Software Quality Journal (2023) 31:655–685
https://doi.org/10.1007/s11219-022-09602-4

Predicting Android malware combining permissions and API call sequences

Xin Chen[1] · Haihua Yu[1] · Dongjin Yu[1] · Jie Chen[1] · Xiaoxiao Sun[1]

Accepted: 26 September 2022 / Published online: 18 November 2022

Abstract
Malware detection is an important task in software maintenance. It can effectively protect user information from the attack of malicious developers. Existing studies mainly focus on leveraging permission information and API call information to identify malware. However, many studies pay attention to the API call without considering the role of API call sequences. In this study, we propose a new method by combining both the permission information and the API call sequence information to distinguish malicious applications from benign applications. First, we extract features of permission and API call sequence with a decompiling tool. Then, one-hot encoding and Word2Vec are adopted to represent the permission feature and the API call sequence feature for each application, respectively. Based on this, we leverage Random Forest (RF) and Convolutional Neural Networks (CNN) to train a permission-based classifier and an API call sequence-based classifier, respectively. Finally, we design a linear strategy to combine the outputs of these two classifiers to predict the labels of newly arrived applications. By an evaluation with 15,198 malicious applications and 15,129 benign applications, our approach achieves 98.84% in terms of precision, 98.17% in terms of recall, 98.50% in terms of F1-score, and 98.52% in terms of accuracy on average, and outperforms the state-of-art method Malscan by 2.12%, 0.27%, 1.20%, and 1.24%, respectively. In addition, we demonstrate that the method combining two features achieves better performance than the methods based on a single feature.

Keywords Android malware · Malware detection · Permission · API call sequence · CNN

✉ Dongjin Yu
yudj@hdu.edu.cn

Xin Chen
chenxin4391@hdu.edu.cn

Haihua Yu
yuhaihua@hdu.edu.cn

Jie Chen
cjie@hdu.edu.cn

Xiaoxiao Sun
sunxiaoxiao@hdu.edu.cn

[1] School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou 310018, China

1 Introduction

Along with the rapid growth of mobile devices, mobile applications become more and more important and powerful in our daily lives and work. In the first quarter of 2020, there are about 2.56 and 1.85 millions available applications in Google Play[1] and Apple Store[2] for downloading. Due to the open source nature, Android systems have attracted many mobile terminal providers and developers, and become a mainstream operating system in mobile devices. However, motivated by the strong business benefit, some developers attempt to develop malicious applications to steal user information for seeking economic benefits or other motivations. The G DATA report showed that about 750,000 new malicious Android applications were discovered during the first quarter of 2017[3]. Also, in Q1 2020, Kaspersky's mobile products and technologies detected 1,152,662 malicious installation packages, which are 171,669 more than that in the previous quarter[4]. It can be foreseen that a large number of malicious applications will continue to be developed and spread, which will cause various cyber crimes on Android devices. However, due to the invisibility of Android malware, it is difficult to distinguish malicious applications from benign applications for smartphone users.

Recently, researchers have developed many methods for detecting Android malware. In existing studies, the permission feature is widely used for Android malware detection (Alazab et al., 2020; Arp et al., 2014; Peiravian & Zhu, 2013; Wang et al., 2014). Since Android systems apply the permission mechanism to control the accessibility of sensitive resources, some researchers adopt source code analysis tools to extract the permission feature and build classifiers to determine whether an application is malicious. However, many of these methods are easily evaded by obfuscations because the permission feature lacks both semantic information and contextual information of program behaviors. To overcome this challenge, some researchers adopted the Android API call sequence feature to detect malware (Jerbi et al., 2020; Pektaş & Acarman, 2020; Wu et al., 2012). Generally, they leverage some tools to decompile an application for extracting API call sequence information and analyze the potential malicious patterns contained in the API call sequences. Then these features are encoded to train a classifier for predicting the label of each new application. In summary, these studies adopt either the permission feature or the API call sequence feature to detect Android malware, the performance is not promising.

In this paper, we propose a new method based on both the permission feature and the API call sequence feature. Firstly, we extract permission lists from the manifest files contained in the installation packages and API call sequences from the source code files by a decompiling tool. Then, given the great difference between permissions and API call sequences, we adopt two strategies to represent these two features. For the permission feature, we employ a one-hot vector to represent each permission. For the API call sequence feature, we leverage the Word2Vec technique to transform each API into a low-dimensional vector. After that, the Random Forest (RF) algorithm and the Convolutional Neural Network (CNN) algorithm are applied to train a permission-based classifier and an API call sequence-based classifier, respectively. Finally, a linear strategy is designed to combine the outputs of the two classifiers to predict the labels of newly arrived applications.

[1] https://play.google.com/store/apps

[2] https://www.apple.com/ios/app-store/

[3] https://www.gdatasoftware.com/blog/2018/11/31255-cyber-attacks-on-android-devices-on-the-rise

[4] https://securelist.com/it-threat-evolution-q1-2020-statistics/96959/

To validate the effectiveness of the proposed approach, we conduct extensive experiments on the dataset including 15,129 benign applications and 15,198 malicious applications provided by Wu et al. (2019). Then, we investigate five research questions and employ widely used metrics, namely precision, recall, F1-score and accuracy, to evaluate the performance of our approach in malware detection. In addition, we select two classic methods (DroidAPIMiner (Aafer et al., 2013) and Drebin (Arp et al., 2014)), and two state-of-the-art methods (Malscan (Wu et al., 2019) and MalDozer (Karbab et al., 2018)) as baselines for comparison. The experimental results show that our approach achieves 98.84%, 98.17%, 98.50%, and 98.52% in terms of precision, recall, F1-score, and accuracy on average, respectively. Compared with the methods based on a single feature, our method combining two features can achieve better performance for detecting Android malware. Furthermore, we experimentally investigate the most important API call sub-sequences and permissions for identifying Android malware.

In this paper, we make the following contributions:

- In this paper, we present our attempts towards resolving the problem of malware detection by combining both the permission information and the API call sequence information.
- We build a permission-based classifier and an API call sequence-based classifier and design a linear strategy to integrate the outputs of these classifiers to predict the label of each new application.
- We conduct extensive experiments to evaluate the performance of the proposed approach on a public available dataset. Experimental results show that our approach achieves better performance than the baseline methods.

The remainder of this paper is organized as follows. Section 2 describes the background and the motivation for conducting this paper. The details of our approach are detailed in Sect. 3. Sections 4 and 5 present the experimental setup and the experimental result, respectively. The threats to validity are discussed in Sect. 6 and the related work is reviewed in Sect. 7. Finally, we conclude this paper and outline the future work in Sect. 8.

2 Background and motivation

In this section, we detail the background of Android malware and explain the motivation for performing this work.

Malware is a kind of intentionally designed software which will damage a computer, server, client, or computer network (Nash, 2005). Android malware aims at attacking Android systems and contains a wide variety of malware types, including spyware (Herley et al., 2015), ransomware (Young & Yung, 1996), and trojans (Landwehr et al., 1994). For example, the spyware monitors specific users by illegally recording the voice call, chat messages and other personal information. Thus, user privacy and secret are exposed by these malicious applications. The ransomware can illegally encipher the important data of users. If the users do not know the secret key, they cannot acquire the enciphered information. By this way, the attackers can threat the users to pay ransom. As a consequence, these malicious applications seriously damage information security. Actually, most new malicious applications are variants of known malware applications by reusing malicious components. Zhou and Jiang (2012) found that 86% of Android malicious applications

Table 1 The partial list of Android permissions

Permission	Protection level
ACCESS_NETWORK_STATE	normal
ACCESS_WIFI_STATE	normal
CHANGE_WIFI_STATE	normal
WAKE_LOCK	normal
BATTERY_STATS	signature
BIND_DEVICE_ADMIN	signature
SYSTEM_ALERT_WINDOW	signature
ACCESS_COARSE_LOCATION	dangerous
READ_PHONE_STATE	dangerous
WRITE_EXTERNAL_STORAGE	dangerous

are produced by repacking legitimate applications with malicious components. Generally, malicious developers download and decompile some popular benign applications from Google Play. After injecting the malicious components or code into these applications, the developers adopt obfuscation tools to evade the detection, and finally repack and upload them to third-party markets to attract users.

To prevent malicious attacks, Android systems usually provide multiple layers of protection. The most important is the permission control strategy which is one of the central design points of the Android security mechanism. It can effectively protect the privacy of users by restricting some special operations (Alazab et al., 2020) based on the fact that Android applications must request corresponding permissions for accessing sensitive data (such as contacts and GPS) and certain system functions (such as camera and recorder). In the development phase of an Android application, the developer declares the permission requests in a manifest file, namely *AndroidManifest.xml*. When installing an application, a dialog will be displayed to exhibit the requested permission list and ask the user whether these relevant permissions can be granted or not. Empirically, different permissions have different impacts on an application. Google divides permissions into three levels according the riskiness degree of permissions, namely *normal*, *signature* and *dangerous*[5]. The *normal* permissions are very little risk to the privacy of users or the operation of other applications, while the *dangerous* permissions potentially damage some important files or data of users. Table 1 shows some permissions and the corresponding protection levels. For example, the protection level of the ACCESS_NETWORKST_ATE permission is normal, and it allows applications to access the network information.

