SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2015; **8**:3311–3322
Published online 22 April 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1255

RESEARCH ARTICLE

DLLMiner: structural mining for malware detection

Masoud Narouei[1], Mansour Ahmadi[2]*, Giorgio Giacinto[2], Hassan Takabi[1] and Ashkan Sami[3]

[1] Department of Computer Science and Engineering, University of North Texas, Denton, TX, U.S.A.
[2] Department of Electrical and Electronic Engineering, University of Cagliari, Italy
[3] CSE and IT Department, School of Electrical and Computer Engineering, Shiraz University, Shiraz, Iran

ABSTRACT

Existing anti-malware products usually use signature-based techniques as their main detection engine. Although these methods are very fast, they are unable to provide effective protection against newly discovered malware or mutated variant of old malware. Heuristic approaches are the next generation of detection techniques to mitigate the problem. These approaches aim to improve the detection rate by extracting more behavioral characteristics of malware. Although these approaches cover the disadvantages of signature-based techniques, they usually have a high false positive, and evasion is still possible from these approaches. In this paper, we propose an effective and efficient heuristic technique based on static analysis that not only detect malware with a very high accuracy, but also is robust against common evasion techniques such as junk injection and packing. Our proposed system is able to extract behavioral features from a unique structure in portable executable, which is called dynamic-link library dependency tree, without actually executing the application.

KEYWORDS
malware analysis; dependency tree; closed frequent tree; evasion

***Correspondence**
Mansour Ahmadi, University of Cagliari, Italy.
E-mail: mansour.ahmadi@diee.unica.it

1. INTRODUCTION

Malware is the short form for malicious software that consists of different hostile code combinations. Malware is mainly designed to gain unauthorized access, steal user's critical information, and sometimes cause damage to computers. Malware is categorized into viruses, worms, Trojan horses, spywares, rootkits, scareware, adware, riskware, and so on. According to [1], "As much malware was produced in 2007 as in the previous 20 years altogether." In 2011, Kaspersky [2] stated that about 413 billion infections have been detected by host-based anti-viruses. Symantec [3] encountered more than 286 million unique variants of malware and also Web-attack toolkits, and a phenomenon called Polymorphism drive the number of malware variants in common circulation.

Improving the intelligence of anti-malware software is critical. A lot of effort has been put in the area of malware detection, which can be broadly categorized into static [4] and behavioral-based [5] malware detection techniques. Static detection techniques such as anomaly, heuristic or signature-based, mostly operate on disassembled instructions or Application Programming Interface (API) calls. These techniques use a specific signature of each malware, such as byte sequences or strings, for detection. The key advantage of these approaches is their high detection rate and low false alarms. Another benefit of these approaches is their speed in extracting features because they do not need to run each executable file, which is a time consuming task. New "anti-anti-malware" techniques such as code packing, control-flow and entry point obfuscation change the specific signature of discovered malware, making static analyzers toothless in detecting previously detected malware [6]. Although there are lots of learning-based detection techniques such as [7,8] or, distance-based signature matching [9], evasion from them is still possible for new malware variants. Therefore, a malware detection system is actually a trade-off between detection rate and robustness that should be considered during development.

Because of the weaknesses of static-based detection techniques, dynamic analysis techniques have made lots of progress in recent years. In comparison to static-based, dynamic analysis techniques consider the behavior of malware during run time. They mostly monitor the behavior of malware using API calls [10,11]. Dynamic-based techniques act almost robustly against polymorphism malware because obfuscation techniques only change the static

signature of malware and do not affect the behavior of the malware. Despite of its advantages, dynamic-based techniques have a large preprocessing overhead during run time and monitoring, which lowers system performance. In fact, dynamic analysis is usually performed when anti-malware companies want to look for a new sample in the wild and then they use its static features in the anti-malware products. Therefore, implementation of these techniques in the commercial host-based anti-malware products is not applicable. Another drawback of such approaches is that monitoring behavior of dynamic libraries, such as OCX and dynamic-link library (DLL), is difficult to perform. According to [12], about 60% of malwares collected at KingSoft anti-malware lab are DLL files, which cannot be run and analyzed dynamically.

A portable executable (PE) depends on some DLLs for execution, and each DLL has a relationship with other DLLs for completing the task. This hierarchical dependencies between PE and DLLs is known as DLL dependency tree. In this paper, we will propose a hybrid system that by statically extracting the DLL dependency tree from the PE, has the advantages of both static and dynamic techniques. Although the PE is not executed, the extracted tree can reveal coarse-grain behaviors of a PE because it is created based on the interactions of the PE with the operating system, which leads to relationships among DLLs. The novelty of our method is the extraction of the coarse-grain behaviors from PE's import address table (IAT), which was not considered in previous methods. It means that the analysis system is also very fast because it only considers the header of PE, and also, the extracted behaviors are semantic and influential to discriminate malicious and benign programs. In addition of the efficiency of static methods and the effectiveness of dynamic approaches, our method cannot simply be circumvented. The evaluations show the effectiveness against several kinds of packed malware as well as robustness against junk injection techniques. The rest of the paper is organized as follows: related work are renewed in section 2; definitions are presented in section 3, and section 4 presents the proposed method. Results of the experiments are discussed in section 5, and conclusions and future work will wrap up the paper.

2. RELATED WORK

Analysis techniques for detecting malicious programs are usually divided into static and dynamic methods. The technique of analyzing a program without its execution is called static analysis, while dynamic analysis addresses techniques that analyze a program by actually running it. Analyzers extract various structures from programs. If the source code is available, it is possible to extract the control flow graph from programs [13–15], and the graph is then used for analysis. But in general, analyzers usually use signatures [16] because of the lack of source-code, disassemble malware with disassemblers [17], and extract different structures like opcode [18] from the output, or directly focus on the byte code [19,20]. Sung *et al*. [21] proposed SAVE that considers a sequence of API calls as signature of malware. Kumar and Spafford [22] proposed a method that detects malware based on regular expression matching. Christodorescu and Jha [9] proposed SAFE that takes patterns of malicious behavior and then creates an automata from it. Some other approaches [7,8] considered IAT of PE and its content. Another work [23] also considered some metadata, such as number of bitmaps, size of import, and export address table, besides IAT content. There is only one research [24] that considered DLL dependency of PE but they did not extract semantic behaviors from all DLLs. They considered single APIs and DLL dependency paths of the tree as features that can be circumvented by junk injection techniques. In contrast, our method is more resilient against evasion techniques and also the extracted behaviors by our method is more semantic because we consider the relationships between all DLLs not only the paths of the DLL dependency tree. The last four approaches are the most similar methods to ours but opposite to our approach, they did not consider any semantic relation between all DLLs.

