Contrastive Learning for Robust Android Malware Familial Classification

Yueming Wu, Shihan Dou, Deqing Zou, Wei Yang, Weizhong Qiang, and Hai Jin, *Fellow, IEEE*

Abstract—Due to its open-source nature, Android operating system has been the main target of attackers to exploit. Malware creators always perform different code obfuscations on their apps to hide malicious activities. Features extracted from these obfuscated samples through program analysis contain many useless and disguised features, which leads to many false negatives. To address the issue, in this paper, we demonstrate that obfuscation-resilient malware family analysis can be achieved through contrastive learning. The key insight behind our analysis is that contrastive learning can be used to reduce the difference introduced by obfuscation while amplifying the difference between malware and other types of malware. Based on the proposed analysis, we design a system that can achieve robust and interpretable classification of Android malware. To achieve robust classification, we perform contrastive learning on malware samples to learn an encoder that can automatically extract robust features from malware samples. To achieve interpretable classification, we transform the function call graph of a sample into an image by centrality analysis. Then the corresponding heatmaps can be obtained by visualization techniques. These heatmaps can help users understand why the malware is classified as this family. We implement *IFDroid* and perform extensive evaluations on two datasets. Experimental results show that *IFDroid* is superior to state-of-the-art Android malware familial classification systems. Moreover, *IFDroid* is capable of maintaining a 98.4% F1 on classifying 69,421 obfuscated malware samples.

Index Terms—Android malware, Obfuscation-resilient, Familial classification, Contrastive learning

1 Introduction

As the most widely used mobile operating system [1], the security of Android platform has become more and more closely related to personal privacy and financial security. Meanwhile, due to the open-source and market openness of Android operating system, it is more likely to be exploited by malware [2]. To hide their malicious tasks, different code obfuscations have been applied by attackers [3], [4]. After obfuscations, malware samples become more complex, resulting in features obtained from them containing many useless and camouflage features. These futile features make it difficult to perform accurate behavioral analysis of Android malware. Therefore, it is important to provide obfuscation-resilient Android malware analysis.

Most traditional Android malware analysis methods [5], [6], [7] cannot resist code obfuscations. For example, for familial classification of Android malware, it can be roughly divided into two main categories [8], namely string-based approaches (*e.g.,* permissions [5]) and graph-based techniques (*e.g.,* function call graph [9]). Some methods [5], [6], [7] focus on permissions requested by apps and search for the presence of several strings (*e.g.,* API calls) from disassembling code to build models to analyze Android malware. However, they can be easily evaded by obfuscations because of the lack of structural and contextual information of the program behaviors. To achieve more robust malware classification, studies [8], [9] distill the program semantics of apps into graph representations and apply graph matching to analyze the malware families. For example, *FalDroid* [8] extracts the function call graph of an app and applies frequent subgraph analysis to classify Android malware. However, Hammad and Dong *et al.* [3], [4] report that malware creators always perform complex obfuscations (*e.g.,* control-flow obfuscations) to hidden their malicious tasks. In this case, features extracted from graphs obtained by *FalDroid* [8] may not be accurate since graphs may change a lot after applying advanced code obfuscations. In one word, due to different code obfuscations, features obtained from malware samples may contain many useless and disguised features, making it difficult to achieve accurate behavioral analysis.

To address the issue, we propose to use contrastive learning on Android malware analysis. Due to the powerful high-level feature extraction of contrastive learning, it has been widely used in different areas, such as text representation learning [10] and language understanding [11]. To the best of our knowledge, we are the first to use contrastive learning to resist code obfuscations. To demonstrate the ability of contrastive learning on analyzing obfuscated Android malware, in this paper, we propose a novel approach that can achieve obfuscation-resilient Android malware classifi-

- *Y. Wu, D. Zou (corresponding author), and W. Qiang are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China. E-mail: wuyueming21@gmail.com, deqingzou@hust.edu.cn, wzqiang@hust.edu.cn*
- *S. Dou is with Shanghai Key Laboratory of Intelligent Information Processing, School of Computer Science, Fudan University, Shanghai, 200433, China. E-mail: shihandou@foxmail.com*
- *W. Yang is with University of Texas at Dallas, Dallas, USA. E-mail: wei.yang@utdallas.edu*
- *H. Jin is with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China. E-mail: hjin@hust.edu.cn*

cation.

Specifically, we first obtain the function call graph of an app and then apply centrality analysis [12] to transform the graph into an image. The generated images are used to train an encoder by contrastive learning. The use of contrastive learning is to maximize the similarity between positive samples and minimize the similarity between negative samples. In practice, although applying obfuscations may change the app codes, the inherent program semantics do not change. In other words, the obfuscated app can be treated as one of the positive samples of the original app. Therefore, we can leverage contrastive learning to reduce the differences introduced by code obfuscations while enlarging the differences between different types of malware, making it possible to correctly classify the obfuscated malware into the corresponding family.

To further show how contrastive learning improves the usability of malware analysis, we apply visualization techniques to visualize the valuable features extracted by contrastive learning. Specifically, we apply *Gradient-weighted Class Activation Mapping++* (Grad-CAM++) [13], [14] on our images to obtain the corresponding heatmaps. Grad-CAM++ is a class-discriminative localization technique that generates visual explanations for any CNN-based network without changing the architecture or retraining. According to the intensity of the color in the heatmap, we can know which features are more effective in classifying this malware as this family. These valuable features can represent the essential behaviors to explain why the malware is classified as this family.

We implement *IFDroid* and conduct evaluations on two datasets. Through the comparative experimental results, we find that *IFDroid* is superior to ten state-of-the-art Android malware familial classification systems (*i.e., Dendroid* [15], *Apposcopy* [16], *DroidSIFT* [17], *MudFlow* [18], *DroidLegacy* [19], *Astroid* [20], *FalDroid* [8], *AOM* [21], *MVIIDroid* [22], and *CDFG* [23]). As for obfuscations, *IFDroid* can maintain a 98.4% F1 on classifying 69,421 obfuscated malware samples. As for interpretability, our experiments show that the heatmaps of most malware in the same family are similar, and the heatmaps of malware in different families are different. This result is in line with expectations and it can help security analysts analyze the specific reasons why malware samples are classified as corresponding families. As for runtime overhead, *IFDroid* requires an average of 1.78 seconds to complete the classification and 1.62 seconds to interpret the classification result in our dataset. Such results indicate that *IFDroid* can conduct large-scale malware analysis.

In summary, this paper makes the following contributions:

- To the best of our knowledge, we are the first to use contrastive learning to resist code obfuscations of Android malware. Contrastive learning can not only improve the accuracy but also enhance the robustness of Android malware analysis.
- We design a novel system (*i.e., IFDroid*) by transforming the function call graph of an app into an image and performing contrastive learning on generated images. *IFDroid* can achieve robust and interpretable classification of Android malware.
- We conduct evaluations on two datasets and results indicate that *IFDroid* is superior to ten state-of-the-art Android malware classification systems (*i.e., Dendroid* [15], *Apposcopy* [16], *DroidSIFT* [17], *MudFlow* [18], *DroidLegacy* [19], *Astroid* [20], *FalDroid* [8], *AOM* [21], *MVIIDroid* [22], and *CDFG* [23]).

Paper organization. The remainder of the paper is organized as follows. Section 2 presents our motivation. Section 3 introduces our system. Section 4 reports the experimental results. Section 5 discusses the future work and limitations. Section 6 describes the related work. Section 7 concludes the present paper.

2 Motivation Scenario

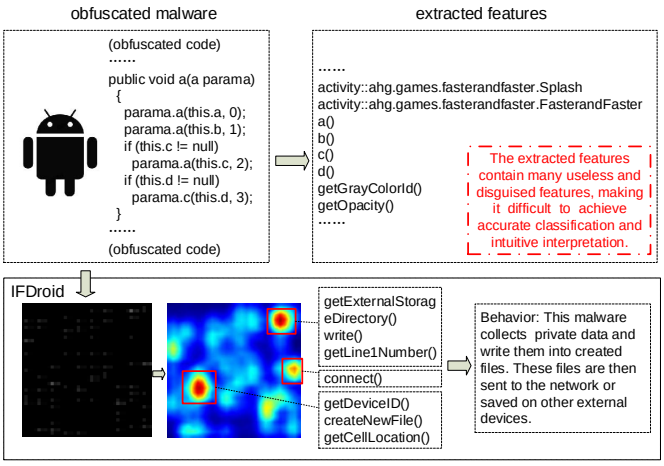

Fig. 1: Motivation scenario of *IFDroid*

Assuming that there is a security analyst, his daily task is to classify newly detected malware into corresponding families to enrich their malware family dataset. The richer the family dataset, the more accurately the unknown malware behavior can be predicted. AV-TEST Institute [24] reported that the average number of new malware detected per day is about 9,000. It will be a very time-consuming project if the analyst conducts in-depth manual analysis on these malware samples one by one. Therefore, the analyst decides to extract the semantic information of the samples by program analysis, and then classify them into their corresponding families through semantic similarity matching (*e.g.,* graph matching). But in fact, obfuscation technology has become more and more advanced, and it is used more and more frequently by attackers [3], [4]. In other words, these samples may be applied to different obfuscation techniques, resulting in the extracted features containing many useless and camouflage features. At the same time, after classifying the samples into their families, the analyst wants to know which semantic features make these malware samples classified into corresponding families. However, the classification method can only tell us which family the malware belongs to, and will not explain which features are used to determine that they are classified into this family. The whole scenario is shown in Figure 1.

