ISSN 1330-3651 (Print), ISSN 1848-6339 (Online) https://doi.org/10.17559/TV-20210202132203
Original scientific paper

Malicious Behavior Detection Method Using API Sequence in Binary Execution Path

Jihun KIM, Sungwon LEE, Jonghee YOUN*

Abstract: Today, the amount of malware is growing very rapidly, and the types and behaviors of malware are becoming very diverse. Unlike existing malicious codes, new types or variants of malicious codes are being identified, and it takes a lot of time to analyze all malicious codes. To solve these problems malware analysts analyze and research effective ways to reduce analysis time and cost. In this paper, we propose a method to express characteristics and detect malicious codes by using API Sequence for malicious code detection and classification. It compares and analyzes several existing expression methods and verifies the effectiveness through actual malicious code samples. Using the expression method proposed in the paper, we detected six malicious behaviors: DLL Injection, Downloader, IAT Hooking, Key Logger, Screen Capture and Antidebugging. As a result, more detection was detected than by conventional detection methods, and it can be seen that the more complex the malicious behavior, the higher the detection efficiency. In addition, static analysis was adopted as the main method, but because it searches execution compression, the flow of malicious behavior can be analyzed.

Keywords: API sequence; binary execution path; malware analysis; malware detection

1 INTRODUCTION

In 1949, Von Neumann published a theory that self-replication and proliferation can be performed in computers as with biological viruses [1]. Thereafter, those computer programs that are equipped with self-replication and proliferation functions started to be called "virus". However, since the term virus is quite limited in expressions, a concept that can comprehensively express the concept became necessary. The term made consequently is malware, which is an abbreviation of malicious software, which means soft-ware that contains malicious codes. Recently, with the rapid development of network and ICT technologies, the amount of malware has been increasing exponentially. To prevent and respond to such security threats, analysts and anti-virus program producers are trying to maximize the efficiency of malware analysis using various analysis methods.

In general, malware analysis is performed in two formats, static analysis and dynamic analysis [2]. First, static analysis is a technique to detect malicious behaviors by analyzing the structure of malware or specific binary patterns at the code level. Although static analysis enables more in-depth and detailed analysis, if technologies that obstruct static analysis such as executable file packing and code obfuscation are applied to malware, much time and effort will be required and the analysis will become considerably more difficult [3]. Second, dynamic analysis is a method of analyzing malicious behaviors by executing actual malware in a virtual machine. This method is advantageous in that malicious behaviors can be clearly observed even when executable file packing or code obfuscation has been applied to malware because malware is actually executed for analysis. However, it is not suitable for analyzing trigger-based malware that runs at a certain time or is executed when the user's specific action is taken.

In this paper, a malicious behavior detection method using static analysis based execution path searches was proposed to maximize the efficiency of malicious behavior detection. It aims to improve detection efficiency by detecting the execution path through static analysis and determining malicious behavior based on the correlation of APIs. Although the main analysis method is based on static analysis because it is conducted at the code level, the effects of dynamic analysis can be expected because it traces the binary execution path to grasp the behavior. Even if the malicious code does not function completely due to problems such as dependencies or has a packing problem, analysis is possible because it targets various branches in the binary code. In addition, the API behavior information in the analysis results will be visualized with graphs for clear under-standing of malicious behaviors and the excellence of the malicious behavior detection method proposed in this paper will be proved through comparison with the existing simple API collection and listing methods.

In Chapter 2, the methods of malicious behavior detection in previous studies are introduced and the limitations of those methods are mentioned. In Chapter 3, the method proposed in this paper is introduced. In Chapter 4, experiments are conducted based on the proposed method, the results are verified, and the efficiency of the method is mentioned and in Chapter 5, this paper is finished with conclusions.

2 RELATED WORKS

This paper checks the process and limitations of previous malware analysis research. It improves the limitations of the analysis method by referring to existing studies and suggests an efficient analysis method. Previous binary code based static analysis studies have been carried out by extracting the features of attributes in codes. Statistical algorithms were generally grafted on such studies and utilized for analysis. Such statistics are mainly utilized for comparison with normal programs. For instance, the statistics of op codes [4] or strings [5] extracted from the malware code sections are compared with those of general normal programs for utilization in analysis. These methods simply collect signatures and identify malicious behaviors by extracting information on the structures of executable files [6]. The most basic methods among the sequence-based malicious behavior identification methods mentioned above are those that list the sequences of op codes [7] or the sequences of strings [8]. These studies have been developed to carry out studies

that identify malicious behaviors by using the n-gram technique [9], which cuts the information in the file according to a certain standard and processes the cut pieces of information. In addition, studies intended to express the byte sequences for binary codes with n-grams with a view to classifying malware were also carried out [10]. Such methods of collecting signatures for the internal structure of a file can be defined with the unique DNA of the file [11] and are used for similarity and classification of malware based on the foregoing. A clear and definite basis for judging malicious behaviors is the discovery of the functions used by malware. Previous studies have attempted to identify or classify malware by processing such APIs within programs. [12-14] First, methods that list the sequences of APIs [15], or collect log information on the use of APIs to determine malicious behaviors [16] are representative. Since APIs are functions used when the program is executed, such APIs are either statically collected [17] or dynamically monitored [18]. In addition to the methods that simply list the APIs, there are methods that extract the features of malware according to the frequency of use of the APIs inside the file [19]. The above studies are statistical methods that have advantages such as not so large amounts of data to be stored, small amounts of operation, and high speed. However, they cannot respond to malware in real time and cannot accurately judge diverse malware behaviors because they are based on simple statistics [20]. To compensate for the foregoing, some studies carried out recently grafted various algorithms onto the statistical properties as such to detect malicious behaviors. The eigenvalues of op code based graph images can be calculated by measuring the distances between the nodes based on the K Nearest Neighbor Algorithm (KNN Algorithm), which is one of the machine learning algorithms [21]. In addition, the processed strings can be reprocessed with the Logistic Common Subsequence (LCS) algorithm to measure the eigenvalues of the strings [22]. In the studies introduced above, static analysis-based methods collect signatures or list the signatures in sequence, but cannot identify the accurate features of behaviors because they are based on code-based feature extraction. To compensate for this this problem, dynamic analysis is adopted as the main detection method [23] or a mixture of static and dynamic analyses is adopted [24]. However, since dynamic analysis is a method that directly executes malware for analysis, it has disadvantages of energy efficiency and analysis time [25]. In addition to this, several studies and methods are under way to classify malware [26, 27]. In our previous study, we studied how to express the features of malware using APIs [28]. In this paper, a method that is based on static analysis but tracks the execution flow was proposed so that the effect of dynamic analysis can be expected to compensate for studies in which static and dynamic analyses are mixed.