Figure 1 shows the distribution of applications with different number of request permissions on our dataset. We can observe that more than 50% of benign applications request less than 5 permissions. In addition, compared with benign applications, malicious applications usually request extra permissions which can effectively support their malicious behaviors. Therefore, many researchers tend to leverage permissions to identify malware (Alazab et al., 2020; Arp et al., 2014; Peiravian & Zhu, 2013; Wang et al., 2014). Although these studies have proven that the permission feature is helpful for Android malware detection, there exist some limitations in the permission-based methods. For example, an application may declare the requested permissions in the manifest file, but these permissions are

[5] https://developer.android.google.cn/guide/topics/permissions/overview?hl=en

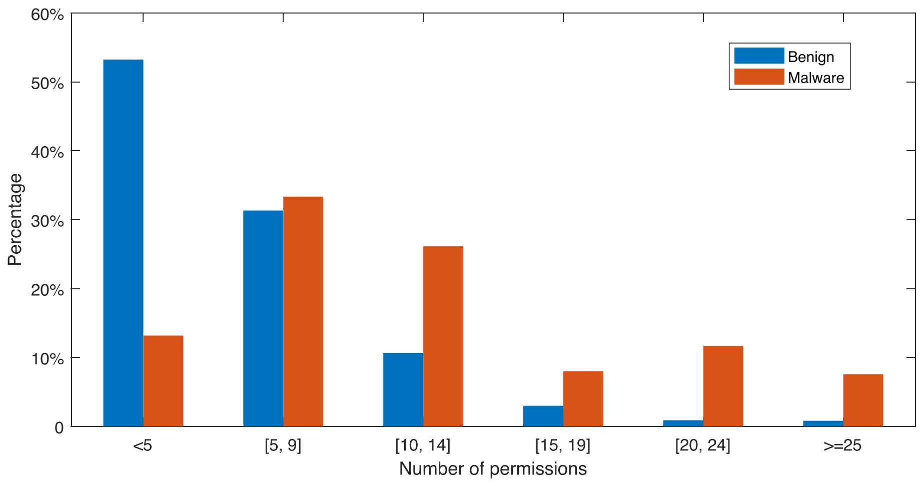


Fig. 1 The statistics of permissions

not used at all in practice. This will provide misleading permission information to influence the detection results. In addition, Shao et al. (2016) discovered that some applications can still access sensitive resources without permissions, thus these applications are hard to be identified by the permission-based methods.

Therefore, some studies leverage API call information to distinguish malicious applications from benign applications (Alazab et al., 2020; Peiravian & Zhu, 2013). In the Android platform, an API is used to interact with the underlying Android system and an application can implement its functions by invoking different APIs. Thus, researchers can analyze the malicious behaviors of applications by the API call information for identifying malware. However, most of these methods just extract API call information but ignore the sequence information between API calls. Thus, they are hard to identify some malicious applications that have similar API calls in benign applications. Figure 2 shows a decompiled fragment of a malicious application, which is a spyware of SMS. This fragment orderly invokes some APIs, namely [*init, init, getLastScan,...,query, getColumnIndex, ...*]. In this fragment, the current operation is based on the previous one and influences the next one. By invoking these APIs in order, the malicious application can obtain the SMS list. Thus, analyzing the API call sequences can help understand the intention of an application and improve the accuracy of detecting malware.

```
getSMSList-BB@0x0 : [ getSMSList-BB@0xb2 getSMSList-BB@0x158 ]
        0       (00000000) new-instance         v6, Ljava/util/ArrayList;
        1       (00000004) invoke-direct        v6, Ljava/util/ArrayList;-><init>()V
        2       (0000000a) new-instance         v0, Lcom/carrotapp/smsspypro/Setting;
        3       (0000000e) invoke-direct        v0, v9, Lcom/carrotapp/smsspypro/Setting;-><init>(Landroid/content/Context;)V
        4       (00000014) invoke-virtual       v0, Lcom/carrotapp/smsspypro/Setting;->getLastScan()J
        ...
        32      (0000007a) invoke-virtual/range v0 ... v5, Landroid/content/ContentResolver;->query(Landroid/net/Uri;
                           [Ljava/lang/String; Ljava/lang/String; [Ljava/lang/String; Ljava/lang/String;)Landroid/database/Cursor;
        33      (00000080) move-result-object   v4
        34      (00000082) const-string         v0, 'body'
        35      (00000086) invoke-interface     v4, v0, Landroid/database/Cursor;->getColumnIndex(Ljava/lang/String;)I
        ...
        45      (000000ae) if-eqz               v0, +55
```

Fig. 2 Decompiled code of a basic block

Based on the above background, we attempt to explore a new classification method which combines the permission feature and the API call sequence feature to detect malicious applications.

3 Methodology

In this section, we systematically describe the whole flow of our approach. As shown in Fig. 3, our approach takes in the corresponding vectors of Android application files as the input and then outputs the class label of an application, namely malware and benign. More specifically, we firstly extract permissions and API call sequences from the decompiled application. Then, in the stage of the vector representation, we adopt two different strategies to represent the permissions and the API call sequences, respectively. After that, RF is applied to train a permission-based classifier, meanwhile CNN is applied to train an API call sequence-based classifier. Finally, we use a linear strategy to combine the results of these two classifiers to predict the labels of newly arrived applications.

3.1 Feature extraction

Normally, an Android application package includes a manifest file and a dex file. The manifest file describes essential information about the application, and the dex file contains all the code of the application. In our approach, we respectively extract the permission feature and the API call sequence feature from the manifest file and the dex file by utilizing Androguard (Mercaldo et al., 2016), a widely used python tool for reverse engineering of Android applications.

For the permission feature, we use *Androguard* to decompile the Android application and return an *apk* object which contains some information about this application, such as permissions, decompiled instructions, and activities. Thus, we can directly obtain the permission list from this *apk* object.

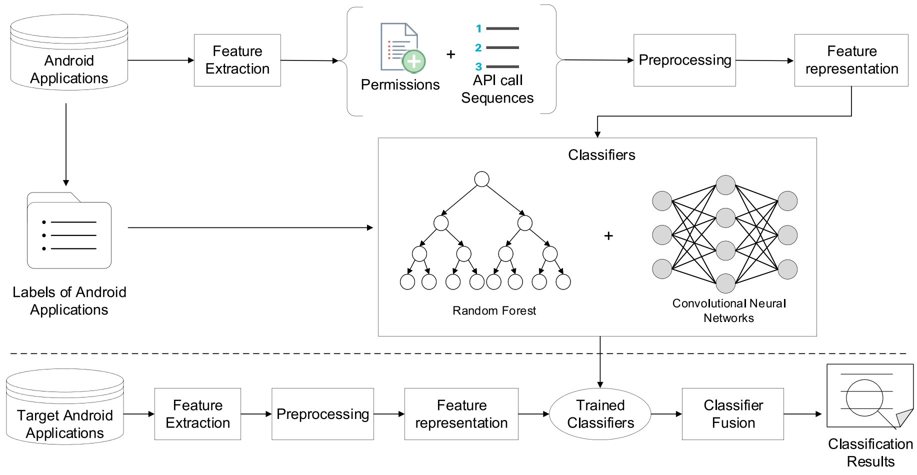


Fig. 3 The overview architecture of our approach

For the API call sequence feature, we can collect API calls and orderly organize them to form API call sequences through decompiled instructions. As shown in Fig. 2, a normal API call instruction is formatted as follows:

invoke-direct v0, Ljava/util/ArrayList;-><init>*()V*

The opcode is *invoke-direct*, and the invoked API is *java/util/ArrayList;->*<init>*()V*. Generally, the opcode used for invocation includes *invoke-direct, invoke-static*, *invoke-interface*; *java/util* is the package name of this API, *ArrayList* is the class name, and *init* is the function name. This API is called for the initialization of an array object. Thus, through checking the type of opcode, we can discriminate that those instructions are used to invoke APIs. Inspired by Karbab et al. (2018), the API call sequence of an Android application is generated by merging the API call sequence of each basic block within this application in order. Suppose the decompiled application denoted by d is composed by n basic blocks, $d = \{b_1, b_2, ..., b_n\}$. Each basic block consists of a series of instructions. Thus the API call sequence of a basic block is produced by recording the operands of the selected instructions. For the whole application, we ignore the relationship between basic blocks, and orderly combine the API call sequence of each basic block as the API call sequence of the whole application.

3.2 Preprocessing

After feature extraction, we preprocess the permission feature and the API call sequence feature by filtering out the useless information, such as custom permissions and user-defined functions.

For the permission feature, we build a dictionary containing the officially defined permissions. Given an application, we first extract all the permissions. Then, if a permission does not match any permission in the dictionary, it is deleted. Finally, the retained permissions are the officially defined ones. In fact, Google continuously updates permissions in every version of API levels by adding some permissions or deleting some permissions. Thus the number of permissions between different API levels is different. For example, the latest Android version (API level 30 released in 2020) contains 166 permissions, while the earliest Android version (API level 1) involves 76 permissions. Considering that the applications of our dataset are from 2011 to 2018, we collect the officially defined permissions from API levels 1 to 28 and remove the duplicate permissions to form the final permission dictionary.

By investigating the API calls, we observe that there exist some third-party libraries including API calls and user-defined function calls, which are considered as noise in API call sequences since they are usually composed of mixed symbols or words, thus impeding the classification results. Therefore, we need to filter out the noise data. We identify the package name of each API call. If an API call involves the official packages, including *android, org, java, dalvik, javax* and *junit*, we keep these API calls. Meanwhile, all other API calls are filtered out.

3.3 Vector representation

After feature extraction, the main task is to represent the feature information for each sample. In this study, we extract two types of features. There are differences between them. First, the number of APIs far exceeds that of permissions. Therefore, we need to adopt two different strategies to represent these two types of features. Second, the API call sequence feature contains semantic information. Thus, the strategy for API call sequences should effectively reflect the semantic information and the context information.

3.3.1 Vector representation for permissions

For the permission feature, we adopt the one-hot encoding method. More specifically, each permission can be represented by an N-dimensional vector, where N is the number of permissions. The vector consists of 0s in all cells with the exception of a single 1 in a cell used uniquely to identify the corresponding permission (Harris & Harris, 2010). For example, there are three permissions a, b, c. [1, 0, 0] represents a, [0, 1, 0] represents b, and [0, 0, 1] represents c. Thus, if an application requests the permissions a and b, its vector representation is [1, 1, 0].

