Code Graph for Malware Detection

Kyoochang Jeong, Heejo Lee

Division of Computer and Communication Engineering
Korea University
Seoul 136-713, South KOREA
{kyoochang,heejo}@korea.ac.kr

***Abstract*— When an application program is executed for the first time, the results of its execution are not always predictable. Since the host will be damaged by a malware as soon as it is executed, detecting and blocking the malware before its execution is the most effective means of protection. In contrast to current research into the detection of malwares based on their behavior while being executed, we propose a new mechanism which can preview the effect of a program on a system. The mechanism we developed is to represent the distinctions between portable executable binaries. The proposed mechanism analyzes the instructions related to the system-call call sequence in a binary executable and demonstrates the result in the form of a topological graph. This topological graph is called the code graph and the preview system is called the code graph system. We have tested various real application programs with the code graph system and identified their distinctive characteristics which can be used for distinguishing normal softwares from malwares such as worm codes and botnet programs. Our system detected all known malwares used in the experiment, and distinguished 67% of unknown malwares from normal programs. In this paper, we show how to analyze the effects of executable binaries before their execution and normal softwares can be effectively distinguished from malwares by applying the code graph.**

I. Introduction

Internet attack programs, so called malwares, infect hosts connected to the network, and they not only exert a mischievous influence on the system but also degrade the networking performance of the system. According to a study in which network operators were surveyed about Internet attacks, Internet worms and DDoS attacks are two most serious attacks, amounting to as much as 70∼80% of the attacks [1].

An Internet worm infects hosts and uses them to infect more hosts on the networks. The worm propagation model shows the progression of the damage resulting from a worm attack over time [2]. Once a host is infected by a worm, the subsequent damage increases rapidly. Therefore, the best method of preventing a worm from infecting a host is to notify the user of its presence before its execution.

DDoS attacks are mostly launched by a large pool of compromised hosts, which are called software robots or simply "bots." Such a network of bots is commonly referred to as a botnet, which is also used for sending spam mails, stealing personal information, as well as launching DDoS attacks. According to the results of a recent study, the number of botnet hosts amounts to 100 million [3]. When a PC is infected by a bot code, the user is not typically aware of it because of its stealthy behavior. Therefore, it is better to know what a program is before it is installed.

The traditional method of detecting malwares before their execution is pattern matching using malware signatures. As well, most malware cleaning tools heavily rely on the predefined signatures. However, many problems with these tools have been pointed out and the most serious problem is the rapid growth of the signature database and its timely update whenever new malwares are detected [7]. One way to decrease the number of signatures is the use of normalization techniques that have been actively studied recently [4], [5]. However, this approach does not provide a complete solution for detecting unknown malwares.

Most computer users do not know what kind of functions are included in a program before it is executed for the first time. Moreover, they sometimes cannot know the internal functions in the program even if they execute it. It means that a user may install or execute a program which they do not intend to use. This motivates us to study a preview mechanism which can decrease the danger associated with the execution of such programs.

We propose a mechanism to determine what kinds of functions are included in a program before its execution. The proposed mechanism analyzes the instructions related to the system-call call sequence in a binary executable and demonstrates the result in the form of a topological graph. This topological graph is called the code graph and the preview system is called the code graph system.

The code graph system is able to determine what kind of functions are contained in the program before it is executed. Moreover, the system analyzes the portable executable binary for the purpose of determining what kinds of functions are programmed on the operating system before executing the program. We employ the code graph system to analyze malwares for the purpose of finding the difference between them and normal programs. From the experiments with real application programs and malwares, it is shown that the code graph can detect even unknown malwares, as well as typical malwares, using their distinctive characteristics of malwares and normal programs.

II. Background

A. CPU instructions in binaries

To extract the characteristics of a program before its execution, we need to analyze its binary. We interpreted the instructions in portable executable binaries using the Intel x86 instruction set reference provided by Intel [8]. There are

thousands of instructions in Intel x86 architectures. However, we need only a few instructions related to the system call calling sequence. Therefore, we divided the required instructions into 4 groups and extracted the instructions included in these groups from the binaries. These grouped instructions are illustrated in Table. I.

