Detecting Metamorphic Malwares using Code Graphs

Jusuk Lee, Kyoochang Jeong, and Heejo Lee
Div. of Computer & Communication Engineering
Korea University
Seoul, korea
{jupiter7, heejo}@korea.ac.kr

ABSTRACT

Malware writers and detectors have been running an endless battle. Self-defense is the weapon most malware writers prepare against malware detectors. Malware writers have tried to evade the improved detection techniques of anti-virus(AV) products. Packing and code obfuscation are two popular evasion techniques. When these techniques are applied to malwares, they are able to change their instruction sequence while maintaining their intended function. We propose a detection mechanism defeating these self-defense techniques to improve malware detection. Since an obfuscated malware is able to change the syntax of its code while preserving its semantics, the proposed mechanism uses the semantic invariant. We convert the API call sequence of the malware into a graph, commonly known as a call graph, to extract the semantic of the malware. The call graph can be reduced to a code graph used for semantic signatures of the proposed mechanism. We show that the code graph can represent the characteristics of a program exactly and uniquely. Next, we evaluate the proposed mechanism by experiment. The mechanism has an 91% detection ratio of real-world malwares and detects 300 metamorphic malwares that can evade AV scanners. In this paper, we show how to analyze malwares by extracting program semantics using static analysis. It is shown that the proposed mechanism provides a high possibility of detecting malwares even when they attempt self-protection.

Keywords

Code graph, metamorphic malware, static analysis, code obfuscation

1. INTRODUCTION

Malware is software designed to infiltrate or damage a computer system without the owner's informed consent. Examples of such malware include viruses, worms, Trojans and bots. Such malware threatens computer security, because it exhausts system/network resources or infringes on a person's privacy. Although, there is much effort to detect malware, malware writers achieve their intention by thwarting the efforts of malware detectors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'10 March 22-26, 2010, Sierre, Switzerland.
Copyright 2010 ACM 978-1-60558-638-0/10/03 ...$10.00.

The major difficulty of malware detection is the dramatic increase in malware. According to a Symantec report, malware signatures have increased extremely over recent five years from 18K to 3M. Of course, malware writers have improved their malware writing skill. However, the main reason for the rapid increase is that there is too much malware since the malware variants can be produced easily using code obfuscation techniques. Malware writers are able to create numerous malware variants, using code obfuscation tools, from malware sample code. Obfuscation refers to techniques that preserve the program's semantics and functionality, while simultaneously making it more difficult for the analyst to extract and comprehend the program's structure. The cost due to code obfuscation is inexpensive, since obfuscated malware is able to evade AV scanners, much malware is generated by code obfuscation. According to the recent study, more than 80% of malware uses packing to hide from malware detectors [16]. Although packing is a form of code obfuscation techniques, it is usually classified separately due to its fame. In addition, recent research shows that AV scanners have difficulty in detecting malware -applied bit obfuscation, the common technique used by malware writers to evade detection is program obfuscation [20]. Moreover, there are numerous examples of obfuscation techniques designed to avoid detection [3, 11, 14, 17, 22].

Malware detection using signatures has been used for malware detection. It is known to be efficient in known malware detection. As compared with dynamic analysis using sandbox, signature based analysis has less scanning time because of small overhead, has few false-positives and needs not to worry about system infection by malware. However, signature based analysis has a fatal disadvantage. Unknown malware can easily evade detection. Moreover, signature based analysis cannot deal with simple obfuscation such as binary pattern modulation. Christodorescu and Jha [6] pointed out that such detection methods can be easily defeated by the use of the principle of metamorphism. Metamorphism uses code obfuscation techniques to transform the representation of programs. The number of malware signatures has increased explosively, due to the rapid increase of the amount of malicious code. Therefore, existing signature based detection mechanisms have become inefficient in terms of time and space complexity. One way to decrease the number of signatures is the use of normalization techniques that have been actively studied recently [8, 28]. However, this approach does not provide a complete solution to detect unknown malware. Theoretical studies on malware detection have revealed that there is no algorithm that can detect all types of malware. [9, 10]

There are two Problems for signature based detection. First, signature based systems cannot detect unknown malware. Second, as malware's growth is exponential, the number of malware signatures

increases alarmingly. To solve these problems, a malware detection mechanism is needed to detect unknown malware with few signatures. To address this, we propose a malware detection mechanism using the malware's semantic rather than its syntax.

This is based on the assumption that a malware M contains the API call sequence S and its variant, M' is generated from M, then S and S', derived from M', have very high similarity. That is, when M is obfuscated to M', the syntax of M' looks different from the syntax of M but the semantics of M' cannot deviated from the semantics of M. Programs (including malware) have to behave to perform a certain function; code obfuscation cannot affect on program behavior. Thus, we are to construct malware signatures with the semantic characteristics of the malware. First, if the malware is packed, an unpacking process is performed. Next, we represent the unpacked malware as a call graph. The call graph has hundreds or thousands of nodes and graph isomorphism is known to be NP-complete. It is inefficient to use the call graph as the semantic signatures of malware. Therefore, we convert the call graph to another form of graph based on the feature of the API call. We call this other form of graph, a code graph. The code graph is used to construct semantic signatures of known malware. We extract code graphs from malware. These code graphs are stored. The code graph is extracted from input program. This code graph is compare to previously stored code graphs to measure similarity. This similarity is used to determine if the input program is benign or malicious.

This paper is an extension of the previous work [13], where the concept of code graph was proposed for finding malwares. Code graphs are an effective graph structure which represents a complex binary code into a single graph. In this paper, we apply code graphs to find metamorphic malwares.

