International Journal of Security and Its Applications
Vol.7, No.5 (2013), pp.29-42
http://dx.doi.org/10.14257/ijsia.2013.7.5.03

Improving the Detection of Malware Behaviour Using Simplified Data Dependent API Call Graph

Ammar Ahmed E. Elhadi[1,2], Mohd Aizaini Maarof[1] and Bazara I. A. Barry[3]

[1]*Information Assurance and Security Research Group*
[1]*Faculty of Computing*
[1]*Universiti Teknologi Malaysia*
[2]*Elmashreq College for Science and technology*
[3]*Faculty of Mathematical Sciences - University of Khartoum*
ammareltayeb@gmail.com, aizaini@utm.my, bazara.barry@gmail.com

Abstract

Malware stands for malicious software. It is software that is designed with a harmful intent. A malware detector is a system that attempts to identify malware using Application Programming Interface (API) call graph technique and/or other techniques. Matching the API call graph using graph matching algorithm have NP-complete problem and is slow because of computational complexity .In this study, a malware detection system based on API call graph is proposed. Each malware sample is represented as data dependent API call graph. After transforming the input sample into a simplified data dependent graph, graph matching algorithm is used to calculate similarity between the input sample and malware API call graph samples stored in a database. The graph matching algorithm is based on Longest Common Subsequence (LCS) algorithm which is used on the simplified graphs. Such strategy reduces the computation complexity by selecting paths with the same edge label in the API call graph. Experimental results on 85 samples demonstrate 98% detection rate and 0% false positive rate for the proposed malware detection system.

Keywords: *Malware; Malware Detection; API call graph; API call graph matching; Longest common subsequence*

1. Introduction

Malicious software, or malware, refers to programs that intentionally exploit vulnerabilities in computing systems for a harmful purpose. Malware can be differentiated between based on whether the software needs or does not need a host program to function. Another way of categorizing malware is by identifying if the software produces copies of itself or not. In the following, some of the key categories of malicious software are briefly surveyed [1]:

- Virus: a piece of software that when executed, tries to replicate itself into other executable code. If the virus succeeds, it infects the code, which causes the virus to execute whenever the code executes.
- Trojan horse: a computer program that has an apparent benign and useful function, but also a hidden and potentially malicious function that avoids system security measures.
- Worm: a computer program that self-replicates and disseminates versions of itself across a network;

ISSN: 1738-9976 IJSIA

- Backdoor: a secret entry point into a program that allows someone who is aware of its existence to gain unauthorized access to the system without going through the normal security check.
- Spyware: software that gathers information from victim computer and sends it to spyware creator.
- Rootkit: a collection of tools developed and used by attackers when a computer system is broken into and root-level access is gained.
- Botnet: a computer program that is activated by a certain trigger on a machine to attack and infect other machines

Malware writers often use various techniques to modify or morph existing malware into new polymorphic versions to evade detection. The availability of advanced toolkits has made it easier for malware writers to use methods such as dead-code insertion and register reassignment to perform this modification. Generally, malware modification or obfuscation can be classified into polymorphism and metamorphism [2].

A malware detector is a computer program that attempts to detect and identify malware using a variety of techniques that include recognizing malware signature, utilizing heuristic rules, and identifying malware behaviour or actions. Malware detectors can operate locally on the system that is being protected or provide protection remotely through a computer network. Usually, two types of input are needed by malware detectors, namely, knowledge of the malware signature or behaviour which can be gained through a learning process and the program under inspection. Once the two inputs become available, the malware detector employs its detection techniques to determine whether the program is malware or benign [1].

Nowadays, millions of samples of potentially harmful executables are submitted regularly for analysis to data security companies which are faced with the problem of discerning whether the samples are malware or not. For example, more than 286 million different variants of malware have been faced by Symantec in 2010 [3]. Attackers generate new malware samples from old ones using code obfuscation, polymorphism, and new delivery mechanisms such as web-attack toolkits, which greatly contributes to the significant increase in the number of malware variants being distributed [3]. Moreover, there is a tendency among malware writers to use high-level programming languages to write malware and compile it into binary afterward, which adds more complexity to the existing problem and demonstrates the need for effective and efficient solutions.

