International Journal of Grid and Distributed Computing
Vol. 10, No. 12 (2017), pp.125-136
http://dx.doi.org/10.14257/ijgdc.2017.10.12.12

Software Similarity Detection Scheme Exploiting Executable Fingerprint

Su-Jin Oh[1], Jin Kim[1], Jeong-Gun Lee, Sung-Bong Jang[2] and Young-Woong Ko[1]*

[1]*Department of Computer Engineering, Hallym University, Chuncheon, Gangwon, 200-702, Republic of Korea*
[2]*Department of Industry-Academy Cooperation, Kumoh National Institute of Technology, 61 Daehak-ro, Gumi, Kyoung-Buk, 39177, Republic of Korea*
yuko@hallym.ac.kr

Abstract

Windows Operating System is widely used and it is dominant than other operating systems. Most of the commercial software developing companies provide a number of software mainly for Windows OS users. Furthermore, the number of software plagiarism cases and copyright infringement cases has increased. Therefore, software similarity analysis systems for Windows OS are highly useful for software plagiarism detection and copyright protection. However, mostly the software similarity analysis systems and techniques work on source codes, not executables, even though the majority of commercial software barely makes public its source codes. In this paper, we propose a novel scheme for software plagiarism detection for Windows executables. Our technique exploits both the extended basic block concept and the Linux command objdump. We measured the performance of proposed system through several kinds of experiments using various types of software including different versions of commercial software.

Keywords: *Similarity; Executable; Forensic; Software Plagiarism; Commercial software*

1. Introduction

In the operating system market share, the percentage of Windows OS series is still dominant than that of other operating systems in the world. Besides, a number of software engineers and commercial software developing companies typically consider Windows OS users as their potential clients. For this reason, there are a considerable number of software which are based on the Windows OS environment. Through that, the number of software plagiarism and copyright infringement cases related to the Windows environment is going high. To contribute for solving these issues, many researchers have focused on the field of software plagiarism detection and the related works on the Windows environment. Nonetheless, most of the existing techniques for software plagiarism detection require source codes [1][2], even though commercial software developing corporations hardly release their source codes.

Thus, using binary contents in software can be effective to design an efficient software plagiarism detecting technique. Furthermore, the techniques based on binaries can be exploited for different purposes, such as code refactoring, bug detection [3][4][5] and malware detection [6][7]. Basically, the detecting techniques based on binary contents analyze all the binary information in the software. However, the analysis process for binary contents is quite complex because of the compilation process. In the compilation process, the compiler reconstructs the given high-level instructions into a number of low-level instructions, which are called as binary codes. Simply, a bunch of binary codes

Received (October 16, 2017), Review Result (December 9, 2017), Accepted (December11, 2017)

ISSN: 2005-4262 IJGDC

indicate all the execution processes of the corresponding software. For this reason, conducting analysis binary codes can be useful to distinguish whether a specific software is related to another one or not.

In this paper, we proposed a fingerprint technique to detect software plagiarism for binary executables in the Windows OS environment. Our proposed scheme exploits binary codes in .text section of the PE executable. The key idea of our scheme is to reflect the similarity of executable code by eliminating duplicated opcode information and exploiting the extended basic block concept. With this approach, we can find similar executable files. To evaluate our scheme, we implemented the fingerprint technique on a Linux system. Normally, Linux platform provides various open source code for implementing file similarity analysis software. Especially, we exploit the Linux command *objdump* to disassemble PE executables. Furthermore, to evaluate the suggested system, we conduct experiments using different versions of commercial software in the Windows environment and MSDN [8] developer code samples. The experiment result shows that the proposed executable fingerprint scheme is effective to find whether a binary executable (Intel Executable [9]) is derived from another executable or not.

2. Related Works

In recent years, there are a lot of studies related to executable file similarity analysis that is closely associated with the binary clone detection and the malicious software detection. These all topics aim to distinguish whether a binary executable is associated with another specific software or not. Especially, in the case of malware detection, a suspected software is decided by its signatures, which are a sum of strings what it has or particular sequences of its binary codes, or its behavior. Here are some works that aim to binary detection including clone, plagiarism, and malware.

