Hindawi
Security and Communication Networks
Volume 2023, Article ID 5393890, 11 pages
https://doi.org/10.1155/2023/5393890

Research Article

ImageDroid: Using Deep Learning to Efficiently Detect Android Malware and Automatically Mark Malicious Features

Pengfei Liu,[1] **Weiping Wang**,[1] **Shigeng Zhang**,[1,2] **and Hong Song**[1]

[1]*School of Computer Science and Engineering, Central South University, Changsha, Hunan, China*
[2]*The State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*

Correspondence should be addressed to Hong Song; songhong@csu.edu.cn

Received 1 July 2022; Accepted 13 September 2022; Published 7 April 2023

Academic Editor: Shudong Li

The popularity of the Android platform has led to an explosion in malware. The current research on Android malware mainly focuses on malware detection or malware family classification. These studies need to extract a large number of features, which consumes a lot of manpower and material resources. Moreover, some malware use obfuscation to evade decompiler tools extracting features. To address these problems, we propose ImageDroid, a method based on the image format of Android applications that can not only detect and classify malware without prior knowledge but also detect the obfuscated malware. Furthermore, we utilize the Grad-CAM interpretable mechanism of the deep learning model to automatically label the image that play a key role in determining maliciousness in a visual way. We evaluate ImageDroid over 10,000 Android applications. Experimental results show that the accuracy of malicious detection and multifamily classification achieve 97.2% and 95.1%, respectively, and the detection accuracy of obfuscated malware achieves 94.6%.

1. Introduction

With the rapid development and popularization of 5G networks, smartphones are recognized as an integral part of our lives, such as chatting, taking photos, electronic payment, and so on.

According to the Counter Point statistical report [1], the number of smartphones sold in 2021 has reached 1.35 billion. The Android system is the most widely used operating system within smartphones. The Android system is considered to be the biggest target of malware attacks, making the Android system very vulnerable to network attacks due to its openness.

When a malware variant owns evasion detection technology, even if the malware function of the new variant does not change, the malware cannot be detected. Therefore, to adapt to the variety of malicious applications, many researches on malware detection methods rely on feature extraction [2–5]. Feature extraction usually requires a lot of manual work and material resources, which relies heavily on prior knowledge. At present, there are two methods for malware detection: static method and dynamic method [6, 7]. In the static method, if the code is encrypted, the detection efficiency will be reduced or even the detection model will be invalid. Although the dynamic method can solve this problem, it must be configured with a specific running environment, which means higher requirements for hardware and detection time. Moreover, the dynamic method has the weakness of incomplete trigger path coverage [8]. If the malicious execution action is not triggered, the detection efficiency will be reduced. Given the rapid development of current deep learning models in image-based recognition [9–11], some deep learning models have achieved good results in windows malware detection [12].

In this paper, we propose ImageDroid, an Android malware classification method based on image, which directly classifies the maliciousness of Dex file without decompilation. Different from other methods, on the basis of analyzing the structure of the Dex file, we only retain the Data Area that plays an important role in the semantic logic

of the code, then convert it into image, and then apply the deep learning model (Inception-ResNet-v2) for classification. The experimental results verify the effectiveness of the method. After extracting the Data Area of the Dex, the classification performance is improved regardless of whether the target Dex file is obfuscated or not. On this basis, we use the interpretable mechanism of the deep learning model to mark the image part that plays a key role in determining the maliciousness.

The remainder of the paper is structured as follows. Section 2 gives the detailed design and implementation of ImageDroid. Section 3 evaluates the effectiveness of ImageDroid. We conclude the paper in Section 4. Section 5 reviews the related work.

2. ImageDroid Design and Implementation

As obfuscation and other mechanisms are increasingly used for code protection, it makes code reverse and static analysis more difficult, and many malicious Android applications also use this mechanism to evade malicious detection. The purpose of ImageDroid is to determine code maliciousness without decompilation. The main idea is to directly extract the part of the Dex file that represents the semantics of the code and then convert it into an image. Finally, the image is fed into a deep learning model for classification.

The specific implementation of ImageDroid is shown in Figure 1, which mainly includes the following two stages. (1) *Malicious Classification*. We convert the Data Area into image and put it into the Inception-ResNet-v2 for malicious classification. (2) *Marking Key Parts of the Image*. The malicious weights extracted from the model are saved based on the Grad-CAM mechanism, and finally the important part of the image is calculated by the weight and the image blocks obtained from the deep learning model. In the rest of this section, we will describe each stage in detail.

2.1. Observation and Analysis of the Dex File. Since the extraction of Data Areas in Dex files is the key to realizing ImageDroid, in this section, we mainly analyze the Dex structure and describe the reasons why Data Areas are chosen to represent Dex files.

After unzipping the APK file, we can directly obtain the Dex file. The Dex file format is a compressed format designed for Dalvik that stores data in bytecode. The structure of Dex file is shown in Figure 2. It is composed of three parts: Dex Header, Index Area, and Data Area. Finally, we extracted only the Data Area from the Dex file as a representation of the APK. Specific observations and analyses are shown below.

The Header of Dex describes the information of the Dex file and the index (offset address) of each area. For example, it describes the length field of the Dex file, the version number, the offset address of the area corresponding to the string, and the statistics of the number of strings is included. The size of the whole file header is fixed at 112 bytes. The header length of the Dex file of different Android applications does not change, but the value of the corresponding field changes. However, these changes in numbers have little to do with the maliciousness of Android applications. Therefore, we choose to remove the Dex Header.

