Journal of Visual Languages and Computing 23 (2012) 154–162

Contents lists available at SciVerse ScienceDirect

Journal of Visual Languages and Computing

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

A graph mining approach for detecting unknown malwares ☆

Mojtaba Eskandari, Sattar Hashemi *

Department of Computer Science & Engineering, Shiraz University, Iran

ARTICLE INFO

Article history:
Received 3 December 2011
Received in revised form
14 January 2012
Accepted 19 February 2012
Available online 8 March 2012

Keywords:
Malware
Detection
Unknown malwares
PE-file
CFG
API

ABSTRACT

Nowadays malware is one of the serious problems in the modern societies. Although the signature based malicious code detection is the standard technique in all commercial antivirus softwares, it can only achieve detection once the virus has already caused damage and it is registered. Therefore, it fails to detect new malwares (unknown malwares). Since most of malwares have similar behavior, a behavior based method can detect unknown malwares. The behavior of a program can be represented by a set of called API's (application programming interface). Therefore, a classifier can be employed to construct a learning model with a set of programs' API calls. Finally, an intelligent malware detection system is developed to detect unknown malwares automatically. On the other hand, we have an appealing representation model to visualize the executable files structure which is control flow graph (CFG). This model represents another semantic aspect of programs. This paper presents a robust semantic based method to detect unknown malwares based on combination of a visualize model (CFG) and called API's. The main contribution of this paper is extracting CFG from programs and combining it with extracted API calls to have more information about executable files. This new representation model is called API-CFG. In addition, to have fast learning and classification process, the control flow graphs are converted to a set of feature vectors by a nice trick. Our approach is capable of classifying unseen benign and malicious code with high accuracy. The results show a statistically significant improvement over *n*-grams based detection method.

1. Introduction

Most of industrial antivirus softwares utilize signature-based methods to deal with malwares. Malware is any kind of software intentionally developed for malicious purposes without user's awareness. There are many kinds of malware such as computer virus, worm, trojan horse, spyware, adware, botnet, and so on [36,21]. However, signature-based methods can effectively detect known malwares; they cannot detect variants of known malwares or new ones. On the other hand, they require a database of signatures that needs update continuously. To keep this database up to date, security experts should analyze every new reported malicious program to extract the corresponding signature. A signature is usually a sequence of bytes, a regular expression, or in other formats hand crafted by security experts to determine specific malware. The complexity of such analysis is illustrated by [33] though of simplicity of computation and high detection accuracy rate. It was reported that they cannot detect unseen (unknown) malware whose signature has not been generated yet. So, it has a problem in covering all malwares because of an explosion in the number of malwares. The story becomes worse, a large number of malwares use evasion techniques to prevent signature-based detection methods using simple encryptions or code obfuscation. It can be easily overrun using simple program obfuscations

☆ This paper has been recommended for acceptance by Shi Kho Chang.

* Corresponding author. Tel.: +98 917 708 4720.
E-mail addresses: m_eskandari@cse.shirazu.ac.ir (M. Eskandari), s_hashemi@shirazu.ac.ir (S. Hashemi).

1045-926X/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jvlc.2012.02.002

due to the fact that this approach is sensitive to slight changes in malicious code [23].

To confront unknown malwares, behavior-based detection method is developed, which uses its knowledge of what constitutes normal behavior to decide the maliciousness of a program. These methods mostly use data mining techniques. The first known data mining application for malware detection is automatic signature extraction [20]. Results were not very impressive but it opened a way for data mining research in malware detection.

Data mining methods use statistical tools and machine learning algorithms on a set of features, to recognize the malicious programs from benign ones. There exists two important phases in all of these procedures. Realizing what features are suitable for further analysis and extracting and comparing these features to get better precision. *n*-Grams is the most common feature used by the data mining methods. An *n*-gram is a sequence of bytes of fixed or variable length, extracted from the hexadecimal dump of an executable program. *n*-Grams can extract from dynamic analysis reports of an executable program. The analysis can be performed at the source code level or at the binary level where the executable program is available. The information from the executables can be gathered by running them or performing static reverse engineering. The reverse engineering method is called static analysis while the process in which the program is actually executed to gather the information is called dynamic analysis. Many researchers utilize *n*-grams to detect unknown malwares [12,28,29,32]. *n*-Gram based methods have shown great promise in detecting zero-day attacks, but there are drawbacks related to these approaches. A zero-day attack or threat is a computer threat that tries to exploit computer application vulnerabilities that are unknown to others or the software developer. The two parameters generally associated with *n*-gram models are *n* and *L*. The parameter *n*, the length of the *n*-grams, takes values from 1 to the specified maximum. Larger values of *n* do not always result in a better performance. The maximum value of *n* should be chosen large enough as to demonstrate that the optimal value of *n* is within the tested range. The profile size, *L*, indicates the number of the most frequent *n*-grams of a file that are used as a signature for that file. Similar to *n*, larger values of *L* do not always result in a better performance and a maximum value should be chosen as to encompass the optimal value of *L*. In other words, for larger values of n and *L*, there is a much more expressive feature space that should be able to discriminate between malware and benign software more easily. But with these larger values of *n* and *L*, the feature space becomes too large and we do not have enough data to sufficiently condition the model. With smaller values of *n* and *L*, the feature space becomes too small and detection accuracy is lost.