Besides static analysis, researchers have put a lot of effort in proposing behavior-based malware detection methods that monitor programs during the run-time. Different approaches and techniques can be applied to perform such dynamic analysis. In this technique, analyzers usually consider system calls or instructions sequences as discriminating features between malware and benign programs. Some approaches [11,25,26], monitored the program's behavior by analyzing API calls. In another work [27], the authors extracted sequences of instructions from both malicious and benign programs and use some combination of frequent assembly instructions to build the dataset. In some techniques [28,29], a behavioral graph is extracted from each malware based on API call parameters and understood behavior of malware. Instead of program-centric detection approaches, some approaches try to generalize the detection method to the whole system. Lanzi *et al*. [30] proposed an access activity model that captures the generalized interactions of benign applications with operating system resources such as files and registry, and the model can detect the malware well with very low false positive. Nappa *et al*. [31] also proposed a technique to detect exploited servers, managed by the same organization. They collected information about how exploited servers are configured and the kind of malware they distributed and then grouped servers with similar characterization over time.

Static analysis techniques find it hard to cope with code obfuscation, polymorphism, and metamorphism. The deficiencies of dynamic analysis are run-time overhead and weakness in monitoring DLLs. Our method covers these deficiencies and takes into account a tree structure called DLL dependency that reveal coarse-grain behaviors of malware without actually running it. These behaviors were represented by sub-trees in DLL dependencies,

DOI: 10.1002/sec

Table I. A comparison between analysis techniques and their pitfalls (each part is sorted by year).

Authors	Platform	Feature	DR (%)	FP (%)	Freq pat	Pitfall
Dynamic analysis techniques						
Dai *et al.* [27]	Windows	OP SEQ	91.9	9.6	×	Reordering, substitution, injection
Kolbitsch *et al.* [28]	Windows	API GRA	64	×	×	Graph complexity
Fredrikson *et al.* [35]	Windows	API GRA	86	0	*GRA*	Graph complexity
Rieck *et al.* [10]	Windows	API NG	≈99	≈1	×	Reordering, injection
karbalaie *et al.* [36]	Windows	API GRA	96.6	3.4	*GRA*	Graph complexity
Ahmadi *et al.* [11]	Windows	API SEQ	98.1	11.9	*SEQ*	Reordering, injection
Perdisci *et al.* [37]	Windows	HTTP	≈68	≈0	×	HTTP specific malware
Palahan *et al.* [38]	Windows	API GRA	86.77	0	*GRA*	Graph complexity
Static analysis techniques						
Ye *et al.* [8]	Windows	HED API	97.19	12.5	*ITM*	Injection
Shafiq *et al.* [23]	Windows	HED MET	>99	<1	×	Injection
Tabish *et al.* [19]	Windows	BYT NG	>90	×	×	Substitution, reordering, injection
Griffin *et al.* [16]	Windows	BYT SEQ	×	<0.1	×	Substitution, reordering, injection
Kim *et al.* [29]	Windows	DG SRC	88.9%	×	×	Based on *SRC*
Sami *et al.* [7]	Windows	API	99.7	1.5	*ITM*	Injection
DiCerbo *et al.* [33]	Android	PER	×	×	*ITM*	Injection
Shahzad *et al.* [39]	Linux	HED	>99	<0.1	*ITM*	Injection
Santos *et al.* [18]	Windows	OP SEQ	≈95.8	≈2	×	Substitution, reordering, injection
Yang *et al.* [34]	Android	CFG, API	95.3	0.4	*ITM*	Obfuscation
Nissim *et al.* [20]	Windows	BYT NG	≈85	≈0.8	×	Substitution, reordering, injection

DR, detection rate; FP, false positive; *PER*, permissions; *HED*, executable header; *API*, application programming interface; *OP*, operation code; *DG*, dependency graph; *CFG*, control flow graph; *SRC*, source code; *NG*, n-gram; *MET*, metadata; *BYT*, byte code; *SEQ*, sequence; *GRA*, graph; *ITM*, item-set.

which were not considered in previous works. Some approaches consider frequent patterns [32] such as item-sets [7,8,33,34], sequences [11], or graphs [35,36] for malware detection but to the best of our knowledge, this is the first work that considers this tree structure for malware detection.

An overall comparison between dynamic and static analysis approaches is shown in Table I. As our proposed system tries to detect Windows malware families, we focus on systems that work on Windows malware and only one of these systems analyzes Linux executable. There are also two systems that use frequent patterns as features for Android platform and are included in the diagram because our approach is also using frequent patterns. In the table, the third column shows the type of features that are used by the systems. DR column shows the detection rate of the related system (× means that the authors did not evaluate on malware dataset or did not mention in the paper). FP shows the false positive of the system (× means that the authors did not evaluate on benign dataset or did not mention in the paper). Freq pat column shows if the system uses frequent patterns like frequent sub-graphs, item-sets or sub-sequences as features for extracting behaviors and then uses them for detection. The final column shows the challenge of each method for detecting malware and mainly what are the possible evasion techniques against them. Although there are many coarse-grain evasion techniques for both dynamic (like virtual machine detection) and static analyses (like packing), here, we mentioned some finer-grain evasion techniques like changing instructions with their equivalence (substitution), changing order of APIs or assembly instructions (reordering) and injecting junk data (injection).

3. DEFINITIONS

In this section, some definitions that are used in this research are introduced.

Definition 1. *(Tree) A tree is an acyclic connected directed graph with the node at the top defined as the root*[T]. *A tree is denoted by* $T = (r, V, E, \sum, L)$, *where r the root has no entering edges; V is the set of nodes; E is the set of edges in the tree;* $\sum$ *is an alphabet; L is a function:* $V \rightarrow \sum$ *that assigns labels to nodes.*

Definition 2. *(Subtree) For a tree T with node set V, edge set E, we say that a tree* T' *with node set* V', *edge set* E', *is a subtree of T if and only if (1)* $V' \subseteq V$; *(2) the labeling of nodes of V in T is preserved in* T'; *(3)* $(v1, v2) \in E'$, *where* $v1$ *is the parent of* $v2$ *in* T' *if and only if* $v1$ *is the parent of* $v2$ *in T; (4) for* $v1, v2 \in V'$, $preorder(v1) < preorder(v2)$ *in* T' *if and only if* $preorder(v1) < preorder(v2)$ *in T. And if* T' *is a subtree of T , it will be denoted as* $T' \in T$.