The Index Area describes the offset address of each area's specific content in the Dex file. For example, in Figure 3, String_Ids record the offset addresses of all strings, but it is not real data, just an index, and the data are indexed by this value. These index values do not represent real data, so we remove the Index Area. The Data Area retains not only the structure of the entire APK file but also the real data, that is to say, all the real data used in the entire Android application are in this area, so we keep Data Area.

For the obfuscation of Android application code, there are usually the following three situations: substitution obfuscation, hidden obfuscation, and repeated function definitions. These obfuscations are implemented through the HackPoint configuration file to realize the obfuscation of the Dex file. The modified HackPoint is saved to the end of the Dex file, and there is no change in the Data Area area during the Dex obfuscation process. Therefore, using the Data Area of the Dex file can avoid the obfuscation technology to detect Android applications.

Through the above analysis, we find that different parts of Dex file are not effective for Android detection and classification. Finally, we choose Data Area extracted from Dex file as the research object. Because the Data Area not only contains all the structure information and real data of APK, it can also ensure that the data will not be affected during the confusion of Android applications.

2.2. Malicious Classification. Different from the existing studies, we do not need prior knowledge and feature extraction and only extracts the Data Area in the Dex file of the APK to realize the classification.

The implementation of classification consists of the following three steps, as depicted in Figure 3: (1) Extracting the Data Area of APK (corresponding to ① and ② in Figure 3); (2) converting Data Area to the RGB image (corresponding to ③ and ④ in Figure 3); (3) implementing classification of Android applications (corresponding to ⑤ and ⑥ in Figure 3). The specific implementation is shown in Figure 3.

(1) Extrating the Data Area of APK. We use the unzip tool to directly obtain the Dex file of the APK, as shown in ① in Figure 3. The Dex file is an executable file of the Android virtual machine, which is composed of three parts, as shown in Figure 2. Inspired by the observation of the Dex file structure in Section 2.1, we extract the most effective Data Area part for classification from the Dex file as the next operation object, as shown in ② in Figure 3.

(2) Converting Data Area to the RGB Image. The Data Area is in the form of bytecode, and it is much longer than the image format. Therefore, we need to process the Data Area to make it more consistent with the input of the deep learning model. We convert the bytecode into a multidimensional array by replacing the bytecode with a decimal number. We choose a

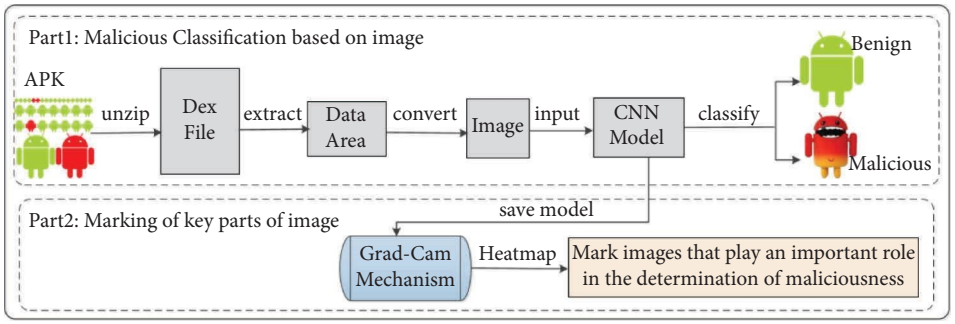

Figure 1: The framework of ImageDroid.

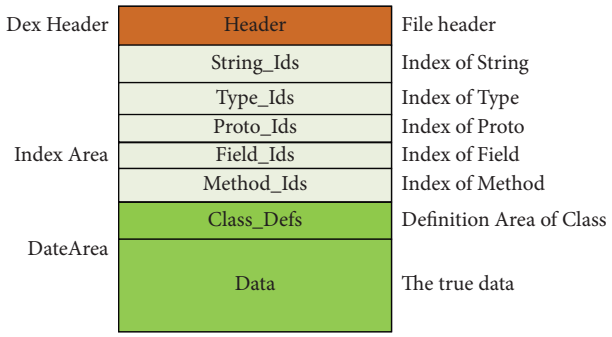

Figure 2: The structure of the Dex file.

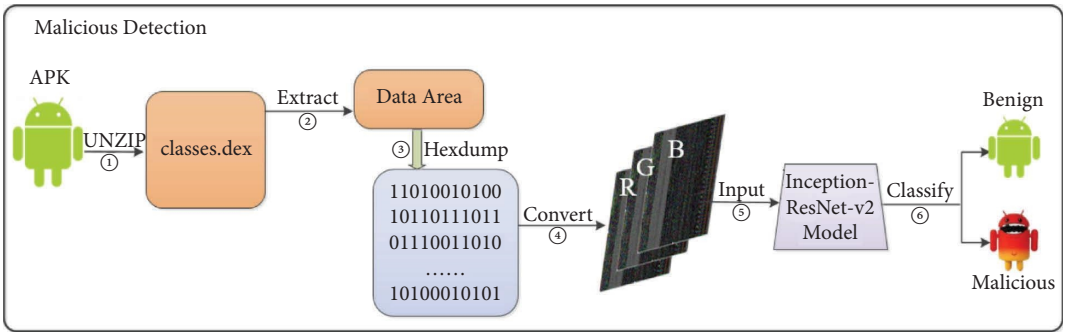

Figure 3: The framework of malicious detection.

three-dimensional array of $900 \times 900 \times 3$. This makes it possible to accommodate the length of most Android applications. This array can be converted into the RGB image. Each pixel in the image is the three consecutive bytecodes in the original bytecode. If the single channel is used, the length of gray image is too large. In order to reduce the size of the image, we use three channel color image. Due to the different lengths of bytecode, we discard the code segments larger than $900 \times 900 \times 3$ and add 0 after the code segments smaller than $900 \times 900 \times 3$. Our approach retains most of the bytecode sequence, but the original spatial structure may be changed during the period of transformation. This is also the shortcoming of our method. The bytecode of Data Area is shown in Figure 4. A square (two hexadecimal values) represents one pixel, for example, 64 represents a pixel.

