International Journal of Security and Its Applications
Vol.9, No.8 (2015), pp.127-136
http://dx.doi.org/10.14257/ijsia.2015.9.8.10

Design and Development of a Command-line Tool for Portable Executable File Analysis and Malware Detection in IoT Devices

June Ho Yang and Yeonseung Ryu

Dept. of Security Administration Engineering, Myongji University
116 Myongji-Ro, Cheoin-Gu, Yongin, Gyeonggi-Do, 449-728, KOREA
mjuboan2005@daum.net, ysryu@mju.ac.kr

Abstract

Recently, Microsoft unveiled a new operating system called Windows 10. As it is highly expected that Windows 10 will play a significant role in the Internet of Things era, the Portable Executable (PE) format is drawing attention even more widely than before. PE is a standard file format for executables and object code used in MS Windows operating systems. Since a number of various malwares have widely spread by exploiting vulnerabilities of PE structure, the need of automatic tools for PE-malware detection is being magnified. In this paper, we designed and developed a command-line PE file analysis tool using Python language for automatic detection of Windows malware.

Keywords: *Portable Executable File, Malware, Static Analysis, Detection Tool*

1. Introduction

The Internet of Things (IoT) is the network of things embedded with electronics, sensors, and connectivity to enable objects to exchange data [1]. Things can be a variety of devices such as heart monitoring implants, biochip transponders on farm animals, automobiles, consumer electronics, and so on. These devices collect data from their environment and then autonomously flow the data between other devices. As the Internet of Things spread widely, cyberattacks are likely to become a physical threat [2]. For example, internet-connected consumer electronics, such as television, kitchen appliances and thermostats, can spy on people in their homes. Furthermore, computer-controlled devices in automobile, such as brakes, heat and dashboards, can be accessed by the attackers who operate automobile in abnormal ways. The malware prevention or detection function, therefore, is significantly important in a variety of operating system platforms for IoT.

Recently, Microsoft released a new operating system called Windows 10 [3]. Windows 10 is no longer just an operating system for 32 and 64-bit PCs. It will also run on the ARM platform for smaller tablets and smartphones. Windows 10 introduces a "*universal*" application architecture; applications can be designed to run across multiple Microsoft product families with nearly identical code—including PCs, tablets, smartphones, and Xbox One, as well as new products such as Surface Hub and HoloLens. It is highly expected that Windows 10 will play a significant role in the IoT era.

In MS Windows operating systems, the Portable Executable (PE) format is a standard file format for executables and object code [4]. PE currently supports the IA-32, IA-64, x86-64 (AMD64/Intel64), and ARM instruction set architectures. Prior to Windows 2000, PE supported the MIPS, Alpha, and PowerPC ISAs. Because PE is used on Windows CE, it continues to support several variants of the MIPS, ARM, and SuperH. Analogous formats to PE are ELF (used in Linux and Unix) [5] and Mach-O (used in Mac OS X) [6].

There have been a number of techniques for malware detection or prevention for PE files. There are two types of malware detection techniques according to how code is

ISSN: 1738-9976 IJSIA

analyzed [7-26]: First, static analysis identifies malicious code by unpacking and disassembling (or decompiling) the application [7-9]. The most important defense methods against malicious code are virus scanners. These scanners typically rely on a database of signatures that characterize known malware instances. Second, dynamic analysis identifies malicious behaviors after deploying and executing the application on an emulator or a controlled device [10-12]. The dynamic detection systems automatically load the sample to be analyzed into a virtual machine environment and execute it. While the program is running, its interaction with the operating system is recorded.

In this paper, we developed a command-line tool of PE file as a first step to automated malware detection tool. The proposed tool scans major data structures of PE, finds abnormalities using the rule database, and prints results in a variety format according to user options. We expect that our tool can be efficiently integrated with automated malware detection tools. The rest of this paper is organized as follows. In Section 2, we describe structure of PE file format briefly. Section 3 describes the details of our analysis tool. Finally, section 4 describes conclusion and future work.

