Effective Malware Detection based on Behaviour and Data Features

Zhiwu Xu, Cheng Wen, Shengchao Qin, and Zhong Ming

College of Computer Science and Software Engineering, Shenzhen University, China
{xuzhiwu,sqin,mingz}@szu.edu.cn, 2150230509@email.szu.edu.cn

Abstract. Malware is one of the most serious security threats on the Internet today. Traditional detection methods become ineffective as malware continues to evolve. Recently, various machine learning approaches have been proposed for detecting malware. However, either they focused on behaviour information, leaving the data information out of consideration, or they did not consider too much about the new malware with different behaviours or new malware versions obtained by obfuscation techniques. In this paper, we propose an effective approach for malware detection using machine learning. Different from most existing work, we take into account not only the behaviour information but also the data information, namely, the opcodes, data types and system libraries used in executables. We employ various machine learning methods in our implementation. Several experiments are conducted to evaluate our approach. The results show that (1) the classifier trained by Random Forest performs best with the accuracy 0.9788 and the AUC 0.9959; (2) all the features (including data types) are effective for malware detection; (3) our classifier is capable of detecting some fresh malware; (4) our classifier has a resistance to some obfuscation techniques.

1 Introduction

Malware, or malicious software is a generic term that encompasses viruses, trojans, spywares and other intrusive codes. They are spreading all over the world through the Internet and are increasing day by day, thus becoming a serious threat. According to the recent report from McAfee [1], one of the world's leading independent cybersecurity companies, there are more than 650 million malware samples detected in Q1, 2017, in which more than 30 million ones are new. So the detection of malware is of major concern to both the anti-malware industry and researchers.

To protect legitimate users from these threats, anti-malware software products from different companies provide the major defence against malware, such as Comodo, McAfee, Kaspersky, Kingsoft, and Symantec, wherein the signature-based method is employed. However, this method can be easily evaded by malware writers through the evasion techniques such as packing, variable-renaming, and polymorphism [2]. To overcome the limitation of the signature-based method, heuristic-based approaches are proposed, aiming to identify the malicious behaviour patterns, through either static analysis or dynamic analysis. But the

increasing number of malware samples makes this method no longer effective. Recently, various machine learning approaches like Support Vector Machine, Decision Tree and Naive Bayes have been proposed for detecting malware [3]. These techniques rely on data sets that include several characteristic features for both malware and benign software to build classification models to detect (unknown) malware. Although these approaches can get a high accuracy (for the stationary data sets), it is still not enough for malware detection. On one hand, most of them focus on the behaviour features such as binary codes [4–6], opcodes [6–8] and API calls [9–11], leaving the data information out of consideration. While a few of them do consider the data information, but only simple features like strings [12, 13] and file relations [14, 15]. On the other hand, as malware continues to evolve, some new and unseen malware have different behaviours and features. Even the obfuscation techniques can make malware difficult to detect. Hence, more datasets and experiments are still needed to keep the detection effective.

In this paper, we propose an effective approach to detecting malware based on machine learning. Different from most existing work, we take into account not only the behaviour information but also the data information. Generally, the behaviour information reflects what the software intends to behave, while the data information indicates which datas the software intends to perform on or how data are organised. Our approach tries to learn a classifier from existing executables with known categories first, and then uses this classifier to detect new, unseen executables. In detail, we take the opcodes, data types and system libraries that are used in executables, which are collected through static analysis, as representative features. As far as we know, our approach is the first one to consider data types as features for malware detection. Moreover, in our implementation, we employ various machine learning methods, such as K-Nearest Neighbor, Native Bayes, Decision Tree, Random Forest, and Support Vector Machine to train our classifier.

Several experiments are conducted to evaluate our approach. Firstly, we conducted 10-fold cross validation experiments to see how well various machine learning methods perform. We found that the classifier trained by Random Forest performs best here, with the accuracy 0.9788 and the AUC 0.9959. Secondly, we conducted experiments to illustrate that all the features are effective for malware detection. The results also show that in some case using type information is better than using the other two. Thirdly, to test our approach's ability to detect genuinely new malware or new malware versions, we ran a time split experiment: we used our classifier to detect the malware samples which are newer than the ones in our data set. Our classifier can detect 81% of the fresh samples, which indicates that our classifier is capable of detecting some fresh malware. The results also suggest that malware classifiers should be updated often with new data or new features in order to maintain the classification accuracy. Finally, one reason that makes malware detection difficult is that obfuscation techniques can be used to evade the detection. So for that, we performed experiments to test our approach's ability to detect new malware samples that are obtained by obfuscating the existing ones through some obfuscation tools. All the obfus-

cated malware samples can be detected by our classifier, demonstrating that our classifier has a resistance to some obfuscation techniques.