The proposed mechanism has been implemented and evaluated on malware variants. We collected 3270 malware programs obtained from `offensiveComputing.net` and VX heaven. We found 100 variants and measured their similarities. We detected 91 variants of the 100 variants. In addition, we created metamorphic malwares. From 10 malware instances, we generated 30 versions of malware per malware using code obfuscation. We develop some obfuscation programs to generate metamorphic malware. The programs generate metamorphic malware using code insertion, code reordering, and code replacement. We generated total 300 obfuscated malware programs using these code obfuscation tools. The obfuscated malware generated by us can evade existing AV scanners but our mechanism detected all of them.

The three main contributions of our paper are the following.

- In order to detect metamorphic malwares, we propose an approach to define semantic signatures rather than syntactic signatures. We tried to solve an existing malware detection issue, by describing a program with its semantic signature. This contribution makes it possible to detect malware even though the malware uses self-defense.

- We demonstrate a method to make up for the weakness in malware detection with static analysis. This gives us the strength of static analysis, while complementing the drawbacks of static analysis, which is poor for unknown malware.

- We reduce the number of malware signatures. With this contribution, we need not worry about signatures for exponentially increasing malware. In addition, this technology requires smaller space and greatly reduces scanning time. This contribution has great significance, since analysis time is an important element in malware detection and the amount of malware has been increasing dramatically.

The remainder of paper is organized follows. In section 2, we discuss related work. System architecture is described in section 3. The system is composed of code analyzer, code graph generator, and graph analyzer. We give a detailed explanation of each component in this section. Section 4 evaluates our system. Finally, we conclude with future work in section 5.

2. RELATED WORKS

While malware detection has been studied for decades, code obfuscation has attracted recent attention since malware is able to evade existing malware detector too easily using code obfuscation. Accordingly, researchers have focused on the issue of obfuscated malware using code obfuscation techniques to bypass signature based approaches. Since established signature based approaches have obvious limits, work has been undertaken to improve signature based detection. There have also been a few attempts to apply data mining and machine learning techniques, such as Naive Bayes method [25], support vector machine (SVM) [5] and Decision Tree classifiers [15, 29], to detect new malicious executables.

Malware detection falls into two categories, static analysis and dynamic analysis. Static analysis analyzes malware without their execution. Existing malware detectors commonly use static analysis. Binary pattern matching and data flow and code flow analysis represent examples of static analysis. In contrast to static analysis, dynamic techniques execute malware in a simulated environment and analyze its malicious behavior. VM(Virtual machine)s are commonly used for dynamic analysis as a sand box.

Static analysis has some advantages. Static analysis enables fast and safe analysis of malware. In addition, Static analysis can cover the entire malware code and achieves a low level of false positives. Commercial AV products use static analysis for these reasons. However, static analysis is flawed. Static analysis has difficulty analyzing unknown malware. Malware can evade detection easily using obfuscation techniques.

Several dynamic analysis approaches have been proposed to overcome a flaw of static analysis. To detect unknown malware, dynamic analysis executes malware samples in a simulated environment, monitors all system calls, and automatically generates a report to simplify and automate the malware analyst's task. Williams *et al.* presented automated dynamic malware analysis using CWSandbox [30]. TTAnalyze [27] is a tool for dynamically analyzing the behavior of Windows executables. The binary is run in an emulated operating system environment and its actions are monitored. They recoded the Windows native system calls and Windows API functions that the program invokes. However, Dynamic analysis involves system infection. Moreover, malware writers have developed anti-VM (virtual machine) techniques. Malware with anti-VM can recognize that they are in a simulated environment. In addition, there is one observable difference between an emulated and a real system, namely speed of execution. Dynamic analysis needs too much time. Detecting unknown malware before execution, where possible, offers the best solution.

Much effort has been made to overcome the limitations of static analysis. Some research presents several code normalization techniques. Code normalization techniques normalize obfuscated malware to their original form. Christodorescu *et al.* present malware normalization [8], a system that takes an obfuscated executable, undoes the obfuscations, and outputs a normalized executable. Therefore, a malware normalizer can be used to improve the detection