To convert the bytecode to the RGB image, the RGB image corresponds to a three-dimensional array. Figure 5 shows that the bytecode corresponding to Figure 4 is converted into a three-dimensional array. For example, the first three-dimensional array shown in Figure 5 is [64,65,78], where 64, 65, and 78 are set to R: 64, G: 65, and B: 78, respectively.

Because the malicious application of the same family has the same malicious behavior, and the code similarity is extremely high, some malicious applications of the same family have great similarities in images. As shown in Figure 6, we show four malicious application images of the AnserverBot family.

(3) Implement Classification of Android applications. In order to realize the classification of Android applications, in this section, we complete the selection of the deep learning model and the implement detection or classification. The specific implementation details are given in the following sections.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000h :	64	65	78	0A	30	33	35	00	64	C1	29	B8	43	12	46	8C
0010h :	57	E0	2F	47	73	50	ED	1D	DC	12	27	DB	A0	78	BD	1A
0020h :	78	4F	08	00	70	00	00	00	78	56	34	12	00	00	00	00
0030h :	00	00	00	00	B4	4E	08	00	4B	10	00	00	70	00	00	00
0040h :	42	04	00	00	9C	41	00	00	0E	05	00	00	A4	50	00	00
0050h :	12	09	00	00	4C	8F	00	00	2C	14	00	00	DC	D7	00	00
0060h :	D6	02	00	00	3C	79	01	00	7C	7B	06	00	FC	D3	01	00
0070h :	04	5A	06	00	06	5A	06	00	09	5A	06	00	0D	5A	06	00
0080h :	1A	5A	06	00	23	5A	06	00	2E	5A	06	00	32	5A	06	00
0090h :	35	5A	06	00	3B	5A	06	00	3F	5A	06	00	4C	5A	06	00
00A0h :	61	5A	06	00	66	5A	06	00	7E	5A	06	00	83	5A	06	00
00B0h :	87	5A	06	00	9F	5A	06	00	B0	5A	06	00	B4	5A	06	00
00C0h :	B9	5A	06	00	C2	5A	06	00	D1	5A	06	00	DE	5A	06	00
00D0h :	EF	5A	06	00	F6	5A	06	00	FB	5A	06	00	FE	5A	06	00
00E0h :	12	5B	06	00	38	5B	06	00	59	5B	06	00	8B	5B	06	00
00F0h :	98	5B	06	00	A1	5B	06	00	BA	5B	06	00	EA	5B	06	00

Figure 4: Bytecode of Data area.

[64,65,78]	[0A,30,33]	[35,00,64]	[C1,29,B8]	[43,12,46]
[8C,57,E0]	[2F,47,73]	[50,ED,1D]	[DC,12,27]	[DB,A0,78]
[BD,1A,78]	[4F,08,00]	[70,00,00]	[00,78,56]	[34,12,00]
…	…	…	…	…
[5B,06,00]	[A1,5B,06]	[00,BA,5B]	[06,00,EA]	[5B,06,00]

Figure 5: Bytecode of Dalvik.

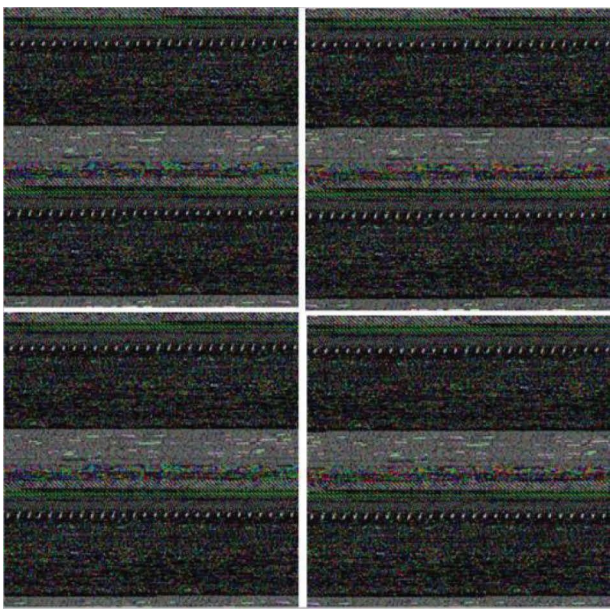

Figure 6: Four malicious applications of the AnserverBot family.

2.2.1. Selecting the Deep Learning Model. When designing the deep learning model, we need to consider the adaptive of different models. According to the needs of our method, we put forward the following three requirements for the deep learning model. (1) The model should be able to handle a large length of input, corresponding to our high pixel image of malicious applications. (2) Since the local and global features are considered in the deep learning model, the model can still achieve high detection accuracy although the malicious code is separated or discontinuous in the underlying bytecode. (3) We also want to obtain the important features learned from the network structure through the model, which can help the analysis of Android malicious applications in the next step.

In order to get the best structure of deep learning, we selectd the model based on the above three requirements. We tried several classic image classification models, such as VGGNet [13], GoogleNet [14], and ResNet [15]. After several years of development, these models have been proven to have a good generalization ability. These models are described in detail below.