To address the above challenges in the scenario, we first transform the program semantics of samples into images and then train a robust encoder by contrastive learning. In classification phase, we use a visualization technique to obtain the corresponding heatmaps of generated images. These heatmaps can help security analysts understand which features are more valuable in classifying them as corresponding families. We implement *IFDroid* to complete the whole analysis process automatically.

3 System Architecture

In this section, we introduce *IFDroid*, a novel contrastive learning-based robust and interpretable Android malware classification system.

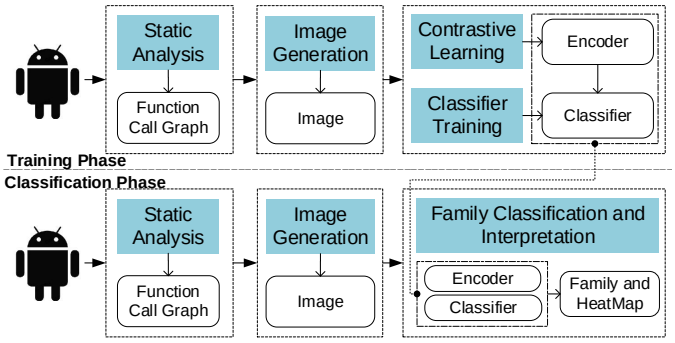

Fig. 2: System overview of *IFDroid*

3.1 Overview

As shown in Figure 2, *IFDroid* consists of two main phases: *Training Phase* and *Classification Phase*.

The goal of *Training Phase* is to train a robust encoder and an accurate classifier. This phase consists of four steps as follows. *1) Static Analysis*: This step aims to extract the function call graph of a malware sample based on static analysis where each node is an API call or a user-defined function. *2) Image Generation*: This step aims to transform the function call graph into an image based on centrality analysis. *3) Contrastive Learning*: This step aims to learn an encoder that can automatically extract robust features from an image. *4) Classifier Training*: This step aims to use vectors encoded by the learned encoder to train an accurate classifier.

The purpose of *Classification Phase* is to classify unlabeled malware into their corresponding families. This phase includes three steps: *1) Static Analysis*, *2) Image Generation*, and *3) Family Classification and Interpretation*. The first two steps are the same as in *Training Phase*. Given an image, it will be fed into a learned encoder in *Training Phase* to obtain the vector representation. The vector is then labeled as corresponding family by a trained classifier in *Training Phase*. To interpret the classification result, we use a deep visualization technique to obtain the corresponding heatmap of the image to help the security analyst understand why it is classified as this family.

3.2 Static Analysis

Empirical studies [8], [17] have demonstrated that graph representation is more robust than string-based features. In this paper, we aim to achieve efficient malware analysis. Therefore, we perform low-cost program analysis (*e.g.*, context- and flow-insensitive analysis) to distill the program semantics of a malware sample into a function call graph. More specifically, we leverage a widely used Android reverse engineering tool namely Androguard [25] to complete our static analysis.

To better describe the detailed steps in *IFDroid*, we choose a real-world malware sample[1] as our example. Figure 3 shows the sample's function call graph where each node is an API call or a user-defined function. The number of nodes and edges are 117 and 170, respectively.

3.3 Image Generation

On the one hand, deep-learning-based image classification can process millions of images while maintaining high accuracy. On the other hand, the output of image classification can be visualized to give a better intuition to users rather than giving a single decision. Because of these advantages, image-based methods have been widely used in malware analysis. However, most of these approaches [26], [27], [28], [29] only use simple mapping algorithms to transform malware samples into images and then apply deep learning to analyze them. Thus the semantics of the malware samples may be ignored.

To achieve efficient and semantic malware analysis, we use the technique (*i.e.*, centrality analysis) in our previous work [12] to transform the function call graph into an image. The centrality concept was first proposed in social network analysis whose purpose is to dig out the most important persons in the network. It can measure the importance of a node in a network and is very useful for network analysis. In practice, there have been proposed many studies to use centralities in different areas such as biological network [30], co-authorship network [31], transportation network [32], criminal network [33], etc

Different centralities analyze the importance of a node in a network by performing different network analyses, therefore, they have the potential to preserve different structural properties of a network. In our paper, we select four widely used centrality measures (*i.e., Degree centrality* [34], *Katz centrality* [35], *Closeness centrality* [34], and *Harmonic centrality* [36]) to commence our image generation phase. These four centralities can represent graph details from four different aspects. By this, we can achieve more complete preservation of a function call graph's semantics. Specifically, the definitions of these four centralities are as follows.

- *Degree Centrality* [34] assigns an importance score based simply on the number of edges held by each node. It is normalized by dividing by the maximum possible degree in a graph $N-1$, where N denotes the number of nodes within the graph, $deg(i)$ is the degree of node i.

$$C_d(i) = \frac{deg(i)}{N-1} \tag{1}$$

1. 485c85b5998bfceca88c6240e3bd5337

- *Katz Centrality* [35] computes the relative influence of a node within a graph by measuring the number of the immediate neighbors and also all other nodes in the graph that connect to the node under consideration through these immediate neighbors. If $C_k(i)$ denotes Katz centrality of a node i, where the element at location (i, j) of the adjacency matrix A raised to the power k (*i.e.*, A^k) reflects the total number of k degree connections between nodes i and j. The α denotes an attenuation factor, then mathematically:

$$C_k(i) = \sum_{k=1}^{\infty} \sum_{j=1}^{\infty} \alpha^k (A^k)_{ji} \tag{2}$$

- ***Closeness centrality*** [34] indicates how close a node is to all other nodes in the network. It is calculated as the average of the shortest path length from the node to every other node in the graph. The smaller the average shortest distance of a node, the greater the closeness centrality of the node. In other words, the average shortest distance and the corresponding closeness centrality are negatively correlated. If $d(i, j)$ is the distance between nodes i and j and N is the number of nodes in the graph, then mathematically:

$$C_c(i) = \frac{N-1}{\sum_{i \neq j} d(i, j)} \tag{3}$$

- ***Harmonic centrality*** [36] reverses the sum and reciprocal operations in the definition of closeness centrality. If $d(i, j)$ is the distance between nodes i and j and N is the number of nodes in the graph, then mathematically:

$$C_h(i) = \frac{\sum_{i \neq j} \frac{1}{d(i,j)}}{N-1} \tag{4}$$

On the one hand, Android apps use API calls to access operating system functionality and system resources. On the other hand, malware samples always invoke sensitive API calls to perform malicious tasks. For example, *getDeviceID* can get your phone's IMEI and *getLine1Number* can obtain your phone number. Therefore, we can leverage sensitive API calls to characterize the malicious behaviors of malware samples. Specifically, we choose 426 sensitive API calls [37] as our concerned objectives which compose of three different API call sets. The first API call set is the top 260 API calls with the highest correlation with malware, the second API call set is 112 API calls that relate to restrictive permissions, and the third API call set is 70 API calls that are relevant to sensitive operations. Since the same API call may exist in different subsets, after calculating the union set of these three API call sets, the total number of API calls is not 442, but 426.

Given a function call graph, we first apply centrality analysis to obtain four centrality values of sensitive API calls. If sensitive API calls do not appear in the function call graph, the four centrality values will all be zero. For example, the malware sample in Figure 3 invokes a total of three sensitive API call (*i.e.*, *sendTextMessage()*, *getDefault()*, and *setText()*). Then the four centrality values of these three sensitive API calls can be computed by centrality analysis.

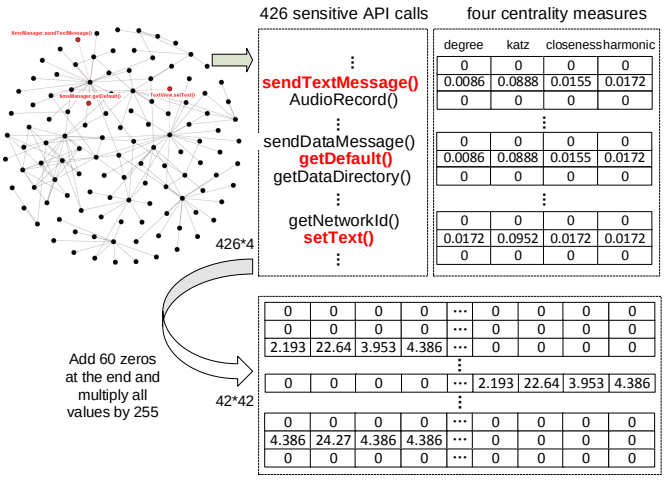

Fig. 3: An example to illustrate the image generation step of *IFDroid*

Other 423 sensitive API calls do not appear in the call graph, therefore, their centrality values are all zero. After centrality analysis, we can obtain a 426∗4 vector representation. If we directly transform it into an image, the generated image will be too narrow, making it difficult to distinguish which area it is when using heatmap for interpretation. Moreover, it is not conducive to viewing. Therefore, we crop the vector and turn it into a more square image. Specifically, we add 60 zeros at the end and then reshape it as a 42∗42 (*i.e.*, 426∗4+60=42∗42) vector. At the same time, in order to be able to see more clearly, we multiply all the values in the vector by 255 to brighten the pixels in the image. The range of centrality values is between zero and one, and the range of image pixels is between 0 and 255. After the two are multiplied, the range is also between 0 and 255. Finally, we can obtain a 42∗42 image. Each pixel in the image represents a certain centrality value for a sensitive API call. Even if these pixels are separated, each pixel is still meaningful and can still represent the graph details of the sensitive API call.