Definition 3. *(Parent) the parent node of* v(*parent*[v]) *is defined as its predecessor, and each node in the tree can only have one parent.*

Definition 4. *(Unordered subtree) A tree $T'(r', V', E', L')$ is an unordered tree of a tree $T(r, V, E, L)$ if and only if (1) $V' \subseteq V$, (2) $E' \subseteq E$, (3)$L' \subseteq L$ and $L'(v) = L(v)$, (4) $\forall v' \in V', \forall v \in V, v'$ is not the root node, and v' has a parent in T', then parent(v') = parent(v) and the left-to-right ordering among siblings in T' does not need to be preserved.*

Definition 5. *(Isomorphic trees) Two labeled rooted unordered trees $T1$ and $T2$ are isomorphic to each other if there is a one-to-one mapping from the vertices of $T1$ to the vertices of $T2$ that preserves vertex labels, edge label , adjacency, and the root. A subtree isomorphism from $T1$ to $T2$ is an isomorphism from $T1$ to some subtree of $T2$.*

Definition 6. *(Frequent subtree) Let D denote a dataset (list) where each element $T \in D$ is a labeled rooted unordered tree. For a given pattern t, which is a rooted unordered tree, we say t occurs in a transaction T if there exists at least one subtree of T that is isomorphic to t. The occurrence $\delta_t(T)$ of t in T is the number of distinct subtrees of T that are isomorphic to t. Let $\sigma_t(T) = 1$ if $\delta_t(T) > 0$, and 0 otherwise. We say T supports pattern t if $\delta_t(T) > 0$, and we define the supports of a pattern t as Supp$(t) = \sum \sigma_t(T)$ where $T \in D$.*

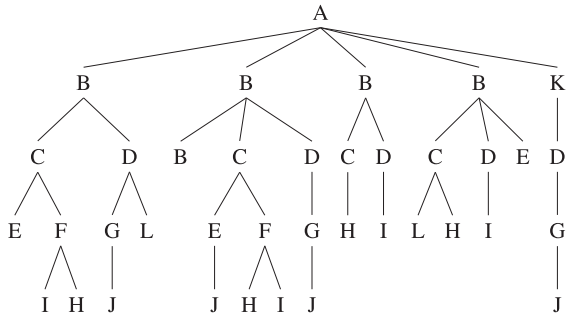


Figure 1. Sample tree.

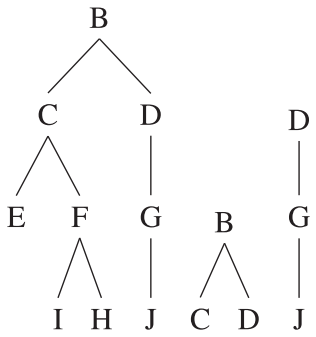


Figure 2. Three closed frequent trees.

Definition 7. *(Frequent pattern) A pattern t is called frequent if its support is greater than or equal to a minimum support (minsup) specified by a user.*

The frequent subtree mining problem is to find all frequent subtrees in a given dataset. A challenging step toward mining frequent trees from a large dataset is that such mining often leads to generating a huge number of subtrees, which satisfy the minsup threshold. This is because of a priory property that if a tree is frequent, any of its subtrees is also frequent. To overcome this difficulty, we used closed frequent trees.

Definition 8. *(Closed frequent tree) Let B_t be the set of all super trees of t that have one more vertex than t, a frequent subtree t is closed if and only if for every $t' \in B_t$, support(t') < support(t).*

Definition 9. *(String encoding format) Tree's String encoding is obtained by Tree's depth first traversal.*

The reason for mining closed frequent trees instead of all frequent trees is that the total number of closed frequent trees is far less than the number of frequent trees. Because the set of closed frequent trees maintain the same information as the set of all frequent trees, we only consider closed frequent trees. Figure 1. shows three closed tree patterns of the tree in Figure 2., with minimum support of two.

4. PROPOSED SYSTEM

An overview of the system is shown in Figure 3. Each step is described in the following: First, dependency trees from both malicious and benign PEs are extracted. These trees are then converted to their string encoding format. After that, closed frequent trees are extracted and feature vectors are constructed based on the closed frequent trees. Finally, the vectors are classified by Random Forest classifier using 10-fold cross validation.

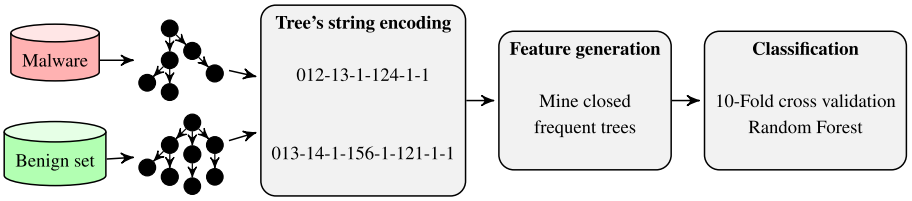


Figure 3. Our system.

DOI: 10.1002/sec

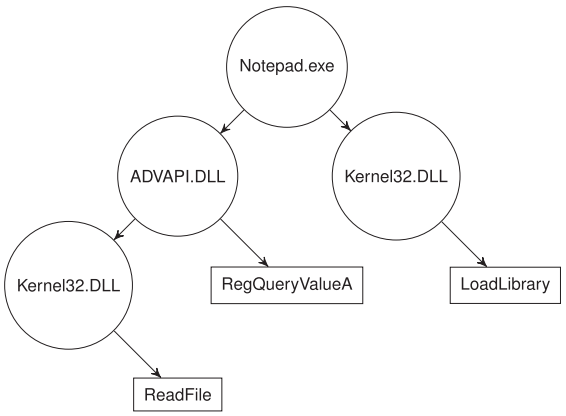


Figure 4. An example of DLL dependency tree. DLL, dynamic-link library.

