Metamorphic Malware Detection by PE Analysis with the Longest Common Sequence

Thanh Nguyen Vu[1(✉)], Toan Tan Nguyen[1], Hieu Phan Trung[1], Thao Do Duy[1], Ke Hoang Van[1], and Tuan Dinh Le[2]

[1] University of Information Technology, Vietnam National University, HCM City, Ho Chi Minh City, Vietnam
{nguyenvt, toannt, hieupt}@uit.edu.vn,
{13520797, 13520376}@gm.uit.edu.vn
[2] Long An University of Economics and Industry, Tan An, Long An Province, Vietnam
le.tuan@daihoclongan.edu.vn

Abstract. Metamorphic malware detection is one of the most challenging tasks of antivirus software because of the difference in signatures of new variants from preceding one [1]. This paper proposes the method for the metamorphic malware detection by Portable Executable (PE) Analysis with the Longest Common Sequence (LCS). The proposed method contains the following phase: The raw feature extraction obtains valuable features like the information of Windows PE files which are PE header information, dependencies imports and API call functions, the code segments inside each of Windows PE file. Next, these segments are used for generating the detectors, which are later used to determine affinities with code segments of executable files by the longest common sequence algorithm. Finally, header, imports, API call information and affinities are combine into vectors as input for classifiers are used for classification after a dimensionality reduction. The experimental results showed that the proposed method can achieve up to 87.1% precision, 63.3% recall for benign and 92.6% precision, 93.7% for average malware.

Keywords: Malware detection · Data mining · Longest common sequence · Neural network

1 Introduction

The battle against computer threats is an endless game. In recent years, the considerable developments in mobile, wearable and IoT devices attract a lot of cybercriminals. However, the prevalent PC usages for important tasks in enterprise, finance and banking always keep them the main target for malware creators with the largest number of Windows malware samples. And .EXE (extension for PE files) is still the most dangerous extension [2]. Therefore, this paper focuses on problem how to detect the metamorphic malware on PE files.

Computer security experts have applied wide range of methods from traditional signature-based recognition, non-signature based malware detection techniques like

T.K. Dang et al. (Eds.): FDSE 2017, LNCS 10646, pp. 262–272, 2017.
https://doi.org/10.1007/978-3-319-70004-5_18

heuristic analysis, behavior analysis or a combination of both for malware detection. The prior work showed that non-signature based malware detection techniques are not affective because of high false positive rate and high processing costs [3]. Most of common Anti-Virus products utilize signature based detection schemes that provide low false positive rate and have acceptable cost. However, their limitation is that they are unable to detect zero-day malware while the numbers of malware sample increases sharply years by years [2].

In the non-signature based schemes, the two main approaches for malware analysis are static and dynamic analysis approaches. In the static analysis approach, code and structure of a program is examined without running the program whereas in the dynamic analysis approach, the program can run in sandbox (a simulating environment).

This paper proposed a static malware detection system using data mining techniques. The raw extracting step acquires valuable features of PE files: PE header information, DLL imports, API call functions and code segments. Afterward, these segments are used for generating the detectors, which are later used to determine affinities with code segments of executable files. Finally, header, imports, API call information and affinities are combine into input for classifiers are used for classification after a dimensionality reduction.

2 Related Work

In 2001, Shultz et al. proposed the method using data mining (DM) techniques for detecting new malicious executable files [3]. Three different types of features were extracted from the executables, i.e. the list of DLLs used by the binary, the list of DLL function calls, and number of different system calls used within each DLL.

A similar approach was used by Kolter et al. [4], in which they used n-gram analysis and data mining approaches to detect malicious executable files in the wild. The authors used a hex dump utility to convert each executable file to hexadecimal code in an ASCII format and to produce n-gram features by combining four-byte sequence into a single term. Information gain was selected to get the valued features which are input to several classifiers.

In [5], Zhenlong Yuan et al. used a combination of static and dynamic analysis in feature extraction for Android OS. The two static phases were to retrieve the required permission and the sensitive API call while the dynamic phase provided dynamic behaviors. The extracted features were used for the deep learning model constructed by Deep Belief Networks.