In this paper we introduce a method that tries to understand the intent of a PE-file, namely a standard Windows portable executable file. Most of unknown malwares under this strategy can be detected with better flexibility and accuracy in comparison with other well known static methods. Also, it is worth mentioning that our method has low false positive rate with much less complexity in comparison with dynamic behavior methods. There exists two important phases in all of data mining based methods: first, finding most suitable features for further analysis and second find the best way to extract and compare these features to get better accuracy. Sometimes, these features can be *n*-grams, instruction sequences, API call sequences, dependency graph, control flow graph, etc. Control flow graph is a graph representing the structure sequence of a PE-file showing the mechanism of malware [6,8]. However control flow graph can give suitable information about malicious program structure, API call sequences and frequency in the code sequence have indicated valuable information about malicious activity [38,30]. API (Application Programming Interface) is an interface provided by operating system for program's interactions. Here, we enriched the simple control flow graph (CFG) with API call on its edges. We benefited appropriate features used by classifiers in order to classify malicious and benign PE-files. Final results show reasonable accuracy and detection rate in comparison with *n*-gram based method indicate satisfying improvement.

The structure of this paper is as follows. Section 2 describes related works in unknown malware detection. Section 3 discusses the proposed method, an overview of our malware detection system, feature selection methods and classification algorithms used in experiments. The system evaluation is presented in Section 4 whose results are interpreted and commented in the same section. Section 5 concludes our achievements, future works and summarizes the results.

2. Related work

From the early beginning, lots of works has been fulfilled on malware detection using structure sequence and control flow analysis [3,37]. Moreover API calls have been investigated for its beneficial information about malware activity [10,38]. Also machine learning has been utilized in various detection phases [31].

In the first days of virus appearance, only static and simple viruses were introduced to world. So, the simple signature based methods were able to overcome them [24,25]. These were good at the beginning but rapid evolution in malicious activities of malware persuaded researchers to turn on new methods. One of the most attractive methods that born at early days is applying data mining on *n*-grams. Gerald et al. proposed a method where extended the *n*-grams analysis to detect boot sector viruses using neural networks. The *n*-grams were selected based upon the frequencies of occurrence in viral and benign programs. Feature reduction was obtained by generating a 4-cover such that each virus in the dataset should have at least four of these frequent *n*-grams present in order for the *n*-grams to be included in the dataset [34].

Since *n*-grams fail to capture the semantics of the program, another feature should be used instead of it. Hofmeyr et al. were benefiting from simple sequences of system calls as a guidance to realize malicious codes [18]. These API call sequences showed the latent dependencies between code sequences. Bergeron et al. took the behavior and dynamic attributes into consideration to combat the metamorphism [4]. Gao et al. represented a new

method to find out differences in binary programs. Syntactic differences indeed have the potentiality to cause noise, so finding the semantic differences would be challenging. They utilized a new graph isomorphism technique and symbolic execution to analyze the control flow graph of PE-files by identifying the maximum common sub-graph. Their method found the semantic difference between a PE-file and its patched version; however, its false positive does not satisfy zero-day requirements [16]. Cesare and Xiang proposed a new classification method that used flow graphs to detect polymorphic malwares. They applied a heuristic algorithm for the flow graph matching to find graph isomorphisms; therefore, they could estimate similarity between PE-files, and finally they represented a classification algorithm based on their method [9]. Jeong and Lee extracted a graph by sequence of API calls and named that code graph. They worked on binary files directly and fetched the jump and call instructions from that PE-file and created the code graph. Their method can detect 67% of unknown malwares [19]. It seems that their achievement is realistic and has practical application, but the detection rate is low yet. Dullien and Rolles, assumed each function as a flow graph, and then drew call relation between them (i.e. from caller node to callee); consequently, each PE-file is represented as a graph of graphs. Under this trend, they could have an improved isomorphism between the sets of basic blocks and sets of functions in two disassembled PE-files. Therefore, they could detect code theft [15]. Abadi et al. formalized the semantic of Control Flow Integrity (CFI) by machine code rewriting. CFI represents a method to verify the execution proceeds within a given control flow graph which is derived from static program analysis. In other words, CFI relies on dynamic checks for enforcing control flow integrity and has an efficient implementation of a program shadow call stack with high protection [1].

3. Proposed approach

The general framework of our proposed approach is outlined in Fig. 1. This approach is composed of several phases. First, the system disassembles the PE-file to a standard assembly code and then extracts the CFG from the assembly code. A CFG (Control Flow Graph) is a connected and directed graph consisting of a set of vertices that correspond to the lines of assembly code, and a set of directed edges that correspond to the execution sequence of decision making operations (e.g., normal sequence, conditional jump, unconditional jump, function call and return instruction). Second, the system begins to make wealth the CFG with called API's. This simple CFG will turn out to API calls on CFG which we call API-CFG. Next, system uses a classification algorithm to make a decision, whether the PE-file is malicious or not.

