IET Information Security

Research Article

IET Journals
The Institution of Engineering and Technology

Dynamic API call sequence visualisation for malware classification

ISSN 1751-8709
Received on 28th November 2017
Revised 31st May 2018
Accepted on 27th September 2018
E-First on 6th March 2019
doi: 10.1049/iet-ifs.2018.5268
www.ietdl.org

Mingdong Tang[1], Quan Qian[1,2] ✉

[1]School of Computer Engineering & Science, Shanghai University, Shanghai 200444, People's Republic of China
[2]Materials Genome Institute, Shanghai University, Shanghai 200444, People's Republic of China
✉ *E-mail: qqian@shu.edu.cn*

Abstract: Due to the development of automated malware generation and obfuscation, traditional malware detection methods based on signature matching have limited effectiveness. Thus, a novel approach using visualisation and deep learning technology can play an important role in malware detection and classification. In this study, the authors extract sequences of API calls using dynamic analysis and then use colour mapping rules to create feature images representing malware behaviour. Finally, they train a convolutional neural network to classify different feature images with 9 malware families, and 1000 variants in each family. Experimental results show the effectiveness of the authors' method. The classification TPR, precision, recall and F1 are all >99%, while the FPR is <0.1%.

1 Introduction

Malicious code or malware is defined as software that deliberately intends to harm a computer [1]. Nowadays, automated tools allow adversaries or hackers to easily generate new malware. Symantec reported >159 million new malware variants in the first quarter of 2018 [2]. Additionally, automatic malware generation tools do not simply repackage similar modules that are easily fingerprinted and identified. They now implement sophisticated anti-detection technologies, such as packing, confusing, anti-debugging, compression, encryption, which renders most of the traditional detection methods ineffective [3, 4]. One type of malware analysis, called static analysis, mostly operates on machine-level code or disassembled instructions [5–7] and works well against known malware with typical signatures [8]. However, static analysis is not robust against obfuscation techniques [9] and cannot identify new malware. In comparison, dynamic analysis extracts features while executing malware in a controlled environment, and as a result, is relatively robust to obfuscation and polymorphic techniques. However, execution path coverage, data analysis and methods selection are also open and challenging problems for dynamic analysis.

In this paper, we propose a different and novel approach to malware identification. First, we extract sequences of a malware's application programming interface (API) sequence by dynamic execution. Then we create a visual fingerprint for each malware by constructing feature images using API categorisation and a colour mapping. Finally, we train a CNN (convolutional neural network) on the different malware family features and build a classifier to identify them. The main contributions of the paper are as follows:

(i) Propose a set of colour mapping rules to visualise the API call sequence. Our coarse-grained approach is robust to equivalent API replacement techniques, which shield the details of concrete API calls.
(ii) Use CNN to classify malware fingerprint images. Our experiments show that this method has high accuracy and efficiency.

The remaining of the paper is organised as follows: Section 2 is a brief background on malware classification. The detailed visualisation method of API call sequences is explained in Section 3. Data source and experimental results are discussed in Section 4. Section 5 summarises the paper and outlines future work.

2 Related work

There are two main approaches to malware classification: static analysis and dynamic analysis. Visualisation is an emerging technology used in malware analysis. Next, we will give a brief introduction to the background related work.

2.1 Malware static analysis

Analysing malware without executing it is called static analysis. Kong and Yan [10] present a framework for automated malware classification based on structural function call graphs. After extracting fine grained features from the function call graph for each malware sample, they are clustered with discriminate distance metrics. Malware samples belonging to the same family are clustered together, and a distance margin separates clusters of different malware families

Hu *et al.* [11] present a malware classification system based on multifaceted features such as machine instruction and AV label features. This system extracts an aggregated feature vector from each malware program based on Opcode representation and the intelligence from antivirus software. It then trains a random forest classifier to learn the unique set of features that best distinguish between malware families. The selection of hyperparameters in the random forest is optimised using grid search. To improve the logloss result, they devised a new probability assignment strategy that concentrates all of the probability mass to a single family for a confident classification. Empirical results based on >10,000 malware samples showed a 99.8% accuracy in fivefold cross-validation and a logloss value of 0.0258.

Sun *et al.* [12] propose a malware family classification method based on static feature extraction from three aspects (hexadecimal bytecode features, Assembler code features, and PE structure view features). A random forest classifier achieved 93.56% F1. However, a large amount of time was spent on feature design, feature extraction, feature selection, and other preprocess stages. These steps were completed by adapting Scikit-learn machine learning libraries including SVM, decision tree, random forest and other algorithms.

Hassen and Chan [13] propose a linear time function call graph (FCG) vector representation based on function clustering. By using a novel technique to convert FCG representation into a vector representation, this method showed performance gains in addition to improved classification accuracy via the FCG based approach

[10, 14, 15]. In addition, the graph feature vector that is extracted from FCGs can be easily combined with other non-graph features.

In general, static analysis depends on machine-level or disassembled instructions, for instance, string signatures, byte-sequence n-grams, syntactic library calls, control flow graph, operational codes and their frequency distribution and so on. Sometimes, static analysis is efficient and enables greater code coverage than dynamic analysis because the binary code contains very useful information about the malicious behaviour of a program in the form of opcode sequence, functions and its parameters [8]. However, static analysis has some limitations. First, malware writers usually use obfuscation techniques to evade detection. Second, static analysis usually depends on a large signature database and thus only works effectively with known malware. Unknown malware is not detected by signature based methods.

2.2 Malware dynamic analysis

Network behaviour-based malware analysis is a typical method for dynamic analysis. Nari and Ghorbani [16] present a framework for automated malware classification based on network behaviour. Network traces are taken as input in the form of pcap files, from which the network flows are extracted. Then a behaviour graph is created to represent the malware network activities and the dependencies among network flow. From behaviour graphs, features like graph size, root out-degree, average out-degree, maximum out-degree and number of specific nodes are identified. These features are trained by WEKA [17] and their J48 decision tree performs better than other classifiers. Lim *et al.* [18] present a malware classification method based on clustering of flow features and sequence alignment algorithms for computing sequence similarity, which represents network behaviour of malware. They focus on analysing the sequence similarity between the sequence patterns of malware traffic flow generated by executing malware on the dynamic analysing system. Boukhtouta *et al.* [19] present two methods for malware classification: based deep packet inspection (DPI) and based IP packet headers. Moreover, they produced a comparison between these two approaches. But they have not studied possible evasion from malware trying to avoid detection at the network level. Although network activity effectively identifies some types of malware such as spyware and botnets, is not suitable for identifying malware with little or no network activity.

There is previous research on malware classification based on dynamic API call sequences. Lin *et al.* [20] extracted the function call word for each analysis report, using the bag-of-words model to generate a high-dimensional feature space corpus. Then, they used term frequency inverse document frequency (TF-IDF) and principal component analysis (PCA) algorithm to reduce the dimension of feature dataset. Finally, machine learning techniques were applied to classify the malware families. Pektaş and Acarman [21] propose mining and searching n-gram over the API call sequence to discover episodes representing behaviour-based features of malware. Then, using a Vote Experts algorithm to extract malicious API patterns. Finally, the classification model was built by applying online machine learning algorithms. The model is trained and tested with 17,400 malware samples and achieved a classification accuracy of 98%.