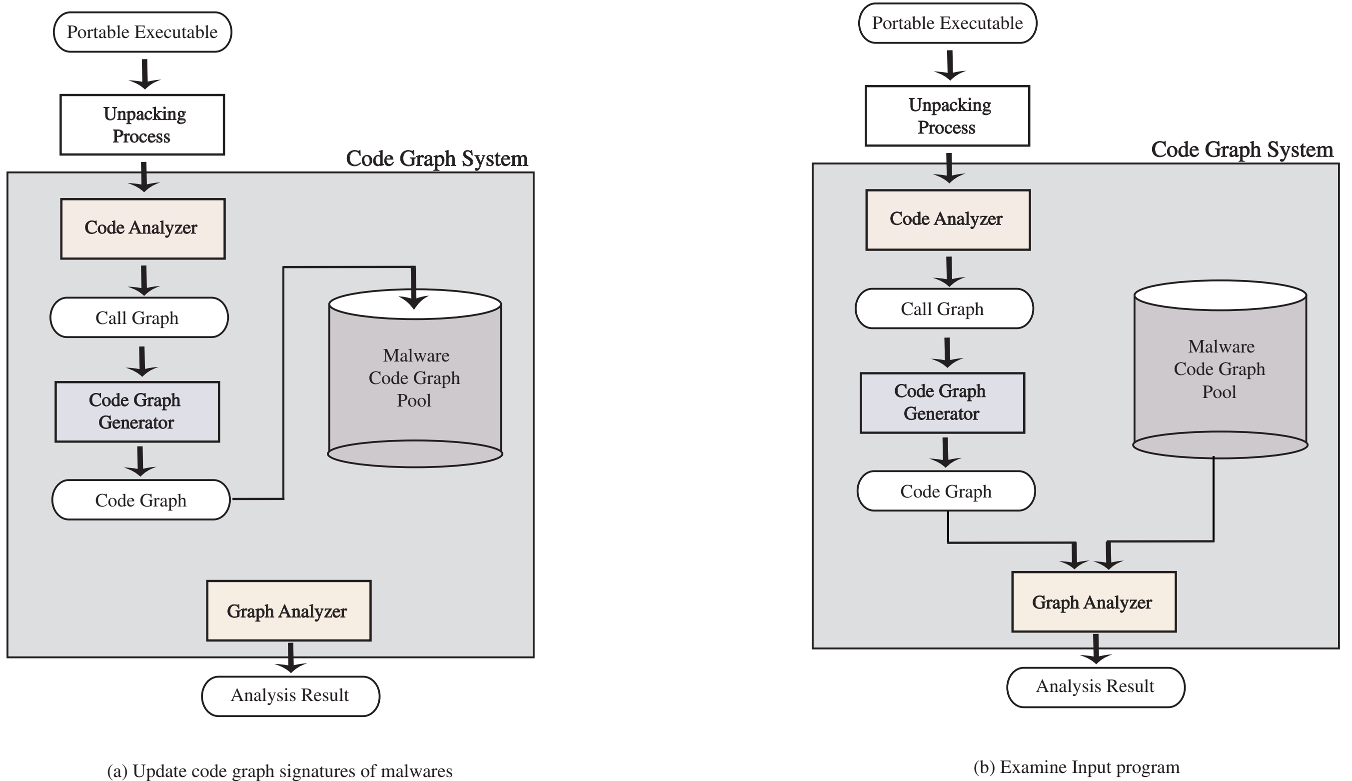


Figure 1: The architecture of the code graph system

rate of an existing malware detector. Walenstein et al. [28] present a method for normalizing multiple variants of metamorphic programs that perform their transformations using finite sets of instruction-sequence substitutions. Normalization is one solution to detect obfuscated malware before execution. However, normalization depends on obfuscating techniques. They cannot cover all forms of code obfuscation.

There has been a lot of work using semantic detection techniques to strengthen static analysis. Christodorescu *et al*. [7] exploited semantic heuristics to detect obfuscated malware. Although, their approach works well for obfuscated malicious programs, the time taken by their approach makes it impractical for use in commercial antivirus scanners. Preda *et al*. [21] use a trace semantics to characterize the behaviors of the malware and the program being analyzed. Sathyanarayan *et al*. [24] focus on critical API calls to generate signatures of malware and detect them. Although they observe the semantics of malware by critical API calls, their approach has a weakness to new type of code obfuscation techniques such as red herring system calls because they use only a frequency of critical API calls.

To summarize, static analysis has difficulty finding unknown malware; while, dynamic analysis can detect unknown malware, but is inefficient. The best method to detect malware efficiently is to detect malware with static analysis, while covering the existing defects of static analysis. To this end, we try to abstract semantic characteristics of malware rather than syntactic features, since the program syntax can be changed easily, but the semantics cannot but help be retained.

We now present a new method to detect obfuscated variants of malware using semantic signatures. The method is introduced and evaluated experimentally.

3. CODE GRAPH SYSTEM

In this section, we describe how to construct the code graph of an executable program, and explain how to determine malware variants using code graphs. Existing malware detection mechanisms commonly detect malware using their syntactic signature. Malware writers have tried to evade malware detectors, and evasion techniques, such as code obfuscation, make it possible for malware to bypass commercial malware detectors. Therefore, our mechanism focuses on semantic characteristics of malware rather than syntactic characteristic. To this end, we describe a program as one graph called code graph. The code graph is a directed graph that represents the characteristics of a portable executable binary. The code graph system is a preview system that enables the program characteristics to be viewed before its execution by generating and analyzing the code graph. Then, the system determines a program whether malicious or not.

Figure 1 shows the architecture of the code graph system. Figure 1(a) is the process of updating code graph signatures of malware. Figure 1(b) is the process of examining the input program. The input program is compared to code graph signatures stored by the update process. Our system consists of three parts, the code analyzer, code graph generator, and graph analyzer. The code analyzer transforms a portable executable binary into a directed graph, the call graph. The call graph is reduced to a simple form, termed code graph by code graph generator. The code graph is saved and used to measure similarity using the graph analyzer. Thereby, the graph analyzer determines if the input program is a variant of a previously stored malware.

3.1 Code Analyzer

The code analyzer transforms a binary into a call graph using

the transformation algorithm. The call graph is a directed graph that represents the characteristics of a portable executable binary. We use the system-call call sequence as the program characteristics. We extract only those instructions related to the system call sequence in the binary executable program and represent the result in the form of a call graph. The call graph is a directed graph $G = (V,E)$, where V is a set of nodes and E is a set of edges. A node is a system call selectively chosen among the system calls in a given program. An edge is determined by the call sequence of the system calls in V, *e.g.* $E = \{(v_i,v_j)|v_i,v_j \in V\}$, where v_i denotes the *caller* system-call, and v_j denotes the *call-in* system-call.

First the code analyzer builds the node set V to transform a binary into a call graph. The code analyzer obtains the system-call set through the IAT (Import Address Table) contained in a binary that denotes the node set V. Next, it builds up a node set by connecting one node to one system-call.

The code analyzer generates the edge (v_i,v_j) of the call graph G using the system-call sequence, where v_i is the caller system-call and v_j is the call-in system-call. Algorithm 1 shows the entire transformation algorithm. Figure 2 shows how to transform each instruction into an edge.

