COMPUTERS & SECURITY 84 (2019) 376–392

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

A feature-hybrid malware variants detection using CNN based opcode embedding and BPNN based API embedding

Jixin Zhang[a,b], *Zheng Qin*[a,*], *Hui Yin*[a,c], *Lu Ou*[a], *Kehuan Zhang*[b]

[a] *College of Information Science and Engineering, Hunan University, Changsha, China*
[b] *Department of Information Engineering, Chinese University of Hong Kong, Hong Kong, China*
[c] *College of Computer Science and Technology, Changsha University, Changsha, China*

ARTICLE INFO

Article history:
Received 12 November 2018
Revised 9 April 2019
Accepted 12 April 2019
Available online 17 April 2019

Keywords:
API call
Back-propagation neural network
Convolutional neural network
Feature-hybrid
Malware variants detection
Malware family classification
Opcode

ABSTRACT

Being able to detect malware variants is a critical problem due to the potential damages and the fast paces of new malware variations. According to surveys from McAfee and Symantec, there is about 69 new instances of malware detected in every minutes, and more than 50% of them are variants of existing ones. Such a large volume of diversified malware variants has forced researches to investigate new methods based on common behavior patterns using machine learning.

However, such methods only use single type of features such as opcode, system call, etc., which faces several drawbacks: Firstly, the methods lose a part of useful information since different types of features show different characteristics of malware. This severely limits detection precision and recall. Secondly, the accuracy and the speed (as a trade-off) of such methods fail to meet users' expectation. Thirdly, the precise classification of malware families is still a hard problem and is also important in malware analysis.

In this work, we propose a feature-hybrid malware variants detection approach which integrates multi-types of features to address these challenges. We first represent opcodes by a bi-gram model and represent API calls by a vector of frequency, then we use principal component analysis to optimize the representations to improve the convergence speed, the next we adopt a convolutional neural network and a back-propagation neural network for opcode based feature embedding and API based feature embedding respectively, and finally we embed these features to train a detection model by using softmax.

Theoretical analysis and real-life experimental results show the efficiency and optimization of our approach which achieves more than 95% malware detection accuracy and almost 90% classification accuracy of malware families. The detection speed of our approach is less than 0.1 s.

* Corresponding author at: College of Information Science and Engineering, Hunan University, Changsha, China.
E-mail addresses: 478238376@qq.com (J. Zhang), zqin@hnu.edu.cn (Z. Qin).

https://doi.org/10.1016/j.cose.2019.04.005
0167-4048/

1. Introduction

Malware is one of the major security threat today. With large amount of sophisticated tools readily available over the Internet, malware now are able to quickly evolve into many different variants and evade existing detection mechanism, rendering the ineffectiveness of traditional signature based malware detection systems.

As a survey in Gandotra et al. (2014), McAfee catalogs over 100,000 new malwares every day, which means about 69 new instances every minute, or more than one sample per second. Symantec stated in its reports (Symantec, 2017) that more than 50% of new malwares are actually variants of existing ones.

This has forced researchers to come up with modern malware detection techniques which mainly search for similarities between unknown and previously known malware samples, because malware variants from the same families still share many common behavioral patterns which reflect their origins and purposes.

For a typical architecture of modern malware detection system, when a user is going to install an unknown application onto his/her device, the detection client will upload the binary of such application to cloud servers for security check. After receiving the binary, the detection server will first use tools such as PolyUnpack (Royal et al., 2016), PEiD[1], etc., to detect packers and unpack the code if necessary. Then the detection system will launch disassembly or reverse engineering tools to extract one type of data (such as opcode, system calls, etc.), and train a classification model for detection.

However, such modern malware detection systems are also facing some challenges of their own. First of all, a single type of data, such as byte code, operation code (opcode), API call, system call, etc., can only cover a part of characteristics of malware, that means it loses some information in malware which will be useful to represent behavioral patterns as well. In this way, the accuracy is severely limited (Wang et al., 2018a). Secondly, the accuracy and the speed (as a trade-off) of such systems fail to meet users' expectation. Thirdly, the precise classification of malware families is still a hard problem and is also important in malware analysis.

To address these challenges, we aim to propose a novel malware variants detection technique which integrates different types of data and fuses multi-features to improve detection accuracy while retaining detection speed. However, it remains two major problems. One is which type of data should be chose to make a feature integration that can indeed improve accuracy. The other one is how to integrate different types of data since different types of data are unstructured and heterogenous. Santos et al. (2013c) proposed a hybrid malware variant detector called OPEM, which utilizes a set of data obtained from both static and dynamic analysis of executables. However, their method take disadvantages both of static analysis and dynamic analysis.

In this paper, we propose to choose two types of data: one is opcodes and the other is API calls. Since opcodes are widely used to represent binaries, and they are treated as a fine gained semantic representation of binaries, we choose opcodes for our approach. Although byte codes are also a semantic representation of binaries, they may be treated as "noisy opcodes" since the byte codes not only contain opcodes but also contain operands (noise). We also choose API calls for our approach since they can be extracted together with opcodes by disassembly tools, such as IDA[2], W32dasm[3], etc.. This does not take extra time consumption of data preparation. What's more, API calls show high-level behaviours of binaries as a supplement of opcodes because they represent binaries in different levels and angles. Although system calls also show behaviours of binaries like API calls, they have to be extracted in runtime by sandboxes, which costs a lot of time of data preparation. What's the worse, these system calls sometimes lead a mistake when a malicious binary checks the operating environment and hides its malicious behaviours.

Since the information in different types of data are unstructured and are organized in different ways, building a model to cover the information in different data are not an easy work. A direct way to handle this case is to build different models for each type of data to make decisions respectively, and take an overall consideration of the decisions from these models by adopting ensemble learning. However its accuracy is limited by the decisions made by the worst model. To avoid the bias, here we integrate them in feature-level to generate a unified model. The idea is to use different neural networks to train different types of data and extract their features, then use another classifier to merge these features and make an overall decision. We integrate opcodes and API calls by adopting appropriate neural networks since neural networks are easier to achieve high accuracy compared with statistical models. Our approach first uses opcode bi-gram model for opcode semantic representation and uses a vector of API call frequency for API behaviour representation, and then optimizes the representations by using principal component analysis to increase the convergence speed. The next it adopts a convolutional neural network (CNN) to train a model for opcode embedding and adopts a back-propagation neural network (BPNN) for API embedding. Finally it combines with these embedded features by a serial manner and trains a malware detection model by a softmax classifier. Our approach fuses opcodes and API calls to generate a unified model to make full use of the information in opcodes (fine gained) and API calls (high-level), and the model is self-adaptive to an optimized decision.

Due to the different data structures and the different meanings of opcode bi-gram model and API call frequency, we generate our hybrid features by using structure-adaptive neural networks to train high-level features respectively, and embed them by a serial manner. The high-level features are extracted from the last layer of the neural networks that they can be used to represent malware or benign instead of opcode bi-gram model and API call frequency. Besides, since the high-level features are represented by one-dimensional vectors (as the same data structure), they will be easily integrated. The

[1] https://forensicswiki.org/wiki/PEiD.

[2] https://www.hex-rays.com/products/ida/.

[3] https://www.softpedia.com/get/Programming/Debuggers-Decompilers-Dissasemblers/WDASM.shtml.

high-level features can be treated as a set of common features shared by the instances in the same class, Since the features of the instances in the same class will be closer in feature space by training with gradient descent method.

Since the opcode is a fine-gained representation of executables, the opcode bi-gram model can represent the inner relations among opcodes to capture more information. Many researchers have proposed opcode bi-gram based approaches for representing executables, such as Santos et al. (2013a), Kang et al. (2016), etc.. Since the API call is a high-level representation of executables, the vector of API call frequency can distinguish the distributions of behaviours in malware and benign. Because the data structures of the opcode bi-gram model and the vector of API call frequency are different, so we use two different neural networks to train their features respectively. Since the opcode bi-gram model is a 2-D matrix, the relevant elements in this matrix are crossed (clustered), and its feature dimensions is very large, the CNN is applicable for training in this case by computing weighted dot product with convolutional cores and reducing dimensions with pooling cores. Since the vector of API call frequency is 1-D structured, the elements in this vector are relatively independent, and its feature dimensions is small, the BPNN is applicable for fast training in this case by directly using Sigmoid function. When the neural networks have trained features for opcodes and API calls, a softmax classifier can easily train a classification model for malware detection.

Our approach is applicable for addressing the problem of heterogeneous data integration by training from heterogeneous network. For malware detection, our approach integrate heterogeneous data representation (opcode bi-gram, API call frequency) and generate unified hybrid features to improve detection precision and detection recall.

Contributions. The main contributions of this paper are summarized as follows:

1. To effectively and efficiently classify malicious executables and legitimate ones, we proposed our feature-hybrid malware variants detection technique, which embeds opcodes and API calls by using a convolutional neural network and a back-propagation neural network. When comparing with the opcode based method and the API call based method respectively, our approach performs much better.
2. To represent opcodes and API calls in binaries, we propose to use opcode bi-gram model and a vector of API call frequencies. In order to improve the convergence speed of neural networks for feature embedding, we adopt principal component analysis to optimize these representations. For feature fusion, we merge such embedded features which are extracted by neural networks from opcodes and API calls, and train an decision model with the merged features.
3. We implement a prototype and evaluated with large scale of real world data sets. Evaluation results shows that our approach can achieve more than 95% detection precision when detecting binaries from Windows platform. Overall, when comparing with the state-of-art methods, our approach can significantly improve detection accuracy while retaining detection speed, which we believe that is very attractive when facing the large volume of malware variants coming out everyday. In addition, we also implement our approach to classify malware families and achieve almost 90% classification accuracy.

Paper organizations. The remaining of this paper is organized as follows. Section 2 presents the related works. Section 3 introduces our feature-hybrid malware variants detection technique. Section 4 presents experimental results. Section 5 shows the limitations and Section 6 shows the conclusions.