3.1. Overall view

The proposed approach consists of three principal components, PE-file disassembler, API-CFG generator and classification module which are illustrated in Fig. 1. In the first step, the assembly code of input PE-file is generated. Next, the unnecessary instructions are removed from the assembly code by a simple pre-processor. Now, we have a clear code comprising only needed instructions to construct CFG. So, the CFG is created by this cleared code. On the other hand, all called API's are stored in API repository. In the API repository, each API name is mapped to a global unique number which is API-Id. After that, some edges of CFG are labeled by corresponding API-Id and finally API-CFG is attained. These edges are connected to an API call that is called on such statements of assembly code. Since performing a classification algorithm on these graphs is complex and takes significant time, a nice trick is used to convert these graphs into a set of vectors (i.e. one vector for each graph). These control flow graphs have considerable size; therefore, the vectors that crated with them are too large also. A feature selection algorithm is employed to reduce the size of these vectors. When these vectors are sufficient small the classification module can classify the input files quickly. After the feature selection, a subset of these vectors is selected as training set to construct the learning model by a machine learning algorithm. Decision module uses this learning model to realize the input file is malware or benign. In the following parts of this section, these phases are described by detail.

3.2. Constructing API-CFG

Based on our idea, feature extraction and feature selection are most considerable phases in malware detection systems. Therefore, the main part of our study is focused on these phases. Input PE-files are disassembled by a third party tool which is PE-Explorer [35]. The PE-Explorer disassembler is designed to be easy to use compared with other disassemblers. To that end, some of the functionality found in other products has been left out in order to keep the process simple and fast. While as powerful as the more expensive, dedicated disassemblers, PE-Explorer focuses on ease of use, clarity and navigation. This disassembler supports the most common Intel x86 instruction sets and extensions such as MMX, 3D Now!, SSE, SSE2 and SSE3. Furthermore, when a file is opened with PE Explorer, the UPX Unpacker plug-in detects whether the file is packed with UPX, and then the input file will be unpacked automatically. The UPX Unpacker plug-in works on packed malware executables and can handle a file even if it has been packed with UPX and modified manually so that UPX cannot be used directly to unpack the file, because internal structures have been modified, for example the names of the sections have been changed from UPX to XYZ, or the version number of the UPX format has been changed from 1.20 to 3.21. This technique often is used by malware authors to make unpacking and reverse engineering harder. Nevertheless, this tool cannot disassemble a set of PE-files automatically; hence, user must do it one by one. An auxiliary tool is developed to disassemble all files of a directory by utilizing PE-Explorer tool. This auxiliary tool read all needed files, passes them to PE-Explorer one by one, and save their assembly codes in corresponding files. Therefore, we can disassemble a set of PE-files automatically. At the pre-processing step the necessary instructions are remained and the others are removed from assembly