Pektaş and Acarman [22] present a method of malware classification based on multi dynamic feature integration. They extract the behaviour features of malware, such as the file system, network, registry activities observed during the execution traces and n-gram modelling over API call sequences. They evaluated five online machine learning algorithms with distributed and scalable architecture on 17,900 samples belonging to 51 families. CW algorithm had the best performance with training and testing accuracy of 94 and 92.5%, respectively.

In general, dynamic analysis extracts features while executing malware. These features include function call monitoring, function parameter analysis, information flow tracking, instruction traces and stain trace and so on [23]. In contrast to static analysis, dynamic analysis is robust to obfuscation and polymorphic techniques [24]. However, there are also limitations to dynamic analysis. First of all, each malware sample must be executed within a secure and controlled environment in order to monitor its behaviour. Differences between the secure environment and a real runtime environment, however, may cause the malware may behave differently, causing an inexact behaviour log. Secondly, some actions of malware are activated or triggered under some certain conditions, for instance, the system date and time, or some particular inputs by end users, which may not be triggered by the secure virtual environment. Finally, in dynamic analysis, guaranteeing the complete execution path coverage of malware is another challenging problem.

2.3 Malware visualisation

Recently, various visualisation techniques for malware analysis have been proposed to support human analysts to quickly assess and classify new malware samples. Malware visualisation includes two main categories: visualisation based on static analysis or dynamic analysis.

Nataraj *et al.* [25] propose a method for visualising and classifying malware using grey-scale image processing techniques. A K-nearest neighbour (KNN) technique with Euclidean distance method is used for malware classification. Though it is a very fast method compared to others, its limitation is that an attacker can adopt countermeasures to beat the system because this method uses global image based features.

Han and Lim *et al.* [26] propose a malware analysis method that uses visualised images and entropy graphs. Windows PE binary files are converted into bitmap images. After the bitmap image conversion, they calculate the entropy value of each line of bitmap images and generates entropy graphs to analyse the similarities of original binary files. In addition, binary files in the Windows PE format are divided into sections for further analysis. Finally, the bitmap images and entropy graphs are stored in the database as malware features that are used to detect and classify malwares by calculating similarities of entropy graphs. However, malware with packed binary files usually have high entropy values and are difficult to classify using entropy graphs and similarity calculation.

Wang *et al.* [27] got the first place in the Kaggle Microsoft Malware Classification Challenge (BIG 2015) [28] by using visualisation technology. Inspired by the authors [25, 29], they extract a grey-scale image from.asm file. Then they combine features including byte 4-gram instruction counts, function names and derived assembly features, assembly Opcode n-grams, disassembled code segment counts and disassembled code file pixel intensity. Finally, they use a Gradient Boosting model(Xgboost) to do multiclass classification with a softmax objective and achieve 0.9983 accuracies with 0.0031 logloss on 4-fold cross validation.

Compared to static visualisation, there is little work on dynamic visualisation. Trinius *et al.* [30] use visualisation to enhance comprehension of malicious software behaviour. They use treemaps and thread graphs to display the actions of the executable and to help a security analyst identify and classify malicious behaviour. Zhuo and Nadjin [31] develop a processing and visualisation tool, MalwareVis, that filters, selects, visualises, and compares malware network traces. It reads pcap files, processes them into streams and generates cell views for the user-selected malware entities interactively.

3 Dynamic API call sequences visualisation

Our visualised malware analysis consists of three steps, (see Fig. 1). In Step 1, we run the malware samples in a sandbox and collect the dynamic analysis report, and then according to the start-up time to extract the API call sequences and the start/end time of the program. In Step 2, feature images are generated according to the colour mapping rules, API category, number of times that each category occurs in per unit time. In Step 3, we train a CNN on feature images to obtain a classifier that identifies unknown malware. In the following sections, each step will be explained in detail.

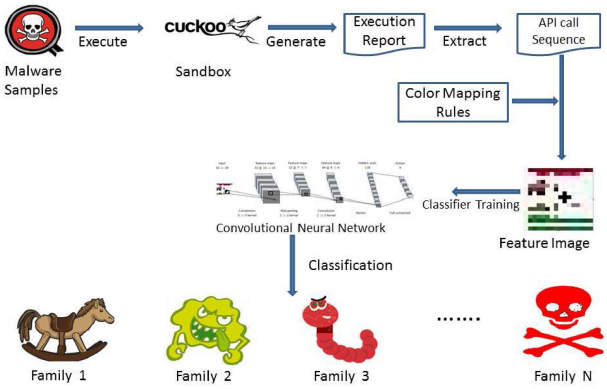

Fig. 1 *Overview of dynamic API call sequences visualisation*

Table 1 Overview of API categories partitions

No.	Category name	Partition criterion
1	networking	API functionality
2	register	
3	service	
4	file	
5	hardware and system	
6	message	
7	process and thread	
8	system	
9	Shellcode	malicious intentions
10	Keylogging	
11	Obfuscation	
12	password dumping and password hash	
13	anti-debugging and anti-reversing	
14	handle manipulation	
15	high risk	high frequency with sensitive function
16	other	appearing in the experiment but not in the above categories

3.1 Extract API call sequence

Submit the executable samples to a sandbox, e.g. Cuckoo Sandbox [32], which is open source software for automated analysis of suspicious files. The sandbox generates a report in a controlled virtual environment. Then the API call sequences and the calling time, as well as the start and end time of the program are extracted.

3.2 API category partition

Because of the extremely large number of APIs, considering all of them all would provide little to no meaningful information for malware identification. So, we only select some critical APIs that can be classified by functionality and malicious intentions into 14 different categories

- By functionality, can divide APIs into Network, Registry, Service, File, Hardware, Message, Process/Thread and System.
- By malicious intent, can divide APIs into Shellcode, Keylogging, Obfuscation, Password dumping/Password hashing, Anti-Debugging/Anti-Reversing and Handle Manipulation.

There may exist intersections between different categories. For example

- DWORD GetAdaptersInfo (_Out_ PIP_ADAPTER_INFO *pAdapterInfo*, _Inout_ PULONG *pOutBufLen*)

In some cases, this API is used to obtain the network adapter in a system, so its functionality belongs to the *Network* category. In other cases, this API is used to gather MAC addresses to check VMW are as part of anti-virtual machine techniques. Therefore, from the perspective of the malicious intention, this API also belongs to the *Anti-Debugging/Anti-Reversing* category.

Here, we not only categorise APIs according to the functionality and malicious intention but also take into account the frequency of different APIs appearing in malware. Romanian Security Team [33] created a report that counted the imported APIs from 549,035 malware samples. In our experiment, we select those APIs from the report that were imported >10,000 times, which we call 'High Frequency' and denote with the symbol 'F'. We label the union of the 14 functional and malicious categories as 'Sensitive' with the symbol 'S'. We label APIs that fall under both 'High Frequency' and the 'Sensitive' as 'High Risk' with the symbol 'R'. We label any API that is not labelled as 'Sensitive' as 'Other' category with the symbol 'O'. An overview of the API categories is shown in Table 1, and the formal expressions are as follows:

- $\boldsymbol{U}$ is the set of API that appears in the domain.
- $\boldsymbol{F}$ is the set of API that imported more than ten thousand times.
- $\boldsymbol{S}$ is the set of API that represents the union of the previous 14 categories. That is to say, $\boldsymbol{S}$ = Networking ∪ Register ∪ Service ∪ File ∪ Hardware ∪ Message ∪ Process and Thread ∪ System ∪ Shellcode ∪ Keylogging ∪ Obfuscation ∪ Password dumping and Password Hash ∪ Anti-debugging and Anti-Reversing ∪ Handle Manipulation.
- $\boldsymbol{R}$ is the set of API that appears in the set of $\boldsymbol{F}$, as well as in the set of $\boldsymbol{S}$. That is, $\boldsymbol{R} = \boldsymbol{F} \cap \boldsymbol{S}$.
- $\boldsymbol{O}$ is the set of API that lies in $\boldsymbol{U}$ but not in $\boldsymbol{S}$. That is, $\boldsymbol{O} = \boldsymbol{U} \setminus \boldsymbol{S}$.