2. Portable Executable File Format

The Portable Executable (PE) format is a file format for executables, object code, DLLs, FON Font file and others used in 32-bit and 64-bit versions of Windows operating systems [4]. The PE format encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code. This includes dynamic library references for linking, API export and import tables, resource management data and thread-local storage data. On NT operating systems, the PE format is used for EXE, DLL, SYS (device driver), and other file types. The Extensible Firmware Interface (EFI) specification states that PE is the standard executable format in EFI environments [27].

In fact, PE is a modified version of the Unix COFF (Common Object File Format). PE/COFF is an alternative term in Windows development. Microsoft migrated to the PE format with the introduction of the Windows NT 3.1. All later versions of Windows, including Windows 95/98/ME, support the PE file structure. The format has retained limited legacy support to bridge the gap between DOS-based and NT systems. For example, PE/COFF headers still include an MS-DOS executable program, which is by default a stub that displays a message like "This program cannot be run in DOS mode", though it can be a full-fledged DOS version of the program. PE also continues to serve the changing Windows platform. Some extensions include the .NET PE format, a 64-bit version called PE32+ (i.e., PE+), and a specification for Windows CE.

2.1. PE File Structure

A PE file consists of a number of headers and sections that tell the dynamic linker how to map the file into memory (see Figure 1). An executable image consists of several different regions, each of which requires different memory protection; so the start of each section must be aligned to a page boundary. For instance, typically the *.text* section (which holds program code) is mapped as execute/readonly, and the *.data* section (holding global variables) is mapped as no-execute/readwrite. However, to avoid wasting space, the different sections are not page aligned on disk. The dynamic linker maps each section to memory individually and

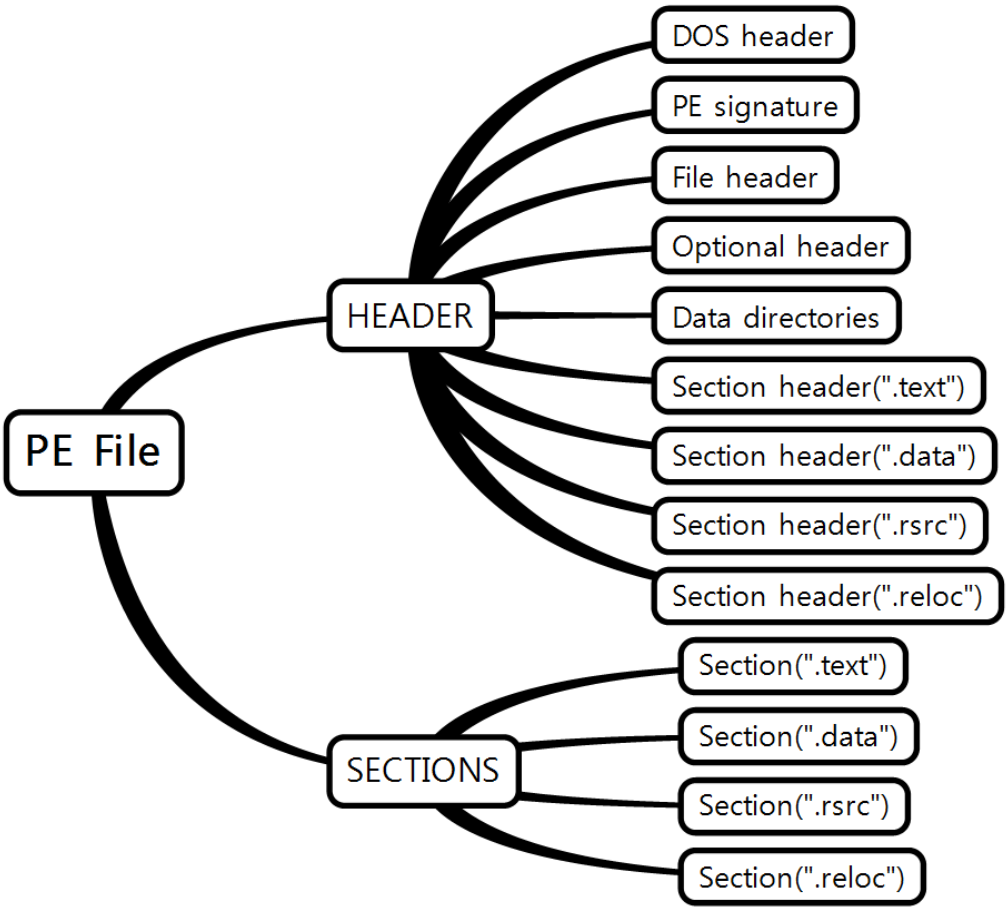


Figure 1. Overall Structure of a PE File

assigns the correct permissions to the resulting regions, according to the instructions found in the headers.