Sæbjørnsen suggests a code clone detection system for binary executables [3]. They use both opcodes and operands in disassembled binary codes. Especially, they generate normalized features and feature vectors. After that, they conduct locality sensitive hashing (LSH) technique and some special processes for improving the correctness. Besides, they show their result visually. Bonfante proposes a malware detection scheme exploiting morphological approach [6]. They utilize CFG as a signature for classifying malware and achieve to analysis a suspected software in both syntactic and semantic ways. For reducing the false positive ratio, they gathered a bunch of malware collections. Hence, their false positive ratio became under 0.1%. In Deckard system, tree-based code clone detection scheme is proposed [10]. Their work is based on source codes and provides scalability with a high degree of accuracy. Particularly, they use the Euclidean distance metric for generating practical feature vectors for clone detection. On the other hand, Koschke suggests a syntax suffix tree scheme for detecting clones in a bunch of abstract syntax trees [11]. As we mentioned before, clone detection techniques are associated with the malware detection technique as well. Rieck [12] proposes an automatic analysis system applying machine learning algorithm [12][13][14]. Their system classifies unknown malware through behavior-based analysis and machine learning. Besides, this topic has extended to the malicious application detection in smartphones [15][16][17][18]. Here are some recent studies looking for source code similarity. CCFinder [5] detects token-based code clone for large scale source code. The key idea of this system is to transform input source text and analyze using token-by-token comparison. With this approach, it can extract code clones in the multi-linguistic environment. In [19], they focus on C or C++ language, while [20] is about Java language. In [21][22], behavior-based scheme is introduced. Their aim is deciding whether an unknown binary is a variant of another malware or not. In

[21], they adapt random forests classifier for performing both malware detection and family classification. Also, they used feature representations for malware classification to enhance predictive performance while reduces the feature space. The proposed system shows high malware detection rate and promising predictive performance in the family classification. For design a practical classification technique, [22] generates multi-function behavior dependency chains, where they implemented a dynamic analysis framework which can capture all the malware behaviors. Especially, they capture API functions while the malware running by exploiting registry, service, and process information. Furthermore, they reflect the input parameters, output parameters and return value for the malware behavior analysis.

3. System Architecture

In this section, we explain some basic knowledge related to our proposed scheme. At the first, we will introduce the PE format. After that, we will introduce the analysis process of binary instructions of a PE executable and the concept of CFG.

3.1 PE (Portable Executable) File Format

The portable executable (PE) file format is a derived version of the UNIX common object file format (COFF). It is a file format for several types of files for Windows OS, such as executable files, object files, and dynamic-link libraries. The PE format is one kind of binary data structure for storing all of the paramount information. Thus, Windows OS loader can manage them. The PE executable format can be divided into two types that PE32 for the 32-bit Windows version and PE32+ for the 64-bit version. At this moment, the latest revision is the 8.3 version, and it was updated on February 6th, 2013. The MSDN web page has continuously released the information **Error! Reference source not found.**. A PE file consists of several headers and some sections. Among them, DOS header and DOS stub exist for MS-DOS compatibility. The size of DOS header is 64-byte, while that of DOS stub is unfixed. Thereby, the last 4-byte of DOS header, the location is 0x3C, specifies the pointer to PE header. DOS stub has a code which can be run under MS-DOS. When MS-DOS tries to execute a PE file, then its DOS stub shows a message like “This program cannot be run in DOS mode.”

PE header consists of three parts that the field Signature, the COFF file header, and the optional header. The size of the field Signature is 4-byte, and its content is “PE\0\0.” The size of COFF file header is 20-byte, and it specifies some information. Especially, both *NumberOfSections* field and *SizeOfOptionalHeader* field are quite important for indicating the following headers and sections. The field *SizeOfOptionalHeader* is required for executable files, not for object files. Thus, in an object file, this field is filled with zero. Optional header is optional, in the literal sense of the word that object files do not have an optional header. Some fields in this header are significant for static analysis of PE executable files. The field Magic shows whether a PE file follows the PE32 format or the PE32+ format. This factor affects some fields’ length and the instruction code in PE executable file. The field *SizeOfCode* indicates the size of the .text section which contains all of the actual instructions. The field *NumberOfRvaAndSizes* describes the number of data directory entries in the optional header. Data directory is located in the last part of the optional header, and it consists of 16 entries. The size of each entry is 8-byte, and it is composed of two fields that *VirtualAddress* and Size. Furthermore, the last entry is filled with zero. The following table illustrates each data directory entry’s name and its respective index.