Algorithm 1 Transform algorithm

Input : *portable Executable Binary B*
Output : *Call graph G = (V,E)*

1: $V \leftarrow$ *BuildNode*(); $E \leftarrow \phi$;
2: $v_i \leftarrow 0$;
3: $r \leftarrow$ *EntryPoint*(B);
4: **while** r is not te end of B **do**
5: //The instruction at address r in B
6: $I_c \leftarrow I[r]$;
7: //The parameter of the instruction
8: $P_c \leftarrow P[r]$;
9: ***if*** I_c *is System Call* ***then***
10: ***if*** v_i *is equal to 0* ***then***
11: $v_i \leftarrow P_c$;
12: ***else***
13: $v_i \leftarrow P_c$;
14: $E \leftarrow E \cup$ *Edge*(v_i,v_j);
15: $v_i \leftarrow v_j$;
16: ***end if***
17: ***else if*** I_c *is JMP* ***then***
18: $v_j \leftarrow$ *GetFirstSCALL*(r);
19: $E \leftarrow E \cup$ *Edge*(v_i,v_j);
20: $v_i \leftarrow 0$;
21: ***else if*** I_c *is CJMP* ***then***
22: $v_j \leftarrow$ *GetFirstSCALL*(r);
23: $E \leftarrow E \cup$ *Edge*(v_i,v_j);
24: ***else if*** I_c *is Procedure Call* ***then***
25: $v_j \leftarrow$ *GetFirstSCALL*(r);
26: $E \leftarrow E \cup$ *Edge*(v_i,v_j);
27: $v_j \leftarrow$ *GetEndSCALL*(r);
28: ***end if***
29: $r \leftarrow$ *CurrentProgramCounter*(B);
30: **end while**

3.2 Code Graph Generator

Call graphs are generated by the code analyzer. However, it is difficult to compare call graphs directly. Call graphs generated in the code analyzer have usually hundreds or thousands of nodes and graph isomorphism is a well known NP-complete problem. Thus, we transform the call graph to the code graph for fast simple analysis. The code graph generator creates code graphs. Code graphs are used for signatures of our code graph system.

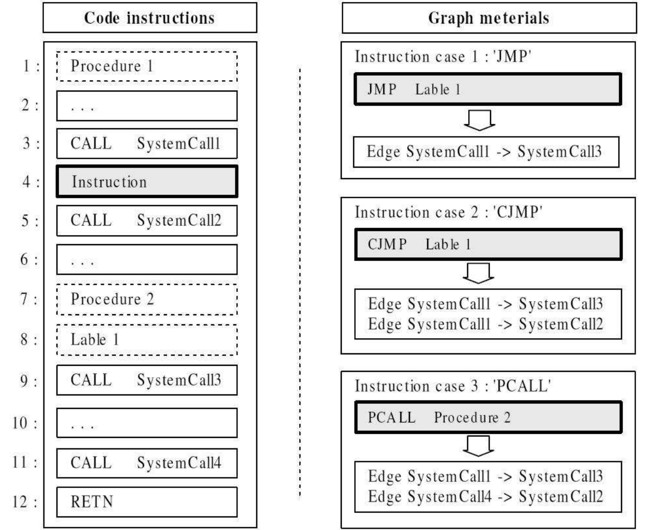


Figure 2: An example of transform

The signature must satisfy the following two conditions to be signature for detection.

- ***Condition 1*** : Signature should express characteristic of program exactly and simply as possible.

- ***Condition 2*** : To distinguish signatures of two programs, signature is distributed uniformly in possible data structure space.

As mentioned in the previous section, the call graph is a directed graph that has nodes and directed edges. API calls can be nodes, and directed edges represent the program calling relationship. We have classified API calls to 128 groups(32 objects $\times$ 4 behaviors), to reduce this call graph. To obey the previous conditions, nodes of the call graph are grouped by objects of a system call. The objects are process, memory, socket, and so forth. There are 32 objects, we reference to MSDN. At the first step, the system call is classified by a related object. At the next step, each system call is classified by the behavior of the related object. We define four basic behaviors of system calls which are open, read, write, and close. In this way, API calls are classified to one group of 128 groups. For example, CloseSocket() becomes a member of the socket-close group and RegSaveKey() becomes a member of the registry-write group.

After grouping, the nodes within the same group in the call graph are unified. During node unification, edges represented call relationships are maintained.

When the nodes are unified, we have a problem. We cannot cover all API calls. Therefore, some nodes do not belong in our grouping. In this case, we ignore the node, and keep the link information of the node.

This method transforms the call graph to a code graph. We extract code graphs from known malware and store it to detect unknown malware. The code graphs from known malware are used to compare code graphs of suspicious programs. We use an adjacent matrix to store code graphs. An adjacent matrix is most proper data

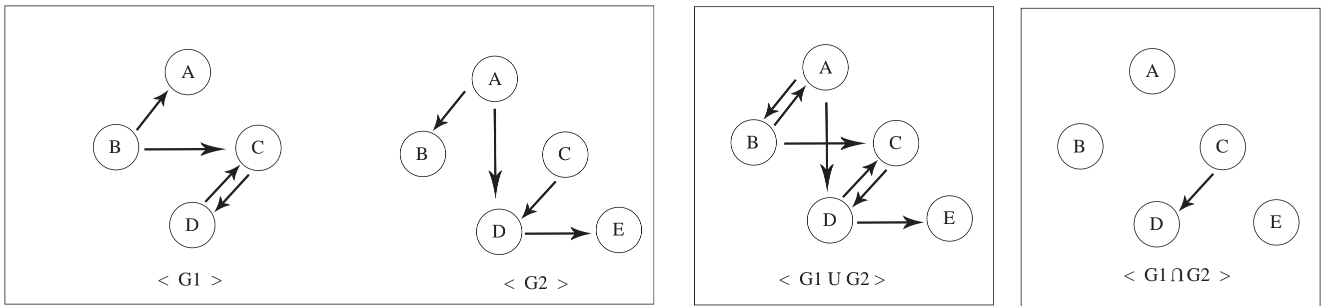


Figure 3: An Example of Similarity Measurement., which is used to determine metamorphic malware. First, we compute the intersection and the union graph of two measured graphs. Next, we divide the number of the union graph's edges by the number of the intersection graph's edges. In this example, an union graph has seven edges and an intersection graph has one edge. Thus, the similarity of G_1 and G_2 is 0.1429.

structure to store code graph, because the code graph is a directed graph whose number of edges is fixed to 128. Thus, a code graphs is represented by a 128×128 adjacent matrix and saved. A malware code graph occupies 16KB.