Group	Instructions
1. SCALL: System-call call	CALL(system-call)
2. PCALL: Procedure call	CALL(procedure)
3. JMP: Jump	JMP
4. CJMP: Conditional jump	JA, JAE, JB, JBE, JC, JCXZ, JECXZ, JRCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, JA, JAE, JB, JBE, JC, JE, JZ, JG, JGE, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ

TABLE I
THE INSTRUCTIONS INCLUDED IN 4 GROUPS

B. Code obfuscation of malwares

The code obfuscation is a code protection method by complications such as encryption and compression. There are three common code obfuscation methods: code reordering, junk-insertion, packing. The code reordering obfuscation changes the syntactic order of instructions in a program while maintaining the execution order through the insertion of jump instructions. The junk-insertion obfuscation randomly adds dummy codes that do not change the program behavior. The packing obfuscation replaces a code sequence with a data block containing the code sequence in encrypted or compressed form and an unpacking routine that, at runtime, recovers the original code from the data block [5],[6].

Most malwares use code obfuscation methods in order to make it difficult to analyze them or to reduce their size. The code obfuscation method itself is not within the scope of our study. Nonetheless, we will show that the code graph is effective even if code obfuscation methods are used, as explained in the evaluation section.

III. CODE GRAPH

In this section, we describe how to construct the code graph of an executable program, and explain how to detect suspicious programs before their execution.

A. Motivation

The motivation of the code graph is based on a simple proposition. A program consists of a certain number of functions arranged in a certain sequence designed by a programmer for the purpose of achieving its goal. However, this does not mean that if we know the function call sequence of a program, we can know what the goal of the program is. This relationship is explained in Fig. 1.

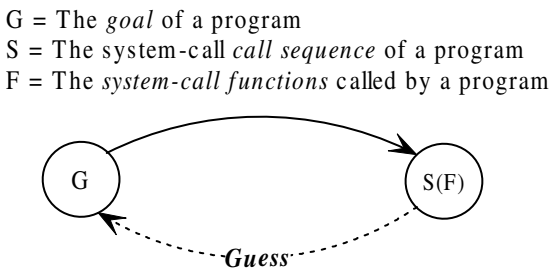

Fig. 1. The motivation of the code graph

In this paper, we identify the call sequence of system call functions in a program. And, even if we do not know the exact goal of each program, we can predict the malevolence of the program by analyzing the call sequence comparing with predefined behavioral patterns.

B. System architecture

The code graph is a directed graph which represents the characteristics of a portable executable binary. The code graph system is a preview system which enables the characteristics of a program to be viewed before its execution by generating and analyzing the code graph. Our system analyzes the portable executable binary, which is the design drawing of a program. the characteristics of a program, we use the system-call call sequence. We extract only those instructions related to the system call call sequence in the binary executable program and represent the result in the form of a code graph. Here we define the code graph as follows.

A *code 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 $call\text{-}in$ system-call.

Fig. 2 shows the architecture of the code graph system. Our system consists of two parts, the code analyzer and the graph analyzer. The code analyzer transforms a portable executable binary into a directed graph, which is called the code graph. And the graph analyzer analyzes and measures the code graph to find out whether the program is malicious and to determine the characteristics of the program.

C. Code analyzer

The code analyzer transforms a binary into the code graph using the transformation algorithm. To transform a binary into a code graph, first the code analyzer builds the node set V. The code analyzer obtains the system-call set through the IAT (Import Address Table) contained in a binary which means the node set V. And it builds up a node set by connecting one node to one system-call.

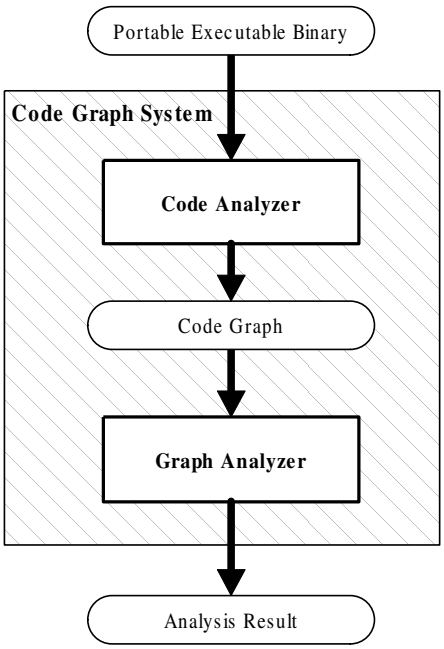

Fig. 2. The architecture of the code graph system

The code analyzer generates the edge (v_i, v_j) of the code graph G using the system-call sequence, where v_i is the caller system-call and v_j is the call-in system-call. Fig. 3 shows the whole transformation algorithm. And Fig. 4 shows how to transform each instruction into an edge.