3.3.2 Vector representation for API call sequences

Different from permissions, the number of APIs is very large. The one-hot encoding is not suitable for representing API call sequences since an overly high number of dimensions may lead to the curse-of-dimensionality problem. Therefore, we attempt to adopt neural network methods for the vector representation of API call sequences, such as Word2Vec and Auto-Encoder. Since we need to consider the semantic information contained in API call sequences, we use the Word2Vec technique to represent API call sequences since Word2Vec can transform each API call sequence into a low-dimensional vector and meantime keep the certain context information of each vector.

Word2Vec is a widely used technique for Natural Language Processing (NLP). It was originally designed to transform each word within word sequences into a low-dimensional vector. In general, Word2Vec includes continuous bag-of-words (CBOW) (Mikolov et al., 2013a) and continuous skip-gram (Skip-gram) (Mikolov et al., 2013b). The CBOW model predicts the current word based on the context, while the Skip-gram model aims to predict the context based on the given input word. Considering the Skip-gram model works much better than the CBOW model on semantic tasks (Mikolov et al., 2013a), we select the Skip-gram model for representing API call sequences in our approach. Figure 4 shows the framework of Skip-gram model and gives an example to explain how it works. In Fig. 4a, the skip-gram model contains three layers, an input layer, a hidden layer and an output layer. It aims to obtain the weight parameter matrix $W_{V\times N}$ between the input layer and the hidden layer, where V is the vocabulary size of the training set and N is the hidden layer size. In the input layer, the word is represented by a one-hot vector and the vector length is also V. Based on $W_{V\times N}$, Skip-gram propagates the input to the hidden layer h which is defined as follows:

$$h = x^T W_{V\times N} = W_{(k,)} \tag{1}$$

where x is the vector representation of the k-th word in the vocabulary list. Since x is a vector with $x_k = 1$ and $x_{k'} = 0$ for all $k' \neq k$, the output of h is the k-th row of $W_{V\times N}$, namely $W_{(k,)}$.

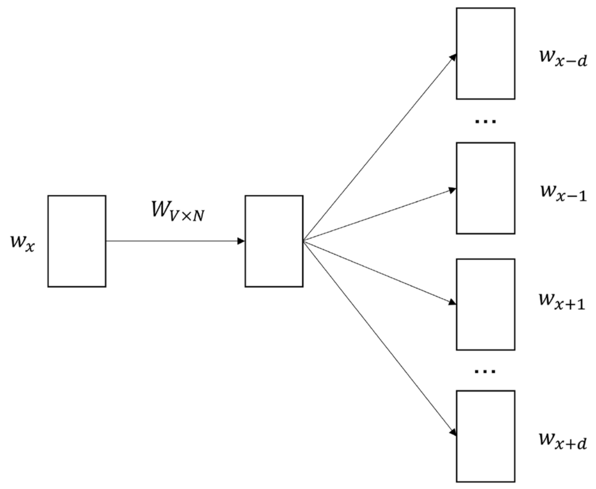
Then, the output of the hidden layer continues to propagate forward. In the figure, the input word is w_x and the context length is set to $2d + 1$. The output layer outputs $2d$ multinomial distributions, and each output is computed by the softmax function with the same parameter matrix. According to the input of w_x, Skip-gram tries to predict the context words of w_x, and the probability is calculated by the following formula:

$$p(w_{i,j} = w_{O,i}|w_x) = \frac{exp(u_{i,j})}{\sum_{i'=1}^{2d} exp(u_{i',j})} \tag{2}$$

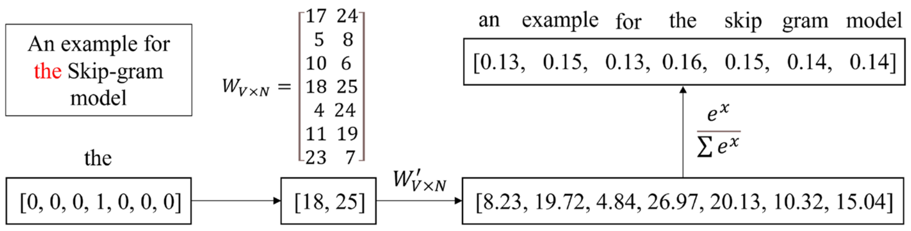
where $w_{i,j}$ is the j-th word on the i-th panel of the output layer, $w_{O,j}$ is the actual context words of the j-th word, $u_{i,j}$ is the value of the j-th unit on the i-th panel of the output layer. The Skip-gram model aims to maximize the probability of the actual output $[w_{x-d}, ..., w_{x-1}, w_{x+1}, ..., w_{x+d}]$, the loss function is transformed into:

$$E = -\log p(w_{x-d}, ..., w_{x-1}, w_{x+1}, ..., w_{x+d}|w_x) \tag{3}$$

Through the back propagation, Skip-gram optimizes the output by tuning the parameter matrix $W_{V\times N}$. After the training, we obtain the matrix $W_{V\times N}$, thus each API call sequence can be represented by a low-dimensional vector. In Fig. 4b, we select the text "an example for the skip-gram model" to predict the surrounding context of the center word "the". With the one-hot vector [0, 0, 0, 1, 0, 0, 0] as the input, the model outputs another vector [0.13, 0.15, 0.13, 0.16, 0.15, 0.14, 0.14] which is very different from the target vector [0, 1, 1, 0, 1, 1, 0] when $d = 2$. In this work, we utilize a python tool *Gensim*



(a) Skip-gram model


An example for the Skip-gram model
the
[0, 0, 0, 1, 0, 0, 0]
$W_{V\times N} = \begin{bmatrix} 17 & 24 \\ 5 & 8 \\ 10 & 6 \\ 18 & 25 \\ 4 & 24 \\ 11 & 19 \\ 23 & 7 \end{bmatrix}$
[18, 25]
$W'_{V\times N}$
[8.23, 19.72, 4.84, 26.97, 20.13, 10.32, 15.04]
$\frac{e^x}{\sum e^x}$
an example for the skip gram model
[0.13, 0.15, 0.13, 0.16, 0.15, 0.14, 0.14]

(b) Skip-gram example

Fig. 4 The Skip-gram model for word embedding

(Srinivasa-Desikan, 2018), an open-source library for unsupervised topic modeling and NLP, to train the Skip-gram model.

3.4 Classifiers

Similarly, given the characteristics of the vector representation methods, we adopt two different method, namely Random Forest (RF) and Convolutional Neural Networks (CNN), to train the classifiers in this study, respectively.

3.4.1 Random Forest

RF is a classification and regression algorithm developed by Breiman et al. (2001). It is an ensemble of a set of decision trees learned on reduced training sets, and these decision trees are independent of each other. A reduced training set is constituted by randomly sampling from both samples and features. To determine the class of a test sample, the outputs from all decision trees are voted to determine the final output. Since each decision tree is unpruned and grown fully, it has the characteristic of low bias (Zhu et al., 2018). Meantime, due to the randomness of these two operations, RF can correct the overfitting problem of decision trees. The final prediction result is determined based on the prediction results of these decision trees.

In this study, we adopt RF to build a permission-based classifier to distinguish malicious applications from benign applications. Specifically, we implement the RF classifier by utilizing scikit-learn (Pedregosa et al., 2011), a python library for machine learning. Besides, there are some parameters impacting the performance of the RF classifier, e.g., *min_samples_leaf* and *n_estimators*. *min_samples_leaf* refers to the minimum number of samples required in a leaf node. *n_estimators* refers to the number of trees in the forest. To optimize the classifier, we tune these parameters via a grid search (LeCun et al., 1998a).

3.4.2 CNN

Many studies have demonstrated the good performance of deep learning in classification and regression. In our approach, we adopt CNN (Hui et al., 2021) to implement the API call sequence-based classifier. The overview of the neural network architecture of the CNN model is depicted in Fig. 5.

1. *Input and embedding layers*: Firstly, each application is represented as a sequence of API calls $d = (API_1, API_2, ..., API_n)$, and input into the input layer. Then, in the embedding layer, a dictionary is generated by Word2Vec and each API is transformed into a d-dimensional vector. Thus, each application can be represented by a matrix, $d = (v_1, v_2, ..., v_n)$, where v_i is the vector of the i-th API call.
2. *Convolutional layer*: A convolutional layer contains a set of filters whose parameters need to be learned. In our approach, we adopt three kinds of sized convolutional kernels, namely 3, 4 and 5, to extract different features from the input layer. In addition, *ReLU* is applied as the activation function, which is defined as follows:

$$ReLU = max(0, w^T x + b) \tag{4}$$

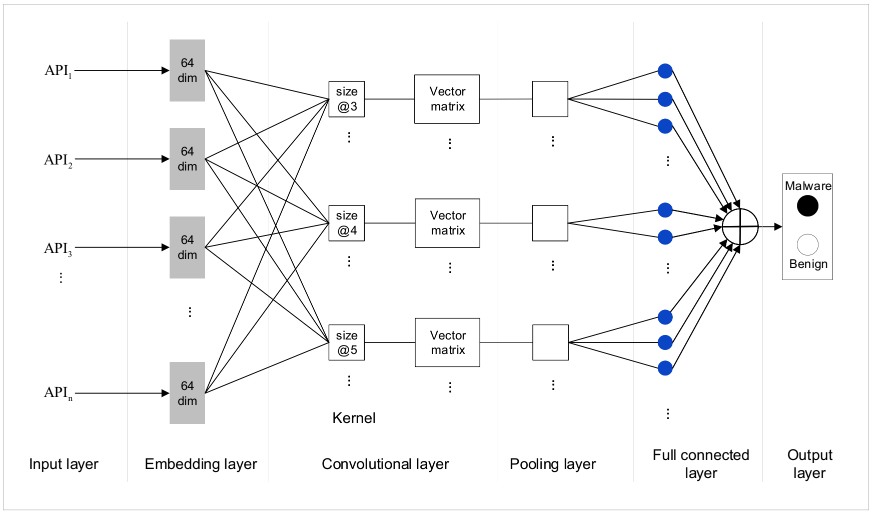


Fig. 5 The neural network architecture of the CNN model

where w is the convolutional kernel, x is the input vector, and b is the offset that needs to be learned during the model training.