VGGNet uses a smaller convolution kernel as a whole. The first several layers of the model are a stack of convolution layers, and the last several layers are full connection layer (FCL) and softmax layer. The activation function of all hidden layers uses the ReLU function. It uses several smaller convolution kernels instead of large convolution kernels to reduce the parameters and introduce more nonlinear factors to increase the fitting expression capability of the network.

GoogLeNet is derived from LeNet. At present, there are mainly four versions of Inception-(v1–v4). Each version is a little bit better than before and gets a better image classification effect. In this series of network structures, convolution kernels of different sizes are used to obtain receptive fields of different sizes. Finally, these features of different sizes are fused to extract better features. In addition, Inception-v [16] proposes batch normalization to reduce the variation of internal neuron data distribution. This setting normalizes the output of each layer to N(0,1) distribution, thus increasing the robustness of the model. It can also use larger learning rate training, faster convergence, and less influence of weight initialization. In addition, the model uses two 3×3 convolution kernels instead of one 3×3 convolution kernel to make the network deeper. After that, the Inception-v4 and Inception-ResNet use residual network to improve the previous network structure.

In consideration of the experimental comparative analysis of the above models (as shown in 3.2), and combined them with our needs, we finally chose the Inception-ResNet-v2.

2.2.2. The Implement of Classification or Detection. To implement detection or classification of Android apps using images, we first convert each APK into a $900 \times 900 \times 3$ RGB image. For example, if there is a dataset of N samples, an image input of $N \times 900 \times 900 \times 3$ is generated. We directly input these data into the Inception-ResNet-v2 model for training, as shown in Figure 3 ⑤. The trained model is to realize the detection and classification of Android applications.

2.3. Marking of Key Parts of the Image. To explain the neural network features, we used the Grad-CAM method [17], which achieves good results in the interpretation of image classification. The specific Grad-CAM method implementation framework is shown in Figure 7.

(1) *The Mechanism of Grad-CAM.*

Grad-CAM is an extended version of CAM and is commonly used in image classification. The goal of Grad-CAM is to obtain the heatmap for the images. In particular, the heatmap is the contribution score for every single pixel of the image. Grad-CAM believes that the last feature maps generated by the convolutional layer have the valuable information of the input data, and the final decision of the model is performed on it. Yet the influence of each feature map on the decision of the model is different. To reflect the difference, Grad-CAM computes the important scores of the feature maps by multiplying each feature map with its corresponding importance weight. Then, Grad-CAM takes the sum of the importance scores to summarize the scores of the feature maps contributing to the classification results. More specifically, we denote the one of the feature map from the last convolutional layer as A^m and the classification results as L^C. We can calculate the importance weight of A^m as

$$\alpha_m^c = \frac{1}{Z} \sum \frac{\partial L^c}{\partial A_i^m}, \tag{1}$$

where α_m^c is a constant that represents the importance score, and Z is the number of elements in A^m.

Assuming that we have M feature maps, the contribution scores of the input data can be calculated using a weighted combination of each feature map:

$$C_{\text{score}} = \text{ReLU} \sum_m^M \alpha_m^c A^m, \tag{2}$$

where the ReLU is applied to preserve the features that only have a positive influence on the classification result of *C*. Note that the size of the C_{score} should be smaller than the input data.

(2) *The Implementation of Marking Malicious Image.*

To implement the Grad-CAM, we use the AGP (average global pooling) technique to calculate the weighted class activation map.

As shown in Figure 7, since we only do maliciousness detection in the model, we only need two forms of labeling: normal and malicious. From the output of the model, we get the weights of all features judged as malicious (the red square in Figure 7 is maliciousness). Then the feature and its corresponding weight are multiplied to form a heatmap. The

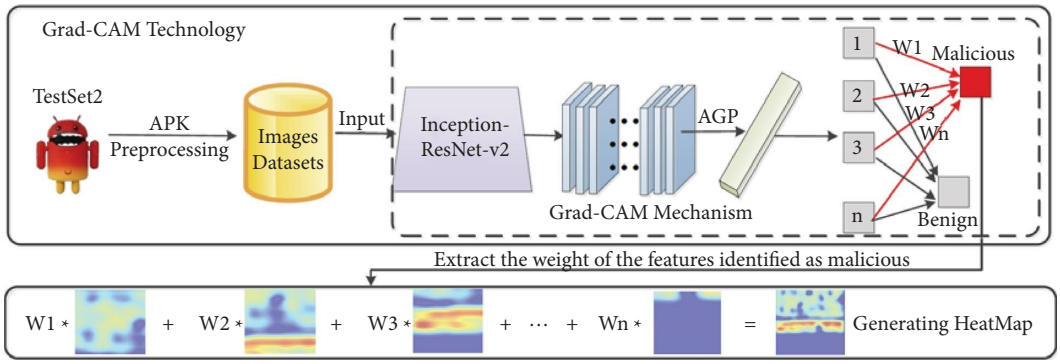

Figure 7: Use Grad-CAM to achieve interpretation.

brighter the heatmap, the more important that part is judged malicious. The malicious part of the image is explained by the heatmap.

3. Evaluation

In this section, we first introduce the dataset and experimental environment we use in the verification process. Then the feasibility of ImageDroid and its effectiveness in classifying malicious applications are verified. The specific detailed description is as follows.

3.1. Experimental Datasets and Environment. During the experiment, we evaluate ImageDroid using six datasets. The samples in all these datasets are not only labeled as normal or malicious but also contain family labels. The details of these datasets are shown in Table 1. Our experiment is based on the Tensorflow framework, and our model is trained on 4 Nividia Titan XP. In this experiment, we use four indicators to evaluate the performance of the model, and they are accuracy, precision, recall, F1-score.