2.2. Header

In a PE file, there are some headers. The PE header consists of an MS-DOS header, the PE signature, the file header, an optional header, and data directories. The file headers are followed immediately by section table.

The file header contains machine type, the number of sections, the time stamp, the size of optional header, and so on. The optional header is optional in the sense that some files (i.e., object file) do not have it. For image files, this header is required to provide information to the loader. The magic number determines whether an image is a PE32 or PE32+ executable. The optional header contains the size of code section, the size of initialized data section, the size of uninitialized section, the base of code, the address of the entry point, and so on. The data directory gives the address and size of a table or string that Windows uses (see Table 1). These data directory entries are all loaded into memory so that the system can use them at run time.

Table 1. Contents of Image Data Directory

Field	Explanation
Export Table	Address and size of the export table. The export table contains the .edata section.

Import Table	Address and size of the import table. Import table is included in the .idata section.
Resource Table	Address and the size of the resource table. A resource table is contained in .rsrc section.
Exception Table	Address and the size of the exception table. .pdata Exception table is included in the section.
Certificate Table	Address and size of the Attribute Certificate Table.
Base Relocation Table	Address and size of the Base Relocation Table. Base Relocation Table is contained in .reloc section.
Debug	Starting address and size of the Debug data. Debug data is contained in the section .debug.
Architecture	Reserved area. Filled with a zero.
Global Ptr	RVA of the values stored in the global pointer register. In I386 series it does not use IA-64 is used in.
TLS Table	Address and size of the Thread Local Storage table. TLS is included in the section .tls.
Loading Config Table	Address and size of the Loading Config Table.
Bound Import	Address and size of the Bound Import.
Import Address Table	Address and size of the Import Address Table. Do not be mistaken with the Import Table.
Delay Import Descriptor	Address and size of the Delay Import Descriptor.
CLR Runtime Header	Address and size of the CLR Runtime Header. This information is contained in .cormeta section.

The MS-DOS header is not new for the PE file format. It is the same MS-DOS header that has been around since version 2 of the MS-DOS OS. The main reason for keeping the same structure intact at the beginning of the PE file is so that, when attempting to load a file created under Windows version 3.1 or earlier, the operating system can read the file and understand that it is not compatible. In other words, when attempting to run a Windows NT executable on MS-DOS version 6.0, get this message: "This program cannot be run in DOS mode." PE file header is located by indexing the *e_lfanew* filed of the MS-DOS header. The next follows other headers.

In section table, each row is a section header. The number of entries in the section table is given by the *NumberOfSections* field in the file header. Each section header (section table entry) has a total of 40 bytes and contains section name, the size of the section when loaded into memory, and so on. Table 2 illustrates the name and meaning of possible sections in a PE file.

2.3. Section

Typical sections contain code or data that linkers and loaders process without special knowledge of the section contents. A section consists of simple blocks of bytes. However, for sections that contain all zeros, the section data need not be included. The data for each section is located at the file offset that was given by the *PointerToRawData* field in the section header. The size of this data in the file is indicated by the SizeOfRawData field. If *SizeOfRawData* is less than *VirtualSize*, the remainder is padded with zeros. In an image file, the section data must be aligned on a boundary as specified by the *FileAlignment* field in the optional header.

Table 2. Sections in a PE File

Name	Explanation	Attribute
.text	Execution code, and the example, the format is not limited.	IMAGE_SCN_CNT_CODE IMAGE_SCN_MEM_EXECUTE IMAGE_SCN_MEM_READ
.data	contains various variables are initialized. There are no format restrictions.	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE
.edata	contains the Export Table.	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ
.idata	contains the Import Table.	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE
.rdata	contains read-only values among initialized variables.	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ
.rsrc	File version information and icons, which contains other data, such as dialogue.	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ

Some sections have special meanings when found in object files or image files. Tools and loaders recognize these sections because they have special flags set in the section header, because special locations in the image optional header point to them, or because the section name itself indicates a special function of the section. For example, if a section name is ".bss", the section is an uninitialized data section.