3. *Pooling layer*: The pooling layer is used to reduce the redundancy of the output of the convolution layer. In this layer, we adopt the max pooling which can divide the input into regions without overlap and select the maximum value of each region as the output.
4. *Full connected and output layers*: The full connected layer combines each output of the pooling layer as a feature vector. After that, the feature vector is forwarded as an input to the output layer. Through the softmax function, the output layer predicts the probability of each class, namely malicious or benign.
5. *Model training*: During the training of this model, we adopt the adaptive learning rate method Adam (Kingma & Ba, 2014) which is the extension of the Stochastic Gradient Descent algorithm (Bottou, 1998) to optimize the model. Through this method, the parameters of our model can be updated efficiently in the training process.

3.5 Classifier fusion

This step is to integrate these two classifiers of which each contributes to the final result. In general, there are two methods to merge the outputs of different classifiers. One is calculating the average value of the outputs of all classifiers. However, this method cannot reflect the importance of different features. The other is calculating the weighted result of each classifier. This method can effectively reflect the importance of different features, but needs to set an extra parameter. In this study, since the contribution of each classifier to the final result is different, we calculate the weighted result of each classifier, which is defined as follows:

$$R = (1 - \alpha)R_{RF} + \alpha R_{CNN}, R \in \{0, 1\} \tag{5}$$

where α is a parameter to reflect the contribution of different classifiers, R_{RF} is the result of the RF-based classifier, R_{CNN} is the result of the CNN-based classifier, and R is the round value, 0 means benign and 1 means malicious.

4 Experiment setup

In this section, we detail the experimental setup. First, we present the research questions (RQs). Then we explain the dataset used in this study. After that, we introduce the baseline methods. Finally, we describe the experimental platform and parameter settings.

4.1 Research questions

In this paper, we mainly investigate the following five RQs to validate the effectiveness of our method from different aspects.

- **RQ1:** Can our method outperform the baseline methods?
- **RQ2:** How effective is the combination of these two features?
- **RQ3:** How does the parameter α impact the effectiveness of our approach?
- **RQ4:** What is the performance of our method when the training set and the test set belong to different years?
- **RQ5:** What are the most important permissions and API call sub-sequences for identifying malware?

RQ1 is designed to evaluate the effectiveness of our approach by comparing with the baseline methods. *RQ2* is designed to investigate whether the method based on the combination of these two features outperforms the methods based on a single feature. *RQ3* is designed to investigate the impact of different α on the classification results. *RQ4* is designed to test the robustness of our approach when the testing set and training set belong to different years. *RQ5* is designed to investigate the most important permissions and API call sub-sequences for identifying malware.

4.2 Dataset

In the literature, Wu et al. (2019) created a dataset by crawling *apks* from Andrzoo (Allix et al., 2016), which currently contains over twelve millions *apks* of which each is detected by tens of different antiVirus products to determine whether it is malicious or not. In this study, we also crawl *apks* from Andrzoo according the *apk* list provided by Wu et al. (2019) and remove 388 damaged *apk* files. Eventually, the dataset is composed of 30,327 samples, including 15,129 benign applications and 15,198 malicious applications.

To evaluate the effectiveness of our method better, we divide the dataset into sub-datasets by year. Table 2 shows the summary information of the dataset. We can see that the dataset is constituted by 8 sub-datasets and the applications of each sub-dataset are from the same year, such as 2011 and 2012. As shown in the table, the number of benign applications and malicious applications in each sub-dataset is close to each other. For example, the number of benign applications and malicious applications is 1,919 and 1,916 in 2011, respectively. Meanwhile, the total number of benign applications and malicious applications is close to 1:1.

Table 2 Summary of the datasets

Dataset	Benign	Malware	Total	Average Size (MB)
2011	1919	1916	3835	2.48
2012	1872	1996	3868	3.70
2013	1896	2000	3896	6.55
2014	1826	1982	3808	7.15
2015	1811	1839	3650	8.36
2016	2014	1939	3953	8.46
2017	1792	1705	3497	7.60
2018	1999	1821	3820	7.94
Total	15129	15198	30327	6.51

Notably, we only crawl data following the list provided by Wu et al. (2019) and do not perform extra operations to keep this balance. The balance kept by Wu et al. may be because it benefits the training of classifiers. Besides, we can find that the average size of *apks* from 2013 to 2018 is about 7MB while the average size of *apks* from 2011 and 2012 is 2.48MB and 3.70MB. The reason may be that there are a small number of mobile applications in 2011 and 2012. The configuration of mobile devices is hard to run complex applications and users have no expectation about mobile applications.

4.3 Metrics

In our study, we aim to detect malware from raw applications, which can be regarded as a binary classification problem. In many studies about binary classification, precision, recall, F1-score (F1) and accuracy, are frequently applied to evaluate the effectiveness of automated techniques. Therefore, we also employ these four metrics to evaluate the effectiveness of our approach.

Let *TP* refer to the number of correctly classified malicious samples, *TN* refer to the number of correctly classified benign samples, *FP* refer to the number of incorrectly classified malicious samples, *FN* refer to the number of incorrectly classified benign samples.

Precision evaluates the correctness degree of the prediction results. The formula for calculating precision is as follows.

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

Recall reflects the consistency degree between the prediction results and the ground truth. The formula for calculating recall is as follows.

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

F1-score is the tradeoff between precision and recall. The formula for calculating F1-score is as follows.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{8}$$

Accuracy refers to the degree of correctness between the predicted results of all classes and the ground truth. The formula for calculating accuracy is as follows.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{9}$$

4.4 Evaluation method

In order to reduce the impact of overfitting and the selection bias, researchers usually adopt tenfold cross-validation to evaluate their approaches (Kohavi et al., 1995). Thus, in this study, we also employ tenfold cross-validation to evaluate our approach. More specifically, we randomly split the dataset into 10 folds and each fold includes the same number of samples. In each run, a model is trained using 9 folds as the training set; then, the trained model is evaluated by the remaining fold. We run the experiment 10 times for each dataset and measure the metrics. Then the average values of these metrics are calculated as the final results.

4.5 Baselines

In order to evaluate our approach, we select two classic methods and two state-of-the-art methods for comparison.

Drebin (Arp et al., 2014) A classic Android malware detection method, which conducted a broad analysis to extract as many features as possible from an apk, such as permissions, urls, and intents. After embedding these features into a vector, they trained a SVM (Vapnik, 2013)-based model to detect malware.

DroidAPIMiner (Aafer et al., 2013) A classic Android malware detection method, which leveraged permissions, API calls and the parameters of APIs to represent an application. Then they adopted the k-nearest neighbor algorithm to identify malware.

Malscan (Wu et al., 2019) A state-of-the-art Android malware detection method, which treats the function call graphs of Android applications as a social network and conducts the social network-based centrality analysis to extract semantic features for detecting malware.

MalDozer (Karbab et al., 2018) A state-of-the-art Android malware detection method, which extracts the API call sequences as a feature. After embedding the API in a low-dimensional vector space, they adopt a CNN-based model to identify malware.

4.6 Experimental platform and parameter settings

In this paper, all the experiments are conducted with Python 3.7, compiled by Pycharm 2019.2 and run on a PC with 64-bit Ubuntu 16.04, an Intel Core (TM) i9-7900X CPU and a GeForce GTX 1080Ti GPU.

RF and CNN contain a series of parameters which have great impacts on the performance of our method. Thus, to achieve good performance, the parameters are setting as follows. In the RF model, we conduct an experiment for determining these two parameters.

Table 3 The parameter settings of our CNN model

Hyper-parameter	Value
Number of epochs	10
Batch size	64
Dropout rate	0.5
Embedding size	64
Number of the convolution filters	512
Convolution filter size	[3, 4, 5]

Eventually, *the number of trees* is set to 131 and *the minimum number of leaf nodes* is set to 1. Meanwhile, other parameters are default. For the CNN model, the parameter settings are presented in Table 3. CNN involves six hyperparameters. We leverage the gradient search method (LeCun et al., 1998b) to obtain the suitable values for the hyperparameters. *Number of epochs* is the number of epochs to train the model. *Batch size* is the number of samples in each gradient updating. *Dropout rate* is a rate which refers to the fraction of the input units to drop. *Embedding size* is the dimensionality of the API call vector in the embedding layer. *Number of the convolution filters* is the number of convolution filters in the convolution layer. *Convolution filter size* is the size of a convolution kernel.

5 Experiment results

In the section, we aim to evaluate our proposed method and answer the previous mentioned RQs.

5.1 Investigation to RQ1

Motivation In this paper, we propose a novel method to detect Android malware by combining the permission feature and the API call sequence feature. Currently, there have been many methods that are adopted to resolve this problem. To evaluate the effectiveness of our method, we select two classic methods (DroidAPIMiner (Aafer et al., 2013) and Drebin (Arp et al., 2014)) and two state-of-the-art methods (Malscan (Wu et al., 2019) and MalDozer (Karbab et al., 2018)) as baselines for comparison. In this RQ, we investigate whether our approach can outperform the baseline methods in detecting Android malware.