The remainder of this paper is organised as follows. Section 2 presents the related work. Section 3 describes our approach, and followed by the experimental results in Section 4. Finally, Section 5 concludes.

2 Related Work

There have been lots of work on malware detection. Here we focus on some recent related work based on machine learning. Interested readers can refer to the surveys [3, 16] for more details.

Over the past decade, intelligent malware detection systems have been developed by applying machine learning techniques. Some approaches employ the *N-grams* method to train a classifier based on the binary code content[4–6]. Rather surprisingly, a 1-gram model can distinguish malware from benign software quite well for some data set. But these approaches can be evaded easily by control-flow obfuscation. Some approaches [6–8] take the opcodes, which reflect the behaviours of the software of interest quite well, as features to classify malware and benign software. Some other approaches consider API calls, intents, permissions and commands as features [9–11], based on different classifiers. All these works focus more on the behaviour information, while our current work considers not only behaviour information but also data information.

There have been some research work that takes data information into account. Ye et al. [12] propose a malware detection approach based on interpretable strings using SVM ensemble with bagging. Islam et al. [13] consider malware classification based on integrated static and dynamic features, which includes printable strings. Both Karampatziakis et al. [14] and Tamersoy et al. [15] propose the malware detection approach based on the file-to-file relation graphs. More precisely, a file's legitimacy can be inferred by analyzing its relations (co-occurrences) with other labeled (either benign or malicious) peers. Clearly, both the string information and the file relation are quite simple and not all malware share the same information.

Some recent research work employ deep learning to help detect malware. Saxe and Berlin [17] propose a deep neural network malware classifier that achieves a usable detection rate at an extremely low false positive rate and scales to real world training example volumes on commodity hardware. Hardy et al. [18] develop a deep learning framework for intelligent malware detection based on API calls. Ye et al. [19] propose a heterogeneous deep learning framework composed of an AutoEncoder stacked up with multilayer restricted Boltzmann machines and a layer of associative memory to detect new unknown malware. In addition, concept drift is also borrowed into malware detection. Roberto et al. [20] propose a fully tuneable classification system *Transcend* that can be tailored to be resilient against concept drift to varying degrees depending on user specifications. Our approach employ several supervised machine-learning methods.

3 Approach

In this section, we first give a definition of our malware detection problem, then present our approach.

The malware detection problem can be stated as follows: given a dataset $D = \{(e_1, c_1), \ldots, (e_m, c_m)\}$, where $e_i \in E$ is an executable file, $c_i \in C = \{benign, malicious\}$ is the corresponding actual category of e_i which may be unknown, and m is the number of executable files, the goal is to find a function $f : E \to C$ such that $\forall i \in \{1, \ldots, m\}$. $f(e_i) \approx c_i$.

Indeed, the goal is to find a classifier which classifies each executable file as precise as possible. Our approach tries to learn a classifier from existing executables with known categories first, and then uses this classifier to predict categories for new, unseen executables. Figure 1 shows the framework of our approach, which consists of two components, namely the feature extractor and the malware classifier. The feature extractor extracts the feature information (*i.e.*, opcodes, data types and system libraries) from the executables and represents them as vectors. While the malware classifier is first trained from an available dataset of executables with known categories by a supervised machine-learning method, and then can be used to detect new, unseen executables. In the following, we describe both components in more detail.

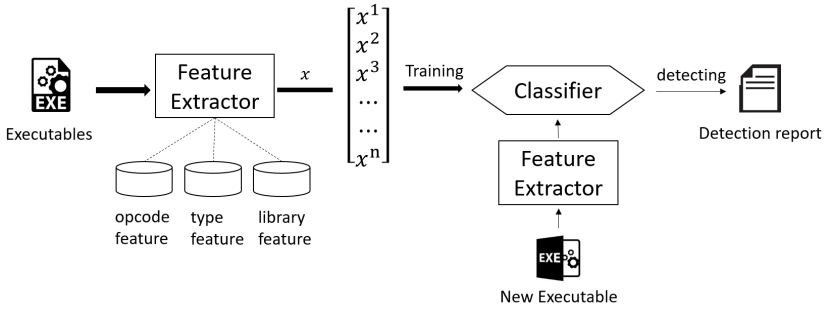

Fig. 1. the framework of our approach

3.1 Feature Extractor

This section presents the processing of the feature extractor, which consists of the following steps: (1) decompilation, (2) information extraction, and (3) feature selection and representation.