Chao et al. [6] made used of artificial immune system to build a set of detectors which are malicious estimation units. Then executable files converted into affinity vectors before coming as input for different classifiers. Their result showed a positive performance in virus detection.

Unlike the previous works, in this paper, we propose the malware detection method which includes the extraction of DLL imports, DLL function calls from [3] and building set of detectors from [6] to robust malware detection system effectiveness.

3 The Proposed Malware Detection Method

Our malware detection system is based on the information extraction from the .EXE files which are formatted by PE format structure. The information is then selected and extracted to use for classification. The proposed method has two phases: the training and the testing as shown in Fig. 1.

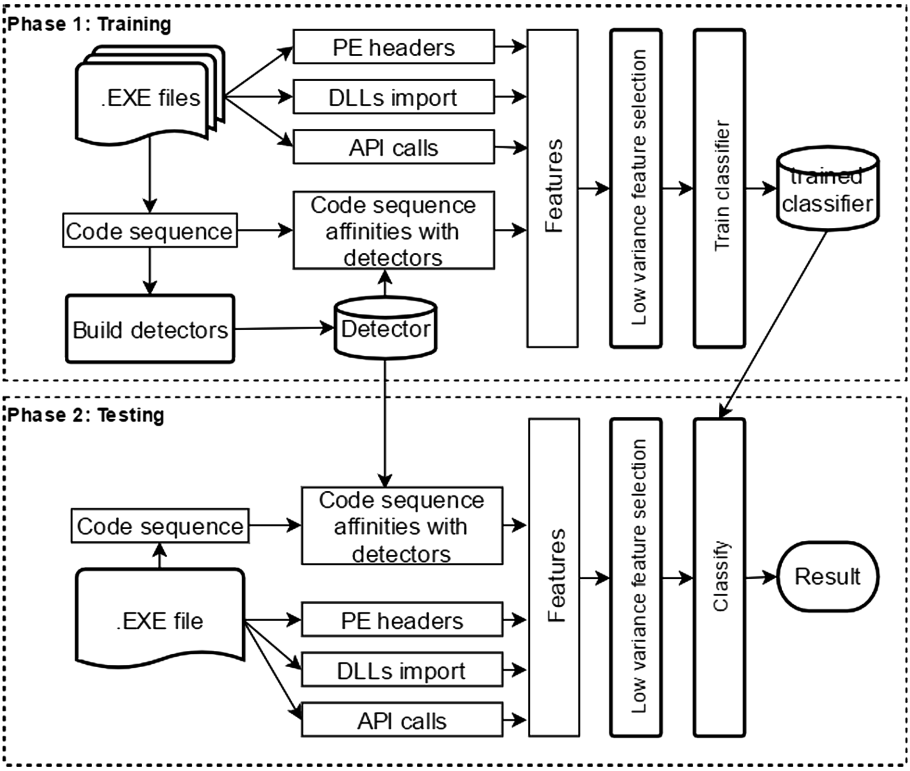

Fig. 1. Overall malware detection model

In the training phase, firstly, the raw feature extraction obtains the useful features such as the information of Windows PE files which are PE header information, dependencies imports and API call functions, the code segments inside each of Windows PE file. Next, these segments are used for generating the detectors, which are later used to determine affinities with code segments of executable files by the longest common sequence algorithm. Finally, header, imports, API call information and affinities are combine into vectors as input for classifiers are used for classification after dimensional reduction. In the testing phase, set of detector and classifier model obtained from the training phase are used for detecting and classifying which classes executable files is belong to.

3.1 The Portable Executable Format Structure

The Portable Executable (PE) format is the file format for executable files, object code, DLLs and others used in 32-bit and 64-bit versions of Windows OS. The PE format is the data structure which encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code. On NT OSs, the PE format is used for EXE, DLL and other file types. The PE file header consists of the MS DOS stub, the PE the file signature, the COFF (Common Object File Format) header, and the optional header. It contains the important information of the file such as the number of sections, the size of the stack and the heap, etc. The section table has the section information such as their name, offset and size, etc. These sections contain the actual data such as code, initialized data, exports, imports and resources [7]. Feature extraction makes use of a tool to exploit executable files based on this format.