3. Related Works

There have been some traditional signature-based malware detection methods for PE file [21-24]. In order to improve the signature-based detection, a few attempts to apply data mining and machine learning techniques, such as Naive Bayes method, support vector machine (SVM) and Decision Tree classifiers, to detect new malicious executables [25-26]. In [25], authors gathered 1,971 benign executables and 1,651 malicious executables in Windows PE format, and examined the performance of different classifiers such as Naïve Bayes, support vector machine (SVM) and Decision Tree using tenfold cross validation and plotting Receiver Operating Characteristics (ROC) curves. [26] Applied Naive Bayes method to detect previously unknown malicious code. The authors downloaded 1,001 benign executables and 3,265 malicious executables from several FTP sites and labeled them by a commercial virus scanner. Furthermore, they examined a small data set of 38 malicious programs and 206 benign programs in Windows PE format. Although these works improve the traditional signature-based detection, they still fail against the unknown malware.

4. Implementation Results

The proposed tool is composed of PE parser, malware analysis rule generator & DB, and analyzer (see Figure 2). The PE parser decomposes a PE file into components such as headers and sections. The malware rule database has a set of rules which represent malware signatures. The analyzer compares the components of the parsed PE file with the malware rule DB to decide whether the PE file is malware or not.

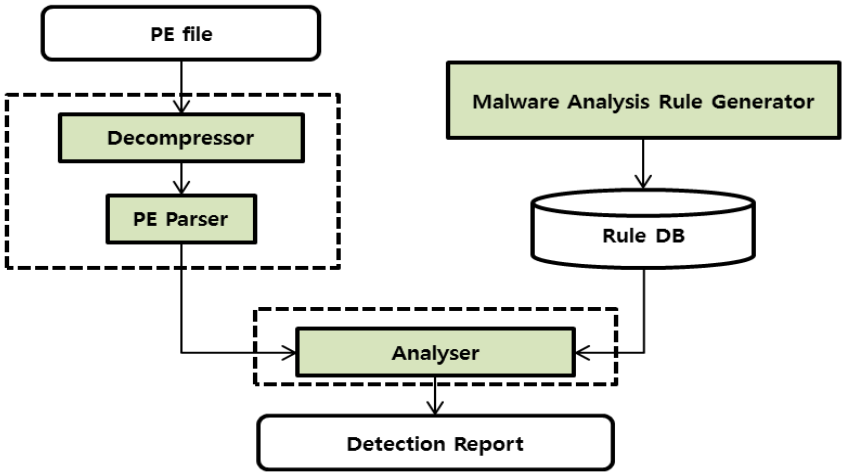


Figure 2. Architecture of Proposed Tool

The tool usage is shown in Figure 3. Our tool is developed using Python language and executed in the command-line environment. Several options are provided to extract necessary information from PE file. For example, option '*hdrs*' can print DOS header and PE headers and check if there is abnormality in the headers.

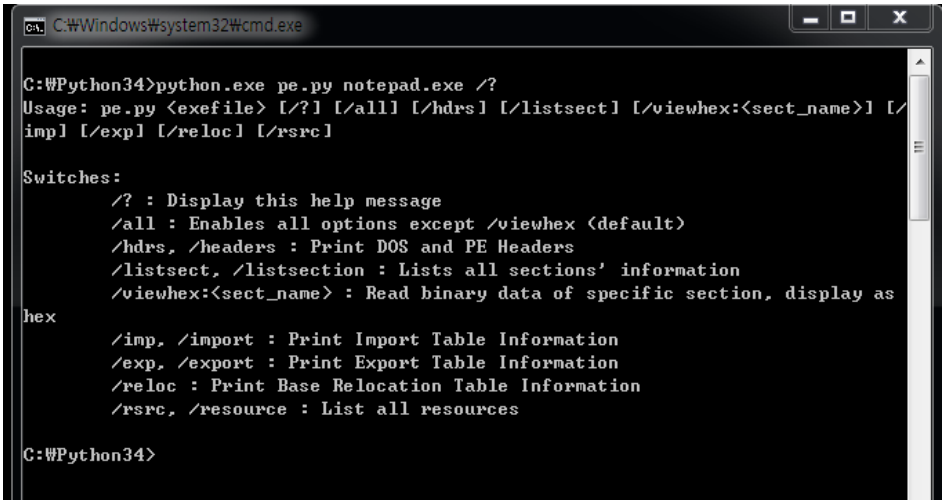


Figure 3. Our Tool Usage

Figure 4 and 5 show examples of PE headers and sections printed by our tool, respectively. In this case, there is no abnormality in PE.

