A Survey on Algorithmic Techniques for Malware Detection

Ioannis Chionis Stavros D. Nikolopoulos Iosif Polenakis

Department of Computer Science and Engineering, University of Ioannina
GR-45110 Ioannina, Greece
{ichionis, stavros, ipolenak}@cs.uoi.gr

***Abstract*—Malware is a specific type of software intended to breed damages ranging from computer systems fallout to deprivation of data integrity and confidentiality. Recently, along with the high usage of distributed systems and the increasing speed in telecommunications, the early detection of malware constitutes one of the major concerns in information society. A strong advantage that malware employs in order to elude detection is the ability of polymorphism (metamorphic or polymorphic engines). In this work we present efficient algorithmic techniques that, leveraging higher level abstractions of malware structure, perform an isomorphism check in malware's produced graph structures, such as function call-graphs and control flow-graphs, in order to detect every possible polymorphic version of a malware. Moreover, we propose an algorithmic approach for malware detection which focuses on the use of behavioural graphs as a more flexible representation of malware's functionality with respect to its interaction with the operating system. The main idea of our approach is mainly based on behavioural graph similarity issues.**

I. Introduction

Malicious software has become one of the most major concerns for the security of computer systems. Starting in early 80's, malware constitutes a significant threat, so for the integrity and availability of computer systems as for the confidentiality, integrity and availability of data stored into these systems. During the last ten years, it is observed that the implementation ability has become more sophisticated, including extremely elaborated techniques for detection avoidance.

Most Antivirus software employ traditional methods of static analysis, in order to ascertain if a software is malicious or benign. The most used technique for malware detection is based on signatures, treating malware as sequence of bytes and trying to detect specific sequences (patterns) that uniquely identify a specific malware. Methods like the afore-mentioned one, even though they are fast, providing realtime protection, on the other hand they are liable against polymorphism. The reason is that malware is written in high level programming languages, so a slight change in their source code leads to significant changes in their binary file. As a result, the sequence of bytes that could be used as their signature changes, making them unidentifiable from Antivirus Programs that use signatures as detection method.

Related Work. In literature, systems have been proposed that detect new variants of existing known malware by using function call-graphs or control flow-graphs. Specifically in [7], [4] a significant effort to check graph or subgraph similarity has been done. It is well known that graph matching requires a solution to the graph isomorphism problem, that is not a polynomially solvable problem. So, they have designed and implemented metrics [4] and indexing techniques [7] that can approach the solution in a more efficient manner.

Specifically, in [4] there have been proposed metrics that use normalized weight of procedures and the Dice coefficient for the measure of similarity between two sets of procedures. In [7] an alternative technique is proposed that, by leveraging the Hungarian Algorithm, performs graph-matching to approximate the graph-edit distance between two graphs.

Motivation. The major challenge that constitutes our motivation for this work, is a procedure called polymorphism. With this ability, malware authors are in position to mutate a basic verision of a known malware and produce variants that can evade the detection from common Antivirus softwares that deploy traditional detection techniques such string signatures and byte level detection in general.

Our main target is the description of systems that use efficient algorithms in the stage of sub-graph matching between function call-graphs or control flow-graphs. For this purpose, we have collected a set of such systems in order to construct and adduce a "general approach", meaning a system that assembles components from different approaches. This, we expect that will result in the creation of a more global and integrated point of view for the systems proposed for malware detection using control flow-graphs or function call-graphs.

Additionally, in the last section, we propose an alternative approach for a common system that detects a known malware's variants. Specifically, we plan to leverage behavioural graphs produced from dynamic malware analysis, instead from those produce from static malware analysis. Having the intuition that such graphs, since represent the functionality of a malicious software, will perform better and will be more flexible in malware's mutations, we focus on substitute the function call-graphs and the control flow-graphs with behavioural graphs.

Analysis. There are two types of malware analysis according to if execution of malware takes place or not. The two basic types of malware analysis are the static and the dynamic malware analysis [11]. The main result of both analyses that

we leverage, is that both can produce graphs as representations of malware's structure.

The first type of malware analysis is the static malware analysis. In static analysis, the malware is never executed. Instead, involving the use of dissembling, file hashing and packing detection, is trying to examine the binary code to detect specific patterns that are similar into a known (classified) malware. Some of the most common methods for static analysis include string signature detection, byte N-grams, entropy analysis, syntactic library call and control flow graph matching.

The second type of malware analysis is the dynamic malware analysis. In dynamic analysis the malware is executed in an isolated (possibly virtual) environment and its runtime behaviour is examined. The most examined source is the operating system and the interaction it has with the malware. In other words dynamic analysis examines the effects that malware bring to the operating system. Analyzing the executed code and examining its behaviour, then behavioural graphs can be constructed that are more flexible in presenting an abstraction of malware's structure, providing us with a more effective method in malware detection.