3.2 Feature Extraction

As the overall model shown in the previous section, the .EXE files are extracted into four information groups: PE header, DLLs import, API calls, and Code Sequences (in Assembly Code) by using the official utility named *dumpbin* from Microsoft Corp [8].

PE header. The PE file header consists of a MS DOS stub, a PE file signature, a COFF header, and an optional header. The COFF header contains different information such as machine type, number of sections, time stamp, file pointer to symbol table, number of symbols, characteristics, etc. The optional header contains linker version, size of code, size of initialized/uninitialized data, OS version, image version, subsystem version, etc. They are used as numeric features. In our dataset, 62 figures were selected after removing those figures which cause missing values.

DLLs import. Most of executable files on Windows OS must import dynamic link library (DLL) for executing command in lower layer. Each DLL has a group of certain functions related to each other. Therefore, obtaining the import of an executable provides an overview of its functionalities. For instance, the WS2_32.DLL contains the Windows Sockets API used by most Internet and network applications to handle network connections so an executable import this DLL could run some network activities. Similarly, the spyware which steals sensitive information and sends to its author server must import network-related DLLs directly or indirectly to complete its mission. In [3], Schultz et al. have used the conjunction of DLL names, with a similar functionality, as binary features. Their experimental results showed that this feature helps to attain reasonable detection accuracy. However, Shafiq's experimental studies have showed that using them as individual binary features can describe more information, and hence can be more helpful in detecting malicious PE files [9]. In our approach, we have 96 DLLs after removing the low frequency DLLs as binary features. 15 of them are listed in Table 1 with the frequency in our dataset.

API calls. Each of DLL provides several APIs for using the executables. For example, Kernel32.dll exports *WriteFile* function which is used for writing data to the specified file or input/output device; User32.dll offers *IsWindow* function is used to determine whether the specified window handle identifies an existing window. Like DLLs import,

Table 1. Top 15 DLLs by frequency

Name	Frequency	Description
KERNEL32.DLL	8940	WINDOWS NT BASE API CLIENT DLL
USER32.DLL	7317	MULTI-USER WINDOWS USER API CLIENT
ADVAPI32.DLL	3926	ADVANCED WINDOWS 32 BASE API
GDI32.DLL	1939	WIN32 GDI CORE COMPONENT
WINMM.DLL	1700	MCI API DLL
MSVCRT.DLL	1590	MICROSOFT (R) C RUNTIME LIBRARY
OLE32.DLL	1545	MICROSOFT OLE FOR WINDOWS
OLEAUT32.DLL	1077	MICROSOFT OLE AUTOMATION
MSVFW32.DLL	977	MICROSOFT VIDEO FOR WINDOWS DLL
WS2_32.DLL	956	WINDOWS SOCKET 2.0 32-BIT DLL
RPCRT4.DLL	846	REMOTE PROCEDURE CALL RUNTIME
SHLWAPI.DLL	772	SHELL LIGHT-WEIGHT UTILITY LIBRARY
MSACM32.DLL	675	MICROSOFT ACM AUDIO FILTER
HAL.DLL	672	HARDWARE ABSTRACTION LAYER DLL
SHELL32.DLL	648	WINDOWS SHELL COMMON DLL

the calls to these functions also infer their intention of the executable call to. In our pre-analysis, there were more than 3000 APIs called in the full dataset. The APIs were later reduced by removing the low frequency APIs before being investigated on the official API descriptions of Microsoft Developer Network (MSDN) website [10] to decide whether to keep or remove from the set. The criterion for removal are common and non-malicious related activities are:

Allocate memory: this is almost essential for every program.
Pure multimedia creating: Set a menu, draw a line, an eclipse, rectangle and fill color, change font color, play a sound, convert color palette, show icon, images, etc.
Error and exception throw: these activities almost report bug and crash for Windows log and Windows trouble shooter.