Approach We train the RF classifier and the CNN classifier using the dataset with malicious applications and benign applications. In this study, we divide the samples from each year into 10 equivalent folds of which each includes both malicious applications and benign applications according to the tenfold cross-validation method. For each year, we use the samples from 9 folds to create the training set. The samples from the remaining 1 fold are used to evaluate the effectiveness of these two classifiers in detecting malware. We repeat this process for each of 10 folds, each training on 9 folds and testing on the remaining 1 fold. The average results are calculated in the 10 runs for each year.

Since these four methods only adopt some of the metrics used in this paper to evaluate the effectiveness of their methods, and the used datasets in their experiments are different from this study, we cannot directly copy the experimental results from their studies to validate the effectiveness of our method. Therefore, in this experiment, we reproduce these four methods by fully following the descriptions from their studies. First, we leverage the same python tool to extract features used in these four baseline methods. Then, we create the training set and the testing set based on the used evaluation method in the studies (Aafer et al., 2013; Arp et al., 2014; Wu et al., 2019; Karbab et al., 2018). Finally, we train the classifiers by the training set and predict the labels of testing set to evaluate the effectiveness of these four methods.

Results Table 4 presents the experimental result of our approach and the baseline methods in terms of precision, recall, F1-score and accuracy, respectively. In the table, we use "P", "R", "F1" and "A" to represent precision, recall, F1-score and accuracy, respectively. In addition, the best result on each sub-dataset is highlighted in bold.

As shown in Table 4, our approach can obtain the highest F1-score on each sub-dataset. On average, our approach achieves 98.50% in terms of F1-score, which is 1.21%, 4.71%, 2.29% and 2.32% higher than Malscan, MalDozer, DroidAPIMiner and Drebin, respectively. Meantime, our approach also achieves the best results in terms of precision and accuracy on these sub-datasets. The potential reason is that our approach combines the advantages of the permission feature and the API call sequence feature to detect malware comprehensively. However, on some sub-datasets, some baseline methods achieve slightly higher recall than our approach. For example, Malscan achieves 98.58% in terms of recall and is 1.03% higher than our method in 2018. This may due to that our approach conducts classification by combining the outputs of two classifiers with a weight which may have impacts on the final result when selecting different weight values. That is, our approach may achieve better results when the parameter α is set to another different value on this sub-dataset. Additionally, as seen from Table 4, we can observe that the precision achieved by our method is higher than the recall achieved by our method on all sub-datasets. This indicates that our approach may achieve higher precision at expense of recall. We will answer this problem by presenting the results of our method under different weights in RQ3.

In these baseline methods, Malscan and MalDozer both apply a single feature to detect malware. Compared with MalDozer, Malscan achieves better performance in terms of F1-score on most of the sub-datasets. The potential reason is that Malscan can extract the structural semantics of sensitive APIs to detect malware. Similarly, DroidAPIMiner and Drebin apply permissions and API calls to detect malware. They focus on the syntactic information, i.e., they only consider whether some APIs are called, but ignore the role of sequence information of API calls. In addition to the syntactic information, we also consider the sequence information of API calls to detect malware, hence our approach has better performance than DroidAPIMiner and Drebin.

In summary, our approach achieves 98.84%, 98.17%, 98.50%, and 98.52% in terms of precision, recall, F1-score, and accuracy on average, respectively, and outperforms these baseline methods.

5.2 Investigation to RQ2

Motivation Existing methods usually adopt either the permission feature or the API call sequence feature for malware detection. In this study, given their respective advantages, we attempt to combine these two features to detect malware. However, we do not acknowledge whether the combination of two features outperforms a single feature in malware detection.

Table 4 The experimental result of our approach and baseline methods

Dataset	Our method				Malscan				MalDozer				DroidAPIMiner				Drebin			
	P%	R%	F1%	A%	P%	R%	F1%	A%	P%	R%	F1%	A%	P%	R%	F1%	A%	P%	R%	F1%	A%
2011	**98.94**	98.34	**98.63**	**98.64**	96.38	97.63	97.00	97.00	95.88	96.96	96.40	96.37	97.40	**98.38**	97.88	97.86	97.41	97.23	97.32	97.31
2012	**99.00**	98.56	**98.77**	**98.73**	97.24	**98.85**	98.03	97.96	94.56	96.56	95.51	95.61	97.39	83.30	89.77	90.20	97.96	98.19	98.07	98.01
2013	**99.15**	**98.86**	**99.00**	**98.97**	96.14	98.29	97.20	97.10	96.44	95.31	95.80	96.00	97.59	97.96	97.77	97.71	97.48	98.15	97.81	97.74
2014	**99.34**	**98.38**	**98.86**	**98.82**	96.44	96.97	96.70	96.56	92.87	96.62	94.58	94.67	98.54	97.67	98.10	98.05	96.93	97.51	97.21	97.12
2015	**98.74**	**98.22**	**98.48**	**98.47**	96.55	97.16	96.83	96.82	94.75	92.44	93.51	93.70	97.26	94.66	95.94	95.97	95.15	96.15	95.63	95.59
2016	**98.65**	97.93	**98.29**	**98.33**	96.94	**98.37**	97.64	97.67	89.99	95.07	92.37	92.01	97.91	96.68	97.29	97.19	95.91	95.99	95.94	96.09
2017	**98.24**	**97.55**	**97.89**	**97.97**	96.53	97.34	96.92	97.00	90.12	90.98	90.28	90.05	97.98	94.12	96.00	96.18	94.78	94.81	94.78	94.92
2018	**98.68**	97.53	**98.10**	**98.19**	97.53	**98.58**	98.04	98.12	90.33	93.84	91.88	91.47	97.28	96.61	96.93	97.08	92.94	92.50	92.69	93.07
Average	**98.84**	**98.17**	**98.50**	**98.52**	96.72	97.90	97.30	97.28	93.12	94.72	93.79	93.73	97.67	94.92	96.21	96.28	96.07	96.32	96.18	96.23

In this RQ, we mainly focus on investigating the effectiveness of the combination of these two features.

Approach In this RQ, we compare our approach with two variants of our method, namely the permission-based method and the API call sequence-based method. In this experiment, we remove one classifier and keep another classifier. Additionally, in the process of model training, we keep the other steps unchanged to verify its detection effectiveness. Notably, we adopt different vector representation methods for these two variants since we have experimentally demonstrated that the one-hot encoding is more suitable for the permission-based method and the skip-gram model is more suitable for the API call sequence-based method.

Results Table 5 shows the experimental results of our approach and these two variants in terms of precision, recall, F1-score and accuracy. As shown in the table, our approach achieves 98.84%, 98.17%, 98.50% and 98.52% in terms of precision, recall, F1-score, and accuracy on average, respectively, and outperforms both the permission-based method and the API call sequence-based method. In addition, our approach and the API call sequence-based method outperform the permission-based method in terms of precision, recall, F1-score and accuracy on all the sub-datasets. The potential reason may be that the dimensionality of API call sequences is much higher than that of permissions. The higher the number of dimensions is, the more information the API call sequences convey. In addition, the API call sequence feature contains some semantic information, e.g., the program behavior information, which can improve the performance of classification.

Besides, we can find that our approach achieves high results in terms of F1-score in the last five years, but slightly low results in the first three years. The potential reason is that in the first years android applications are relatively simple and with small size. As shown in Table 2, the average size of Android applications in 2011, 2012 and 2013 is lower than that of the next five years. Generally, with the increase of the size of Android applications, the complexity of these applications may also increase correspondingly, which may impact the detection performance of the API call sequence-based method. Compared with the API call sequence-based method, our approach also combines the

Table 5 The experimental result of two single feature-based methods and our approach

Dataset	API call sequence				Permission				Our approach			
	P%	R%	F1%	A%	P%	R%	F1%	A%	P%	R%	F1%	A%
2011	**99.15**	**98.44**	**98.79**	**98.80**	91.51	93.80	92.62	92.54	98.94	98.34	98.63	98.64
2012	**99.24**	**98.70**	**98.97**	**98.94**	94.38	92.29	93.31	93.17	99.00	98.56	98.77	98.73
2013	**99.30**	**98.86**	**99.08**	**99.05**	93.73	94.51	94.12	93.97	99.15	**98.86**	99.00	98.97
2014	**99.59**	97.84	98.70	98.66	93.14	95.62	94.35	94.04	99.34	**98.38**	**98.86**	**98.82**
2015	98.69	98.16	98.42	98.41	94.93	93.76	94.34	94.33	**98.74**	**98.22**	**98.48**	**98.47**
2016	97.78	97.72	97.74	97.80	95.95	94.36	95.13	95.30	**98.65**	**97.93**	**98.29**	**98.33**
2017	98.29	97.37	97.81	97.91	81.77	94.01	87.46	86.85	**98.24**	**97.55**	**97.89**	**97.97**
2018	97.97	97.16	97.55	97.67	87.07	76.09	79.76	81.81	**98.68**	**97.53**	**98.10**	**98.19**

permission feature to detect malware, thus achieving better detection performance in these years.

In summary, our approach combining permission and API call sequences can achieve better performance than the methods based on a single feature in detecting malware.

5.3 Investigation to RQ3

Motivation In our method, we adopt the permission information and the API call sequence information to build classifiers, respectively. Considering that the outputs of these two classifiers contribute the prediction result of a sample from the test set, we design a linear strategy to integrate the outputs of classifiers to determine the final result by setting a parameter α. Intuitively, α has an important impact on the effectiveness of our method in identifying malware. In this RQ, we conduct experiments to investigate the impact of α on our approach and seek a suitable value for α to achieve good results on these sub-datasets.

Approach In this experiment, we vary the value of α from 0 to 1. Meanwhile, we set the step to 0.1. We select two sub-datasets to run the experiment for determining the best value of α, and present the results of our method on other sub-datasets to analyze whether the selected parameter value is suitable for other sub-datasets. Specifically, we select the middle part (namely 2014 and 2015) of all the years as the tuning sub-datasets. Since F1-score is the tradeoff between precision and recall, we select F1-score as the main evaluation metric to determine the best value of α.