3.3 Colour mapping and feature image generation

We divide the run time of the program into N equal parts (for instance, $N = 16$) and each part represents a unit time (see (1)).

$$\text{unit time} = \frac{\text{ET(program end time)} - \text{ST(program start time)}}{N} \quad (1)$$

Since the run time of each sample differs, the unit times for each are also different. According to the total number of occurrences of each API category in a unit time, different API categories are mapped to different colours. The formal representation of the colour map rules can be represented by the following equation:

$$y = f(c, t) \quad c \in \text{API category}, \quad t \in [0, +\infty) \quad (2)$$

Among them, the input parameter c represents the API category, t represents the number of times this API category is called in the unit time and the output value y is the corresponding colour value.

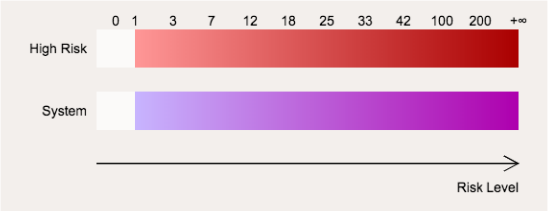

Fig. 2 *Colour mapping rule samples for the system and high risk category, the more times API called in the unit time, higher lightness of this colour to present*

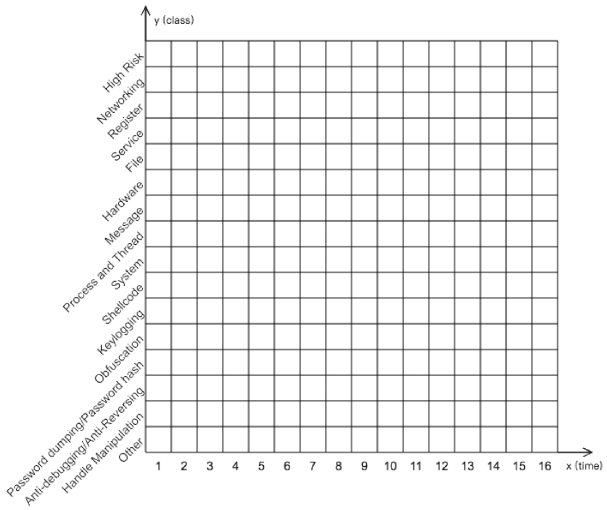

Fig. 3 *Feature image with 16 × 16 pixels, horizontal x-axis represents the time, and the vertical y-axis represents the API categories*

Parameter c determines the mapping to a certain colour, and t is related to the brightness of this colour. Larger t means the API is called more, and the risk level is higher. We choose high lightness of this colour to present. If one API category does not appear in the unit time, we represent it with white. For instance, we use purple and red to represent the *System* category and *High Risk*s category, respectively. The mapping rules are illustrated in Fig. 2.

We divided the APIs into 16 categories and selected 16 unit times, so 16 × 16 pixel size feature images are used to represent each malware's behaviour. The image's horizontal axis represents the time, and the vertical axis represents the API categories (see Fig. 3). Then we apply the colour mapping rules, as shown in Table 2 of the, to colour each pixel of the feature image. The detailed mapping process from API sequences to feature images is given in Algorithm 1 (see Fig. 4).

Feature images generated with colour mapping rules have the following advantages:

(i) Malware variants from the same family call very similar API categories. For example, worms often involve the *File* and *NetWorking* APIs to copy and propagate themselves; Spyware often involves *Keylogging*, *Password dumping/Hash* and *NetWorking* APIs to monitor and send the content of the user input. Thus, because the y-axis of a feature image represents the API category, feature images from the same malware family are very similar in the horizontal direction.

(ii) Malware variants from the same family have a very similar calling sequence. The x-axis of a feature image represents time, so feature images from the same family are very similar in the vertical direction.

(iii) Colour mapping rules are theoretically robust to equivalent API replacement technology and garbage insertion technology. These two obfuscation technologies not only quickly generate variants by changing API call sequence but also bypass signature-based detection. Equivalent API replacement technology usually uses APIs belonging to the same category, so the malware variants created by API replacement map to the same colour according to the colour mapping rules. Garbage insertion technology inserts garbage APIs into the original API sequence. Often, the inserted APIs are 'Safe-APIs'. Because colour mapping rules only consider critical 'High-risk' APIs, the inserted 'Safe-API' does not affect the input parameters of the colour mapping rules.

(iv) Colour mapping rules have scale flexibility. We can flexibly modify the scale of the mapping rules by the following four methods.

- Adjust the number of intervals. We divide the range of API calls in unit time [0,+∞) into 11 intervals. As we add more intervals,

Table 2 Hexadecimal color values for different API categories

API category occurrences per unit time	0	(0,3]	(3,7]	(7,12]	(12,18]	(18,25]	(25,33]	(33,42]	(42,100]	(100,200]	(200,+∞]
networking	#FFFFFF	#FFC1E0	#FFAAD5	#FF95CA	#FF79BC	#FF60AF	#FF359A	#FF0080	#F00078	#D9006C	#BF0060
register	#FFFFFF	#FFBFFF	#FFA6FF	#FF8EFF	#FF77FF	#FF44FF	#FF00FF	#E800E8	#D200D2	#AE00AE	#930093
service	#FFFFFF	#FFDAC8	#FFCBB3	#FFBD9D	#FFAD86	#FF9D6F	#FF8F59	#FF8040	#FF5809	#F75000	#D94600
file	#FFFFFF	#D3FF93	#CCFF80	#B7FF4A	#A8FF24	#9AFF02	#8CEA00	#82D900	#73BF00	#64A600	#548C00
hardware and system	#FFFFFF	#CAFFFF	#BBFFFF	#A6FFFF	#4DFFFF	#00FFFF	#00E3E3	#00CACA	#00AEAE	#009393	#005757
message	#FFFFFF	#C1FFE4	#ADFEDC	#96FED1	#4EFEB3	#1AFD9C	#02F78E	#02DF82	#01B468	#019858	#01814A
process and thread	#FFFFFF	#D6D6AD	#CDCD9A	#C2C287	#B9B973	#AFAF61	#A5A552	#949449	#808040	#707038	#616130
system	#FFFFFF	#DCB5FF	#D3A4FF	#CA8EFF	#BE77FF	#B15BFF	#9F35FF	#921AFF	#8600FF	#6E00FF	#5B00AE
Shellcode	#FFFFFF	#FFFF6F	#FFFF37	#F9F900	#E1E100	#C4C400	#A6A600	#8C8C00	#737300	#5B5B00	#5B5B00
Keylogging	#FFFFFF	#FFE66F	#FFE153	#FFDC35	#FFD306	#EAC100	#D9B300	#C6A300	#AE8F00	#977C00	#796400
Obfuscation	#FFFFFF	#D8D8EB	#C7C7E2	#B8B8DC	#A6A6D2	#9999CC	#8080C0	#7373B9	#5A5AAD	#5151A2	#484891
password dumping/hash	#FFFFFF	#97CBFF	#84C1FF	#66B3FF	#46A3FF	#2894FF	#0080FF	#0072E3	#0066CC	#005AB5	#004B97
anti-debugging/reversing	#FFFFFF	#B9B9FF	#AAAAFF	#9393FF	#7D7DFF	#6A6AFF	#4A4AFF	#2828FF	#0000E3	#0000C6	#0000C6
handle manipulation	#FFFFFF	#FFD1A4	#FFC78E	#FFBB77	#FFAF60	#FFA042	#FF9224	#FF8000	#EA7500	#D26900	#BB5E00
high risk	#FFFFFF	#FF9797	#FF7575	#FF5151	#FF2D2D	#FF0000	#EA0000	#CE0000	#AE0000	#930000	#750000
other	#FFFFFF	#93FF93	#79FF79	#53FF53	#28FF28	#00EC00	#00DB00	#00BB00	#00A600	#009100	#007500