3 PROPOSED METHOD

3.1 Method Architecture

First, the malware executable file is converted into disassembled binary codes through the binary reverse engineering tool, IDA [29] (IDA PRO 6.6). The IDA extracts the disassembled code for the executable into .asm and uses it as data for static execution path search. Since malicious behaviors are determined by the API call sequences found in the execution path search, an API extraction process is undergone. In this study, graphical imaging is performed based on APIs. However, since not all the extracted APIs can be applied to imaging because they amount tens of thousands in kind, they are made into graph images through classification to represent malicious behaviors, and finally, the mutual similarity relations of pieces of malware are shown through image based similarity determining work.

3.2 Static Execution Path Exploration

The core of the static execution path search in this paper is that the instruction set and subroutine are divided into true and false ones according to the branch instruction before they are searched. First, among the assembly instructions, the instructions for branching are jz, jzr, and so on. As for the branch point, comparison instructions such as cmp and test that occur before the branch instructions are issued, are made through logical operation instructions such as xor. We divide true and false marks according to the branch instructions to search all instruction sets and subroutines.

The IDA's disassembly codes can identify the instruction sets and subroutines used in the search. The instruction sets, which are functions that perform some behaviors in the functions, are represented by loc_xxxxxx, and the subroutines or basic blocks are represented by special prefixes such as sub_xxxxxx. The IDA provides IDAPython, which is a python script, to provide powerful processing activities for binary codes. In this study, searches were performed using IDA APIs such as GetFunctionName, CodeRefsFrom, and CodeRefsTo [30].

Table 1 Example binary code to display static execution path search

1	loc_401460 :		
2		mov	eax, [esp + argc]
3		sub	esp, 44h
4		cmp	eas, 2
5		Push	ebx
6		push	ebp
7		push	ebi
8		jzn	loc_401488
9	loc_401484:		
10		xor	eax, eax
11		jmp	short loc_40148D
12	loc_401488:		
13		sbb	eax, eax
14		sbb	eax,0FFFFFFFFh

Tab. 1 is the binary code for displaying static execution path searches. After CMP instruction in the fourth line, the binary code loc_401460 is branched into the instruction sets of loc_40148 due to the JNZ in the eighth line, which is a branch instruction. If the result of the comparison is true, the binary code will be branched into loc_401488, and if false, into loc_401484. To summarize finally, loc_401460 is a binary code, which is branched into loc_401488 when it is true and into loc_401488 when it is false.

Visualizing the binary code in Tab. 1 will look like Fig. 1.

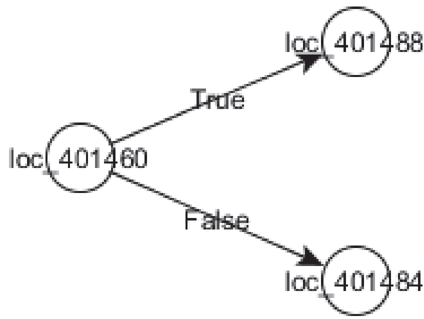

Figure 1 Visualization of binary code

This mechanism is applied equally even when there are subroutines in the instruction set. For example, if there is a subroutine in the instruction set as with the binary code shown in Tab. 2, the subroutine is entered and the above execution path search is performed.

Table 2 Subroutine in The instruction Set

1	start	Proc near	
2		…	
3		…	
4		cmp	eax, 6
5		jz	short loc_403366
6		push	ebx
7		call	sub_405B0E
8		…	…
9	loc_403366:		
10		mov	esi, offset
11	…	…	…
12	sub_405B0E		
13		…	…
14		test	eax,eax
15		jnz	short loc_405B32

The relevant binary code is visualized as shown in Fig. 2.

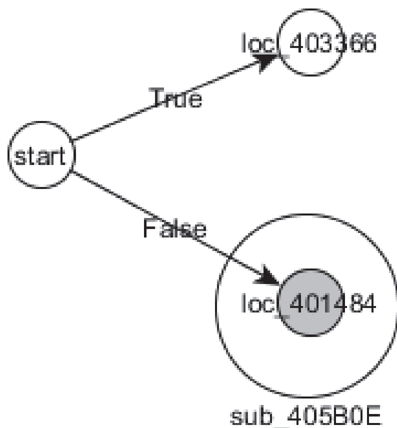

Figure 2 Visualization with subroutines

3.3 Processing of the RGL Scheme

In this study, malicious behaviors are detected based on Windows APIs. APIs are predetermined functions provided by the operating system or the programming language so that applications can use system resources or libraries. Applications use APIs for the purpose of using system resources or interacting with other applications and APIs are called from the program. Windows applications use Windows APIs in most cases and the APIs are included in dynamic libraries (DLLs). Those pieces of malware that are executed based on Windows also use the Windows APIs, and information on the APIs can be used as good information to determine malware behaviors.

In previous studies, pieces of malware API information were simply collected based on signatures or simply listed as with n-grams to detect the similarity and behaviors of malware. These methods are efficient for simple classification of pieces of malware and the detection of the variants of the relevant pieces of malware because they simply list the APIs but they have a shortcoming that they cannot accurately detect the behaviors of malware.