Polymorphism and Detection. One of the most elaborated techniques for malware, to evade detection is called polymorphism. Using this technique, either automated using polymorphic engines or by themselves, malware authors having only one version of their malware are creating new versions of it. The new versions, have the same basic functionality inherited by their ancestor, but with the difference that these versions have either an extra functionality added (extension) or some non-functional additions in their source code. Mutations of code [11] can include "control flow preserving transformations" inserting instructions that do not change the data/control flow of malware, "dead code insertion" where code is inserted in malware basic code, but is never executed, and finally "equivalent code substitution" where instructions in malware's source code are replaced from equivalent ones.

As one can observe the traditional techniques are ineffective to detect any new version of a known malware because of polymorphism. This, means that we need to observe the high-level structure of malware and proceed with detection in a more abstract manner. In other words, abstractions of malware structure, provide us with a more flexible method to detect malware, even if it has passed over a polymorphism process. These techniques for malware detection include comparison of control flow-graphs (CFGs) or function call-graphs (FCGs) [3]. Flow-graphs represent the internal control flow of a procedure, while call-graphs represent the inter-procedural control flow.

II. Main Concept

The whole process of detecting a new version/variant of a known malware, as proposed in literature, is in general the same. The main concept in the use of control flow-graphs or function call-graphs, is that since this type of graphs act as abstractions of malware's structure will be invariant (or at least some parts of them) after a polymorphism process. In other words, what has been proposed leverages the main property of polymorphism, where the basic functionality of malware will be "similar" after the mutation of code.

In the first phase, it is needed to check if the malware (test sample) that we want to classify is packed. A packed malware is actually a compressed malware (using softwares called packers), that malware authors constructed in order to evade the traditional signature based detection employed by common Antivirus softwares. If so, then, specific softwares are used to unpack the malware as many times needed (multilevel packing), in order to get the original malware. After that, the process continues with the transformation of binary in a higher level language by disassembling it, using common disassemblers like IDA Pro [7], [5].

In the second phase, the process continues, producing a higher level abstraction of malware's structure, such as a control flow-graph or a function call-graph. To this point, we offer to explain that the type of the graph, used to supply to the next stage of the whole process, depends firstly on the available software for this purpose and secondly on the experimental notion on what type of graph (CFG, FCG) offers the most generalization ability.

The third phase is the most complicate. In this phase, beggins the graph matching procedure, where the state of the art proposed systems differentiate themselves from each other. Specifically, there have been proposed systems, that in the graph matching process, search for the maximum common subgraph like binHunt [4], [6], or search for common subgraphs of size k, like the technique proposed in [9]. Additionally, there are other approaches that use more elaborated methods to approach the graph matching. For example, in [4] a set of metrices have been proposed to measure the weights of each procedure, extracted from a binary file and compute a coefficient with these wheights, from known malwares. Finally in [7], there have been proposed efficient indexing techniques for quick binary classification, based on a nearest-neighbor search in the supporting database.

III. A Typical System Design

In this section we present a typical system which combines characteristic from different proposed systems so far in the literature. In fact, we have collected characteristics from a set of such systems in order to construct and adduce a "general approach" system, meaning a system that assembles components from different approaches. A typical system consists of the phase of *training* and the phase of *detection*. Both phases include a pre-process procedure to transform every object (sample) in a "proper" form. The two phases differentiate at the point after the generation of all possible subgraphs of the produced malware's graph (Procedure B; see Figure 1). In other words, it means that given a graph G as a result of the pre-process transformation procedure, the generated subgraphs of G constitute signatures of the known malware.

A. Preprocessing and Transformation

The pre-process transformation procedure starts by accepting as input a sample (known malware) and proceeds by checking if the current sample is packed or not. If so, it repeats as many times as needed the unpacking process as described in [7]. Then, the result is passed as input to a disassembler in order to transform the binary in a higher level