we can detect smaller differences in API calls in unit time at the cost of a larger mapping scale.

- Adjust the interval size. Smaller intervals better reflect the difference in the number of times an API is called in unit time. In this experiment, the interval size gradually increases to capture differences when the API is called only a few times. Smaller interval size means more intervals are required, so the scale increases.
- Adjust the number N of unit times (in the experiment, $N = 16$). As N increases, the width of the feature image increases and each unit of time becomes shorter at the cost of larger scale.
- Adjust the API category. As we add more API categories the height of the feature image increases, and we achieve more granularity at the cost of larger scale.

Require: API call sequence, ST(program start time), ET(program end time)
Ensure: Image
1: $\{S_k\}$ is an empty set to store API call sequence fragment, and $k = 16$;
2: $\{C_j\}$ is an empty set to store color value for each API category, and $j = 16$;
3: **function** APISEQUENCE2IMAGE($APIcallsequence, ST, ET$)
4: $\{S_k\} \leftarrow$ SplitAPISequence($APIcallsequence, ST, ET$); ▷ Get API call sequence of each unit time
5: **for** each $k \in [0, 15]$ **do**
6: $\{C_j\} \leftarrow$ CountAPIForEachCategory(S_k); ▷ Counting the number of occurrences of each API category
7: **for** each $j \in [0, 15]$ **do**
8: Image[k][j] $\leftarrow$ ColorMapping(C_j);
9: return Image

Fig. 4 *Algorithm 1: Mapping API sequence to feature image*

3.4 Feature extraction and learning

Recently, researchers have been attempting to use deep learning models to improve malware classification. Dahl *et al.* [34] first studied deep learning for malware classification in the context of dynamic analysis. DeepSign [35] is another project that applies deep learning on Windows malware signature generation and classification. Later, Kolosnjaji *et al.* [36] constructed a neural network based on the convolutional and recurrent network to identify the best features for classification.

In this paper, we adopt CNN to classify malware visualisation images because CNN is designed to process data that come in the form of multiple arrays. A CNN architecture is formed by a stack of distinct layers that transform the input image volume into an output volume (see Fig. 5*a*). The convolutional layer is the core building block of a CNN. The layer's parameters consist of a set of learnable filters that have a small receptive field. During the forward pass, each filter is convolved across the width and height of the input volume (see Fig. 5*b*). Another important layer of CNN is the pooling layer, which is a form of non-linear down-sampling. Max pooling is the most common non-linear functions used (see Fig. 5*c*). The pooling layer serves to progressively reduce the spatial size of the representation, reducing the number of parameters and amount of computations in the network.

The CNN architecture we used is shown in Fig. 5*d*. Feature images are RGB colourised, so the architecture takes images with dimension $16 \times 16 \times 3$ as input. The architecture has two convolutional layers, and the first convolutional layer is followed by a max-pooling layer. The number of filters of the two convolutional layers is 32 and 64, and the kernel size of the two convolutional layers is 3×3 and 2×2, respectively. To improve the training speed, ReLU is used to help the networks converge faster. Furthermore, Dropout [37] is used in order to prevent over-fitting, and a softmax layer is used to output the malware family label.

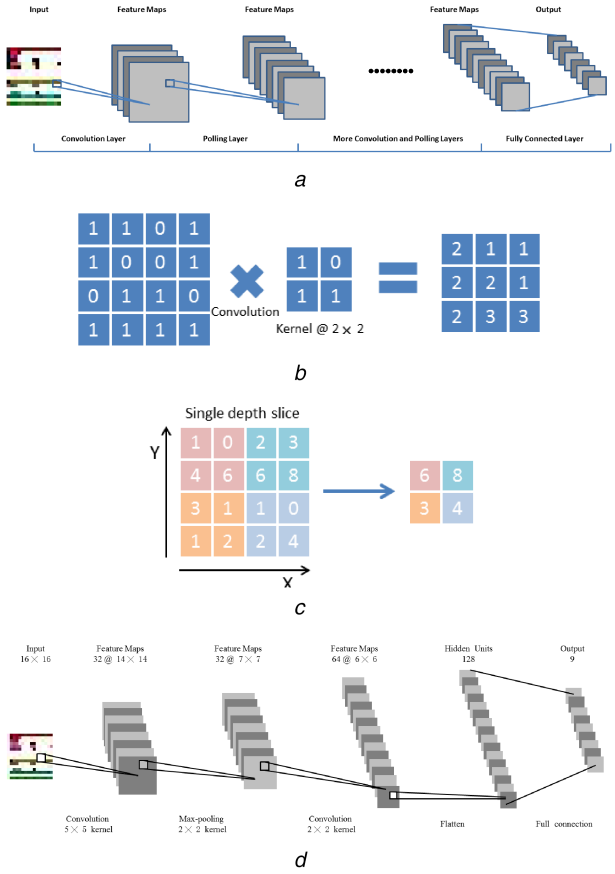

Fig. 5 *Typical architecture of CNN and it's application for malware feature images classification*
(a) Typical architecture of CNN, ***(b)*** Convolution operation, ***(c)*** Max pooling with a 2×2 filter and stride = 2, ***(d)*** CNN architecture for malware feature images classification

Table 3 Experiment dataset and description

#	Malware families	Description
1	Swizzor	Swizzor is a type of malware flies that under the radar to deliver unsolicited advertisements, modifying browser setting without user permission.
2	Vundo	Vundo is either a Trojan horse or a computer worm that is known to cause Pop-up advertising for rogue antispyware programs.
3	Spybot	Spybot is a type of worm that usually arrives on a computer through peer-to-peer file sharing, specifically through the Kazaa file sharing network. Its many variants sometimes have other ways of spreading.
4	Ransom	Ransom is a type of malware that can be covertly installed on a computer without knowledge or intention of the user that restricts access to the infected computer system in some way, and demands that the user pay a ransom to the malware operators to remove the restriction.
5	Ramnit	Ramnit is a type of virus that infects Windows executable files and HTML files. It can also give a malicious hacker access to your computer. It spreads through infected removable drives, such as USB flash drives.
6	Lollipop	Lollipop is a type of adware program that shows advertisements as you browse the web. It can also redirect your search engine results, monitor what you do on your computer, download applications, and send information about your computer to a hacker.
7	Kelihos	Kelihos is a type of trojan that can give a malicious hacker access and control of your computer. The family spreads by sending spam emails that have links to other malware.
8	Delf	Delf is a type of trojan that reports and intercepts Internet traffic and may also download unwanted applications onto your computer.
9	Banker	Banker is a type of data-stealing trojans that can capture your online banking details, such as your login credentials and account numbers, and then send this information to a malicious hacker.

