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

A R T I C L E I N F O

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

A B S T R A C T

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/© 2014 Elsevier Ltd. All rights reserved.

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