Decompilation. In this paper, we focus on the executables on Windows, namely, *exe* files and *dll* files. An instruction or a datum in an executable file can be represented as a series of binary codes, which are clearly not easy to read. So the first step is to transform the binary codes into a readable intermediate representation such as assembly codes by a decompilation tool.

Information Extraction. Next, the extractor parses the *asm* files decompiled from executables to extract the information, namely, opcodes, data types and system libraries. Generally, the opcodes used in an executable represent its intended behaviours, while the data types indicate the structures of the data it may perform on. In addition, the imported system libraries, which reflect the interaction between the executable and the system, are also considered. All such information describes the possible mission of an executable in some sense, and similar executables share the similar information.

The opcode information can be extracted either statically or dynamically. Here we just collect and count the opcodes from an asm file instruction by instruction, yielding an opcode list l_{opcode}, which records the opcodes appearing in the *asm* file and their corresponding times. For data type information, we first discover the possible variables, including local and global ones, and recover their types using BITY [21]. Then we do a statistical analysis on the recovered types, wherein we just consider their data sizes for the composed types for simplicity, yielding a type list l_{type}. The extraction of library information is similar to the one of opcodes, yielding the library list $l_{library}$.

Feature Selection and Representation. There may be too many different terms in the collected lists from all the executables, and not all of them are of interest. For example, *push* and *mov* are commonly used in both malware and benign softwares. For that, the extractor creates a dictionary to keep record of the interesting terms. Only the terms in this dictionary are reserved in the lists. Take the opcode lists for example. The extractor performs a statistical analysis on all the opcode lists from an available dataset, using TF-IDF to measure the statistical dependence. Next the extractor selects the top k weight opcodes to form an opcode dictionary, and then filters out the terms not in this dictionary for each l_{opcode}. The same applies to l_{type} and $l_{library}$.

After that, the extractor builds a *profile* for each executable file by concatenating these three lists collected from its corresponding *asm* file, namely, the executable is represented as a profile $[l_{opcode}, l_{type}, l_{library}]$. Assume that there is a fixed order for all the features. Then a profile can be simplified as a vector $(x^1, x^2, ..., x^n)$, where x^i is the value of the i-th feature and n is the total number of the feature. An example vector is shown in Figure 2.

3.2 Malware Classifier

In this section, we present the malware classifier. As mentioned before, we first train our malware classifier from an available dataset of executables with known categories by a supervised machine-learning method, and then use it detect new, unseen executables.

Classifier Training. Now by our feature extractor, an executable e can be represented as a vector x. Let X denote the feature space for all possible vectors, D_0 represent the available dataset with known categories, and 0 and 1 represent *benign* and *malicious* respectively. Our training problem is to find a classifier $\mathsf{C} : X \to [0, 1]$ such that min $\Sigma_{(x,c)\in D_0} d(\mathsf{C}(x) - c)$, where d denotes the distance function. Clearly, there are many classifier algorithms to solve this problem, such

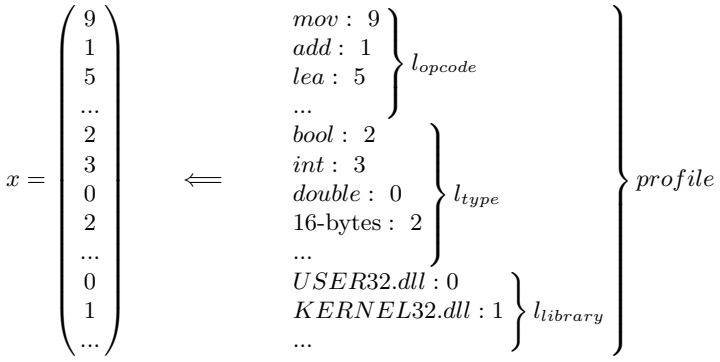

Fig. 2. Example for Vector Representation

as K-Nearest Neighbor, Native Bayes, Decision Tree, Random Forest, Support Vector Machine, and SGD Classifier. In our implementation we have made all these methods available, so users can select any one they like. Moreover, we have also conducted some experiments with these algorithms and found out that the classifier trained by Random Forest performs best here (more details will be given in Section 4).

