Enhancing the Detection of Metamorphic Malware using Call Graphs

ARTICLE *in* COMPUTERS & SECURITY · OCTOBER 2014

Impact Factor: 1.03 · DOI: 10.1016/j.cose.2014.07.004

4 AUTHORS, INCLUDING:

Erbiai Elhadi
Abdelmalek Essaâdi University
2 PUBLICATIONS **10** CITATIONS
SEE PROFILE

Mohd Aizaini Maarof
Universiti Teknologi Malaysia
123 PUBLICATIONS **288** CITATIONS
SEE PROFILE

Bazara Barry
University of Khartoum
22 PUBLICATIONS **40** CITATIONS
SEE PROFILE

COMPUTERS & SECURITY 46 (2014) 62–78

Available online at www.sciencedirect.com

ScienceDirect

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

Computers & Security

Enhancing the detection of metamorphic malware using call graphs

Ammar Ahmed E. Elhadi [a,b,*], *Mohd Aizaini Maarof* [a], *Bazara I.A. Barry* [c], *Hentabli Hamza* [a]

[a] *Information Assurance and Security Research Group, Faculty of Computing, Universiti Teknologi Malaysia, Malaysia*

[b] *Elmashreq College for Science and Technology, Sudan*

[c] *Faculty of Mathematical Sciences, University of Khartoum, Sudan*

ARTICLE INFO

Article history:
Received 27 June 2013
Received in revised form
30 June 2014
Accepted 13 July 2014
Available online 22 July 2014

Keywords:
Computer security
Malware
Malware detection
API call graph
API call graph construction algorithm
API call graph matching algorithm

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. API call graph techniques follow two main steps, namely, transformation of malware samples into an API call graph using API call graph construction algorithm, and matching the constructed graph against existing malware call graph samples using graph matching algorithm. A major issue facing malware API call graph construction algorithms is building a precise call graph from information collected about malware samples. On the other hand call graph matching is an NP-complete problem and is slow because of computational complexity .In this study, a malware detection system based on API call graph is proposed. In the proposed system, each malware sample is represented as an API call graph. API call graph construction algorithm is used to transform input malware samples into API call graph by integrating API calls and operating system resource to represent graph nodes. Moreover, the dependence between different types of nodes is identified and represented using graph edges. After that, graph matching algorithm is used to calculate similarity between the input sample and malware API call graph samples that are stored in a database. The graph matching algorithm is based on an enhanced graph edit distance algorithm that simplifies the computational complexity using a greedy approach to select best common subgraphs from the integrating API call graph with high similarity, which helps in terms of detecting metamorphic malware. Experimental results on 514 malware samples demonstrate that the proposed system has 98% accuracy and 0 false positive rates. Detailed comparisons against other detection methods have been carried out and significant improvement over them is shown.

* *Corresponding author*. Information Assurance and Security Research Group, Faculty of Computing, Universiti Teknologi Malaysia, Malaysia.

E-mail addresses: ammareltayeb@gmail.com (A.A.E. Elhadi), aizaini@utm.my (M.A. Maarof), bazara.barry@gmail.com (B.I.A. Barry), hentabli_hamza@yahoo.fr (H. Hamza).

http://dx.doi.org/10.1016/j.cose.2014.07.004
0167-4048/

1. Introduction

Malicious software, or malware, is one of the most pressing security problems on the Internet. The number of Internet attacks increased by 42% in 2012, and 31% of attacks aimed at businesses with less than 250 employees and 500 organizations in a single day (Corporation, 2013). Attackers generate new malware samples from old ones using code obfuscation, polymorphism, and new delivery mechanisms such as web-attack toolkits, which greatly contribute to the significant increase in the number of malware variants being distributed. Moreover, there is a tendency among malware writers to use high-level programming languages to write malware and compile it into binary afterwards, 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. 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 (Ryder, 1979). 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 (Lakhotia, 1993). They can also be used to find procedures that are never called.

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 (Microsoft, 2012).

The API calls list is extracted from a binary executable through static analysis of the binary with disassembly tools such as IDA Pro (Pro, 2012) or through dynamic analysis after executing the binary in a simulated environment, which is the technique adopted by tools such as API monitor (Monitor, 2012). 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 analyse 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.

API call graph construction using static tools can build a multipath graph easily, but fails to get the real API calls since hackers apply techniques like packing and obfuscation to hide malware calls. Once all API calls (i.e., the vertices in the API call graph) are identified, the edges between the vertices are added based on the function calls extracted at the binary executable analysis step.

The construction of the API call graph for a program without API operating system resources (i.e., without the parameters used by API calls) is very simple. One pass through the API call collected list enables the discovery of all the nodes and all the edges in the call graph simply by making a table of all API calls and the references they contain. Each API call must be analysed just once and the order in which API calls are examined is not significant. When API operating system resources are present, however, the work done in constructing the call graph depends upon the order in which API calls are analysed. In programs containing API operating system resources, it is possible to have a reference to API operating system recourses which may represent invocations of several distinct other API calls. In order to ascertain all possible invocations that result from such a reference in an API call, it is important to know all the other API calls associated with that API operating system resource.

Detecting malware through the use of call graphs requires means to compare call graphs, namely, model and data graphs, to distinguish call graphs representing benign programs from call graphs that are based on malware samples. To compare call graphs, a graph matching algorithm is used. Matching can be classified into two categories, namely, *exact matching* and *inexact matching*. Exact graph matching applies when the two graphs have the same number of vertices, whereas in inexact graph matching the two graphs have different number of vertices (Riesen et al., 2010). Graph matching can be done with one of the following techniques:

i. Graph isomorphism.
ii. Maximum common subgraphs (MCS).
iii. Graph edit distances (GED).

Graph isomorphism and MCS are proven to be an NP-Complete problem (Garey et al., 1976; Michael and Johnson, 1979), and GED is proven to be an NP-hard problem (Zeng et al., 2009; Hu et al., 2009). Furthermore, both MCS and GED are computationally expensive to calculate (Kinable and Kostakis, 2011). A considerable amount of research has been devoted to develop fast and accurate approximation algorithms for these problems, mainly in the field of image processing (Gao et al., 2008) and for bio-chemical applications (Raymond and Willett, 2002; Weskamp et al., 2007). Graph edit distance (GED) is the best algorithm for matching inexact graph type (Riesen et al., 2010; Gao et al., 2010) but its complexity makes it slow (Riesen and Bunke, 2009).

The work presented in this study is different from the previous API call graph-based systems in that it addresses two shortcomings. First, most of the previous API call graph-based systems focused on constructing the API call graph-based on dependence between the nodes of API call graph, and the node itself could either be an API call or parameter of the API call (i.e., operating system resource). However, to get more precise

API call graph for malware detection, it is recommended to integrate API call graph and use its node to represent both the API calls and their parameters. Second, call graph matching is an NP-complete problem (i.e., problems that can be solved in polynomial-time on a nondeterministic Turing machine) and suffers from slowness because of its computational complexity.

Furthermore, many existing graph-similarity query processing methods cannot scale to large graphs. For example, a variant of the XVirus.Win32.HLLO.Ant.b.apm family has 4343 nodes and 1,538,954 edges in its integrating API call graph. To deal with large graphs, the proposed system in this study divides the integrating API call graph into subgraphs based on structure and attributes of the nodes and edges. Moreover, it applies a modified greedy algorithm that supports graph edit distance (GED) metric. An approximate algorithm for the computation of graph edit distance is used to map an edge from the query graph with a path or an edge in the data graph when the query graph edge does not match any edge in the data graph. Furthermore, when a query graph edge has more than one match in the data graph, the algorithm maps this edge with the one that has the most matching neighbours in the data graph.

The rest of the paper is organized as follows. Section 2 discusses the previous studies related to malware detection based on call graph. Section 3 introduces the proposed malware detection system. Section 4 describes in detail the proposed malware detection system. Section 5 discusses the proposed system experimental setup alongside the evaluation metrics and analyzes the obtained results. Limitations and conclusions are presented in Sections 6 and 7 respectively.

2. Related work

Traditional techniques in malware detection such as signature-based and behaviour-based approaches have many limitations. 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 (Hu et al., 2009; Elhadi et al., 2012). Limitations in signature-based and behaviour-based techniques have been overcome by applying machine learning techniques (Rieck et al., 2011; Shahzad et al., 2011), data mining techniques (Schultz et al., 2001; Santos et al., 2011), and graph-based techniques, namely, Control Flow Graph (Bonfante et al., 2007), and API Call graph (Lee et al., 2010).

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.