Table 1. Data Directory Entries

Index, Directory Name	Index, Directory Name
0, Export Directory	1, Import Directory
2, Resource Directory	3, Exception Directory
4, Security Directory	5, Base Relocation Table
6, Debug Directory	7, Architecture Specific Data
8, Relative Virtual Address of Global Pointer	9, Thread Local Storage Directory
10, Load Configuration Directory	11, Bound Import Directory in Headers
12, Import Address Table	13, Delay Load Import Descriptors
14, COM Runtime Descriptor	15, Reserved Directory

The section header is followed by the optional header. This header has some entries which are 40-byte, respectively. In addition, the number of entries is already specified by the field *NumberOfSections* in the optional header. Each entry consists of several fields, and the following four fields are vital for static analysis. The Name field shows the corresponding section's name. *VirtualAddress* indicates the address of the first byte of the corresponding section when it is loaded into memory. *SizeOfRawData* and *PointerToRawData* describe the relevant section's size and its file pointer, respectively.

3.2. Disassembly Process

Normally, there are two types of instruction sets that the reduced instruction set computer (RISC) and the complex instruction set computer (CISC). In the RISC architecture, there are a few number of fixed-length instructions. Besides, all the instructions in the RISC architecture are operated during a single-clock cycle. Aside from that, when a complex operation is performed in the RISC architecture, it should be treated by a bunch of RISC instructions. On the other hand, the CISC architecture is very complex. The length of each CISC instruction is inconstant. Furthermore, the CISC architecture adopts multi-clock cycle instructions. Moreover, in the CISC architecture, a complex operation can be treated by only one instruction.

The PE executable follows the Intel 64 and IA-32 Architecture for 64-bit version and 32-bit version, respectively. Both types of structures follow the CISC architecture. Thus, interpreting binary instructions in a PE executable is rather hard. There are some well-made tools for analysis binary instructions in the PE executable, which are called disassemblers, like IDA Pro, ODA, and PE Explorer. Furthermore, the Intel website has offered the document Intel 64 and IA-32 Architectures Software Developer's Manual **Error! Reference source not found.**. Thereby, we can do analysis binary instructions in the PE executable without any tool.

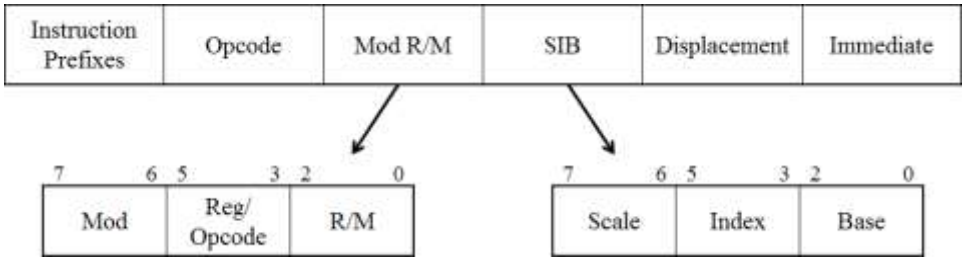


Figure 1. Intel 64 and IA-32 Architectures' Instruction Format

Figure 1 shows the Intel 64 and IA-32 architectures' instruction format which is in the Intel manual. Instruction in a PE executable consists of six parts like Figure 1. In this structure, every part is an optional value, except the second part. The opcode part is a primary part which specifies the specific behavior what the instruction does. Furthermore, its size can be up to 3-byte, while the first, the third, and the fourth parts' lengths are 1-byte, respectively. Notably, the Mod R/M and the SIB parts are used as addressing-form specifiers. The displacement value and the immediate value mean address displacement and immediate data respectively. Furthermore, their lengths can be zero, one, two, or four. Especially, some rare instructions need 8-byte as a displacement value or an immediate value.