Malware Detection. Once it is trained, the classifier C can be used to detect malware: returns the category as close as the classifier returns. Formally, given an executable e and its feature vector x, the goal of the detection is to find $c \in C$ such that min $d(\mathsf{C}(x) - c)$.

4 Experiments

In this section, we conduct a series of experiments to evaluate our approach. Firstly, we conduct a set of cross-validation experiments to evaluate how well each classifier method performs based on the behaviour and data features. Meanwhile, we also measure the runtime for each classifier method. The runtime of the feature extractor is measured as well. Secondly, based on each kind of feature we also conduct experiments to see their effectiveness. Thirdly, to test our approach's ability to detect genuinely new malware or new malware versions, we also run a time split experiment. Finally, we conduct some experiments to test our approach's ability to detect new malware samples that are obtained by obfuscating the existing ones. In addition, the first two experiments also give a comparison of our method and most of existing methods, since most of existing methods adopt various classifier methods, which are considered here, with only behaviour features.

Our dataset consists of a malware dataset and a benign software dataset. The malware dataset consists of the samples of the BIG 2015 Challenge [22] and the samples before 2017 in theZoo aka Malware DB [23], with 11376 samples in total; while the benign software dataset are collected from QIHU 360 software

company, which is the biggest Internet security company in China, with 8003 samples in total.

To quantitatively validate the experimental results, we use the performance measures shown in Table 1. All our experiments are conducted in the environment: 64 Bit Windows 10 on an Intel (R) Core i5-4590 Processor (3.30GHz) with 8GB of RAM.

Table 1. Performance Measures in Malware Detection

Measure	Description
True Positive (TP)	Number of files correctly classified as malicious
True Negative (TN)	Number of files correctly classified as benign
False Positive (FP)	Number of files mistakenly classified as malicious
False Negative (FN)	Number of files mistakenly classified as benign
Precision (PPV)	TP / (TP+FP)
TP Rate (TPR)	TP / (TP+FN)
FP Rate (FPR)	FP / (TP+FN)
Accuracy (ACY)	(TP+TN) / (TP+TN+FP+FN)

Cross Validation Experiments. To evaluate the performance of our approach, we conduct 10-*fold cross validation* experiments: in each experiment, we randomly split our dataset into 10 equal folds; then for each fold, we train the classifier model based on the data from the other folds (*i.e.*, the training set), and test the model on this fold (*i.e.*, the testing set). The learning methods we use in our experiments are listed as follows:

- K-Nearest Neighbour (KNN): experiments are performed under the range $k = 1$, $k = 3$, $k = 5$, and $k = 7$.
- Native Bayes (NB): several structural learning algorithms, that is, *Gaussian Naive Bayes* (GNB), *Multinomial Naive Bayes* (MNB), and *Bernoulli Naive Bayes* (BNB) are used in our experiments.
- Decision Trees (DT): we use decision tree with two criteria, namely, "gini" for the *Gini impurity* and "entropy" for the *information gain*. *Random Forest* (RF) with 10 trees are used as well.
- Support Vector Machines (SVM): *linear kernel*, *sigmoid kernel*, and *Radial Basis Function based kernel* (RBF) are used in our experiments, as well as the linear models with *stochastic gradient descent* (SGD).

Table 2 shows the detailed results of the experiments and Figure 3 shows some selected *Receiver Operating Characteristic* (ROC) curve. All the KNN classifiers and DT classifiers perform quite well, with an accuracy greater than 97%, among which Random Forest with 10 entropy trees have produced the best result. Rather surprisingly, 1-KNN performs better than the other KNNs. While all the NB classifiers perform a bit worse, with the accuracy from 66.38% to 82.79%. For SVN classifiers, most of them get an accuracy greater than 95%, except for the one with sigmoid kernel whose accuracy is 85.80%. Concerning

ROC curve, most classifiers can produce much better classification results than random classification results, except for Gaussian Naive Bayes with the Area Under the ROC (AUC) 0.5931, which is just a little bit bigger than 0.5. Random Forest with 10 entropy trees perform the best as well (with the AUC 0.9959). One of the main reasons for Random Forest to outperform others is that it is an ensemble learning method which may correct the Decision Trees' habit of overfitting to the training set. In the future, some other ensemble learning methods will be also considered.