2. Related works

2.1. Opcode based method

Some researchers prefer to adopt opcode based static analysis. The opcodes will be embedded into a 1-D or 2-D vector and then sent to classifiers later to classify malicious programs and legitimate ones. Santos et al. (2013a) proposed a data mining technique to mine the 2-tuple opcodes similarities. Zhang et al. (2018a) proposed to build a opcode graph and extract its topology features to detect Android malware. Kang et al. (2016) presented an n-opcode analysis based approach that utilizes machine learning to classify Android malware. de la Puerta et al. (2017) proposed to adopt several machine learning methods to detect the opcode vectors. McLaughlin et al. (2017) used a opcode embedding matrix for representation and used a n-gram based CNN model (Kalchbrenner et al., 2014) to train and classify Dalvik opcode sequences. However, their embedding matrix is inefficient while the sequences is very long. Zhang et al. (2016a) proposed to convert opcodes into 2-D matrix, and adopted convolutional neural networks (CNN) to train the 2-D opcode matrix for malware recognition. Yan et al. (2018) proposed to convert Android opcode into 2-D gray image with fixed size and adopt CNN to train and detect Android malware.

2.2. Byte code based method

Some researchers prefer to use byte code to represent binaries. Nataraj et al. (2011) proposed a method for visualizing and classifying Windows malware binaries as gray-scale images. They converted executables into gray images, and then searched the texture similarities among the signatured samples. However, their detection speed is too slow to implement in practice, due to the similarity search can only be performed in a serial manner which would incur a long delay facing the large volume of detection set. Raff et al. (2017) presented a method that uses convolutional neural networks with byte n-gram to achieve good performance, but no more theoretical explanation or technical detail about their method. Kim et al. (2018) proposed a transferred deep-convolutional generative adversarial network for malware detection, which generates fake malware and learns to distinguish it from real malware. The trained discriminator passes down the ability to capture malware features to the detector.

2.3. API call based method

Some researchers propose to use API calls to represent executables. Fan et al. (2018) proposed to construct frequent subgraphs of API calls to represent the common behaviors of malware samples that belong to the same family. Patanaik et al. (2012) checked whether a target's API call dependency follows the same dependency of the signatured malware. Huang et al. (2014) analyzed the user interface component associated with the top level function and found the mismatch of the two to detect stealthy behaviour. However, in some malware cases (such as some virus), we cannot extract any API calls, that leads the API call based methods cannot work.

2.4. System call based method

There are also some researchers propose to use system calls. Rieck et al. (2011) proposed to automatically identify classes of malware with similar sequential system calls and assign unknown malware to these discovered classes. Xu et al. (2016) implemented graph-based representation for system calls, then used the graph kernels to compute pair-wise similarities and fed these similarity measures into a support vector machine for classification. Kolbitsch et al. (2009) proposed to build a graph of data flows among system calls, and then use graph matching for malware detection. However, these methods have to capture system calls in runtime by sandboxes, which costs a lot of time of data preparation. What is the worse, the system call based methods sometimes make mistakes when a malicious executable checks the sandboxes and hides its malicious behaviours.

2.5. Control data flow graph based method

These researchers extract the control flow or data flow of binaries and then use graph matching or subgraph similarity searching methods to label unknown samples according to known malware. Tian et al. (2018) proposed to utilize class-level dependence graph and method-level call graph to represent an application, and extracted static behaviour features to detect Android malware. Cesare et al. (2014a) proposed a technique that performs similarity searching of sets of control flow graphs. Martín et al. (2017) used third-party calls to bypass the effects of obfuscation, and then combined with clustering and multi-objective optimisation to classify third-party call groups. However, their methods cost much more detection time when comparing / matching graphs in a serial manner.

2.6. File via file graph based method

These approaches extract the relationships between files and other entities, such as hosts, domains, etc. to build a graph, and then use belief propagation to label the unknown nodes according to the labeled nodes. Tamersoy et al. (2011) proposed to generate file via file relationships according to the interactions between files and machines, and then adopted belief propagation methods to assign scores to every unlabeled file node. If the score of a file node is bigger than a threshold, then the file will be treated as a malicious file. Similar to Chen et al. (2015). Stringhiniq et al. (2017) proposed a semi-supervised Bayesian label propagation to propagate the reputation of known files across a download graph that depicts file delivery networks (both legitimate and malicious). However, these file via file graph based methods are behind-time, which means when a malicious binary is detected, many of its copies have already been propagated in the network. Some of the copies may have executed malicious behaviours

2.7. Other method

Besides, Massarelli et al. (2017a) proposed to extract features on resource consumption through detrended fluctuation analysis and correlation, and employed SVM method to classify malware into families. Enck et al. (2014) proposed to simultaneously track multiple sources of sensitive data and identified the data leakage. Data from privacy sensitive sources are automatically labeled (tainted) and labels are transitively applied as sensitive data moves through interprocess messages, program variables and files. Wang et al. (2014) proposed to analyze the permission-induced risk of an individual permission and the risk of a group of collaborative permissions in Android apps to detect Android malware. Later, they (Wang et al., 2018b) proposed to integrate different types of features from Android applications into a feature matrix for Android malware detection. Burguera et al. (2011) collected traces from a large number of real users based on crowd-sourcing and then clustered these traces using k-means to detect malware. Liu et al. (2019) collected and analyzed users' action (API) in Android platform to detect privacy leakage. Shabtai et al. (2012) used machine learning algorithms to continuously monitor device state to differentiate between benign and malicious programs. Yan et al. (2012) proposed a method in which both the Java-level and OS level semantics are reconstructed seamlessly and simultaneously. By staying out of the execution environment and moreover, privilege based attacks can be detected. However, this approach also has a drawback of having very limited code coverage. Oberheide et al. (2008), proposed their malware detection system by integrating several detection engines. However, it can only detect malware while it has known which platform the malicious programs belong to.

3. Methodology

3.1. Overview of our approach

In this section, we propose a feature-hybrid malware variants detection technique by embedding opcode based feature and API call based feature to optimize the performance of malware variants detection. The architecture of our malware detection technique is presented in Fig. 1, which includes five steps: unpacking and disassembly process, representation process, representation optimization process, neural networks based feature embedding process, and feature fusion based training and classification process.

Step 1: Unpacking and disassembly process. Unpacking and disassembly process aims to unpack and disassemble binaries (.exe files) to obtain their opcodes and API calls. Since some binaries are packed by some packing tools, making disassembly

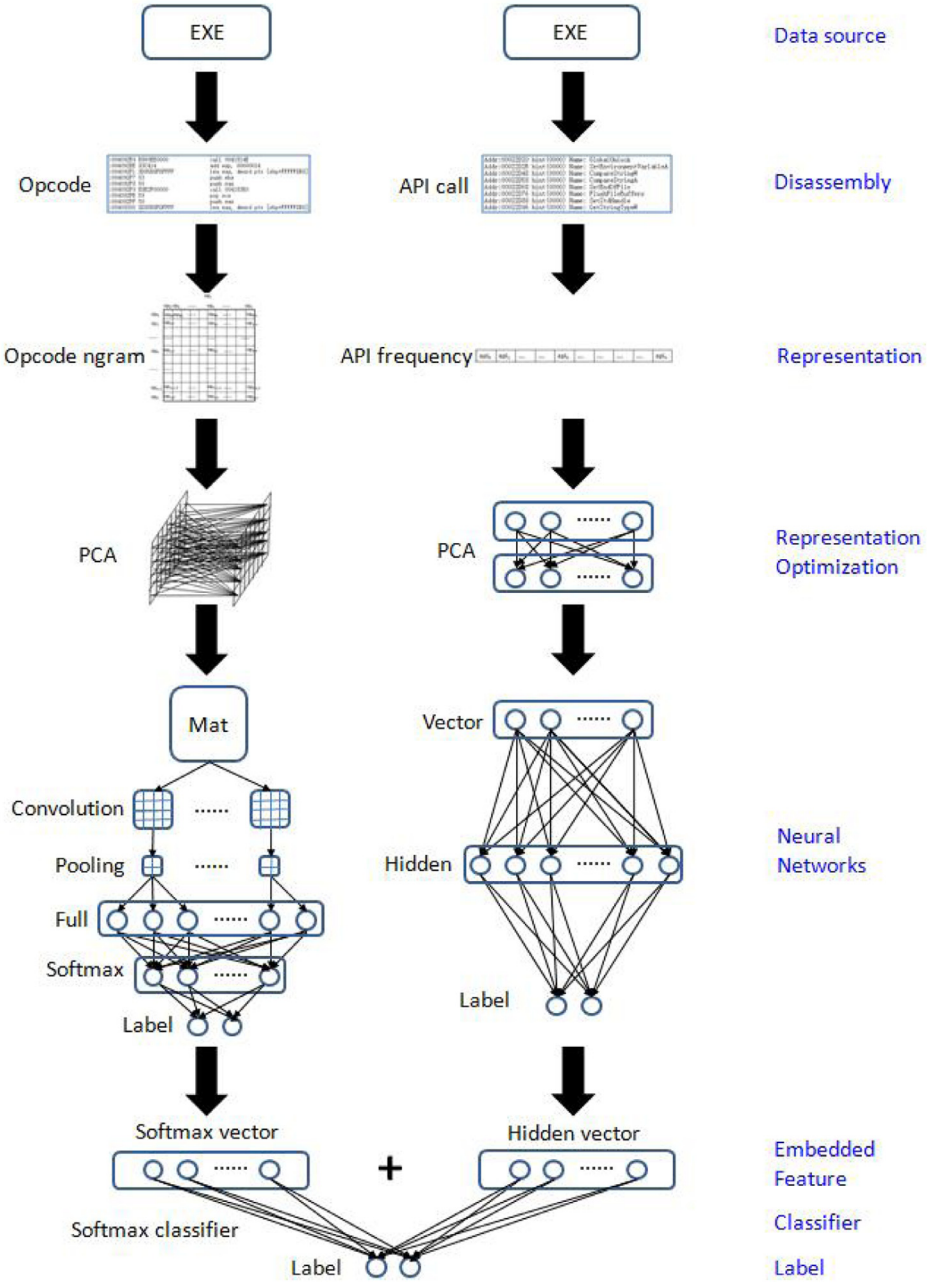

Fig. 1 – The architecture our approach.

harder, we check these binaries and unpack them first by several unpacking tools, allowing further analysis of the binaries, such as ASPack[4], UPX[5], etc.. Unpacking is not necessary if a binary was not packed before. Then we disassemble the unpacked.exe files to generate their.alf files by using W32dasm in order to extract their opcodes and API calls. We do not use many different disassembly tools to avoid distortion of the results. After scanning the.alf files, we built an opcode profile and an API call profile for each binary, where the opcode profile contains a list of opcodes, and the API call profile contains a list of API calls. For example, an opcode profile may contain a list *{call, mov, mov, test, je,....}*, and an API call profile may contain a list *{WriteFile, VirtualQueryEx, UnmapViewOfFile, Sleep....}*.

