A Method for Shellcode Extractionfrom Malicious Document Files Using Entropy and Emulation

Kazuki Iwamoto and Katsumi Wasaki

Abstract—**We propose a method for the dynamic analysis of malicious documents that can exploit various types of vulnerability in applications. Static analysis of a document can be used to identify the type of vulnerability involved. However, it can be difficult to identify unknown vulnerabilities, and the application may not be available even if we could identify the vulnerability. In fact, malicious code that is executed after the exploitation may not have a relationship with the type of vulnerability in many cases. In this paper, we propose a method that extracts and executes "shellcode" to analyze malicious documents without requiring identification of the vulnerability or the application. Our system extracts shellcode by executing byte sequences to observe the features of a document file in a priority order decided on the basis of entropy.Our system was used to analyze 88 malware samples and was able to extract shellcode from 74 samples. of these, 51 extracted shellcodes behaved as malicious software according to dynamic analysis.**

Index Terms—**Malware, shellcode, entropy, dynamic analysis, vulnerability.**

I. Introduction

At the start of a typical attack aimed at stealing information from a targeted organization, a piece of malware is supplied by targeted email [1]. An email is sent to a specific person, and often has an attached document file containing malicious code. The victim may then open the document file and execute the malicious code without knowing that it was created for attacking purposes. As a part of measures to deal with targeted attacks, we would like to analyze the behavior of malicious code via dynamic analysis.

It is often not possible to use dynamic analysis directly, because we cannot reproduce an appropriate vulnerable environment. The reason for this is that the vulnerability usually depends on the environment of the operating system (OS) or the application software. However, versatile malicious code (or "shellcode"), which often does not depend on a specific OS or application software, can be executed. Therefore, our system extracts shellcode from the document file to analyze the malicious document. It then outputs an executable file containing shellcode to enable dynamic analysis.

Before building our system, we conducted a preliminary survey of malware samples that we had already analyzed. In this preliminary survey, we determined parameter values for calculating the entropy, an algorithm for shellcode priority and byte sequences to be excluded from the document file. Our system executes those byte sequences that are shellcode candidates and observes their behavior to detect features. We determined the appropriate number of instructions to be executed by the emulator to enable feature detection (the number of steps) in the preliminary survey.

The remainder of this paper is organized as follows. Section II explains the proposed method by describing the environment, the file format, the features of shellcode and the definition of entropy. Section III describes the sample set used to identify the parameter values and the algorithm that are the most appropriate for our system. Section IV explains the results obtained by using our system with the sample set and evaluates the performance. Section V discusses the results. Section VI briefly reviews related work and compares it with our system. Section VII describes our plans for future work.

Manuscript received September 24, 2014; revised November 20, 2014.

Kazuki Iwamoto is with Advanced Research Laboratory at SecureBrain Corporation, Kojimachi RK Building 4F 2-6-7 Kojimachi, Chiyoda-ku Tokyo, Japan (e-mail: kazuki_iwamoto@securebrain.co.jp).

Katsumi Wasaki is with Interdisciplinary Graduate School of Science and Technology, Shinshu University, 4-17-1 Wakazato, Nagano-shi, Japan (e-mail: wasaki@cs.shinshu-u.ac.jp).

II. Environment and Proposed Method

Our system can execute byte sequences that are shellcode candidates by using an emulator, extracting a byte sequence as an actual shellcode if it observes shellcode-like features. First, though, our system produces a list of candidates in order of their shellcode likelihood to promote the efficient extraction of actual shellcodes. Fig.1 shows the overall flow of our system.

A. Target Environment

Our system focuses on the following types of malicious documents for the 32-bit Windows environment.

1) Microsoft Office Word (.doc)
2) Microsoft Office Excel (.xls)
3) Microsoft Office PowerPoint (.ppt)
4) Rich Text Format (RTF) (.rtf)

Our system does not need application software to open the document files. Therefore, our system cannot deal with the following types of malicious documents, which depend on the application software.

- Return-oriented Programming [2]
- Here, instructions in the OS or application software are executed based on values that are pushed on a stack. Therefore, some of these documents will not include shellcode. Even if the document does include shellcode, our system cannot deal with it whenever there is a premise that the shellcode must be loaded to a specific address.
- Malicious Document without Shellcode
- If code that is not included in the document file is executed

DOI: 10.7763/IJET.2016.V8.866

after the vulnerability is exploited, our system cannot deal with it. For example, in CVE-2011-1980 [3], the Dynamic Link Library (DLL) in the same folder as the malicious document is executed.

- Malicious Document Strongly Dependent on the Environment

Our system cannot deal with shellcode that is not versatile or that depends on specific conditions such as an allocated memory or address.