In this study, malicious behaviors will be detected based on the API information discovered during binary execution path searches according to the proposed method.

Table 3 Static Execution Path Exploration and APIs' Behaviors

1	loc_417AA4 :		
2		…	…
3		mov	[esp + 128Ch + var_1274], ebx
4		call	ds:CreateMutexA
5		mov	[esp + 128Ch + hObject], eax
6		call	ds:GetLastError
7		cmp	eax, 0b7h
8		jz	loc_41A5DD
9	loc_41A5DD:		
10		..	…
11		call	ds:CloseHandle
12		…	…
13		cmp	byte ptr [esi + 4], 0
14		jz	short loc_428986
15	loc_428986:		
16		…	…
17		call	ds:EnterCriticalSection
18		cmp	word ptr [ebx + 40h]
19		mov	[esp + 78h + var_64], 1
20		jnz	loc_4268DE
21	loc_4268DE:		
22		lea	ecx,, [esp + 78h + var_68]
23		call	sub_428930
24	sub_488930:		
25		…	
26		call	ds:LeavecriticalSection

Tab. 3 is an example of binary codes used to examine the behaviors of APIs in static execution path searches. The corresponding code branches from loc_4017AA4 into loc_41A5DD when it has been found to be true through the comparison instruction on the seventh line and loc_41A5DD branches into loc_428986 when it has been found to be true through the comparison instruction. The 15th through the 21st lines are the same search process as the one shown above and loc_4268DE is defined as a normal mark instead of true or false mark because the subroutine sub_488930 is simply called in line 23 without any comparison instruction. The mutual relationship between the instruction set and the subroutine of the binary code in Tab. 3 can be visualized as shown in Fig. 3 using the method proposed in this paper.

From the binary code in Tab. 3, it can be seen that Windows API functions such as CreateMutexA,

GetLastError, and CloseHandle are called in lines 4, 6, 11, 17, and 26 respectively. In this paper, the Normal, True, and False mark application mechanism is equally applied to APIs to analyze the interactions between the APIs and the behaviors of the APIs.

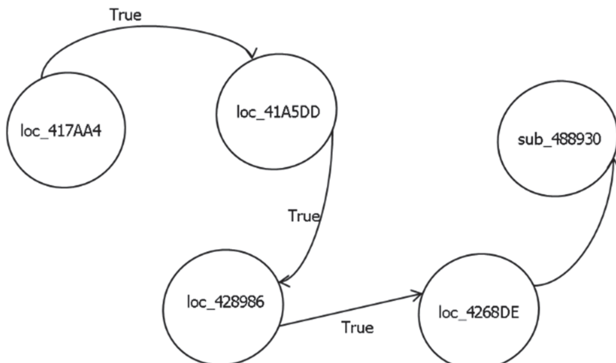

Figure 3 Search for the static execution path of the binary code

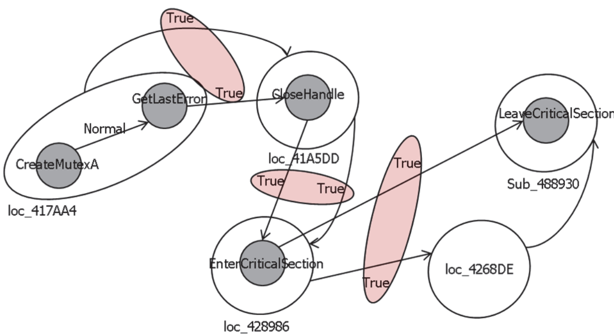

Figure 4 APIs' Interactions in the Binary Code

Fig. 4 is a figure visualized to show the interactions between the APIs of the binary codes shown in Tab. 3. CreateMutexA is called first in line 4 of Tab. 3, and GetLastError is called in line 6 thereafter. Since there is no instruction for branching by any comparison instruction between lines 4 and 6, the interactions are marked as normal between CreateMuetexA and GetLastError. Since the interaction between the instruction sets loc_417AA4 and loc_41A5DD is marked as true by lines 7 and 8, the interaction between GetLastError, which is called last in loc_417AA4, and GetLastErrorhe, which is called first in loc_41A5DD, is marked as true. One thing noteworthy is the interaction between EnterCriticalSection and LeaveCriticalSection. Even though it is marked as true in loc_428986 by loc_4268DE and loc_4268DE is marked as normal by sub_488930, since there is no instruction to call any API in loc_4268DE and no API is called between EnterCiritcalSection and LeaveCriticalSection, it is marked as true. Finally, only the interactions between APIs are visualized as shown in Fig. 5.

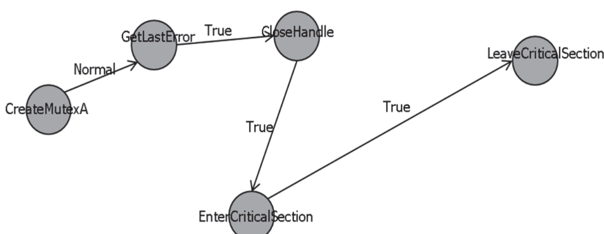

Figure 5 APIs' Interactions Decision

3.4 API Classification

In this study, malicious behaviors are visualized in the form of graphs expressed with nodes and intermediate lines as shown in Fig. 5. However, there is a problem that the number of APIs is too large to make the APIs into nodes. In this study, to solve such problems, the APIs will be reclassified into 24 upper categories through the functions of the APIs so that behaviors can be clearly judged and the temporal efficiency can be enhanced [31, 32]. For instance, CreateFile and CreateProcess are APIs that perform functions related to "files" or "processes" and APIs such as GetSystemTime and GetLocalTime have the function to collect information on "time" in the system. In addition, APIs such as strcmp and stcat all perform functions related to strings. Such a classification not only has many categories to which APIs commonly belong although they have been already classified in MSDN but also is too abstract to understand behaviors. For instance, all process-related APIs are included in the category process but whether the relevant APIs created, deleted, or accessed processes cannot be known. Therefore, the functions of APIs were reclassified into three, which are CREATE_OR_OPEN, READ_OR_ACCESS, and CLOSE. Tab. 4 shows the final 24 API categories.