Step 2: Representation process. Representation process proposes to generate a structured data to represent each binary. The structured data will be treated as an input for the next computations. Once we have obtained opcode profiles and API call profiles, here we use n-gram model to represent opcode data and use a vector of frequency to represent API data, respectively.

Step 3: Representation optimization process. Since we adopt neural networks to train the next feature embedding models, which will cost huge computations through many iterations, we aim to optimize the representations of opcodes and API calls to improve the convergence speed. The representation

[4] http://www.aspack.com.
[5] https://upx.github.io.

optimization process proposes to use principal component analysis (PCA) to improve the performance of neural networks while retaining useful information.

Step 4: Neural networks based feature embedding process. After representation optimization process, we get a PCA-initialized opcode bi-gram matrix and a PCA-initialized API frequency vector which will be sent to two different neural networks. We choose a convolutional neural network (CNN) to train a feature embedding model for opcodes and choose a back-propagation neural network (BPNN) to train a feature embedding model for API calls. The hidden layers before the last classifiers of the two neural networks will be used as embedded features. Here, we use CNN based opcode embedding model and BPNN based API embedding model for feature embedding. The reasons why we use different neural networks in our approach are that: On one hand, the PCA-initialized opcode bi-gram matrix is 2-D structured and its number of dimensions is very large, the CNN architecture is very suitable in this case since it computes the dot product between the entries of the filter and the matrix, produces a 2-D activation map of that convolutional core, and compresses the dimensions by pooling core. The PCA-initialized API frequency vector is 1-D structured and its number of dimensions is small, the BPNN architecture is applicable for fast training since it directly sums over the weighted features in the vector by several Sigmoid functions. On the other hand, the BPNN architecture is unapplicable for training the PCA-initialized opcode bi-gram matrix. It is hardly converged since the number of dimensions of the matrix is too large, so that the sigmoid units are hard to activate. The CNN architecture is unapplicable for training the PCA-initialized API frequency vector, since the dot product used in this case is unreasonable and the number of dimensions in the vector is small while the CNN is more suitable for larger number of dimensions.

Step 5: Feature fusion based training and classification process. Once we have obtain the embedded features, then we use softmax as a classifier to train and detect malware variants. This process combines with the two vectors of embedded features in a series manner and sends them into a softmax classifier. By iterations, it trains a classification model which identifies if a binary is malicious or legitimate.

Processing the above five steps, we can effectively and efficiently detect malware variants. The algorithm representation is presented in Algorithm 1. Besides, since classifying malware families is also important in researches of malware, we change the labels of all of the classifiers in our approach from malware/benign to malware families and retrain a new model for classifying malware families.

Algorithm 1 Feature-hybrid malware variants detection.

Input: A set of sequences of opcodes $OPs = \{OP_1, OP_2, \ldots, OP_k\}$. $OP_j = \{op_1, op_2, \ldots, op_n\}$ is an opcode sequence in *OPs*, where op_i is the ith opcode in the sequence; A set of sequences of API calls $APIs = \{API_1, API_2, \ldots, API_k\}$. $API_j = \{api_1, api_2, \ldots, api_m\}$ is an API call sequence in *APIs*, where api_i is the ith API calls in the sequence.

Output: A vector of confident scores $score = \{s_1, s_2\}$, where s_1 means the probability of malware and s_2 means the probability of benign.

```
function MalwareDetector(OPs, APIs)
    for OP_j in OPs do
        Generate an opcode bi-gram matrix M(op_i, op_j) according to the opcode sequence OP_j.
        Generate a PCA initialized opcode bi-gram matrix PCAMat according to M(op_i, op_j).
        Adopt CNN to train a vector of features u_op,i according to PCAMat.
    end for
    for API_j in APIs do
        Generate a vector of API frequencies V(api_i) according to the API call sequence API_j.
        Generate a PCA initialized API frequency vector PCAVec according to V(api_i).
        Adopt BPNN to train a vector of features u_api,i according to PCAVec.
    end for
    for OP_j in OPs do
        Merge the features u_op,i and u_api,i to generate a vector of hybrid features U = {u_op,1, u_op,2, ..., u_api,1, u_api,2, ...}.
        Adopt Softmax to train U and gain score as a decision.
    end for
end function
```

3.2. *Representation process*

Since opcodes and API calls in binaries are unstructured data, we need to convert them to structured representations. For opcode based representation, because opcode sequences are too long to remember the whole semantic, so we consider the local semantic between an opcode and its neighbors. Here we use n-gram model which is a contiguous sequence of n items to represent opcode sequences. Usually, *n* is less than 3, since the large number of dimensions of opcodes severely limits the training speed and the detection speed, and does not significantly improve the detection accuracy (Kang et al., 2017), (Santos et al., 2013b), we choose *n*=2 to achieve high accuracy while retaining speed. (The number of dimensions is $OpNum^n$ where *OpNum* is the total number of the types of opcodes. The training time cost and the detection time cost exponentially grow when n increases). By statistics, we generate a list of opcode 2-tuples with their probabilities, like *{{mov mov 7.250437828371279E-4}, {mov sub 8.523058960887332E-5}, ...}*. These opcode 2-tuples will be mapped into a matrix, we call it opcode bi-gram matrix, as shown in Fig. 2. The element $op_{i,j}$ in the opcode bi-gram matrix is the probability of the opcode belonging to type *i* that has connected to the opcode belonging to type *j*. The opcode bi-gram matrix can be used to represent a binary since the elements and their neighbors in the opcode bi-gram matrix actually mean the probabilities of clusters of opcodes which are important characteristics of binaries, and the distributions of the probabilities between malicious binaries and legitimate ones are different.

For API based representation, since the API call is a high-level representation of behaviours of binaries, each API call reflect many fine-gained operations. The API calls are more independent with each other compared with the opcodes. Here we use a vector of frequencies of API calls to represent API call sequences. By statistics, we generate a list of API calls with their probabilities, like *{{WriteFile 0.0126582278481013},*

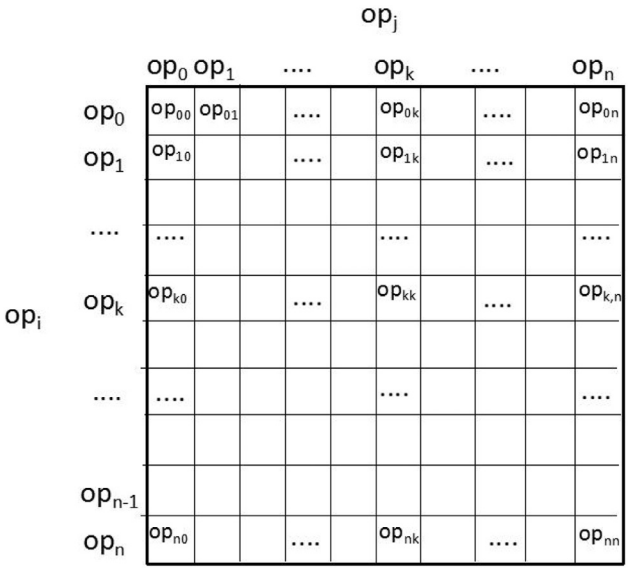

Fig. 2 – The opcode bi-gram matrix.

{*LocalAlloc 0.0253164556962025*}, ...}. These probabilities will be mapped into a vector, we call it API frequency vector. The element api_i in the API frequency vector is the probability of the API belonging to type i. The API frequency vector can be used to represent a binary since the distributions of the probabilities between malicious binaries and legitimate ones are different.

3.3. *Representation optimization process*

Once we have represent opcodes by opcode bi-gram matrix and represent API calls by API frequency vector, then we optimize these representation in order to reduce the inner complexity while retaining the information. Although these representations are effective, they are inefficient due to the complex co-occurrence relations among the elements which increase the computations in training iterations. To reduce the computation while ensuring accuracy, we use principal component analysis (PCA) to achieve it. PCA is an orthogonal transformation to convert a set of inputs into a set of features of linearly uncorrelated variables called principal components, which reduce the inner complexity while retaining as much information as possible (Hotelling, 1933).