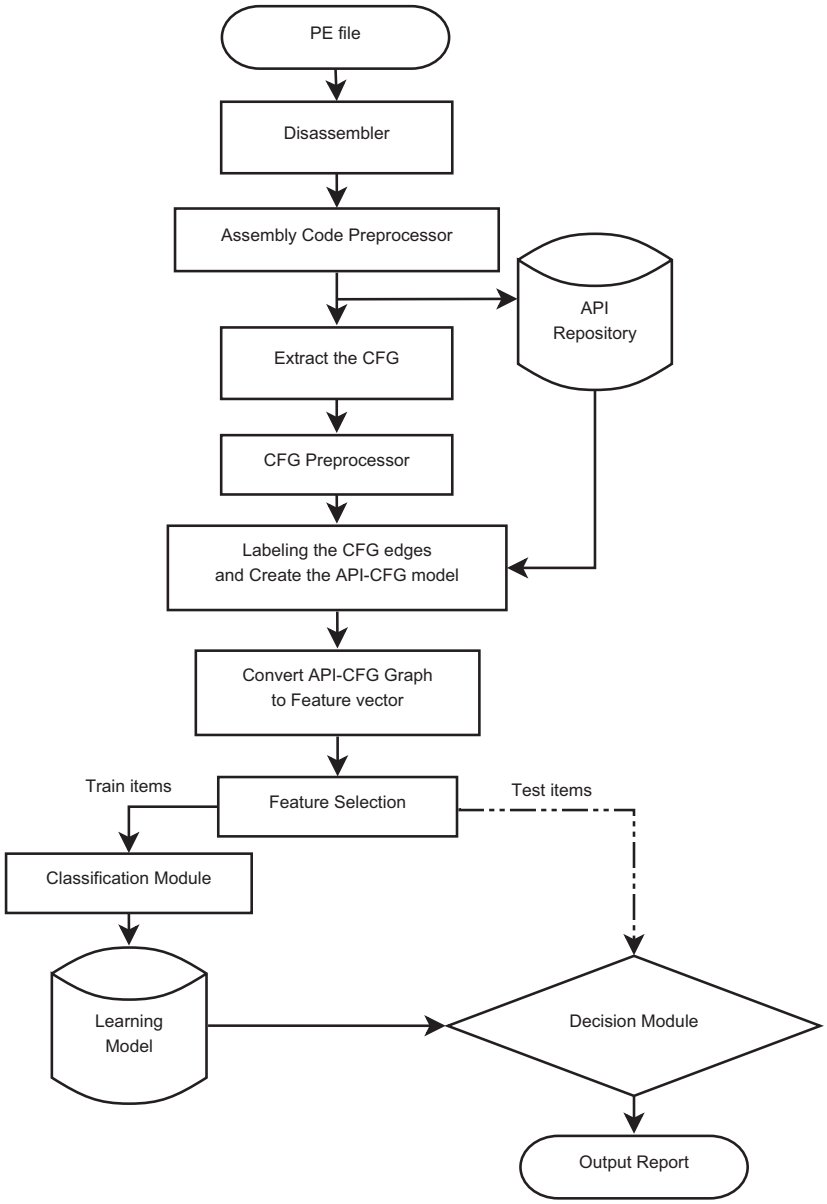

Fig. 1. System perspective. The detection process includes the following phases: control flow graph extraction, API-CFG generation, converting graph to feature vector, feature selection and classification phases. In the final phase, system can perform any classification algorithms to decide whether the input file is malware or not.

codes. The necessary instructions are as following: jump instructions, procedure calls, API calls and all lines which are targets of jump instructions. After that, control flow graph (CFG) is constructed with these instructions. As mentioned earlier, the vertices of CFG are formed by jump instructions, procedure calls and jump targets. The edges of CFG are created by relations between each two assembly statement. For example, a conditional jump has two relations with next statement and the jump target statement. So, in the CFG, this node (vertex) has two edges that point to next statement and jump target statement. When the CFG is constructed, it is traversed and put the related API-Id on its edges, and then API-CFG is attained.

3.3. Selecting suitable features

Since the graphs isomorphism problem is a well known NP-complete problem, we use a nice trick to convert the API-CFG to a feature vector. This trick uses a suitable property of API-CFG. This graph is a sparse graph, so it can be shown as a sparse matrix. On the other hand, a sparse matrix can be converted to a vector which stores only nonzero entries. The situation of each nonzero entry is obtained

$$s = (i-1)*n+j \quad (1)$$

where i is the row number, j is column number, n is the number of nodes in API-CFG and s is situation number of

item [*i*,*j*] in the sparse matrix. This value is computed for all nonzero items and then added to a list which is $S = \{s_1, s_2, s_3, \ldots, s_n\}$. In this vector each s denoted as a feature of PE-file; therefore, *S* represents a feature vector. In this feature vector, each item represents an edge and possibly its API call. An example of this conversion is illustrated in Fig. 2.

Each feature vector is an abstraction of an input PE-file. On the other hand, each input file in the training set has a label that indicates the file is malware or not. Therefore, we have a dataset to apply a learning algorithm on it. There is a problem to do it. Since the number of nodes (n) in various API-CFG's are different, dissimilar values of s is obtained for same [*i*,*j*] in several API-CFG's. To solve this problem, all API-CFG's must resize to a fix same size, hence, the size of largest API-CFG is used to satisfy this constraint. All graphs are padded to a same size to have a global size (n). When a general n is used, in Eq. (1), the features are meaningful, for example, in Fig. 2(c) number 20, in all graphs has same semantic.

The conversion process is shown in Algorithm 1. After conversion, a nominal dataset is generated by all feature vectors which indicates by $D = \{S_1, S_2, S_3, \ldots, S_k\}$. *D* is a nominal dataset where each data-item represents a PE-file and each feature represents an edge in corresponding API-CFG. As you can see, this dataset has lots of features for each item. A simple feature selection method is employed to reduce the number of features. All features gain their weights by Eq. (2), and then first-best set of features is selected according to their weights

$$W_k = \sum_{i=0}^{n} (\xi_{k,i} + \delta_{k,i} \cdot \eta) \tag{2}$$

where W_k is *k*th feature weight, n is the number of data items, η is API weight coefficient that would be obtained in experiments, $\xi_{k,i}$ denotes the profitable function for *k*th feature that is shown in Eq. (3) and $\delta_{k,i}$ represent the API exist function that is shown in Eq. (4). Since each API call has significant information in contrast to an edge of API-CFG, the value of η must be greater than 1. In this paper several experiments are done to find the best value for this coefficient

$$\xi_i = \begin{cases} +1 & \text{if } \theta = \gamma \\ -1 & \text{if } \theta \neq \gamma \end{cases} \tag{3}$$

where θ is the value of the feature and γ is the class label of *i*th data item

$$\delta_i = \begin{cases} 1 & \text{if have an API on this feature of } i\text{th data item} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Algorithm 1. Convert API-CFG to feature vector.