Some malware researches focus on graphs, including control flow graphs (CFG), call graphs, and code graphs. 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 that represent possible control flows from one basic block to another. After the generation of a CFG, all the computable nodes are ignored while only the nodes with API calls are kept to extract the API graph from the interprocedural CFG. To transform the call graph to a code graph, the nodes in the call graph are grouped to reduce this call graph so that nodes within the same group in the call graph are unified. During node unification, edges representing call relationships are maintained (Lee et al., 2010). Such researches build their graphs in different ways and analyse and compare graphs using different methods.

To build the graph, researchers present graph nodes as system calls. For example, Lee et al. (2010) 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 (Park et al., 2010). Some other studies use graph nodes to denote kernel objects instead of system calls (Park and Reeves, 2011).

On the other hand, Kostakis et al. (2011) build a graph from subroutines as nodes and their call references as edges, whereas Kim and Moon (2010a) 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. In Haoran et al. (2010), Bai (2009) a Critical API Graph (CAG) is extracted from a CFG for each malware to define its behaviour.

Christodorescu et al. (2008) proposed an algorithm to construct a dependence call graph, where graph nodes represent system calls and two types of dependences exist between system calls to present the edges. Kolbitsch et al. (2009) proposed behaviour graphs that share similarities with the graphs of Christodorescu et al. (2008) but they do not have unconstrained system call arguments, and the semantics of edges is somewhat different. An edge is introduced from node x to y when the output of call x produces a taint label that is used in any input argument for call y. An edge represents only a data dependency between system calls x and y. Fredrikson et al. (2010) enhanced the dependence graphs proposed by Christodorescu et al. (2008) and Kolbitsch et al. (2009) by assigning labels to particular files, directories, registry keys, and devices based on their significance to the system (e.g., system startup list, firewall settings, and system executables).

Previous works have shown that different strategies can be used to build API call graph such as sequential profile and data dependent approaches which are API call node base, and kernel object API call graph which is parameter node based. The main problem in building precise API call graph form information collected for malware samples is that API Call graph construction algorithm build call graph that is not complete in terms of the number of nodes because API call and operating system resource are not included in one call graph as graph node, which makes the call graph inaccurate (Park et al., 2010; Park and Reeves, 2011). Ahmed et al. (2009) prove that combined API call and parameters (as spatio-temporal features set) increases the detection accuracy rather than standalone API call or parameters set.

Table 1 – Summary of some recent malware detection approaches based on call graph.

Approach	Graph generation	Weakness	Graph matching	Weakness	Note
(Lee et al., 2010)	Graph nodes: System calls Graph edges: System call sequence	Misses to include information about Parameter in the call graph	Graphs are compared by computing intersections and unions	No Assumption about missing edge and mapping of edge with neighbours	Achieves 91% detection ratio of real-world malware samples
(Park et al., 2010)	Graph nodes: System calls Graph edges: Dependency of two system calls	No information about the sequence of system call	Graphs are compared by building maximal common subgraphs and calculating the distance between them	No Assumption about missing edge and mapping of edge with neighbours	An automated classification technique, fast, and has a low false positive rate
(Park and Reeves, 2011)	Graph nodes: Kernel objects Graph edges: Kernel objects dependency	No information about dependency between system calls is included in the call graph	Graphs are compared using Weighted Common Behavioural Graph (WCBG)	No Assumption about missing edge and mapping of edge with neighbours	Shows high malware detection rates, and false positive rates close to 0%
(Bai, 2009; Guo et al., 2010)	Graph nodes: System calls Graph edges: System call sequence	Misses to include information about Parameter in the call graph	Graphs are compared using defined Malware Behaviour Template	No Assumption about missing edge and mapping of edge with neighbours	Highest detection and low false alarm rates compared to three commercial antivirus
(Kim and Moon, 2010b)	Graph nodes: a line in the semantic code. Graph edges: The dependency between two lines	No information about the sequence of system call	Comparison is done using subgraph-isomorphism to determine whether the target is a polymorphic variant	No Assumption about missing edge and mapping of edge with neighbours	Improves detection accuracy and reduces computational cost
(Zhao et al., 2012; Eskandari and Hashemi, 2012)	Graph nodes: System calls Graph edges: System call dependency	Parameter information is not included in the call graph	Data mining & Feature selection	Base on training and learning model	Accuracy range from 84% to 97% for different machine learning techniques

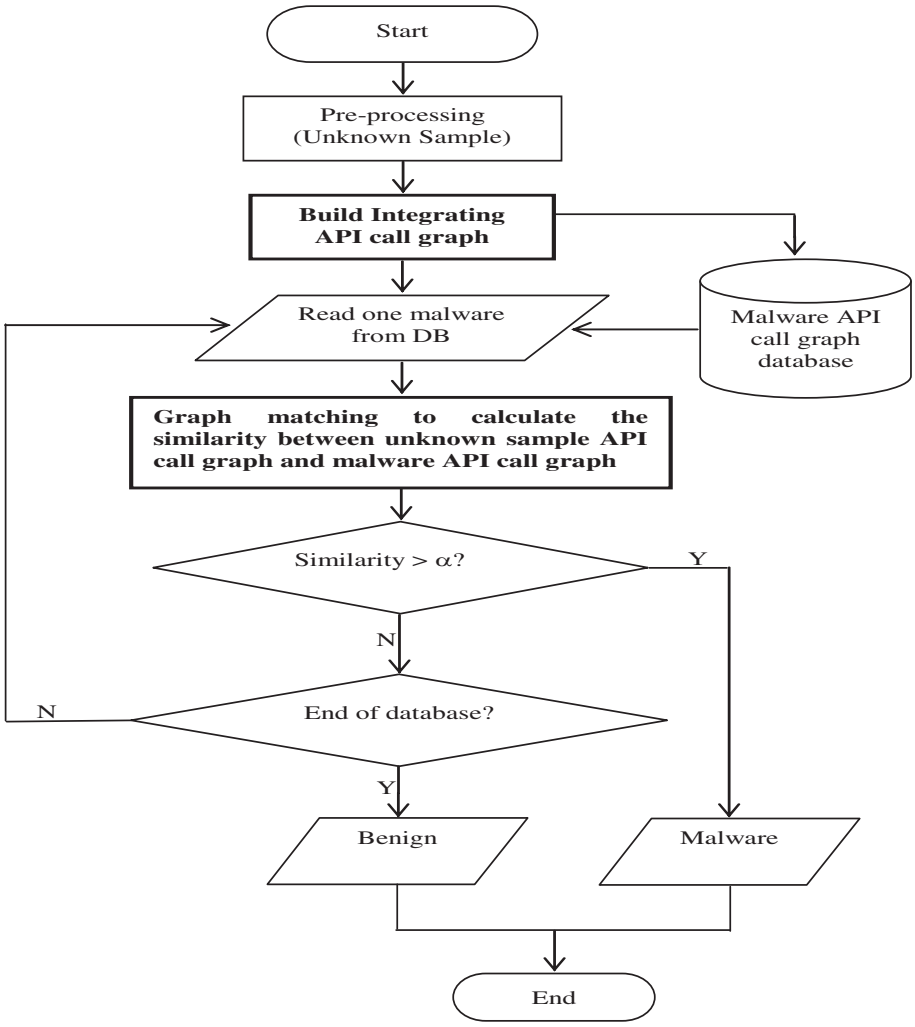


Fig. 1 – The general framework of the proposed system.

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. Malware detection researches compare graph using different graph matching techniques, some of them use maximal common subgraph (Park and Reeves, 2011; Kim and Moon, 2010) and Weighted Common Behavioural Graph Generation based on an approximate algorithm, while others build formula using intersection and union of the graphs (Lee et al., 2010). However, all these techniques require time and space due to NP- completeness of the problem and computational complexity which make them prohibitively expensive to use for large graphs (Riesen and Bunke, 2009). However, to reduce the computational cost, Riesen et al. developed a polynomial-time algorithm to compute approximate graph-edit distance using Bipartite Graph Matching (Riesen and Bunke, 2009).

Hu proposed Graph Edit Distance (GED) based on bipartite graph matching and neighbour-based Hungarian algorithm (Hu et al., 2009). The main problem in this proposal is that it assumes a fixed cost of matching two function nodes (Xu et al., 2013). To overcome bipartite matching problem Kostakis proposed using simulated annealing to improve the call graph matching (Kostakis et al., 2011). His experiment shows an improvement in both execution time and solution quality. Table 1 summarizes some of the recent malware detection techniques and compares between them in terms of graph generation, graph matching and detection rate.