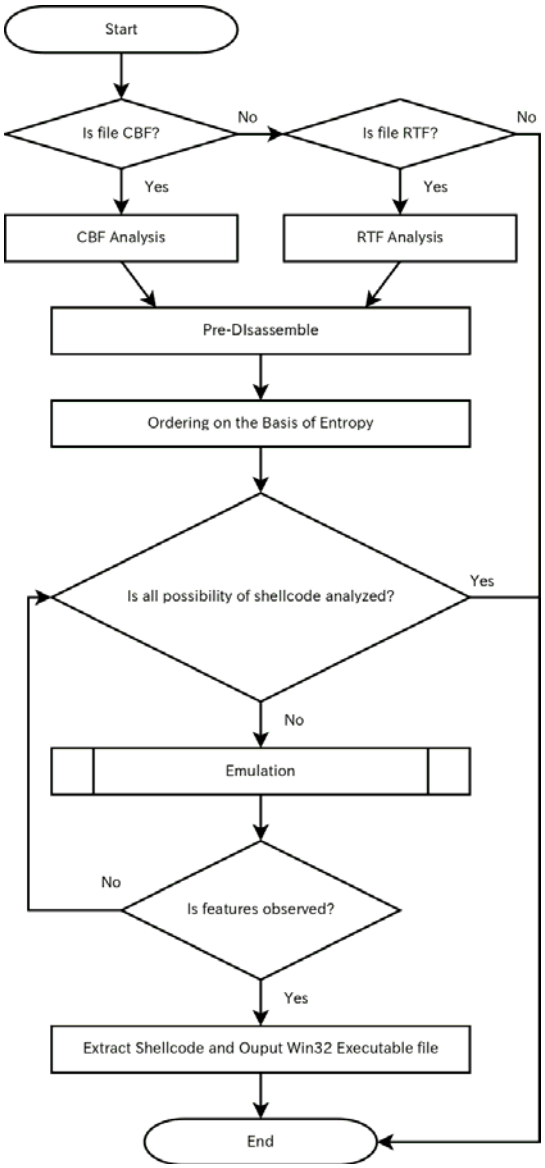

Fig. 1. Flowchart.

B. Narrowing Down the Candidate List and Priority

Our system narrows down the list of shellcode candidates and determines their priority. Narrowing down is possible by using Compound File Binary (CFB) analysis or RTF analysis, because our system focuses on document files. The proposal ofPolychronakis et al. [4] for extracting shellcode does not have a narrowing-down method, because it uses network packet extraction. The proposal of Li et al. [5] analyzes file structures, but not for the purpose of extracting shellcode.

1) CFB analysis

The file formats identified in Section II.A are all CFB formats [6], [7], except for RTF. CFB formats have a similar file structure, such as that shown in Fig. 2. Elements in CFB are categorized as follows: Header, DiFAT, FAT, Mini FAT, Directory, Stream, Mini Stream and Free. Header is the information area at the start of the file. DiFAT, FAT and Mini FAT correspond to the File Allocation Table (FAT) in the file system. Directory corresponds to directory data. Finally, Stream and Mini Stream correspond to file data. Free refers to unused areas. Our system analyzes the CFB, categorizes its elements and specifies whether they are shellcode candidates.

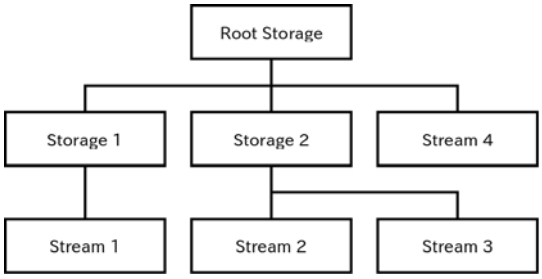

Fig. 2. CFB Hierarchy.

2) RTF analysis

RTF is a type of text format, with binary data (including strings) encoded within the text [8]. Because shellcode is a form of binary data, our system focuses on this binary data. It is not possible to have executable shellcode in a small section of binary data, because the features described in Section II.C must be included. We estimate that the minimum size of code that can include these features is about 128 bytes. Therefore, our system ignores binary data sequences smaller than 128 bytes.

3) Pre-disassembly

Our system disassembles targeted byte sequences before performing the emulation. If our system cannot disassemble successfully, the byte sequence is not emulated, and our system concludes that the byte sequence is not shellcode. The purpose of disassembly is to reduce the number of booted emulations, because the booting process requires substantial CPU resources. Our system uses the results of disassembly only for the narrowing-down process. This is different from the method of Polychronakis *et al*. [4], which uses disassembly to extract shellcode.