Table 2. Results of Different Methods

Classifier	**PPV**	**TPR**	**FPR**	**ACY**	**Classifier**	**PPV**	**TPR**	**FPR**	**ACY**
KNN K=1	0.9609	0.9830	0.0281	0.9764	KNN K=3	0.9572	0.9798	0.0307	0.9736
KNN K=5	0.9556	0.9763	0.0319	0.9715	KNN K=7	0.9556	0.9763	0.0319	0.9715
DT gini	0.9658	0.9797	0.0243	0.9773	DT entropy	0.9685	0.9787	0.0223	0.9781
RF gini	0.9678	0.9787	0.0228	0.9778	RF entropy	0.9692	0.9800	0.0218	0.9788
GNB	0.9381	0.1990	0.0092	0.6638	MNB	0.9494	0.6590	0.0247	0.8446
BNB	0.8726	0.6831	0.0701	0.8279	SVM linear	0.9581	0.9896	0.0304	0.9778
SVM rbf	0.9686	0.9119	0.0207	0.9514	SVM sigmoid	0.9671	0.7308	0.0232	0.8580
SGD	0.9582	0.9862	0.0302	0.9765					

Runtime Performance. Meanwhile, we count the training times (the runtime costed by building the classifier based on the training set) and the testing time (the time needed by evaluating the testing set) in seconds for each cross validation experiment. The results are shown in Table 3. In term of the training time, SVM has been the worst for our data set (with time ranging from 150.022s to 1303.607s), while Naive Bayes has been the best (with time ranging from 1.535s to 3.093s). Random Forest has also performed well on training, with time between 3.791s and 4.115s. Concerning the testing time, KNN has been the worst (due to its lazy learning), while Random Forest, Naive Bayes and SGD classifier have performed quite well, with time small than 1s. Considering the accuracy, ROC and the runtime, we suggest users to use the Random Forest classifier.

Table 3. Runtime for Various Classifiers

Classifier	**Training(s)**	**Testing(s)**	**Classifier**	**Training(s)**	**Testing(s)**
KNN K=1	16.477	178.789	KNN K=3	16.369	199.474
KNN K=5	16.517	207.052	KNN K=7	16.238	210.557
DT gini	23.442	0.067	DT entropy	13.485	0.066
RF gini	4.115	0.086	RF entropy	3.791	0.077
GNB	3.093	0.480	MNB	1.535	0.035
BNB	1.826	0.828	SVM linear	150.022	14.494
SVM rbf	799.310	50.196	SVM sigmoid	1303.607	130.178
SGD	22.569	0.048			

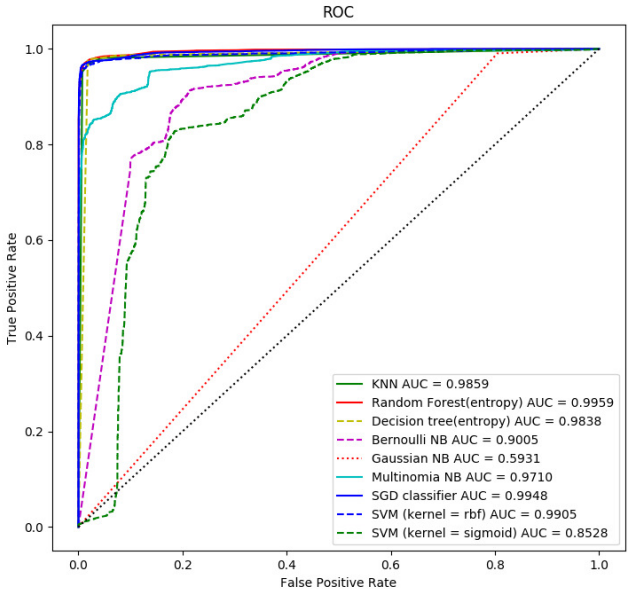

Fig. 3. ROC Curve of Different Methods

We have also evaluated how the feature extractor performs. For that we first measure the total time needed by the decompilation tool IDA Pro to decompile the executables. Our data set contains 19379 executables, with the total size of 250GB. It takes 15.6 hours to decompile all the files, with an average 0.22s/MB. IDA Pro is a heavy-weight tool, so using a lightweight one might have improved the time. Secondly, we measure the total time needed by extracting features from the decompiled asm files. The sizes of asm files range from 142KB to 144.84MB, with the total size of 182GB. It takes 10.5 hours to extract the features for all the asm files, with an average 0.20s/MB. Both the decompilation time and the extracting time are considered to be acceptable.