Table 4 API Categorization

FILE-CREATE_OR_OPEN	FILE-READ_OR_ACCESS	FILE_CLOSE
PROCESS-CREATE_OR_OPEN	PROCESS -READ_OR_ACCESS	PROCESS _CLOSE
NETWOKR -CREATE_OR_OPEN	NETWOKR -READ_OR_ACCESS	NETWOKR _CLOSE
REGEDIT-CREATE_OR_OPEN	REGEDIT -READ_OR_ACCESS	REGEDIT _CLOSE
SERVICE	STRING	DEBUGGING
RESOURCE	TIME	MUTEX
WINDOW-GUI-AND-BITMAP	SHELL-AND-CONSOLE	THREAD
STSTEM-INFORMATION	LIBRARY	HANDLE

However, not all APIs are reclassified into four behaviors. Since APIs such as Strcat and strcmp do not perform the function to create or access strings separately, APIs related strings are determined to be in a single category, and APIs related to "time" such as GetLocalTime and GetSystemTime are included in the category Time despite the fact that they access the system to obtain time information, because the behavior "Time" has the most important value in the identification of malicious behaviors.

4 EXPERIMENTS AND DISCUSSION

4.1 Malicious Behavior Detection

In this study, malicious behaviors are detected based on the interactions between APIs using binary static execution path searches. Previous API sequence based static analysis studies had a shortcoming of being unable to accurately understand malware behavior because they simply listed or collected APIs. However, the method proposed in this study enables the understanding of the interactions between APIs because it uses execution path

searches despite the fact that it is a static analysis so that the effects of dynamic analysis can be expected.

As a representative example, when the malicious behavior of Trojan.Graftor.D4C56B has been analyzed by the method proposed in this paper, the graphic image shown in Fig. 6 appears.

Fig. 6 shows that the malware uses APIs such as String, SystemInformation, Module, and Process. In particular, a detailed analysis of the red shaded API behaviors is as follows. In light of the fact that the relevant APIs use processes such as OpenProcess, Process32Next, WriteProcessMemory, and VirutalAllocEx and APIs used for manipulation of dlls, the API behaviors can be confirmed as DLL injection that inserts code into the remote process of calling LoadLibrary to forcibly make the DLL to be loaded into the context of the relevant process.

4.2 Comparison with Dynamic Analysis

With regard to the behaviors shown in 4.1, the existing simple API collection and listing method, the API Monitor based [33] dynamic analysis, and the method proposed in this paper are compared as shown in Tab. 5.

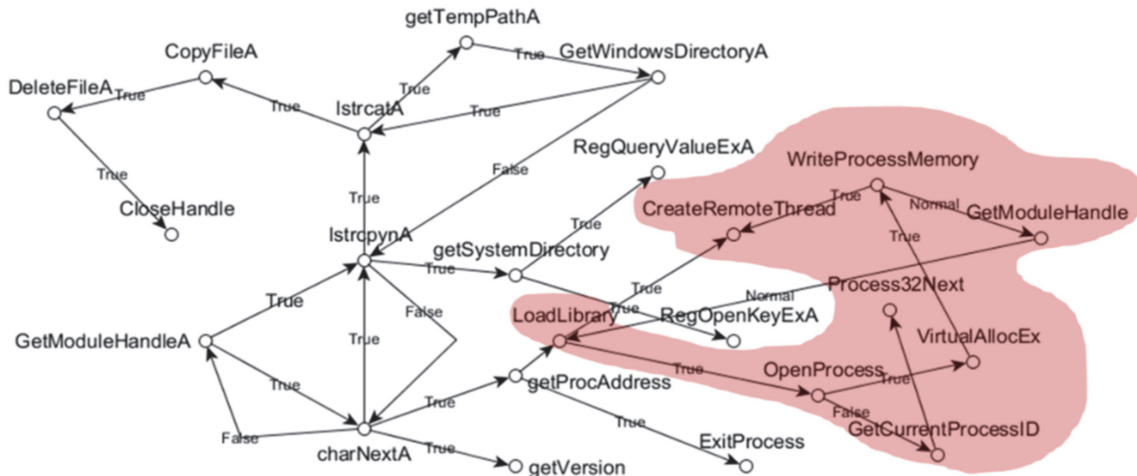

Figure 6 Trojan.Graftor.D4C56B's malicious behavior

Table 5 Comparison with Dynamic Analysis

API Sequence listing method	….LoadLibrary **VirtualAllocEx** lstrcmpA … **OpenProcess** GetCurrentProcessID **Process32Next** … **GetModuleHandleA** **GetProcAddress** getCurrentProcess … **CreateRemoteThread** …
Dynamic analysis	… NtCreateMutant … Process32Next **OpenProcess VirtualAllocEx GetProcAddress WriteProcessMemory CreateRemoteThread** … NtClose …
Proposed method	...**OpenProcess → (True)VirtualAllocEx, → (True)WriteProcessMemory → (Normal)GetModuleHandleA → (Normal) CreateRemoteThread** …

The previous sequence listing method can be shown to have API sequences different from those that appear in the dynamic analysis that directly executes APIs to analyze. However, the API sequences in the method proposed in this paper can be identified to be similar to those appearing in dynamic analysis.

4.3 Common Graph of Malicious Behaviors

In this study, common graphs of representative malicious behaviors (Dll injection, Downloader, IAT Hooking, Key Logger, Screen Capture, Antidebugging) were identified as shown in Tab. 6 in the same method.