For opcode based representation, given an opcode bi-gram matrix $M(op_i, op_j)$, we first calculate the average matrix $M(op_i, op_j)_{avg}$ in training data sets. For each binary, we calculate the variance matrix $D(op_i, op_j)$ according to Eq. (1), where k is the ID of the opcode bi-gram matrix. Let $CV(op_i, op_j)$ be the covariance matrix. The next we calculate the covariance matrix and its eigenvectors, according to Eq. (2), where $n_{training}$ is the total number of binaries in the training data sets.

$$D(op_i, op_j)_k = M(op_i, op_j)_k - M(op_i, op_j)_{avg} \tag{1}$$

$$CV(op_i, op_j) = \frac{\sum D(op_i, op_j)_k^T \cdot D(op_i, op_j)_k}{n_{training}} \tag{2}$$

Let *eigenVec* be the column eigenvectors according to $CV(op_i, op_j)$. Let $eigenVec_t$ be the tth eigenvector in *eigenVec* which are ordered by eigenvalue $eigenVal_t$ from large to small, according to Eq. (3).

$$|CV(op_i, op_j) - eigenVal \cdot E| = 0 \tag{3}$$

We organize top T eigenvectors (column vectors, T=159, the number of dimensions of opcodes) to generate a new matrix *eigenMat* and get the PCA-initialized representation *PCAMat* according to Eq. (4). The *PCAMat* will be sent to a convolutional neural network to train a vector of opcode embedded features.

$$PCAMat_k = D(op_i, op_j)_k \cdot eigenMat \tag{4}$$

For API call based representation, we get the PCA-initialized representation of API frequencies as the same as the above operations for opcode based representation. Given an API frequency vector $V(api_i)$, firstly we calculate the average vector $V(api_i)_{avg}$ in training data sets. For each binary, we calculate the variance vector $D(api_i)$ according to Eq. (5), where k is the ID of the API frequency vector. Let $CV(api_i, api_i)$ be the covariance matrix. we next calculate the covariance vector and its eigenvectors, according to Eq. (6), where $n_{training}$ is the total number of binaries in the training data sets.

$$D(api_i)_k = V(api_i)_k - V(api_i)_{avg} \tag{5}$$

$$CV(api_i, api_i) = \frac{\sum D(api_i)_k^T \cdot D(api_i)_k}{n_{training}} \tag{6}$$

Let *eigenVec* be the column eigenvectors according to $CV(api_i)$. Let $eigenVec_t$ be the tth eigenvector in *eigenVec* which are ordered by eigenvalue $eigenVal_t$ from large to small, according to Eq. (7).

$$|CV(api_i, api_i) - eigenVal \cdot E| = 0 \tag{7}$$

Finally we organize top T eigenvectors (column vectors, T=200, in order to reduce the number of dimensions) to generate a new matrix *eigenMat* and get the PCA-initialized representation *PCAVec* according to Eq. (8). The *PCAVec* will be sent to a back-propagation neural network to train a vector of API call embedded features.

$$PCAVec_k = D(api_i)_k \cdot eigenMat \tag{8}$$

3.4. *Neural networks based feature embedding process*

Here we use a convolutional neural network (CNN) to train the opcode embedded features and use a back-propagation neural network to train the API embedded features. As widely used deep learning methods, the convolutional neural networks are more appropriate to our opcode based, 2-D structured, PCA-initialized representation and the back-propagation neural networks (BPNN) are suitable for our API based, 1-D structured, PCA-initialized representation.

Training process. Our CNN has 8 layers, an input layer, two convolution and pooling layers, a full connection layer, a softmax layer and an output layer. During forward pass, we first

push the opcode based PCA-initialized representation of executables as inputs into the CNN. Then, each convolutional core is convolved across the width and height of the inputs, computing the dot product between the entries of the filter and the input and producing a 2-D activation map of that convolutional core. Pooling is a form of down-sampling, and fully connects the next layer. Our approach uses mean-pooling. In the full connection layer, the pooling map fully connects the hidden units by ReLU function according to Eq. (9). After that, a softmax is used to train malicious instances and legitimate ones. It fully connects the units to the output, according to Eq. (10). The output of softmax function is vector of two confidence values, each of them represents the probability of malware or benign.

$$ReLU(w \cdot x) = max(0, w \cdot x) \tag{9}$$

$$Softmax\left(\sum w \cdot x\right) = \frac{e^{\sum w \cdot x}}{\sum e^{\sum w \cdot x}} \tag{10}$$

During back propagation, the CNN uses the gradient descent method (Barzilai and Borwein, 1998) to back propagate the variance from the output layer to the input layer and updates the weight matrixes of the connections between layers. The objective function is presented in Eq. (11), where X is the inputs, W is the weights in the neural network, $H(W \cdot X)$ is the result of the forward passing process of the neural network and Y is the value of the label (1/0). Here we use MSE (mean square error) loss in our neural networks. The rationale of using MSE in malware variants detection is the existed similarities among the variants, on the other words, the vectors of the variants are close to each other in n-dimension Euclidean space. Since several related works have demonstrated that mean square can represent the distance between two variants in Euclidean space, such as Santos et al. (2013a), de la Puerta et al. (2017), etc., by using MSE loss in neural networks to train a model may be reasonable. We update the weights according to Eq. (12), where α is the step size and $(Y - H(W \cdot X))$ is the variance. So the weights in the Softmax function are updated according to Eq. (13), where *var* is the variance.

$$min\ E = (Y - H(W \cdot X))^2 \tag{11}$$

$$\triangle W = \alpha \cdot \frac{dE}{dW} \cdot (Y - H(W \cdot X))) \tag{12}$$

$$\triangle w = \alpha \cdot Softmax\left(\sum w \cdot x\right) \cdot x \cdot var \tag{13}$$

Our BPNN has 3 layers, an input layer, a hidden layer and an output layer. During forward pass, we first input the API based PCA-initialized representation into the BPNN. Each unit in the input layer fully connects the units in the next hidden layer, and each unit in the hidden layer fully connects the units in the next outout layer. The sigmoid function is according to Eq. (14). The vector of labels in the output layer is a vector consisted by 1 and 0 which separately represents malware or benign. During back propagation, the neural network also uses the gradient descent method to back propagate the variance from the output layer to the input layer, and updates weight matrixes of connections between two layers according to Eq. (15), where *var* is the variance.

$$Sig\left(\sum w \cdot x\right) = \frac{1}{1 + exp(-\sum w \cdot x)} \tag{14}$$

$$\triangle w = \alpha \cdot Sig\left(\sum w \cdot x\right) \cdot \left(1 - Sig\left(\sum w \cdot x\right)\right) \cdot x \cdot var \tag{15}$$

Retraining. Through iterations, the output of a neural network is a vector of confidence values which represents probabilities of malware and benign. When the confidence value of malware is big enough, we deem this detection is sufficiently believable. For retraining a new instance, we design a mechanism based on active learning. We get new trained instances with highly confidence score (> 0.9) for retraining. When retraining more and more incremental instances, the decision boundary will continuously expand. Although it has to lose some new instances to retrain, the proportion of retrained instances will increase when the decision boundary expand enough. In retraining process, we use a copy of the current model to retrain. By testing the retrained model with the original data sets to check if it is correct (the accuracy is not suddenly dropped) to avoid poisonous data attacks, we transfer this retrained model to the current model.

Feature embedding. At last, when the neural networks mentioned above are converged, we gain parameters from the convolutional cores, the pooling cores, and the weight matrixes of the CNN, and gain parameters from the weight matrixes of the BPNN. These parameters will be fixed into the CNN and the BPNN respectively as our feature embedding model to generate embedded features. When an unknown instance is separately sent into the para-fixed CNN model and the para-fixed BPNN model, through forward passing, the units in the full connection layer of the CNN model will be treated as the opcode based embedded features and the units in the hidden layer of the BPNN model will be treated as the API based embedded features. These features will be sent to the next feature fusion based classifier for malware variants detection.

This feature embedding method is applicable for heterogenous neural networks, such as CNN, BPNN, etc.. Although the architectures of CNN and BPNN are different, both of them train the features by gradient descent method and map original data representation into a multi-dimension feature space. The features of the instances in the same class will be closer in the multi-dimension space through training so that the neural networks can be converged. The features can be treated as a set of common features shared by the instances in the same class. In this way, the features will be effective in embedding to other classifiers for further training and classification.

3.5. *Feature fusion based training and classification process*

We embed the opcode based features and the API based features by a serial manner. Let $U = \{u_{op,1}, u_{op,2}, \ldots, u_{api,1}, u_{api,2}, \ldots\}$ be the vector of the integrated features, where $u_{op,i}$ is the opcode based feature and $u_{api,i}$ is the API based feature. We input the integrated features U into a softmax classifier to

train and detect malware variants. The forward passing process is according to Eq. (10) and the back propagation process is based on gradient descent method according to Eq. (13). When the softmax classifier is converged, a model is generated to classify malicious instances and legitimate ones.

Multiple features can be merged by a serial manner when the features have a same scale of feature space. In our work, by normalization process and training process, the features from CNN and the features from BPNN will have a same scale [0, 1), so that they can be merged into a joint feature space. Let m be the number of dimensions of the features trained from CNN and let n be the number of dimensions of the features trained from BPNN. By training, the opcode bi-grams are mapped into a new m-dimension space and the API call frequencies are mapped into a new n-dimension space. When integrating these features by a serial manner, the hybrid features are mapped into a (m+n)-dimension space. Since malware variants have some similarities in opcodes and API calls, the opcode bi-grams and the API call frequencies which are used to represent malware variants also have some similarities, and the hybrid features of similar malware instances share similar patterns in the (m+n)-dimension space. Hence, using a classifier can divide the similar malware instances into a same class in the (m+n)-dimension space.

Softmax and Logistic are the two of the most widely used classifiers in neural networks. Since the number of dimensions of our hybrid features (which are trained from CNN and BPNN) is very large, and the softmax classifier is applicable for dealing with large number of dimensions while the logistic classifier is failed to train a classification model, here we use a softmax as the last classifier for the hybrid features. When using the logistic classifier, since the initial weights are random values, the sum of weighted inputs to different sigmoid units are very close so that the sigmoid units are hard to activate.

In addition, we change the labels of the CNN, the BPNN and the last softmax classifier from malware/benign to malware families and train new embedded features and a new classification model which will be used for a classification of malware families.

4. Experiments

In this section, we present several experiments to show the performance of our approach. At the beginning, we present the experiment setup, the data set and the validation. And then, we discuss the performance of our approach and show our approach can perform better by comparing with the state-of-art methods.

4.1. Setup, data set and validation

We implement all of the experiments on one computer. The version of its CPU is Intel i5-3470 @ 3.20 GHz, the RAM is 16.0 GB and the operating system is Linux Ubuntu 16.04. Our approach is developed by Java programming language in which the matrix computations are dependant on Jama-1.0.3.

Two data sets are considered in performance analysis of our approach for Windows malware variants detection: a Windows malware data set and a Windows benign data set. To be close to real-life environment, the Windows benign binaries we used in our experiments is collected from several PCs. The Windows malware instances which we use in our experiments are collected from VxHeaven[6]. To make sure that the instances in our data set are unpacked, we detect packers first. We find some Windows executables have been packed before and then unpack these executables by several unpacking tools. Our final dataset contains 5241 Windows benign binaries and 5250 Windows malware binaries. As is shown in Table 1.