3. Malware detection system overview

The general framework of the proposed system is outlined in Fig. 1. The basic design of the proposed system is based on the work proposed by Lee et al. (2010). To show the enhancement and differences of the malware detection system, the bold boxes show the enhanced components proposed. The steps are: First, Lee use API call graph construction algorithm focused on constructing the API call graph-based on API calls as graph nodes and sequence dependence between the nodes. However, to get more precise API call graph for malware detection, it is recommended to integrate API call graph and

use its node to represent both the API calls and their parameters. Second, Lee (Lee et al., 2010) used API call graph matching algorithm to compare graphs by formula using intersection and union of the graph edges ignoring the case of when the query graph edge does not match any edge in the data graph and also when a query graph edge has more than one match in the data graph.

The following are the steps taken by the proposed system

Step 1 Read unknown sample as input.

The proposed system reads the unknown PE sample and pre-processes it to extract its API calls and parameters. The pre-processing step 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 (Riesen et al., 2010) under secure environment so the API call and its parameters are recorded.

Step 2 Build Integrating API call graph.

In this step, using the proposed enhanced API call construction algorithm, the input list of API calls is converted into call graph. If the input is known malware sample, the constructed API call graph will be used to update malware API call graph database, so it may be used later by the matching algorithm.

Step 3 Read one integrating API call graph sample from database.

To identify the unknown input it is needed to find similarity with pervious samples from the database. This step reads one malware API call graph from database to calculate its similarity with input sample.

Step 4 Calculate the similarity between the query graph and the data graph.

The enhanced graph edit distance matching algorithm is used to match and calculate the similarity between the input (unknown) sample and the read one from database (known) sample.

Step 5 Check the similarity value.

If the similarity $> \alpha$, then the proposed system identifies input sample as malware. Otherwise, it needs to check the remaining database samples by repeating step 3.

Step 6 Check if malware samples in database are finished.

If all samples in malware database have been processed, it means the input (Unknown) sample has no similarity with any malware sample and the system will identify it as benign program and finish. Otherwise, the system needs to finish the database by repeating step 3.

The next section discusses in detail the integrating API call graph construction algorithm and approximate algorithm for the computation of graph edit distance.

4. The proposed system

The proposed detection system is based on the idea that most malware samples are generated from existing samples. To enhance the detection of metamorphic malware, the proposed system takes advantage of sequence profile and the data dependence schemes to construct API call graph, integrating API call with operating system resource which is seen to be more precise, more accurate in terms of representation and less susceptible to the low level obfuscation applied by hackers to avoid detection. By integrating API call with operating system resource, the proposed system combines both signature and behaviour of input samples rather than focussing only on either of them. The details of the proposed algorithm are provided in Section 4.1. Furthermore, approximate algorithm for the computation of graph edit distance and graph-similarity is used to trace common subgraphs into the integrated API call graph. The approximate graph matching algorithm supports graph edit distance metric with $O(|E(Q)||P|)$ computation complexity (Zhu et al., 2011), where $|E(Q)|$ is the number of edges in query graph and $|P|$ is the number of best paths used in the algorithm. Details of the proposed algorithm are provided in Section 4.2.

4.1. *Integrating API call graph construction algorithm*

In this step the integrating API call graph is built using API calls that are collected from the pre-processing step. The input here is text file that contains the list of API calls along with the parameter list, especially operating system resources such as files, registers, network resources, processes, and memory. Each text file represents one malware sample. The API call graph that is constructed by the proposed algorithm has the advantage that it is build from the API calls and the operating system resources as graph nodes. The graph edges represent the reference between the nodes. The nodes have two attributes: API call and operating system resource, and the node label is the API call itself or the operating system resource (OSR).

The Integrating API call graph can be defined as $G = (V, E, I_G, V^G, E^G, \sum)$, where V is a set of nodes of$G$; $E \subseteq V \times V$ is a set of edges of$G$; and IG:$V \rightarrow \sum$ is a function that maps nodes to labels (API call name, operating system resource name) in the alphabet set$\sum$. In addition, each node and edge in the API call graph is associated with attributes, which are denoted by V^G, E^G respectively.

A graph node represents either an API call or an OSR; edges represent several kinds of dependence among the different types of node, which can be distinguished by the label attached to them. Four dependencies are distinguished, namely, sequence, data, declaration, and API call. Each of these dependencies will be briefly discussed shortly.

In the proposed algorithm, a pass is made through the API list once to determine the nodes in the graph. The search through the list will be n times where n is the number of the rows in API call list. Consider the malware sample to be a finite set of API calls $\{ API_i \}_{i=1}^{n}$. Each API call has an ordered list of parameters that appear in the sample list, particularly the OSR. API calls communicate with each other through the use

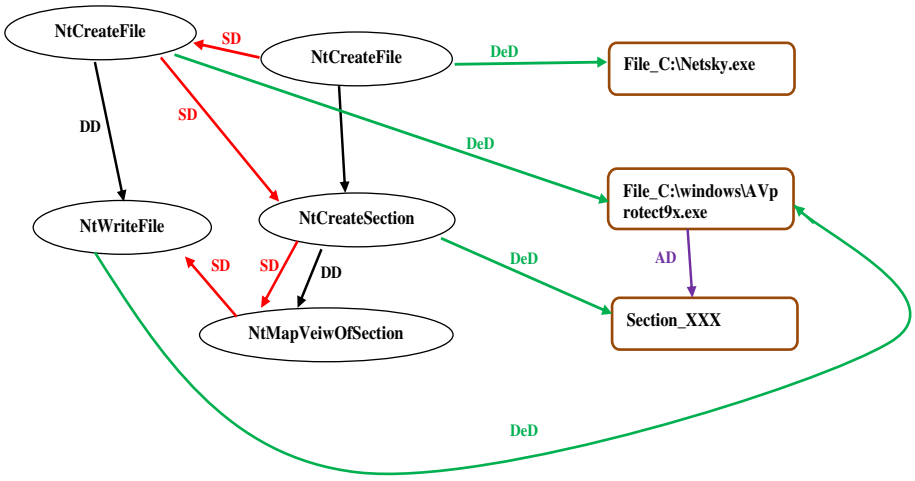


Fig. 2 – Example of an Integrating API call graph.

of handles. In order to understand the communication between the API calls and the OSR, some types of dependence between the API call and the OSR will be defined.

The Integrating API call graph construction algorithm identifies the nodes and the edges for each line of instruction. First, the API call node and its Control Sequence Dependent (SD) edge (connects API call to API call if the execution of the first API call is followed by the execution of the second API call) and Data Dependent (DD) edge (connects API call to API call if the API call is data or flow dependent on the other API call) are defined. Then the operating system resource (OSR) node along with its Declaration Dependent (DeD) edge (connects API call to OSR) and API Dependent (AD) edge (connects OSR to OSR) are defined.

In the following, the different types of edges are further discussed along with their notations. In particular, let v_1 and v_2 be two nodes of graph G, the following can be defined:

a) Sequence Dependence (SD)

An SD edge is an edge that connects two API call nodes. If the execution of the API call node v_1 is followed by the execution of the API call node v_2, then v_2 is SD onv_1. In this case, the following notation is used:

$$v_1 \xrightarrow{SD} v_2$$

b) Data Dependence (DD)

The DD edge also connects two API call nodes. In this case, we say that the API call node v_2 is data flow dependent on the API call node v_1, if and only if v_2 uses an operating system resource that v_1 created or used. The above relationship can be denoted using the following notation:

$$v_1 \xrightarrow{DD} v_2$$

c) Declaration Dependence (DeD)

A DeD edge connects an API call node v_1 to an operating system resource node v_2. The dependence exists from v_1 to v_2 when the API call defines or uses the operating system resource that is mentioned in the parameter list; in this case the following notation is used:

$$v_1 \xrightarrow{DeD} v_2$$

d) API Dependence (AD)

An AD edge connects two operating system resource nodes; once an operating system resource node v_1 is created or used by an API call, its handle is returned and used by another operating system resource node v_2 using API calls that have both nodes (operating system resources) as parameters. This means that the operating system resource node v_1 is accessed by another operating system resource node v_2 using an API call; in this case the following notation is used:

$$v_1 \xrightarrow{AD} v_2$$

For example, Fig. 2 shows an Integrating API call graph in which there are two types of node (Oval, Rounded Rectangle) and four types of edge with four different colours. The figure shows the different types of dependence, namely, the red edge (in web version) is an example of SD edges, the black edge for DD, the green edge (in web version) for DeD and the purple edge (in web version) for AD.

The integrating API call graph can be divided into four main subgraphs according to the type of node and edges; these subgraph features are summarized in Table 2. These subgraphs show that the Integrating API call graph is more precise than the other API call graphs, which contain at least two subgraphs. The subgraphs can also help in the matching algorithms by minimizing the complexity of the comparison process.

According to literature, Ahmed et al. prove that using a combination between API calls and parameters (OS resources) improves malware detection accuracy. Since, SD and AD call graphs are based on API calls and parameters respectively, DD call graph is considered the most effective in terms of enhancing detection accuracy. On the other hand, DeD is

Table 2 – List of features extracted from integrating API call graph.