The criteria for retaining are:

File-related activities: Create a specific file in sensitive folders; Delete, break, override system files or application files; Edit, append, encrypt files; Traverse file directory and find target files, etc.
Process-related activities: load insert dll into other process and create a new thread to duplicate itself; Create new processes to execute its code; Create threads to search and terminate other process, end task etc.
Memory-related activities: Forbid memory allocation and reclaim memory space; Point the interrupt vector address to the initial address of the its code, etc.
Registry manipulation activities: Create/Read/Edit/Delete to the specified registry key, search registry key, Open/Close a handle to specified registry key, etc.

Network-related activities: Open or listen on specific port, obtain mail client profile, creates an HTTP request handle, create/send mail slot, retrieves header information associated with an HTTP request, open proxy handle, etc.
Collecting data activities: get data from SQL server, get geography, file attribute, retrieve handle to the display monitor, read the console, get OS version, get machine OEM information, get dialog info, etc.
String manipulation activities: find, replace, convert, encode, parse string could be relating to code/file/resource/config injection, alteration.
Tracking I/O and others: track the mouse event, monitors keystrokes, privilege access, reboot system, minimize a window, broadcast/send message, create mutex/atom variable, get/set environment string, get/set timer, etc.

All function from third party DLLs (non-Microsoft integrated) with high-mid frequency are also kept. Finally, we got 2029 API calls as binary features.

Find Detectors. The PE format structure showed that there may contain code sections or other resources. With the support of the tool *dumpbin*, we extracted the code section as the code sequence in assembly (Fig. 2 A Code Sequence) so that the code sequence is represented for an executable file.

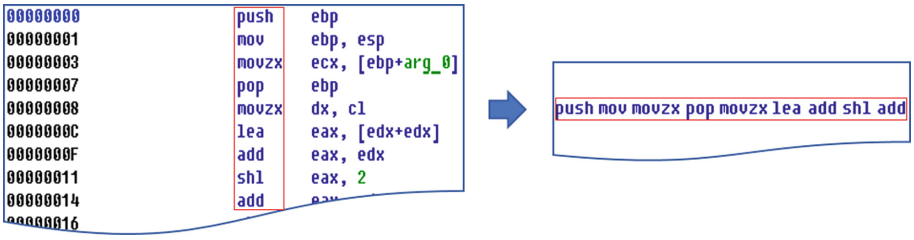

Fig. 2. A code sequence

Like the concept *detectors* in [6], our detectors are used for calculating the affinities with code sequences. The overall procedure to get detectors is shown in the Fig. 3 below.

As the Fig. 3, each malware family is extracted to the local detector set. Each set was then merged into one united set using the benign threshold. Any detector matched more than threshold number of benign files (benign code sequences) would be eliminated (matching standard describes in next section). The find-local-detectors step is shown in Fig. 4. This step applies on executables of each malware family to find the local detector for that family. In this step, the two consecutive executables of a family are used to find the longest common subsequence (LCS) by using the LCS algorithm. After obtaining all LCSs for that family, a similar procedure is applied on the new LCSs to create next generation LCSs. The procedure re-occurs while the number of a new generation LCSs is still higher than the pre-define threshold. When the procedure stops, the latest generation LCSs becomes the local detectors for that family.