3.4 Contrastive Learning

In our daily life, humans can recognize objects in the wild, even if we do not remember the exact appearance of the object. This happens because we have retained enough high-level features of the object to distinguish it from others and ignored pixel-level details. For example, despite we have seen what a dollar bill looks like many times, we rarely draw a dollar bill exactly the same. However, although we cannot draw a lifelike dollar bill, we can easily distinguish it [38]. Therefore, researchers have asked a question: *Can we build a representation learning algorithm that does not pay attention to pixel-level details and only encodes high-level features that are sufficient to distinguish different objects?* To answer the question, contrastive learning is proposed.

The goal of contrastive learning is to maximize the agreement between original data and its positive data while minimize the agreement between original data and its negative data by using a contrastive loss in the vector space. Note that x is a sample, x^+ is a positive (*i.e.*, similar) sample of x, and x^- is a negative (*i.e.*, dissimilar) sample of x. Encoder f can encode samples into vector representations. s is a function

that computes the similarity between two vectors. In self-supervised contrastive learning, the positive sample x^+ of an image x is constructed by data augmentations such as image rotation and image cropping. As for negative sample x^-, any other images can be selected as x^-.

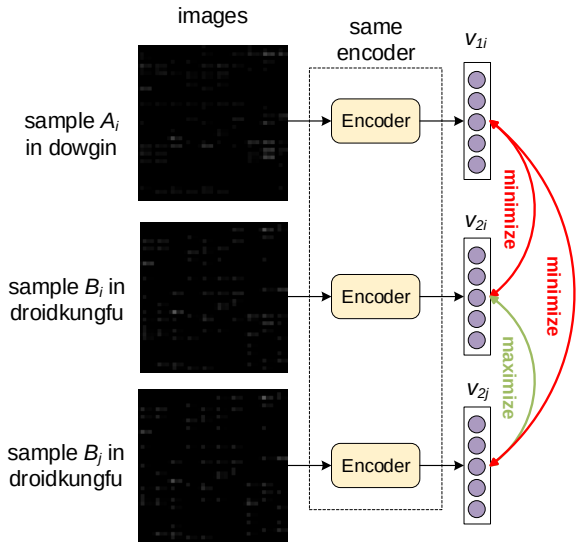

Fig. 4: Supervised contrastive learning in *IFDroid*, the goal is to maximize the similarity between samples in the same family and minimize the similarity between samples in different families.

Many studies have demonstrated the high effectiveness of self-supervised contrastive learning [10], [11], [39], but Khosla *et al.* [40] found that the label information of samples can be used to improve the accuracy of contrastive learning. Therefore, in this paper, we select supervised contrastive learning to train our encoder. In other words, our positive samples x^+ are selected from other malware samples in the same family, rather than being data augmentations of itself, as done in self-supervised contrastive learning. As shown in Figure 4, sample B_i and B_j are both from droidkungfu family while sample A_i is from dowgin family. Images of these three malware samples (*i.e.*, A_i, B_i, and B_j) are passed through an encoder to get their corresponding vector representations (*i.e.*, v_{1i}, v_{2i}, and v_{2j}). Then the goal of our contrastive learning is to maximize the similarity between positive samples (*i.e.*, (v_{2i}, v_{2j})) and minimize the similarity between negative samples (*i.e.*, (v_{1i}, v_{2i}) and (v_{1i}, v_{2j})).

Specifically, supervised contrastive learning (SupCon) calculates contrastive loss in batches. Given an input batch of data, SupCon first applies data augmentation twice to obtain two copies of the batch and then combines these two augmented batches to form a multi-viewed batch. For each anchor sample x_i with label l_i, the positive samples are all samples labeled l_i in the same batch and the negative samples are the remaining samples whose labels are different from l_i. Within a multi-viewed batch of data, let $i \in I \equiv \{1, ..., 2N\}$ be the index of an arbitrary augmented sample, the loss takes the following form.

$$\mathcal{L}_{out}^{sup} = \sum_{i \in I} \frac{-1}{|P(i)|} \sum_{p \in P(i)} log \frac{exp(v_i \cdot v_p / \tau)}{\sum_{a \in A(i)} exp(v_i \cdot v_a / \tau)} \quad (5)$$

Here, v_i represents the embedding of data x_i, $\tau \in \mathcal{R}^+$ is a scalar temperature parameter. $A(i) \equiv I/\{i\}$ and $P(i) \equiv \{p \in A(i) : l_p = l_i\}$ is the set of indices of all positives in the batch apart from i, $|P(i)|$ is its cardinality. The equation (5) uses multiple positive and negative samples per batch, and brings more information advantages to contrastive learning. It uses the positive normalization factor (*i.e.*, $\frac{1}{|P(i)|}$) to remove bias present in multiple positives samples and preserve the summation over negatives in the denominator to increase performance which has been shown in many studies [41], [42].

TABLE 1: Parameters used in our contrastive learning

Parameters	Settings
loss function	SupCon loss in [40]
temperature	0.07
optimizer	SGD
momentum	0.9
weight decay	0.0001
learning rate	0.05
batch size	64
epoch	100

After experimenting with certain widely-used neural networks, we finally choose ResNet-18 [43] as our image encoder since it can achieve a balance between accuracy and efficiency. In computer vision tasks, the input of common datasets is 224∗224 images with three channels. In our paper, the final image size is only 42∗42, and there is only one channel. To make ResNet-18 suitable for our classification task, we modify the fully connected layer at the input of ResNet-18 so that the dimension of the input is the same as the size of our image. At the same time, in the whole process of ResNet-18, we reduce the depth to one-third of the original to adapt to our malware classification task. Moreover, because contrastive learning requires the original sample, the corresponding positive samples, and negative samples when calculating loss, it is necessary to use batch normalization to normalize the output of different samples to ensure that the magnitude of the output of all samples is consistent. In other words, we use the normalized output to train our classifier. Table 1 shows the details of parameters used in our contrastive learning. The loss function is the same as in *SupCon* [40] namely *Supervised Contrastive Loss*. The whole procedure is trained using *Stochastic Gradient Descent* (SGD) with 0.9 momentum and 0.0001 weight decay. The output in this step is a learned encoder, that is, a learned ResNet-18. It can convert an image into a vector whose dimension is 512.

3.5 Classifier Training

In this step, we first use our learned encoder (*i.e.*, ResNet-18) to encode images into corresponding vectors, and then train a classifier (*i.e.*, a one-layer fully connected layer) by using these vectors and their labels. After training 100 epochs, the classifier will be selected as our final classifier. Parameters used in classifier training and contrastive learning are different only in loss function (*i.e.*, *Cross Entropy in classifier training*), and the others are the same.

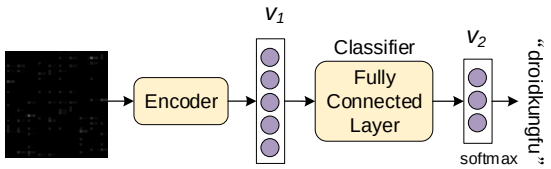

Fig. 5: Familial classification of *IFDroid*

3.6 Family Classification and Interpretation

After training phase, we can obtain a learned encoder and a trained classifier. They will be used to classify newly unlabeled malware samples. Specifically, given a new malware sample, we first perform static analysis to extract the function call graph. Then the graph is transformed into an image by centrality analysis. As shown in Figure 5, given an image, the encoder can encode it as a vector representation. Finally, the classifier takes the input of the vector and predicts the corresponding family.

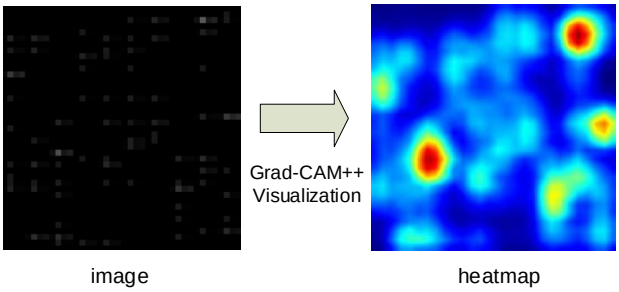

Fig. 6: The heatmap generated by Grad-CAM++ visualization technique to illustrate the classification result

Since we transform a function call graph into an image and leverage a CNN model to classify malware, we can apply visualization techniques to interpret the classification result. In practice, there have been strong efforts focusing on creating meaningful heatmaps that highlight the importance of individual pixel regions in an input image to its classification using a CNN. After trying several visualization techniques (*e.g.,* CAM [44], Grad-CAM [13], and Grad-CAM++ [14]), we find that Grad-CAM++ can achieve better interpretability. Particularly, it introduces pixel-wise weighting of the gradients of the output with respect to a particular spatial position in the final convolutional feature map of the CNN. In this way, it can provide a measure of the importance of each pixel in a feature map towards the overall decision of the CNN. Moreover, the visual explanations of Grad-CAM++ for any CNN-based network can be generated without changing the architecture or retraining. Therefore, we select it to interpret our classification result.