Feature	Number	Description
Node	1	Total nodes of API call graph
Type of node	2	All API call nodes All OSR nodes
Edge	1	Total edges of API call graph
Type of edges	4	All Sequence Dependence edges All Data Dependence edges All Declaration Dependence edges All API call dependence edges
Subgraph	4	All API call nodes and all Sequence Dependence edges All API call nodes and all Data Dependence edges All OSR Nodes and all API call Dependence edges Total nodes of API call graph and all Declaration Dependence edges
Node of subgraph	3	All API call nodes All OSR nodes Total node of API call graph

based on the sequence of API call and a link to its parameters. Such a feature makes it relatively easy for attackers to obfuscate DeD call graphs by changing the sequence.

4.2. *Approximate algorithm for the computation of graph edit distance*

For given large attributed data and query graphs, the problem of approximate graph matching is to compute a subgraph of the data graph that best matches the query graph. Since malware variants are generated from previously seen ones, and by representing malware samples in an integrating API call graph, the problem of identifying malware variants is to find the best matching subgraph with high similarity.

To match an integrating query graph to a data one, an attempt is made to find an exact match between an edge in the query graph and an edge in the data graph. If such an edge is not found in the data graph, it is attempted to find a path in the data graph that matches the edge in the query graph to some degree of similarity. In case the data graph has more than one edge that matches the query graph edge, the data graph edge with the neighbours that best match the neighbours of the query graph edge is chosen. It is important to note that an edge is a special case of a path (i.e., a path that contains one edge), and hence the interchangeable use of the two terms is allowed in this context. To that end, a modified greedy approach that supports graph edit distance (GED) metric is developed to find the set of best paths from the data graph that match the set of query graph edges and construct the best subgraph with high degree of similarity.

The next section discusses the problem of an approximate graph query in an attributed graph and shows the proposed matching algorithm. Table 3 provides the interpretation of the symbols that are used in the below discussion.

Approximate graph matching problem:

For two graphs Q and G, the problem of approximate graph matching (AGM) is to find the best matching subgraph G_a as defined by the following equation:

Table 3 – Description of notation that is used in approximate graph matching problem/algorithm.

Notation	Description
Q	A query graph
G, G_a	A data graph/an answer graph
$V(G)$, $E(G)$	The node/edge set of G
$A^G(v)$, $l^G(v)$	The value of attribute/label of v in G
$p \in G$	A path p that appears in G
u_e, v_e	The two ending nodes of an edge e
$f(u_e)$, $f(v_e)$	The two ending nodes that match the edge e
$argmax_G\ sim(Q,G)$	Argument G at which the function $sim(Q,G)$ is maximized.

$$G_a = \text{argmax}_{G'}\ sim(Q, G') \tag{1}$$

where $G' \subseteq G$ and $G' \approx Q$ and sim(Q,G') is the degree of matching between Q to G'. If sim(Q,G') = 1, then Q is graph isomorphic to G'.

In the following, it is shown how a greedy algorithm corresponds an edge (e) to a path (p) (a path or an edge) and computes the similarity between the query graph and the data graph-based on graph edit distance (GED) metric.

Suppose that there is a query graph Q, a data graph G and a mapping function $f: V_q \rightarrow V_g$. Assume that there is a query edge e in a query graph and two nodes x, y that belong to data graph G, where $A^G(x) = A^Q(u)$ and $A^G(y) = A^Q(v)$. Moreover, the two nodes belong to the same dependence subgraph in the integrating API call graph. In order to support graph edit distance (GED), so a pair of isolated nodes in the data graph G is allowed to be best corresponded to the query edge e, the best path that corresponds to e and connects x and y is defined as:

$$P'(e,x,y) = \text{argmax}_{p \in S_2'}\{sim(e,p)\}, P^* = sim(e,P') \geq sim(e,\{x,y\})?P' : \{x,y\} \tag{2}$$

where there can be a pair of isolated nodes and the $sim(e,p)$ is defined as:

$$sim(e,p) = \frac{sim(u_e, f(u_e)) + sim(v_e, f(v_e))}{2} \times \frac{\sum_{v_0 \in p} sim(f^{-1}(v_0), v_0)}{|p|} \tag{3}$$

where $|p|$ is the number of nodes on$p$ and $sim(u,v) = 1$ if $A^Q(u) = A^G(v)$ and $l^Q(u) = l^G(v)$. The degree of matching between a query graph Q and a data graph G (i.e., the graph matching measure, $sim(Q,G)$) is defined as:

$$sim(Q,G) = \left(\max_f \sum_{e \in E(Q)} \max_f sim(e,p)\right) \Big/ |E(Q)| \tag{4}$$

where $|E(Q)|$ is the number of edges in the query graph.

To deal with the huge number of nodes and edges in the integrating API call graph, the matching process starts by dividing the query graph and the data graph into subgraphs (i.e., Sequence Dependent, Data Dependent, Declaration Dependent and API Dependent subgraph) that may further be divided into paths. Fig. 3 shows the proposed algorithm to match a query graph and a data graph and calculate the similarity. The algorithm starts by finding the similarity between subgraphs from both input graphs (query and data graph) in line 2. For each edge in the query subgraph, the

Algorithm: Finding the best subgraph with high $sim(Q,G)$
Input: Query Q and Data graph G
Output: $sim(Q,G)$
1: $sim = 0$
2: for each subgraph $S_1' \in Q$ and $S_2' \in G$, where $S_1' and S_2'$ are the same dependence subgraphs in the integrating API call graph
3: $sim' = 0$
4: for each edge $(u,v) \in S_1'$
5: for each $x,y \in V(S_2')$, where $A^G(x) = A^Q(u)$ and $A^G(y) = A^Q(v)$
/* compute P' where P' is the solution to Eq. (2) & Eq. (3) */
6: $P' = argmax_{p \in S_2'}\{sim(e,p)\}, P^* = sim(e,P') \geq sim(e,\{x,y\})?\ P' : \{x,y\}$;
7: $sim'(S_1',S_2') = \left(max_f \sum_{e \in E(Q)} max_f\, sim(e,P')\right)/|E(S_1')|$
8: $sim = sim + sim'$
9: $sim(Q,G) = sim/4$;
10: return $sim(Q,G)$

Fig. 3 – Approximate algorithm for the computation of graph edit distance.

algorithm searches for the best path/edge in the data graph with high similarity in lines 4–7. After that the similarity between the averages of the similarities of the subgraphs is calculated to estimate the whole similarity between the query graph and the data graph in lines 8–9.

Proving the correctness of the algorithm requires proving its termination (Conte et al., 2007). The proposed approximate algorithm for the computation of graph edit distance is a greedy algorithm; the similarity is computed by matching the input query graph by greedily selecting best paths after partitioning the data graph into subgraphs. That is, before selecting best paths iteration starts, the partitioning to subgraphs and selecting subgraphs form the two input graphs, namely, query and data graphs take place. For each iteration in best paths selection, an unmatched query edge will be matched to an edge or path in the data subgraph. The number of edges in each query subgraph is limited and the number of edges in the data graph is limited as well, so the iteration will stop when all edges in the query subgraph are matched. Accordingly, the algorithm will terminate when all edges in all subgraphs are matched, which proves the algorithm's correctness.

To analyse the running time of the approximate algorithm for the computation of graph edit distance, it is important to note that for each query edge in each subgraph, the algorithm needs to run lines 4–6 for d times in the worst case, where d is the maximum number of edges in data subgraph that are attribute compatible with the query edge. The time complexity is therefore $O(|E(Q)||P|)$, where $|E(Q)|$ is the number of edges in the query graph and $|P|$ is number of best paths used in the algorithm.

5. Experimentation and 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 system. Since there is no standard benchmark for comparison (Harley, 2009), and malware detection approaches that are proposed by scientific research are tested using their own malware datasets trying different assessment methods, methods such as N-gram and the one proposed in Lee et al. (2010) have been implemented in this study and used alongside the proposed system on the chosen dataset. Also, the results obtained from experimenting with the proposed system has been compared with results from Ahmed et al. (2009) because they use the same dataset that is used in this study.

5.1. *Experimental setup*

A malware dataset has been downloaded from http://www.nexginrc.org/website. The dataset consists of 416 malware and 98 benign programs. The benign executables are obtained from a freshly installed copy of Windows XP and application installers. These malware samples are obtained from a publicly-available database called 'VX Heavens Virus Collection'. API call traces of these files are logged by executing them on a freshly installed Windows XP. The logging process is carried out using a commercial API call tracer (Monitor, 2012). Table 4 gives the basic statistics about the dataset of text files that are used in this study.