3.3 Graph Analyzer

Graph analyzer measures the similarity of two code graphs and determines if the input program is malicious. In previous, we stored code graphs of known malware formed in a 128 by 128 adjacent matrix. When the input program is examined, the code graph is extracted by the code analyzer and the code graph generator. The code graph is compared to code graphs of known malwares. The similarity can be measured in a simple way, which can be described as follows.

Graph G_1 and G_2 are to be compared. We compute an intersection graph and an union graph of G_1 and G_2. We can find the intersection graph of G_1 and G_2 easily as AND operation of the adjacent matrix of G_1 and adjacent matrix of G_2. In the same way, we can find the union graph of G_1 and G_2 easily as the OR operation of the adjacent matrix of G_1 and adjacent matrix of G_2. Next, we evaluate the similarity of two graphs to divide edges of the union graph by edges of the intersection graph. Then, we define the similarity ϕ of graph G_1 and G_2 as follows.

$$\phi(G_1, G_2) = \frac{|E(G_1 \cap G_2)|}{|E(G_1 \cup G_2)|}$$

Figure 3 shows a simple example of our graph similarity measurement. G_1 has 4 vertices and 4 edges. G_2 has 5 vertices and 4 edges. We compute the intersection graph and union graph of two graphs to measure the similarity of G_1 and G_2. The union graph of G_1 and G_2 has 7 edges and the intersection of two graphs has only an edge from node C to node D(Figure3). Thus, the similarity of two graphs is 1/7, 0.1429.

There are many methods to measure the similarity of labeled graphs. Rascal [23] used maximum common edge sub-graphs to measure the similarity of graphs. Zelinka-distance [31] is based on the principle that two graphs are more similar, the bigger the common induced sub-graph. The similarity measurement using cosine distance of vector is another possible method. However, in malware detection, a fast operation is important. In addition, since the number of labeled nodes is finite in our graph, we define a simple operation to measure graph similarity. The time complexity is O(c) in our similarity measurement. That is, whether the program size in the comparison is huge or tiny, we conduct constant operations. We compute only AND & OR operations of the 128×128 matrix and a division operation. We use the number of edges of the intersection and union graphs instead of vertices. Since edges are able to represent characteristics of a graph than vertices, in the example depicted in figure 3, when we use a number of vertices, we get a relatively high similarity, 0.8 although the two graphs look different. There are many cases where the vertex sets are the same but the edge sets are different in the two graphs, but there is no inverse case. Therefore, we focus ob graph edges to measure similarity.

Since the intersection graph is always a sub-graph of the union graph, the number of edges is always less than that of union graph and the similarity must range from 0 to 1. When code graphs are identical, the similarity will be 1. If there is no common edge of the two graphs, the similarity is 0. As a code graph of input program P is compared to the code graph G_i, $\phi(P,G_i)$ is 1, we determine P is a variant of malware i. When $\phi(P,G_i)$ is greater than 0.9, we decide P has a suspiciously high probability of being a variant of G_i. The value, 0.9 is obtained from the experimental result.