Figure 6 shows the visualization of a real-world malware sample. According to the intensity of the color in the heatmap, we can know which features are more effective in classifying this malware as this family. For example, a malware sample is classified as droidkungfu family. After applying Grad-CAM++ visualization technique on the image, we can obtain the corresponding heatmap as shown in Figure 6. The redder area in the heatmap indicates that this area contains more effective features. We can find the corresponding areas in the original image in reverse. Sensitive API calls contained in these areas have higher weights in identifying the malware as the family.

TABLE 2: Descriptions of metrics used in our experiments

Metrics	Abbr	Definition
True Positive	**TP**	#malware in family f are correctly classified into family f
True Negative	**TN**	#malware not in family f are correctly not classified into family f
False Positive	**FP**	#malware not in family f are incorrectly classified into family f
False Negative	**FN**	#malware in family f are incorrectly not classified into family f
True Positive Rate	**TPR**	TP/(TP+FN)
False Negative Rate	**FNR**	FN/(TP+FN)
True Negative Rate	**TNR**	TN/(TN+FP)
False Positive Rate	**FPR**	FP/(TN+FP)
Precision	**P**	TP/(TP+FP)
Recall	**R**	TP/(TP+FN)
F-measure	**F1**	2*P*R/(P+R)
Classification Accuracy	**CA**	percentage of malware which are correctly classified into their families

4 Experimental Evaluation

In this section, we aim to answer the following research questions:

- *RQ1: What is the effectiveness of IFDroid on classifying Android malware without and with obfuscations?*
- *RQ2: Can IFDroid interpret the familial classification results?*
- *RQ3: What is the runtime overhead of IFDroid on classifying Android malware?*

4.1 Datasets and Metrics

We first select one widely used ground truth dataset (*i.e.,* dataset-I [45]) as our experimental dataset to evaluate *IFDroid*. It is provided by Genome project [45] which consists of 1,247 malware samples. In fact, these malware samples were developed in 2011-2012, and are too old. To achieve more comprehensive evaluations, we construct another new larger dataset. Specifically, we choose the malware samples in AndroZoo [46] as our targets since most of them have been scanned by antivirus products in VirusTotal [47]. After collecting all scanning reports, these malware samples can be labeled into their corresponding families by using a technique namely *Euphony* [48]. To construct our new dataset-II, we randomly download 8,000 samples from 15 largest families and the number of samples in our new dataset is 120,000. Note that AndroZoo is one of the largest Android app collections which contains more than 10 million Android apps. *Euphony* is a tool to label the family of a malware sample by analyzing the VirusTotal reports and has been used by many other researchers [49], [50].

To evaluate *IFDroid*, we conduct experiments by performing ten-fold cross-validations. In other words, we first divide our dataset into ten subsets, then nine of them are selected as training sets and the last subset is used to test.

We repeat this ten times and report the average classification results. Furthermore, to measure the effectiveness of *IFDroid*, we leverage certain widely used metrics (Table 2) to present the classification results. We run our experiments on a desktop equipped with a 32-core 2.30GHz CPU, a GTX 1080 GPU, and 128GB of RAM.

TABLE 3: Classification accuracy of *IFDroid* and six state-of-the-art comparative systems on dataset-I [45]

Baseline Approach	Classification Accuracy
Dendroid [15]	0.942
Apposcopy [16]	0.900
DroidSIFT [17]	0.930
MudFlow [18]	0.881
DroidLegacy [19]	0.929
Astroid [20]	0.938
IFDroid	**0.984**

4.2 RQ1: Effectiveness

4.2.1 *Effectiveness on Classifying General Malware*

First, we conduct evaluations to check the ability of *IFDroid* on classifying general Android malware. We first use dataset-I to present the comparative results of *IFDroid* and six state-of-the-art related systems. These systems include: 1) *Dendroid* [15] applies text mining techniques to analyze the code structures of Android malware and classify them into corresponding families; 2) *Apposcopy* [16] performs program analysis to extract both data-flow and control-flow information of malware samples to classify them; 3) *DroidSIFT* [17] pays attention to constructing API dependency graph by analyzing the program semantics to classify Android malware; 4) *MudFlow* [18] extracts the source-and-sink pairs of malware samples and regards them as features to classify malware; 5) *DroidLegacy* [19] conducts app partition to divide the malware sample into sub-modules and labels the corresponding family by analyzing the malicious sub-module; and 6) *Astroid* [20] synthesizes a maximally suspicious common subgraph of each malware family as a signature to classify malware.

Because most of these systems are not publicly available and the dataset used to produce evaluation results are all the same (*i.e.*, dataset-I [45]), we directly adopt the classification accuracy of these six systems in their papers [15], [16], [17], [18], [19], [20]. Through results in Table 3, we see that *IFDroid* is superior to other comparative systems. This happens because *IFDroid* not only considers the program semantics of malware samples but also extracts effective features by a learned robust encoder.

To achieve more comprehensive evaluations, we use our new dataset-II to compare *IFDroid* with four recent state-of-the-art Android malware familial classification methods. They include: 1) *FalDroid* [8] analyzes frequent subgraphs to represent the common behaviors of each malware family and uses them to perform familial classification; 2) *AOM* [21] uses Android-oriented metrics to identify Android malware families; 3) *MVIIDroid* [22] uses a multiple view information integration approach for Android malware detection and family identification; and 4) *CDFG* [23] combines control flow graph with data flow graph to accomplish Android malware family classification. To further examine the contribution of contrastive learning to general malware classification, we implement another system that does not use contrastive learning to learn an encoder first but directly trains the encoder and classifier together in training phase. We name this system *IFDroid* (*wo*) because it differs from *IFDroid* in that it is trained *wi*th*o*ut using contrastive learning.

TABLE 4: Familial classification results (F1) of *FalDroid* (*Fal* for short), *AOM*, *MVIIDroid* (*MVI* for short), *CDFG*, and *IFDroid* on dataset-II. *wo* and *wi* denote *IFDroid* without and with contrastive learning, respectively.

Family	*Fal*	*AOM*	*MVI*	*CDFG*	*wo*	*wi*
adwo	90.5	83.2	85.3	91.6	91.7	**93.5**
airpush	76.3	64.9	70.1	82.3	78.5	**92.5**
dowgin	95.2	84.5	89.1	96.1	97.1	**97.5**
droidkungfu	94.7	80.4	89.1	96.2	98.7	**99.5**
feiwo	94.8	91.3	93	**95.2**	93.1	94.4
gingermaster	92.4	83.9	90.8	93.5	95.1	**97.3**
kuguo	92.5	89.3	91.1	93.6	93.1	**94.9**
leadbolt	95.2	93.3	94.1	96.7	98.8	**99.4**
plankton	99.1	91.4	95.2	98.6	98.2	**99.1**
startapp	90.4	87.2	90.3	95.7	99.2	**100**
umeng	90.3	82.5	87.1	92.1	90.3	**93.2**
utchi	100	100	100	100	100	**100**
waps	93.3	91.3	93.9	96.2	95.8	**96.5**
wooboo	92.4	88.3	90.2	97.3	97.1	**98.2**
youmi	78.4	73.5	77.1	83.6	85.9	**88.7**

TABLE 5: Descriptions of 12 obfuscators used in our experiments

Obfuscators	Descriptions
ClassRename	Change the package name and rename classes
FieldRename	Rename fields
MethodRename	Rename methods
ConstStringEncryption	Encrypt constant strings in code
AssetEncryption	Encrypt asset files
LibEncryption	Encrypt native libs
ResStringEncryption	Encrypt strings in resources
ArithmeticBranch	Insert junk code that is composed by arithmetic computations and a branch instruction
CallIndirection	Modify the control-flow graph without changing the code semantics
Goto	Modify the control-flow graph by adding two new nodes
Nop	Insert random nop instructions within every method implementation
Reorder	Change the order of basic blocks of the control-flow graph

The detailed classification results of 15 families are shown in Table 4. Through the results, we observe that *IFDroid* performs better than *FalDroid*, *AOM*, *MVIIDroid*, and *CDFG* on classifying most malware families. For example, the F1 values of *FalDroid*, *AOM*, *MVIIDroid*, and *CDFG* on classifying "startapp" family are 90.4%, 87.2%, 90.3%, and 95.7%, respectively. However, for *IFDroid*, it has the ability to classify them into "startapp" family without any inaccuracies. Additionally, when we use contrastive learning to learn our encoder first, the classification performance is always better than without contrastive learning. For example, when classifying malware samples with "airpush" family, the F1 value of *IFDroid* (*wo*) is only 78.5% while *IFDroid* with contrastive learning can maintain 92.5% F1. Such results suggest that the use of contrastive learning can indeed enhance the

TABLE 6: F1 values of *FalDroid* (*Fal* for short), *AOM*, *MVIIDroid* (*MVI* for short), *CDFG*, and *IFDroid* on classifying obfuscated malware. *wo* and *wi* denote *IFDroid* without and with contrastive learning, respectively.