4.1. Dependency tree generation

Every PE file needs to interact with the operating system in order to perform its task. This interaction is implemented by application programming interfaces which are categorized into different DLLs. This way, each PE depends on some DLLs for execution and each DLL has a relationship with other DLLs for completing the task. For example, considering the Notepad, it needs to check user configuration before application start-up and so it must search the Registry. Notepad needs LoadLibrary from kernel32.DLL for loading ADVAPI32.DLL which uses RegQueryValueA from ADVAPI32.DLL for reading a registry value. Then ADVAPI32.DLL uses ReadFile from kernel32.DLL to read data from a file (Figure 4). This is a sub behavior from Notepad that can be visualized as a tree. Although we can consider such structural relationship as a graph because of repetitive nodes, a tree is more descriptive and can help us find more unknown behavior. The key point of our proposed method is that using behaviors such as stealth, self-replication, disguising network traffic can distinguish malware from benign PEs.

In order to generate the behavioral tree of each malicious and benign PE, we use Dependency Walker [40]. The Dependency Walker's tree displays a hierarchical view of all the modules' dependencies. A module is dependent on another module in five ways[†]: Implicit Dependency, Delay-load Dependency, Forward Dependency, Explicit Dependency, and System Hook Dependency. The first three are defined statically, and the latter two are defined dynamically. Because we want to minimize the run-time overhead, we only use the first three cases. Dependency Walker extracts such dependencies from IAT, which resides within win32 executable application. When the application calls a windows API function, the IAT is used as a lookup table. Each version of Microsoft Windows operating system has a different address for API functions because of differently structured DLLs. When an application starts, it has a list of all functions that are not originally part of the application. These functions, called imports, are located in the operating system's DLL, but the application does not know where. So before starting, it has to build the IAT table by finding the address of the API it calls. When the executable is compiled, the IAT contains NULL memory pointers to each function. It will have the name of the function and what DLL it comes from, but the addresses are unknown. When the application begins execution, Windows finds the IAT location in the PE header, and overwrites NULL values with the correct memory location for each function.

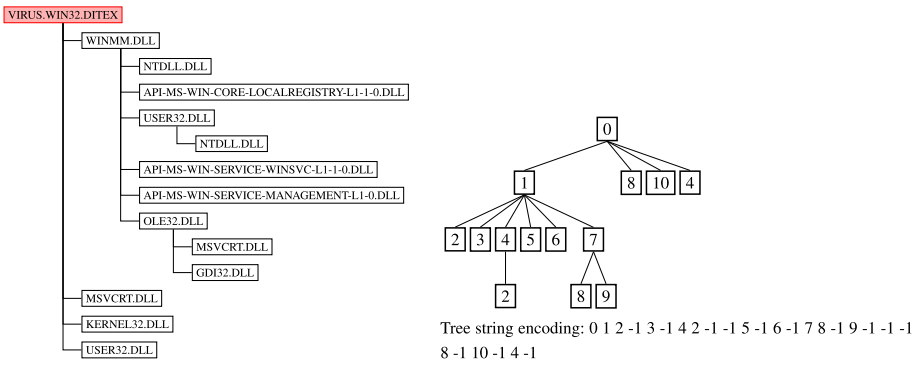
The procedure of Dependency Walker is outlined as follows.

Starting with the root module, Dependency Walker scans the module's import tables to build a list of dependent modules. It then scans each of these dependent modules for their dependent modules. This recursion continues until all modules and their dependent modules have been processed. In order to prevent possible infinite circular loops with dependent modules, Dependency Walker terminates processing of a given branch of the tree when it reaches a module that it has already been processed somewhere else in the tree. The obtained tree is very large and consists of 12 depths. These depths can be limited in order to reduce the computational cost. We will show that the depths of three and four is nearly enough to achieve high detection accuracy; because we are mining the structure of malware, which has been stabilized up to the depth of three and four, and further depths do not have a significant change in this structure. Afterwards, the tree's depth first traversal representation was extracted as shown in Figure 5.

4.2. Feature generation

In order to efficiently extract sub-trees, CMTreeMiner was used [41] because it is a computationally efficient algorithm for mining closed frequent trees. In this algorithm, based on depth-first canonical form, an enumeration tree is defined to enumerate all subtrees, and an enumeration directed acyclic graph is used for pruning branches of the enumeration tree that will not result in closed frequent subtrees Figure 2. By traversing this enumeration tree, CMTreeMiner discovers all closed frequent subtrees in the database. Furthermore, because the tree's string encoding is long, CMTreeMiner is very fast for mining such trees compared with other algorithms [42,43] that we tested. To use the extracted closed trees as features, every PE file can be represented as a binary vector of closed trees, namely, A, where $A_i = 1$ if and only if the PE file contains that closed tree, and $A_i = 0$ if corresponding PE file does not contain that tree.

[†] http://www.dependencywalker.com



(a) A part of Ditex virus dependency tree **(b)** Depth first representation

Figure 5. Construction of tree string encoding: (a) A part of Ditex virus dependency tree and (b) depth first representation.

5. EXPERIMENTAL RESULTS

In this section, we present the results of several experiments that were performed to illustrate the effectiveness of extracted features for detecting unseen malware. We conducted the following experiments:

1. Assessing the effectiveness of extracted features for malware detection,
2. Evaluating the frequent patterns as a signature for malware variant detection,
3. Evaluating junk injection evasion technique, and
4. Evaluating the method against packers.

5.1. Experimental data

For the malicious applications, we used a recently published malicious dataset.[‡] This dataset contains 11 000 very recent and active malicious applications that have been found in 500 drive-by download servers.

We also collected 309 portables of famous applications[§] from different categories such as accessibility (on-screen keyboard, magnifier, etc.), development (text editor, database browser, etc.), education(ear training, touch typing, etc.), game (puzzle, card, etc.), graphic and picture (image editor, animation creator, etc.), internet (browser, downloader, etc.), music and video (player, editor, etc.), office (word possessors, readers, etc.), security (anti-malware, rootkit remover, etc.), and utility (compressor, optimizer, etc.).

5.2. Detection results

To obtain an accurate estimation of the performance of the system, two experiments were performed. For the first experiment, we created a balanced dataset, consisting of the 309 portable executables along with 309 malicious applications. For the second experiment, we created an imbalanced huge dataset. Each of the portable executables consists of a lot of dependent executables as well as DLLs. This time we consider all of these files, resulting in 4700 benign files. By including all 11 000 malicious applications, we create a dataset of a total of 15 700 files.