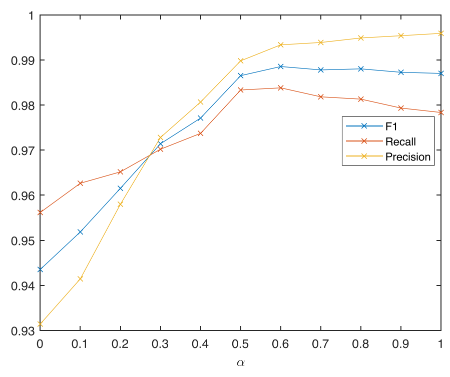
Results Table 6 shows the experimental results of our approach with different α in terms of F1-score. In Fig. 6, we use two sub-figures to present the tuning results when α varies from 0 to 1. As shown in Fig. 6a, in 2014, when α is 0, our approach achieves the lowest result in terms of F1-score. With the increase of α, the F1-score achieved by our approach also increases and reaches the peak when α is set to 0.6. After that the F1-score achieved by our approach stays relatively stable and slightly lower than the maximum value. Similarly, in 2015 our approach also achieves the best result in terms of F1-score when α is equal to 0.6. Besides, in these two years, our approach also achieves the best result in terms of accuracy when $\alpha = 0.6$. Therefore, we select 0.6 as the default parameter value for α since our approach can achieve relatively good performance in malware detection.

In other years, we also conduct experiments with different α to verify whether our approach can achieve good performance when α is set to the default value 0.6. Figure 7 uses six sub-figures to present the experimental results of our method. In these sub-figures, our approach shows the basically similar tendency with the change of α. For example, as shown in Fig. 7f, the result achieved by our approach rises from 79.76% to 98.10% and then falls slightly from 98.10% to 97.55% in terms of F1-score with the continuous growth of α in 2018. As seen from Table 6, we can observe that our approach can achieve the maximum or approximate maximum F1-score in all the years when α is set to 0.6.

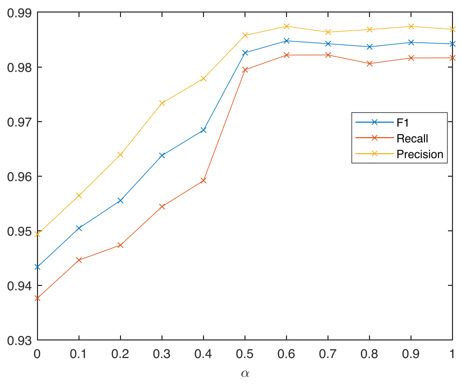
In summary, our approach obtains different results in terms of precision, recall, F1-score and accuracy under different α. When α is set to 0.6, the optimal results or approximate optimal results can be obtained by our method. Thus, we select 0.6 as the default value of α.

Table 6 The experimental result of our approach with different α in terms of F1-score

Dataset	$\alpha = 0$	$\alpha = 0.1$	$\alpha = 0.2$	$\alpha = 0.3$	$\alpha = 0.4$	$\alpha = 0.5$	$\alpha = 0.6$	$\alpha = 0.7$	$\alpha = 0.8$	$\alpha = 0.9$	$\alpha = 1$
2011	92.62%	93.54%	94.36%	95.40%	96.68%	98.36%	98.63%	98.71%	98.74%	98.76%	**98.79%**
2012	93.31%	93.98%	94.66%	95.98%	97.82%	98.57%	98.77%	98.75%	98.80%	98.87%	**98.97%**
2013	94.12%	94.84%	95.93%	96.62%	97.58%	98.82%	99.00%	**99.08%**	**99.08%**	**99.08%**	**99.08%**
2014	94.35%	95.19%	96.15%	97.14%	97.71%	98.65%	**98.86%**	98.78%	98.80%	98.73%	98.70%
2015	94.34%	95.05%	95.55%	96.38%	96.84%	98.26%	**98.48%**	98.42%	98.37%	98.45%	98.42%
2016	95.13%	95.64%	96.11%	96.60%	97.89%	98.26%	**98.29%**	98.00%	97.84%	97.82%	97.74%
2017	87.46%	92.08%	95.49%	96.41%	97.18%	97.81%	**97.89%**	97.88%	97.88%	97.81%	97.81%
2018	79.76%	95.01%	95.70%	96.72%	97.07%	97.93%	**98.10%**	97.96%	97.71%	97.63%	97.55%



(a) 2014



(b) 2015

Fig. 6 The experimental result of our approach with different α in 2014 and 2015

5.4 Investigation to RQ4

Motivation As mentioned above, we adopt tenfold cross-validation to train classifiers. In such a way, the training set and the test set may belong to the same year. However, in a real scenario, we actually need to predict the labels of newly arrived samples through historical samples. There must be a gap between the training set (historical samples) and the test set (newly arrived samples) with respect to features. Thus, the effectiveness of our method may be impacted. In this RQ, we attempt to investigate the effectiveness of our method by modeling a real scenario.

Approach In this experiment, to simulate the real scenario, we leverage the samples from the last year to predict the labels of the samples from the next year. As shown in Table 2, the details for each year in the dataset have been displayed. Therefore, we can directly use the samples from the last year to build classifiers. For example, if we selected the samples from 2017 as the training set, the samples from 2018 will be the test set. In other years, we repeat this way to select the training set and the test set. Notably, we do not report the results of the samples from 2011 since there is no sample from 2010. Meanwhile, we compare the results of our method with those of the baseline methods.

Results Table 7 presents the experimental result of our approach and the baseline methods in terms of precision, recall, F1-score, and accuracy, respectively.

As seen from the table, we can observe that the F1-score achieved by our approach is higher than all baseline methods on most of the sub-datasets, namely, 2013, 2014, 2016 and 2018. This may be due to the small number of samples on each training set. Specifically, we adopt a CNN-based model to implement the API call sequence-based classifier. Since the CNN-based model usually needs a large number of samples for training, our approach performs poorly on some sub-datasets, such as 2012. It is well known that improving the result of a metric may be at expense of another metric when solving a software engineering problem. For example, MalDozer achieves the highest recall on most of the sub-datasets but the precision is low, especially in 2014, MalDozer achieves 100.00% in terms of recall but 72.75% in terms of precision. Conversely, in the table we can observe that the precision

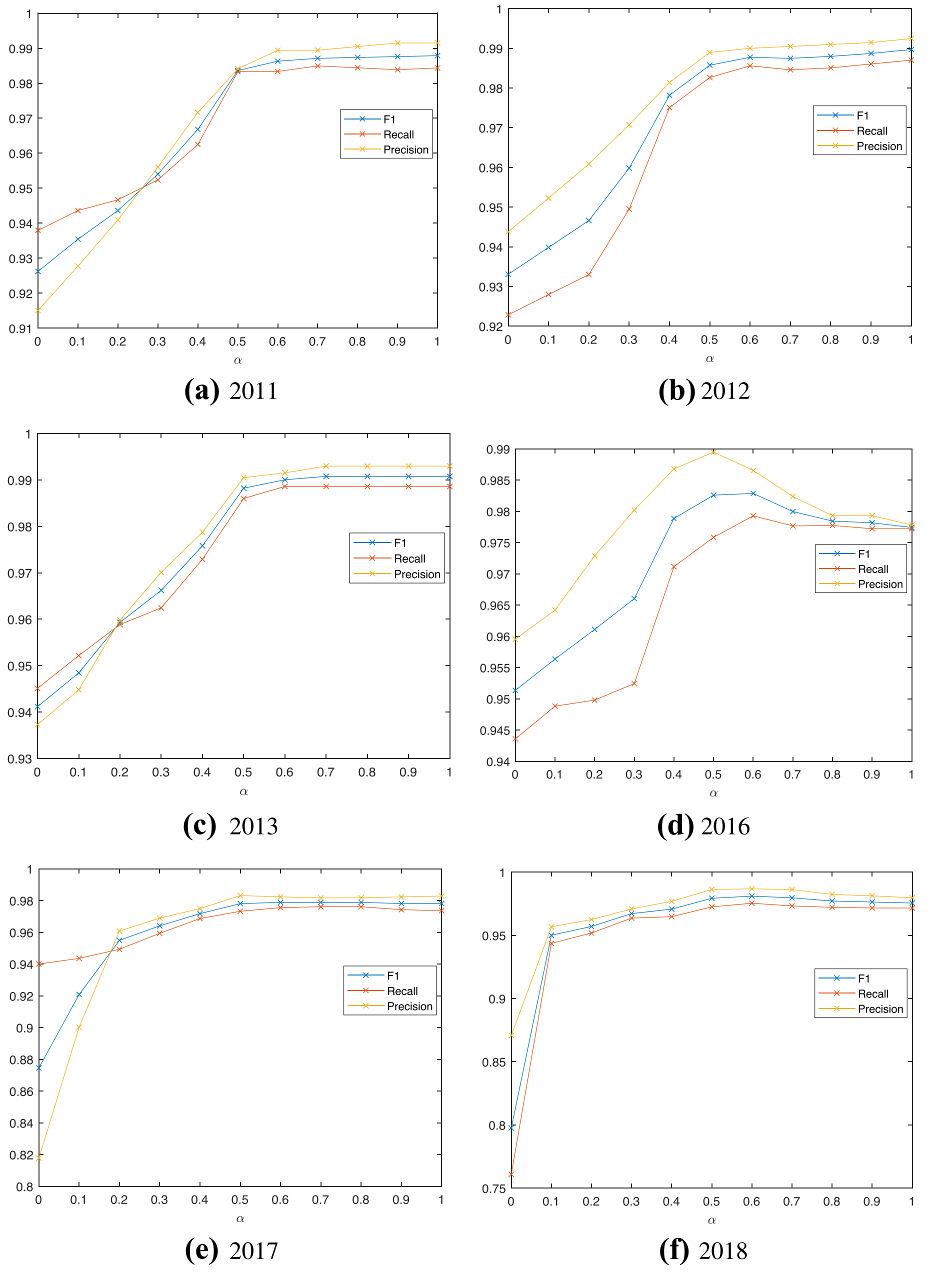


Fig. 7 The experimental result of our approach with different α in rest years

achieved by our approach is high on most of the sub-datasets, but the recall achieved by our method is relatively low. However, as mentioned before, α impacts the final result of our approach, i.e., our approach can achieve different results in terms of recall under