4 Experiments

4.1 Experimental data

The experimental dataset is from the Virus Share community [38], which provides security analysts with a rich repository of malware. Our data consists of 9 malware families with 1000 variants in each family. The malware families and their descriptions are shown in Table 3.

We construct an experimental environment consisting of an emulator, malware database, image generation and CNN classifier. The emulator machine has an Intel i7-3700 processor with 8 GB main memory running the Ubuntu16.04 operating system. The dynamic execution traces are monitored through the Cuckoo Sandbox2.0 [32]. Another machine used for image generation and CNN training and testing has an Intel i7-3700 processor with 8 GB main memory and runs the Windows7 operating system.

4.2 Experimental results and analysis

Each feature image is 16×16 pixels. Fig. 6 shows example feature images generated from nine different malware families. We randomly select a dataset with 8100 samples for training (900 samples from each malware family) and 900 samples for testing (100 samples from each malware family). Then we use 10-fold cross validation to obtain 10 classifiers. During the model selection, we use two metrics, accuracy and logarithmic loss, to measure the model quality. Accuracy is the proportion of the number of samples that are correctly classified to the total number of samples (see (3))

$$\text{accuracy} = \frac{1}{N}\sum_{i=1}^{N} 1(f(x_i) = (y_i)) \tag{3}$$

where N is the total number of observations, $f(x_i)$ represents the prediction label for the ith sample, y_i is the true label of the ith sample and $1(\bullet)$ is the indicator function. When '$\bullet$' is true or false, $1(\bullet)$ are 1 or 0, respectively. See Table 4 in the for classification accuracy of each family.

Although classification accuracy reflects the scores that been correctly predicted, it alone is usually not enough to verify the robustness of the classifier. Logarithmic loss (logloss) is a soft accuracy measurement that incorporates the concept of probabilistic confidence. *logloss* is the cross entropy between the distribution of the true labels and the predicted probabilities (see (4)).

$$\text{logloss} = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij}) \tag{4}$$

where N is the number of observations, M is the number of class labels, the log is the natural logarithm, y_{ij} is 1 if the observation i is in class j and 0 otherwise and p_{ij} is the predicted probability that observation i is in class j. The *logloss* values for k-fold CNN classifiers are shown in Fig. 7*a*.

In terms of accuracy, Fig. 7*a* and Table 4 show that 3-fold, 6-fold and 7-fold models have the same accuracy of 99.0% on average. However, the 7-fold model performs best on *logloss*, so we select the 7-fold CNN classifier as the best final classifier used to classify all samples. The predictive confusion matrix is given in Table 5.

For classification tasks, *true positives* (TP), *true negatives* (TN), *false positives* (FP), and *false negatives* (FN) are common classification evaluation indicators. The terms *positive* and *negative* refer to the classifier's prediction, and *true* or *false* refers to whether the prediction corresponds to the external judgement.

We use five indicators, true positive rate (TPR), false positive rate (FPR), F1, precision and recall, to evaluate our models. TPR (see (5)) is how many correct positive results occur among all positive samples available during the test. Similarly, FPR (see (6)) describes how many incorrect positive results occur among all negative samples available during the test. Precision (see (7)), also named positive predictive value, is the fraction of relevant instances among the retrieved instances. Recall (see (8)), also known as sensitivity, is the fraction of relevant instances that have been retrieved over the total amount of relevant instances. F1, (see (9)), incorporates both the precision and the recall of the test by calculating their harmonic average

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{5}$$

$$\text{FPR} = \frac{\text{FP}}{\text{TN} + \text{FP}} \tag{6}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{7}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{8}$$

$$\text{F1} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{9}$$

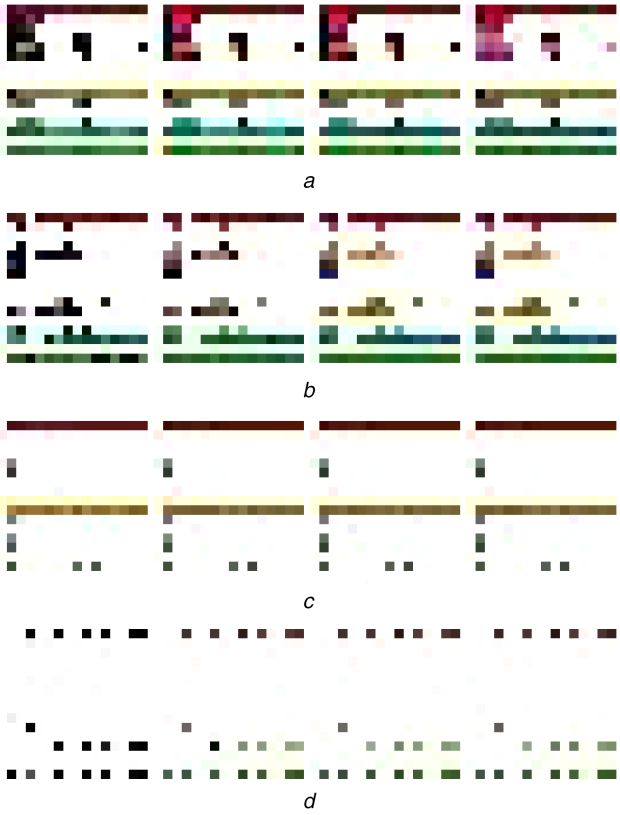

Fig. 6 *Feature images for different malware families*
(a) Delf family, ***(b)*** Lollipop family, ***(c)*** Vundo family, ***(d)*** Spybot family

Table 4 Classification accuracy of 10-fold cross validation

Malware family k-fold	1	2	3	4	5	6	7	8	9	10
Swizzor	0.99	1	1	1	1	1	1	1	1	1
Vundo	1	0.90	0.99	1	1	1	1	1	1	0.93
Spybot	1	1	1	1	1	1	1	1	1	1
Ransom	0.62	0.90	0.97	0.89	1	0.97	0.91	0.91	0.98	0.74
Ramnit	1	1	1	1	1	1	1	1	1	1
Lollipop	1	1	1	1	1	1	1	1	1	1
Kelihos	0.91	0.93	0.97	0.99	0.88	0.94	1	0.93	0.93	0.98
Delf	1	0.96	0.98	1	1	1	1	1	1	1
banker	0.90	1	1	1	1	1	1	1	0.90	0.80
average	0.936	0.966	0.990	0.987	0.987	0.990	0.990	0.982	0.979	0.939

The final classification results are shown in Table 6. The best 7-fold CNN classifier has a great performance with F1 of 99.324% and a false positive rate of only 0.085%.