Feature Experiments. We have also conducted experiments based on each kind of feature to see their effectiveness. For that, we conduct the same experiments as above for each kind of feature. Table 4 gives the experimental results and Figure 4 shows the corresponding ROC curves, where only the results of one classifier are selected to present here for each kind of learning method.

From the results we can see that all the features are effective to help detect malware, and using all of the features together has produced the best results, with the average accuracy 0.9431 and the average AUC 0.9869. The opcode feature has performed better, with the accuracy 0.9339 and the AUC 0.9790 on average, than the other two. The reason for this is that opcodes help describe the intended behaviours of the software quite well. This also explains why most

Table 4. Results of Different Features

Feature	Classifier	PPV	TPR	FPR	ACY	AUC
Opcode	Naive Bayes	0.9492	0.6128	0.0230	0.8266	0.9455
	KNN	0.9521	0.9777	0.0345	0.9705	0.9842
	Random Forest	0.9595	0.9775	0.0290	0.9736	0.9947
	SGD Classifier	0.9462	0.9701	0.0387	0.9649	0.9914
	Average	0.9518	0.8840	0.0313	0.9339	0.9790
Library	Naive Bayes	0.8508	0.7705	0.0950	0.8494	0.9311
	KNN	0.7439	0.9892	0.2395	0.8549	0.8878
	Random Forest	0.9439	0.8328	0.0348	0.9105	0.9736
	SGD Classifier	0.9401	0.8014	0.0358	0.8969	0.9661
	Average	0.8711	0.8485	0.1004	0.8785	0.9397
Type	Naive Bayes	0.8346	0.1829	0.0254	0.6476	0.9020
	KNN	0.8570	0.9735	0.1142	0.9219	0.9493
	Random Forest	0.8751	0.9790	0.0982	0.9336	0.9741
	SGD Classifier	0.8208	0.9427	0.1447	0.8913	0.9452
	Average	0.8483	0.7701	0.0945	0.8496	0.9427
All	Naive Bayes	0.9494	0.6590	0.0247	0.8446	0.9710
	KNN	0.9572	0.9798	0.0307	0.9736	0.9859
	Random Forest	0.9678	0.9787	0.0228	0.9778	0.9959
	SGD Classifier	0.9581	0.9858	0.0303	0.9763	0.9948
	Average	0.9582	0.9008	0.0272	0.9431	0.9869

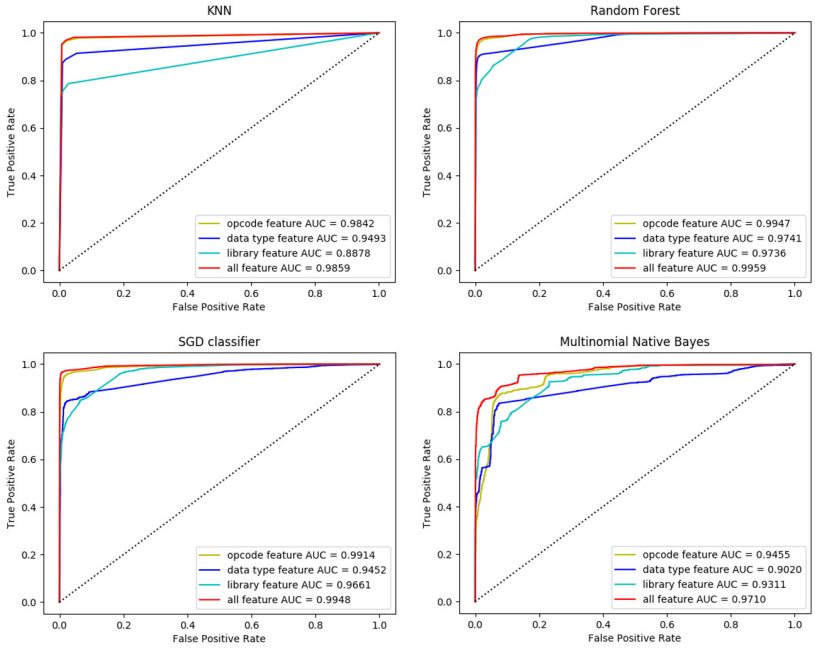

Fig. 4. ROC Curve of Different Features

existing work on malware detection employs the opcode feature. Using the opcode feature alone seems to be enough for malware detection in most cases, but the detection may evade easily by obfuscation techniques like adding unreachable codes. In some cases it is better to use the other two features and adding them will improve the detection. So it is useful to also take the type and library features into account. On average, the type feature has performed better than the library feature in term of AUC, although the library feature has produced slightly better results than the type feature with respect to accuracy. The results also show that Random Forest has performed the best for each feature, which is consistent with the above experiments. Note that, when Random Forest is employed, using type feature can get the best TPR. The opcode and library features have been used by lots of work in practice, so we believe that the type feature, which is not considered by most of existing work, can benefit malware detection as well in practice.