The malware samples that are used as inputs in the experiment are pre-processed using the API monitor tool (Monitor, 2012) to extract API calls and their parameters (OSRs). To implement the integrating API call construction algorithm, there are two approaches. The first approach is to add nodes to the API call graph one by one in such a way that the node and its dependence edges with the previous node are added. The second approach is to first determine all nodes and then proceed to add the corresponding edges on a node by node basis according to their different dependencies. The proposed system applies the second approach. In this study no training dataset is used. Unknown samples are compared

Table 4 – Statistics for dataset that is used in the experiment in terms of malware type quantities and file sizes.

Text type	Quantity	Min. file size (KB)	Max. file size (KB)
Benign	98	162	15,811
Virus	165	103	4565
Worm	134	100	3844
Trojan	117	117	7513
Total	514		

to known samples that are stored in a database, and this process is used for each group separately.

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.

5.2. Evaluation measures

To evaluate the proposed system, four measures are used as described below:

1) Call Graph Preciseness:

The preciseness of API call graph generated by any algorithm depends upon the number of API calls dependence and parameters dependence it computes (Bhat and Singh, 2012).

2) 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.

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

3) 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 benign samples, as shown in the below equation which uses True Negatives (TN).

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

4) Accuracy:

The accuracy is the overall accuracy of the system to detect malware and benign files, as shown in the below equation.

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

5) 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. The ROC curve depicts relative tradeoffs between benefits (i.e., true positives) and costs (i.e., false positives). The curve is used to show the differences of performance between algorithms. ROC curve can be reduced into a single value by calculating the Area Under the ROC Curve (AUC). AUC value is always between 0 and 1.0, where close to 1.0 values mean high TP rate and low FP rate.

5.3. Results and discussion

To evaluate the preciseness of call graph, dataset samples are grouped into three groups, namely, Virus group, Worm group and Trojan group. The proposed system runs for each group separately and the similarity value is calculated for all samples in each group. To evaluate the proposed system, first the similarity between malware in the same family is presented to evaluate detecting of obfuscated malware. Then, the accuracy and detection rate of the proposed system are highlighted to evaluate detecting of malware variation.

5.3.1. Evaluation of the preciseness of call graph

For the experiment, all dataset samples are grouped into three groups, namely, Virus, Worm and Trojan, and the test runs for each group separately. For each group three different families are selected randomly and their statistics are collected from the program and summarized in Table 5.

As shown in Table 5, A variant of the XVirus.Win32.HLLO.Ant.b.apm family has 3440 nodes and 3439 edges in its sequence dependence API call graph, 3440 nodes and 1523669 edges in its data dependence API call graph, 903 nodes and 2210 edges in its API call dependences API call graph and 4343 nodes and 1538954 edges in its integrating API call graph.

The main observation from this statistic is that the number of nodes in the integrating API call graph (4343) is equal to the number of nodes in sequence dependence API call graph (3440) or data dependence API call graph (3440) plus the number of nodes in call dependence API call graph (903). On

Table 5 – Statistics on the constructed API call graphs for three different groups.

Samples	Sequence dependence		Data dependence		API Call dependences		Integrating	
	Nodes	Edges	Nodes	Edges	Nodes	Edges	Nodes	Edges
XVirus.Win32.HLLO.Ant.b.apm	3440	3439	3440	1523669	903	2210	4343	1538954
XVirus.Win32.HLLO.Ant.c.apm	5160	5159	5160	3515048	850	2118	6010	3535758
XVirus.Win32.HLLO.Ant.d.apm	2270	2269	2270	843229	614	1610	2884	854066
XWorm.Win32.Collo.a.apm	2674	2673	2674	1191950	921	2229	3595	1204908
XWorm.Win32.Collo.b.apm	2651	2650	2651	1158348	924	2214	3575	1171195
XWorm.Win32.Collo.c.apm	5555	5554	5555	4322495	903	2458	6458	4346672
XTrojan.Win32.AntiBTC.a.apm	2375	2374	2375	896276	659	1691	3034	907561
XTrojan.Win32.AntiBTC.b.apm	3009	3008	3009	1569774	760	1867	3769	1583525
XTrojan.Win32.AntiBTC.c.apm	2613	2612	2613	1173529	691	1854	3304	1185931

the other hand, the number of edges in the integrating API call graph (1538954) is greater than the total of the number of edges in sequence dependence API call graph (3439), data dependence API call graph (1523669), and API call dependence API call graph (2210), which shows that the proposed integrating API call graph is more precise than the other call graphs, since the proposed algorithm generates call graph with more nodes and edges dependence.

5.3.2. *Evaluation of the proposed system*

This section discusses and analyzes the results produced by the proposed system. The discussion on the results revolves around two axes, namely, detection of obfuscated malware and detection of malware variation. First, the similarities between malware in the same family are presented to evaluate detection of obfuscated malware. Then, the accuracy and detection rate of the proposed system are highlighted to evaluate detection of malware variation.

5.3.2.1. *Detection of obfuscated malware*. For this experiment, all dataset samples are grouped into three groups, namely, Virus, Worm and Trojan, and the test runs for each group separately. The similarity is calculated for all samples in each group and two different samples that belong to different families in each group are selected randomly and their similarity is shown in Table 6. Table 6 is separated into three parts, namely, (a), (b), and (c) that represent the three abovementioned groups respectively. Each part shows similarity information of two different families that belong to the group (e.g., HLLO.Hadefix and HLLO.Homer in the Virus group) and different members that belong to the same family (e.g., B, C, D, and E in HLLO.Hadefix family in the Virus group). The table shows that the calculated similarity within the same family is greater than 60%, except for some cases in the Trojan family where the similarity is about 50%. In some cases, some samples have high similarity with other samples from a different family, but still their similarity within the same family is greater than 60%. Compared to other works, The similarity in the same family equals to or exceeds 60% in Cesare and Xiang (2011), whereas it is a little more than 40% in Han et al. (2012). Based on this experiment, it is clear that the similarity between different families is about 50%, and this empirical value is chosen to be a threshold for identifying unknown samples.

Table 7 shows the highest and the lowest calculated similarity for each group in addition to detection success and failure statistics for the proposed method and other comparable methods. Table 7(a) shows that N-gram and the method proposed in Lee et al. (2010) have higher values of similarity than the proposed system. Such a finding is due to the fact that in the proposed system the names of the operating system resources do not match, which decreases the degree of similarity. However, the proposed system succeeds to detect malware samples that the comparable methods fail to detect.

Table 6 – Similarity between samples from the same family and different families.

(a) Similarity between two different family samples from the Virus group

XVirus.Win32		HLLO.Hadefix				HLLO.Homer		
		B	C	D	E	A	B	C
HLLO.Hadefix	B		66.08	66.29	65.07	60.22	56.86	57.40
	C	66.96		67.92	67.48	56.79	57.47	57.02
	D	67.07	68.10		70.49	60.28	57.29	57.20
	E	66.07	67.58	70.63		60.04	57.14	57.06
HLLO.Homer	A	48.60	45.06	47.52	47.16		62.15	62.11
	B	47.68	47.69	47.61	47.29	65.23		80.96
	C	47.85	47.12	47.37	47.09	64.77	80.67	

(b) Similarity between two different family samples from the Worm group.

XWorm.Win32		Leave			Newbiero			
		B	E	H	01	023	034	52
Leave	B		63.04	63.94	63.92	64.57	63.40	63.64
	E	60.03		60.29	59.07	59.81	59.62	60.28
	H	63.26	62.67		63.08	63.24	63.05	63.32
Newbiero	01	63.82	61.85	63.50		68.33	66.90	68.59
	023	63.80	62.00	62.96	67.66		67.40	68.06
	034	62.57	61.82	62.84	66.18	67.39		70.78
	52	62.70	62.36	62.99	67.70	67.89	70.56	

(c) Similarity between two different family samples from the Trojan group

XTrojan.Win32		Avkillah		AVPatch		
		10	20	A	B	C
Avkillah	10		52.12	58.84	56.80	54.83
	20	52.90		54.11	56.63	53.58
AVPatch	A	63.88	57.65		64.34	64.53
	B	59.87	58.79	62.59		64.19
	C	59.49	57.15	64.52	66.11	

Table 7 – Similarity and detection success/failure statistics for groups.