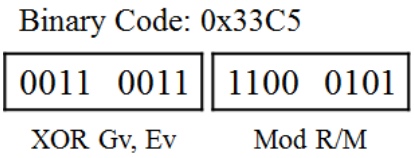


Figure 2. XOR EAX EBP

Here is a simple example. Figure **2**2 shows how to do the analysis the binary code 0x33C5 in the PE32 version. According to the Intel architecture manual, the first byte 0x33 means XOR Gv, Ev. The three alphabetic characters G, E, and v are abbreviations. In short, 0x33 means the operation XOR, which needs two registers that are decided by a Mod R/M field. In this example, the value of the Mod field is 11 in the binary number, and the Reg and the R/M fields are 000 and 101 respectively. According to the manual, in this case, the registers are EAX and EBP. The disassembly process of the PE32+ version is similar to the PE32 version.

3.3. CFG (Control Flow Graph)

The CFG is a way of expressing all the possible paths during the execution process of a program. CFG has a distinguishable characteristic that a CFG is covered from the entry block to the exit block. The CFG is extracted from the .text section which contains all the actual executable codes of a PE executable file. The CFG has some nodes which called basic blocks. Basic blocks represent a straight line sequence of codes without any jump or branching instruction. If two blocks have a relation, which is jump or branching, then a directed edge is on both nodes. Figure 3 illustrates the concept of directed edge between two nodes.

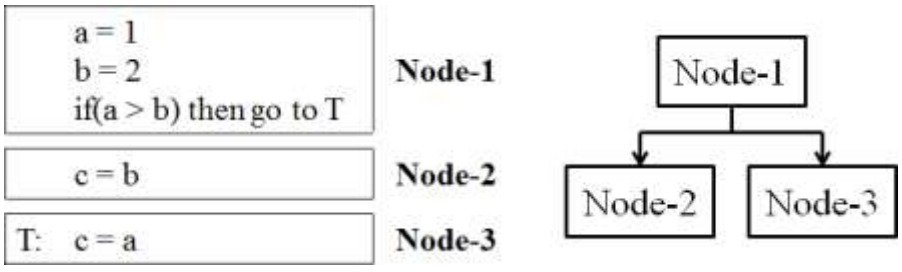


Figure 3. The Concept of Extended Basic Block

Aside from the basic block concept, there is another block concept that the extended basic block. An extended basic block is a maximal sequence of basic blocks. It has a characteristic that except the first basic block in the extended basic block, all the blocks should have only one predecessor basic block in the collection.

Through the extended basic block concept, the scope of the local analysis can be increased. In Figure 3, the node 1-3 can be expressed in an extended basic block.

3.4 Metafile Generator

In this section, we show our system architecture and the process step by step. The metafile generation process of our system is as Figure 4.

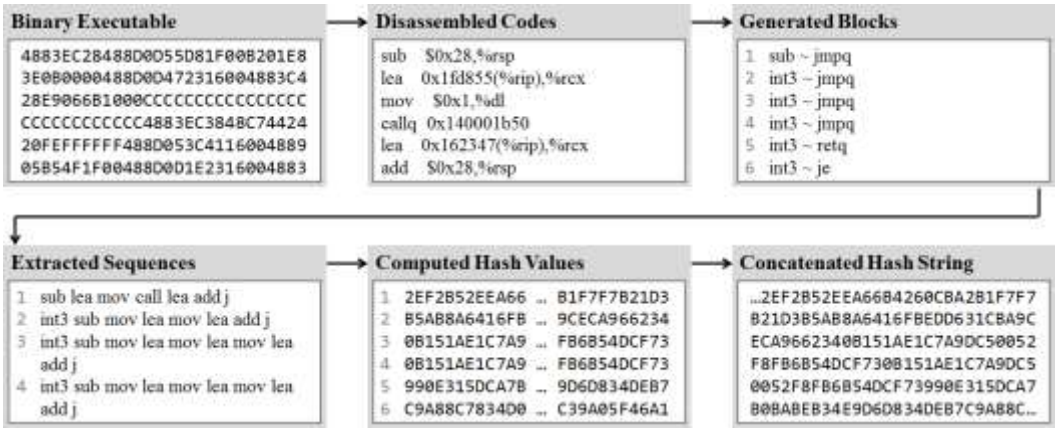


Figure 4. The Metafile Generation Process

To extract opcode sequences, we should disassemble the given software. In our system, we exploit the Linux command *objdump*. After disassembling, we split all the codes as a number of extended basic blocks. As we already said, an extended basic block is a maximal sequence of regular basic blocks. Thus, it can contain more codes. Thereby, using the concept of the extended basic block, then we can practically divide a bunch of codes as some meaningful blocks. After that, we extract opcode sequences of all the extended basic blocks. At this step, we exclude some blocks which contain just only one instruction. Besides, we handle a case that just a single opcode is repeatedly appeared without any disturbance, as putting the opcode into the corresponding sequence just one time. Through this step, we can organize the target software's all the sequential behaviors. Besides, each behavior can be a possible feature to compare to another software that suspected plagiarism. Next, we calculate each sequence's MD5 hash value and sort these hash values for the efficiency of the similarity computation. Furthermore, we concatenate these hash values each other without any duplication. Through this process, a hash string of all the sequences is successfully created. Figure 5 shows a sample metafile's contents.