3.2. Verify the Performance of Deep Learning Model. In order to realize the marking of important parts of images in malicious detection, we must have a good deep neural network model for image classification. Therefore, we verify the effect of different models in using images for malicious detection based on dataset 1. As shown in Figure 8, we use four typical neural network structures for validation. As can be seen from the evaluation metrics, Inception-ResNet-v2 performs the best. Therefore, the deep learning model selected by image is Inception-ResNet-v2.

To verify whether the Inception-ResNet-v2 model converges or not during the training process, we present the ROC curve of the Inception-ResNet-v2 model after multiple iterations, as shown in Figure 9. As the number of iterations increases, the closer it is to the upper left, and the AUC (area under curve) value also increases. These data indicate that the model converges with the iterative growth.

3.3. Verify the Validity of the Extracted Data Area. In order to verify the effectiveness of the ImageDroid for extracting Dex files, we use Dataset 1 and Dataset 4 based on the Inception-ResNet-v2 model to verify the effect on classification. We name the method as AllImageDroid that uses the entire Dex file to convert to images for classification. The effectiveness of Dex file extraction is verified by comparing the classification results of AllImageDroid and ImageDroid. After we input the data extracted by these two methods into the deep learning model for training, the different evaluation indicators obtained are shown in Table 2.

Table 1: Datasets used in our experiment.

Name	Source	#Benign	#Malicious
Dataset 1	Drebin [4]	–	5500
Dataset 2	MalGenome [18]	–	1250
Dataset 3	Android PRAGuard [19]	–	1250
Dataset 4	Google play store	5500	–

Through the classification evaluation indicators in Table 2, we find that the extraction of Dex files can have a certain effect on improving the classification effect.

3.4. Verification of the Effect of Malware Family Classification. Because of the great similarity after converting to images by analyzing the Android malicious applications in the same family, we classify Android malicious app families based on the Data Area which is converted to images. Next, we perform malware family classification validation on the dataset 1.

During the process of malware family classification, we find that the difference in the number of samples in different families is too large to bias the detection results. For example, the number of samples in some malicious application families is only 2. To address this issue, we select the top 20 families from the Drebin dataset for validation. Each family name and the corresponding sample number are shown in Table 3. The classification of ImageDroid for each family is shown in Table 4.

As can be seen from Table 4, the ImageDroid can classify the families in the dataset well, with an average recall rate of 96.7%. Among them, ten families are classified perfectly, and the recall rate achieve 99%. Because the training process of the deep learning model is related to the number of samples, the above experimental data show that our method is also effective in malware family classification.

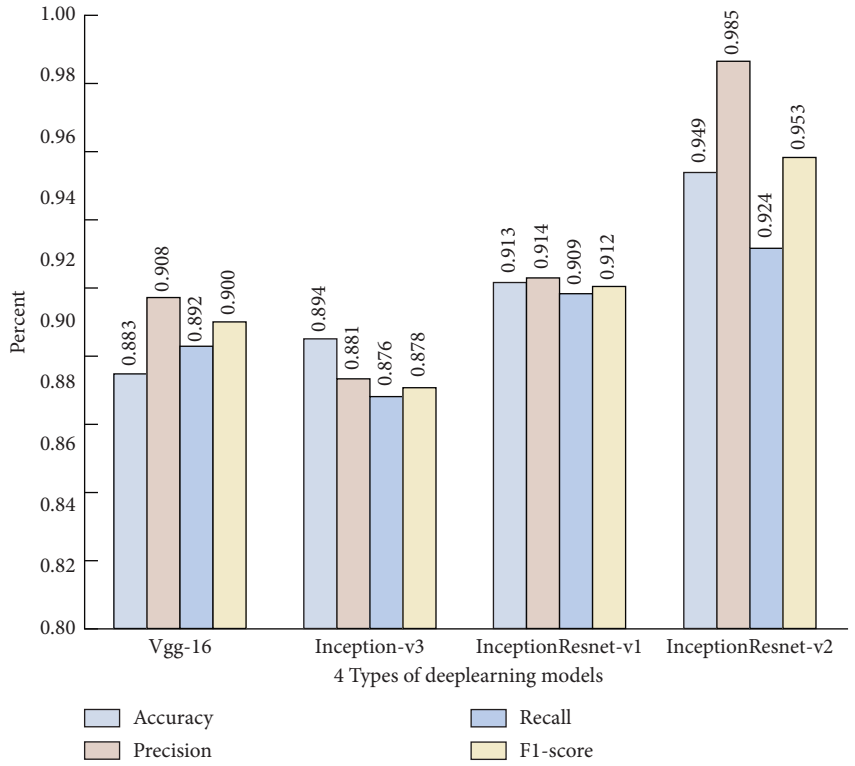

Figure 8: Classification and comparison of different models based on Drebin.