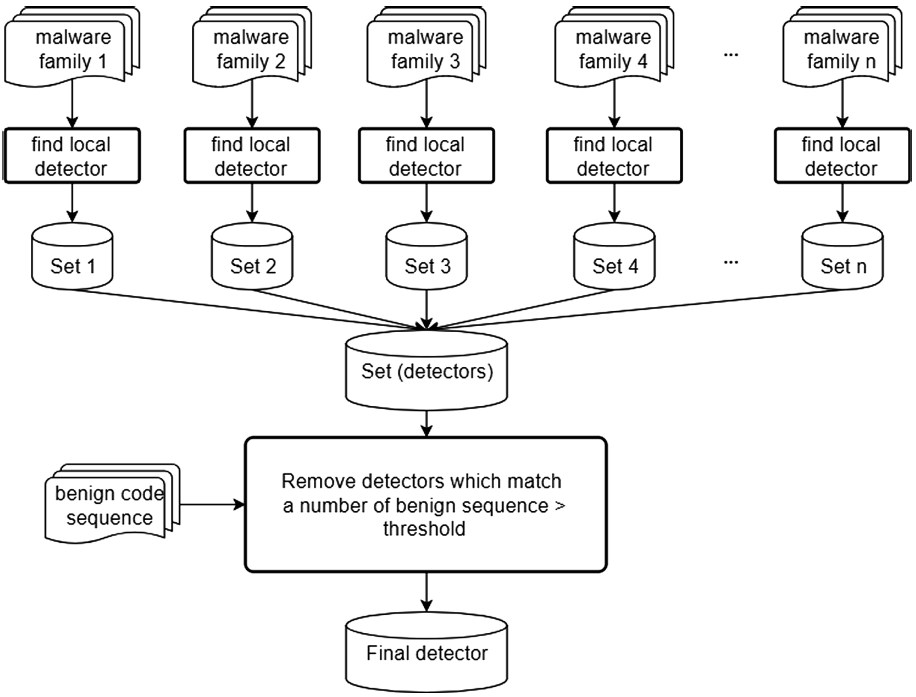

Fig. 3. Build final detector process

Code Sequence Affinities with Detectors and Matching Standard.
Code sequence affinity is the similarity metrics with a detector of that code sequence. There are several sequence metrics such as Levenshtein distance, Jaro-Winkler distance for different purposes. In our case, we decided to use a custom metrics:

$$aff(sequence, detector) = \frac{length(LCS(sequence, detector))}{length(detector)} \quad (1)$$

Matching Standard is a criterion to remove a detector from detectors set when that detector is considered as too common in benign set. A detector and a benign sequence are matched when their affinity is larger than a preset matched threshold.

$$aff(benign, detector) \geq matchThreshold \Leftrightarrow match(detector, benign) \quad (2)$$

3.3 Feature Selection

After all features are extracted, PE header, DLLs import, API calls, and Code Sequences affinities are aggregated in order respectively into one vector per .EXE file. Then, the low variance feature selection applies on the train set to remove all columns which are significantly low variance like same value for all rows. This low variance

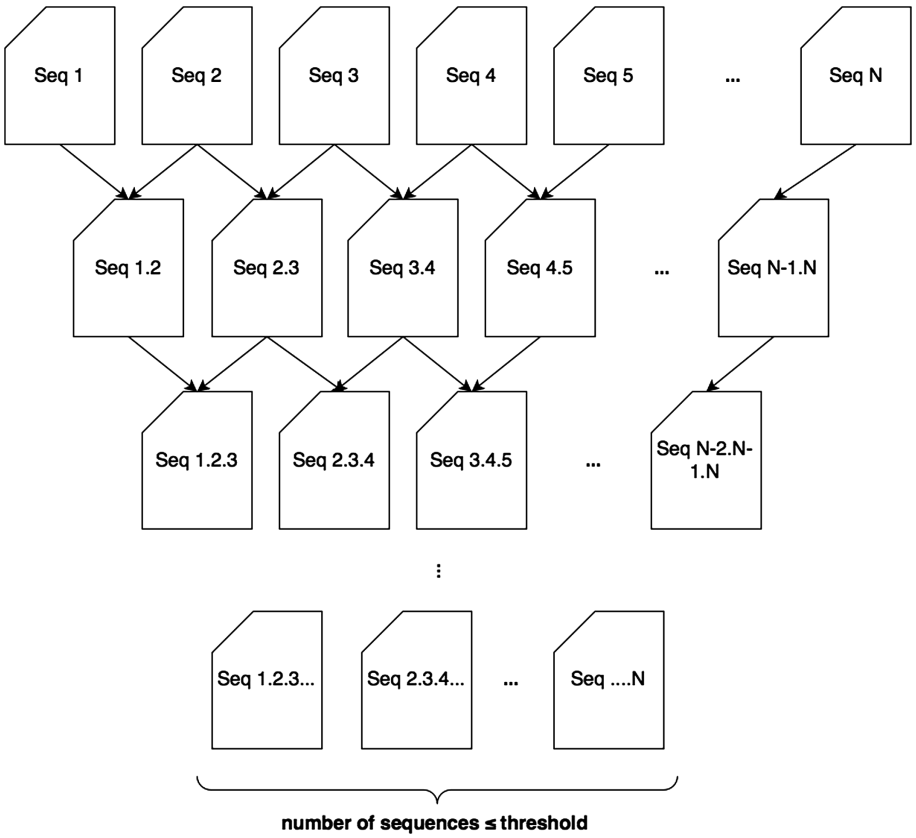

Fig. 4. Find local detectors process

feature selection applied on train set are used on correspond test set for the synchronization.