A potential solution is API call graph which is a high-level structure that abstracts instruction level details and is thus more resilient to representing the code obfuscations commonly employed by malware writers or malware development tools [3]. It is based on the idea of the call graph which is a useful data representation of the control and data flow of programs, and it investigates interprocedural communication (i.e., how procedures exchange information). In addition to investigating the relationship between procedures in a program, call graphs can be used to provide information regarding local data within each procedure and global data that are shared among procedures [4]. In such structure, relationships among program procedures are represented by a directed graph that contains nodes and edges. A node in a call graph represents a procedure in the program whereas a directed edge (u, v) indicates that procedure v is called by procedure u. Call graphs are a basic program analysis tool that can either be used to better understand programs by humans or as a basis for further analysis, such as an analysis that tracks the flow of values between procedures and interprocedural program optimization [5]. They can also be used to find procedures that are never called.

The formal definition of a call graph is provided in terms of a directed graph $G = (V, E)$ as follows:

- V=V(G) is the set of graph vertices that represents the functions in the program.
- E=E(G) is the set of graph edges that represent the function calls in the program, and $E \subseteq V \times V$ [6].

According to [7], an Application Programming Interface (API) is a collection of routines, specifications, and tools that enable software programs to interact with each other. Applications, libraries, and operating systems can benefit from APIs to define vocabularies and resource request conventions, and to provide specifications for the interaction between the consumer program and the implementer program of the API.

The API calls list is extracted from a binary executable through static analysis of the binary with disassembly tools such as IDA Pro [8] or through dynamic analysis after executing the binary in a simulated environment, which is the technique adopted by tools such as API monitor [9]. Although the real API calls can be determined using dynamic analysis, the malware sample must be executed many times to get all the different execution paths. To analyze an executable, obfuscation layers are removed first and unpacking followed by decryption are applied to the executable. Next, functions are identified and symbolic names are assigned to them.

In order to use call graphs to detect malware, it is required to compare call graphs to differentiate between the ones that represent benign programs and those that are based on malware samples. To compare call graphs, a graph matching algorithm is used. To match call graphs, graphs can be classified into model and data graphs, and a model graph needs to be matched with a data graph to find their similarity. Matching can be classified into two categories, namely, exact matching and inexact matching. Exact graph matching applies when the two graphs has the same number of vertices, whereas in inexact graph matching the two graphs has different number of vertices [10].

Graph matching algorithms have NP-Complete problem (*i.e.*, problem that can be solved in polynomial time on a nondeterministic Turing machine) and suffer from slowness because of their computational complexity. Furthermore, many existing graph-similarity query processing methods cannot scale to large graphs. The work proposed in this study uses dynamic analysis to extract API calls list from the provided binary. Furthermore, it applies a Longest Common Subsequence (LCS) matching algorithm [11] by mapping a path from the query graph to a path in the data graph where both paths have the same edge labels for all edges in the path.

This study is organized as follows. Section 2 describes the previous studies related to malware detection and malware detection based on call graph. Section 3 describes in detail the proposed malware detection system paying special attention to the enhancements in matching algorithm. Section 4 discusses the proposed system experimental setup alongside the evaluation metrics and analyzes the obtained results. Section 5 concludes the study.

2. Related Work

Malware Detection can be achieved through one of two techniques, namely, signature-based and behaviour-based techniques. Each of these techniques can be applied using static analysis, dynamic analysis, or hybrid analysis [12]. Figure 1 shows the organization and classification of malware detection techniques.