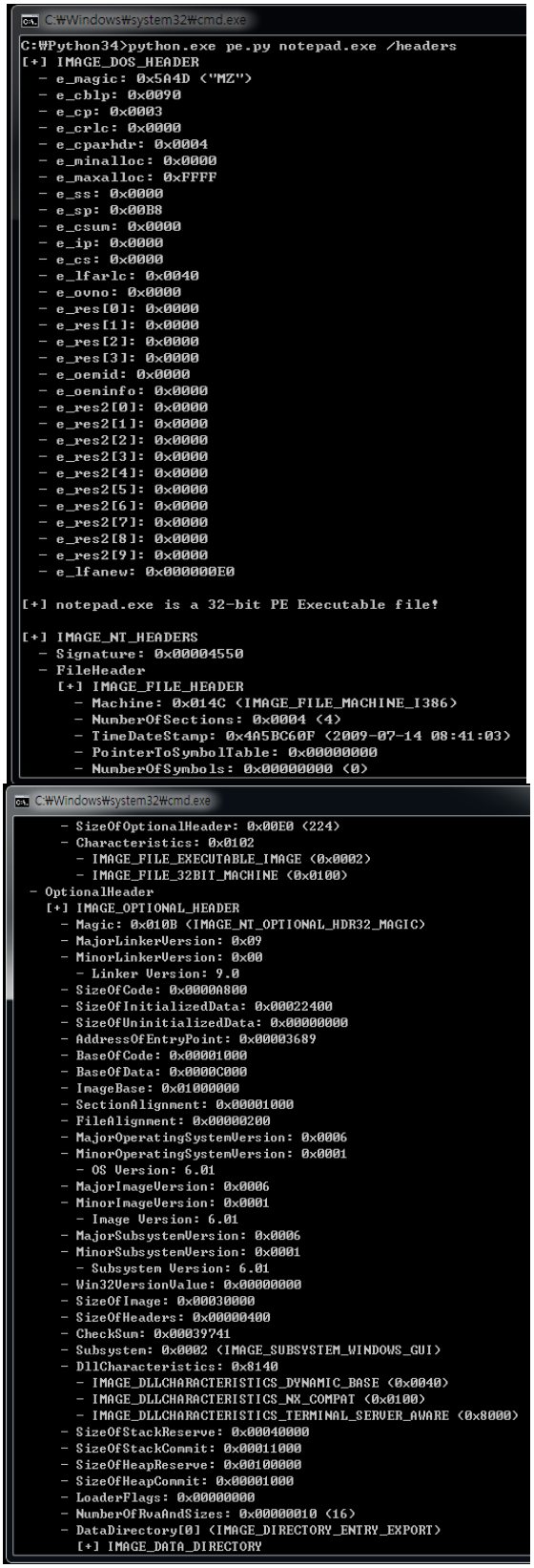


Figure 4. Example of Headers

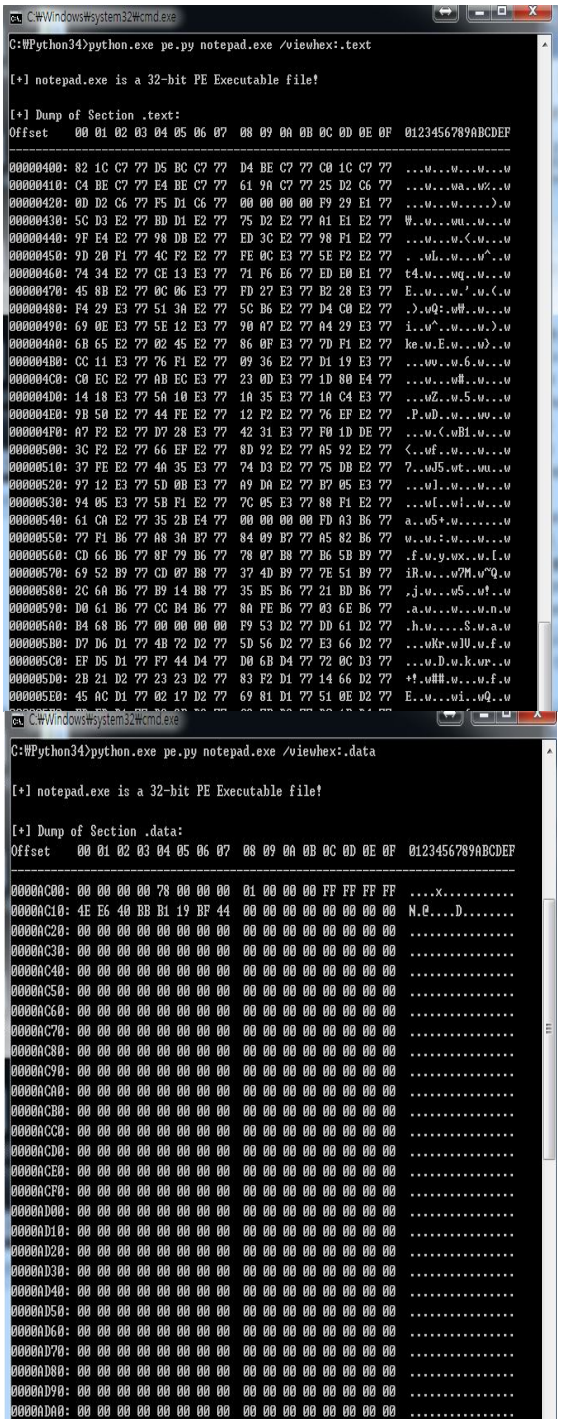


Figure 5. Example of Sections

5. Conclusion

Since Microsoft will expand its Windows operating system business into IoT industry aggressively, the malware detection and prevention for PE files become more important than before. A number of the PE-malwares variations will emerge as we have witnessed over the past several decades and the number of them will increase as fast as the number of IoT devices increases. In order to manage PE-malwares efficiently, it is required that a powerful PE analysis tool should be developed. In this work, we developed a command-line tool for PE analysis using

Python. The newly developed tool can scan PE files rapidly and detect the anomaly in them. For the next work, we will continue studying efficient PE-malware detection techniques and develop automatic tools for preventing the PE-malwares in IoT products.

Acknowledgements

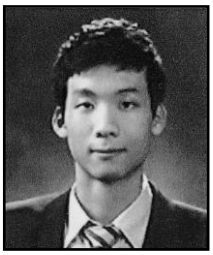
This work was supported by 2015 Research Fund of Myongji University. We would like to thank Mr. Jaemyung Lee for his kind help.