Table 7 The experimental result of our approach and baseline methods to perform classification when the training samples and the test samples belong to different years

Test set	Our approach				Malscan				MalDozer				DroidAPIMiner				Drebin			
	P%	R%	F1%	A%	P%	R%	F1%	A%	P%	R%	F1%	A%	P%	R%	F1%	A%	P%	R%	F1%	A%
2012	**97.90**	72.55	83.34	85.03	89.82	88.88	89.35	**89.06**	76.13	**96.58**	85.14	83.69	84.80	95.89	**90.01**	89.01	93.21	73.72	82.33	83.67
2013	91.92	95.60	**93.73**	**93.43**	82.41	**97.90**	89.49	88.19	93.33	70.15	80.10	83.03	90.23	90.50	90.36	90.09	**93.35**	93.30	93.32	93.13
2014	**99.27**	82.90	**90.35**	**90.78**	95.26	74.02	83.30	84.56	72.75	**100.00**	84.23	82.04	95.41	85.13	89.98	90.23	92.77	80.54	86.22	86.76
2015	**95.08**	46.28	62.25	71.73	90.65	61.66	73.40	77.48	61.42	**95.75**	74.84	68.05	70.57	55.47	62.12	65.89	93.38	69.12	**79.44**	**82.07**
2016	**98.35**	82.99	**90.02**	**90.97**	96.22	76.13	85.01	86.83	71.34	**98.21**	82.65	78.99	87.53	61.17	72.01	75.11	96.91	77.08	85.87	87.77
2017	**96.36**	66.80	78.91	82.59	91.90	77.83	**84.28**	**85.85**	73.02	**94.98**	82.56	79.44	86.58	36.04	50.89	66.05	91.19	74.51	82.01	84.09
2018	**96.52**	89.79	**93.03**	**93.59**	95.17	90.94	93.01	93.48	82.88	**96.65**	89.24	87.80	93.94	27.36	42.38	64.50	88.83	83.34	86.00	87.08

different α. Thus, if we employ a suitable α, our approach may achieve a good result in terms of recall.

In addition, we can observe that the F1-score achieved by our approach, Malscan, MalDozer and Drebin is all above 62% on all sub-datasets, but the F1-score achieved by DroidAPIMiner varies from 42.38% to 90.01%. It indicates that our approach, Malscan, MalDozer and Drebin are more stable than DroidAPIMiner. The potential reason is that our approach, Malscan and MalDozer leverage semantic information contained in API call sequences to detect malware and such semantic information may vary slightly between different years. Although both Drebin and DroidAPIMiner use the permission and API call features for malware detection, they neglect the role of sequence information in API calls.

In summary, compared with the baseline methods, our approach has relatively good performance on identifying newly arrived samples by leveraging historical samples.

5.5 Investigation to RQ5

Motivation In this study, we use permission and API call sequence features to detect malware and conduct experiments to validate the effectiveness of our approach in previous RQs. For further understanding and explaining the classification results of our approach, in this RQ, we attempt to investigate the most important permissions and API call subsequences for identifying malware.

Approach First, we combine all the sub-datasets to create a dataset to train a permission-based classifier and an API call sequence-based classifier. Then we use these classifiers to obtain the most important permissions and API call sub-sequences, respectively. Specifically, in our permission-based classifier, each dimension of a feature vector is corresponding to a permission. Therefore, we use the weight of each permission in the RF classifier to represent the importance of the permission. For the API call sequence-based classifier, we try to extract key sub-sequence by de-convolution. As mentioned at Sect. 3.4.2, our CNN model applies the convolution and max pooling layers to extract features from the API call sequence of an application, and classifies the application based on these features. During these layers, the feature can be traced back to find the corresponding sub-sequence. Meantime we regard the possibility of the feature belonging to malware class as the importance of the sub-sequence. Finally, we can select the top-10 important permissions and API call sub-sequences.

Result Table 8 presents the top-10 important permissions and API call sub-sequences on the whole dataset. It is worth noting that the “THIRD_PART” permission refers to unofficial permissions, such as custom permissions.

The table shows that the “READ_PHONE_STATE” is the most important permission to identify malware. The reason may be that malicious applications usually request “READ_PHONE_STATE” to access the information of the phone which is one of the main targets. In addition, we can find that most of these top-10 permissions are related to access personal information. For example, “GET_TASKS” allows an application to obtain information about currently or recently running tasks, “ACCESS_COARSE_LOCATION” and “ACCESS_FINE_LOCATION” allow an application to access approximate and precise location, respectively. This indicates that many malicious applications attempt to

Table 8 The top-10 permissions and API call sub-sequences

Rank	Permission	API call sub-sequence
1	READ_PHONE_STATE	LayoutParams.addRule, View.init, View.init, View.setLayoutParams, View.setVisibility
2	GET_TASKS	Context.startService, Service.onCreate, Service.onDestroy
3	SYSTEM_ALERT_WINDOW	String.getBytes, String.getBytes, String.getBytes, Object.init, Float.parseFloat
4	ACCESS_WIFI_STATE	String.getBytes, String.getBytes, String.getBytes, String.length, String.length
5	INSTALL_SHORTCUT	StringBuilder.toString, View.findViewById, LayoutParams.init, View.init, View.setLayoutParams
6	ACCESS_COARSE_LOCATION	IntentFilter.addDataScheme, Service.onCreate, Service.onDestroy
7	THIRD_PART	Service.finalize, Service.onCreate, Service.onDestroy
8	ACCESS_FINE_LOCATION	Service.init, Service.onCreate, Service.onDestroy
9	WRITE_EXTERNAL_STORAGE	String.getBytes, String.getBytes, String.getBytes, ByteArrayOutputStream.write, System.currentTimeMillis
10	RECEIVE_BOOT_COMPLETED	TextView.setCompoundDrawablesWithIntrinsicBounds, Service.init, Service.onCreate, Service.onDestroy

access and steal this personal information, and these permissions have higher impact on the classification result and require more attention from developers.

6 Threats to validity

In this paper, we combine the API call sequence and permission features to classify the Android applications into malicious and benign. Although the experimental results show that the proposed method has good performance in resolving this problem, there are still some limitations from the following aspects.

External validity The main threat to external validity may be the difference between API versions since some APIs will be changed during the iteration of API versions. For example, some APIs may be modified, replaced or removed, and new APIs will be added. In our approach, we apply the Word2Vec technique to represent APIs, and each API can be transformed into a vector with semantic information. By this way, some APIs with similar functions may be close to each other in the vector space. In addition, compared with all the APIs, the number of changed APIs is relatively small. Thus, this bias is minimized.

Internal validity The potential threat to internal validity may be the impact of α. As we mentioned before, α can influence the final prediction result. Since our method may obtain the best results under different parameter values of α on different datasets. Thus, it will cause a bias on the scalability of our approach. In our approach, we select two sub-datasets to determine the optimal α, and use the rest sub-datasets to verify its validity. The experimental result shows that our approach can achieve the best or approximate best result when α is set to 0.6. Therefore, this threat can be greatly weakened.

7 Related work

Existing studies have proposed many methods for Android malware detection. In general, they can be divided into static, dynamic, and hybrid.

Static detection is to directly analyze source code or decompiled code for feature extraction and malware detection. For example, Arp et al. (2014) extracted eight features from the decompiled dex file and the manifest file, and then adopted the SVM algorithm to train a classifier for detecting Android malware. Similarly, Peiravian et al. (2013) leveraged some classification methods, such as SVM, Decision Tree (Quinlan, 1986) and Bagging (Burguera et al., 2011), to train classifiers to detect malware by combining the permission and API call features of each application. In addition, Alazab et al. (2020) proposed a grouping strategy which divides API calls into *Ambiguous group, risky group, and disruptive group* and then selected the most valuable API calls to maximize the identification of Android malicious applications. Different from the aforementioned studies, Wang et al. (2014) used only the permission feature for malware detection. They conducted an in-depth investigation through the loop forward selection and the principal-component analysis. They analyzed the riskiness of different permissions and selected the top-k permissions as features for detecting malware. Recently, some researchers combined other domain knowledge to detect Android malware. For instance, Wu et al. (2019) developed an Android malware detection system (Malscan), which combines the concept of social network. Malscan transformed the API call graph into a social network, and then built a SVM-based classifier by performing social network-based centrality analysis as the semantic features.

Although the static detection is widely used and proven its effectiveness, the accuracy of these static methods is relatively low since these methods do not take into account the semantic information contained in the features. To solve this challenge, some researchers attempted to design dynamic methods to capture the runtime information by running applications. For example, Han et al. (2014) collected the execution traces by running programs in a virtual environment. After extracting the instruction sequences from the primary performed blocks, they transformed instruction sequences into matrix images. Then they adopted the similarity calculation to distinguish malicious applications from benign applications. Besides, Karbab et al. (2016) developed a dynamic malware detection system by generating a summary based on the dynamic analysis of malware samples related to existing known malware. Meantime, they leveraged NLP to solve the problem of dynamic code changes. Although the dynamic methods can capture the runtime features to help detect malware, they usually have some inevitable problems, such as low coverage rate and high computational cost.