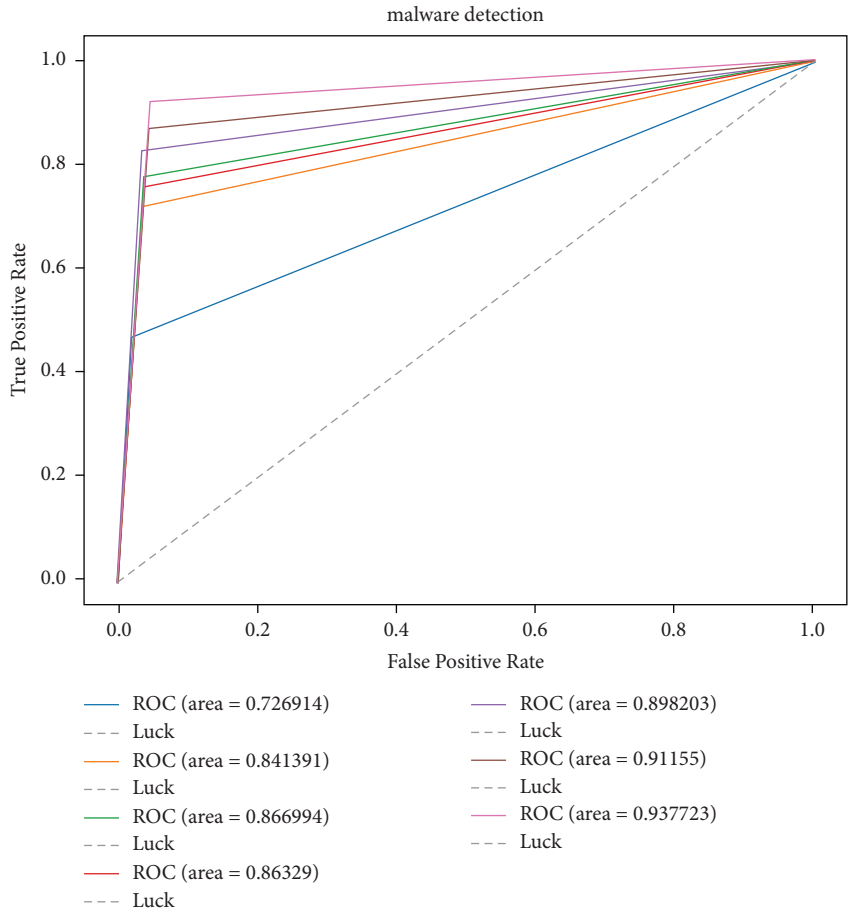

Figure 9: ROC Curve of InceptionResNet-v2.

Table 2: Comparison of classification effects using different regions of Dex.

Method	Accuracy	Precision	Recall	F-value
AllImageDroid	0.931	0.952	0.919	0.934
ImageDroid	0.949	0.985	0.929	0.953

Table 3: Top malware family used in Debin.

No.	Family name	Num	No.	Family name	Num
1	FakeInstaller	925	11	Adrd	91
2	DroidKungFu	667	12	DroidDream	81
3	Plankton	625	13	LinuxLotoor	70
4	OpFake	613	14	GoldDream	69
5	GingerMaster	339	15	MobileTx	69
6	BaseBridge	330	16	FakeRun	61
7	IconoSys	152	17	SendPay	59
8	$K_{\min}$	147	18	Gappusin	58
9	FakeDoc	132	19	Imlog	43
10	Geinimi	92	20	SMSreg	41

Table 4: Evaluation indicator of multifamily classification.

Family	TPR	FPR	Precision	Recall	F-value
FakeInstaller	0.994	0.001	0.998	0.994	0.996
DroidKungFu	0.992	0	0.996	0.992	0.994
Plankton	1	0	1	1	1
OpFake	0.989	0.001	0.957	0.903	0.927
GingerMaster	1	0	1	1	1
BaseBridge	0.892	0.005	0.947	0.892	0.918
IconoSys	1	0	1	1	1
$K_{\min}$	0.935	0.001	0.977	0.935	0.956
FakeDoc	0.989	0	1	0.989	0.995
Geinimi	0.909	0.001	0.909	0.909	0.909
Adrd	0.914	0.001	0.981	0.914	0.946
DroidDream	0.909	0	1	0.909	0.952
LinuxLotoor	0.989	0	1	0.989	0.995
GoldDream	1	0	1	1	1
MobileTx	1	0.004	0.936	1	0.958
FakeRun	1	0	1	1	1
SendPay	0.833	0	1	0.833	0.909
Gappusin	0.818	0	1	0.818	0.9
Imlog	0.894	0	1	0.894	0.944
SMSreg	0.889	0	1	0.889	0.941
Avg	0.967	0.001	0.979	0.967	0.968

3.5. Validation of Detection Validity for Obfuscated Applications. In this section of experimental validation, we evaluate the robustness of ImageDroid in detecting obfuscated malicious applications. The current obfuscation technology on the Android platform is very mature, and there are many obfuscation frameworks available. We choose the dataset 3 of Android PRAGuard [19] for validation. Five obfuscation techniques are used in the dataset 3(the details are shown in Table 5).

We use ImageDroid for detection on the obfuscated dataset. We use all Dex files and ImageDroid method (using partial Dex files) for comparison and verification, respectively.

From the detection results in Table 5, it can be seen that the ImageDroid method still has a good detection rate on different obfuscated malware. This is exactly the benefit of the ImageDroid method using partial Dex files instead of decompilation to obtain features. In this way, we solve the problem of low detection of obfuscated malicious applications by static features. This makes it possible to detect obfuscated malicious applications without the need to dynamically run malicious applications.

3.6. An Example of Image-Based Feature Marking. We use Grad-CAM technology to visualize the parts of the image that are important for maliciousness determination. In this section, we demonstrate two APKs from the Geinimi family of GenoneProject. The two MD5 values are f3736147f7d46c5d96f8ae9f89bfd1f694b-b871a and bc3790cdc8ae0ee7da7d6e3fd397d2a720e00e67. These two APKs are used to generate a malicious heatmap based on Grad-CAM technology.

As shown in the heatmap in Figure 10, different colors indicate different importance in determining maliciousness. The orange color indicates the most malicious part and the blue color indicates the least malicious part. The two images of the malware are marked in orange around the positions of 400, which corresponds to an important part of the image to judge maliciousness.