On average, the maximum level of generated trees is 12, but we were able to achieve very good results by mining the trees up to depth 3 and depth 4. To evaluate the performance of the system reliably, 10-fold cross validation was applied using classifiers such as Naive Bayes, Random Forest, and J48, which are implemented in Weka [44]. In 10-fold cross validation, the dataset is divided into 10 subsamples with the same number of instances. Each time, 9 subsamples are used as training data and the remainder is used for testing. All of the experiments were performed on a 2 GHz Intel Core i7 PC with 8 GB of physical memory. The best results were obtained using Random Forest so we choose it as the main classifier for our system.

As we explained in section 3, the minimum support has to be identified by the user for extracting frequent patterns and it can be absolute or relative. We selected different relative minimum supports from 10% to 90%. For example, if we have 620 samples for evaluation, relative minimum support of 10% means that the frequent patterns must be in at least 62 samples. The results are shown in Tables II and III with various minimum supports. Number of patterns (No) refers to the total number of closed frequent subtrees (or number of single DLLs in the last row). DR or recall is defined as the portion of the total malicious PE files that are classified as malware, FP stands for the number of wrongly classified benign programs. Although we only considered DLL dependencies, the achieved results are significant. We achieved 100% DR with only 0.3% (one sample) FP rate in the small and balanced dataset. In the large and unbalanced dataset, we mined the trees until the sixth depth and achieved more than 98.5% DR and around 4% FP rate. The fair way to see whether the patterns

[‡] http://malicia-project.com/dataset.html
[§] http://www.portableapps.com/

Table II. Balanced dataset contains 309 malware and 309 benign programs.

	Depth 3			Depth 4		
Support (%)	No	DR (%)	FP (%)	No	DR (%)	FP (%)
10	186	**99.7**	**0**	410	99.4	0.3
20	185	99	0.3	400	99.4	0.3
30	183	99.4	0.3	383	99.4	0.3
40	183	99.4	0.3	383	99.4	0.3
50	158	99.4	0.3	353	99.4	0.3
60	51	99.4	0.3	104	**100**	**0.3**
70	32	95.1	0.6	58	98.7	0.6
80	22	83.5	0.6	48	95.1	0.6
90	17	73.8	0.3	33	94.8	0.3
Single DLL	185	98.7	0	209	98.7	0.3

DR, detection rate; FP, false positive; DLL, dynamic-link library.

Table III. Unbalanced dataset contains 11 000 malware and 4700 benign programs.

	Depth 3			Depth 4			Depth 5			Depth 6		
Support (%)	No	DR (%)	FP (%)	No	DR (%)	FP (%)	No	DR (%)	FP (%)	No	DR (%)	FP (%)
10	144	**98.5**	**4.1**	959	**98.7**	**4.8**	3645	98.6	4.7	17 482	98.6	5.1
20	85	98.3	4.3	380	98.5	4.7	621	98.5	4.6	1198	98.6	5.3
30	61	96.1	5.7	255	98.3	5.1	423	98.4	5.8	907	97.8	6.1
40	53	93.9	7.2	182	98	7.2	299	98.2	6	607	97.7	6.5
50	46	94.3	11.7	129	96.9	9	217	97.8	7.4	448	97.6	6.9
60	38	97.1	19.9	92	95.2	9.7	163	97.5	7.7	297	97.7	7.9
70	31	96.8	22.4	54	98.3	21.3	129	96.9	8.1	249	97	8.2
80	28	97.7	53.7	50	97.2	34.4	88	96.1	20.7	134	96.7	17.2
90	25	99.99	99.7	24	95.8	77.7	25	98.8	84.4	25	99.2	86.5
Single DLL	1202	97.6	6.6	1212	97.8	6.7	1219	97.7	7.8	1221	97.8	8.3

DR, detection rate; FP, false positive; DLL, dynamic-link library.

are appropriate features for detection is to compare the results with an approach that uses single DLLs as features. For that purpose, we consider all of the DLLs as features and to build a feature vector, assign one (if the program uses that DLL) or zero (if the program does not use that DLL).

5.3. Variant similarity

The purpose of this section is to present the effectiveness of the proposed method in detecting similar variants or obfuscated versions of a malware. A variant is usually a modification of an existing malware in order to elude anti-malware and cause infection. Based on malware naming convention, the complete name of a malware, which shows the hierarchical grouping of different classes of malware, is as follows:

Family_Name.Group_Name.Major_Variant.Minor_Varient

In order to compare the similarity between two PEs, their feature vectors are considered where each element shows the existence or non-existence of a frequent subtree in PE. These vectors are the following:

$$\text{Vector } PE_A = [X_1, X_2, X_3, X_4, \ldots, X_N]$$
$$\text{Vector } PE_B = [Y_1, Y_2, Y_3, Y_4, \ldots, Y_N]$$

Then, three similarity measures such as Cosine coefficient, Jaccard coefficient, and Pearson correlation [45] are used to compute the similarities of PEs. The formula for each measure is shown in Table IV. The similarity measures were computed for some pair of variants, and the results are shown in Table V. The results illustrate that frequent subtrees are a good representative for detecting a new variant of malware. Table V consists of three parts. The first part shows the comparison between minor variants of a major variant, and it indicates that most of them are absolutely similar to their minor variants, and others are correlated with a very high degree. WIN32.EVOL is an example of a true metamorphic virus that its new variant can be detected based on its previous minor variant. In the second part, the comparison is between two different major variants. The similarity degree in this section is neither very high to detect a minor variant nor very low to

Table IV. Similarity measures.

Name	Formula	Range	No	Perfect
Cosine (M1)	$\frac{(\sum_{i=1}^{n} x_i \times y_i)}{\sqrt{\sum_{i=1}^{n} x_i^2 \times \sum_{i=1}^{n} y_i^2}}$	0 . . . 1	0	1
Jaccard (M2)	$\frac{(\sum_{i=1}^{n} x_i \times y_i)}{\left(\sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n} y_i^2 - \sum_{i=1}^{n} x_i \times y_i\right)}$	0 . . . 1	0	1
Pearson (M3)	$\frac{\left(\sum_{i=1}^{n} x_i \times y_i - \frac{\left(\sum_{i=1}^{n} x_i \times \sum_{i=1}^{n} y_i\right)}{n}\right)}{\sqrt{\left(\sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}\right)\left(\sum_{i=1}^{n} y_i^2 - \frac{\left(\sum_{i=1}^{n} y_i\right)^2}{n}\right)}}$	–1 . . . 0 . . . 1	0	1

Table V. Variant similarity results.