INPUT:
APIGraph is an API-CFG graph;
N is an integer number that represents the number of vertices of the largest graph;
OUTPUT:
Vector is an empty queue;

1: **Foreach** vector $\boldsymbol{V_i}$ in APIGraph **do**
2: **Foreach** outcome edge $\boldsymbol{E_{ij}}$ of V_i **do**
3: $\boldsymbol{Value_{ij}}$ = (label of $\boldsymbol{V_i} - 1$) × $\boldsymbol{N}$ + (label of target vertex of $\boldsymbol{E_{ij}}$);
4: $\boldsymbol{API_{ij}}$ = label of $\boldsymbol{E_{ij}}$;
5: Add pair ⟨$\boldsymbol{Value_{ij}}$,$\boldsymbol{API_{ij}}$⟩ to ***Vector***;
6: **End**;
7: **End**;

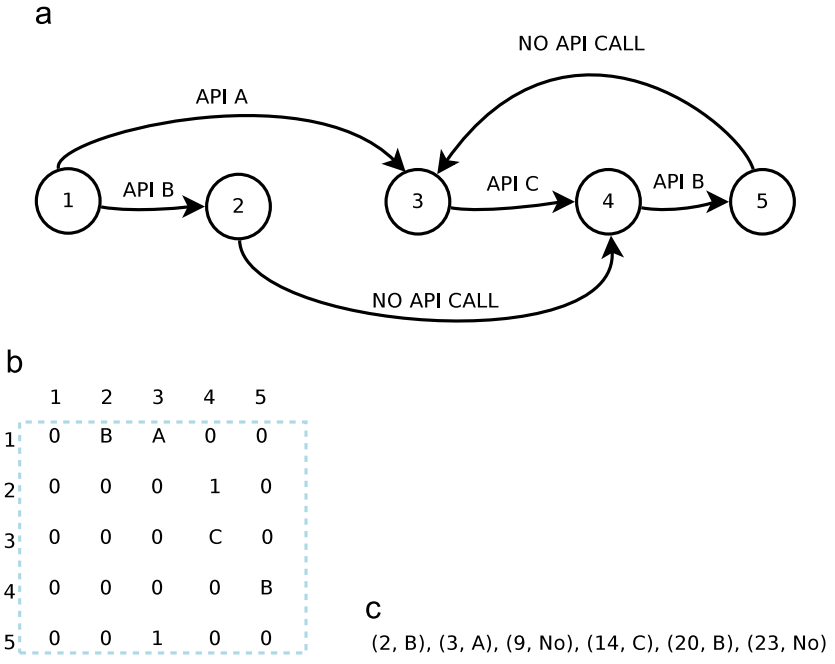

Fig. 2. Converting an API-CFG to a feature vector by Algorithm 1. (a) A CFG-API model. (b) Adjacency matrix of the graph. (c) Feature vector of that graph represents as sorted pair of edges and API calls.

3.4. Constructing the learning model

The input PE-files are partitioned into two categories which are training set and testing set. API-CFG's are generated to use in both training and testing sets. Each API-CFG in training set has a label that indicates its category, malware or benign. The learning model is constructed by utilizing a classifier applying on training set API-CFG's. For more exploration we present new results with different classifiers. The list of these classifiers is as follows: Decision Stump, SMO, Naïve Bayes, Random Tree, Lazy K-Star, and Random Forest.

As follows, a brief description of these classifiers is explained. A decision stump is a machine learning model consisting of a one-level decision tree [2]. That is a decision tree with one internal node (the root) which is immediately connected to the terminal nodes. A decision stump makes a prediction based on the value of just a single input feature. Sequential minimal optimization (SMO) is an algorithm for solving large quadratic programming (QP) optimization problems, widely used for the training of support vector machines. SMO breaks up large QP problems into a series of smallest possible QP problems, which are then solved analytically [27]. A naïve Bayes classifier assumes that the presence (or absence) of a particular feature of a class is unrelated to the presence (or absence) of any other feature. It is a kind of dependent constraint modeling [22]. A random tree is a tree or arborescence that is formed by a stochastic process. Different types of random trees include uniform spanning tree, random minimal spanning tree, random binary tree, random recursive tree, treap or randomized binary search tree, rapidly exploring random tree, Brownian tree, random forest and branching process [13]. K-Star is an instance-based classifier, which is the class of a test instance is based upon the class of those training instances similar to it, as determined by some similarity function. It differs from other instance-based learners in that it uses an entropy based distance function [11]. Random forest is an ensemble classifier that consists of many decision trees and outputs the class that is the mode of the class's output by individual trees [7].

4. Experimental results

There is no standard and valid benchmark for method comparison in the literature yet [17] and each scientific article considers the malware analysis, treats with its own datasets trying to implement different methods on their assessment. Therefore, we implement an n-gram based method and apply it on the dataset that our approach uses it. Since the n-gram methods need parameter selection, before we compare this approach with our approach, the parameter selection is gone. On the other hand, in the presented approach parameter needs be set. This parameter setting is done in same part also.

4.1. Datasets in brief

Most of malwares intend to infect PE-files; therefore, this study focuses on PE-files and analyzing malware that infect them. Many researchers, in this field, used an imbalanced dataset in their experiments in which the number of malwares is much more than the number of benign files [6,39,5]. However, this assumption does not stand in the real world domain, where the numbers of malicious binaries are at most as much as that of benign files. Clearly, it can be conclude that those methods, in which we have a considerable amount of malicious binaries compared to the benign ones, would provide better accuracy. According to this thought we collect 2140 benign Windows PE-files from a fresh installed Microsoft Windows XP SP3 and select 2305 Windows 32-bit network worms from malware repository of APA, Malware Research Center at Shiraz University, randomly.