Time-Split Experiments. As Saxe and Berlin [17] point out, a shortcoming of the standard cross validation experiments is that they do not separate the ability to detect slightly modified malware from the ability to detect new malware samples and new versions of existing malware samples. In this section, to test our approach's ability to detect genuinely new malware or new malware versions, we ran a time split experiment. We first download the malware samples, which were collected dated from January 2017 to July 2017, from the DAS MALWERK website [24]. That is, all the malware samples are newer than the ones in our data set. We then train a classifier based on our data set using Random Forest. Finally, we use our classifier to detect these fresh malware samples.

Table 5. Results of Fresh Samples

Date	Num	RF Classifier	Accuracy
July, 2017	12	6	0.500
June, 2017	61	29	0.475
May, 2017	87	76	0.874
April, 2017	31	28	0.903
March, 2017	48	42	0.875
February, 2017	69	61	0.884
January, 2017	56	53	0.946
Total	364	295	0.810

The experimental results are shown in Table 5. About 81% of the samples can be detected by our classifier, which signifies that our approach can detect some malware samples and new versions of existing malware samples. However, the results also indicate that the classifier becomes ineffective as time passes. For example, the results detected by our approach for the samples collected in June and July are worse than random classification (the accuracies are not greater than 0.5). This is due to the high degree of freedom for malware writer to produce new malware samples continuously. Consequently, most malware classifiers

become unsustainable in the long run, becoming rapidly antiquated as malware continues to evolve. This suggests that malware classifiers should be updated often with new data or new features in order to maintain the classification accuracy.

Obfuscation Experiments. One thing that makes malware detection even more difficult is that malware writers may use obfuscation techniques, such as inserting NOP instructions, changing what registers to use, changing flow control with jumps, changing machine instructions to equivalent ones or reordering independent instructions, to evade the detection. In this section, we report some experiments to test our approach's ability to detect new malware samples that are obtained by obfuscating the existing ones.

The obfuscated malware samples are obtained as follows. Firstly, we use the free version of the commercial tool *Obfuscator* [25], which only supports changing code execution flow, to obfuscate 50 malware samples, which are randomly selected from our data set, yielding 50 new malware samples. Secondly, we use the open source tool *Unest* to obfuscate 15 obj files, which are compiled from malware samples in C source codes through VS 2010[1], by the following four techniques, that is, (a) rewriting digital changes equivalently, (b) confusing the output string, (c) pushing the target code segment into the stack and jumping to it to confuse the target code, and (d) obfuscating the static libraries, yielding 60 new malware samples.

Table 6. Results of Obfuscated Malware Samples

Tools	Number	RF Classifier	Accuracy
Obfuscator	50	50	100%
Unest	60	60	100%

We used our classifier trained by Random Forest to detect the newly generated malware samples. The results are presented in Table 6. The results show that all the obfuscated malware samples have been detected by our classifier. This indicates that our classifier has some resistance to some obfuscation techniques. To change code execution flow, Obfuscator inserts lots of *jump* instructions. It seems that the *jump* instructions may change the opcode feature, while the data feature and the library feature are still the same. Indeed, *jump* is commonly used in both malware and benign software. Therefore, this technique does not change the features we use and thus cannot evade our detection. Similar to Obfuscator, the techniques used in Unset make little changes on the features, thus cannot evade our detection either. Nevertheless, the techniques used here is rather simple. The integration of more (sophisticated) techniques will be considered in the future.

[1] We are able to obfuscate only the obj files compiled from C codes through VS 2010.

5 Conclusion

In this work, we have proposed a malware detection approach using various machine learning methods based on the opcodes, data types and system libraries. To evaluate the proposed approach, we have carried out some interesting experiments. Through experiments, we have found that the classifier trained by Random Forest outperforms others for our data set and all the features we have adopted are effective for malware detection. The experimental results have also demonstrated that our classifier is capable of detecting some fresh malware, and has a resistance to some obfuscation techniques.