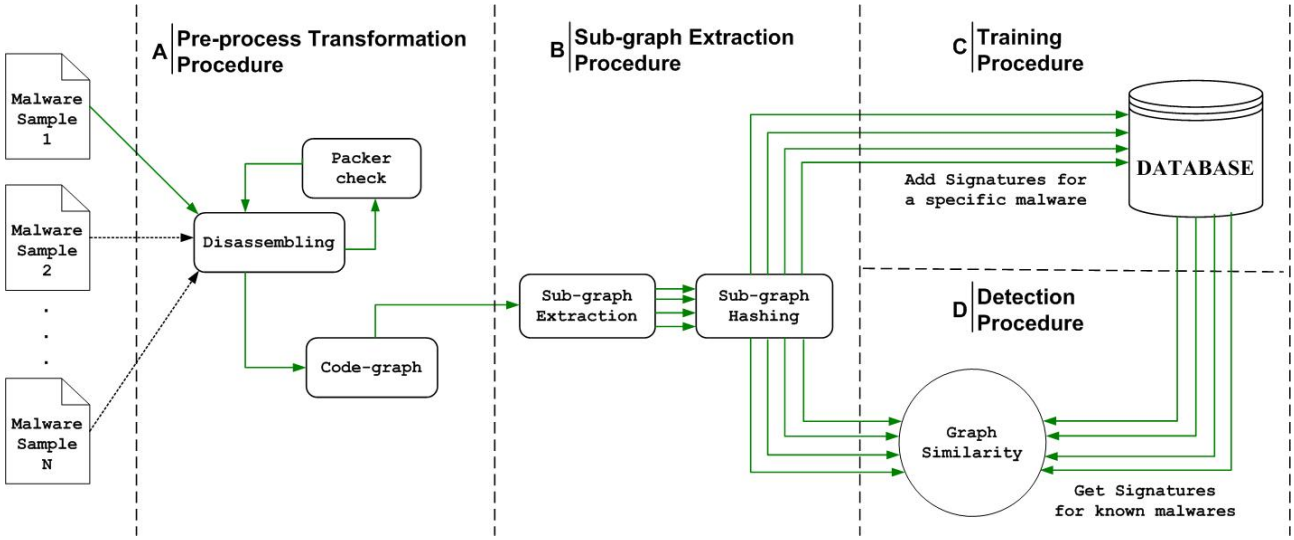

Fig. 1. An overview of a typical malware detection system.

language. Finally, a function call-graph or a control flow-graph is produced by performing static malware analysis.

B. Subgraph Extraction

Having designed only an abstraction (graph representation) of malware's structure, what we next need to do is to extract a type of "possible signatures". Like the traditional byte-level signatures, the possible signatures for such an abstraction are parts of it; in our case, these parts are subgraphs of the produced graph. To continue with this process, a DFS is performed in order to construct a meaningful string representation of the graph, as proposed in [4].

In contrast with the techniques proposed in [9], it could be used a subgraph of size k of malware's control flow-graph or function call-graph, as malware's signature like in the technique proposed in [4]. According this approach, it could be kept every subgraph of size k, meaning that multiple signatures for a specific malware could be produced in order to capture its variants. However, as referred in [4], it does not demonstrate to perform efficiently for end-host use and does not employ a malware database.

Finally, this subgraph can be represented as a table with rows as edges and columns as vertices; this table is hashed using a collision resistant hash function such as SHA-256 or CRC64 as proposed in [4] and inserted into the supporting database as signature, together with sample's malware name for later identity information provision.

C. Training

In the first phase, a training process is taking place collecting an amount of known malware sample in order to be able to compare a test sample with each one of them. Having already collected a sufficient amount of known malware samples, the pre-process transformation procedure is performed, as depicted in Figure 1.

Once the pre-process transformation procedure is completed, we have a higher level abstraction (graph representation) of malware's structure and, then, the subgraph extraction process is performed. Once the subgraph extraction process is completed (every time it produces a possible subgraph), a signature is inserted in the supporting database constructing finally a set of signatures for the current sample.

After the completeness of the training process, a database keeping sets of signatures for each malware has been constructed. In other words, signatures have been produced for every possible variant of every malware belonging to the sample set. To this point, we have to make it clear that the process of training takes place only once (every time we have a new sample we need to repeat the process only for this sample).

D. Detection

In the second phase, the detection process takes place every time a new sample (test sample) needs to be classified. Until the completion of the extraction process, the main process remains the same as in the training phase beginning by checking for packing and executing the whole pre-process transformation procedure producing the main graph for this malware. In the sequence, the process continues with the subgraph extraction procedure, producing signatures for the test sample.

Finally, having all the possible subgraphs for this sample, the detection procedure is differentiated from the training procedure. In this state, no signatures are inserted in the database, instead every signature of the produced ones is compared with every one in the set of signatures of every malware in the database.

IV. Our approach

In order to capture the run-time behaviour of an executed malware, we propose a structural modification of the

previously described system for malware detection by substituting the function call-graphs and control flow-graphs with behavioural graphs.

This approach leverages the dynamic analysis, which seems to be more flexible in polymorphism, since we need to investigate the main functionality and the interaction between the operating system and a malware [2]. Additionally, behavioural graphs have a more convenient labeling made up of event names [8], that simplifies the subgraph similarity check.