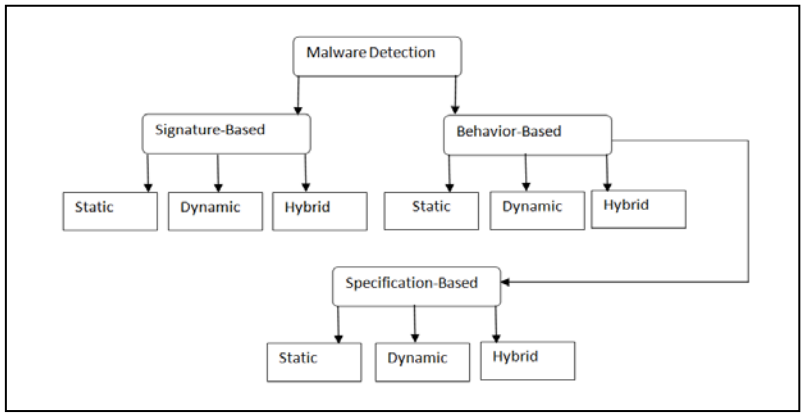

Figure 1. Organization of Malware Detection

Implementing signature based detection without executing the suspected file (*i.e.*, static analysis) was the first attempt to detect malware. Some of the researchers who used this approach applied Objective-Oriented Association (OOA) mining based classification [13]. Their model has three major modules, namely, Portable Executable (PE) parser, OOA rule generator, and rule based classifier. This model was taken to a next level by adopting associative classification method based on the analysis of application programming interface (API) execution calls [14]. Other researchers combined signature-based technique and genetic algorithm technique, but their study focused on three types of malware which are virus, worm and Trojan horse [15].

Signature based detection was also applied during suspected file execution (*i.e.*, dynamic analysis) in which the researchers traced API calls and then built the suspected file signature [16]. These researchers generated signature for an entire malware class instead of individual malware samples. This approach has the advantage that all the metamorphic viruses that are created from a metamorphic generator can be easily detected once a base signature for that metamorphic generator is obtained.

A considerable portion of the existing studies relies on using behaviour based detection, where some researchers apply static analysis while others apply dynamic analysis. Some of the studies adopt the mapping of kernel memory as a way to develop a monitor for malware behaviour. The monitor utilizes time-based view of kernel objects to analyze traces of kernel execution [17]. Other studies trace malware behaviour exhibited by installer and uninstaller software as a way to avoid false positives [18], and suggest a new categorization method for malware based on maximal common subgraph detection[19].

Current researches combine static analysis with dynamic analysis to overcome the limitations of each method. Such strategy is used by Guo [20] in their proposed framework that aims at detecting malware and preventing its execution by combining static and dynamic binary translation features. In that study, behaviour Control Flow Graph (CFG) is used and based on it critical API graph is generated to perform sub-graph matching. Other studies use signature CFG and compare graphs based on edit distance matching. CFG represents control flow dependencies in a program; it is a graph in which the nodes are basic blocks with a sequence of consecutive statements and edges represent possible control flows from one basic block to another.

Limitations can be found in both signature-based and behaviour-based approaches. Signature-based approaches can be overcome by obfuscation and require prior knowledge of malware samples. Behaviour-based approaches generate higher rates of false positives and incur expensive runtime overhead [3, 21].

As a result, malware detection research has recently witnessed a shift towards the use of API call graphs because they have sufficient expressiveness to model complicated structures, and their use is gaining momentum in representing structural information. The following paragraphs review some of the related work in this area.

To build the graph, most researchers present graph nodes as system calls. For example, Lee [22] create their graph by transforming a Portable Executable (PE) file into a call graph with nodes and edges which represent system calls and system call sequence, respectively. After that, minimization is applied to the call graph turning it into a code graph to speed up the analysis and comparison process. Other researchers use the same approach by using 4-tuple nodes to denote a system call, edges, the dependencies between two system calls and a label for nodes and edges [19]. Some other studies use graph nodes to denote kernel objects instead of system calls [23]. On the other hand, Kostakis [24] build a graph from subroutines as nodes and their call references as edges, whereas Kim and Moon [25] use a dependency graph whose vertex represents a line in the semantic code and the dependency between two lines is represented by a directed edge. Finally, in [20, 26], a Critical API Graph (CAG) is extracted from a CFG for each malware to define its behaviour.

The above studies compare graphs using different graph matching techniques such as formula building using intersection and union of graphs, weighted common behavioural graph generation based on an approximate algorithm [22], and maximal common subgraph [19, 25]. However, due to NP- completeness of the problem and the computational complexity inherent in such API call graph matching algorithms [10], they are prohibitively expensive to use for large graphs. This study proposes a system to solve the problem of malware detection based on API call graph achieving better detection rate and performance levels.