Figure 5. A Sample Metafile

3.5. Similarity Calculation

Similarity calculator conducts the process of the similarity computation. This module derives a similarity result between two given metafiles. Among them, one is target file which is compared to the suspected plagiarism file, while the other one is suspected file

which is a dubious file as plagiarism. The generated hash values are already handled sorting. Thus, our similarity calculator just compares both metafiles' data by one-to-one comparison. Algorithm 1 shows the hash string comparison process.

Algorithm 1 The Hash String Comparison Process

```
Require: Target Software's Metafile and Suspected Software's Metafile
tHash = sHash = tEnd = sEnd = SUS = CMP = PER = 0
for Until reading suspect's hash string is over do
    sHash := Read suspect's a hash value
    SUS + +
    if There is no more unread hash value in suspect then
        sEnd := 1
    else
        sEnd := 0
    for Until reading target's hash string is over do
        tHash := Read target's a hash value
        if There is no more unread hash value in target then
            tEnd := 1
        else
            tEnd := 0
        if sHash == tHash then
            CMP + +
            break
        else if sHash > tHash then
            continue
        else if sHash < tHash then
            Move back target's file pointer to before reading a hash value
            break
        if (sEnd == 1) and (tEnd == 1) then
            if sHash == tHash then
                CMP + +
PER := CMP / SUS * 100
```

Figure 6. The Hash String Comparison Process

Through this way, we can measure the similarity that how many sequences are the same as that of the target file among all the sequences of a suspected file. Therefore, when we have a group of suspected files, then we can compute each file's similarity as well.

5. Experiment Result

We conducted experiments to prove the performance of our technique. For the experiment, we utilized MSDN developer code samples and some commercial software as our test files. Our two modules, metafile generator and similarity calculator, were implemented on Linux machine. Thereby, we performed experiments on a Linux machine that consists of Intel Xeon E5530 2.40GHz quad-core processors and 12 GB of RAM. Besides, the machine's OS is Ubuntu 13.10 with GNU/Linux 3.11.0-26-generic kernel. In the case of the test software, all the test programs are PE executables that work on the Windows OS environment. Before introducing experimental results, we will present some statistical values that both the average time required and the size. To measure the average time required to generate metafile, we conducted the process five times.

Table 2. Test Data Specification (file size and processing time)

Test Software	S/W Size (Byte)	Metafile Size (Byte)	Time Required to Generate Metafile (sec)
npp 7.4.2 x64	3,053,480	17,410	0.294775
py 3.6.2rc1 x64	31,425,536	139,906	3.299715
msdn sample 1	35,840	2,498	0.059360
msdn sample 2	313,856	2,562	0.845317

5.1. MSDN Code Sample Software

As we already said, we utilized MSDN developer code samples for this experiment. In this sub-section, we show two different types of MSDN code sample groups. Each group consists of three different versions of software that the original version and two variant versions. The original version is that the compiled software of the downloaded MSDN code sample without any modification. On the other hand, others are modified versions.