Obfuscators		#Samples	*Fal*	*AOM*	*MVI*	*CDFG*	*wo*	*wi*
Rename	ClassRename	5,049	100.0	99.1	99.7	100.0	100.0	**100.0**
	FieldRename	5,297	100.0	98.4	99.8	100.0	100.0	**100.0**
	MethodRename	5,271	100.0	98.6	99.6	100.0	100.0	**100.0**
Encryption	AssetEncryption	5,477	93.8	82.1	91.2	96.5	98.1	**100.0**
	ConstStringEncryption	5,443	84.7	71.4	79.3	88.3	90.2	**96.9**
	LibEncryption	5,391	94.5	87.1	91.9	96.8	96.9	**100.0**
	ResStringEncryption	5,366	95.1	91.4	93.6	99.1	98.0	**100.0**
Code	ArithmeticBranch	5,649	97.1	82.4	87.1	97.5	97.3	**100.0**
	CallIndirection	5,571	73.2	62.4	69.3	77.1	77.3	**91.1**
	Goto	5,546	92.3	82.1	90.4	92.6	94.8	**100.0**
	Nop	5,529	95.8	90.2	90.8	95.6	97.7	**100.0**
	Recorder	5,443	96.0	86.3	92.3	96.1	95.5	**100.0**
Apply 12 obfuscators		4,389	69.2	53.5	65.3	72.6	71.8	**90.4**
ALL		69,421	91.9	83.7	88.7	93.5	93.9	**98.4**

ability of *IFDroid* on classifying malware.

4.2.2 *Effectiveness on Classifying Obfuscated Malware*

Next, we evaluate the effectiveness of *IFDroid* on classifying obfuscated Android malware. For this purpose, we use an automatic Android apps obfuscation tool (*Obfuscapk* [51]) that provides certain obfuscators including typical obfuscations (*e.g.*, class rename and method rename) and some advanced code obfuscations (*e.g.*, call indirection and goto). Specifically, we first learn an encoder and train a classifier by using samples in dataset-II. After completing the training phase, we randomly select 400 malware samples from each family, and the final number of selected samples is 400∗15=6,000.

We select a total of 12 different obfuscators provided by *Obfuscapk*, including three rename obfuscators, four encryption obfuscators, and five advanced code obfuscators. Descriptions of these obfuscators are presented in Table 5. In practice, *Obfuscapk* can not obfuscate some of our malware samples due to certain errors. But fortunately, the failed samples only occupy a small part. To further evaluate the effectiveness of *IFDroid*, we apply 12 obfuscators together to generate more complex obfuscated malware. Finally, we obtain 69,421 obfuscated samples in total. We also conduct comparative evaluations with *FalDroid*, *AOM*, *MVIIDroid*, and *CDFG* on classifying these obfuscated malware. The comparative results are shown in Table 6, which includes the F1 values of *IFDroid* and comparative methods (*i.e.*, *FalDroid*, *AOM*, *MVIIDroid*, and *CDFG*).

Since the typical rename obfuscations (*i.e.*, class rename, method rename, and field rename) do not change the call relationships between functions in an app. Both *FalDroid*, *CDFG*, and *IFDroid* can correctly classify all obfuscated apps into corresponding families. Furthermore, no matter what kind of obfuscation it is for, *IFDroid* can perform better when using contrastive learning. However, the F1 values of *IFDroid* without contrastive learning is only 77.3% when we classify apps that are obfuscated by *CallIndirection*. After our in-depth analysis, we find that the number of nodes and edges in a function call graph change a lot after applying *CallIndirection*. For example, a sample originally has 8,135 nodes and 19,725 edges. After it is obfuscated by *CallIndirection*, the number of nodes becomes 35,396, and the number of edges increases to 58,407. This huge change makes *IFDroid* make mistakes in classification. However, when we adopt contrastive learning, the true positive rate of *IFDroid* can be increased from 77.3% to 91.1%. This result also shows that the encoder learned by contrastive learning can extract more robust features to classify malware.

Moreover, when classifying samples that are obfuscated by 12 obfuscators together, *IFDroid* can also achieve an F1 of 90.4%. However, the F1 values of *FalDroid*, *AOM*, *MVIIDroid*, and *CDFG* are only 67.9%, 50.1%, 61.6%, and 69.3%, respectively. On average, *IFDroid* maintains an F1 of 98.4% on classifying all generated obfuscated samples. Such result suggests that *IFDroid* is more robust than *FalDroid*, *AOM*, *MVIIDroid*, and *CDFG*.

Summary: *IFDroid achieves higher accuracy than Dendroid, Apposcopy, DroidSIFT, MudFlow, DroidLegacy, Astriod, FalDroid, AOM, MVIIDroid, and CDFG on Android malware classification. Using contrastive learning can not only improve the accuracy but also enhance the robustness of IFDroid.*

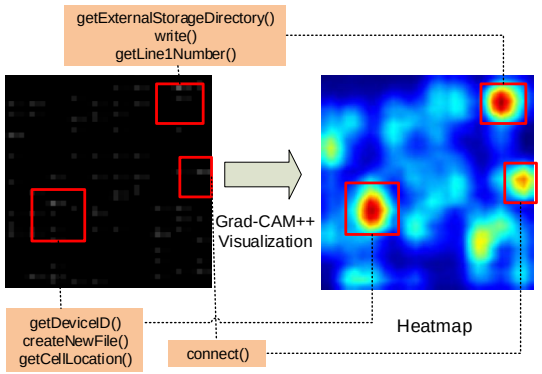

Fig. 7: A real-world malware sample's visualization of classification result

4.3 RQ2: Interpretability

As aforementioned, since *Gradient-weighted Class Activation Mapping++* (Grad-CAM++) [13], [14] performs better and can generate visual explanations for any CNN-based network without changing the architecture or retraining, we use it as our visualization technique to interpret our classification results. These interpretations can help security analysts understand why a malware sample is classified as

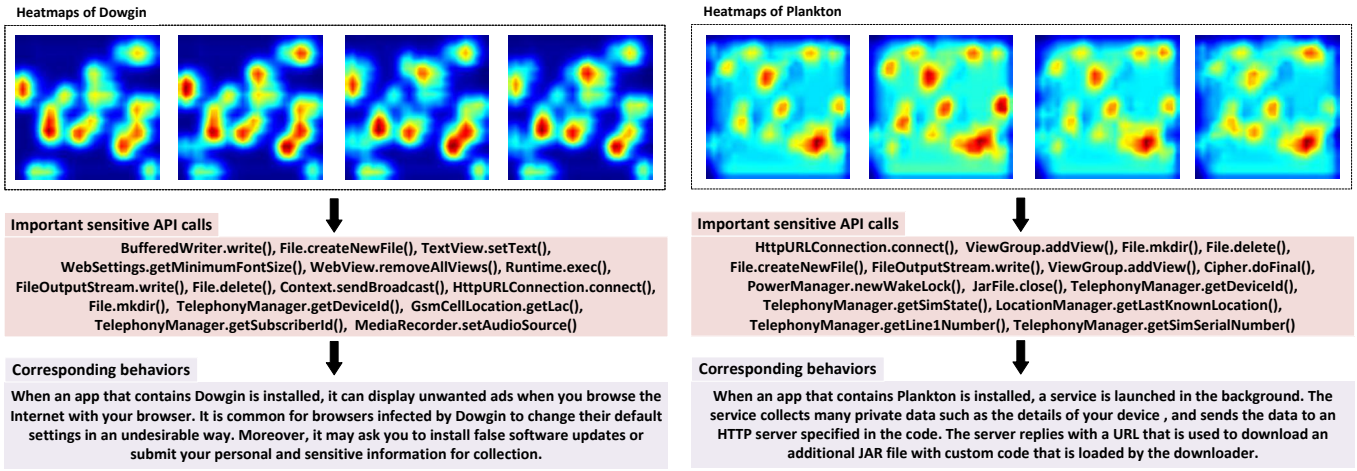

Fig. 8: The most important sensitive API calls of two families obtained by analyzing the heatmaps

this family. Specifically, we use Grad-CAM++ to obtain the corresponding heatmaps of malware samples. The red area in the heatmap indicates that features in this area play more important roles in classifying as this family. In other words, sensitive API calls contained in these areas have higher weights in identifying the malware as the family.

Figure 7 shows the visualization of a real-world malware sample. This sample is correctly classified into droidkungfu family by *IFDroid*. After applying Grad-CAM++ visualization technique on the generated image, we can obtain the corresponding heatmap as shown in Figure 7. The redder the color in a heatmap, the more valuable the features in the area. Therefore, we pay more attention to these red areas. After completing the one-to-one correspondence of three red areas from the original image, we can collect seven sensitive API calls (*i.e.*, *getExternalStorageDirectory()*, *write()*, *getLine1Number()*, *getDeviceID()*, *createNewFile()*, *getCellLocation()*, and *connect()*) from these areas. Through the result, we can see that this malware collects users' private data and write them into created files. These files are then sent to the network or saved on other external devices. Because of this behavior, it is classified into "droidkungfu" family.

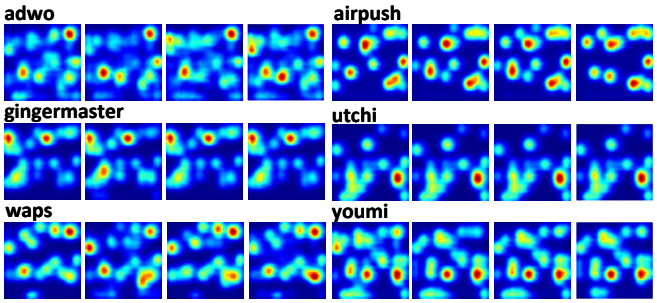

Fig. 9: Visualization of six malware familial classification results, four malware samples are displayed for each family.