3. Proposed Malware Detection Technique

The proposed detection system in this study attempts to enhance malware detection using API call graph. The system is based on the idea that most malware samples are generated from previous existing samples. To improve malware detection, the proposed system takes advantage of data dependent schemes to construct API call graph. In data dependent API call graphs, API calls are call graph nodes, and the edges between nodes are based on data dependence between the API calls.

Graph matching and graph similarity are used to trace common subgraphs into the data dependent API call graph, and the graph matching is based on Longest Common Subsequence (LCS) algorithm. The general framework of the proposed system is outlined in Figure 2.

The proposed mechanism is composed of several phases. First, the framework enters a pre-processing phase which includes malware unpacking and extracting the API calls from input which is portable executable (PE) samples. The extraction is done using API call monitoring tool under secure environment so the API call and its parameters are recorded. In the second phase, the data dependent API call graph is constructed from the recorded list. This phase is also used to update malware API call graph database that is to be used by matching algorithm. Next, matching the data dependent API call graph takes place and the system calculates the similarity and identifies whether the input sample is malware or not. In the following sub-sections, extraction of data dependent API call graph and longest common subsequence matching algorithm are discussed in detail

3.1. Extracting Data Dependent API Call Graph

Malware detection starts by extracting and collecting the API call list along with its parameters, especially operating system resources such as files, registers, network resources,

processes, memory, *etc*. The algorithm is based on one type of node, namely API calls, and has one type of edge, a Data Dependent (DD) edge which connects an API call to another. This phase identifies the nodes and the edges for each line of instruction. First, all API call nodes are defined. Then, for each two nodes the data dependent edges are defined using the parameter list of the two API calls. For example, part of the Netsky program is shown in Figure 3(a) and a corresponding data dependent API call graph is shown in Figure 3(b).

To construct the DD API call graph, each line of instruction consists of an API call along with its parameters list. The API call will be a graph node. For parameters list, if one of the parameters in the list appears in previous API call parameters list, it mean this API call uses a parameter that has been used or defined by pervious API call, and hence there will be data dependence between the new node and previous API call node. Consequently, a data dependent edge between the two nodes will be created with the parameter name as its label.

The Data Dependent API Call Graph can be defined as $G = (V, E, V_G, E_G, \sum)$, where V is a set of nodes of G; $E \subseteq V \times V$ is a set of edges of G; and $V_G: V \rightarrow \sum$ is a function that maps nodes to labels (*e.g.*, NtCreateFile) in the alphabet set $\sum$; and $E_G: E \rightarrow \sum$ is a function that maps edges to labels (*e.g.*, FileHandle A) in the alphabet set $\sum$.

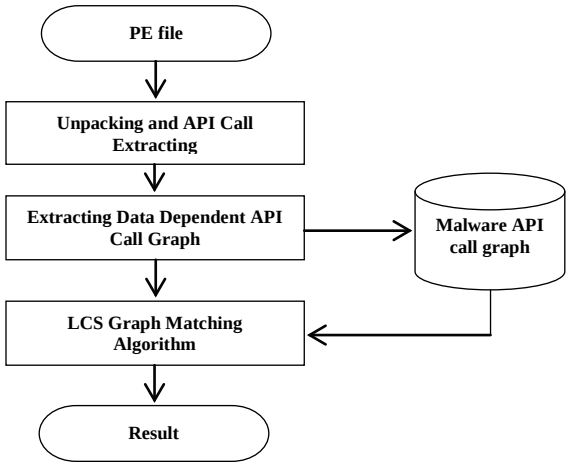

Figure 2. The General Framework of the Proposed System

1. NtCreateFile (Out FileHandle A, □.. , ObjectAttributes -> C:\Netsky.exe)
2. ……..
3. NtCreateFile (Out FileHandle B, □. , ObjectAttributes -> C:\WINDOWS\AVprotect9x.exe)
4. …….
5. NtCreatSection (In FileHandle A, □□Out SectionHandle C, ObjectAtrribtue-> Section_XXX)
6.NtMapViewOfSection (In SectionHandle C, □.)
7. …...
8. NtWriteFile (In FileHandle B, □.)
9. …...