4.3 Comparing with GIST and KNN

Nataraj *et al.* [25] also proposed a method for visualising and classifying malware by reading the malware binary as a vector of 8 bit unsigned integers and then organising it into a 2D array. The 2D array is then visualised as a grey-scale image in the range [0, 255]. The texture feature that they used is the GIST descriptor [39], which is a global description commonly used in image recognition systems, such as scene classification, object recognition and large scale image searching. Finally, they use KNN with Euclidean distance for classification. Given an input image, a GIST descriptor is computed by the following three steps:

(i) Convolve the image with V Gabor filters at M scales and N orientations, where $V = M \times N$. Then generate V feature maps of the same size as the input image.
(ii) Divide each feature map into $S \times S$ regions (e.g. 3×3 blocks), and then average the feature values within each region.
(iii) Concatenate the $S \times S$ averaged values of all V features maps resulting in an $S \times S \times V$ dimension descriptor.

During the experiment, we set $M = 4$, $N = 5$ and $S = 3$ to get a GIST feature with $20 \times 3 \times 3 = 180$ dimensions. Use the same training and testing dataset to do 10-fold cross validation, and get 10 KNN models. The accuracy and logloss of the 10 KNN models are shown in Fig. 7*b*. We found the 2-fold KNN model reaches the highest accuracy of 98.7889% and lowest logloss 0.0639. Therefore, the 2-fold KNN model is selected to classify all malware samples. The detailed results are shown in Table 7. Tables 6 and 7 show that the average performance of the CNN classifier is superior on average to the KNN classifier for each indicator.

Because CNN receives images directly while the KNN model requires time-consuming processing to extract GIST features from images, CNN is more efficient. Table 8 shows the performance of CNN and KNN with varying sample size, where the time unit is in seconds.

Figs. 8*a* and *b* show the precision and recall a comparison between the CNN and KNN classifier for each malware family. In most cases, CNN classifier performs better than KNN, except for those families with few malware samples (e.g. the Kelihos family),

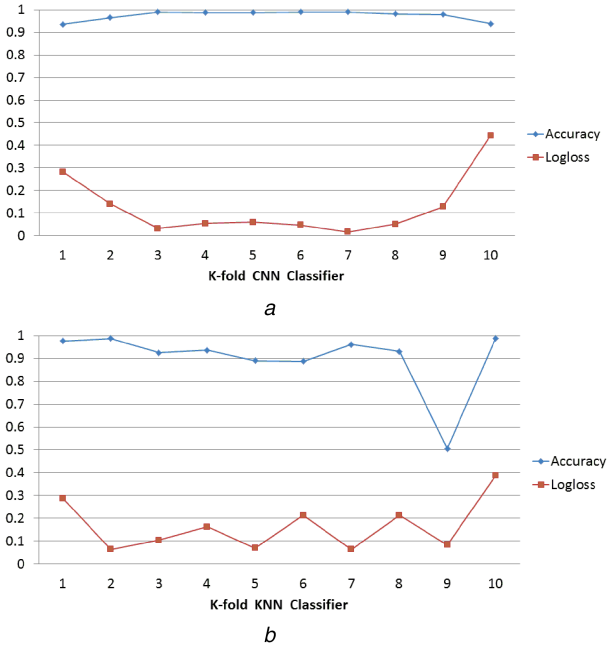

Fig. 7 *Accuracy and logloss value for k-fold CNN and KNN classifiers*
(a) *Accuracy* and *logloss* for k-fold CNN classifiers ($k = 1, 2, \ldots, 10$), ***(b)*** *Accuracy* and *logloss* for k-fold KNN ($K = 3$) classifiers ($k = 1, 2, \ldots, 10$)

Table 5 Confusion matrix among different malware families when selecting the best 7-fold CNN classifier

Confusion Matrix	Swizzor	Vundo	Spybot	Ransom	Ramnit	Lollipop	Kelihos	Delf	Banker
Swizzor	1000	0	0	0	0	0	0	0	0
Vundo	0	979	0	0	0	0	21	0	0
Spybot	0	0	1000	0	0	0	0	0	0
Ransom	0	0	0	965	11	0	24	0	0
Ramnit	0	0	0	0	1000	0	0	0	0
Lollipop	0	0	0	0	0	1000	0	0	0
Kelihos	0	0	0	2	0	0	996	0	2
Delf	0	0	0	0	0	0	1	999	0
banker	0	0	0	0	0	0	0	0	1000

Table 6 Classification results when using the best 7-fold CNN classifier

Malware family	TPR	FPR	Precision	Recall	F1
Swizzor	1	0	1	1	1
Vundo	0.979	0	1	0.979	0.98939
Spybot	1	0	1	1	1
Ransom	0.965	0.00025	0.99793	0.965	0.98119
Ramnit	1	0.00138	0.98912	1	0.99453
Lollipop	1	0	1	1	1
Kelihos	0.996	0.00575	0.95585	0.996	0.97551
Delf	0.999	0	1	0.999	0.99950
banker	1	0.00025	0.99800	1	0.99900
average	0.993	0.00085	0.99343	0.993	0.99324

Table 7 Classification Result when using the best 2-fold KNN classifier

Malware family	TPR	FPR	Precision	Recall	F1
Swizzor	0.994	0	1	0.994	0.99699
Vundo	0.994	0.00463	0.96411	0.994	0.97883
Spybot	1	0	1	1	1
Ransom	0.997	0.00263	0.97896	0.977	0.97798
Ramnit	1	0.00013	0.99900	1	0.99950
Lollipop	1	0	1	1	1
Kelihos	0.963	0.00100	0.99176	0.963	0.97717
Delf	0.979	0.00363	0.97123	0.979	0.97510
banker	0.984	0.00163	0.98696	0.984	0.98548
average	0.990	0.00152	0.98800	0.988	0.98789

Table 8 Time performance comparison between CNN and KNN during malware prediction

Method total samples	900	1800	2700	3600	4500	5400	6300	7200	8100	9000
CNN	0.0017	0.0025	0.0031	0.0039	0.0045	0.0052	0.0060	0.0066	0.0073	0.0081
KNN	39.921	69.094	101.14	140.85	172.92	203.17	238.17	277.90	322.40	352.94