First PE	Second PE	M1	M2	M3
VIRUS.WIN32.EVOL.A	VIRUS.WIN32.EVOL.B	1	1	1
VIRUS.WIN32.EVUL.8192.A	VIRUS.WIN32.EVUL.8192.B	1	1	1
NET-WORM.WIN32.VESSER.A	NET-WORM.WIN32.VESSER.B	1	1	1
VIRUS.WIN32.EVYL.A	VIRUS.WIN32.EVYL.H	1	1	1
VIRUS.WIN32.MAGIC.7045.B	VIRUS.WIN32.MAGIC.7045.H	1	1	1
VIRUS.WIN32.RAMM.A	VIRUS.WIN32.RAMM.E	1	1	1
VIRUS.WIN32.THORIN.B	VIRUS.WIN32.THORIN.E	1	1	1
VIRUS.WIN32.MIAM.1727	VIRUS.WIN32.MIAM.4716	1	1	1
VIRUS.WIN32.SAVIOR.1696	VIRUS.WIN32.SAVIOR.1832	1	1	1
NET-WORM.WIN32.WELCHIA.A	NET-WORM.WIN32.WELCHIA.B	0.9541	0.9104	0.9401
VIRUS.WIN32.INRAR.A	VIRUS.WIN32.INRAR.E	0.9177	0.8421	0.9058
VIRUS.WIN32.DRIVALON.1876	VIRUS.WIN32.MAGIC.3038	0.8572	0.7500	0.8381
VIRUS.WIN32.RAMM.A	VIRUS.WIN32.SAVIOR.1832	0.8255	0.7027	0.8021
VIRUS.WIN32.BLATEROZ	VIRUS.WIN32.CHAMP	0.8291	0.6875	0.8120
VIRUS.WIN32.OROCH.5420	VIRUS.WIN32.PARADISE.2116	0.8291	0.6875	0.8120
BOOTCFG	VIRUS.WIN32.ZOMBIE	0.6370	0.4133	0.5701
DIALER	NET-WORM.WIN32.DOMWOOT.C	0.6170	0.3870	0.5303
check Disk	VIRUS.WIN32.CECILE	0.5979	0.3647	0.5198
Calculator	VIRUS.WIN32.EVOL.C	0.3853	0.2058	0.3439

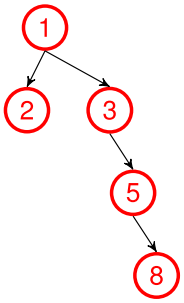
PE, portable executable.

ignore it as a malware so it is good enough to show the relationship between the two major variants. The third part is a comparison between malware and benign PEs. The low score shows that the benign samples are not similar with the malware samples. The separation of three parts depends on a predefined threshold. This threshold can be obtained by evaluating similarities between a large volume of various malware. Therefore, by defining a proper threshold, variants can be detected even without using an learning-based model and significant processing.

There are some approaches [46,47] that considered similarity measures for malware detection by defining thresholds, however, we do not aim to propose another detection method based on similarity. Our main purpose in this paper is to propose a new learning-based malware detection system, and these evaluations are considered only as a validation of the performance of the system.

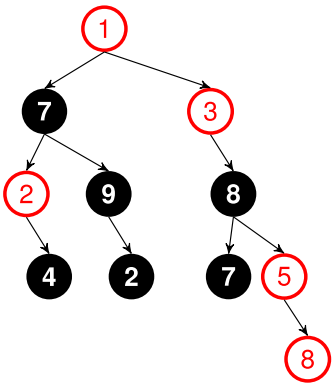
5.4. Fake dynamic-link library injection

One of the problems concerning signature-based methods is junk data insertion. Junk data insertion is a common polymorphic method that uses ineffectual instructions like NOP to modify the signature. In a similar way, malware writers can use fake system calls, without affecting the malware essence, to confuse anti-malware software to mismatch the patterns in the new malware. Inserting junk APIs causes new DLLs to be added to the behavioral tree. In order to prevent this, the Longest Common Subsequence (LCS) [48] algorithm is used for finding the patterns in a new malware. LCS is used to find the longest subsequence that is common in usually two sequences. The LCS is not necessarily unique (e.g., the LCS of "ABC" and "ACB" is both "AB" and "AC"), and the LCS problem is often defined to find all LCS. As is shown in Figure 6, we assume that the left subtrees are benign and malicious patterns that we want to match, and the right tree is the behavioral tree of a new malware. Using the LCS algorithm, the malicious patterns are successfully matched without considering the fake elements between the patterns' elements. In order to achieve a better understanding of robustness in real data, we inject 200 different DLLs into 50 test malware, and make evaluations using both the proposed method as well as single DLL method on a small dataset, which contains



String encoding: 1 2 -1 3 5 8 -1 -1 -1

(a) A malicious pattern



String encoding: 1 7 2 4 -1 -1 9 2 -1 -1 -1 3 8 7 -1 5 8 -1 -1 -1 -1

(b) Malware dependency tree with injected fake DLL

Figure 6. Fake Application Programming Interface injection: (a) A malicious pattern and (b) malware dependency tree with injected fake dynamic-link library.

309 benign programs and 309 malware. The detection rate of both methods on 50 samples are 100% before injection but after injection, the detection rate of single DLL method becomes 0% but the proposed method retains the 100% detection rate. This evaluations show that the proposed method is robust against injecting junk data.

5.5. Packers

The main problem of static analysis is PE packing, which changes the structure of PE without affecting behavior. Anti-viruses usually unpack executable files before analyzing them and probing for signatures, but they usually fail if a new packer comes with its own method. An outstanding point of our method is that we gain the same accuracy toward many common packers without unpacking them. We packed malware with common packers such as UPX, ASPACK, Exe32pack, Exepacker, and Petit and compared their dependency tree with the original version. Figure 7 shows the effect of different packers on the behavioral tree of Win32.Mogul6845 virus on depth 2. Packers mirror the resource trees without changing the DLL dependencies. Although some packers like Themida eliminate some