4. Related Work

Malicious detection of Android has always been the focus of Android research. In view of the different current research methods, we divide the research methods into two categories: decompilation and unable for decompilation.

Detection that can be decompilation. This type detection method needs to decompile the Android application and then extracts different features from the decompiled files for malicious detection. The disadvantage of this type is that Android applications must be decompiled. Liu et al. [20] proposed a malicious application detection method for Android based on the multilevel signature matching algorithm. Through this method, API, method, class, and APK of each APK are signed separately. Finally, the same signature is founded by the matching algorithm to detect malicious application. Arp et al. [4] proposed Drebin, which performs extensive static analysis and collects as many application features as possible, such as permissions, API calls, and strings in the Dalvik code. Then these features are embedded into a joint vector space for Android malware analysis. Zhang et al. [21] proposed DroidSIFT, which constructs a weighted context API dependency graph database and generates graph-based feature vectors through graph similarity query. Fan et al. [22] proposed the faldroid method, which constructs frequent subgraph database through the call relationship of function call graph and classifies malicious applications by frequent subgraph to characterize the maliciousness of malicious applications. Liu et al. [8] proposed to use neighbor signature to classify Android malicious families. Based on neighborhood signature to acquire

Table 5: Classification result of per malware family.

ID	Obfuscated method	Malicious detection using Dex (recall)	Malicious detection using Data Area (%) (recall)
1	String Encrption	91.3	94.8
2	Class Encrption	90.2	93.2
3	Method Encrption	92.8	95.0
4	Combined 1 and 3	91.9	94.9
5	Combined 1, 2, and 3	90.6	95.4
6	No obfuscation	**96.2**	**97.9**

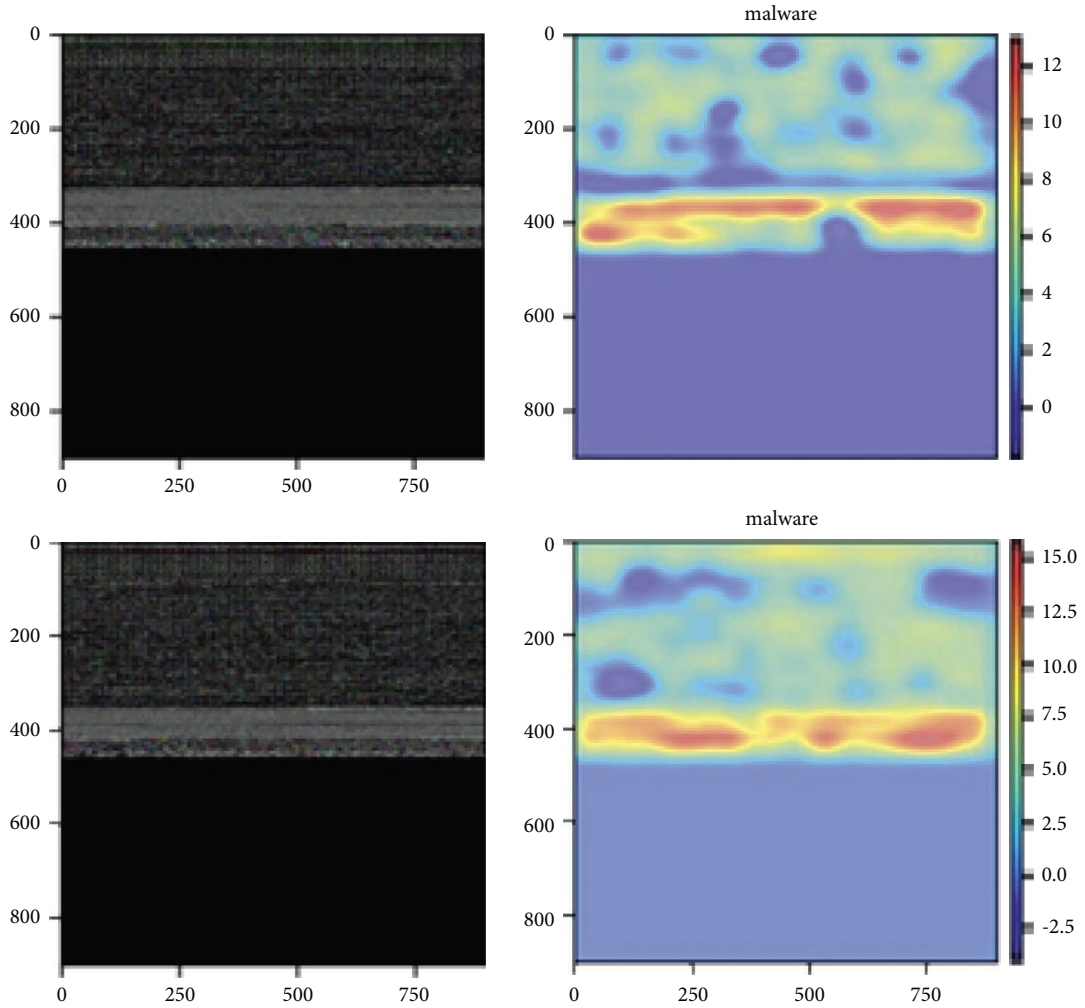

Figure 10: Two malicious applications generate HeatMap through Grad-CAM.

the similarity of different applications' FCGs, which was significantly faster than traditional approaches based on subgraph isomorphism.