To further analyze the interpretability of *IFDroid*, we collect the most important sensitive API calls of each family by analyzing their heatmaps. Due to space limitations, we only show the details of two families (*i.e.*, "dowgin" and "plankton") in Figure 8. From the results in Figure 8, we see that the "dowgin" family behaves by displaying unwanted or malicious advertisements and changing web settings in undesirable ways. For malware of the "plankton" family, once a user installs it, it can collect a lot of sensitive data such as device ID and send them to a remote server. Meanwhile, the server replies with a URL that allows the infected phone to download and install a JAR file containing a dynamic payload. Such payload can have a huge impact on users.

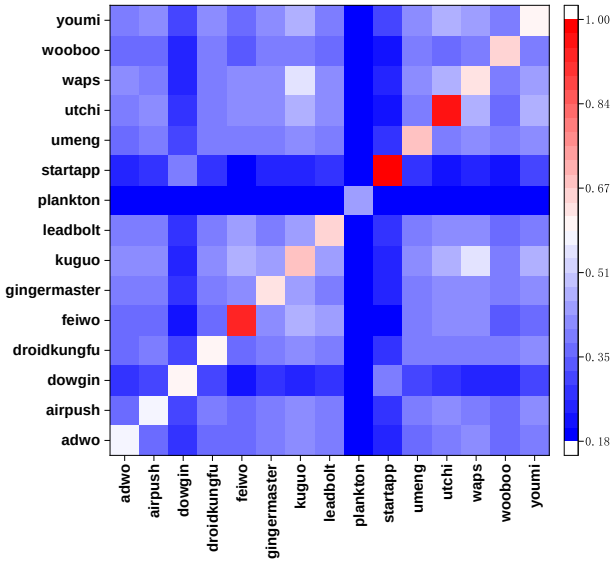

Fig. 10: The average similarity between heatmaps in 15 families

Finally, we conduct a survey to explore the effectiveness of the interpretation of family classification by heatmaps. Due to the limited space, we randomly present six families' heatmaps in Figure 9, and four malware samples are displayed for each family. Given heatmaps of all malware samples, we compute the similarity of two heatmaps by using *Structural SIMilarity* (SSIM) [52] technique one by one. SSIM is widely used to measure the similarity of two images.

After analyzing all heatmaps, the average similarity between different families is calculated and presented in

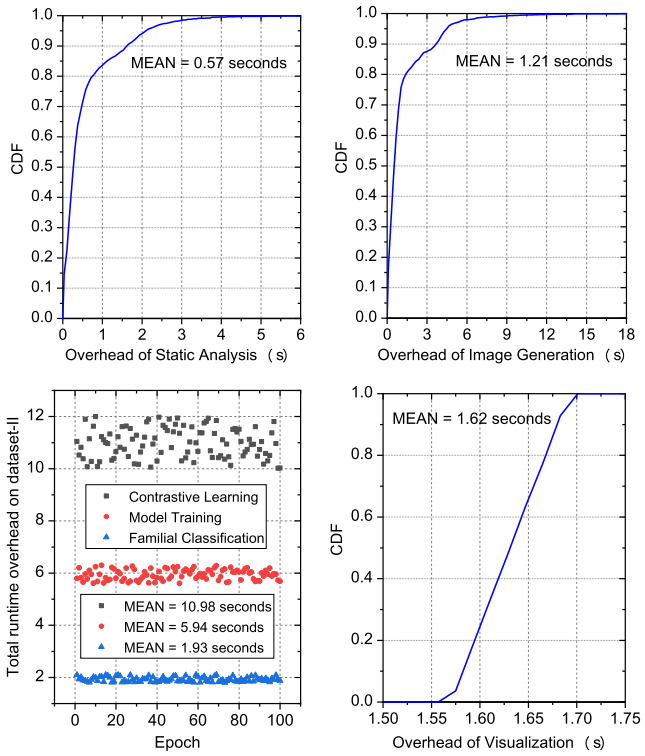

Fig. 11: *Cumulative Distribution Function* (CDF) of runtime overheads of *IFDroid* on different steps (seconds)

Figure 10. From Figure 9 and Figure 10, we observe several phenomenons. The first phenomenon is that the heatmaps of most malware in the same family are similar. It is reasonable because malware samples of certain families are often just repackaged applications with slight modifications [7]. In other words, most malware samples in the same family are polymorphic variants of other malware samples in this family. Therefore, malware samples in the same family always perform similar malicious actions, resulting in similar heatmaps generated by visualization. The second phenomenon is that the heatmaps of malware in different families are basically different. This result is in line with expectations since malware samples in different families exhibit different malicious behaviors. Because of this, they are classified into different malware families.

Summary: *IFDroid can interpret the familial classification results by using visualization techniques. We can even distinguish the malware families directly through the heatmaps since the heatmaps of most malware in the same family are similar and heatmaps of malware in different families are basically different.*

4.4 RQ3: Efficiency

In this step, we aim to study the runtime overhead of *IFDroid*. To this end, we randomly select 8,000 samples from dataset-II as our test target. Given a new malware sample, *IFDroid* performs four steps to classify it into the corresponding family and interpret the classification result.

1) Static Analysis. The first step of *IFDroid* is to distill the program semantics of a sample into a function call graph based on static analysis. Figure 11 and Table 7 show the runtime overhead of static analysis on dataset-II, for more than 80% samples we can extract the graphs in one second. On average, it takes about 0.57 seconds for *IFDroid* to complete static analysis on dataset-II.

2) Image Generation. The second step of *IFDroid* is to transform the call graph into an image to avoid high-cost graph matching. Specifically, we extract four different centralities (*i.e.,* degree centrality, katz centrality, closeness centrality, and harmonic centrality) of sensitive API calls within the graph to construct the image. As shown in Figure 11 and Table 7, *IFDroid* requires about 1.21 seconds on average to analyze the graph and complete the image generation step.

3) Familial Classification. The third step of *IFDroid* is to classify the image into its corresponding family. We first perform contrastive learning to learn an encoder and then train a classifier on generated images. The total runtime overhead of contrastive learning and model training on these images are shown in Figure 11. On average, it takes about 10.98 seconds and 5.94 seconds to finish a round of contrastive learning and model training, respectively. After completing the training phase, we leverage the learned encoder and the trained classifier to complete the familial classification step of *IFDroid*. As shown in Figure 11 and Table 7, this step is the fastest step among all step in *IFDroid*. It consumes about 1.93 seconds to classify all images (*i.e.,* 8,000 images) into corresponding families.

4) Interpretation. The final step of *IFDroid* is to interpret the classification results of 8,000 images. To this end, we use Grad-CAM++ visualization technique to obtain the corresponding heatmaps of these images. This step is the most time-consuming, it requires 1.62 seconds on average to accomplish the visualization of an image.

TABLE 7: Runtime overhead of different steps of *IFDroid* on 8,000 malware samples

Different Steps	Total runtime	Average runtime
Static Analysis	4,792	0.57
Image Generation	10,172	1.21
Familial Classification	1.93	0.00023
Interpretation	13,619	1.62
ALL	28,585	1.78+1.62=3.4

In general, given a new malware sample, *IFDroid* consumes about 1.78 seconds to complete the classification and 1.62 seconds to interpret the classification result. As for *FalDroid*, *AOM*, *MVIIDroid*, and *CDFG*, they need to take about 11.5 seconds, 9.2 seconds, 12.4 seconds, and 26.9 seconds to complete the classification of a malware sample. In other words, if only from the overhead caused by classification, *IFDroid* is about six times, five times, seven times, and 15 times faster than *FalDroid*, *AOM*, *MVIIDroid*, and *CDFG*. Such high efficiency indicates that *IFDroid* can achieve large-scale Android malware classification and interpretation.

Summary: *On average, IFDroid requires about 1.78 seconds to complete the classification and 1.62 seconds to interpret the classification result of a malware sample.*

5 Discussions

5.1 Differences from *MalScan*

In our previous work (*i.e., MalScan* [12]), we use centrality analysis to accomplish scalable Android malware detection.

The goal of *MalScan* is to distinguish malware samples from benign apps while *IFDroid* aims to classify malware into corresponding families. In reality, we also apply *MalScan* to classify our dataset-II, but the result is not ideal. The classification accuracy on 15 families is only 85.2%. In other words, *MalScan* is not suitable for Android malware familial classification. To address the issue, we develop *IFDroid* which is a robust and interpretable Android malware familial classification system.

5.2 Why is IFDroid obfuscation-resilient?

The reasons are mainly three-fold. First, *IFDroid* uses sensitive API calls to form the features and API calls are not obfuscated. Second, *IFDroid* applies centrality analysis to maintain the graph details which is robust against obfuscations. Last and most important, the learned encoder by contrastive learning in *IFDroid* can extract robust features from generated images. The goal of contrastive learning is to maximize the agreement between positive data and minimize the agreement between negative data. Actually, the obfuscated malware can be regarded as one positive sample of the original malware since the inherent program semantics do not change after obfuscations. Therefore, when we use contrastive learning to learn the encoder, it can enlarge the similarity between obfuscated malware and original malware, making it possible to classify the obfuscated malware into its correct family.