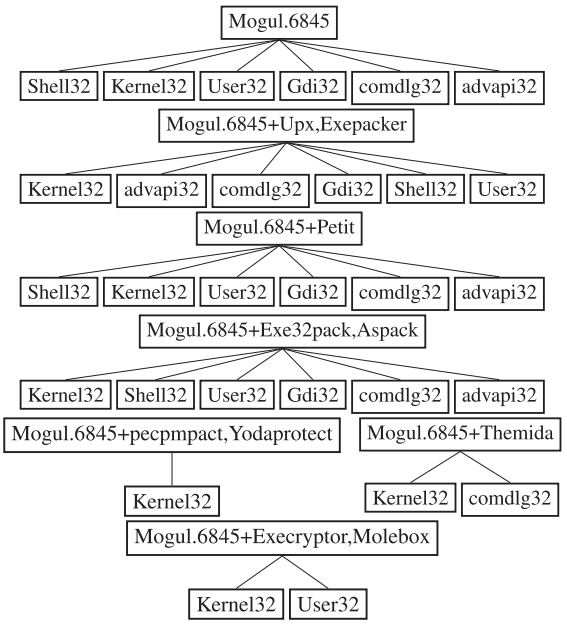


Figure 7. Packers' influence on behavioral tree of Virus.Win32.Mogul6845.

parts of the tree, most of the packers only change the order of subtrees rooted at depth 2. Each sub-tree is a specific behavior, and the packer only replaces the order of behaviors. This is another reason for considering subtrees as featured in this method. Both malware and its packed versions contain similar features, so their feature vectors are similar and the classification can return the same result. In order to see the robustness of our method against packing, we do the same experiments we did in section 5.4 and again achieve 100% after packing the 50 malware samples by UPX, ASPACK, Exe32pack, Exepacker, and Petit. As a result, packers cannot affect behavioral aspects of the dependency tree and so they cannot affect the proposed system significantly.

There are also some other packers which remove the IAT. This problem can be solved by tools such as ImpREC,[¶] which try to recover the IAT during runtime.

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed a new approach toward malware detection. We extracted the malware's behavioral tree, and, in order to find the unique behaviors of malware, mined closed frequent subtrees. Our proposed system can be defined as a hybrid method because we could extract coarse-grain behaviors without actually executing the malware. To assess the performance of the system, we performed various experiments. We gained 100% DR

[¶] http://www.woodmann.com/collaborative/tools/index.php/ImpREC

and 0.03% FP rate in a small balanced dataset and more than 98.5% in a large unbalanced dataset. We also showed that the proposed features can be an effective signature for detecting new and unseen malware variants without any significant processing. Finally, we evaluated some evasion techniques against the system. In the future, we would like to focus on some packers that remove the IAT of PE and also verify the robustness of frequent patterns against some other attacks like mimicry attacks. We are also interested in pruning the tree by removing unimportant substructures from the tree.

REFERENCES

1. F-secure. F-secure reports amount of malware grew by 100% during 2007, 2007. Available from: http://www.businesswire.com/news/home/20071204005453/en/F-Secure-Report s-Amount-Malware-Grew-100-2007#.VEDYtYuUfA4 [Accessed on 2012].
2. Kaspersky. Q1/2011 malware report, 2011. Available from: http://www.kaspersky.com/downloads/pdf/kaspersky_lab_q1_malware_2011_report.pdf [Accessed on 2012].
3. Symantec. Internet security threat report, 2011. Available from: https://www4.symantec.com/mktginfo/downloads/21182883_GA_REPORT_ISTR_Main-Report_04-11_HI-RES.pdf [Accessed on 2012].
4. Idika N, Mathur AP. A survey of malware detection techniques. Department of Computer Science, Purdue University, 2007. Available from: http://cyberunited.com/wp-content/uploads/2013/03/A-Survey-of-Malware-Detection-Techniques.pdf [Accessed on 2013].
5. Egele M, Scholte T, Kirda E, Kruegel C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys* 2008; **44**(2): 6:1–6:42.
6. Moser A, Kruegel C, Kirda E. Limits of static analysis for malware detection. *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, Miami Beach, FL, December 2007; 421–430.
7. Sami A, Yadegari B, Rahimi H, Peiravian N, Hashemi S, Hamze A. Malware detection based on mining api calls. *Proceedings of the 2010 ACM Symposium on Applied Computing*, New York, NY, USA, 2010; 1020–1025.
8. Ye Y, Wang D, Li T, Ye D, Jiang Q. An intelligent pe-malware detection system based on association mining. *Journal in Computer Virology* 2008; **4** (4): 323–334.
9. Christodorescu M, Jha S. Static analysis of executables to detect malicious patterns. *Proceedings of the 12th Conference on USENIX Security Symposium-Volume 12*, Berkeley, CA, USA, 2003; 12–12. Available from: http://dl.acm.org/citation.cfm?id=1251353.1251365 [Accessed on 2011].
10. Rieck K, Trinius P, Willems C, Holz T. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security* 2011; **19** (4): 639–668. Available from: http://dl.acm.org/citation.cfm?id=2011216.2011217 [Accessed on 2014].
11. Ahmadi M, Sami A, Rahimi H, Yadegari B. Malware detection by behavioural sequential patterns. *Computer Fraud & Security* 2013; **2013** (8): 11–19. Available from: http://www.sciencedirect.com/science/article/pii/S1361372313700721 [Accessed on 2014].
12. Ye Y, Li T, Chen Y, Jiang Q. Automatic malware categorization using cluster ensemble. *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, USA, 2010; 95–104.
13. Chen H, Wagner D. Mops: an infrastructure for examining security properties of software. *Proceedings of the 9th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2002; 235–244.
14. Chen H, Dean D, Wagner D. Model checking one million lines of c code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, California, 2004; 171–185.
15. Feng H, Giffin J, Huang Y, Jha S, Lee W, Miller B. Formalizing sensitivity in static analysis for intrusion detection. *Proceedings. 2004 IEEE Symposium on Security and Privacy, 2004*, Oakland, California, May 2004; 194–208.
16. Griffin K, Schneider S, Hu X, Chiueh TC. Automatic generation of string signatures for malware detection. *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, Berlin, Heidelberg, 2009; 101–120.
17. Ida : Disassembler and debugger, 2013. Available from: https://www.hex-rays.com/products/ida/ [Accessed on 2014].
18. Santos I, Brezo F, Ugarte-Pedrero X, Bringas PG. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences* 2013; **231** (0): 64–82. Available from: http://www.sciencedirect.com/science/article/pii/S0020025511004336 [Accessed on 2014], Data Mining for Information Security.
19. Tabish SM, Shafiq MZ, Farooq M. Malware detection using statistical analysis of byte-level file content. *Proceedings of the ACM SIGKDD Workshop on Cybersecurity and Intelligence Informatics*, New York, NY, USA, 2009; 23–31.