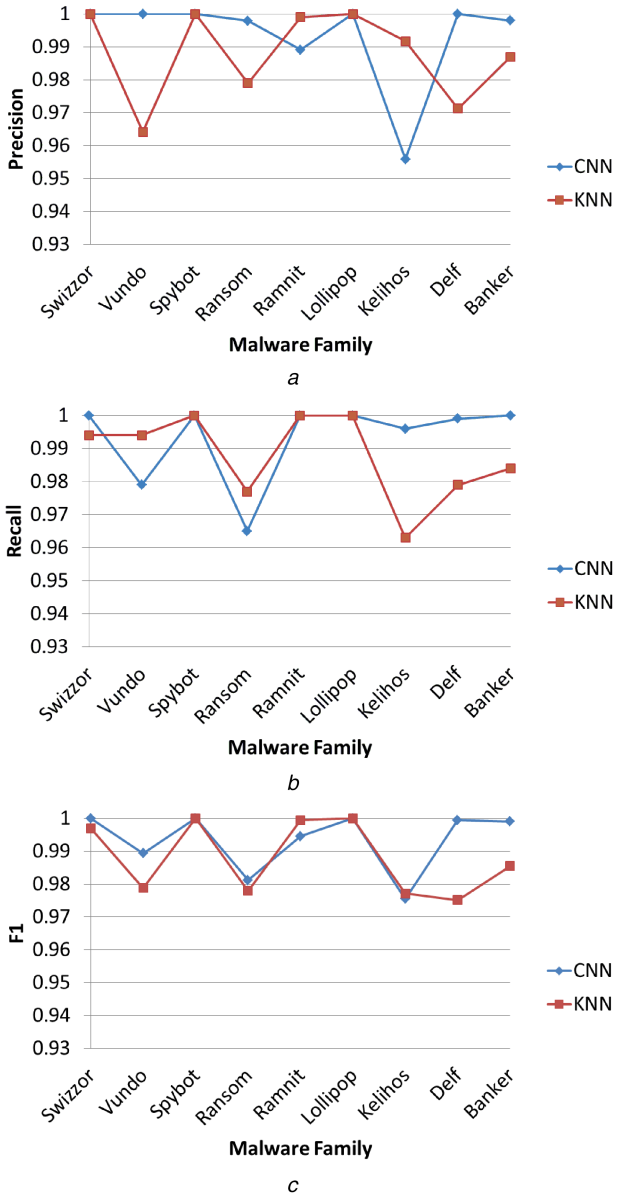

Fig. 8 *Precision, recall and recall a comparison between the CNN and KNN classifier in each malware family*
(a) Precision of CNN and KNN, ***(b)*** Recall of CNN and KNN, ***(c)*** F1 of CNN and KNN

where the precision of CNN is not as good as KNN. Fig. 8*c* shows the F1 comparison with similar results. Fig. 9 shows the ROC curve of CNN and KNN for each malware family. For the Swizzor, Spybot, Ramnit and Lollipop malware families, the ROC curves of CNN and KNN are coincident. And for the rest of malware families, CNN classifier is better than KNN.

5 Conclusion and future work

In this paper, we propose a visualisation and deep learning method for malware classification. We use dynamic analysis to extract API call and then generate feature images that represent the malware behaviour according to the colour mapping rules. Finally, CNN is used to classify the feature images. The experimental results show that visualisation and CNN are effective for malware classification.

In future work, other behaviour features extracted from dynamic and static analysis need to be mined further besides just API call sequences. Additionally, accuracy and robustness of malware identification can be improved with better dynamic execution. We will continue to collect more malware samples and apply our method to a larger dataset in order verify its effectiveness, as well. We also plan to explore the application of other deep learning algorithms in malicious code analysis.

6 Acknowledgments

This work was partially sponsored by the National Key Research and Development Program of China (2018YFB0704400, 2016YFB0700504, 2017YFB0701601), Shanghai Municipal Science and Technology Commission (15DZ2260301), Natural Science Foundation of Shanghai (16ZR1411200). The authors gratefully appreciate the anonymous reviewers for their valuable comments.

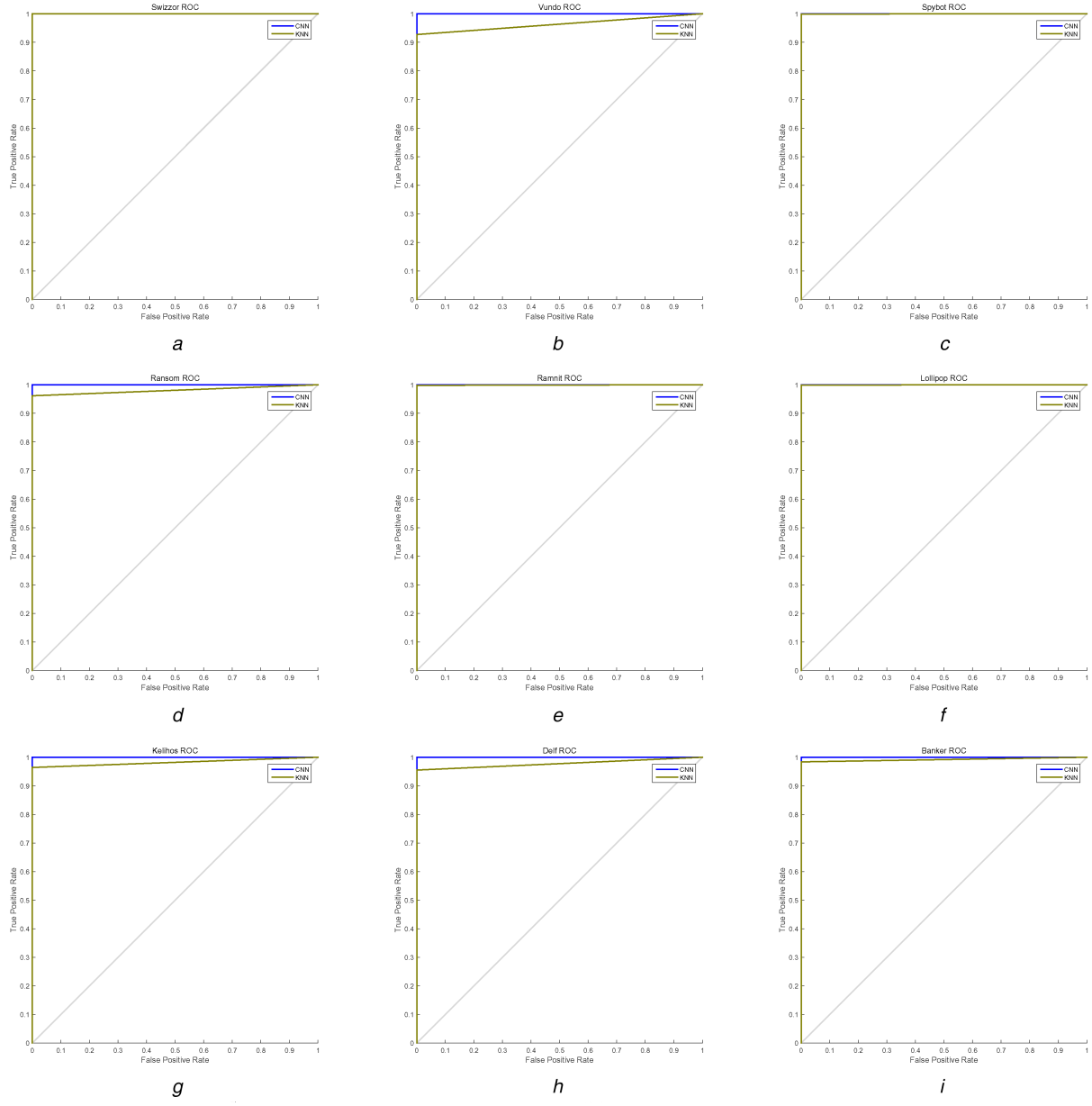

Fig. 9 *ROC curve of different malware families with CNN and KNN*

(a) Swizzor ROC, ***(b)*** Vundo ROC, ***(c)*** Spybot ROC, ***(d)*** Ransom ROC, ***(e)*** Ramnit ROC, ***(f)*** Lollipop ROC, ***(g)*** Kelihos ROC, ***(h)*** Delf ROC, ***(i)*** Banker ROC

7 References

[1] Kruegel, C.: 'Behavioral and structural properties of malicious code'. Malware Detection, MA, Sweden, 2007, pp. 63–83

[2] 'Symantec'. Available at https://www.symantec.com/security_response/publications/monthlythreatreport.jsp, accessed 10 May 2018

[3] Moser, A., Kirda, E., Kruegel, C.: 'Limits of static analysis for malware detection'. Twenty-Third Annual Computer Security Applications Conf. (ACSAC 2007), Miami Beach, FL, USA, 2007, pp. 421–430