	Max similarity	Min similarity	Success	Failure
(a) Statistics for virus group				
Method (Virus vs. Virus)				
N-gram	98.97	28.96	156	9
(Lee et al., 2010)	97.47	57.44	165	0
Proposed system	82.57	53.84	165	0
(b) Statistics for Trojan group				
Method (Trojan vs. Trojan)				
N-gram	87.71	18.46	94	23
(Lee et al., 2010)	95.88	34.72	109	8
Proposed system	76.74	38.69	111	6
(c) Statistics for Worm group				
Method (Worm vs. Worm)				
N-gram	93.97	16.46	119	15
(Lee et al., 2010)	97.81	41.36	130	4
Proposed system	82.25	45.65	130	4

Table 7(b) shows that the N-gram method fails to detect 23 Trojan samples, and the method in Lee et al. (2010) fails to detect 8 which is better than N-gram method, but worse than the proposed system which only fails to detect 6 samples. Table 7(c) shows that the N-gram method fails to detect 15 Worm samples (15 samples have similarity value less than threshold), Lee method and the proposed method fail to detect only 4 samples.

From the tables, the main observation is that virus samples are easier to detect compared to Trojan samples and worm samples. On the other hand, Trojan samples are harder to detect, because Trojans are inherently designed to appear similar to benign programs.

5.3.2.2. *Detection of malware variation.* For this experiment, the benign samples have been used to run three separate tests against the Virus, Worm, and Trojan groups. Results from the experiments that have been described previously are included in this experiment to help in terms of comparison between the different approaches. Table 8 illustrates the results of this experiment.

Table 8(a) shows that all comparable methods succeed in detecting all benign samples with false positive rate that equals 0. Table 8(b) shows the detection rate obtained from the three comparable methods. The proposed system and the method in Lee et al. (2010) have higher detection rates than N-gram in all malware groups. The proposed system achieved the highest detection rates in the Trojan group in particular.

In Table 8(c), the method proposed by Ahmed et al. (2009) is included in the comparison of the accuracy of methods since it uses the same dataset that the proposed system uses. The table shows that N-gram has the lowest accuracy in all malware groups. Moreover, the proposed system has the best accuracy in the Virus and Trojan groups, whereas the method proposed by Ahmed et al. (2009) has the best accuracy in the Worm group. A comparison of the overall performance of the systems is presented in Table 8(d).

Table 8 – Comparison between the proposed system and other comparable methods.

Methods	Virus	Trojan	Worm
(a) Comparison of false positive rate			
Method (False positive rate)			
N-gram	0	0	0
(Lee et al., 2010)	0	0	0
Proposed system	0	0	0
(b) Comparison of detection rate			
Method (Detection rate)			
N-gram	0.9455	0.8103	0.8881
(Lee et al., 2010)	1	0.9397	0.9701
Proposed system	1	0.9569	0.9701
(c) Comparison of accuracy			
Method (Accuracy)			
N-gram	0.9658	0.8930	0.9353
(Lee et al., 2010)	1	0.9628	0.9828
(Ahmed et al, 2009)	0.9900	0.9700	0.9840
Proposed system	1	0.9721	0.9828
(d) Comparison of overall performance			
Methods	Detection Rate	False alarm Rate	Accuracy
N-gram	0.8813	0	0.9086
(Lee et al., 2010)	0.9699	0	0.9767
(Ahmed et al, 2009)		–	0.9700
Proposed system	0.9757	0	0.9805
(e) Comparison of other measures			
Methods	Recall	Precision	f-measure
N-gram	0.8813	1	0.936905
(Lee et al., 2010)	0.9699	1	0.98472
(Ahmed et al, 2009)		–	
Proposed system	0.9757	1	0.987701

Other measures such as precision (i.e., (also called positive predictive value) the fraction of retrieved instances that are relevant), recall (i.e., (also known as sensitivity) the fraction of relevant instances that are retrieved) and f-measure (i.e., interpreted as a weighted average of the precision and recall, where an F-measure reaches its best value at 1 and worst score at 0), are shown in Table 8(e) with clear advantage for the proposed system over other methods.

Based on the above discussion, it is clear that call graph-based methods can deal with benign samples in a good way which is evident from the low false alarm rate shown by the related methods. Furthermore, the method proposed in Lee et al. (2010), which is basically a data dependant scheme, has better results than the N-gram method in sequential profiling scheme. Overall, the proposed system in this study shows better performance than the rest of the discussed methods.

Fig. 4 displays the ROC curves for the proposed method and the method in Lee et al. (2010) for the three abovementioned groups of tests, namely, Virus vs. benign, Trojan vs. benign, and Worm vs. benign. The curves display a better TP rate for the proposed method than the method in Lee et al. (2010) for the three groups. The corresponding AUC values of each curve are shown in Table 9.

Figs. 4 and 5 show the ROC curves of the proposed method, the method by Lee and the method by Ahmed et al. (2009) respectively. In Fig. 5, Ahmed et al. curve shows machine learning classification algorithms, instance based learner (IBk), decision tree (J48), Native Bayes (NB), inductive rule learner (RIPPER), and support vector machine (SMO). To detect malware, this method uses spatial and temporal features set with all classification algorithms and Native Bayes (NB) to get higher true positive (TP) rate than other classification algorithms.

Referring to the results on Worm group from Figs. 4 and 5, the false positive FP rate for Lee method starts with true positive (TP) rate value between 0.82 and 0.84 where the false positive FP rate for the proposed system starts at true positive (TP) rate value between 0.9 and 0.92. Furthermore, the proposed system reaches its highest true positive (TP) rate at 0.1 false positive FP rate, while Lee method reaches its highest true positive (TP) rate before false positive FP rate of 0.2. Clearly, the proposed system has the upper hand in this regard.

From Table 7(c) the minimum similarity for the proposed system (i.e., 45.65) is greater than the minimum similarity of Lee method (i.e., 41.36) where both methods fail to detect 4 Worm samples, which is a clear advantage for the proposed system to have very early true positive (TP) rate. Also, from Table 9 the difference between the areas under the two curve is 0.0195 which can be estimated by subtracting the AUC of Lee method (i.e., 0.9789) from AUC of the proposed system (i.e., 0.9984). A comparison between ROCs of the proposed system and Ahmed method (Native Bayes (NB)) for Worm group shows that for Ahmed method the true positive (TP) rate starts to move from some value between 0 and 0.05 false positive (FP) and reaches its highest true positive (TP) before false positive (FP) rate reaches 0.1. The proposed system has advantage since it starts to move earlier than Ahmed method, but reaches its highest true positive (TP) after 0.1 with advantage for Ahmed method (Native Bayes (NB)).

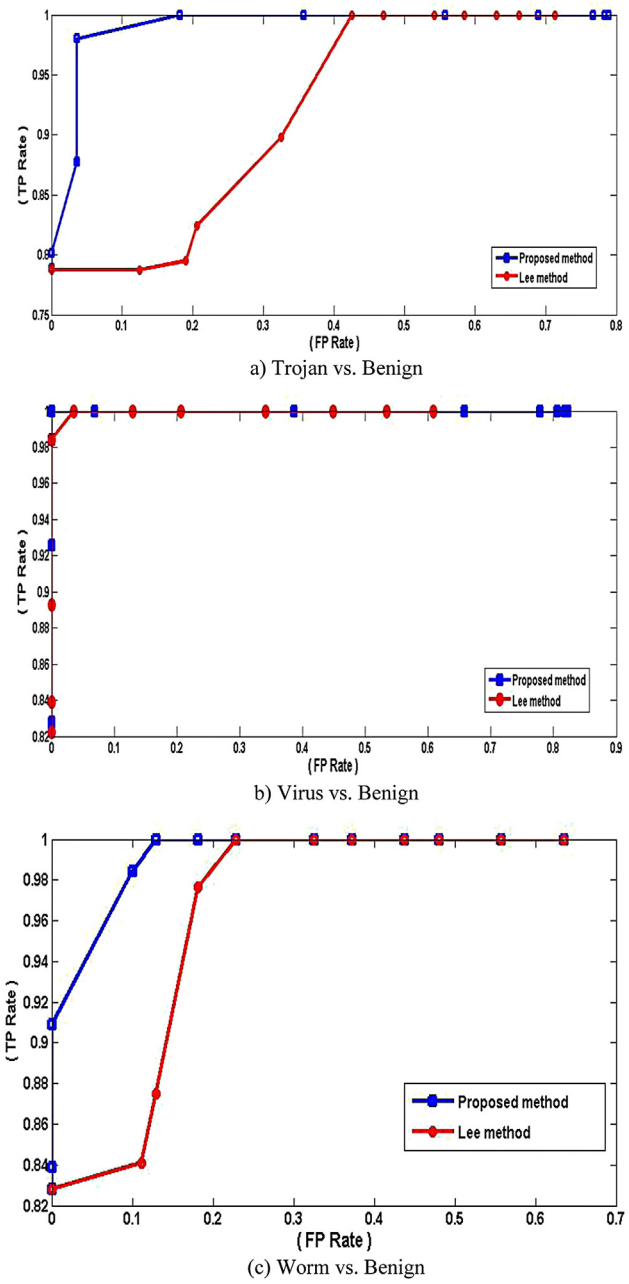
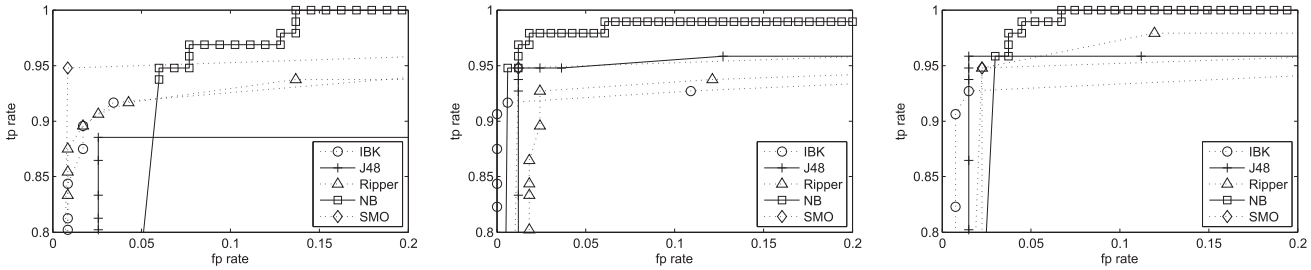