4.2. Evaluation measures

First of all, it is needed to introduce some definitions for further measurements. The term True Positive (TP) shows the number of correctly recognized items which were belonging to the goal set. True Negative (TN) indicates the number of correctly realized items which were not belonging to the goal set. False Positive (FP) represents the number of incorrectly classified item which were belonging to the goal set. False Negative (FN) illustrates the number of incorrectly recognized item which were not belonging to the goal set.

We define "Detection Rate" as the percentage of all PE-files labeled "malicious" that can receive correct label by the system, as is illustrated

$$DetectionRate = \frac{TP}{TP+FN} \tag{5}$$

The "False Alarm Rate" is the percentage labeled "normal" that likewise receive the wrong label by the system, as is illustrated

$$FalseAlarmRate = \frac{FP}{TN+FP} \tag{6}$$

The "Accuracy" is the overall accuracy of the system to detect malwares and benign files, as is showed

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{7}$$

The "cross validation" is a technique for assessing how the results of a statistical analysis will generalize to an independent data set. It is mainly used in settings where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice. One round of cross validation involves partitioning a sample of data into complementary subsets, performing the analysis on one subset that called the training set, and validating the analysis on the other subset that called the testing set. To reduce variability, multiple rounds of cross validation are performed using different partitions, and the validation results are averaged over the rounds, we called each round as a fold [26].

The "ROC curve" is a graphical plot of true positive rate, versus false positive rate, for a binary classifier system as its discrimination threshold is varied. The area under the "ROC curve" called "AUC". The "AUC" is equal to the probability that a classifier will rank a randomly

chosen positive instance higher than a randomly chosen negative one [14].

4.3. Parameter selection

As mentioned ago, the presented approach has a parameter, η, that needs to be set. This parameter is used in Eq. (2) to demonstrate the importance of API calls in feature selection. Obviously, an API call has considerable information about program behavior; on the other hand, the structure of an executable file, CFG, has valuable information about the coding manner. According to these facts, if value of assume too large the role of CFG is missed; in contrast, if this parameter set as a small value the usability of API calls is lost. Since in a feature API call is exist beside a CFG edge data, and the information of API call is much more than edge information. So, the value of this parameter should be greater than 1. The result of this experiment is shown on Fig. 3. According to this figure, when the parameter η is set to 2.2 the system gain the best accuracy in detection.

There is another parameter selection to set n-gram analysis method parameters. Notice that the n-gram method is not contribution of this paper, this method is used only to show the superiority of the presented approach. There are two parameters that associated with n-gram model which are n and L. The parameter n, the length of the n-gram window, takes values from 1 to the specified maximum. Larger values of n do not always cause to have a better accuracy. The parameter L, profile size, shows the number of the most frequent n-grams of a file that are used as a determiner pattern for that file. Similar to n, larger values of L do not always result in a better performance and a maximum value should be chosen as to encompass the optimal value of L. Fig. 4 illustrates the results of this selection experiments.

Note that, in both of these experiments Random Forrest algorithm is used as classifier.

4.4. Observations and analysis

The experiments are done by different classifiers. These results are obtained with 10 fold cross validation test. The results of our approach are illustrated in Table 1

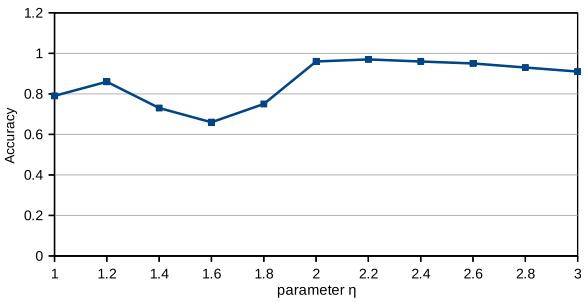

Fig. 3. Parameter selection for feature weighting process. The parameter η is the API coefficient that indicates the importance of API calls in feature selection. As you can see, the best value for this parameter is 2.2 that gained best accuracy in detection.

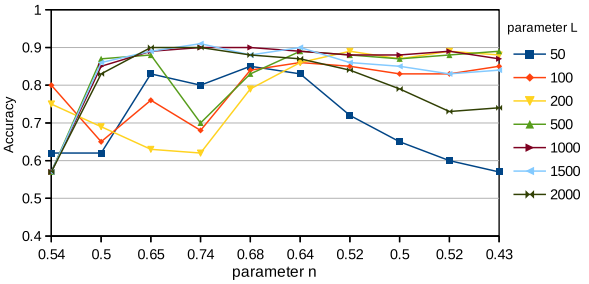

Fig. 4. Parameter selection for n-grams method. There are two parameters should be set to optimum value. According to these results, the best value for n is 4 and the optimum value for L is 1500.

Table 1
Experimental results for proposed approach. In this experiment different classifiers are used as decision module. According to these results the *Random Forest* classifier has gained the best accuracy.

	Detection rate (%)	**False alarm rate** (%)	**Accuracy** (%)	**AUC** (%)
Decision stump	93.81	04.95	94.42	92.29
SMO	89.74	08.20	90.76	87.23
Naïve Bayes	95.23	03.81	95.70	94.06
Random tree	91.11	07.11	91.99	88.94
Lazy KStar	96.61	02.71	96.94	95.78
Random forest	97.53	01.97	97.77	96.92

Table 2
Experimental results for n-grams method. These results indicate that n-grams method is outperformed by the presented approach.