Table 6 Malicious Behavior in Categorization of API

Malicious behavior	Common behavior graphs
DLL Injection	OpenProcess, True, VirtualAllocEx, True, WriteProcessMemory, True, LoadLibarary, Normal, Normal, CreateRemoteThread
Downloader	URLDownladToFile, True, WinExec or ShellExecute
IAT Hooking	LoadLibrary, True, String API, True, VirtualProtect
Key Logger	FindWindowA, True, SetWindowsHookEx; ShowWindow, True, UnhookWindowsHookEx
Screen Capture	GetDC or GetWindowDC, True, CreateCompatibleDC, True, CreateCompatibleBItmap, True, SelectObject, True, BitBlt, Normal, WriteFile
Antidebugging	IsDebuggerPresent, False, True, CheckRemoteDebuggerPresent, True, OutputDebugStringA

Tab. 6 shows the interactions of the behaviors of all APIs. These actions of behaviors are shown after being combined by the categorization of APIs as shown in Tab. 7 for the clarity of analysis methods and the efficiency of analysis time.

Table 7 Malicious Behavior in Categorization of API

Behavior	DLL injection
Behavior Grpah Image	PROCESS-READ_OR_ACCESS, LIBRARY, RESOURCE, THREAD, True, True, Normal
Sequence	PROCESS-READ_OR_ACESS→(TRUE)RESOURCE→(TRUE) LIBRARY)→(NORMAL)THREAD
Behavior	Downloader
Behavior Grpah Image	NETWORK-READ_OR_ACCESS, LIBRARY, True
Sequence	NETWORK-READ_OR_ACESS→(TRUE)LIBRARY
Behavior	IAT Hooking
Behavior Grpah Image	LIBRARY, STRING, RESOURCE, True, NORMAL
Sequence	LIBRARY→(TRUE)STRING→(NORMAL)RESOURCE
Behavior	KeyLogger
Behavior Grpah Image	WINDOW-GUI-BITMAP, HOOK, True
Sequence	WINDOW-GUI-BITMAP→(TRUE)HOOK

4.4 Efficiency

DLL injection, downloader, IAT hooking, key logger, screen capture, and anti-debugging. The studies being compared are those that simply collect or list op codes or APIs such as OPCODE, N-gram and API sequence.

Table 8 Example Binary Code

start :	push	esi
	mov	esi, [esp + 4 + arg_0]
	push	edi
	shl	esi, 3
	mov	edi, off_409068
	push	edi
	call	ds: GetModuleHandleA
	test	eax,eax
	jnz	short loc_405B32
	push	edi
	call	sub_405AA0
	test	eax, eax,
	jz	short loc_405B41
loc_405B32:		
	push	off_40906C
	push	eax
	call	ds: GetProcAddress

In this paper, we show the efficiency of the proposed method compared with previous simple signature information collections. Tab. 9 shows seven methods to simply list or collect code-level information such as OPCODE, N-GRAM, BYTE CODE, API, string and etc.

In addition, an example of applying each of the methods to Tab. 8 code is also shown.

The test set is 1236 pieces of randomly generated malware and all of them include an IAT (Import address Table) because the method proposed in this paper analyzes the interactions between APIs. First, through the identified common behavior graphs, each malicious behavior was analyzed based on the data set consisting of 1,236 pieces of malware.

Figs. 7 to 12 are graphs comparing the method proposed in this study and the existing method. It can be seen that the proposed methods show larger numbers of detection of the malicious behaviors, DLL injection, IAT Hooking, Screen Capture, and Anti Debugging when compared to the existing detection methods.