Table 1 – The Windows malware data set.

Malware family	Number
Backdoor	1479
Worm	795
Trojan Dropper	1117
Trojan Banker	1376
Virus	980
Total	5250

```
:004092E2 6850964200          push 00429650
:004092E7 57                  push edi
:004092E8 50                  push eax
:004092E9 E860EE0000          call 0041814E
:004092EE 83C414              add esp, 00000014
:004092F1 8D85B8FDFFFF        lea eax, dword ptr [ebp+FFFFFDB8]
:004092F7 53                  push ebx
:004092F8 50                  push eax
:004092F9 E8E2F00000          call 004183E0
:004092FE 59                  pop ecx
:004092FF 50                  push eax
:00409300 8D85B8FDFFFF        lea eax, dword ptr [ebp+FFFFFDB8]
:00409306 50                  push eax
:00409307 FF750C              push [ebp+0C]
:0040930A FF15D0894300        call dword ptr [004389D0]
:00409310 8D85A4FCFFFF        lea eax, dword ptr [ebp+FFFFFCA4]
:00409316 50                  push eax
:00409317 E8C4F00000          call 004183E0
:0040931C 83F81E              cmp eax, 0000001E
:0040931F 59                  pop ecx
:00409320 8D85A4FCFFFF        lea eax, dword ptr [ebp+FFFFFCA4]
:00409326 50                  push eax
:00409327 7607                jbe 00409330
```

Fig. 3 – An example of opcode sequence.

We split the malware data set into a training data set and a testing data set, as well as the benign data set. To avoid training biases, the volume of the malware training data set and the volume of the benign training data set are the same size. We randomly choose 2000 malware samples and 2000 benign samples for training, and 3250 malware samples and 3241 benign samples for testing. In order to evaluate the performance of our approach, we use k-fold cross validation in the experiments and choose the average values as our results.

4.2. Opcode sequences and API calls

To extract opcodes and API calls, we use W32dasm to disassemble Windows binaries, and then obtain Windows opcode sequences and API calls. The raw profiles of opcode sequences and API calls are presented in Figs. 3 and 4. We

[6] http://vxheaven.org/vl.php.

```
Import Module 001: KERNEL32.DLL

Addr:00022D20 hint(0000) Name: GlobalUnlock
Addr:00022D2E hint(0000) Name: SetEnvironmentVariableA
Addr:00022D48 hint(0000) Name: CompareStringW
Addr:00022D58 hint(0000) Name: CompareStringA
Addr:00022D68 hint(0000) Name: SetEndOfFile
Addr:00022D76 hint(0000) Name: FlushFileBuffers
Addr:00022D88 hint(0000) Name: SetStdHandle
Addr:00022D96 hint(0000) Name: GetStringTypeW
Addr:00022DA6 hint(0000) Name: GetStringTypeA
Addr:00022DB6 hint(0000) Name: RtlUnwind
Addr:00022DC2 hint(0000) Name: GetFileType
Addr:00022DD0 hint(0000) Name: GetStdHandle
Addr:00022DDE hint(0000) Name: SetHandleCount
Addr:00022DEE hint(0000) Name: GetEnvironmentStringsW
Addr:00022E06 hint(0000) Name: GetEnvironmentStrings
Addr:00022E1E hint(0000) Name: FreeEnvironmentStringsW
Addr:00022E38 hint(0000) Name: FreeEnvironmentStringsA
Addr:00022E52 hint(0000) Name: GetModuleFileNameA
Addr:00022E66 hint(0000) Name: Sleep
Addr:00022E6E hint(0000) Name: MultiByteToWideChar
Addr:00022E84 hint(0000) Name: ReadFile
Addr:00022E8E hint(0000) Name: CloseHandle
Addr:00022E9C hint(0000) Name: WriteFile
Addr:00022EA8 hint(0000) Name: TransactNamedPipe
Addr:00022EBC hint(0000) Name: CreateFileA
Addr:00022ECA hint(0000) Name: SetFilePointer
```

Fig. 4 – An example of API calls.

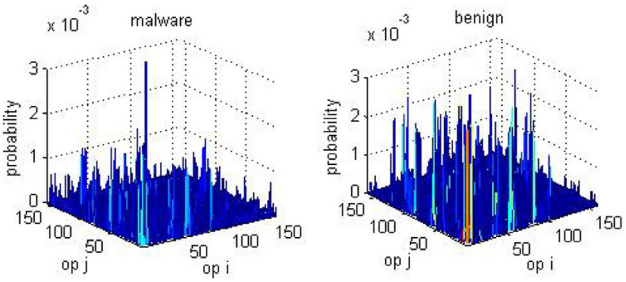

Fig. 5 – The distribution comparison of opcodes between malware and benign.

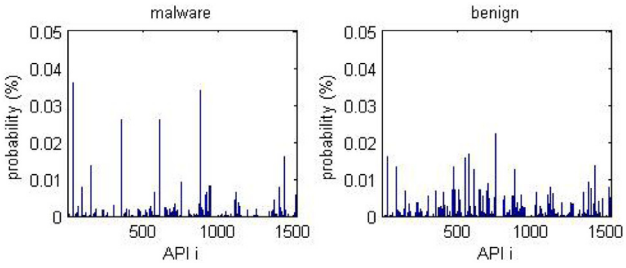

Fig. 6 – The distribution comparison of API calls between malware and benign.

collect 159 opcodes and 1520 API calls, so that the size of opcode bi-gram matrix is 159×159 and the size of API vector is 1×1520. In some cases (e.g. some binaries in virus family), we cannot extract any API call, then we use a vector of all-zero values to represent the binaries. The profiles not only include opcodes and API calls, but also include operands and addresses. Since the opcodes and API calls contains enough information, and the operands and addresses seem as noisy information, so in this work, we only extract the opcodes and the API calls in the profiles. The next we use these opcodes and API calls to represent binaries respectively. After that, we implement two models for feature extraction and embedding. Through feature fusion, our approach train the embedded features to generate a classification model to detect malware variants.

4.3. *Differential analysis of opcode bi-gram and API call frequency between malware and benign*

Here we analyze the reasons why the hybrid features are effective. Since our hybrid features are generated by training opcode bi-gram and API call frequency, the hybrid features are a space projection of the two data representations, the hybrid features can be used to represent the two data representations. If the differences of the two data representations between malware and benign are large enough, the hybrid features will be effective for classifying malware variants and legitimate ones. Hence, we give a differential analysis of the two representations and explain why the differences exist.

In this work, we integrate opcodes and API calls to represent binaries. Figs. 5 and 6 show the average probability distributions of opcode bi-gram and API calls respectively. From the results, we find the distributions of opcode bi-gram and API calls between malware and benign are significantly different, which means the representation of opcode bi-gram and the representation of API call frequency can be used for a classification of malware and benign. The distribution comparisons of opcodes and API calls show that: The opcode bi-grams and the API calls used in malware are concentrated while they are dispersive in benign. The intuition here is the programs of the variants of malicious softwares exist similarities (Cesare et al., 2014a), since both of the opcodes and the API calls are the representation of programs, the opcode bi-grams and the API calls also exist similarities. These similarities make some of the opcode bi-grams and some of the API calls concentrated in malware when comparing with benign. This property makes the neural networks train common features in malware.

Here we use Kullback–Leibler divergence D_{KL} to measure the distance of the distributions between malware and benign, according to Eq. (16). The D_{KL} of opcode bi-gram is 0.5459 and the D_{KL} of API frequency is 0.3117.

$$D_{KL} = \sum \begin{cases} p_{malw} \cdot \ln\left(\frac{p_{malw}}{p_{ben}}\right), p_{malw} > p_{ben} \\ p_{ben} \cdot \ln\left(\frac{p_{ben}}{p_{malw}}\right), p_{malw} < p_{ben} \\ 0, p_{malw} = p_{ben} \end{cases} \tag{16}$$

4.4. *Convergence analysis of PCA-initialization*

To accelerate the convergence speed of the neural networks, we use PCA to optimize the representation. We show examples of the convergence process of PCA-initialized neural networks and non-PCA-initialized neural networks, as shown in Fig. 7. For the CNN based opcode embedding, the example shows the PCA-initialized representation can significantly converge faster compared with the non-PCA-initialized representation. For the BPNN based API embedding, the ex-

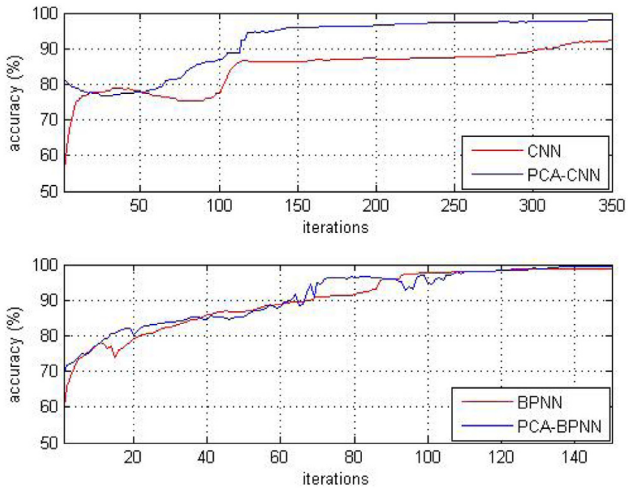

Fig. 7 – The convergence comparison of training process between PCA-initialization and non-PCA-initialization.

Table 2 – The hyper-parameter settings of our CNN based opcode embedding.

Item	Value
Size of input vector	159 × 159
Size of PCA vector	159 × 159
Number of convolutional layers	2
Number of convolutional cores for each layer	5
Size of each convolutional core	5 × 5
Number of pooling layers	2
Number of pooling cores for each layer	5
Size of each pooling core	2 × 2

Table 3 – The hyper-parameter settings of our BPNN based API embedding.

Item	Value
Size of input vector	1 × 1520
Size of PCA vector	1 × 200
Number of hidden layers	1
Number of hidden units	50

ample shows the PCA-initialized representation can converge faster at the beginning of the training process compared with the non-PCA-initialized representation, that means the PCA-initialized representation is more robust for training in neural networks and can avoid potential under-fitting.