4) Priority by entropy

If our system tried to extract shellcode from the file header, it would be inefficient, because byte sequences that could never be shellcode would be emulated. Therefore, our system creates an order of priority on the basis of entropy. This is different from the method of Polychronakis *et al*. [4].

The entropy of a byte sequence $(a_1, a_2, \cdots, a_{n-1}, a_n)$ is defined as:

$$H(X) = \sum_{i=0}^{255} -P_i \log_2 P_i \tag{1}$$

Here, P_i is the probability of byte i (equal to the count of occurrences divided by size n). It is defined as:

$$P_i = \frac{\sum_{j=1}^{n} \begin{cases} 1(a_j = i) \\ 0(a_j \neq i) \end{cases}}{n} \tag{2}$$

The range of $H(X)$ is $0 \leq \mathrm{H}(X) \leq 8$ .If $P_i = 0$, then

$-P_i \log_2 P_i$.

The entropy of any shellcode section in a file will be high, because shellcode contains executable instructions. On the other hand, the entropy of a non-shellcode part will be lower, because it represents the data in the document file. The entropy of any padding between data areas in the file will be very low, because it contains the same bytes.

For example, the start of a shellcode sequence is at (hexadecimal) address 5E00 in Table I. The area ahead of the shellcode is filled by (hexadecimal) 00, whereas the shellcode area has a random distribution of values.

Our system partitions an input document file into identifiable areas and calculates the entropies for these areas, looking for "Higher entropy byte sequence" or "Larger difference in entropy" cases. The system then performs emulations in priority order, starting with the candidate having the highest probability of being shellcode.

TABLE I: BINARY IMAGE AROUND A SHELLCODE ENTRY POINT

5DD0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5DE0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5DF0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5DF0	60	B9	A4	05	00	00	EB	0D	5E	56	46	8B	FE	AC	34	FC
5E00	AA	49	75	F9	C3	90	E8	ED	FF	FF	FF	61	15	C1	FE	FC
5E10	FC	AA	CF	3C	98	77	BC	CC	77	BC	F0	77	8C	E0	51	77

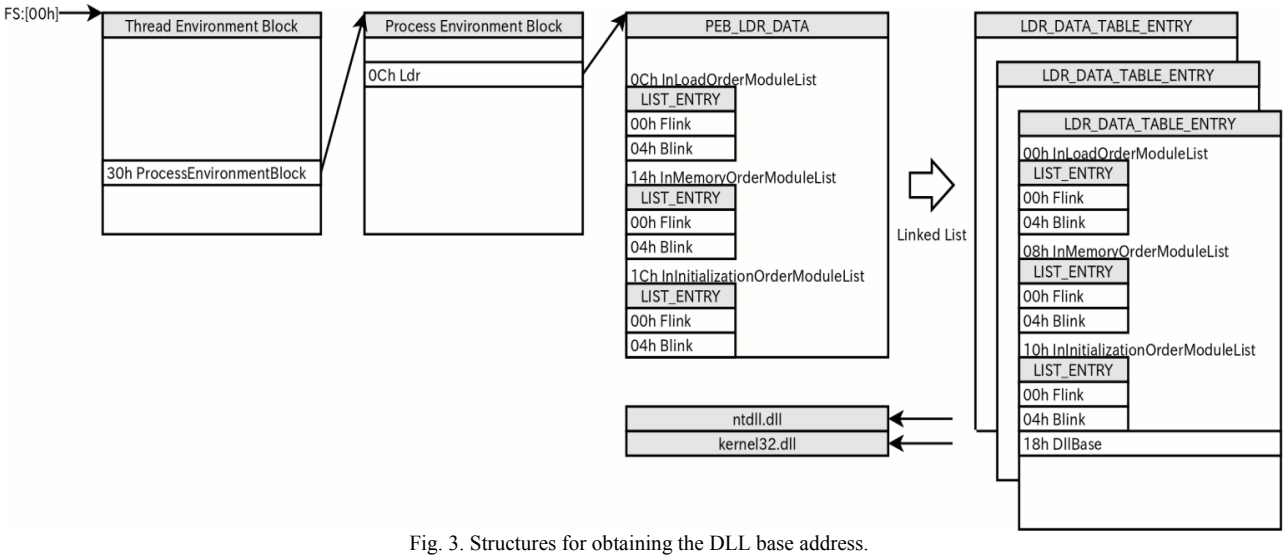

Fig. 3. Structures for obtaining the DLL base address.