Detection that cannot be decompilation. With the development of encryption technology, some Android applications cannot be decompiled. In this case, there are some methods that can detect malice without decompilation technology. These types of methods are to use DEX files directly for malicious detection. Ni et al. [23] transformed the operation code in the disassembled malware code into gray image, and then recognized the classification of malicious multifamily in the Windows system through the convolutional neural network. Han et al. [24] proposed to convert DEX into an image and then extract the entropy graph from the image as a feature for malicious detection. Bakour et al. [9] extracted local and global features from DEX converted images. Then, multiple local feature descriptions are extracted from each image to form a feature vector, which is used for malicious detection. Mercaldo et al. [25] used the GIST method to generate a set of features from the image corresponding to each application to detect malicious applications and classify malware families.

5. Conclusion

In this paper, we propose a method for malicious detection and multifamily classification without decompiling applications and prior knowledge. The results show that our method can not only effectively detect malicious of android Application but also classify multiple families. Based on the method, we annotate the classification results with the interpretable mechanism of deep learning model. This not only provides a good solution for malicious detection of Android applications that cannot be decompiled but also enables further fine-grained analysis by locating the part of the image that is important for determining maliciousness.

Data Availability

So far, we have obtained public datasets such as (1) https://www.malgenomeproject.org/, (2) https://pralab.diee.unica.it/en/AndroidPRAGuardDataset, and (3) https://www.sec.cs.tu-bs.de/~danarp/drebin/. We have already crawled a lot of Android apps across the other web.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant No. 61672543 and the Fundamental Research Funds for the Central Universities of Central South University under Grant no. 2018zzts175. The authors also appreciate the efforts from the reviewers.

References

[1] C. reports, "AMobile malicious application statistics website," 2022, https://www.counterpointresearch.com/global-smartwatch-shipments-market-share/.

[2] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Computing Surveys*, vol. 50, no. 3, pp. 1–40, 2018.

[3] Y. Xue, G. Meng, Y. Liu et al., "Auditing anti-malware tools by evolving android malware and dynamic loading technique," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, pp. 1529–1544, 2017.

[4] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, *Drebin: Effective and Explainable Detection of Android Malware in Your Pocket*, NDSS'2014, California, CL, USA, 2014.

[5] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "Hindroid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information network," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1507–1515, Halifax, NS, Canada, June 2017.

[6] M. Y. Wong and D. Lie, "IntelliDroid: a targeted input generator for the dynamic analysis of android malware," in *Proceedings of the Network and Distributed System Security Symposium*, pp. 21–24, February 2016.

[7] C. Wang, Z. Li, X. Mo, H. Yang, and Y. Zhao, "An android malware dynamic detection method based on service call co-occurrence matrices," *Annals of Telecommunications*, vol. 72, no. 9–10, pp. 607–615, 2017.

[8] P. Liu, W. Wang, X. Luo, H. Wang, and C. Liu, "NSDroid: efficient multi-classification of android malware using neighborhood signature in local function call graphs," *International Journal of Information Security*, vol. 20, pp. 59–71, 2021.

[9] K. Bakour and H. M. Ünver, "DeepVisDroid: android malware detection by hybridizing image-based features with deep learning techniques," *JCR1_Neural Comput. Appl.*vol. 16, 2021.

[10] A. Cvpr and P. Id, "Iterative visual reasoning beyond convolutions," in *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Salt Lake City, UT, USA, June 2018.

[11] Y. Ding, X. Zhang, J. Hu, and W. Xu, "Android malware detection method based on bytecode image," *Journal of Ambient Intelligence and Humanized Computing*, vol. 32, pp. 1–10, 2020.

[12] S. D. Sl and J. Cd, "Windows malware detector using convolutional neural network based on visualization images," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 2, pp. 1057–1069, 2021.

[13] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," *Proc. IEEE Conf. Comput. Vis. pattern Recognit.*, pp. 2921–2929, 2016.

[14] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proceedings of the 2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, pp. 1–14, Kuala Lumpur, Malaysia, November 2015.

[15] G. Zeng, Y. He, Z. Yu, X. Yang, R. Yang, and L. Zhang, "Going deeper with convolutions," in *Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, Boston, MA, June 2015.

[16] V. Sangeetha and K. J. R. Prasad, "Deep residual learning for image recognition," *Proc. IEEE Conf. Comput. Vis. pattern Recognit.*vol. 45, pp. 1951–1954, 2006.

[17] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: visual explanations from deep networks via gradient-based localization," *Rev. do Hosp. das Classifications*, vol. 17, pp. 331–336, 2016.

[18] MalgenomeProject_Dataset, "come from North Carolina state university," 2021, http://www.malgenomeproject.org/.

[19] PraguardDataset, "An Android Dataset Using seven different obfuscation," 2022, http://pralab.diee.unica.it/en/AndroidPRAGuardDataset.

[20] X. Y. Liu, J. Weng, Y. Zhang, B. W. Feng, and J. S. Weng, "Android malware detection based on APK signature information feedback," *Journal on Communications*, vol. 38, pp. 190–198, 2017.

[21] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "DroidSIFT: Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs," *Categories and Subject Descriptors*, vol. 13, pp. 1105–1116, 2014.

[22] M. Fan, L. Jun, L. Xiapu, and K. Chen, "Frequent subgraph based familial classification of android malware," in *Proceedings of the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, Ottawa, ON, Canada, December 2016.

[23] S. Ni, Q. Qian, and R. Zhang, "Malware identification using visualization images and deep learning," *Computers & Security*, vol. 77, pp. 871–885, 2018.
[24] K. S. Han, J. H. Lim, B. Kang, and E. G. Im, "Malware analysis using visualized images and entropy graphs," *International Journal of Information Security*, vol. 14, pp. 1–14, 2015.
[25] F. Mercaldo and A. Santone, "Deep learning for image-based mobile malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 16, no. 2, pp. 157–171, 2020.