	Detection rate (%)	**False alarm rate** (%)	**Accuracy** (%)	**AUC** (%)
Decision stump	87.39	4.17	88.85	87.65
SMO	84.55	8.54	85.18	83.77
Naïve Bayes	88.58	4.03	89.02	88.51
Random tree	87.27	7.63	87.41	85.48
Lazy KStar	91.29	2.59	91.50	90.16
Random forest	90.94	1.95	92.19	91.16

and experimental results of n-grams method are shown in Table 2. The results indicate that the method based on API-CFG provides excellent detection rate and lower false alarm rates than n-gram based approaches for unknown malwares. Detection ability per different classifiers is comparable by "AUC" measure.

Fig. 5 shows the accuracy comparison between n-grams method and our approach. According to this figure our approach attains better accuracy than n-grams. The presented approach utilizes both of the API calls and control flow graph together to model the input program. A set of called API's indicates the behavior of a program and a CFG represents the structure of an executable file. Both of these models are used to represent a PE-file as a data item. A classifier is applied on a set of these data

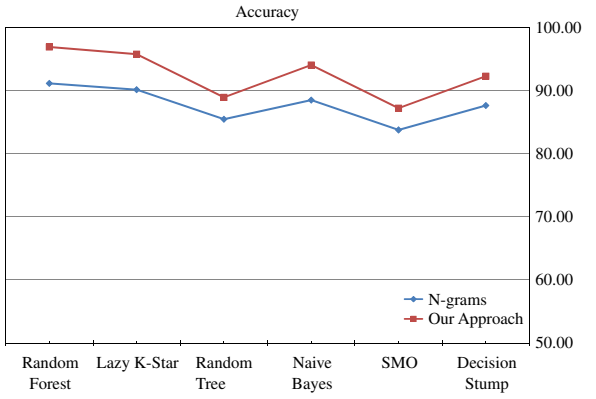

Fig. 5. Accuracy comparison between presented approach and *n*-grams method.

items to create a learning model. This learning model is utilized by a decision module to determine a PE-file is malware or not. So, this study combines the behavioral model and structural aspect of a program to make more accurate decision about its suspiciousness. Therefore, our presented approach acquires more semantic features of a PE-file than traditional *n*-gram based methods. Consequently, our approach is more accurate than *n*-grams method at detection phase.

5. Conclusions and future work

Control flow graphs help us to have a semantic aspect of the PE-files, which can detect them with mining in these graphs. The presented approach enriches the current control flow graph by adding the behavioral property of a malicious program to it. This enriched model, API-CFG, is able to detect unknown malwares strongly. On the other hand, graph isomorphic is an NP-complete problem; hence, it takes lots of time to utilize it in data mining algorithms. Therefore, we proposed a trick to simplify the control flow graphs and facilitate the mining process. This trick converts each control flow graph to a feature vector. Since the number of features, in this vector, is too large, we utilize a feature selection algorithm in our approach. This feature selection method takes both structural information and behavioral one into consideration. Behavioral information is gained by called API's. As matter of fact, an API call has significant information rather than a small sub-graph or a graph edge. So, this importance is satisfied by adding a tunable coefficient to API call features in feature selection equation. This equation is tuned by an experiment.

At the final step for making decision about maliciousness of a new executable file, a classifier is employed to train with seen executable files. This classifier uses those feature vectors as data items. As results are shown, "Random Forrest" detects unknown malwares more accurate than the other classifiers. Therefore, it would be better that this classifier is employed in a practical intelligent malware detection system.

Since, graph mining methods can deal with many graph modification problems, we intend to expand this approach to work with metamorphic malwares which their codes are mutated by control flow modification techniques, that is set of techniques alters the execution sequence of malware.