(a)

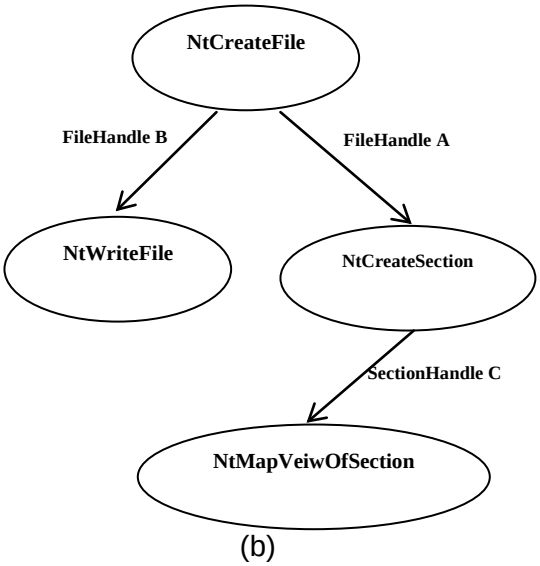

Figure 3. Figure 3.A: Part of API Call Trace for the Netsky Program Figure 3.B: A Corresponding Data Dependent API Call Graph

3.2. Longest Common Subsequence Graph Matching Algorithm

The complexity of the graph matching approaches makes it very difficult to take into account all types of dependencies between the vertices and edges in the query and data graphs. One way to overcome this difficulty is to simplify the graph so that the complexity of graph matching is reduced. Since malware variants are generated from a previously seen one, and by representing malware samples in a Data Dependent API call graph, the problem of identifying malware variants is reduced to finding the best matching subgraph. The problem of graph matching is to compute a subgraph of the data graph that best matches the query graph.

Longest common subsequence (LCS) algorithm [11] is used to measure the similarity over two strings by finding the longest subsequence common for all possible prefix combinations of the input strings. When matching data dependent API call graphs using LCS algorithm, the graph needs to be converted into one sequence forming a string. Due to the complexity of data dependent API call graph, the process of converting the API call graph into a string needs to be optimized before applying LCS algorithm.

To match a query data dependent API call graph with data one, the proposed algorithm allows mapping of a path (*i.e.*, a path or an edge) from the query graph with a path (*i.e.*, a path or an edge) in the data graph using LCS algorithm. These two paths must have the same labels for all edges in the path. The proposed algorithm, as shown in Figure 4, starts by selecting a path from both graphs (query, data) with the same edge label. Step 2 then calculates similarity between the two paths using LCS algorithm. Starting at step 3, the similarity between the two graphs is calculated by taking all paths similarity and dividing that by the number of all paths.

Algorithm: Finding the best subgraph with high $sim(Q,G)$
Input: Query graph (Q) and Data graph (G)
Output: $\mathbf{sim(Q,G)}$
1: $\mathbf{sim = 0}$
2: for each path $\mathbf{P_1' \in Q}$ and $\mathbf{P_2' \in G}$, where $\mathbf{P_1'}$ and $\mathbf{P_2'}$ have the same label for all edges in the path.
3: // find similarity for the two paths using LCS
4: $\mathbf{sim = sim + simLCS(P_1', P_2')}$
5: $\mathbf{sim(Q,G) = \frac{sim}{|P|}}$ where $\mathbf{|P|}$ is the number of paths in Q
6: return $\mathbf{sim(Q,G)}$

Figure 4. The Proposed Algorithm to Find Similarity between Two Subgraphs

The running time of the algorithm is related to LCS algorithm [11] which is $\mathbf{O(N \log N + D^2)}$, where N is the sum of the lengths of $\mathbf{(P_1', P_2')}$ and D is the size of the minimum edit for $\mathbf{(P_1', P_2')}$. In the proposed algorithm, paths were selected from query graph $\mathbf{P}$ to match paths in data graph to simplify the graph, and therefore, the proposed algorithm running time is $\mathbf{O(N \log P + D^2)}$ which is less than the running time of LCS algorithm.