5.3 The Reasons of Image Generation

The reasons are mainly three-fold. First, CNN can support large-scale image analysis. If we can transform a malware sample into an image, then we can use the CNN model to achieve large-scale malware analysis. Second, convolution kernels of different sizes in CNN can automatically extract features from images. In our generated image, each pixel represents a certain centrality value for a sensitive API call. When different sizes of convolution kernels are used, different centrality values of different sensitive API calls can be combined to find the most suitable combined features for malware classification. Third, CNN can be interpreted by some visualization techniques. If we can employ some suitable visualization techniques, then we can interpret the results of malware family classification, making it clear to the security researcher why the sample is classified into this family.

5.4 The Selection of Static Analysis

Compared with dynamic analysis, static analysis does have some shortcomings. However, since different events need to be generated to trigger different behaviors, it is very expensive to dynamically analyze an app. Sometimes it may take hours to analyze an app. Such a huge overhead makes it unsuitable for large-scale malware analysis. In the real world, according to the AV-TEST Institute report [24], an average of about 9,000 Android malware samples were detected every day in 2021. If we use dynamic analysis to analyze the families of these malware samples, it is difficult to achieve daily malware scanning. But the average runtime overhead of our proposed method *IFDroid* is only 3.4 seconds. In other words, if we analyze 16 malware samples at a time, we only need about half an hour to complete all the analysis and interpretation of detection results. Moreover, according to the results in our paper, *IFDroid* can achieve ideal performance in analyzing obfuscated malware due to the use of contrastive learning.

5.5 Limitations

5.5.1 Call Graph

To achieve efficient static analysis, we leverage Androguard [25] to extract the function call graph of a malware sample. This graph is a context- and flow-insensitive call graph. Moreover, malware samples can use reflection [53] to call sensitive API calls, in this case, we may miss the call relationships between these methods. As shown in Table 6, the *CallIndirection* obfuscator significantly lowers the scores of all classifiers. The F1 of *IFDroid* with contrastive learning is only 91.1% which is not good enough to distinguish these obfuscated malware. In our future work, we plan to use advanced program analysis [54] to generate more accurate call graphs to maintain better robustness against obfuscation.

5.5.2 Sensitive API calls

Sensitive API calls used in *IFDroid* consist of 426 API calls that are highly correlated with malicious operations [37]. They occupy a small part of the whole sensitive API calls. We plan to conduct statistical analysis to select more valuable sensitive API calls and use them to generate our images.

As API calls invoked by Android malware may evolve over time, *IFDroid* may suffer from the issue. To resist false positives and false negatives caused by the evolution of Android malware, we will update our sensitive API calls in real-time and adopt the technique in [55] to improve our classification effectiveness.

5.5.3 Image Size

After extracting four centrality values of 426 sensitive API calls, we can obtain a $426*4$ vector representation. To make our interpretation more intuitive, we crop the vector and turn it into a square image. Specifically, we add 60 zeros at the end and then reshape it as a $42*42$ (*i.e.*, $426*4+60=42*42$) vector. Although every pixel in our image is meaningful, the centrality values of the same API call may be segmented adjacent to the beginning and end of different lines. Such case may affect the learning effectiveness of our model. In our future work, we plan to select different image size to commence our experiments. By this, we can find a more suitable size and achieve more effective results. In addition, we will also build a separate image for each centrality and combine the four images to analyze Android malware.

5.5.4 Model

In *MalScan*, we have shown that centrality is not robust enough against tailored adversarial attacks. In other words, an attacker might adjust the frequency of some sensitive API calls by adding some dead code, so that the model incorrectly classifies samples into another family. However, we choose four different centralities, each of which can

represent a kind of structural information of the call graph. For example, degree centrality considers the degree of all nodes within a network, while closeness centrality analyzes the average shortest distance of all nodes. Therefore, when an adversary attacks our model, he may consider four different centrality extraction algorithms to craft adversarial examples, which may result in a large attack overhead. In the future, we plan to choose more different centralities to preserve more graph details of the call graph, making it more robust against adversarial attacks.

When faced with a new family of malware samples, we cannot use the original model to classify them since the model has not seen the family. However, instead of retraining the model, we can use a single-task continuous learning [56] approach to mitigate the limitation. Based on the original model, we only need to use a small number of new family samples to iterate the model in some steps, and then the model can flag the new family.

6 Related Work

6.1 Malware Familial Classification

Recently, many studies [15], [16], [17], [18], [19], [20], [8], [57], [58], [59], [60], [22], [21], [23], [61], [62] have been proposed to classify malware samples into corresponding families. For example, *Dendroid* [15] applies text mining techniques to analyze the code structures of Android malware and classify them into corresponding families. *Apposcopy* [16] considers both data-flow and control-flow information of malware samples to classify them by performing heavy-weight program analysis. *MudFlow* [18] extracts the source-and-sink pairs of malware samples and regards them as features to classify malware. *FalDroid* [8] conducts frequent subgraph analysis to extract common subgraphs of each family and uses them to perform familial classification. *DroidSIFT* [17] extracts the weighted contextual API dependency graph to solve the malware deformation problem based on static analysis. These proposed approaches consider different program information to achieve accurate malware classification. However, heavy-weight program analysis results in low scalability, making them can not scale to large numbers of malware analysis. Moreover, most of them only provide the corresponding labels (*i.e.,* families) to users and can not interpret the classification results.

6.2 Contrastive Learning

Contrastive learning was first introduced by Mikolov *et al.* [63] in 2013 for *natural language processing* (NLP). In recent years, it has been more and more popular on different NLP tasks such as text representation learning [10], language understanding [11], and cross-lingual pre-training [39]. In practice, it has also been used in other domains. For example, Dai *et al.* [64] propose a new method for image caption through contrastive learning. *SimCLR* [65] is a simple framework to use contrastive learning on image classification. Compared with previous work, the accuracy of *SimCLR* is improved by 7%. *COLA* [66] is a self-supervised pre-training approach for learning a general-purpose representation of audio and *CVRL* [67] uses contrastive learning to learn spatiotemporal visual representations from unlabeled videos. The use of contrastive learning in most of the previous studies is self-supervised, however, Khosla *et al.* [40] find that the label information of training dataset can improve the performance of the learned encoder. Therefore, in this paper, to achieve better classification accuracy, we leverage supervised contrastive learning to conduct Android malware familial classification.

7 Conclusion

In this paper, we propose to use contrastive learning to resist code obfuscations of Android malware. To demonstrate the effectiveness of contrastive learning, we implement an obfuscation-resilient system (*i.e., IFDroid*) and the extensive evaluation results show that *IFDroid* is superior to ten state-of-the-art Android malware classification systems (*i.e., Dendroid* [15], *Apposcopy* [16], *DroidSIFT* [17], *MudFlow* [18], *DroidLegacy* [19], *Astroid* [20], *FalDroid* [8], *AOM* [21], *MVI-IDroid* [22], and *CDFG* [23]). Moreover, when analyzing 69,421 obfuscated malware samples, *IFDroid* can achieve a 98.4% F1 score. Such result also suggests that *IFDroid* can achieve obfuscation-resilient malware analysis.

Acknowledgments

This work is supported by the Key Program of National Science Foundation of China under Grant No. U1936211 and Hubei Province Key R&D Technology Special Innovation Project under Grant No. 2021BAA032.

References

[1] "Smartphone market share," https://www.idc.com/promo/smartphone-market-share/os, 2020.
[2] "It threat evolution q2 2020. mobile statistics," https://securelist.com/it-threat-evolution-q2-2020-mobile-statistics/98337/, 2020.
[3] M. Hammad, J. Garcia, and S. Malek, "A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products," in *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, 2018, pp. 421–431.
[4] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding android obfuscation techniques: A large-scale investigation in the wild," in *Proceedings of the 2018 International Conference on Security and Privacy in Communication Systems (ICSPCS'18)*, 2018, pp. 172–192.
[5] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *Proceedings of the 2012 ACM conference on Computer and communications security (CCS'12)*, 2012, pp. 241–252.
[6] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 11, pp. 1869–1882, 2014.
[7] 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 *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014, pp. 1–15.
[8] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 1890–1905, 2018.
[9] M. Fan, X. Luo, J. Liu, M. Wang, C. Nong, Q. Zheng, and T. Liu, "Graph embedding based familial analysis of android malware using unsupervised learning," in *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*, 2019, pp. 771–782.

[10] J. M. Giorgi, O. Nitski, G. D. Bader, and B. Wang, "Declutr: Deep contrastive learning for unsupervised textual representations," *arXiv preprint arXiv:2006.03659*, 2020.

[11] H. Fang and P. Xie, "Cert: Contrastive self-supervised learning for language understanding," *arXiv preprint arXiv:2005.12766*, 2020.

[12] Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang, and H. Jin, "Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*, 2019, pp. 139–150.

[13] 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," in *Proceedings of the 2017 IEEE International Conference on Computer Vision (ICCV'17)*, 2017, pp. 618–626.

[14] A. Chattopadhay, A. Sarkar, P. Howlader, and V. N. Balasubramanian, "Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks," in *Proceedings of the 2018 IEEE Winter Conference on Applications of Computer Vision (WACV'18)*, 2018, pp. 839–847.

[15] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families," *Expert Systems with Applications*, vol. 41, no. 4, pp. 1104–1117, 2014.

[16] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*, 2014, pp. 576–587.

[17] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, 2014, pp. 1105–1116.

[18] 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 2015 IEEE International Conference on Software Engineering (ICSE'15)*, 2015, pp. 426–436.

[19] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: Automated familial classification of android malware," in *Proceedings of the 2014 ACM SIGPLAN on Program Protection and Reverse Engineering Workshop (PPREW'14)*, 2014, pp. 1–12.