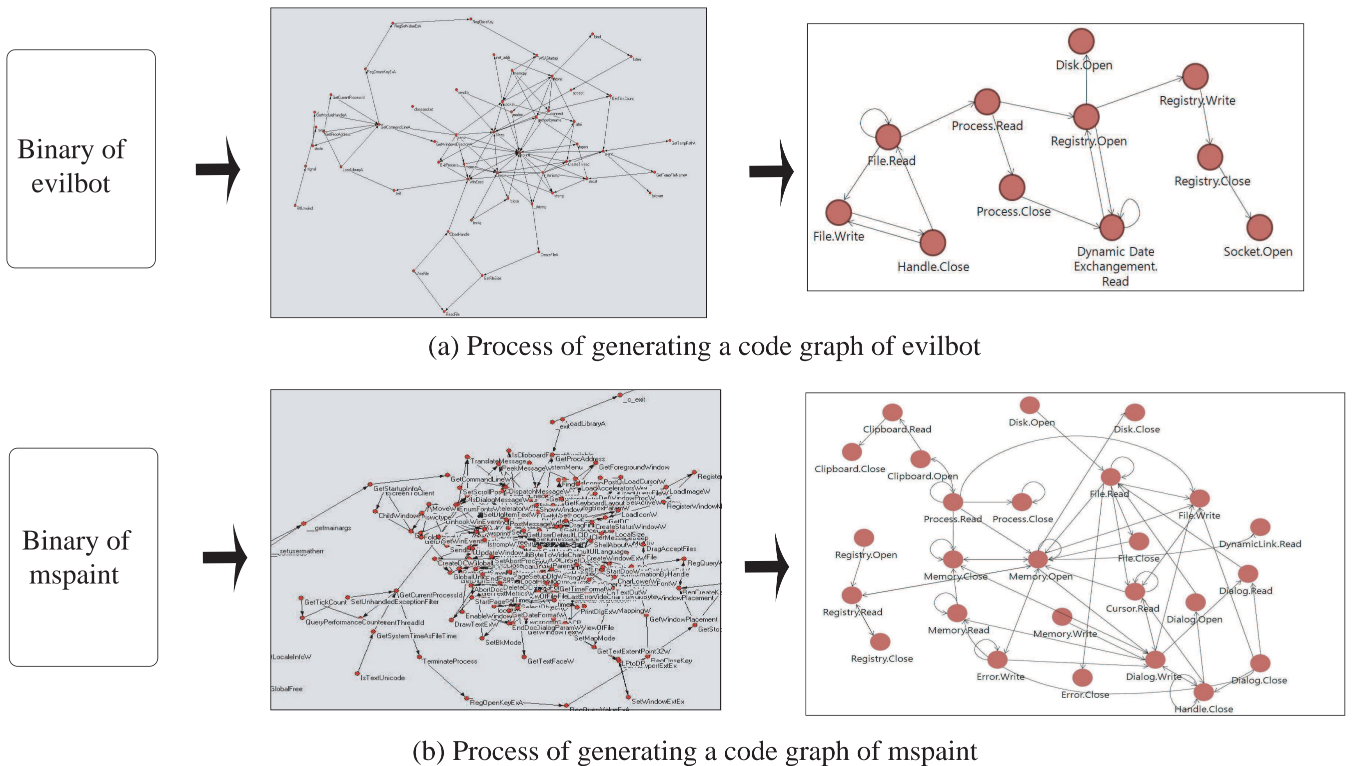
4. EVALUATION

We performed a series of experiments with the code graph system to evaluate its efficiency. In the first experiment, we show how uniquely our semantic signatures represent the program characteristics. The second experiment shows the similarity of real-world malware variants collected from wild environment. In the last experiment, we create 300 metamorphic malwares using three code obfuscation techniques. We evaluate the efficiency of our mechanism compared to existing AV scanners. All experiments were performed on a machine running Windows XP, with a Pentium 4 CPU of 3GHz, and 2GB of RAM.

4.1 Construction of Semantic Signatures

This section provides the process to form semantic signatures. As mentioned in a previous section, we perform the following process to create a signature. First, we extract the sequence of API calls from the executable binary. With the sequence of API calls, we construct a call graph by relations of function calls. The call graph is reduced to a code graph based on the object and the behavior of the API call. The code graph generated by this process is stored by form of 128×128 adjacent matrix of the graph.

Figure 4 shows the process to construct code graphs from the windows PE file. Figure 4(a) is an example of a malicious program, evilbot. Figure 4(b) is an example of a benign program,



(a) Process of generating a code graph of evilbot

(b) Process of generating a code graph of mspaint

Figure 4: Construction of code graphs

mspaint. The evilbot is a kind of malware, its size is 15KB. We extract a sequence of API calls with a debugger from evilbot code. The sequence is represented as a call graph. Figure 4 shows the call graph is too complex to be compared. We reduce the call graph to a code graph for comparison. The call graph takes 1.273 seconds to make and 62 milliseconds to construct the code graph. Mspaint is a benign program provided by microsoft and its size is 66 KB. As can be seen in figure 4, the call graph of mspaint is much more comlicated than that of evilbot. The number of nodes of the evilbot and mspaint call graphs are 54 and 188 respectively. Generally, the greater the size, the more complex the code graph. It takes 2.547 seconds to make the call graph and 172 milliseconds to generate the code graph for mspaint. In evilbot, the code graph has 11 nodes and 15 edges. The code graph of mspaint has 29 nodes and 75 edges. Two graphs have only one common edge. Therefore, in this example, the similarity of two graphs is the 0.01124.

4.2 Uniqueness of Semantic Signatures

In first experiment, we measure how our semantic signatures are distributed uniquely in the possible data structure. We obtained a low similarity of the benign program and the malicious program in the previous section. However, we could not tell if our semantic signatures are distributed indiscriminately in the possible space. To this end, we collect code graphs from various programs and measure their similarities. This experiment is highly related to false-positives of our system. The 128 by 128 binary matrix can represent $2^{2^{14}}$ graphs. However, if the graphs are not distributed uniformly in this data structure space, different programs may have the same code graph. In this case, false-positives is increased.

300 benign programs and 100 malicious programs were analyzed. First, we extracted code graphs from the 300 benign programs. Next, we measured the similarities of all cases where we picked two graphs. There are ${}_{300}C_2$, 44850, combinations. The figure 5 (a) shows the results in rank order of 44850 similarities. Then, with the 100 malicious programs, similarities were evaluated using the same method. Last, we measured similarity of 300 benign programs and 100 malicious programs. If two graphs have a similarity of 1, this is false-positive from our mechanism. Only 0.003% percentage had a similarity greater than 0.9, representing a Benign-Malicious case. Only one case in the 30000 cases had a similarity greater than 0.9 between malware and a benign program.

	Size	Time to call graph	Time to code graph	Similarity
agobot	45KB	1.571s	0.110s	1.0
G-spot	439KB	16.784s	0.235s	1.0
evilbot	15KB	1.273s	0.062s	1.0
backdoor.Bot	87KB	2.393s	0.094s	0.8182
p2p-worm	48KB	1.714s	0.081s	1.0
Trojan.mybot	69KB	2.193s	0.068s	1.0

Table 1: Detection of malware variants

4.3 Detection of Real-world Malware Variants

Malware writers have created many variants of common exploits, in attempt to evade AV scanners. Such examples can be found at popular hackniig web sites. Such examples make use of a wide range of obfuscation techniques. We can get many malware variants at such sites.