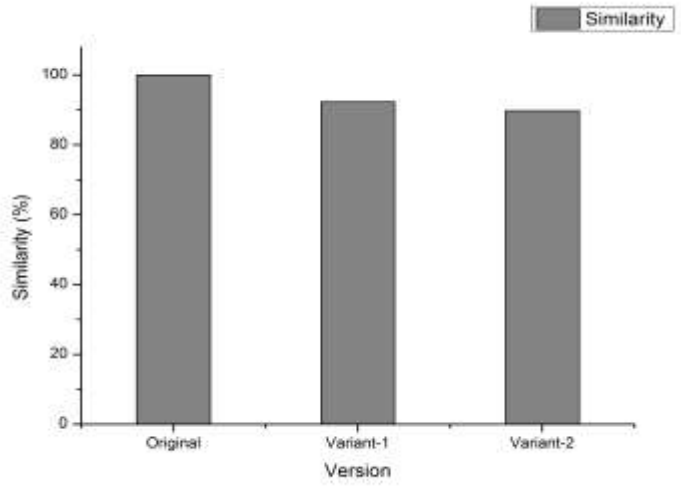


Figure 7. The Similarity Result of MSDN Code Sample-1

Figure show the similarity result related to the first MSDN code sample group. In Figure , we compared three different versions of software with the original version. At this experiment, the total number of hash values in the original version is 78. Among them, 72 are exactly the same as the variant-1 version. On the other hand, the number of matched hash values between the original version and the variant-2 version is 70.

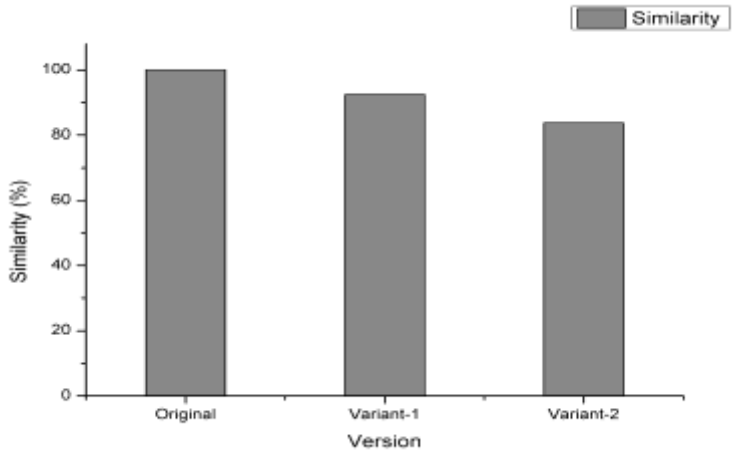


Figure 8. The Similarity Result of MSDN Code Sample-2

Error! Reference source not found. shows the similarity result related to the first MSDN code sample group. In **Error! Reference source not found.**, we compared three different versions of software with the original version. At this experiment, the total number of hash values in the original version is 80. Among them, 74 are exactly the same as the variant-1 version. On the other hand, the number of matched hash values between the original version and the variant-2 version is 67.

5.2. Commercial Software

In this sub-section, we show some experiment results related to two different kinds of commercial software. For that, we utilized Notepad++ install software series and some Windows OS default programs as our test files. Furthermore, we measured the similarity between the latest versions of two software series and the other versions of those two series.

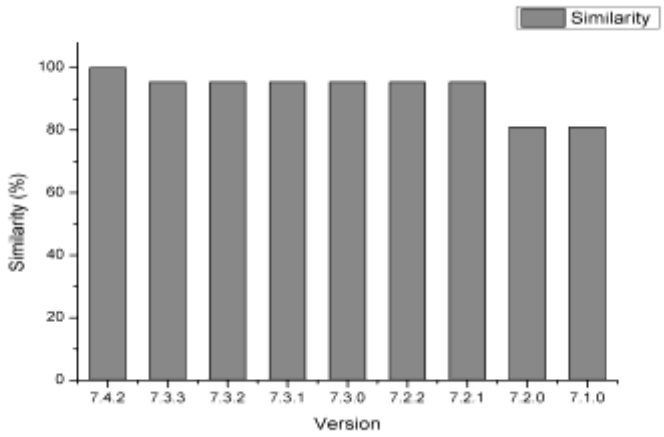


Figure 9. The Similarity Result of Commercial Software-1

Error! Reference source not found. shows the similarity result related to a series of Notepad++ installer software. In **Error! Reference source not found.**, we compared a series of Notepad++ installer software with the latest version of Notepad++ installer software that 7.4.2 x64 installer software. At this experiment, the total number of hash values in the Notepad++ installer 7.4.2 x64 version is 544. More specifically, from 7.3.3

version to 7.2.1 version, the number of matched hash values is 519, respectively. On the other hand, that of 7.2.0 version and 7.1.0 version is 441.

