.

REFERENCES

[1] Statcounter, "Operating System Market Share Worldwide Jan 2021 - Jan 2022," 2022. [Online]. Available: https://gs.statcounter.com/os-market-share

[2] D. Kalinin, "Not quite an Easter egg: a new family of Trojan subscribers on Google Play," May 2023. Accessed: May 09, 2023. [Online]. Available: https://securelist.com/fleckpe-a-new-family-of-trojan-subscribers-on-google-play/109643/

[3] S. Badhani and S. K. Muttoo, "CENDroid—A cluster-ensemble classifier for detecting malicious Android applications," *ComputSecur*, vol. 85, pp. 25–40, 2019, doi: 10.1016/j.cose.2019.04.004.

[4] H. Cai, N. Meng, B. Ryder, and D. Yao, "DroidCat: Effective android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2019, doi: 10.1109/TIFS.2018.2879302.

[5] H. Gao, S. Cheng, and W. Zhang, "GDroid: Android malware detection and classification with graph convolutional network," *ComputSecur*, vol. 106, 2021, doi: 10.1016/j.cose.2021.102264.

[6] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "DL-Droid: Deep learning based android malware detection using real devices," *ComputSecur*, vol. 89, 2020, doi: 10.1016/j.cose.2019.101663.

[7] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, "A Review of Android Malware Detection Approaches Based on Machine Learning," *IEEE Access*, vol. 8, pp. 124579–124607, 2020, doi: 10.1109/ACCESS.2020.3006143.

[8] S. Badhani and S. K. Muttoo, "Analyzing Android Code Graphs against Code Obfuscation and App Hiding Techniques," *Journal of Applied Security Research*, vol. 14, no. 4, 2019, doi: 10.1080/19361610.2019.1667165.

[9] J. Garcia, M. Hammad, and S. Malek, "Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 3, pp. 1–29, 2018, doi: 10.1145/3162625.

[10] A. Balakrishnan and C. Schulze, "Code Obfuscation Literature Survey," 2005. http://pages.cs.wisc.edu/~arinib/writeup.pdf (accessed May 16, 2019).

[11] M. D. Preda and F. Maggi, "Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 3, pp. 209–232, 2017, doi: 10.1007/s11416-016-0282-2.

[12] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, ACM, 2013, pp. 329–334. doi: 10.1145/2484313.2484355.

[13] S. Badhani and S. K. Muttoo, "Evading android anti-malware by hiding malicious application inside images," *International Journal of Systems Assurance Engineering and Management*, vol. 9, no. 2, pp. 482–493, 2018, doi: 10.1007/s13198-017-0692-7.

[14] F. Idrees, M. Rajarajan, M. Conti, T. M. Chen, and Y. Rahulamathavan, "PIndroid: A novel Android malware detection system using ensemble learning methods," *ComputSecur*, vol. 68, pp. 36–46, 2017, doi: 10.1016/j.cose.2017.03.011.

[15] S. B. Almin and M. Chatterjee, "A Novel Approach to Detect Android Malware," in *International Conference on Advanced Computing Technologies and Applications (ICACTA)*, 2015, pp. 407–417. doi: 10.1016/j.procs.2015.03.170.

[16] H. J. Zhu, Z. H. You, Z. X. Zhu, W. L. Shi, X. Chen, and L. Cheng, "DroidDet: Effective and robust detection of android malware using static analysis along with rotation forest model," *Neurocomputing*, pp. 638–46, 2017, doi: 10.1016/j.neucom.2017.07.030.

[17] M. Zheng, P. P. C. Lee, and J. C. S. Lui, "ADAM: An automatic and extensible platform to stress test android anti-virus systems," in *International conference on detection of intrusions and malware, and vulnerability assessment*, Springer, Berlin, Heidelberg, 2012, pp. 82–101. doi: 10.1007/978-3-642-37300-8-5.

[18] Z. Ma, H. Ge, Y. Liu, M. Zhao, and J. Ma, "A Combination Method for Android Malware Detection Based on Control Flow Graphs and Machine Learning Algorithms," *IEEE Access*, vol. 7, pp. 21235–21245, 2019, doi: 10.1109/ACCESS.2019.2896003.

[19] S. Badhani and S. K. Muttoo, "Android malware detection using code graphs," in *System Performance and Management Analytics*, Springer Singapore, 2019, pp. 203–215.

[20] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "DroidEvolver: Self-evolving android malware detection system," in *Proceedings - 4th IEEE European Symposium on Security and Privacy, EURO S and P 2019*, 2019, pp. 47–62. doi: 10.1109/EuroSP.2019.00014.

[21] S. Alam and I. Sogukpinar, "DroidClone: Attack of the android malware clones - a step towards stopping them?," *Computer Science and Information Systems*, vol. 18, no. 1, pp. 67–91, 2020, doi: 10.2298/CSIS200330035A.

[22] S. Khalid, F. B. Hussain, and M. Gohar, "Towards Obfuscation Resilient Feature Design for Android Malware Detection-KTSODroid," *Electronics (Basel)*, vol. 11, no. 24, p. 4079, Dec. 2022, doi: 10.3390/electronics11244079.

[23] J. Tang, R. Li, Y. Jiang, X. Gu, and Y. Li, "Android malware obfuscation variants detection method based on multi-granularity opcode features," *Future Generation Computer Systems*, vol. 129, pp. 141–151, Apr. 2022, doi: 10.1016/j.future.2021.11.005.

[24] C. Jeon, I. Yun, J. Jung, M. Wolotsky, and T. Kim, "AVPASS: automatically bypassing android malware detection system," in *Black Hat USA* , 2017.

[25] Z. Li, J. Sun, Q. Yan, W. Srisa-an, and Y. Tsutano, "Obfusifier: Obfuscation-Resistant Android Malware Detection System," in *Security and Privacy in Communication Networks: 15th EAI International Conference, SecureComm*, 2019, pp. 214–234. doi: 10.1007/978-3-030-37228-6_11.

[26] W. Y. Lee, J. Saxe, and R. Harang, "SeqDroid: Obfuscated Android Malware Detection Using Stacked Convolutional and Recurrent Neural Networks," 2019, pp. 197–210. doi: 10.1007/978-3-030-13057-2_9.

[27] S. Millar, N. McLaughlin, J. Martinez del Rincon, P. Miller, and Z. Zhao, "DANdroid," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, New York, NY, USA: ACM, Mar. 2020, pp. 353–364. doi: 10.1145/3374664.3375746.

[28] A. Desnos, "Android: Static analysis using similarity distance," in *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2011, pp. 5394–5403. doi: 10.1109/HICSS.2012.114.

[29] A. Desnos, "Androguard," 2011. https://github.com/androguard/androguard/releases/tag/v2.0 (accessed May 16, 2019).

[30] M. Xu *et al.*, "A similarity metric method of obfuscated malware using function-call graph," *Journal in Computer Virology*, vol. 9, no. 1, pp. 35–47, 2013, doi: 10.1007/s11416-012-0175-y.

[31] A. Rahman and B. Verma, "Cluster-based ensemble of classifiers," *Expert Syst*, vol. 30, no. 3, pp. 270–282, Jul. 2013, doi: 10.1111/j.1468-0394.2012.00637.x.