20. Novel active learning methods for enhanced {PC} malware detection in windows {OS}. *Expert Systems with Applications* 2014; **41** (13): 5843–5857. Available from: http://www.sciencedirect.com/science/article/pii/S095741741400133X [Accessed on 2014].
21. Sung AH, Xu J, Chavez P, Mukkamala S. Static analyzer of vicious executables (save). *Proceedings of the 20th Annual Computer Security Applications Conference*, Washington, DC, USA, 2004; 326–334.
22. Kumar S, Spafford E. A generic virus scanner for c++. *Computer Security Applications Conference, 1992. Proceedings., Eighth Annual*, San Antonio, TX, November 1992; 210–219.
23. Shafiq M, Tabish S, Mirza F, Farooq M. Pe-miner: mining structural information to detect malicious executables in realtime. In *Recent Advances in Intrusion Detection*, vol. 5758, Kirda E, Jha S, Balzarotti D (eds), Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2009; 121–141.
24. Wang TY, Wu CH, Hsieh CC. A virus prevention model based on static analysis and data mining methods. *IEEE 8th International Conference on Computer and Information Technology Workshops, 2008. CIT Workshops 2008*, Sydney, QLD, July 2008; 288–293.
25. Willems C, Holz T, Freiling F. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy* 2007; **5**(2): 32–39.
26. Rieck K, Holz T, Willems C, Dussel P, Laskov P. Learning and classification of malware behavior. *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Berlin, Heidelberg, 2008; 108–125.
27. Dai J, Guha R, Lee J. Efficient virus detection using dynamic instruction sequences. *Journal of Computers* 2009; **4** (5). Available from: http://ojs.academypublisher.com/index.php/jcp/article/view/0405405414 [Accessed on 2011].
28. Kolbitsch C, Comparetti PM, Kruegel C, Kirda E, Zhou X, Wang X. Effective and efficient malware detection at the end host. *Proceedings of the 18th Conference on USENIX Security Symposium*, Berkeley, CA, USA, 2009; 351–366. Available from: http://dl.acm.org/citation.cfm?id=1855768.1855790 [Accessed on 2012].
29. Kim K, Moon BR. Malware detection based on dependency graph using hybrid genetic algorithm. *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, New York, NY, USA, 2010; 1211–1218.
30. Lanzi A, Balzarotti D, Kruegel C, Christodorescu M, Kirda E. Accessminer: using system-centric models for malware protection. *Proceedings of the 17th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2010; 399–412.
31. Nappa A, Rafique M, Caballero J. Driving in the cloud: an analysis of drive-by download operations and abuse reporting. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 7967, Rieck K, Stewin P, Seifert JP (eds), Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2013; 1–20.
32. Han J, Cheng H, Xin D, Yan X. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery* 2007; **15**(1): 55–86.
33. Di Cerbo F, Girardello A, Michahelles F, Voronkova S. Detection of malicious applications on android os. *Proceedings of the 4th International Conference on Computational Forensics*, Berlin, Heidelberg, 2011; 138–149. Available from: http://dl.acm.org/citation.cfm?id=1991416.1991433 [Accessed on 2014].
34. Yang C, Xu Z, Gu G, Yegneswaran V, Porras P. Droidminer: automated mining and characterization of fine-grained malicious behaviors in android applications. *19th European Symposium on Research in Computer Security*, Wroclaw, Poland, 7–11 September 2014; 163–182.
35. Fredrikson M, Jha S, Christodorescu M, Sailer R, Yan X. Synthesizing near-optimal malware specifications from suspicious behaviors. *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2010; 45–60.
36. Karbalaie F, Sami A, Ahmadi M. Semantic malware detection by deploying graph mining. *International Journal of Computer Science Issues* 2012; **9**(1).
37. Perdisci R, Ariu D, Giacinto G. Scalable fine-grained behavioral clustering of http-based malware. *Computer Networks* 2013; **57** (2): 487–500. Available from: http://www.sciencedirect.com/science/article/pii/S1389128612002678 [Accessed on 2014], Botnet Activity: Analysis, Detection and Shutdown.
38. Palahan S, Babić D, Chaudhuri S, Kifer D. Extraction of statistically significant malware behaviors. *Proceedings of the 29th Annual Computer Security Applications Conference*, New York, NY, USA, 2013; 69–78.
39. Shahzad F, Farooq M. Elf-miner: using structural knowledge and data mining methods to detect new (Linux) malicious executables. *Knowledge and Information Systems* 2012-03; **30**(3): 589–612.
40. Dependency walker : disassembler and debugger, 2013. Available from: http://www.dependencywalker.com/ [Accessed on 2013].
41. Chi Y, Yang Y, Xia Y, Muntz R. Cmtreeminer: mining both closed and maximal frequent subtrees. In *Advances in Knowledge Discovery and data Mining*, vol. 3056, Dai H, Srikant R, Zhang C (eds),

Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2004; 63–73.
42. Zaki MJ. Efficiently mining frequent trees in a forest: algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering* 2005; **17** (8): 1021–1035, special issue on Mining Biological Data.
43. Asai T, Abe K, Kawasoe S, Sakamoto H, Arikawa S. Efficient substructure discovery from large semi-structured data, 2002; 158–174.
44. Weka : data mining software in java, 2013. Available from: http://www.cs.waikato.ac.nz/ml/weka/ [Accessed on 2013].
45. Lucia L, Lo D, Jiang L, Budi A. Comprehensive evaluation of association measures for fault localization, *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, Washington, DC, USA, 2010; 1–10.
46. Walenstein A, Venable M, Hayes M, Thompson C, Lakhotia A. A.: Exploiting similarity between variants to defeat malware: vilo? method for comparingand searching binary programs. *In: Proceedings of BlackHat DC 2007*, Las Vegas, 2007. Available from: https: //blackhat.com/presentations/bh-dc-07/Walenstein/paper/bh-dc-07-walenstein-WP.pdf [Accessed on 2014].
47. Cesare S, Xiang Y. Malware variant detection using similarity search over sets of control flow graphs. *2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM)*, Changsha, China, November 2011; 181–189.
48. Wikipedia. Longest common subsequence problem, 2013. Available from: http://en.wikipedia.org/wiki/Longest_common_subsequence_problem [Accessed on 2013].