Acknowledgments

This work is supported in part by APA, Malware Research Center at Shiraz University.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, J. Ligatti, Control-flow integrity principles, implementations, and applications, ACM Transactions on Information and System Security (TISSEC) 13 (1) (2009) 4.
[2] W. Ai, P. Langley, Induction of one-level decision trees, in: Proceedings of the Ninth International Conference on Machine Learning, Citeseer, 1992.
[3] W. Arnold, G. Sorkin, Automated Analysis of Computer Viruses, 1996.
[4] J. Bergeron, M. Debbabi, M. Erhioui, B. Ktari, Static analysis of binary code to isolate malicious behaviors, in: IEEE Eighth International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises. (WET ICE'99) Proceedings, IEEE, 1999, pp. 184–189.
[5] G. Bonfante, M. Kaczmarek, J. Marion, Architecture of a morphological malware detector, Journal in Computer Virology 5 (3) (2009) 263–270.
[6] G. Bonfante, M. Kaczmarek, J. Marion, et al., Control Flow to Detect Malware, 2007.
[7] L. Breiman, Random forests, Machine Learning 45 (1) (2001) 5–32.
[8] D. Bruschi, L. Martignoni, M. Monga, Detecting self-mutating malware using control-flow graph matching, Detection of Intrusions and Malware & Vulnerability Assessment (2006) 129–143.
[9] S. Cesare, Y. Xiang, A fast flowgraph based classification system for packed and polymorphic malware on the endhost, in: 24th IEEE International Conference on Advanced Information Networking and Applications (AINA), IEEE, 2010, pp. 721–728.
[10] F. Chen, Y. Fu, Dynamic detection of unknown malicious executables base on api interception, in: First International Workshop on Database Technology and Applications, IEEE, 2009, pp. 329–332.
[11] J. Cleary, L. Trigg, K*: an instance-based learner using an entropic distance measure, in: International Workshop then Conference on Machine Learning, Morgan Kaufmann Publishers, Inc., 1995, pp. 108–114.
[12] J. Dai, R. Guha, J. Lee, Efficient virus detection using dynamic instruction sequences, Journal of Computers 4 (5) (2009) 405–414.
[13] T. Dietterich, An experimental comparison of three methods for constructing ensembles of decision trees: bagging, boosting, and randomization, Machine Learning 40 (2) (2000) 139–157.
[14] L. Dodd, M. Pepe, Partial auc estimation and regression, Biometrics 59 (3) (2003) 614–623.
[15] T. Dullien, R. Rolles, Graph-based comparison of executable objects (English version), Symposium sur la sécurité des technologies de l'information et des communications 5 (2005) 1–3.
[16] D. Gao, M. Reiter, D. Song, Binhunt: automatically finding semantic differences in binary programs, Information and Communications Security (2008) 238–255.
[17] D. Harley, Making Sense of Anti-malware Comparative Testing, Information Security Technical Report 14 (1) (2009) 7–15.
[18] S. Hofmeyr, S. Forrest, A. Somayaji, Intrusion detection using sequences of system calls, Journal of Computer Security 6 (3) (1998) 151–180.
[19] K. Jeong, H. Lee, Code graph for malware detection, in: International Conference on Information Networking. ICOIN 2008, IEEE, 2008, pp. 1–5.
[20] J. Kephart, W. Arnold, Automatic extraction of computer virus signatures, in: Fourth Virus Bulletin International Conference, 1994, pp. 178–184.
[21] D. Kienzle, M. Elder, Recent worms: a survey and trends, in: Proceedings of the 2003 ACM workshop on Rapid Malcode, ACM, 2003, pp. 1–10.
[22] D. Lewis, Naive (Bayes) at forty: the independence assumption in information retrieval, Machine Learning: ECML 98 (1998) 4–15.
[23] A. Moser, C. Kruegel, E. Kirda, Limits of static analysis for malware detection, in: Twenty-Third Annual Computer Security Applications Conference, ACSAC 2007, IEEE, 2007, pp. 421–430.
[24] G. Necula, Proof-carrying code, in: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 1997, pp. 106–119.

[25] V. Paxson, Bro: a system for detecting network intruders in real-time, Computer Networks 31 (23–24) (1999) 2435–2463.
[26] R. Picard, R. Cook, Cross-validation of regression models, Journal of the American Statistical Association (1984) 575–583.
[27] J. Platt, Fast Training of Support Vector Machines Using Sequential Minimal Optimization, 1998.
[28] D. Reddy, S. Dash, A. Pujari, New malicious code detection using variable length *n*-grams, Information Systems Security (2006) 276–288.
[29] D. Reddy, A. Pujari, *n*-Gram analysis for computer virus detection, Journal in Computer Virology 2 (3) (2006) 231–239.
[30] V. Sathyanarayan, P. Kohli, B. Bruhadeshwar, Signature generation and detection of malware families, in: Information Security and Privacy, Springer, 2008, pp. 336–349.
[31] A. Shabtai, R. Moskovitch, Y. Elovici, C. Glezer, Detection of malicious code by applying machine learning classifiers on static features: a state-of-the-art survey, Information Security Technical Report 14 (1) (2009) 16–29.
[32] S. Stolfo, K. Wang, W. Li, Towards stealthy malware detection, Malware Detection (2007) 231–249.
[33] P. Ször, P. Ferrie, Hunting for metamorphic, in: Virus Bulletin Conference, 2001.
[34] G. Tesauro, J. Kephart, G. Sorkin, Neural networks for computer virus recognition, IEEE Expert 11 (4) (1996) 5–6.
[35] P.-E. Tool, 2011. 〈http://www.pe-explorer.com/peexplorer-tour-disassembler.htm〉.
[36] T. Tran, K. Alsubhi, A Comprehensive Survey on Malware and Malware Detection Techniques, CS 854-Hot Topics in Computer and Communications Security Project Report, 2006.
[37] P. Vinod, V. Laxmi, M. Gaur, G. Kumar, Y. Chundawat, Static cfg analyzer for metamorphic malware code, in: Proceedings of the Second International Conference on Security of Information and Networks, ACM, 2009, pp. 225–228.
[38] J. Xu, A. Sung, P. Chavez, S. Mukkamala, Polymorphic malicious executable scanner by api sequence analysis, in: Fourth International Conference on Hybrid Intelligent Systems. HIS'04, IEEE, 2004, pp. 378–383.
[39] Y. Ye, D. Wang, T. Li, D. Ye, Q. Jiang, An intelligent pe-malware detection system based on association mining, Journal in Computer Virology 4 (4) (2008) 323–334.