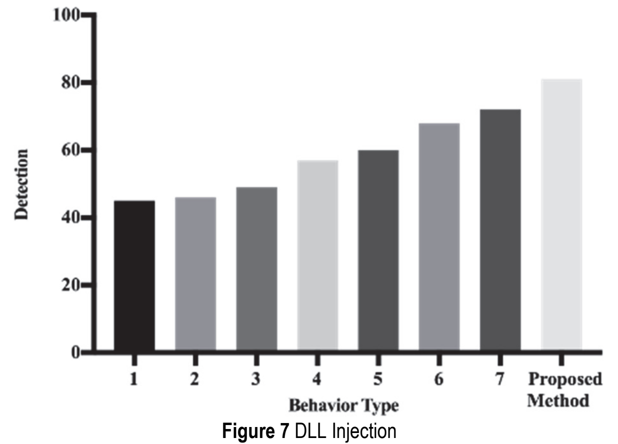

Figure 7 DLL Injection

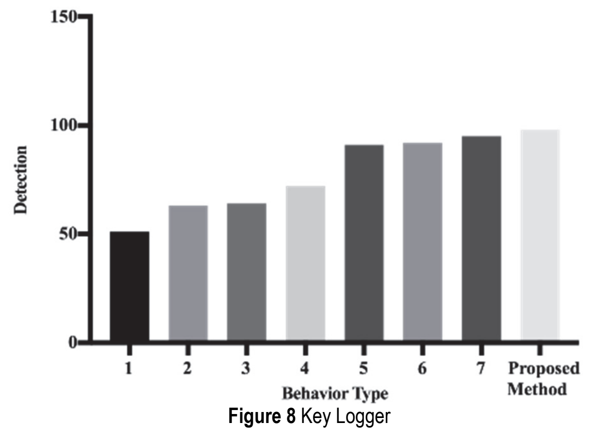

Figure 8 Key Logger

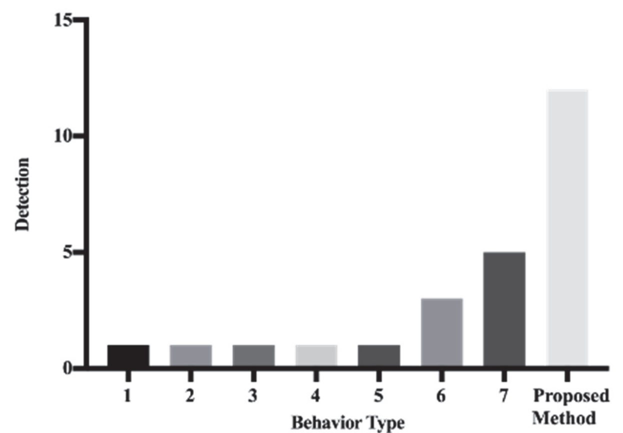

Figure 9 Screen Capture

Table 9 Comparison of previous method

No.	Analysis Method	Example
1	OPCODE	pushmovpushshlmovpushcall…
2	OPCODE, 3-GRAM	pusmovpusshlmovpuscaltesjnz…
3	OPCODE, N-GRAM	P(n-gram)m(n-gram)p(n-gram)…
4	API Listing	GetModuleHandleAGetProcAddress…
5	API Listing, N-GRAM	G(n-gram)G(n-gram)….
6	API Frequency	GetModuleHandleA(1)GetProcAddress(1)…
7	STRING Listing, N-gram	Inst(n-gram)Erro(n-gram)uxth(n-gram)… (The string is not visible in Tab. 8)

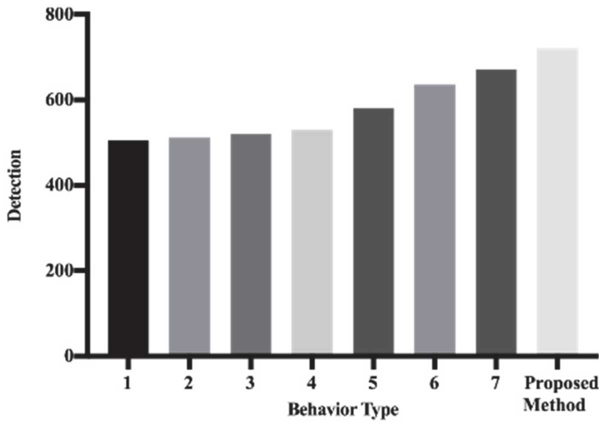

Figure 10 Anti Debugging

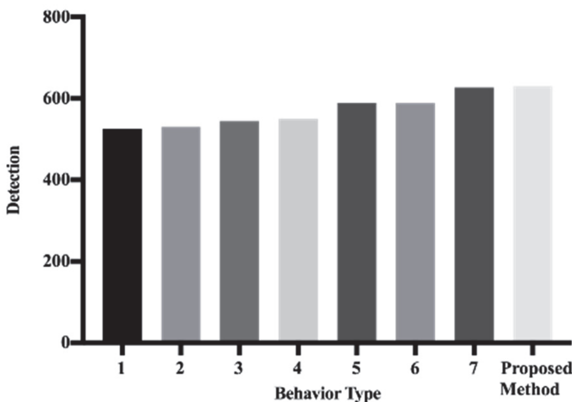

Figure 11 Downloader

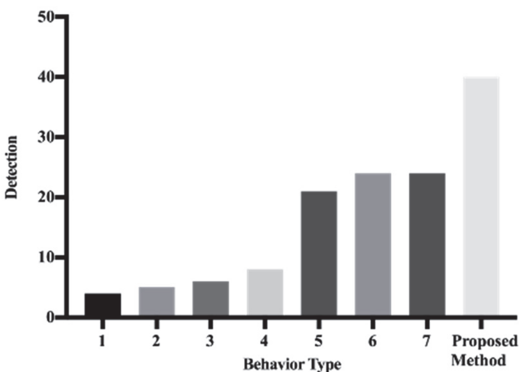

Figure 12 IAT Hooking

However, it can be seen that the method proposed in this study shows very similar numbers of detection of other malicious behaviors such as Downloader and key logger when compared to the existing detection methods. This is because the relevant behaviors perform API interactions that are too simple to make the relevant behaviors to be judged to be malicious. In other words, the two malicious behaviors, Downloader and Key Logger are composed of two nodes, and the marks of the intermediate lines that show the interactions are True, so that only one sequence of each of the relevant behaviors is identified. This means that the structure of the sequence is too simple to detect malicious behaviors, which is the reason why the accuracy of detection is lowered. However, it can be seen that the more complex malicious behavior, the higher the detection efficiency.

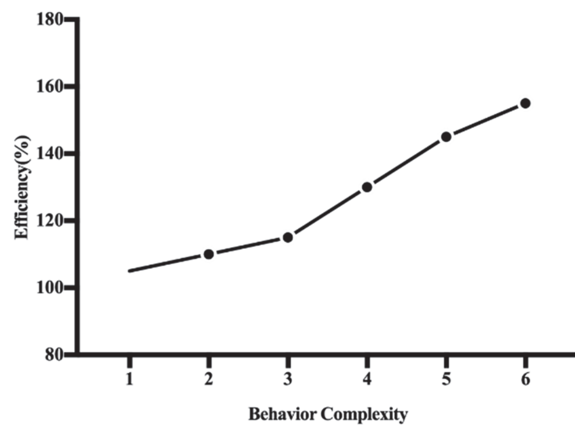

Figure 13 Detection efficiencies according to behavior complexity

Fig. 13 is a graph showing the detection efficiencies according to the behavior complexity. The relevant efficiencies shown in the graph are the efficiencies of the method proposed in this paper in comparison with the highest efficiencies shown in the existing studies. The complexity of malicious behaviors in this study paper may also be regarded as the complexity of sequences. The efficiency of the method proposed in this paper was shown to be 105% compared to existing studies when the complexity of sequences was low because the grounds for detection are reduced when the complexity decreases. On the contrary, the efficiency of the method proposed in this paper was shown to be 158% compared to existing studies when the complexity of sequences was high because the grounds for detection increase when the complexity increases. This means that the more complex the malicious behaviors, the higher the efficiency of detection.