Fig. 4 – ROC plots for the proposed method and the method in Lee et al. (2010) for the 3 different test groups.

With regard to the Trojan group, the false positive FP rate for Lee method starts with true positive (TP) rate that is less than 0.8, whereas the false positive FP rate for the proposed system starts at 0.8. The proposed system reaches its highest

Table 9 – AUCs for the proposed method and Lee method.

Methods AUC	Virus	Trojan	Worm
(Lee et al., 2010)	1	0.9452	0.9789
Proposed system	1	0.9956	0.9984



(a) Trojan-Benign (Zoomed) (b) Virus-Benign (Zoomed) (c) Worm-Benign (Zoomed)

Fig. 5 – ROC plots for spatial and temporal features in API Calls with Machine Learning Algorithms.

true positive (TP) rate at 0.2 false positive FP rate, while Lee method reaches its highest true positive (TP) rate before 0.4 false positive FP rate, with advantage for the proposed system.

From Table 9, the difference between the areas under the two curves is 0.0504 which can be estimated by subtracting the AUC of Lee method (i.e., 0.9452) from AUC of the proposed system (i.e., 0.9956). The difference between the areas under the two curves for Worm group (i.e., 0.0195) shows that Worms are easier to detect that Trojans. Table 7(b) shows that the minimum similarity for the proposed system (i.e., 38.69) is greater than the minimum similarity of Lee method (i.e., 34.72). Moreover, the proposed system fails to detect 6 Trojan samples, which is better than the 8 samples Lee method fails to detect 8 samples. A comparison between ROCs of the proposed system and Ahmed method (Native Bayes (NB)) for Trojan group shows that for Ahmed method the true positive (TP) rate starts to move from 0.05 false positive (FP) and reaches its highest true positive (TP) before false positive (FP) rate reaches 0.115. The proposed system has advantage since it starts to move earlier than Ahmed method, but both methods reach their highest true positive (TP) rate before the false positive rate reaches 0.2.

As for the Virus group, both curves of the proposed system and Lee method show high true positive (TP) rate compared with false positive FP rate, with a little advantage for the proposed system.

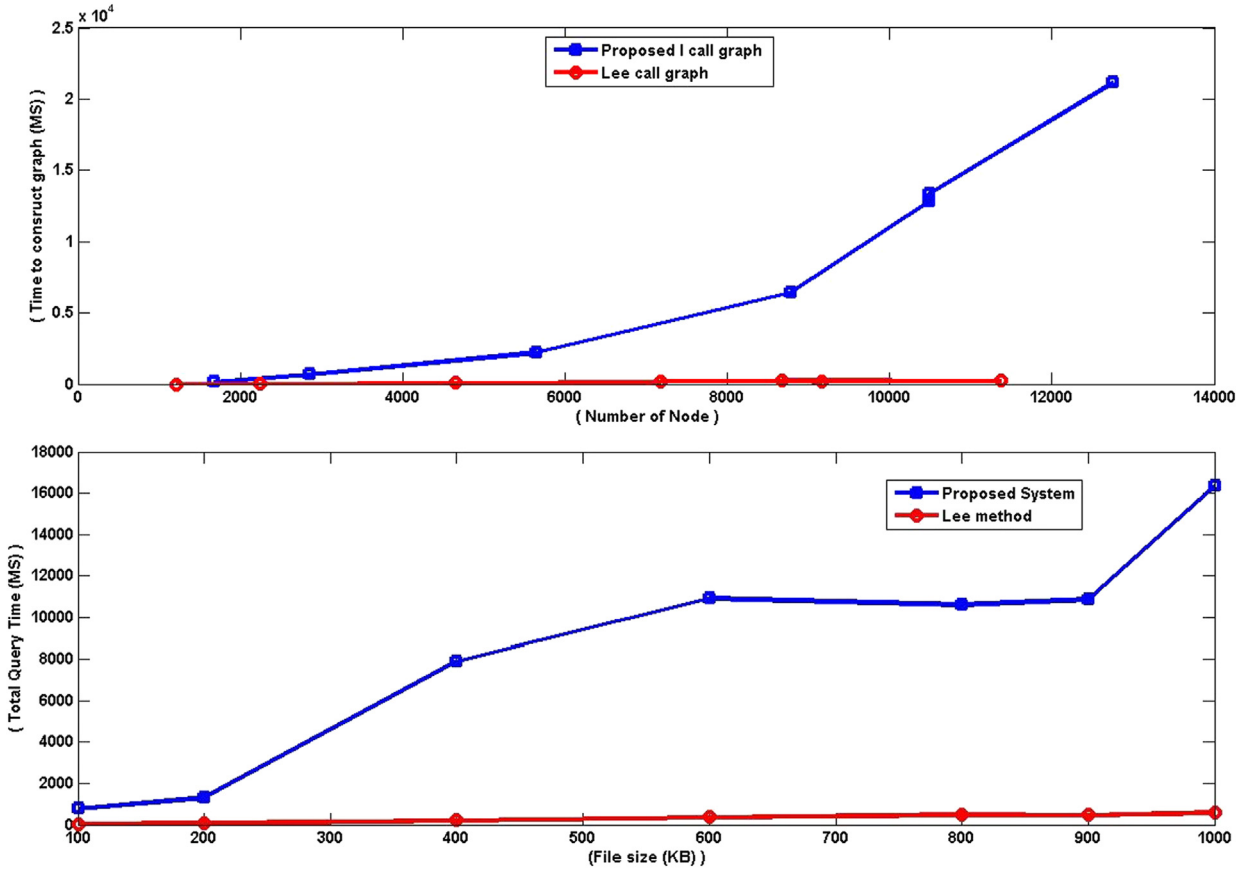


Fig. 6 – Performance of the proposed system.

From Table 7(a) the minimum similarity for the proposed system (i.e., 53.83) is less than the minimum similarity of Lee method (i.e., 57.44). Both methods succeed to identify all Virus samples and reach their highest true positive (TP) before false positive FP rate reaches 0.1. Table 9 shows that the difference between the areas under the two curves is 0 which can be estimated by subtracting the AUC of Lee method from AUC of the proposed system and both AUCs have the value 1. Compared to other groups, the above statistic shows that the Virus group is the easiest to detect. A comparison between ROCs of the proposed system and Ahmed method (Native Bayes (NB)) for Virus group shows that for Ahmed method the true positive (TP) rate starts to move from some point between the value 0 and 0.05 false positive (FP) and does not reach the highest true positive (TP) rate. The proposed system has advantage over Ahmed method because it detects all viruses with 0 false positive FP and maximum true positive (TP) rate.

6. Limitations

Analysing the performance of the proposed system shows that it takes longer time compared to Lee method in terms of graph construction and matching processes, which can be attributed to the bigger and more detailed integrating API call graph the proposed method produces. Fig. 6 shows the time needed to construct the call graph and perform the matching process for both the proposed system and Lee method.

The proposed system assumes that the query malware sample belongs to the same family when it has the best subgraph with high similarity. This assumption is not always true since the distance between an input malware sample and its dominant family may be large, because malware writers could use very efficient tools that perform advanced obfuscation techniques that detract from the proposed system overall effectiveness.

Moreover, there are several limitations with dynamic analysis. First, dynamic tools cannot explore all execution paths in malware samples and may omit some paths which contain some important API calls. Second, the extracted API call list and its parameters may be incorrect when a hacker packs the malware samples and applies polymorphic techniques.