Table 3. The Similarity Result of Commercial Software-2

Version		Similarity (%)	The Number of Matched Hash Values
control.exe	Windows 10 x64 Eng.	100.00%	196/196
	Windows 10 x64 Kor.	78.57%	154/196
calc.exe	Windows 10 x64 Eng.	27.04%	53/196
	Windows 10 x64 Kor.	22.45%	44/196
cmd.exe	Windows 10 x64 Eng.	37.24%	73/196
	Windows 10 x64 Kor.	34.69%	68/196
notepad.exe	Windows 10 x64 Eng.	40.31%	79/196
	Windows 10 x64 Kor.	37.76%	74/196
write.exe	Windows 10 x64 Eng.	28.57%	56/196
	Windows 10 x64 Kor.	23.98%	47/196

Error! Reference source not found.3 shows the similarity result related to some Windows OS default programs. In **Error! Reference source not found.**3, we compared some default programs with control.exe in the Windows 10 x64 English version. At this experiment, the total number of hash values in control.exe in the Windows 10 x64 English version is 196. Surprisingly, the similarity between control.exe in the English version and that of Korean version is 78.57%, while that of other software is lower than that.

Table 4. The Similarity Result of Commercial Software-3

Version		Similarity (%)	The Number of Matched Hash Values
notepad.exe	Windows 10 x64 Eng.	100.00%	1400/1400
	Windows 10 x64 Kor.	71.93%	1007/1400
calc.exe	Windows 10 x64 Eng.	5.21%	73/1400
	Windows 10 x64 Kor.	3.50%	49/1400
cmd.exe	Windows 10 x64 Eng.	13.79%	193/1400
	Windows 10 x64 Kor.	18.21%	255/1400
control.exe	Windows 10 x64 Eng.	5.64%	79/1400
	Windows 10 x64 Kor.	5.36%	75/1400
write.exe	Windows 10 x64 Eng.	3.71%	52/1400
	Windows 10 x64 Kor.	3.36%	47/1400

Error! Reference source not found.4 shows a similar result. In **Error! Reference source not found.**4, we compared default programs with notepad.exe in the Windows 10 x64 English version. At this experiment, the total number of hash values in notepad.exe in the Windows 10 x64 English version is 1400. Furthermore, the similarity between notepad.exe in the English version and that of Korean version is 71.93%. On the other hand, that of other software is lesser than one-fifth.

6. Conclusion

In this paper, we show the software plagiarism detection technique. We have suggested a practical scheme by adapting executable fingerprint using opcode information. To accomplish accurate executable similarity analysis, we propose an executable fingerprint technique for binary executables in the Windows OS environment. The proposed scheme

exploits the Linux command *objdump*, the extended basic block concept and opcode sequences in the .text section of the corresponding binary executable. The key idea of the proposed scheme is to reflect the similarity of executable code by eliminating duplicated opcode information. With this approach, we can summarize the core opcode part in executable files. To show the usefulness of the proposed system, we implemented the executable fingerprint concept as a system on a Linux system. After that, we evaluated the system by using various sample executable files including commercial software. For future work, we will extend the executable fingerprint to contain runtime behaviors. With this challenge, we can achieve more practical way for detecting software plagiarism.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and future Planning(2014R1A2A1A11054160). This research was supported by The Leading Human Resource Training Program of Regional Neo industry through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and future Planning(2016H1D5A1910630)