Besides, some researchers proposed hybrid detection algorithms which combine static detection and dynamic detection to integrate their advantages. For example, Xu et al. (2016) used the dynamic analysis to record system call sequences as the dynamic feature and combined some static features. Then these features were represented by vectors and the SVM algorithm was leveraged to train a classifier for malware detection. In addition, Garg et al. (2019) also used the hybrid analysis to detect malware. They trained multiple classifiers by using static and dynamic features respectively, and implemented the final classification based on different ensemble techniques.

Recently, researchers proposed some methods based on neural networks to detect malware. For example, MalDozer (Karbab et al., 2018), which was based on API call sequences, transformed each API call into a vector through Word2Vec and adopted a CNN-based model for classification. Besides, Vasan et al. (2020) tried to directly convert raw malicious binary code into colored images, and trained a CNN-based model to distinguish malicious applications from benign applications. In this paper, we proposed a novel static detection method combining the API call sequence and permission features. But unlike the existing studies, we respectively trained an API call sequence-based classifier and a permission-based classifier, and then designed a linear strategy to combine these two classifiers to distinguish malicious applications from benign applications.

8 Conclusion and future work

In this paper, we propose an effective Android malware detection method based on permission and API call sequence features. More specially, we first leverage a decompiling tool to extract the permission feature and the API call sequence feature. After preprocessing, we adopt two methods to transform permissions and API calls into vectors, respectively. Then we apply RF and CNN to learn a permission-based classifier and an API call sequence-based classifier, respectively. Finally, we linearly combine the outputs of these two classifiers to predict the labels of new applications. For evaluating our approach, we conduct experiments and compare the results with four baseline methods. Experimental results indicate that our approach can effectively detect malware and outperform the baseline methods. In addition, when the training samples and the test samples belong to different years, our approach can also achieve relatively good detection performance.

In the future, we will collect more datasets to evaluate the effectiveness and generalizability of our method, and try to design new methods to improve the classification results. Furthermore, in this paper, the problem we study is a binary classification problem, which means we only predict whether the application is malicious or not. In the future, we will design a multi-classification model to detect which malicious family the malware belongs to.

Funding This work was supported in part by the Natural Science Foundation of Zhejiang Province under Grant LY21F020020, in part by the National Natural Science Foundation of China under Grant 61902096, and in part by Key Project of Science and Technology of Zhejiang Province under Grant 2020C01165.

Declarations

Conflict of interest The authors declare no competing interests.

References

Aafer, Y., Du, W., & Yin, H. (2013). Droidapiminer: Mining API-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems* (pp. 86–103). Springer.

Alazab, M., Alazab, M., Shalaginov, A., Mesleh, A., & Awajan, A. (2020). Intelligent mobile malware detection using permission requests and API calls. *Future Generation Computer Systems*, *107*, 509–521. Publisher: Elsevier.

Allix, K., Bissyand, T. F., Klein, J., & Le Traon, Y. (2016). Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)* (pp. 468–471). IEEE.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., & Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket. *NDSS, 14*, 23–26.

Bottou, L. (1998). Online learning and stochastic approximations. *On-line Learning in Neural Networks, 17*(9), 142.

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32. Publisher: Springer.

Burguera, I., Zurutuza, U., & Nadjm-Tehrani, S. (2011). Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (pp 15–26).

Garg, S., & Baliyan, N. (2019). A novel parallel classifier scheme for vulnerability detection in android. *Computers & Electrical Engineering*, *77*, 12–26. Publisher: Elsevier.

Han, K., Kang, B., & Im, E. G. (2014). Malware analysis using visualized image matrices. *The Scientific World Journal*, *2014*. Publisher: Hindawi.

Harris, D., & Harris, S. (2010). *Digital design and computer architecture*. Morgan Kaufmann.

Herley, C. E., Keogh, B. W., Hulett, A. M., Marinescu, A. M., Williams, J. S., & Nurilov, S. (2015). Spyware detection mechanism. *Google Patents*.

Hui, T., Tang, X., & Loy, C. C. (2021). A lightweight optical flow CNN - revisiting data fidelity and regularization. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 43*(8), 2555–2569.

Jerbi, M., Dagdia, Z. C., Bechikh, S., & Said, L. B. (2020). On the use of artificial malicious patterns for android malware detection. *Computers & Security*, *92*, 101743. Publisher: Elsevier.

Karbab, E. B., Debbabi, M., Alrabaee, S., & Mouheb, D. (2016). DySign: Dynamic fingerprinting for the automatic detection of android malware. In *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)* (pp. 1–8) IEEE.

Karbab, E. B., Debbabi, M., Derhab, A., & Mouheb, D. (2018). MalDozer: Automatic framework for android malware detection using deep learning. *Digital Investigation*, *24*, S48–S59. Publisher: Elsevier.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. Preprint retrieved from http://arxiv.org/abs/1412.6980

Kohavi, R., et al. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI* (vol. 14, pp. 1137–1145). Montreal, Canada.

Landwehr, C. E., Bull, A. R., McDermott, J. P., & Choi, W. S. (1994). A taxonomy of computer program security flaws. *ACM Computing Surveys (CSUR)*, *26*(3), 211–254. Publisher: ACM New York, NY, USA.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998a). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324. Publisher: IEEE.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*(11), 2278–2324.

Mercaldo, F., Visaggio, C. A., Canfora, G., & Cimitile, A. (2016). Mobile malware detection in the real world. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)* (pp. 744–746) IEEE.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013a). Efficient estimation of word representations in vector space. Preprint retrieved from http://arxiv.org/abs/1301.3781

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems* (pp. 3111–3119).

Nash, T. (2005). *An undirected attack against critical infrastructure*. US-CERT Control Systems Security Center: Technical Report.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research*, *12*, 2825–2830. Publisher: JMLR.org

Peiravian, N., & Zhu, X. (2013). Machine learning for android malware detection using permission and API calls. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence* (pp. 300–305). IEEE.

Pektaş, A., & Acarman, T. (2020). Deep learning for effective Android malware detection using API call graph embeddings. *Soft Computing*, *24*(2), 1027–1043. Publisher: Springer.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*(1), 81–106. Publisher: Springer.

Shao, Y., Chen, Q. A., Mao, Z. M., Ott, J., & Qian, Z. (2016). Kratos: Discovering inconsistent security policy enforcement in the android framework. In *NDSS*.

Srinivasa-Desikan, B. (2018). *Natural language processing and computational linguistics: A practical guide to text analysis with Python, Gensim, spaCy, and Keras*. Packt Publishing Ltd.

Vapnik, V. (2013). *The nature of statistical learning theory*. Springer Science & Business Media.

Vasan, D., Alazab, M., Wassan, S., Naeem, H., Safaei, B., & Zheng, Q. (2020). IMCFN: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networks, 171*, 107138. Publisher: Elsevier.

Wang, W., Wang, X., Feng, D., Liu, J., Han, Z., & Zhang, X. (2014). Exploring permission-induced risk in android applications for malicious application detection. *IEEE Transactions on Information Forensics and Security*, *9*(11), 1869–1882. Publisher: IEEE.

Wu, D. J., Mao, C. H., Wei, T. E., Lee, H. M., & Wu, K. P. (2012). Droidmat: Android malware detection through manifest and API calls tracing. In *2012 Seventh Asia Joint Conference on Information Security* (pp 62–69). IEEE.

Wu, Y., Li, X., Zou, D., Yang, W., Zhang, X., & Jin, H. (2019). MalScan: Fast market-wide mobile malware scanning by social-network centrality analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 139–150). IEEE

Xu, L., Zhang, D., Jayasena, N., & Cavazos, J. (2016). HADM: Hybrid analysis for detection of malware. In *Proceedings of SAI Intelligent Systems Conference* (pp. 702–724). Springer.

Young, A., & Yung, M. (1996). Cryptovirology: Extortion-based security threats and countermeasures. In *Proceedings 1996 IEEE Symposium on Security and Privacy* (pp 129–140). IEEE.

Zhou, Y., & Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy* (pp 95–109). IEEE.

Zhu, H. J., Jiang, T. H., Ma, B., You, Z. H., Shi, W. L., & Cheng, L. (2018). HEMD: A highly efficient random forest-based malware detection framework for Android. *Neural Computing and Applications*, *30*(11), 3353–3361. Publisher: Springer.

Xin Chen received the Ph.D. degree in software engineering in 2018 from the School of Software, Dalian University of Technology, in Dalian, China. He is currently an Associate Professor of Hangzhou Dianzi University, in Hangzhou, China. His research interests include mining software repositories, search based software engineering, and evolutionary computation. He is a member of the CCF and the IEEE. He is also a member of Technical Committee of Software Engineering CCF (TCSE CCF).

Haihua Yu is currently a master candidate in Hangzhou Dianzi University. His main research interests include mining software repositories and intelligent software engineering.

Dongjin Yu is currently a professor at Hangzhou Dianzi University, China. His research efforts include intelligent software engineering, data engineering and service computing. He is the director of Big Data Institute, and the director of Computer Software Institute of Hangzhou Dianzi University. He is a member of IEEE, and a senior member of China Computer Federation (CCF). He is also a member of Technical Committee of Software Engineering CCF (TCSE CCF) and a member of Technical Committee of Service Computing CCF (TCSC CCF).

Jie Chen is an Assistant Professor in the College of Computer Science at Hangzhou Dianzi University, China. She received the PhD degree from the Lab of Internet Software Technologies, Institute of Software Chinese Academy of Sciences (ISCAS), in Beijing, China, in 2016. She was a visiting scholar in the Department of Computer Science, University of Massachusetts Amherst from September 2012 to September 2013. Her research interests are in software process simulation, resource scheduling and code analysis.

Xiaoxiao Sun received her Ph.D. degree from Zhejiang University in 2017. She is now a lecturer at Hangzhou Dianzi University. Her research efforts include spatio-temporal data analysis, business process management and big data.