100 malware pairs are used for experiment. We collected malware variants from VX Heavens [2], VX Chaos [1] and offensive-Computing.net. The malware programs tested had sizes ranging

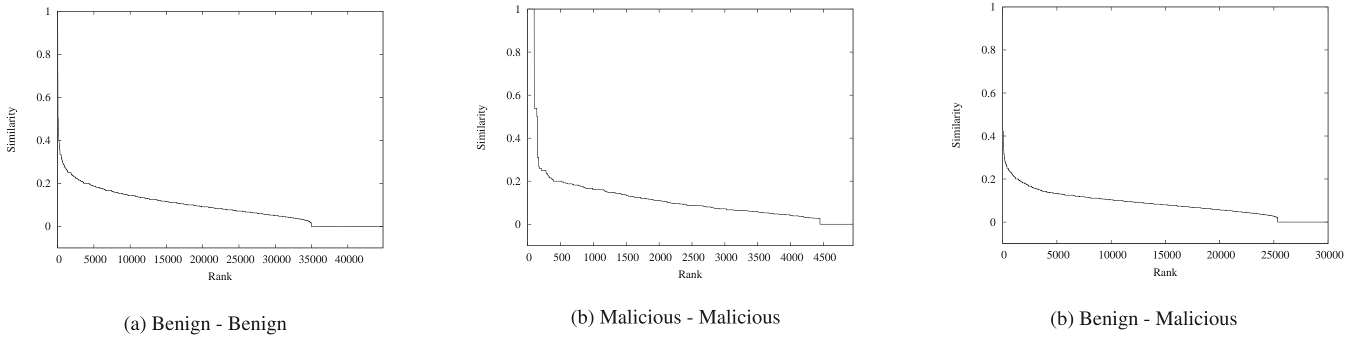


Figure 5: Distribution of Semantic Signature

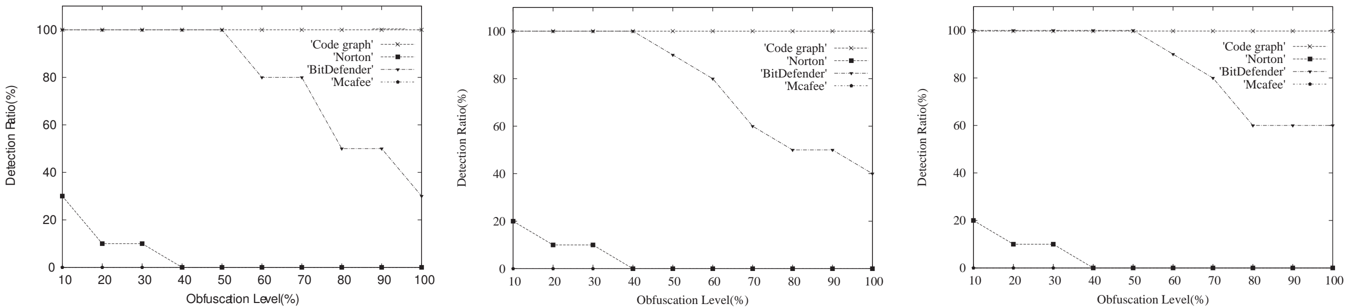


Figure 6: Detection of obfuscated malwares

from 15KB to 1MB bytes. Table 1 shows the result for some cases. Of the 100 cases, 91% of malware varinats had a similarity of 1. 3% of variants had a similarity of greater than 0.9 but not 1 and our system determines these cases to be suspicious. Last, it shows 6% false-negative ratio. One example of false-negative was backdoor.bot($\phi = 0.8182$) in Table 1. The 0.8182 is the lowest similarity of 100 pairs.

Average time of 200 malware variants(100×2) from binary to code graph is 2.183s. In case of most malware variants, it takes usually from one second to three seconds to construct the code graph. The bigger the size of malware is, the longer exponetially it takes. We construct a code graph only one time per a malware because we stored signatures of malware in malware signature pool. Therefore, few seconds is negligible levels. Similarity measurement calculates constant operations, AND and OR operation of two 128 by 128 matrix and one division. The consumed time is also very low cost.

4.4 Detection of Obfuscated Malwares

We sample 10 malicious programs in this experiment. The malicious programs are composed of two backdoors, two worm, two trojans, two virus and two bot codes. We applied three code obfuscations for each malware, including code insertion, code reordering, and code replacement from 10% to 100%. Code insertion adds useless code into the program. Code reordering modifies the execution flow of a program using unconditional branch instructions. Code replacement replace a given instruction block with another instruction block while keeping the same code semantics. Malware is divided into basic blocks. The basic block describes a sequence of instructions without any jumps or jump targets in the middle. Then, we apply each code obfuscation techniques according to obfuscation level.

Figure 6 shows the results of our investigation of obfuscated malwares, using three commercial scanners and our mechanism. Most AV scanner using syntactic signatures cannot detect metamorphic malware made by code obfuscation. It is shown that Self-defense is a serious menace to malware detector. We achieved 100% detection of the metamorphic malware instances using semantic characteristics of malware. The code graph system has another advantage that the number of malware signatures is reduced significantly. In syntactic signature based system, a linear relationship is existed between malware and a signature. That is, whenever malware is generated, one signature is needed for malware. Since many variants of malware are generated from malware, signatures are made as many as the number of variants. By contrast, the code graph system needs only one signature. The code graph system can detect all of malware variants with a signature of original malware. It is pertinent to note that 50% new malware are obfuscated versions of existing known malware [26]. This situation will be accelerated. Therefore, keeping down malware signature is an important issue. In addition, small number of signatures provides malware detectors to save signature matching time.