References

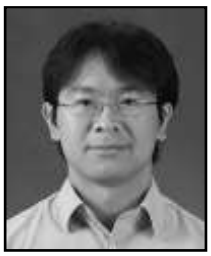
[1] D. Quinlan and T. Panas, "Source code and binary analysis of software defects", Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies. ACM, **(2009)**.
[2] T. Yamamoto, "Measuring similarity of large software systems based on source code correspondence", International Conference on Product Focused Software Process Improvement, Springer Berlin Heidelberg, **(2005)**.
[3] A. Sæbjørnsen, "Detecting code clones in binary executables", Proceedings of the eighteenth international symposium on Software testing and analysis. ACM, **(2009)**.
[4] Z. Li, "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code", OSdi., vol. 4, no. 19, **(2004)**.
[5] T. Kamiya, S. Kusumoto and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code", IEEE Transactions on Software Engineering, vol. 28, no. 7, **(2002)**, pp. 654-670.
[6] G. Bonfante, M. Kaczmarek and J.-Y. Marion, "Morphological detection of malware", Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on IEEE, **(2008)**.
[7] A. Walenstein, "Exploiting similarity between variants to defeat malware", Proc. BlackHat DC Conf., **(2007)**.
[8] Microsofte PE and COFF Specification, https://msdn.microsoft.com.
[9] Intel 64 and IA-32 Architecture Software Developer Manuals, http://www.intel.com.
[10] L. Jiang, "Deckard: Scalable and accurate tree-based detection of code clones", Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, **(2007)**.
[11] R. Koschke, R. Falke and P. Frenzel, "Clone detection using abstract syntax suffix trees", Reverse Engineering, 2006. WCRE'06. 13th Working Conference on IEEE, **(2006)**.
[12] K. Rieck, "Automatic analysis of malware behavior using machine learning", Journal of Computer Security, vol. 19, no. 4, **(2011)**, pp. 639-668.
[13] J. Sahs and L. Khan, "A machine learning approach to android malware detection", Intelligence and security informatics conference (eisic), 2012 european, IEEE, **(2012)**.
[14] Z. Markel and M. Bilzor, "Building a machine learning classifier for malware detection", Anti-malware Testing Research (WATeR), 2014 Second Workshop on. IEEE, **(2014)**.
[15] I. Burguera, U. Zurutuza and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android", Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, ACM, **(2011)**.
[16] T. Isohara, K. Takemori and A. Kubota, "Kernel-based behavior analysis for android malware detection", Computational Intelligence and Security (CIS), 2011 Seventh International Conference on. IEEE, **(2011)**.
[17] A. Shabtai, "Andromaly: a behavioral malware detection framework for android devices", Journal of Intelligent Information Systems, vol. 38, no. 1, **(2012)**, pp. 161-190.
[18] D.-J. Wu, "Droidmat: Android malware detection through manifest and api calls tracing", Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on. IEEE, **(2012)**.

[19] R. Koschke and S. Bazrafshan, "Software-Clone Rates in Open-Source Programs Written in C or C++", Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on IEEE, vol. 3, **(2016)**.
[20] D. Mazinanian, "JDeodorant: clone refactoring", Proceedings of the 38th International Conference on Software Engineering Companion. ACM, **(2016)**.
[21] S. Strandlund Hansen, "An approach for detection and family classification of malware based on behavioral analysis", Computing, Networking and Communications (ICNC), 2016 International Conference on IEEE, **(2016)**.
[22] G. Liang, J. Pang and C. Dai, "A Behavior-Based Malware Variant Classification Technique", International Journal of Information and Education Technology, vol. 6, no. 4, **(2016)**.

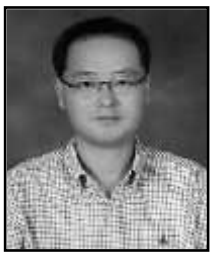
Authors

Su-Jin Oh received both a bachelor and a master degree in computer science from Hallym University, Chuncheon, Korea, in 2015 and 2017, respectively. She is currently working at Security Company. Her research interests include operating system, security file system and data deduplication.

Jin Kim received an MS degree in computer science from the College of Engineering at Michigan State University in 1990, and in 1996 a PhD degree from Michigan State University, USA. Since then he has been working as a Professor in computer engineering and as a researcher at the Hallym University. His research includes bioinformatics and data mining.



Jeong Gun Lee received the B.S. degree in computer engineering from Hallym University in 1996, and M.S. and Ph.D. degree from Gwangju Institute of Science and Technology (GIST), Korea, in 1998 and 2005. He is currently an assistant professor in the Computer Engineering department at Hallym University.



Sung-Bong Jang received his a M.S. and Ph.D. degrees from Korea University, Seoul, Korea in 1999 and 2010, respectively. He worked at the Mobile Handset R&D Center, LG Electronics from 1999 to 2012. Currently, he is an associate professor in the Department of Industry-Academy Cooperation, Kumoh National Institute of Technology in Korea. His interests include BigData *k*-anonymization, Mobile Video Communication, and Privacy Protection.

Young-Woong Ko received both a M.S. and Ph.D. in computer science from Korea University, Seoul, Korea, in 1999 and 2003, respectively. He is now a professor in Department of Computer engineering, Hallym University in Korea. His research interests include operating systems, embedded systems and multimedia systems.