4. Evaluation

This section describes the conduction of a set of experiments on real malware files to evaluate the detection rate and the accuracy of the proposed mechanism. Since there is no standard benchmark for comparison [27], and malware detection approaches that are proposed by scientific research are tested using their own malware datasets trying different assessment methods, methods like LCS and N-gram has been implemented in this study and used alongside the proposed mechanism on the testing dataset.

4.1. Experimental Setup

The malware dataset has been downloaded from http://www.nexginrc.org/ web site. The malware samples that are used as inputs are pre-processed using the API monitor tool to extract API calls and their parameters (OSRs). The dataset consists of 75 malware and 10 benign programs; these malware samples are obtained from a publicly-available database called 'VX Heavens Virus Collection' [28]. Table 1 gives the basic statistics about the dataset of text files used in this study.

All experiments are conducted on an AMD Phenom™ II X4 machine with 3.25 GHz processor, 4.00 GB of memory, and Microsoft Windows 7 as the operating system. A prototype of the proposed system has been implemented using Delphi programming language. In the experiment, any sample has been tested and its similarity with all other samples has been calculated. One of the aims of such strategy is to determine a threshold value that will allow the system to classify the sample as either malware or benign. The experiment took ten days to run over all samples in the dataset.

Table 1. Statistics for Dataset that is used in the Experiment in Terms of Malware Type Quantities and File Sizes

Text Type	Quantity	Min File Size in KB
Benign	10	162
Virus	21	103
Worm	34	100
Trojan	20	117

4.2. Evaluation Measures

To evaluate the proposed mechanism, three metrics are used as discussed below:

Detection rate:

The detection rate is defined as the percentage of correctly identified malware samples as malware and is calculated using True Positives (TP) and False Negatives (FN) as shown in the below equation.

$$Detection\,Rate = \frac{TP}{TP + FN}$$

False alarm rate:

The false alarm rate is the percentage of benign samples labelled as malware, which is the number of benign samples classified as malware (*i.e.*, False Positives (FP)) divided by the total number of malware samples, as shown in the below equation.

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

Accuracy:

The accuracy is the overall accuracy of the system to detect malware and benign files, as shown in the below equation which introduces True Negatives (TN).

$$Accurcy = \frac{TP + TN}{TP + TN + FP + FN}$$

A receiver operating characteristics (ROC):

ROC curve is a two-dimensional graph in which TP rate is plotted on the Y axis and FP rate is plotted on the X axis. An ROC curve depicts relative tradeoffs between benefits (true positives) and costs (false positives).

4.3. Results and Discussion

The results obtained based on experimenting with the proposed system and the other comparable methods are illustrated in Table 2. According to these results, the highest levels of detection rate and accuracy (*i.e.*, 98.6% and 98.8% respectively) are achieved by the proposed system. All methods have achieved zero percent false alarm rates, which means that all call graph methods achieve good results when dealing with benign samples. Furthermore, ROC curve has been plotted to measure the performance of the proposed method in comparison with LCS method as depicted in Figure 5.

Table 2. Results of Detection Rate, Accuracy, and False Alarm Rate for Proposed Method, Normal LCS and N-gram

	Detection Rate%	Accuracy %	False alarm Rate %
LCS	97.3	97.6	0
N-gram	97.3	97.6	0
Proposed system	98.6	98.8	0

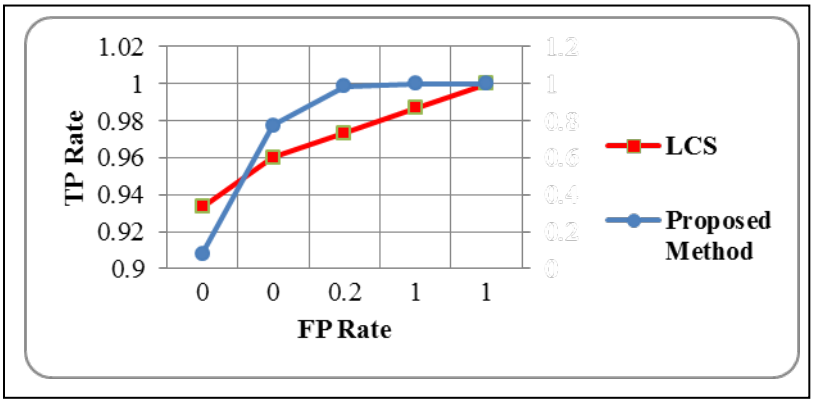

Figure 5. The ROC Curves of the Proposed Method and Normal LCS

The plotted ROC curve confirms that the proposed method starts with lower true positive rate compared with LCS method but quickly converges to its highest values.