4.5. *Hyper-parameter settings*

The hyper-parameters which are set by designers may have a great affect on performance. In this subsection, we show the hyper-parameter settings in our approach. Since it has two feature embedding models: the CNN based opcode embedding model and BPNN based API embedding model, we show the hyper-parameters in these models respectively, as shown in Tables 2 and 3. The principle we set the hyper-parameters here is to ensure accuracy while retaining training speed.

4.6. *Performance analysis of malware detection*

To show that our approach is optimized and efficient, we compare our approach with several state-of-art methods. Santos et al. (2013a) represented binaries by using opcode 2-tuples and information gain, then adopted several machine learning methods to detect malware variants. de la Puerta et al. (2017) proposed to adopt several machine learning methods to classify the opcode vectors. Kang et al. (2016) presented an n-opcode analysis based approach that utilizes machine learning to detect malware. Raff et al. (2017) used byte n-gram matrix to represent executables and adopted convolutional neural networks to detect malicious executables. Canzanese et al. (2015) proposed to use a vector of system call n-gram frequencies and use several classifiers to detect malware. Santos et al. (2013c) proposed a hybrid malware variant detector which utilizes a set of features, we implement their method with opcodes and API calls as a comparison.

All of the methods are implemented to run in the same configurations which include: the same machine, the same operating system, the same Java virtual machine (JVM), the same training data set and the same detection data set.

The benchmarks which we use for performance comparison include classification accuracy, detection precision, detection recall, detection false positive rate, F1-score, training time cost and detection time cost. The classification accuracy is according to Eq. (17). *TPR* is true positive rate (malware detection recall) according to Eq. (18), where *TP* is the number of malware cases which are correctly classified and *FN* is the number of malware cases which are misclassified as benign binaries. *TNR* is true negative rate, as shown in Eq. (19), where *FP* is the number of benign cases which are incorrectly classified as malware binaries and *TN* is the number of benign binaries which are correctly classified. *FPR* is false positive rate, *Precision* is malware detection precision, and *F1 - score* is according to *Precision* and *Recall*, as shown in Eqs. (20)–(22).

$$accuracy = \frac{TP + TN}{TP + FN + TN + FP} \tag{17}$$

$$TPR(Recall) = \frac{TP}{TP + FN} \tag{18}$$

$$TNR = \frac{TN}{FP + TN} \tag{19}$$

$$FPR = \frac{FP}{FP + TN} \tag{20}$$

$$Precision = \frac{TP}{TP + FP} \tag{21}$$

$$F1\ score = \frac{2 * Precision * Recall}{Precision + Recall} \tag{22}$$

As shown in Table 4, the performance evaluations of several state-of-art methods show that:

Table 4 – The performance evaluations of several state-of-art approaches.

Method	Classification **accuracy (%)**	Detection **precision (%)**	Detection **recall (%)**	Detection **1-FPR (%)**	Detection **F1-score (%)**
Our approach	95.1	95.7	94.3	95.9	95.0
Santos et al. (KNN, opcode)	87.7	84.8	92.0	83.4	88.3
Santos et al. (KNN, API)	79.1	92.7	63.2	95.0	75.2
Santos et al. (KNN, opcode & APIcall)	89.8	93.3	85.8	93.8	89.4
Puerta et al. (SVM, opcode)	83.5	86.5	80.6	87.4	83.4
Kang et al. (NB, opcode)	79.7	78.3	82.2	77.2	80.2
Raff et al. (CNN, bytecode)	83.8	82.5	85.8	81.8	84.1
Canzanese et al. (SVM, syscall)	86.6	92.4	79.8	93.4	85.6

Method	**Detection Time cost (s)**	**Training Time cost (s)**
Our approach	0.034	140747.0
Santos et al. (KNN, opcode)	5.130	0.0
Santos et al. (KNN, API)	1.114	0.0
Santos et al. (KNN, opcode & APIcall)	6.536	0.0
Puerta et al. (SVM, opcode)	<0.001	31.0
Kang et al. (NB, opcode)	0.005	134.0
Raff et al. (CNN, bytecode)	0.053	213467.0
Canzanese et al. (SVM, syscall)	0.094	179.0

1. By comparing with the other state-of-art methods, our approach significantly improves the classification accuracy, the detection precision, the detection recall, the 1-FPR and the F1-score of malware variants detection while retaining the detection speed. The only sacrifice is the training time cost, which will cost more than one day.
2. Santos et al.' (KNN, opcode) method uses K-nearest neighbor (KNN) to search similarities of opcode 2-tuples between targeted binaries and labeled binaries in a serial manner. Since the KNN method only needs to search similarities between the targeted binary and each labeled binaries in detection process and does not need to train a model like other machine learning methods, so it does not take any training time cost (the training time cost is 0.0 s), but cost much more time in detection process. However, users who need the malware detection engine care more about detection time cost rather than training time cost.
3. Santos et al.' (KNN, API call) method uses K-nearest neighbor (KNN) to search similarities of API probabilities, it performs poorly in terms of accuracy.
4. Santos et al.' (KNN, opcode & API call) method search similarities of features which include opcodes and API calls. By comparing with their opcode based method and API call based method, it improves the accuracy.
5. Puerta et al.' (SVM, opcode) method uses opcode frequencies to represent binaries and adopts support vector machine to detect malware variants. However, their features are simple, which cannot contains enough information to ensure the accuracy.
6. Kang et al.' (NB, opcode) method adopts naive bayes (NB) to detect the 2-opcode vectors of malware variants. The method is less accurate since naive bayes is based on a simple assumption that the features are independent with each other. However, this assumption is not really true in practise.
7. Raff et al.' (CNN, bytecode) method detects malware variants by using convolutional neural networks (CNN) and bytecode n-grams. Since bytecodes contains a lot of noise compared with opcodes, the accuracy of their method are limited by the noisy information.
8. Canzanese et al.' (SVM, syscall) method uses system call n-gram to represent binaries, and adopts support vector machine (SVM) to detect malware variants. The performance seems not too bad since the system call is a high-level representation of binaries, like API calls, which can precisely represent binaries' behaviour. However, this dynamic analysis based method cannot work if a malware lurks in a computer and hides its malicious behaviours.

Our feature-hybrid approach includes two feature embedding models: one is the CNN based opcode embedding model, the other is BPNN based API embedding model. Here we show the performance of these two models respectively to reflect the improvement of our feature-hybrid model, as shown in Table 5. Since we can hardly extract API calls from some malicious binaries in virus family, in the results of BPNN based API embedding model, we use a vector of all-zero values to represent the binaries. The performance comparison with our feature-hybrid model and the other two models shows that:

1. Our feature-hybrid model significantly improves the performance by comparing with the other two models since it embeds both of opcode features and API features.
2. For our CNN based opcode embedding model, since the opcode bi-gram can represent semantic features of binaries, which has been widely used in many researches, and the CNN can extract the features of the clusters of opcode bi-grams, which retains much more useful information, this model performs well in terms of accuracy by comparing the above state-of-art methods.

Table 5 – The performance comparison with our feature-hybrid model and the single-feature model used in our method.

Method	Classification accuracy (%)	Detection precision (%)	Detection recall (%)	Detection 1-FPR (%)	Detection F1 - score (%)
Our approach	95.1	95.7	94.3	95.9	95.0
CNN based opcode embedding	91.2	92.0	89.6	92.8	90.8
BPNN based API embedding	89.8	85.0	96.9	82.9	90.6

Method	Detection Time cost (s)	Training Time cost (s)
Our approach	0.034	140747.0
CNN based opcode embedding	0.023	137582.0
BPNN based API embedding	<0.001	1369.0

3. For our BPNN based API embedding model, by using a high-level representation of binaries, API call (static analysis), and a efficient classifier BPNN, the model can detect malware variants. The detection recall of the BPNN based API embedding model is a bit larger than that of our approach, because we cannot capture any API calls both in some benign binaries (such as some executables in Windows kernel) and some malware binaries (such as some executables in virus family), and this make the BPNN treats these instances (all-zero padding vectors) as malware, so that the detection recall is a bit higher. However, in this case, it makes the detection precision lower. When our approach integrates opcode features and API features, the vectors of features are conducted by opcode features and API features, so that the vectors of features are not all-zero padding any more. This improves detection precision while retaining detection recall.

4.7. *Stability evaluation of malware detection*

Since malware variants are rapidly growing in volume facing the Internet today, the volume of training samples is always smaller than the volume of detecting set. When the detection set contains a large quantity of binaries and the training set is smaller (the radio of $\frac{training}{training+detection}$ is small), the detection accuracy will severely limited (Zhang et al., 2016b). So here we evaluate the stability of our approach by testing in different volumes of training sets. The size of our training sets are 500, 1000 and 2000.

The results from Fig. 8 show the accuracy, the precision, the recall, the 1-FPR and the F1-score of our approach is stable, which means our approach is stable when the radio of $\frac{training}{training+detection}$ is different. The results also show our approach performs well even the radio of $\frac{training}{training+detection} < 0.2$. In addition, the results from Fig. 9 show the training time cost of our approach when training in different volume of data sets. Since the proportion of benign to malware is not balanced in the real world, usually the volume of benign is much more than that of malware. So here we demonstrate the effectiveness of our approach at 90%-10%, 85%-15%, 80%-20% proportion of benign to malware. To avoid the data bias, we retrain malware data 9 times, 6 times, 4 times respectively when the proportion of benign to malware is 90%-10%, 85%-15%, 80%-20%. The results show our approach is also effective when the proportion of benign to malware is unbalanced, as shown in Fig. 10.

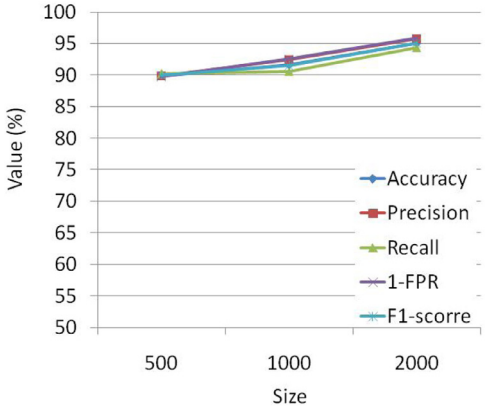

Fig. 8 – The stability evaluation of accuracy (The proportion of benign to malware is balanced).