4.5 Binary Classification Result

When compared to previous studies, the method proposed in this paper did not show high detection rates for behaviors such as Downloader and KeyLogger. This is because the relevant behaviors conduct simple API interactions so that grounds for judgment as being malicious are insufficient. However, it can be seen that the higher the complexity of malicious behaviors, the higher the detection efficiency of the method proposed in this

paper. These results can be proved based on the accuracy and f-measure values based on the wrong detection rates and detection missing rates of existing studies.

In this paper, we compared the proposed method with previous studies and binary classification results. Since this paper is an API sequence based on static analysis, all previous comparative studies are based on static analysis. [19, 34] used API frequency, and [35, 36] used API sequence as the main method. All the accuracy and f-measure values of the method proposed in this paper were measured to be higher compared to previous studies. In addition, the pieces of malware detected by the method proposed were identified to show an average Virustotal [37] detection rate of 69%.The summary of the contents can be found in Tab. 10.

Table 10 Binary Classification Result

Previous Works	Used Method	Malware Sample	accuracy	*f*-measure	Virus Total Average
[19]	API Frequency	66,703	0.985	0.984	Not mentioned
[33]	API Frequency	32,000	0.983	0.878	
[34]	API Sequence	800	0.841	0.909	
[35]	API Sequence	17,366	0.0.930	0.941	
Proposed Method	API Sequence	1,236	0.991	0.992	69

5 CONCLUSIONS AND FUTURE WORK

In this paper, a method to detect execution paths based on static analysis and judge malicious behaviors based on APIs' interrelationships was proposed. Although static analysis is the main analysis, the method proposed in this paper enables analyzing the flow of behaviors because it searches execution paths. This means that although static analysis is adopted as a main method, the advantages of dynamic analysis that directly executes APIs to analyze the APIs are applied to the method proposed in this paper. In this study, execution flows were analyzed according to branch instructions and the interactions of APIs collected during the flows were analyzed. API interactions are marked as normal, true, and false and are reclassified into and listed as 24 upper categories. In this study, the detection method based on the relevant method was compared with the existing simple API collecting method and API listing method. The malicious behaviors used for the comparison are six behaviors, which are dll injection, downloader, IAT hooking, key logger, screen capture, and anti-debugging. The method proposed in this paper showed high efficiencies in the discrimination of four behaviors among the six behaviors except for downloader and the key logger. This is because the API interactions of downloader and key logger are insufficient for judgment of the behaviors as being malicious. This is related to the complexity of malicious behaviors. As malicious behaviors became more complicated, higher efficiencies of detection appeared because the grounds for judgment of malicious behaviors became more sufficient. In future studies, the frequencies of behaviors will be added to prepare grounds for judgment of detailed behaviors. The utilization of such numerical data can be extended to apply machine learning and various statistics based algorithms, and based on such data, malware will be visualized and malware similarity will be calculated.

Acknowledgements

This research was supported by the 2018 Yeungnam University Research Grant (218A061016, 218A380138) and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2018R1D1A1B07050647).

6 REFERENCES

[1] Neumann, J. & Burks, A. W. (1966). *Theory of self-reproducing automata*. Urbana: University of Illinois press.

[2] Gandotra, E., Bansal, D., & Sofat, S. (2014). Malware analysis and classification: A survey. *Journal of Information Security*, 2014. https://doi.org/10.4236/jis.2014.52006

[3] Sharif, M. I., Lanzi, A., Giffin, J. T., & Lee, W. (2008). Impeding Malware Analysis Using Conditional Code Obfuscation. *NDSS*.

[4] Bilar, D. (2007). Opcodes as predictor for malware. *International journal of electronic security and digital forensics, 1*(2), 156-168. https://doi.org/10.1504/IJESDF.2007.016865

[5] Griffin, K., Schneider, S., Hu, X., & Chiueh, T. C. (2009). Automatic generation of string signatures for malware detection. *International workshop on recent advances in intrusion detection*, 101-120. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-04342-0_6

[6] Shafiq, M. Z., Tabish, S. M., Mirza, F., & Farooq, M. (2009, September). Pe-miner: Mining structural information to detect malicious executables in real time. *International workshop on recent advances in intrusion detection*, 121-141. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-04342-0_7

[7] Santos, I., Brezo, F., Nieves, J., Penya, Y. K., Sanz, B., Laorden, C., & Bringas, P. G. (2010, February). Idea: Opcode-sequence-based malware detection. *International Symposium on Engineering Secure Software and Systems*, 35-43. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-11747-3_3

[8] Hu, K. G. S. S. X. & Chiueh, T. C. (2008). Automatic Generation of String Signatures for Malware Detection. *Symantec Research Laboratories*, 1-29.

[9] Santos, I., Penya, Y. K., Devesa, J., & Bringas, P. G. (2009). N-grams-based File Signatures for Malware Detection. *ICEIS, 9*(2), 317-320. https://doi.org/10.5220/0001863603170320

[10]Moskovitch, R., Feher, C., Tzachar, N., Berger, E., Gitelman, M., Dolev, S., & Elovici, Y. (2008). Unknown malcode detection using opcode representation. *European conference on intelligence and security informatics*, 204-215. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-89900-6_21

[11] Choi, Y. H., Han, B. J., Bae, B. C., Oh, H. G., & Sohn, K. W. (2012). Toward extracting malware features for classification using static and dynamic analysis. *8th International Conference on Computing and Networking Technology (INC, ICCIS and ICMIC)*, 126-129).

[12] Zhang, M., Duan, Y., Yin, H., & Zhao, Z. (2014). Semantics-aware android malware classification using weighted

contextual api dependency graphs. *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 1105-1116. https://doi.org/10.1145/2660267.2660359

[13] Lu, H., Wang, X., & Su, J. (2013). SCMA: Scalable and collaborative malware analysis using system call sequences. *International Journal of Grid and Distributed Computing, 6*(2), 11-28.