D. System-call grouping

We have classified system-calls into five categories, in order to satisfy the following two conditions.

- **Condition 1**: The system-call group should express simple information to determine the characteristics of malwares easily.
- **Condition 2**: The system-call group should include detailed information to represent the characteristics of malwares exactly.

We drew the code graph on layer 2 and 3 according to the results of the following equations. If there are n code graphs, the edge size of each graph can be denoted by e_i. Then, the subgraph G_s is the intersection graph of the n graphs. Thus, we can define the rate of a subgraph G_s as follows.

$$R_s = \frac{1}{n}\sum_{i=1}^{n}\frac{e_s}{e_i} \tag{1}$$

When a code graph G_x is given on layer x, we let e_x denote the number of edges in G_x. Then, the rate of information accuracy can be defined as follows.

$$R_a = \frac{e_0}{e_x} \tag{2}$$

Input: Portable Executable Binary B
Output: Code Graph $G = (V, E)$

Transformation Algorithm

$V \leftarrow BuildNode(); E \leftarrow \phi$;
$v_i \leftarrow 0$;
$r \leftarrow EntryPoint(B)$;
WHILE r is not the end of B
    // Instruction at address r in B
    $I_c \leftarrow I[r]$;
    // Parameter of the instruction at address r
    $P_c \leftarrow P[r]$;
    IF I_c is in $SCALL$
        IF v_i is equal to 0
            $v_i \leftarrow P_c$;
        ELSE
            $v_j \leftarrow P_c$;
            $E \leftarrow E \cup Edge(v_i, v_j)$;
            $v_i \leftarrow v_j$;
        END IF
    ELSE IF I_c is in JMP
        $v_j \leftarrow GetFirstSCALL(r)$;
        $E \leftarrow E \cup Edge(v_i, v_j)$;
        $v_i \leftarrow 0$;
    ELSE IF I_c is in $CJMP$
        $v_j \leftarrow GetFirstSCALL(r)$;
        $E \leftarrow E \cup Edge(v_i, v_j)$;
    ELSE IF I_c is in $PCALL$
        $v_j \leftarrow GetFirstSCALL(r)$;
        $E \leftarrow E \cup Edge(v_i, v_j)$;
        $v_i \leftarrow GetEndSCALL(r)$;
    END IF
    $r \leftarrow CurrentProgramCounter(B)$;
END WHILE
END of Algorithm

Fig. 3. Description of Transformation Algorithm

E. Graph analyzer

Let us explain the graph analyzer which is one of the two main components in the code graph system. The graph analyzer should extract the characteristics of the program and determine its goal using the code graph drawn by the code analyzer. However, it is difficult to identify the meaning of the graph because it is not easy for a human being to understand it. Therefore, we need another graph form. This alternative representation of the code graph should satisfy the following conditions.

- It should represent all information included in the code graph.
- It should be easy to understand by a human being.
- It should be easy to identify the similarity of multiple code graphs.
- It should be easy to find the difference between the code graphs.

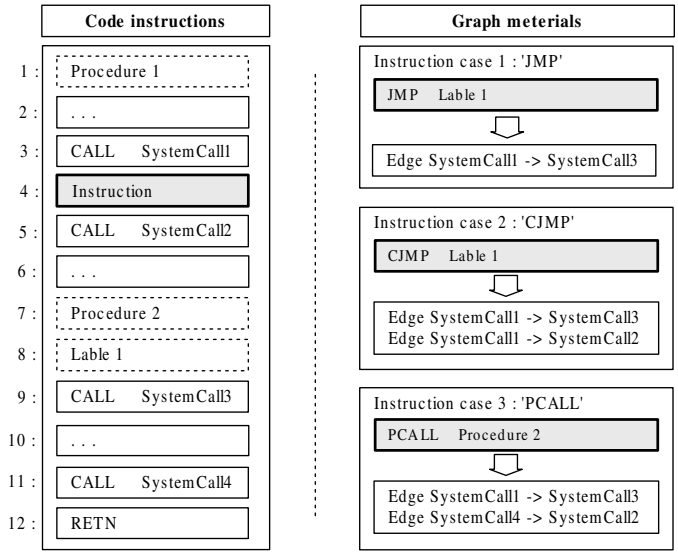

Fig. 4. An example of the transformation

We define the code graph matrix satisfying the above four conditions. The code graph matrix M is an $n \times n$ size matrix for which each entry has the following structure.