The proposed system in this study has been evaluated and tested using a small and old dataset, since there is no standard dataset and researchers in this area evaluate and test their models using their own malware datasets trying different assessment methods. Therefore, the experiments in this study could be repeated on a different and bigger dataset. The proposed system uses the principle that if two programs are similar, then they have the same behaviour. Therefore, a labelled malware sample is needed to detect unknown samples with the same behaviour.

A final area of further research would be to explore the value of similarity between operating system resources, since malware samples can use different operating system resource names to perform the same attack.

7. Conclusion

This study has proposed a malware detection system based on API call graph and graph matching algorithm. The proposed system enhances the current API call graph construction algorithm by integrating API call and operating system resource as graph node, and using four types of dependence between the nodes to represent the edges. Graph matching algorithms suffer from computational complexity due to NP-Complete problem and the large size of integrating API call graphs. Therefore, this study has proposed an approximate algorithm for the computation of graph edit distance that divides the API call graph into subgraphs based on structure and attribute. Analysis has shown the correctness of the algorithm and its reasonable running time. The proposed system has been implemented using Delphi programming language. Experiments show that the proposed system has 98% accuracy and 0 false positive rates, which reflects significant improvement over the previous attempts.

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 & Technology (MCST) for the support and incisive comments in making this study a success.

REFERENCES

Ahmed F, Hameed H, Shafiq MZ, Farooq M. Using spatio-temporal information in API calls with machine learning algorithms for malware detection. In: Proceedings of the 2nd ACM workshop on Security and Artificial Intelligence. ACM; 2009.

Bai L, Pang J, Zhang Y, Fu W, Zhu J. Detecting malicious behavior using critical api-calling graph matching. In: Information Science and Engineering (ICISE), 2009 1st International Conference on, 2009. IEEE; 2009. p. 1716–9.

Bhat SA, Singh J. A practical and comparative study of call graph construction algorithms; 2012.

Bonfante G, Kaczmarek M, Marion J-Y. Control flow graphs as malware signatures. In: International workshop on the Theory of Computer Viruses; 2007.

Cesare S, Xiang Y. Malware variant detection using similarity search over sets of control flow graphs. In: Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on. IEEE; 2011.

Christodorescu M, Jha S, Kruegel C. Mining specifications of malicious behavior. In: Proceedings of the 1st India Software Engineering Conference. ACM; 2008.

Conte D, Foggia P, Vento M. Challenging complexity of maximum common subgraph detection algorithms: a performance analysis of three algorithms on a wide database of graphs. J Graph Algorithms Appl 2007;11(1):99–143.

Corporation S. Internet security threat report; 2013.

Elhadi AAE, Maarof MA, Osman AH. Malware detection based on hybrid signature behaviour application programming interface call graph. Am J Appl Sci 2012;9(3):283–8.

Eskandari M, Hashemi S. A graph mining approach for detecting unknown malwares. J Vis Lang Comput 2012;23:154–62.

Fredrikson M, Jha S, Christodorescu M, Sailer R, Yan X. Synthesizing near-optimal malware specifications from suspicious behaviors. In: Security and Privacy (SP), 2010 IEEE Symposium on. IEEE; 2010. p. 45–60.
Gao X, Xiao B, Tao D, Li X. Image categorization: graph edit distance+ edge direction histogram. Pattern Recognit 2008;41(10):3179–91.
Gao X, Xiao B, Tao D, Li X. A survey of graph edit distance. Pattern Analysis Appl 2010;13(1):113–29.
Garey MR, Johnson DS, Stockmeyer L. Some simplified NP-complete graph problems. Theor Comput Sci 1976;1(3):237–67.
Guo H, Pang J, Zhang Y, Yue F, Zhaok R. HERO: a novel malware detection framework based on binary translation; 2010. p. 411–5.
Han KS, Kim IK, Im EG. Malware classification methods using API sequence characteristics. In: Proceedings of the International Conference on IT Convergence and Security 2011. Springer; 2012.
Haoran G, Jianmin P, Yichi Z, Feng Y, Rongcai Z. HERO: a novel malware detection framework based on binary translation. In: International Conference on Intelligent Computing and Intelligent Systems (ICIS), 2010 IEEE; 29–31 Oct. 2010. p. 411–5.
Harley D. Making sense of anti-malware comparative testing. Inf Secur Tech Rep 2009;14(1):7–15.
Hu X, Chiueh TC, Shin KG. Large-scale malware Indexing using function-call graphs. In: Ccs'09: Proceedings of the 16th Acm Conference on Computer and Communications Security; 2009. p. 611–20.
Kim K, Moon B-R. Malware detection based on dependency graph using hybrid genetic algorithm. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation. ACM; 2010a.
Kim K, Moon BR. Malware detection based on dependency graph using hybrid genetic algorithm; 2010.
Kinable J, Kostakis O. Malware classification based on call graph clustering. J Comput Virology 2011:1–13.
Kolbitsch C, Comparetti PM, Kruegel C, Kirda E, Zhou X, Wang X. Effective and efficient malware detection at the end host. In: Proceedings of the 18th Conference on USENIX Security Symposium. USENIX Association.; 2009. p. 351–66.
Kostakis O, Kinable J, Mahmoudi H, Mustonen K. Improved call graph comparison using simulated annealing. In: Proceedings of the 2011 ACM Symposium on Applied Computing. ACM; 2011. p. 1516–23.
Lakhotia A. Constructing call multigraphs using dependence graphs. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages. ACM; 1993.
Lee J, Jeong K, Lee H. Detecting metamorphic malwares using code graphs. In: Proceedings of the 2010 ACM Symposium on Applied Computing. ACM; 2010.
Michael RG, Johnson DS. Computers and intractability: a guide to the theory of NP-completeness. San Francisco: WH Freeman & Co; 1979.
Microsoft. MSDN library; 2012 [cited 2012 1-February-2012]; Available from, http://msdn.microsoft.com/library/default.aspx.
Monitor A. Spy and display API calls made by Win32 applications; 2012. Available from, http://www.apimonitor.com.
Park Y, Reeves D, Mulukutla V, Sundaravel B. Fast malware classification by automated behavioral graph matching. In: Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research. ACM; 2010. p. 45.
Park Y, Reeves D. Deriving common malware behavior through graph clustering. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM; 2011.
Pro I. Interactive DisAssembler; 2012. Available from, http://www.hex-rays.com/.
Raymond JW, Willett P. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. J computer-aided Mol Des 2002;16(7):521–33.
Rieck K, Trinius P, Willems C, Holz T. Automatic analysis of malware behavior using machine learning. J Comput Secur 2011;19(4):639–68.
Riesen K, Bunke H. Approximate graph edit distance computation by means of bipartite graph matching. Image Vis Comput 2009;27(7):950–9.
Riesen K, Jiang X, Bunke H. Exact and inexact graph matching: methodology and applications. Manag Min Graph Data 2010:217–47.
Ryder BG. Constructing the call graph of a program. Softw Eng IEEE Transactions 1979;(3):216–26.
Santos I, Brezo F, Ugarte-pedrero X, Bringas PG. Opcode sequences as representation of executables for data-mining-based unknown malware detection. Inf Sci 2011;231:64–82.
Schultz MG, Eskin E, Zadok E, Stolfo SJ. Data mining methods for detection of new malicious executables. In: sp; 2001. p. 0038.
Shahzad F, Shahzad M, Farooq M. In-execution dynamic malware analysis and detection by mining information in process control blocks of Linux OS. Inf Sci 2011;231:45–63.
Weskamp N, Hullermeier E, Kuhn D, Klebe G. Multiple graph alignment for the structural analysis of protein active sites. Transactions Comput Biology Bioinforma IEEE/ACM 2007;4(2):310–20.
Xu M, Wu L, Qi S, Xu J, Zhang H, Ren Y, Zheng N. A similarity metric method of obfuscated malware using function-call graph. J Comput Virology Hacking Tech 2013;9(1):35–47.
Zeng Z, Tung AKH, Wang J, Feng J, Zhou L. Comparing stars: on approximating graph edit distance. Proc VLDB Endow 2009;2(1):25–36.
Zhao Z, Wang J, Wang C. An unknown malware detection scheme based on the features of graph. Secur Commun Networks 2012;6:239–46.
Zhu L, Keong Ng W, Cheng J. Structure and attribute index for approximate graph matching in large graphs. Inf Syst 2011;36:958–72.

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.

Hentabli Hamza received his B.Sc. degree in computer science from the University Saâd Dahlab in 2003, Blida, Algeria. He is currently working on his M.S. in Universiti Teknologi Malaysia (UTM) the area of Chemioinformatics, Information retrieval and Artificial intelligence.