The proposed method has an advantage over normal LCS algorithm in that, it increases the chance of finding similarity between malware samples. This advantage is achieved by dividing both input API call graphs, namely, data graph and query graph, into paths and performing comparisons based on these paths rather than performing comparisons based on whole graphs as normal LCS algorithm does.

A threshold value of 50% has been determined as a measure of similarity to classify a sample as either malware or benign and has been used during the experiment. Any sample whose similarity to a malware sample is more than or equal to the threshold, has been classified as malware. The approach taken by the proposed system greatly contributes to finding similarity between provided samples and malware samples and overcomes other approaches. Table 3 shows how the proposed system manages to detect similarity with samples taken from different malware families, namely, Trojan horse, virus, and worm. The table shows that, based on the threshold, normal LCS has failed to find similarity for one of

the samples, which is the virus, whereas the proposed system succeeds with all provided samples.

Table 3. The Similarity to Some Sample from Different Malware Family

Sample	Proposed system	LCS
XTrojan.Win32.Sweet.apm	67.10	48.29
XVirus.Win32.BingHe-ee.apm	61.45	14.61
XWorm.Win32.Bymer.a.apm	66.77	56.73

5. Conclusion

In this study, a malware detection system has been proposed. The system is based on API call graph and differs from other systems in that the proposed system simplifies the data dependent API call graph to reduce the computation complexity of the matching process by selecting paths with the same edge label in the two input API call graphs, namely, the query graph and the data graph. Then, the system calculates the similarity using Longest Common Subsequence (LCS) algorithm. The proposed system has been implemented using Delphi programming language, and the experimental results that are conducted on a malware dataset show that the system has 98% detection rate and 0% false positive rates, which reflects a significant improvement over previous methods.

Acknowledgements

This work is supported by International Doctoral Fellowship IDF in Universiti Teknologi Malaysia. The authors would like to thank Research Management Centre (RMC) Universiti Teknologi Malaysia and ELmashreq College for Science and Technology (MCST) for the support and incisive comments in making this study a success.

References

[1] S. J. Mihai Christodorescu, D. Maughan, D. Song, and C. Wang, "Malware Detection", **(2007)**.
[2] I. You and K. Yim, "Malware obfuscation techniques: A brief survey", **(2010)**, pp. 297-300.
[3] X. Hu, T. Chiueh and K. G. Shin, "Large-scale malware indexing using function-call graphs", **(2009)**, pp. 611-620.
[4] B. G. Ryder, "Constructing the call graph of a program," Software Engineering, IEEE Transactions on, **(1979)**, pp. 216-226.
[5] A. Lakhotia, "Constructing call multigraphs using dependence graphs", **(1993)**, pp. 273-273.
[6] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering", Journal in Computer Virology, **(2011)**, pp. 1-13.
[7] Microsoft. (2012, 1-February-**2012**). MSDN library. Available: http://msdn.microsoft.com/library/default.aspx.
[8] Hex-rays. (12-2-**2013**). The IDA Pro disassembler and debugger. Available: http://www.hexrays.com/idapro/.
[9] API Monitor . Spy and display API calls made by Win32 applications. Available: http://www.apimonitor.com, **(2012)**.
[10] K. Riesen, X. Jiang and H. Bunke, "Exact and Inexact Graph Matching: Methodology and Applications", Managing and Mining Graph Data, **(2010)**, pp. 217-247.
[11] E. W. Myers, "An O (ND) difference algorithm and its variations", Algorithmica, **(1986)**, vol. 1, pp. 251-266.
[12] N. Idika and A. P. Mathur, "A survey of malware detection techniques", Purdue University, **(2007)**.
[13] D. W. Yanfang Ye, Tao Li and Dongyi Ye, "An intelligent PE-malware detection system based on association mining", Springer, **(2008)**, pp. 12.