3.4 Classification

After all features are extracted and selected as shown in Fig. 1, they are separated by using classifiers. In this study, we used 3 classifiers: Gaussian Naïve Bayes (NB), Support Vector Machine (SVM) and Multilayer Perception with Rectified Linear Unit activation function (ReLU).

4 Experiment

4.1 Dataset

In this section, we described the datasets used in our study. We collected a lot of benign executable files from installed folders of applications of legitimate software from

different categories. They are all verified by VirusTotal.com [11]. We also collected 5 malware families from virusshare.com and MALICIA Set in [12]. Because of the obscured techniques and the anti-reverse techniques in some executable files, the tool dumpbin was unable to exploit information in that executables. Therefore, we cleaned and finally got dataset as Table 2.

Table 2. Dataset summary

Name	Amount	Source
Locker	330	virusshare.com
Mediyes	1640	
Winwebsec	4400	MALICIA set
Zbot	2100	
Zeroaccess	920	
Benign	300	Authentic sources

4.2 Evaluation Criteria

Because the objective of the malware detection system is to detect the new variants of well-known malware families, we evaluated on multi-classification to families, as shown on Table 2. Because of the sharp skew between families, this multi-classification uses precision, recall as evaluation metrics, which are defined as following:

- Precision is the fraction of relevant instances among the retrieved instances,

$$Precision = \frac{true\ positive}{true\ positive\ +\ false\ positive}$$

- Recall is the fraction of relevant instances that have been retrieved over total relevant instances,

$$Recall = \frac{true\ positive}{true\ positive\ +\ false\ negative}$$

(true positive: hit, false positive: false alarm, false negative: miss)

Another metric is F-score, which is the harmonic mean of precision and recall, could be useful for comparing models when one of those model is just higher than the other in either precision or recall.

$$F - score = 2.\frac{Precision.Recall}{Precision\ +\ Recall}$$

Besides, in a malware detection system, benign false alarm (categorize malwares as benign binaries) is very hazardous so *benign precision* is also considered as an important metric (higher benign precision is better).

4.3 Experimental Results

After running stratified 5-fold cross validation with matching threshold is 0.95, match-benign threshold is 0.8, generate-detector threshold is 0.6, low-variance threshold is 0 (features having same value all rows are removed); we tabulated our results for this study, as shown in Table 3. *Avg. Malware* is the average of figures of all malware families and other figures are the average of 5 results from 5 folds.

Overall, MLP achieves better results when comparing to the SVM and the NB in both benign precision and F-score for benign and average malware, respectively. We also observe that SVM has the lower performance than the others, only 0.4 precision, 0.013 F-score for benign, respectively. This may be because the model is not native multiclassification classifier. Besides, low number of samples of classes like benign (300) made them difficult to be converged to the high results as majority classes such as Winwebsec (4400), Zbot (2100), etc. Thus, most of the Winwebsec, Zbot result metrics are higher than 0.95 whereas these figures for Benign, Locker are just between 0.5 and 0.8. Also, the best model belonged to MLP with 2 hidden layers, which generally has better results in both benign file and malware than other classifiers, with 87.1% precision, 73.3% F-score for benign and 93% F-score for average malware. In addition, the best model also accomplishes the best figures of the table with 99.9% F-score, 100% recall for Winwebsec family – the largest family of the dataset.