[14] Elhadi, A. A. E., Maarof, M. A., & Barry, B. I. (2013). Improving the detection of malware behaviour using simplified data dependent API call graph. *International Journal of Security and Its Applications, 7*(5), 29-42. https://doi.org/10.14257/ijsia.2013.7.5.03

[15] Uppal, D., Sinha, R., Mehra, V., & Jain, V. (2014, September). Malware detection and classification based on extraction of API sequences. *International conference on advances in computing, communications and informatics (ICACCI)*, 2337-2342. https://doi.org/10.1109/ICACCI.2014.6968547

[16] Fan, C. I., Hsiao, H. W., Chou, C. H., & Tseng, Y. F. (2015). Malware detection systems based on API log data mining. *39th annual computer software and applications conference, 3*, 255-260. https://doi.org/10.1109/COMPSAC.2015.241

[17] Alazab, M., Venkataraman, S., & Watters, P. (2010, July). Towards understanding malware behaviour by the extraction of API calls. *Second cybercrime and trustworthy computing workshop*, 52-59. https://doi.org/10.1109/CTC.2010.8

[18] Rajagopalan, M., Hiltunen, M. A., Jim, T., & Schlichting, R. D. (2006). System call monitoring using authenticated system calls. *IEEE Transactions on Dependable and Secure Computing,* 3(3), 216-229. https://doi.org/10.1109/TDSC.2006.41

[19] Alazab, M., Venkatraman, S., Watters, P., & Alazab, M. (2010). Zero-day malware detection based on supervised learning algorithms of API call signatures.

[20] Moser, A., Kruegel, C., & Kirda, E. (2007). Limits of static analysis for malware detection. *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 421-430. https://doi.org/10.1109/ACSAC.2007.21

[21] Firdausi, I., Erwin, A., & Nugroho, A. S. (2010, December). Analysis of machine learning techniques used in behavior-based malware detection. *Second international conference on advances in computing, control, and telecommunication technologies*, 201-203. https://doi.org/10.1109/ACT.2010.33

[22] Blount, J. J., Tauritz, D. R., & Mulder, S. A. (2011, July). Adaptive rule-based malware detection employing learning classifier systems: a proof of concept. *35th Annual Computer Software and Applications Conference Workshops*, 110-115. https://doi.org/10.1109/COMPSACW.2011.28

[23] Nair, V. P., Jain, H., Golecha, Y. K., Gaur, M. S., & Laxmi, V. (2010). Medusa: Metamorphic malware dynamic analysis using signature from api. *Proceedings of the 3rd International Conference on Security of Information and Networks*, 263-269. https://doi.org/10.1145/1854099.1854152

[24] Roundy, K. A. & Miller, B. P. (2010). Hybrid analysis and control of malware. *International Workshop on Recent Advances in Intrusion Detection*, 317-338. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-15512-3_17

[25] Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2008). A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR), 44*(2), 1-42. https://doi.org/10.1145/2089125.2089126

[26] Sharma, A. & Sahay, S. K. (2016). An effective approach for classification of advanced malware with high accuracy. https://doi.org/10.14257/ijsia.2016.10.4.24

[27] Hordri, N. F., Ahmad, N. A., Yuhaniz, S. S., Sahibuddin, S., Ariffin, A. F. M., Saupi, N. A. M., Senan, M. F. E. M., et al. (2018). Classification of malware analytics techniques: a systematic literature review. *International journal of security and its applications, 12*(2), 9-18. https://doi.org/10.14257/ijsia.2018.12.2.02

[28] Jihun, K., Sung, W. L., & Jonghee, Y. (2021). Expression of malware characteristics using API sequence. *Journal of Smart Technology Applications, 2*(1).

[29] Eagle, C. (2011). *The IDA pro book*.

[30] See https://www.hexrays.com/products/ida/support/ idapython _docs/

[31] Zhou, B., Xia, X., Lo, D., Tian, C., & Wang, X. (2014). Towards more accurate content categorization of API discussions. *Proceedings of the 22nd International Conference on Program Comprehension*, 95-105. https://doi.org/10.1145/2597008.2597142

[32] Uppal, D., Sinha, R., Mehra, V., & Jain, V. (2014). Exploring behavioral aspects of API calls for malware identification and categorization. *International Conference on Computational Intelligence and Communication Networks*, 824-828. https://doi.org/10.1109/CICN.2014.176

[33] See http://www.rohitab.com/apimonitor

[34] Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., & Hamze, A. (2010). Malware detection based on mining API calls. *Proceedings of the 2010 ACM symposium on applied computing*, 1020-1025. https://doi.org/10.1145/1774088.1774303

[35] Sathyanarayan, V. S., Kohli, P., & Bruhadeshwar, B. (2008). Signature generation and detection of malware families. *Australasian Conference on Information Security and Privacy*, 336-349. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-70500-0_25

[36] Ye, Y., Wang, D., Li, T., & Ye, D. (2007). IMDS: Intelligent malware detection system. *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 1043-1047. https://doi.org/10.1145/1281192.1281308

[37] See https://www.virustotal.com

Contact information:

Jihun KIM, M.S.
Dept. of Computer Engineering, Yeungnam University,
280 Daehak-Ro, Gyeongsan, Gyeongbuk, Republic of Korea
E-mail: f13521@naver.com

Sungwon LEE, M.S.
Dept. of Computer Engineering, Yeungnam University,
280 Daehak-Ro, Gyeongsan, Gyeongbuk, Republic of Korea
E-mail: noke15@ynu.ac.kr

Jonghee YOUN, PhD, Professor
(Corresponding author)
Dept. of Computer Engineering, Yeungnam University,
280 Daehak-Ro, Gyeongsan, Gyeongbuk, Republic of Korea
E-mail: youn@yu.ac.kr