As for future work, we may consider some other meaningful features, including static features and dynamic features, to improve the approach. We can employ the unsupervised machine learning methods or the deep learning methods to train the classifiers. More experiments on malware anti-detecting techniques are under consideration.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments. This work was partially supported by the National Natural Science Foundation of China under Grants No. 61502308, 61373033 and 61672358, Science and Technology Foundation of Shenzhen City under Grant No. JCYJ20170302153712968.

References

1. *McAfee Labs Threats Report.* June 2017.
2. Philippe Beaucamps and Eric Filiol. On the possibility of practically obfuscating programs towards a unified perspective of code protection. *Journal in Computer Virology*, 3(1):3–21, 2007.
3. Yanfang Ye, Tao Li, Donald Adjeroh, and S. Sitharama Iyengar. A survey on malware detection using data mining techniques. *Acm Computing Surveys*, 50(3):41, 2017.
4. Yuval Elovici, Asaf Shabtai, and Moskovitch. Applying machine learning techniques for detection of malicious code in network traffic. In *German Conference on Advances in Artificial Intelligence*, pages 44–50, 2007.
5. Mohammad M Masud, Latifur Khan, and Bhavani Thuraisingham. A scalable multi-level feature extraction technique to detect malicious executables. *Information Systems Frontiers*, 10(1):33–45, 2008.
6. Blake Anderson, Curtis Storlie, and Terran Lane. Improving malware classification:bridging the static/dynamic gap. In *ACM Workshop on Security and Artificial Intelligence*, pages 3–14, 2012.
7. Yanfang Ye, Tao Li, Yong Chen, and Qingshan Jiang. Automatic malware categorization using cluster ensemble. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2010.
8. Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231(9):64–82, 2013.

9. Tzu Yen Wang, Shi Jinn Horng, Ming Yang Su, and Chin Hsiung Wu. A surveillance spyware detection system based on data mining methods. In *IEEE International Conference on Evolutionary Computation*, pages 3236–3241, 2006.
10. Yanfang Ye, Dingding Wang, Tao Li, and Dongyi Ye. Imds: intelligent malware detection system. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1043–1047, 2007.
11. Yanfang Ye, Tao Li, Kai Huang, Qingshan Jiang, and Yong Chen. Hierarchical associative classifier (hac) for malware detection from the large and imbalanced gray list. *Journal of Intelligent Information Systems*, 35(1):1–20, 2009.
12. Yanfang Ye, Lifei Chen, Dingding Wang, Tao Li, Qingshan Jiang, and Min Zhao. Sbmds: an interpretable string based malware detection system using svm ensemble with bagging. *Journal in Computer Virology*, 5(4):283, 2009.
13. Rafiqul Islam, Ronghua Tian, Steve Versteeg, and Steve Versteeg. Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*, 36(2):646–656, 2013.
14. Nikos Karampatziakis, Jack W. Stokes, Anil Thomas, and Mady Marinescu. *Using File Relationships in Malware Classification*. Springer Berlin Heidelberg, 2013.
15. Acar Tamersoy, Kevin Roundy, and Duen Horng Chau. Guilt by association: large scale malware detection by mining file-relation graphs. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2014.
16. Gamal Abdel Nassir Mohamed and Norafida Bte Ithnin. Survey on representation techniques for malware detection system. *American Journal of Applied Sciences*, 2017.
17. Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 11–20, 2015.
18. William Hardy, Lingwei Chen, Shifu Hou, Yanfang Ye, and Xin Li. Dl4md: A deep learning framework for intelligent malware detection. In *Proceedings of the International Conference on Data Mining*, 2016.
19. Yanfang Ye, Lingwei Chen, Shifu Hou, William Hardy, and Xin Li. Deepam: a heterogeneous deep learning framework for intelligent malware detection. *Knowledge and Information Systems*, pages 1–21, 2017.
20. Roberto Jordaney, Kumar Sharad, Santanu K. Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 625–642, 2017.
21. Zhiwu Xu, Cheng Wen, and Shengchao Qin. Learning Types for Binaries. In *19th International Conference on Formal Engineering Methods*, 2017.
22. *Microsoft Malware Classification Challenge.* `https://www.kaggle.com/c/malware-classification`.
23. *theZoo aka Malware DB.* `http://ytisf.github.io/theZoo/`.
24. *DAS MALWERK.* `http://dasmalwerk.eu/`.
25. *Obfuscator.* `https://www.pelock.com/products/obfuscator`.