There are many other code obfuscation techniques in addition to three techniques. The register reassignment transformation replaces usage of one register with another in a specific live range. Code integration used to merge two separate code sections without the need to obscure or recompile the existing code sections. Entry point obscuring tries to hide its entry point in order to avoid detection. Our mechanism can defend these type of code obfuscation techniques because we use API call sequence. However, our approach may suffer the code obfuscation techniques such as insertion of meaningless system calls. This is related to false-negative of proposed mechanism. In future work, we expand to defend these types of obfuscation.

5. CONCLUSION

In this paper, we provided a new approach to generate semantic signatures from programs to detect metamorphic malware. The key idea is that the sequence of API call is preserved during the obfuscation process. Obfuscation techniques change the malware syntax, but cannot affect its behavior. We describe API call sequence of malware as directed graphs called code graphs. The code graphs are used for the semantic signature. The semantic signature makes us possible to detect malwares using obfuscation that would otherwise easily evade commercial AV products using obfuscation. Evaluations show that our semantic signature represents characteristic of program uniquely and correctly. In addition, we show that our code graph system can detect real-world malware variants and metamorphic malware that can evade AV scanners.

In the future work, we plan to extract semantic signatures from various kinds of malware. We will consider more accurate and more sophisticated analysis in static analysis. We will try to deduce common features each type of malware using improved analysis. Reducing false-positive and false-negative is also one part of our future work. White list for benign program is one possible solution to reduce false-positive. New type of code obfuscation techniques can lead to false-negative. We will find the method to deal with the code obfuscation techniques. In addition, since most malware is packed, a generic unpacking technique will optimize our code graph system. Eventually, we will expand our work to vulnerability and abnormality analysis.

6. ACKNOWLEDGMENTS

This research was supported by the MKE(Ministry of Knowledge Economy), Korea, under the ITRC support program supervised by the NIPA(National IT Industry Promotion Agency) (NIOA-2009-(C1090-0902-0016)). Additionally, this research was sponsored in part by the MSRA (Microsoft Research Asia), Korea SW Industry Promotion Agency (KIPA) under the program of Software Engineering Technologies Development and Experts Education, and the IT RD Program of MKE/KEIT. [2009-KI002090, Development of Technology Base for Trustworthy Computing]

7. REFERENCES

[1] Vx chaos file server. http://vxchaos.official.ws.

[2] Vx heavens. http://vx.netlux.org.

[3] AVV. Antiheuristics. *29A Magazine*, 1(1), 1999.

[4] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *IEEE Security & Privacy*, 5(2):46–54, 2007.

[5] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, 2(2):121–167, 1998.

[6] M. Christodorescu and S. Jha. Testing malware detectors. In *ISSTA*, pages 34–44, 2004.

[7] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, pages 32–46, 2005.

[8] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. Malware normalization. Technical report, University of Wisconsin, November 2005.

[9] F. Cohen. Computer viruses: Theory and experiments. In *7th DOD/NBS Computers and Security Conference*, volume 6, pages 22–35, September 1987.

[10] D.Chess and S. White. An undetectable computer virus. In *Virus Bulletin Conference*, September 2000.

[11] M. Driller. Metamorphism in practice. *29A Magazine*, 1(6), 2002.

[12] G. Jacob, H. Debar, and E. Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4(3):251–266, 2008.

[13] K. Jeong and H. Lee. Code graph for malware detection. In *Information Networking. ICOIN. International Conference on*, Jan 2008.

[14] L. Julus. Metamorphism. *29A Magazine*, 1(5), 2000.

[15] J. Z. Kolter and M. A. Maloof. Learning to detect malicious executables in the wild. In *KDD*, pages 470–478, 2004.

[16] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45, 2007.

[17] D. Mohanty. Anti-virus evasion techniques and countermeasures, August 2005. http://www.hackingspirits.com/eth-hac/papers/whitrepapers.asp.

[18] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *ACSAC*, pages 421–430, 2007.

[19] A. Moser, C. Krügel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[20] C. Nachenberg. Computer virus-antivirus coevolution. *Commun. ACM*, 40(1):46–51, 1997.

[21] M. D. Preda, M. Christodorescu, S. Jha, and S. K. Debray. A semantics-based approach to malware detection. *ACM Trans. Program. Lang. Syst.*, 30(5), 2008.

[22] Rajaat. Polimorphism. *29A Magazine*, 1(3), 1999.

[23] J. W. Raymond, E. J. Gardiner, and P. W. 0002. Rascal: Calculation of graph similarity using maximum common edge subgraphs. *Comput. J.*, 45(6):631–644, 2002.

[24] V. S. Sathyanarayan, P. Kohli, and B. Bruhadeshwar. Signature generation and detection of malware families. In *ACISP*, pages 336–349, 2008.

[25] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *IEEE Symposium on Security and Privacy*, pages 38–49, 2001.

[26] G. Taha. Counterattacking the packers. McAfee Avert Labs, Aylesbury, UK.

[27] C. K. Ulrich Bayer and E. Kirda. Ttanalyze: A tool for analyzing malware. In *Proc. 15th Ann. Conf. European Inst. for Computer Antivirus Research (EICAR)*, pages 180–192, 2006.

[28] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. Normalizing metamorphic malware using term rewriting. In *SCAM*, pages 75–84, 2006.

[29] J.-H. Wang, P. Deng, Y.-S. Fan, L.-J. Jaw, and Y.-C. Liu. Virus detection using data mining techinques. In *Proceedings. IEEE 37th Annual 2003 International Carnahan Conference on*, pages 71– 76, 2003.

[30] C. Willems, T. Holz, and F. C. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2):32–39, 2007.

[31] B. Zelinka. On a certain distance between isomorphism classes of graph. In *Casopis pro pestovani Matematiky*, volume 100, pages 371–373, 1975.