[14] Y. Ye, T. Li, Q. Jiang and Y. Wang, "CIMDS: Adapting postprocessing techniques of associative classification for malware detection", IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews, vol. 40, **(2010)**, pp. 298-307.
[15] M. F. Zolkipli and A. Jantan, "A Framework for Malware Detection Using Combination Technique and Signature Generation", Computer Research and Development, 2010 Second International Conference on, **(2010)**, pp. 196-199.
[16] H. J. Vinod P., Y. K. Golecha, M. Singh Gaur and V. Laxmi, "MEDUSA: MEtamorphic malware Dynamic analysis Using Signature from API", ACM, **(2010)**, pp. 7.
[17] J. Rhee, R. Riley, D. Y. Xu and X. X. Jiang, "Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory", Recent Advances in Intrusion Detection, vol. 6307, **(2010)**, pp. 178-197.
[18] A. S. Yoshiro Fukushimayz, Y. Horiyz, and K. Sakuraiyz, "A Behavior Based Malware Detection Scheme for Avoiding False Positive", IEEE, **(2010)**, pp. 6.
[19] Y. Park, D. Reeves, V. Mulukutla and B. Sundaravel, "Fast malware classification by automated behavioral graph matching", **(2010).**
[20] H. Guo, J. Pang, Y. Zhang, F. Yue and R. Zhaok, "HERO: A novel malware detection framework based on binary translation", **(2010)**, pp. 411-415.
[21] A. A. E. Elhadi, M. A. Maarof and A. H. Osman, "Malware detection based on hybrid signature behavior application programming interface call graph", American Journal of Applied Sciences, vol. 9, **(2012)**, pp. 283-288.
[22] J. Lee, K. Jeong and H. Lee, "Detecting metamorphic malwares using code graphs", **(2010)**, pp. 1970-1977.
[23] Y. Park and D. Reeves, "Deriving common malware behavior through graph clustering", **(2011)**, pp. 497-502.
[24] O. Kostakis, J. Kinable, H. Mahmoudi and K. Mustonen, "Improved call graph comparison using simulated annealing", **(2011)**, pp. 1516-1523.
[25] K. Kim and B. R. Moon, "Malware detection based on dependency graph using hybrid genetic algorithm", **(2010)**, pp. 1211-1218.
[26] L. B. J. P. Y. Z. W. F. J. Zhu, "Detecting Malicious Behavior using Critial API calling Graph Matching", IEEE, **(2009)**, pp. 4.
[27] D. Harley, "Making sense of anti-malware comparative testing", Information Security Technical Report, **(2009)**, vol. 14**.**
[28] **(2011)**. VX Heavens Virus Collection. Available: http://vx.netlux.org/.

Authors

Ammar Ahmed E. Elhadi is PhD researcher at Universiti Teknologi Malaysia. His research area includes Information and Communication Security, Malware Prevention and Detection System. He received his M.Sc in Computer Science degree from University of Khartoum- Sudan. Currently he is associated with the Information Assurance & Security Research Group (IASRG) at UTM. Formerly he is working as Lecture in Department of Computer Science at University of Bahri –Sudan also work as Lecture in Department of Software Engineering at Elmashreq Collage for Science and Technology – Sudan.

Mohd Aizaini Maarof received his B.Sc (Computer Science) from WMU - USA, M.Sc (Computer Science) from CMU - USA, and PhD (IT Security) degree from Aston University, Birmingham, UK. He is a Professor at Faculty of Computer Science & Information System, UTM. His research interest is in Information System Security. He is also a member of Information Assurance & Security Research Group (IASRG) at UTM.

Bazara Barry received his Computer Science first class B. Sc. (Honors) and M. Sc. degrees in 2001 and 2004 respectively from Faculty of Mathematical Sciences - University of Khartoum, Sudan and his PhD in Electrical Engineering from University of Cape Town – South Africa in 2009. In 2002 he started his academic career at University of Khartoum as a teaching assistant and became a lecturer later in 2004. He is currently working as an assistant professor at and heading the department of Computer Science – University of Khartoum.