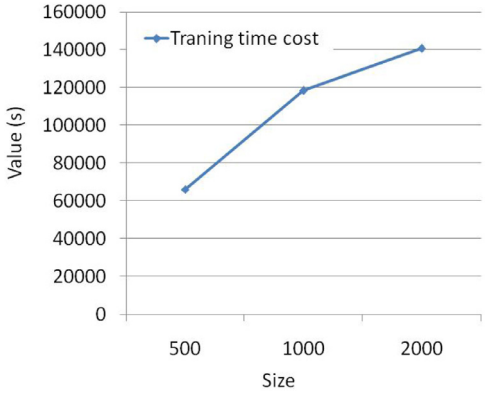

Fig. 9 – The training time cost of our approach when training in different volume of data sets.

4.8. *Accuracy evaluation of malware family classification*

Malware variants detection aims to classify malware variants and benign instances, which protect operating systems from attacks. The classification of malware family is also important to help people understand which family a malware instance belongs to. Many researches have presented that solving malware family classification problem is also important, such as (Massarelli et al., 2017b), (Zhang et al., 2018b), etc.. Since single-feature model (e.g. opcode based model, API call based

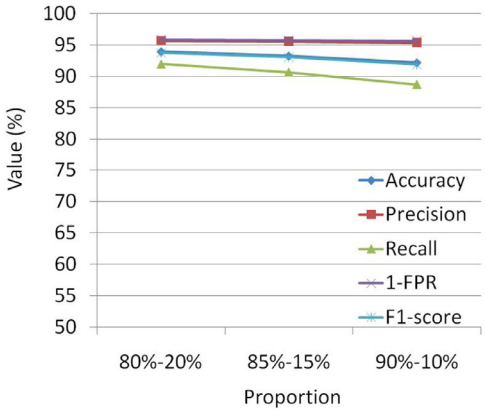

Fig. 10 – The stability evaluation of accuracy (The proportion of benign to malware is unbalanced).

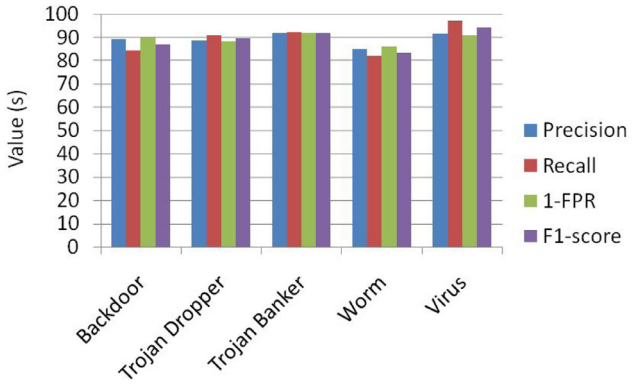

Fig. 11 – The precision, recall, 1-FPR, F1-score of our approach for malware family classification.

model) poorly classify malware families due to insufficient information in single type of representations (e.g. opcode or API call), our approach which uses complementary features can significantly improve the accuracy of malware family classification.

Here we implement our approach to classify malware families and evaluate the accuracy of our approach. As shown in Fig. 11, the results show that our approach can effectively classify malware families and precisely identify most of malware variants. More details are presented in Fig. 12. Confusion matrix M gives a quick graphical overview of the performance of a classifier. The generic element $M_{i,j}$ of the matrix is the number of samples belonging to class i that has been classified as j by the classifier. The precision, recall, FPR and F1-score are related measures, and are defined according to Eqs. (23)–(25) and (22).

$$Precision_i = \frac{M_{i,i}}{\sum_i M_{j,i}} \tag{23}$$

$$Recall_i = M_{i,i} \tag{24}$$

$$FPR_i = \sum_{i \neq j} M_{i,j} \tag{25}$$

As shown in Fig. 12, by comparing with the three models, we can find both the opcode based model and the API based model perform poorly for malware family classification since the information is not enough, while our feature-hybrid model performs well by considering both of opcode features and API call features. For CNN based opcode embedding model, since the malware families, worm and virus, contains much less information compared with other malware families, the opcode based model makes more mistakes when identifying malware variants in these malware families. For BPNN based API embedding model, because we can hardly extract API calls from the malicious binaries in virus family, so this model is not appropriate for identifying virus family.

5. Limitations

Varied packers sometimes allow malware to bypass the modern malware detection techniques. In most cases, packers can be unpacked by unpacking techniques or tools such as PolyUnpack (Royal et al., 2016), which recovers original software sources. Because packed softwares have to unpack their inner original codes before execution, so that unpacking tech-

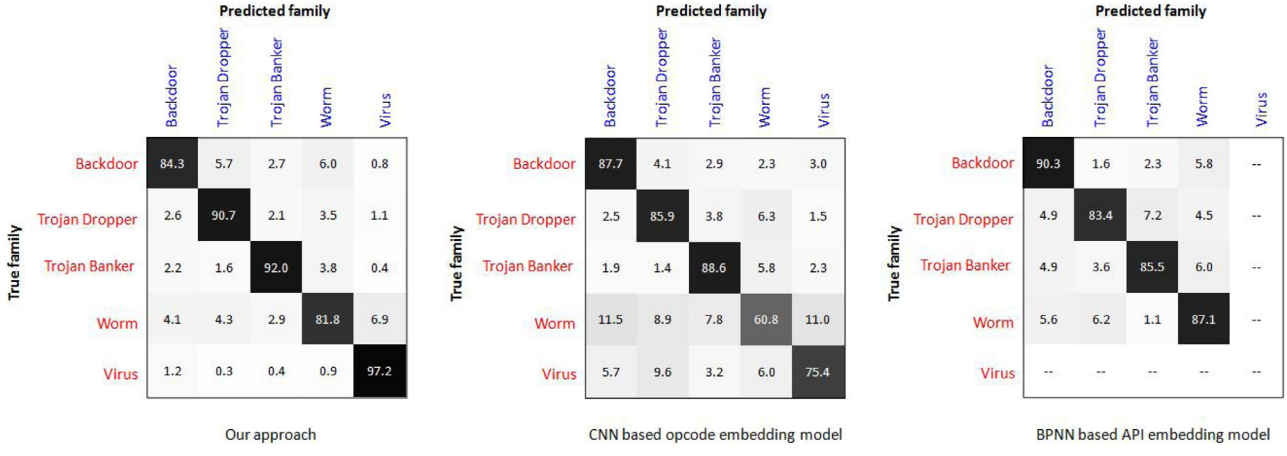

Fig. 12 – The confusion matrix comparison of malware family classification (The darkness determines the percentage).

niques always have a chance to get the original codes. But obfuscated software is more harder to be de-obfuscated. So in this section, we mainly discuss the limitations caused by obfuscation.

Obfuscation is defined by Cesare et al. (2014b) as a result of a semantic preserving transformation and obfuscated programs. Two obfuscated programs which are derived from the same source program are similar. The the similarities support our technique to detect a degree of obfuscated malware variants. Several types of obfuscation have been found out, such as identifier renaming, junk code injection, control flow based obfuscation, etc.. The identifier renaming obfuscation renames variables which could prevent manual analysis, but it is hard to impact distributions of instructions, opcodes, or other representations of binaries. The junk code injection could change the distributions, however it is easily to be detected and denoised. The control flow based obfuscation changes the control flow graph which makes the control flow based detection uncorrect. It generates some noisy instructions but retains the original instructions, which keeps partly similarities. Most of obfuscation techniques are simple obfuscation such as renaming which cannot make a significant change to the distribution (Dong et al., 2018). Some obfuscated malware instances are treated as malware variants. In general, our approach can resist a degree of mistakes caused by obfuscation. The obfuscated malware variants detection accuracy is relevant to the obfuscation degree and the noise distribution.

Besides, adversarial machine learning attack is also a limitation of all of machine learning based applications. However it still needs more researches to solve such problem.

6. Conclusions

In this paper, we propose a feature-hybrid malware variant detection technique which effectively and efficiently identifies malware variants from legitimate executables, and classifies their malware families as well. This technique first collects opcodes and API calls from disassembling malicious executables and legitimate ones, then represents each executable by opcode bi-grams and API frequencies, the next trains opcode features and API features through principal component analysis initialized convolutional neural networks and principal component analysis initialized back-propagation neural networks respectively, and finally embeds these features into a softmax classifier to train a classification model. The classification model will be used for malware variants detection.

Our approach smoothly integrates fine-grained features (opcodes) and high-level features (API calls) to cover more characteristics of malware and improve detection precision and detection recall of more than 5%. Real-life experimental results show our approach achieves more than 95% malware detection accuracy and almost 90% classification accuracy of malware families. The detection speed of our approach is less than 0.1 s.

As a future work, we will integrate more static analysis based features to our approach, such as control data flow graph, etc.. Beside, our feature fusion based framework can not only be sued for Windows malware detection, but also be used for Android malware detection and other domains, such as traffic anomaly detection, anti-fraud, etc.

Conflict of Interest

No conflict of interest exits in the submission of this manuscript, and manuscript is approved by all authors for publication.

Acknowledgment

This work is partially supported by the National Natural Science foundation of China under Grant nos. 61772191, 61472131.

REFERENCES

Barzilai J, Borwein JM. Two-point step size gradient methods. IMA J Numer Anal 1998;8(1):141–8.

Burguera I, Nadjm-Tehrani S, Zurutuza U. Crowdroid: behavior-based malware detection system for android. Proceedings of ACM workshop on security and privacy in smartphones and mobile devices, 2011.

Canzanese R, Mancoridis S, Kam M. System call-based detection of malicious processes. Proceedings of IEEE international conference on software quality, reliability and security, 2015.

Cesare S, Xiang Y, Zhou W. Control flow-based malware variant detection. IEEE Trans Depend Secur Comput (TDSC) 2014a;11(4):304–17.