Table 3. Summary of experimental result

Family			Benign	Locker	Mediyes	Winwebsec	Zbot	Zeroaccess	Avg. Malware
MLP	1 hidden layer	precision	0.667	0.647	0.957	0.998	0.942	0.981	0.905
		recall	0.617	0.773	0.998	0.975	0.961	0.888	0.919
		F-score	0.641	0.704	0.977	0.987	0.951	0.932	0.910
	2 hidden layers	precision	0.871	0.729	0.960	0.998	0.958	0.984	0.926
		recall	0.633	0.808	0.998	1.000	0.963	0.915	0.937
		F-score	0.733	0.767	0.978	0.999	0.960	0.948	0.930
	3 hidden layers	precision	0.787	0.737	0.949	0.997	0.957	0.981	0.924
		recall	0.667	0.717	0.998	0.991	0.952	0.973	0.926
		F-score	0.722	0.727	0.973	0.994	0.954	0.977	0.925
NB		precision	0.593	0.350	0.927	0.983	0.881	0.937	0.816
		recall	0.554	0.072	0.944	0.956	0.977	0.996	0.789
		F-score	0.573	0.119	0.936	0.970	0.927	0.966	0.783
SVM		precision	0.400	0.975	1.000	0.828	0.999	0.999	0.960
		recall	0.007	0.106	0.921	1.000	0.951	0.897	0.775
		F-score	0.013	0.191	0.959	0.906	0.975	0.945	0.795

In MLP, the number of node in each hidden layer was 2/3 of (number of feature +1), with random weights, learning rate was 0.001 and momentum was 0.5. For the SVM, the best result come with RBF kernel, one-vs-one strategy and C = 1.0, gamma = 1/(number of feature).

5 Conclusion

This paper proposed the combinational model of statistics and code sequence similarity algorithms which determines whether a new executable file is a metamorphic malware of a known family or benign. It can be used for new malware variant detection while most of new malwares are created based on known families nowadays. The experimental results showed that the proposed method can achieve better performance in the certain condition with MLP classifier.

However, the limitation of the proposed method is that several parameters of this method are not calculated automatically. The future works will investigate the dynamic features extracted automatically in sandbox; adding more families to widen detection ability, more samples to improve the balance of the data set and replacing LCS with better solution in finding detectors.

References

1. Symantec Corporation: Detecting Complex Viruses. https://www.symantec.com/connect/articles/detecting-complex-viruses. Accessed 10 June 2017
2. AV-TEST Institute: The AV-TEST Security Report, Magdeburg (2016)
3. Schultz, M.G., Eleazar, E., Erez, Z., Salvatore, S.J.: Data mining methods for detection of new malicious executables. In: IEEE Symposium Security and Privacy, S&P 2001, Proceedings, pp. 38–49 (2001)
4. Kolter, J.Z., Maloof, M.A.: Learning to detect and classify malicious executables in the wild. J. Mach. Learn. Res. **7**, 2721–2744 (2006)
5. Yuan, Z., Lu, Y., Xue, Y.: Droiddetector: android malware characterization and detection using deep learning. Tsinghua Sci. Technol. **21**(1), 114–123 (2016)
6. Rui, C., Tan, Y.: A virus detection system based on artificial immune system. In: Computational Intelligence and Security – CIS 2009, vol. 1, pp. 6–10 (2009)
7. Microsoft Corporation: Microsoft Portable Executable and Common Object File Format Specification, Microsoft Corporation (2017)
8. Microsoft Corporation: DUMPBIN Reference. https://msdn.microsoft.com/en-us/library/c1h23y6c.aspx. Accessed 10 June 2017
9. Shafiq, M.Z., Tabish, S.M., Mirza, F., Farooq, M.: PE-Miner: mining structural information to detect malicious executables in realtime. In: Kirda, E., Jha, S., Balzarotti, D. (eds.) RAID 2009. LNCS, vol. 5758, pp. 121–141. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04342-0_7
10. Microsoft Corporation: Desktop App Technologies, Microsoft Corporation. https://msdn.microsoft.com/library/windows/desktop/bg126469.aspx. Accessed 10 June 2017
11. Total, Virus: VirusTotal-Free online virus, malware and URL scanner (2017)
12. Antonio, N., Zubair, R.M., Juan, C.: The MALICIA dataset: identification and analysis of drive-by download operations. Int. J. Inf. Secur. **14**(1), 15–33 (2015)