[4] Gandotra, E., Bansal, D., Sofat, S.: 'Malware analysis and classification: a survey', *J. Inf. Secur.*, 2016, **5**, (2), pp. 56–64

[5] Shabtai, A., Moskovitch, R., Feher, C., *et al.*: 'Detecting unknown malicious code by applying classification techniques on OpCode patterns', *Secur. Inf.*, 2012, **1**, (1), p. 1

[6] Gandotra, E., Singla, S., Bansal, D., *et al.*: 'Clustering morphed malware using Opcode sequence pattern matching', *Recent Pat. Eng.*, 2018, **12**, (1), pp. 30–36

[7] Fan, Y., Ye, Y., Chen, L.: 'Malicious sequential pattern mining for automatic malware detection', *Expert Syst. Appl.*, 2016, **52**, (C), pp. 16–25

[8] Shijo, P., Salim, A.: 'Integrated static and dynamic analysis for malware detection', *Procedia Comput. Sci.*, 2015, **46**, pp. 804–811

[9] You, I., Yim, K.: 'Malware obfuscation techniques: A brief survey'. Proc. of the 2010 Int. Conf. on Broadband, Wireless Computing, Communication and Applications, Washington, USA, 2010, pp. 297–300

[10] Kong, D., Yan, G.: 'Discriminant malware distance learning on structural information for automated malware classification'. ACM Sigmetrics/Int. Conf. on Measurement and Modeling of Computer Systems, Pittsburgh, PA, USA, 2013, vol. 41, no. 1, pp. 347–348

[11] Hu, X., Jang, J., Wang, T., *et al.*: 'Scalable malware classification with multifaceted content features and threat intelligence', *IBM J. Res. Dev.*, 2016, **60**, (4), pp. 6:1–6:11

[12] Sun, B., Li, Q., Guo, Y., *et al.*: 'Malware family classification method based on static feature extraction'. 2017 3rd IEEE Int. Conf. on Computer and Communications (ICCC), Chengdu, China, 2017

[13] Hassen, M., Chan, P.K.: 'Scalable function call graph-based malware classification'. Conf. on Data and Application Security and Privacy, Scottsdale, Arizona, USA, 2017, pp. 239–248

[14] Kinable, J., Kostakis, O.: 'Malware classification based on call graph clustering', *J. Comput. Virol.*, 2010, **7**, (4), pp. 233–245

[15] Xu, M., Wu, L., Qi, S., *et al.*: 'A similarity metric method of obfuscated malware using function-call graph', *J. Comput. Virol. Hacking Tech.*, 2013, **9**, (1), pp. 35–47

[16] Nari, S., Ghorbani, A.A.: 'Automated malware classification based on network behavior'. Proc. of the 2013 Int. Conf. on Computing, Networking and Communications (ICNC), Washington, USA, 2013, pp. 642–647

[17] Hall, M., Frank, E., Holmes, G., *et al.*: 'The WEKA data mining software: an update', *SIGKDD Explor. Newsl.*, 2009, **11**, (1), pp. 10–18

[18] Lim, H., Yamaguchi, Y., Shimada, H., *et al.*: 'Malware classification method based on sequence of traffic flow'. 2015 Int. Conf. on Information Systems Security and Privacy (ICISSP), Washington, USA, 2016, pp. 1–8

[19] Boukhtouta, A., Mokhov, S.A., Lakhdari, N.E., *et al.*: 'Network malware classification comparison using DPI and flow packet headers', *J. Comput. Virol. Hacking Tech.*, 2016, **12**, (2), pp. 69–100

[20] Lin, C.T., Wang, N.J., Xiao, H., *et al.*: 'Feature selection and extraction for malware classification', *J. Inf. Sci. Eng.*, 2015, **31**, (3), pp. 965–992

[21] Pektaş, A., Acarman, T.: 'Malware classification based on API calls and behaviour analysis', *IET Inf. Sec.*, 2018, **12**, (2), pp. 107–117

[22] Pektaş, A., Acarman, T.: 'Classification of malware families based on runtime behaviors', *J. Inf. Secur. Appl.*, 2017, **37**, pp. 91–100
[23] Egele, M., Scholte, T., Kirda, E.*, et al.*: 'A survey on automated dynamic malware-analysis techniques and tools', *ACM Comput. Surv.*, 2008, **44**, (2), pp. 6:1–6:42
[24] Zhao, H., Xu, M., Zheng, N.*, et al.*: 'Malicious executables classification based on behavioral factor analysis'. Proc. of the 2010 Int. Conf. on e-Education, e-Business, e-Management and e-Learning, Washington, USA, 2010, pp. 502–506
[25] Nataraj, L., Karthikeyan, S., Jacob, G.*, et al.*: 'Malware images: visualization and automatic classification'. Proc. of the 8th Int. Symp. on Visualization for Cyber Security, New York, USA, 2011, pp. 4:1–4:7
[26] Han, K., Lim, J.H., Kang, B.*, et al.*: 'Malware analysis using visualized images and entropy graphs', *Int. J. Inf. Secur.*, 2015, **14**, (1), pp. 1–14
[27] https://github.com/xiaozhouwang/kaggle_Microsoft_Malware/blob/master/Saynotooverfitting.pdf, accessed 10 May 2018
[28] 'Kaggle'. Available at https://www.kaggle.com/c/malware-classification, accessed 10 May 2018
[29] https://sarvamblog.blogspot.com, accessed 10 May 2018
[30] Trinius, P., Holz, T., Freiling, F.*, et al.*: 'Visual analysis of malware behavior using treemaps and thread graphs'. 6th Int. Workshop on Visualization for Cyber Security, 2009, VizSec 2009, Atlantic City, USA, 2009, pp. 33–38
[31] Zhuo, W., Nadjin, Y.: 'Malwarevis: entity-based visualization of malware network traces'. Proc. of the Ninth Int. Symp. on Visualization for Cyber Security, New York, USA, 2012, pp. 41–47
[32] https://www.cuckoosandbox.org, accessed 10 May 2018
[33] https://rstforums.com/forum/topic/95273-top-maliciously-used-apis/, accessed 10 May 2018
[34] Dahl, G.E., Stokes, J.W., Deng, L.*, et al.*: 'Large-scale malware classification using random projections and neural networks'. 2013 IEEE Int. Conf. on Acoustics, Speech and Signal Processing, Vancouver, BC, Canada, 2013, pp. 3422–3426
[35] David, O.E., Netanyahu, N.S.: 'Deepsign: deep learning for automatic malware signature generation and classification'. 2015 Int. Joint Conf. on Neural Networks (IJCNN), Killarney, Ireland, 2015, pp. 1–8
[36] Kolosnjaji, B., Zarras, A., Webster, G.*, et al.*: 'Deep learning for classification of malware system call sequences'. AI 2016: Advances in Artificial Intelligence: 29th Australasian Joint Conf., Hobart, Australia, 5–8 December 2016, pp. 137–149
[37] Srivastava, N., Hinton, G., Krizhevsky, A.*, et al.*: 'Dropout: a simple way to prevent neural networks from overfitting', *J. Mach. Learn. Res.*, 2014, **15**, (1), pp. 1929–1958
[38] https://virusshare.com/, accessed 10 May 2018
[39] Oliva, A., Torralba, A.: 'Modeling the shape of the scene: a holistic representation of the spatial envelope', *Int. J. Comput. Vision*, 2001, **42**, (3), pp. 145–175