Cesare S, Xiang Y, Zhou W. Control flow-based malware variant detection. IEEE Trans Depend Secur Comput (TDSC) 2014b;11(4):304–17.

Chen L, Hardy W, Ye Y, Li T. Analyzing file-to-file relation network in malware detection. Proceedings of the 16th international conference on web information systems engineering (WISE), 2015.

Dong S, Li M, Diao W, Liu X, Liu J, Li Z, Xu F, Chen K, Wang X, Zhang K. Understanding android obfuscation techniques: a large-scale investigation in the wild, 2018. arXiv:1801.01633.

Enck W, Han S, Gilbert P, Tendulkar V, Cox LP, Chun BG, Jung J, Sheth AN, McDaniel P. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans Comput Syst (TOCS) 2014;32(2). Article 5.

Fan M, Liu J, Luo X, Chen K, Tian Z, Zheng Q, Liu T. Android malware familial classification and representative sample selection via frequent subgraph analysis. IEEE Tran Inf Forensics Secur 2018;13(8):1890–905.

Gandotra E, Bansal D, Sofat S. Malware 2014 survey- analysis and classification. Inf Secur 2014;5(2):56–64.

Hotelling H. Analysis of a complex of statistical variables into principal components. J Educ Psychol 1933;24:417–41.

Huang J, Zhang X, Tan L. AsDroid detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In: Proceeding of ACM/IEEE international conference on software engineering; 2014. p. 1036–46.

Kalchbrenner N, Grefenstette E, Blunsom P. A convolutional neural network for modelling sentences, 2014. arXiv:1404.2188 [cs.CL].

Kang B, Yerima SY, McLaughlin K, Sezer S. N-opcode analysis for android malware classification and categorization. Proceedings of international conference on cyber security and protection of digital services (Cyber Security), 2016.

Kang B, Yerima SY, McLaughlin K, Sezer S. N-opcode analysis for android malware classification and categorization. Proceedings of international conference on cyber security and protection of digital services (Cyber Security), 2017.
Kim JY, Bu SJ, Cho SB. Zero-day malware detection using transferred generative adversarial networks based on deep autoencoders. Inf Sci 2018;460:83–102.
Kolbitsch C, Comparetti PM, Kruegel C, Xiaoyong Zhou EK, Wang X. Effective and efficient malware detection at the end host. Proceedings of the 18th USENIX security symposium, 2009.
Liu X, Liu J, Zhu S, Wang W, Zhang X. Privacy risk analysis and mitigation of analytics libraries in the android ecosystem. IEEE Trans Mob Comput 2019. doi:10.1109/TMC.2019.2903186.
Martín A, Menéndez HD, Camacho D. MOCDroid: multi-objective evolutionary classifier for Android malware detection. Soft Computing 2017;21:7405–15.
Massarelli L, Aniello L, Ciccotelli C, Querzoni L, Ucci D, Baldoni R. Android malware family classification based on resource consumption over time, 2017a. arXiv:1709.00875v1 [cs.CR].
Massarelli L, Aniello L, Ciccotelli C, Querzoni L, Ucci D, Baldoni R. Android malware family classification based on resource consumption over time. In: Proceedings of international conference on malicious and unwanted software (MALWARE); 2017b. p. 31–8.
McLaughlin N, del Rincon JM, Kang B, Yerima S. Deep android malware detection. Proceedings of ACM conference on data and application security and privacy (CODASPY), 2017.
Nataraj L, Yegneswaran V, Porras P. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. Proceedings of the 4rd ACM workshop on security & artificial intelligence (AISec), 2011.
Oberheide J, Cooke E, Jahanian F. CloudAV: N-version antivirus in the network cloud. Proceedings of the 17rd ACM international conference on Usenix security symposium (USENIX), 2008.
Patanaik C, Barbhuiya F, Nandi S. Obfuscated malware detection using API call dependency. In: Proceedings of ACM international conference on security of internet of things; 2012. p. 289–300.
de la Puerta JG, Sanz B, Santos I, Bringas PG. Using Dalvik opcodes for malware detection on android. Logic J. IGPL 2017;25:938–48.
Raff E, Barker J, Sylvester J, Brandon R, Catanzaro B, Nicholas C. Malware detection by eating a whole EXE, 2017. arXiv:1710.09435.
Rieck K, Trinius P, Willems C. Automatic analysis of malware behavior using machine learning. J Comput Secur 2011;19:639–68.
Royal P, Halpin M, Dagon D, Edmonds R, Lee W. PolyUnpack: automating the hidden-code extraction of unpack-executing malware. Proceedings of the 22nd annual computer security applications conference (ACSAC), 2016.
Santos I, Brezo F, Ugarte-Pedrero X, Bringas PG. Opcode sequences as representation of executables for data mining based malware variant detection. Inf Sci 2013a;231(9):1–13.
Santos I, Brezo F, Ugarte-Pedrero X, Bringas PG. Opcode sequences as representation of executables for data mining based malware variant detection. Inf Sci 2013b;231(9):64–82.
Santos I, Devesa J, Brezo F, Nieves J, Bringas PG. OPEM: a static-dynamic approach for machine learning based malware detection. In: Proceeding of international joint conference CISIS; 2013c. p. 271–80.
Shabtai A, Elovici Y, Kanonov U, Weiss Y, Glezer C. Andromaly: a behavioral malware detection framework for android devices. J Intell Inf Syst 2012;38(1):161–90.
Stringhiniq G, Shen Y, Han Y, Zhang X. Marmite: spreading malicious file reputation through download graphs. Proceedings of the 33rd annual computer security applications conference (ACSAC), 2017.
Symantec. Internet security threat report, 2017.
Tamersoy A, Rouncy K, Chau DH. Guilt by association: large scale malware detection by mining file-relation graphs. Proceedings of the ACM international conference on knowledge discovery and data mining (SIGKDD), 2011.
Tian K, Yao D, Ryder BG, Tan G, Peng G. Detection of repackaged android malware with code-heterogeneity features. IEEE Trans Depend Secure Comput (TDSC) 2018. doi:10.1109/TDSC.2017.2745575.
Wang W, Gao Z, Zhao M, Li Y, Liu J, Zhang X. DroidEnsemble: detecting android malicious applications with ensemble of string and structural static features, 6; 2018a. p. 31798–807.
Wang W, Li Y, Wang X, Liu J, Zhang X. Detecting Android malicious apps and categorizing benign apps with ensemble of classifiers. Future Generation Computer System 2018b;78:987–94.
Wang W, Wang X, Feng D, Liu J, Han Z, Zhang X. Exploring permission-induced risk in android applications for malicious application detection. IEEE Trans Inf Forensics Secur 2014;9(11):1869–82.
Xu L, Zhang D, Alvarez MA. Dynamic android malware classification using graph-based representations. In: Proceeding of IEEE international conference on cyber security and cloud computing; 2016. p. 220–31.
Yan J, Qi Y, Rao Q. Detecting malware with an ensemble method based on deep neural network. Secur Commun Netw 2018:1–16. doi:10.1155/2018/7247095.
Yan LK, Yin H, Droidscope. Seamlessly reconstructing the os and Dalvik semantic views for dynamic android malware analysis. Proceedings of USENIX security symposium, 2012.
Zhang J, Qin Z, Yin H, Ou L, Hu Y. IRMD: malware variant detection using opcode image recognition. Proceedings of the 23rd IEEE international conference on parallel and distributed systems (ICPADS, 2016a.
Zhang J, Qin Z, Yin H, Ou L, Xiao S, Hu Y. Malware variant detection using opcode image recognition with small training sets. Proceedings of the 25rd IEEE international conference on computer communication and networks (ICCCN), 2016b.
Zhang J, Qin Z, Zhang K, Yin H, Zou J. Dalvik opcode graph based android malware variants detection using global topology features, 6; 2018a. p. 51964–61974.
Zhang J, Zhang K, Heng Qin, Yin H, Wu Q. Sensitive system calls based packed malware variants detection using principal component initialized MultiLayers neural networks. Cybersecurity 2018b;1(10):1–13.

Jixin Zhang received the B.S. degree in Mathematics, and the M.S. degree in computer science and technology from Wuhan University of Technology, in 2011 and 2014, respectively. Currently, he is pursuing the Ph.D. degree in the College of Information Science and Engineering at Hunan University, and doing researches in the Department of Information Engineering at Chinese University of Hong Kong. His primary researches focus on security and machine learning.

Zheng Qin received the B.S. degree from Wuhan University of Technology, in 1991. He got his Ph.D.degree in computer software and theory at Chongqin University, China, in 2001. Then he worked in industry between 2001 and 2005. Currently, he is a professor of computer science and technology in Hunan University, China. He is a member of China Computer Federation (CCF) and ACM, respectively. His main interests are information security, computer network and big data.

Hui Yin received the B.S. degree from Hunan Normal University and the M.S. degree from Central South University, both in computer science, in 2002 and 2008, respectively. He is currently pursuing the Ph.D. degree in the College of Information Science and Engineering at Hunan University, China. His main interests are information security, privacy protection, and secure search in the cloud computing environment.

Lu Ou received the B.S. degree from Changsha University of Science and Technology in computer science in 2009 and received the M.S. degree and the Ph. D degree from Hunan University both in software engineering, in 2012 and 2018, respectively. She is a member of IEEE. Her research focuses on security, privacy, optimization and big data.

Kehuan Zhang received his B.S. and M.E. degree from Hunan University in 2001 and 2004, respectively. He worked in industry between 2004 and 2007 before starting his Ph.D. program at Indiana University, Bloomington. He got his Ph.D. degree there and joined CUHK in 2012. During his Ph.D.program, he also worked as an intern at IBM T J Watson Research Center in the summer of 2010. Currently, he is a professor at the department of Information Engineering, the Chinese University of Hong Kong. His primary research interests are: Security and Privacy in computer network, Web, Clouds, smart phones, embedded systems and other distributed computing systems. He has published several high quality paper on all of the four top conferences in network and system security area, including IEEE Oakland, ACM CCS, USENIX Security, and NDSS. He is also an active external reviewers of these top conferences.