[20] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand, "Automated synthesis of semantic malware signatures using maximum satisfiability," *arXiv preprint arXiv:1608.06254*, 2016.

[21] W. Blanc, L. G. Hashem, K. O. Elish, and M. H. Almohri, "Identifying android malware families using android-oriented metrics," in *Proceedings of the 2019 IEEE International Conference on Big Data (ICBD'19)*, 2019, pp. 4708–4713.

[22] Q. Wu, M. Li, X. Zhu, and B. Liu, "Mviidroid: A multiple view information integration approach for android malware detection and family identification," *IEEE MultiMedia*, vol. 27, no. 4, pp. 48–57, 2020.

[23] X. Zhiwu, K. Ren, and F. Song, "Android malware family classification and characterization using cfg and dfg," in *Proceedings of the 2019 International Symposium on Theoretical Aspects of Software Engineering (TASE'19)*, 2019, pp. 49–56.

[24] "Developmemt of android malware," https://www.av-test.org/en/statistics/malware/, 2022.

[25] A. Desnos *et al.*, "Androguard," https://github.com/androguard/androguard, 2011.

[26] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: Visualization and automatic classification," in *Proceedings of the 8th International Symposium on Visualization for Cyber Security (ISVCS'11)*, 2011, pp. 1–7.

[27] A. Makandar and A. Patrot, "Malware class recognition using image processing techniques," in *Proceedings of the 2017 International Conference on Data Management, Analytics and Innovation (ICDMAI'17)*, 2017, pp. 76–80.

[28] L. Liu and B. Wang, "Malware classification using gray-scale images and ensemble learning," in *Proceedings of the 2016 International Conference on Systems and Informatics (ICSAI'16)*, 2016, pp. 1018–1022.

[29] X. Xiao, "An image-inspired and cnn-based android malware detection approach," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*, 2019, pp. 1259–1261.

[30] H. Jeong, S. Mason, A. Barabási, and Z. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, no. 6833, pp. 41–42, 2001.

[31] X. Liu, J. Bollen, M. Nelson, and H. Van de Sompel, "Co-authorship networks in the digital library research community," *Information Processing & Management*, vol. 41, no. 6, pp. 1462–1480, 2005.

[32] R. Guimera, S. Mossa, A. Turtschi, and L. N. Amaral, "The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles," *the National Academy of Sciences*, vol. 102, no. 22, pp. 7794–7799, 2005.

[33] N. Coles, "It's not what you know—it's who you know that counts. analysing serious crime groups as social networks," *British Journal of Criminology*, vol. 41, no. 4, pp. 580–594, 2001.

[34] L. Freeman, "Centrality in social networks conceptual clarification," *Social Networks*, vol. 1, no. 3, pp. 215–239, 1978.

[35] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953.

[36] M. Marchiori and V. Latora, "Harmony in the small-world," *Physica A: Statistical Mechanics and its Applications*, vol. 285, no. 3-4, pp. 539–546, 2000.

[37] L. Gong, Z. Li, F. Qian, Z. Zhang, and Y. Liu, "Experiences of landing machine learning onto market-scale mobile malware detection," in *Proceedings of the 15th European Conference on Computer Systems (EuroSys'20)*, 2020, pp. 1–14.

[38] A. Anand, "Contrastive self-supervised learning," https://ankeshanand.com/blog/2020/01/26/contrative-self-supervised-learning.html, 2020.

[39] Z. Chi, L. Dong, F. Wei, N. Yang, S. Singhal, W. Wang, X. Song, X.-L. Mao, H. Huang, and M. Zhou, "Infoxlm: An information-theoretic framework for cross-lingual language model pre-training," *arXiv preprint arXiv:2007.07834*, 2020.

[40] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, "Supervised contrastive learning," *arXiv preprint arXiv:2004.11362*, 2020.

[41] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, "Momentum contrast for unsupervised visual representation learning," in *Proceedings of the 2020 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'20)*, 2020, pp. 9729–9738.

[42] O. Henaff, "Data-efficient image recognition with contrastive predictive coding," in *Proceedings of the 2020 International Conference on Machine Learning (ICML'20)*, 2020, pp. 4182–4192.

[43] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*, 2016, pp. 770–778.

[44] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*, 2016, pp. 2921–2929.

[45] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P'12)*, 2012, pp. 95–109.

[46] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th Working Conference on Mining Software Repositories (MSR'16)*, 2016, pp. 468–471.

[47] "Free online virus, malware and url scanner," www.virustotal.com/, 2022.

[48] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. L. Traon, J. Klein, and L. Cavallaro, "Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware," in *Proceedings of the 14th International Conference on Mining Software Repositories*, 2017, pp. 425–435.

[49] S. Zhu, J. Shi, L. Yang, B. Qin, Z. Zhang, L. Song, and G. Wang, "Measuring and modeling the label dynamics of online anti-malware engines," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, 2020, pp. 2361–2378.

[50] Y. Zhang, Y. Sui, S. Pan, Z. Zheng, B. Ning, I. Tsang, and W. Zhou, "Familial clustering for weakly-labeled android malware using hybrid representation learning," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3401–3414, 2019.

[51] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo, "Obfuscapk: An open-source black-box obfuscation tool for android apps," *SoftwareX*, vol. 11, p. 100403, 2020.

[52] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity,"

IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600–612, 2004.
[53] 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*, vol. 9, no. 1, pp. 99–108, 2013.
[54] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*, 2016, pp. 318–329.
[55] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, "Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS'20)*, 2020, pp. 757–770.
[56] R. Hadsell, D. Rao, A. A. Rusu, and R. Pascanu, "Embracing change: Continual learning in deep neural networks," *Trends in Cognitive Sciences*, vol. 24, no. 12, pp. 1028–1040, 2020.
[57] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, "Semantic modelling of android malware for effective malware comprehension, detection, and classification," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*, 2016, pp. 306–317.
[58] 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, 2018.
[59] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications," in *Proceedings of the 2014 European Symposium on Research in Computer Security (ESORICS'14)*, 2014, pp. 163–182.
[60] 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.
[61] R. Feng, S. Chen, X. Xie, G. Meng, S.-W. Lin, and Y. Liu, "A performance-sensitive malware detection system using deep learning on mobile devices," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1563–1578, 2020.
[62] R. Feng, J. Q. Lim, S. Chen, S.-W. Lin, and Y. Liu, "Seqmobile: An efficient sequence-based malware detection system using rnn on mobile devices," in *Proceedings of the 25th International Conference on Engineering of Complex Computer Systems (ICECCS'20)*, 2020, pp. 63–72.
[63] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their vompositionality," in *Proceedings of the 2013 Conference on Neural Information Processing Systems (NIPS'13)*, 2013, pp. 1–9.
[64] B. Dai and D. Lin, "Contrastive learning for image captioning," in *Proceedings of the 2017 Conference on Neural Information Processing Systems (NIPS'17)*, 2017, pp. 898–907.
[65] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," *arXiv preprint arXiv:2002.05709*, 2020.
[66] A. Saeed, D. Grangier, and N. Zeghidour, "Contrastive learning of general-purpose audio representations," *arXiv preprint arXiv:2010.10915*, 2020.
[67] R. Qian, T. Meng, B. Gong, M.-H. Yang, H. Wang, S. Belongie, and Y. Cui, "Spatiotemporal contrastive video representation learning," *arXiv preprint arXiv:2008.03800*, 2020.

Yueming Wu received the B.E. degree in Computer Science and Technology at Southwest Jiaotong University, Chengdu, China, in 2016 and the Ph.D. degree in School of Cyber Science and Engineering at Huazhong University of Science and Technology, Wuhan, China, in 2021. He is currently a research fellow in the School of Computer Science and Engineering at Nanyang Technological University. His primary research interests lie in malware analysis and vulnerability analysis.

Shihan Dou received the B.E. degree in Cyberspace Security from Huazhong University of Science and technology, Wuhan, China, in 2021, and is currently pursuing the M.E. degree in School of Computer Science at Fudan University. His primary research interests lie in malware analysis and vulnerability analysis.

Deqing Zou received the Ph.D. degree at Huazhong University of Science and Technology (HUST), in 2004. He is currently a professor of School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST), Wuhan, China. His main research interests include system security, trusted computing, virtualization and cloud security. He has always served as a reviewer for several prestigious journals, such as IEEE TDSC, IEEE TOC, IEEE TPDS, and IEEE TCC. He is on the editorial boards of four international journals, and has served as PC chair/PC member of more than 40 international conferences.

Wei Yang received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign and his bachelor degree from Shanghai Jiao Tong University. He was a visiting researcher in University of California, Berkeley. His research interests are in software engineering and security. His current primary projects relate to Mobile Security, Software engineering/Security for Machine Learning, Intelligent SE/Security, and IoT/Blockchain Security.

Weizhong Qiang received the PhD degree from Huazhong University of Science and Technology (HUST), in 2005. He is currently a professor of School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST), Wuhan, China. His research interest is mainly about software system security. He has published more than 30 research papers.

Hai Jin received the Ph.D. degree in computer engineering from Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994. He is a Cheung Kung Scholars Chair Professor of computer science and engineering at HUST in China. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a fellow of the IEEE, a fellow of the CCF, and a member of the ACM. He has co-authored 22 books and published over 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.