Entry structure of the code graph matrix {
boolean $valid$
boolean $fill\text{-}in$
boolean $caller[4]$
boolean $call\text{-}in[4]$
}

Each entry M_{ij} is the code graph edge $E = (v_i, v_j)$ on layer 2. Row i of the code graph matrix means v_i of the code graph edge and column j means v_j. A valid bit of an entry structure means the edge's validation. If there exists an edge then the $fill\text{-}in$ bit is true. The caller bits are v_i's attribute on layer 3 and $call\text{-}in$ bits are v_j's attribute on layer 3.

To analyze the code graph matrix we define the matrix examination in two steps and the matrix operation of intersection $\cap$. The intersection is an operation of the code graph matrix and it generates a new matrix having common entries in the two matrices. The two steps in the matrix examination are as follows.

- Step 1. Generation of a filter matrix :
  $M_a \cap M_b = M_f$
- Step 2. Examination of an input matrix :
  $M_i \cap M_f = M_r$

The first step is to generate a filter matrix. In this step, the filter matrix M_f which is generated has common entries between the two program's matrices M_a and M_b. In this paper, our objective is to detect malwares. So, we express the characteristics of malwares as the filter matrix M_f in advance, in order to detect a similar malware matrix.

The next step is the examination input matrix step using the filter matrix, M_f. In this step, we apply the intersection operation to the input matrix, M_i, and the filter matrix, M_f, and generates the result matrix, M_r. This result matrix is used by the matrix measurement operation.

We define the matrix operation intersection used by the two matrix examination steps as described below.

> If A and B are code graph matrices of the same size, then the intersection $A \cap B$ is the same size code graph matrix whose entries are determined as follows.
> 1. If each valid bit of corresponding entries is $true$ and the $fill\text{-}in$ bits of the corresponding entries are equal then the valid bit of the result entry is $true$. Otherwise it is $false$.
> 2. The $fill\text{-}in$ bit is true if each $fill\text{-}in$ bit of the corresponding entries is $true$. Otherwise is $false$.
> 3. The $caller$ bit is $true$ if each $caller$ bit of the corresponding entries is $true$. Otherwise is $false$.
> 4. The $call\text{-}in$ bit is $true$ if each $call\text{-}in$ bit of the corresponding entries is $true$. Otherwise is $false$.

IV. Evaluation

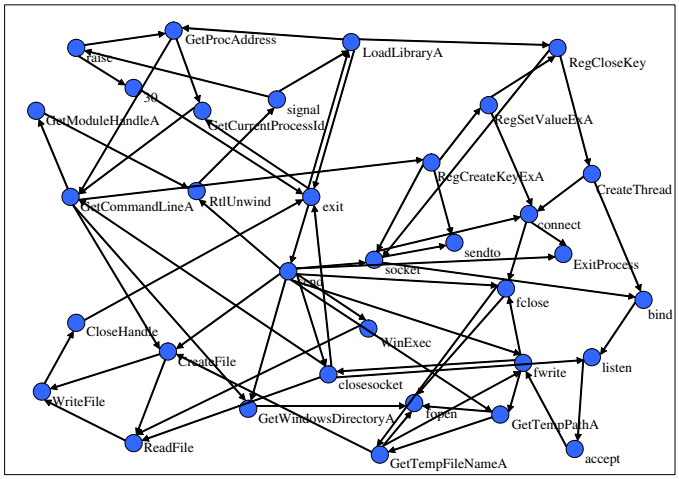

Fig. 5. The code graph of the part of the evilbot

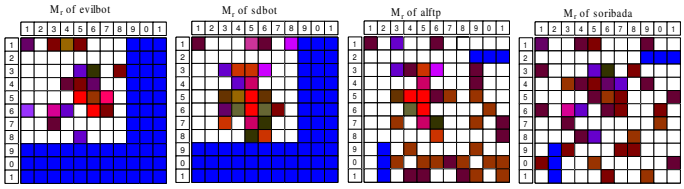

Fig. 6. The M_r of malwares and normal programs

Fig. 5 shows the code graph of the part of the evilbot, and fig. 6 shows the M_r of malwares and normal programs. We can see the difference between the programs by the matrix

	sdbot	evilbot	agobot	scvhost	alftp	soribada	msn	starcraft
S_m	1.00	1.00	0.92	0.97	0.58	0.84	0.62	0.55
D_m	0.00	0.00	0.12	0.06	0.36	0.33	0.46	0.36
S_m - D_m	1.00	1.00	0.80	0.91	0.22	0.51	0.16	0.19