References

[1] L. Atzonri, A. Iera and G. Morabito, "The Internet of Things: A survey, Computer Networks", vol. 54, no. 15, **(2010).**
[2] R. Weber, "Internet of Things – New Security and Privacy Challenges", Computer Law & Security Review, vol. 26, no. 1, **(2010).**
[3] N. Ralph, "Microsoft Windows 10 Review", www.cnet.com, **(2015).**
[4] Microsoft Portable Executable and Common Object File Format Specification. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx **(2013).**
[5] Wikipedia, Executable and Linkable Format (ELF), https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
[6] Wikipedia, Mach-O, https://en.wikipedia.org/wiki/Mach-O
[7] A. Moser, C. Kruegel and E. Kirda, "Limits of Static Analysis for Malware Detection", Proceedings of International Conference on Computer Security Applications, **(2007),** pp. 421-430.
[8] S. Tabish, M. Shafiq and M. Farooq, "Malware Detection using Statistical Analysis of Byte-level File Content", Proceedings of ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics, **(2009),** pp. 23-31.
[9] Y. Feng, S. Anand, I. Dillig and A. Aiken, "Apposcopy: Semantic-based Detection of Android Malware through Static Analysis", Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering, **(2014),** pp. 576-587.
[10] C. Willems, T. Holz and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox", Proceedings of IEEE Symposium on Security & Privacy, **(2007),** pp. 32-39.
[11] B. Anderson, D. Quist, J. Neil, C. Storlie and T. Lane, "Graph-based Malware Detection using Dynamic Analysis. Journal in Computer Virology", vol. 7, no. 4, **(2011),** pp. 247-258.
[12] M. Egele, T. Scholte, E. Kirda and C. Kruegel, "A Survey on Automated Dynamic Malware-Analysis Techniques and Tools", ACM Computing Surveys, vol. 44, no. 2, **(2012).**
[13] Y. Ye, D. Wang, T. Li, D. Ye and Q. Jiang, "An Intelligent PE-malware Detection System based on Association Mining", Journal in Computer Virology, vol. 4, no. 4, **(2008),** pp. 323-334.
[14] Y. Choi, I. Kim, J. Oh, and J. Ryu, "PE File Header Analysis-based Packed PE File Detection Technique (PHAD)", Proceedings of International Symposium on Computer Science and Its Applications, **(2008).**
[15] A. Zaidan, B. Zaidan and F. Othman, "New Technique of Hidden Data in PE-File with in Unused Area One", International Journal of Computer and Electrical Engineering, vol. 1, no. 5, **(2009).**
[16] R. Merkel, T. Hoppe, C. Kraetzer and J. Dittmann, "Statistical Detection of Malicious PE-Executables for Fast Offline Analysis", Lecture Notes in Computer Science, vol. 6109, **(2010)**, pp. 93-105.
[17] J. Yang and Y. Ryu, "Toward an Efficient PE-Malware Detection Tool", Advanced Science and Technology Letters. 109, Proceedings of International Workshop on Security, Reliability and Safety, **(2015)**, August 19-22; Jeju, Korea.
[18] M. Christodorescu, S. Jha, S.A. Seshia, D. Song and R. E. Bryant, "Semantics-Aware Malware Detection", Proceedings of IEEE International Symposium on Security and Privacy, **(2005).**
[19] M. D. Preda, M. Chritodorescu, and S. Jha, "A Semantics-Based Approach to Malware Detection", Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, **(2007)**, January 17-19; Nice, France.
[20] A. Moser and C. Kruegel and E. Kirda, "Limits of Static Analysis for Malware detection", Proceedings of the 23rd Annual Computer Security Application Conference, **(2007).**
[21] J. Kephart and W. Arnold, "Automatic Extraction of Computer Virus Signatures", Proceedings of the 4th International Conference on Virus Bulletin, **(1994).**
[22] E. Filiol, "Malware Pattern Scanning Schemes Secure Against Blackbox Analysis", Journal in Computer Virology, vol. 2, no. 1, **(2006).**
[23] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns", Proceedings of the 12th USENIX Security Symposium, **(2003).**

[24] A. Sung, J. Xu, P. Chavez and S. Mukkamala, "Static Analyzer of Vicious Executables (SAVE)", Proceedings of the 20th Annual Computer Security Applications Conference, **(2004)**, December 6-10; Tucson, USA.
[25] J. Kolter and M. Maloof, "Learning to Detect Malicious Executables in the Wild", Proceedings of ACM conference on Knowledge Discovery and Data Mining, **(2004).**
[26] M. Schultz, E. Eskin and E. Zadok, "Data Mining Methods for Detection of New Malicious Executables", Proceedings of IEEE symposium on Security and Privacy, **(2001)**, May 14-16; Oakland, USA.
[27] UEFI Specification, Version 2.5, www.uefi.org

Authors



June Ho Yang, he received his BS degree in Computer Engineering from Myongji University, Korea, in 2015, and he is currently studying a master's degree in Security Administration Engineering from Myongji University which started this year (2015). His research interests include operating systems and convergence security.

Yeonseung Ryu, he received his BS degree in Computer Science and Statistics from Seoul National University, Korea, in 1990, and his MS and PhD degrees in Computer Science from Seoul National University in 1992 and 1996, respectively. In 1996 he joined Samsung Electronics, Co. as a senior researcher. Since 2003, he has been with Myongji University, Korea, where he is currently a full time professor in the Computer Engineering Department and Security Administration Engineering Department. From March 2009 to February 2010, he was a visiting scholar at the University of Minnesota, USA. His research interests include operating systems and convergence security.