More precisely, our system's design is based on the typical system that we described in Section III. We point out that in the system we propose we by-pass the pre-processing transformation procedure because, since we perform dynamic analysis, we need to capture only the malware's behaviour and not its internal structure.

A. Dynamic Analysis

In our approach, we perform dynamic analysis in order to extract information about the interaction between the malware and the operating system. Specifically, executing the analysis in a virtual environment we examine the interaction between the malware and the guest operating system ensuring thus the security of our host operating system. For the purpose of dynamic malware analysis, we deploy specific on-line tools, such as ANUBIS [1], that provide us with all the information needed.

The information that is crucial for the construction of the behavioural graph, as proposed in [8], includes events like the existence of running services or processes, file events, registry key and registry value alterations, dynamic link library events (or, for sort, DLL events), API calls and network activity monitoring. However, we do not exclude the case of "function-call hooking" by intercepting in function calls.

B. Graph Construction

A behavioural graph is a type of graph that represents specific events as vertices and their in between relations as edges. Thus, after the completeness of dynamic analysis we proceed to the behavioural graph construction. Compiling all this event information and their in between relations from the phase of dynamic analysis, we can transform them into a directed graph that actually represents the malware's behaviour. Once the behavioural graph is constructed, we store it in a supporting database in order to be used later for the detection process.

C. Graph Matching

As we mentioned in the beginning of this section, in the phase of detection we faithfully follow the typical system presented in Section III by performing graph similarity check. In our approach we check for similarities between behavioural graphs produced by a known malware and a test sample. To this end, we need to design algorithms that leveraging the property of specific node labeling in behavioural graphs, as their nodes are discrete events, which will try to detect similar subgraphs of a specific size. In our approache we check for sub-graph similarity matching between behavioural graphs that actually capture the malware's functionality with respect to its behaviour while, on the other hand, the previously proposed approaches use control flow-graphs or function call-graphs representing only malware's internal structure.

Additionally, we can optimize the sub-graph similarity matching process based on anomalies during the execution of a malware, such as abuse of specific resources and unusual parameters to specific functions or network protocols. This optimization process could be implemented by searching only for a subgraph, in the test sample's behavioural graph, that includes specific nodes (events). This, implies that we do not need to search the whole behavioural graph for matchings but only in the neighbours of a specific node (event), reducing in this way the time needed for the similarity matching process.

Finally, since the nodes of the behavioural graph are proper labeled, one alternative solution which as far as we know has not been proposed yet, is that of leveraging subgraph coloring in order to perform subgraph similarity by identifying nodes correlated to the same label (event) and also to the same color in two behavioural graphs. This, implies that the coloring procedure must start from the same node in both graphs, meaning a specific event. Even though this solution is in preliminary stage yet it will be one of our main concerns for our forthcoming research.

REFERENCES

[1] Anubis: *Analyzing Unknown Binaries*. http://anubis.iseclab.org.

[2] A. Barthels, Bahavior-based Malware Detection, Det Technischen Universität München, Fakultät Für Informatik, Master Thesis, 2009.

[3] G. Bonfante, M. Kaczmarek, and J-Y. Marion, "Control flow graphs as malware signatures," Int'l Workshop on the Theory of Computer Viruses, Nancy France, 2007.

[4] S. Cesare and Y. Xiang, "A fast flowgraph based classification system for packed and polymorphic malware on the endhost," Proc. 24th IEEE Int'l Conference on Advanced Information Networking and Applications, pp. 721–728, 2010.

[5] C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*, San Francisco, No Starch Press Inc., 2008.

[6] D. Gao, M.K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," Information and Communications Security, Springer Berlin Heidelberg, pp. 238–255, 2008.

[7] X. Hu, T-C. Chiueh, and K.G. Shin, "Large-scale malware indexing using function-call graphs," Proc. 16th ACM conference on Computer and Communications Security (CCS'09), pp. 611–620, 2009.

[8] G. Jacob, H. Debar, and E. Filiol, "Behavioral detection of malware: from a survey towards an established taxonomy," Journal in Computer Virology 4(3), pp. 251–266, DOI 10.1007/s11416-008-0086-0, 2008.

[9] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," Proc. Rapid Advances in Intrusion Detection (RAID'06), Springer Berlin Heidelberg, pp. 207–226, 2006.

[10] M. Mungale and M. Stamp, "Software similarity and metamorphic detection," 11th Int'l Conference on Security and Management (SAM'12), Las Vegas, 2012.

[11] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, No Starch Press Inc., 2012.