TABLE II
THE RESULT OF THE M_r MEASUREMENT

	N	F	V	Code Graph
Packing	0	0	0	0
Code reordering	3	1	2	4
Junk insertion	2	2	3	4
total	5	3	5	8

TABLE III
THE EXAMINATION RESULT OF NEW GENERATED MALWARES BY CODE OBFUSCATION METHODS

color, and we can see that malwares do not have system-calls related to user interface.

We examined the code graph system with four malwares and four normal programs. And we measured each the similarity and the difference of the result matrix in each case. Also, we tested the code graph system with three common code obfuscation methods and compared the results with those obtained using three popular malware detection tools.

To measure the result matrix, M_r, we count the number of entries with the entry structure value. These counter values are shown in Table. IV.

	valid	fill-in
N_1	true	true
N_2	false	true
N_3	true	false
N_4	false	false

TABLE IV
THE COUNTER VALUES OF M_r

We compute the similarity of matrix, S_m, and the difference of matrix, D_m, as follows.

$$S_m = N_1/(N_1 + N_2) \quad (3)$$

$$D_m = N_4/(N_3 + N_4) \quad (4)$$

To determine whether the examined program is a malware or a normal program, we define the variable Φ, as follows.

$$\Phi = S_m - D_m \quad (5)$$

Table II and Table III show the results of the examination. According to the result if Φ is close to 1 then it is a malware.

V. CONCLUSIONS AND FUTURE WORK

We developed a program preview system which can examine the characteristics of a program and detect malwares before their execution. This code graph system transforms a portable executable binary into a topological directed graph and matrix and then analyzes them. We distinguished malwares from normal programs using the code graph system effectively and the system was shown to work well under code obfuscation methods.

Most malware codes are packed with binary packaging techniques. Many kinds of packing techniques and unpacking techniques are in public and many malware codes use this public information. However, a malware code can be packed by its own packing method which is hardly to unpacking. The unpacking code contains themselves on the character of the packing technique. If we use this self unpacking code we can unpack unknown packing methods.

There are two ways to call system-calls on Windows. The representative method is to link libraries. The other method is to call the system-calls at run time. Currently, we use the former way to implement the code graph system, and the latter way will be solved with further research, since the names of the system calls in the code can be found in the latter way.

The code graph system can be used to analyze the distinction of other applications as well as malware. We show the first application of this method of detecting malware in this paper, but this approach could also be used to classify programs into groups and check to see if a function is made for a specific need.

Acknowledgments

This work was supported in part by the ITRC program of the Korea Ministry of Information & Communications, the Basic Research Program of the Korea Science & Engineering Foundation, and the Defense Acquisition Program Administration and Agency for Defense Development under the contract UD060048AD.

REFERENCES

[1] Arbor Networks, "Worldwide ISP Security Report," 2005.
[2] C.C. Zou, L. Gao, W. Gong, D. Towsley, "Monitoring and Early Warning for Internet Worms," Proc. of the 10th ACM Conf. on Computer and Communications Security, 2003.
[3] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, Andreas Terzis, "My Botnet is Bigger than Yours (Maybe, Better than Yours) : Why Size Estimates Remain Challenging," Proc. of HotBots, Apr. 2007.
[4] J. Newsome, B. Karp, D. Song, "Polygraph: automatically generating signatures for polymorphic worms," Proc. of IEEE Symp. on Security and Privacy, 2005.
[5] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, "Malware normalization," Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, November 2005.
[6] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, July 1997.
[7] M. Christodorescu, S. Jha, "Testing malware detectors," ACM SIGSOFT Software Engineering Notes, 2004.
[8] Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manuals," http://www.intel.com/products/processor/manuals/index.htm.