C. Shellcode Detection

Our system regards a byte sequence having a high probability of being shellcode as executable code for a 32-bit Windows environment and executes it by using an emulator. If a byte sequence represents the start of shellcode, the byte sequence is executed and our system observes the features of the execution. If the emulation can no longer continue, or the system ceases to observe features after a certain number of steps, the emulation stops and applies this process to the next shellcode candidate. If no features are observed for any candidate, our system decides that the document file does not include shellcode.

Our system regards the following behaviors as features of shellcode.

1) self-modifying code,
2) access to the Process Environment Block (PEB) using the FS register, or
3) invoking the Application Programming Interface (API)

If the payload in the shellcode is encrypted, the payload is executed after the initial code decrypts the payload. Therefore, (1) is observed in this case. However, (2) is observed without (1) if the payload is not encrypted.

For 32-bit Windows, the FS register (a CPU register) is set to the address of the Thread Environment Block (TEB), which contains the current thread information. Fig. 3 shows the structures used to obtain the DLL base address. The shellcode obtains a linked list to LDR_MODULE by following the pointers in the structures for TEB, PEB and PEB_LDR_DATA. Finally, the shellcode resolves API addresses using the DLL base address obtained from a member of LDR_MODULE. Therefore, (2) is regarded as a feature of shellcode.

The shellcode invokes the API after obtaining the API address. Therefore, (3) is regarded as a feature of shellcode.

Our system detects the byte sequence as shellcode if it observes (1) and (2) or (2) and (3).

D. Executable File

If our system detects shellcode, it outputs a 32-bit Windows executable file to execute the shellcode. The executable file contains a document file, a filename and the address of the shellcode within the document file. Shellcode often obtains a file handle of itself by examining the file handles opened by an application. The executable file output by our system therefore creates a document file in a temporary folder and opens it to reproduce the state of the application before the shellcode is executed. The executable file disguises its filename as a standard application (e.g., WINWORD.EXE) to open the document file by hooking GetCommandLine and GetModuleFileName.

This method of disguise by API hooking has already been used by a previous analysis tool [9].

III. Preliminary Survey

Before building our system, we conducted a preliminary survey to determine the most appropriate parameters and algorithm using malware samples that we had already analyzed.

A. Sample Set

For the preliminary survey and the experiments, we chose a random subset of samples from the set of samples that could be estimated to satisfy the conditions of Section II.A. We used static analysis to confirm that the samples in the sample set contained shellcode, and noted the address of the shellcode in the document file. Of the 125 CFB samples chosen randomly, 73 satisfied the conditions of Section II. A, as did 15 of the 25 RTF samples. Table II shows the number of vulnerabilities for various file formats in the sample set. There were 15 types of vulnerability, with three samples having unknown vulnerability. There were 42 unique types of samples, in terms of the address of the shellcode and the vulnerability type.

The ratios for each of the CFB file formats are disproportionate, because we chose randomly without paying regard to the CFB file format. At this stage of obtaining the samples, we did not aim to equalize the file-format ratios. Therefore, the ratios in Table II are the approximate file-format ratios in the samples we obtained, which seems to reflect the file-format ratios for the attacks that we were targeting. The ratios for the vulnerabilities are also disproportionate. For the same reason, they also seem to reflect the proportion of attacks using the vulnerability in question.

B. The CFB Element Containing the Shellcode

In our analysis, all identified shellcodes were in the Stream area of the CFB. Note that our system does not distinguish a Mini Stream element from a Stream element.

C. Determination of the Number of Steps

We determined the necessary number of steps executed by the emulator to detect the features described in Section II.C. Table III shows the results of using an emulator to execute from the start addresses of the shellcodes in the sample set. Table IV shows the results of measuring the maximum number of steps required to observe the features.

D. Entropy Calculations and Algorithms

It is necessary to specify the sizes of byte sequences in entropy calculations. To determine an appropriate size n for equation (2), we calculated entropies for samples in the sample set ranging between 128 bytes and 2,048 bytes in length. The "difference in entropy" is the difference between the entropy for the target byte sequence and that for the byte sequence preceding the target byte sequence. Whenever a byte-sequence range would require consideration of out-of-file bytes, we terminated the range at the beginning (or end) of the file. We changed the byte-sequence size in steps of 16 bytes, because it would be too time-consuming to calculate entropies for all possible sizes. Table V shows the averages of the ratio of the number of emulation trials and the expectation (the number of emulation trials where the byte sequence was chosen randomly), calculated for sizes between 128 bytes and 2,048 bytes and for each algorithm ("Higher entropy byte sequence" or "Larger difference in entropy").

TABLE II: File Type and Vulnerability

Vulnerability	doc	xls	ppt	rtf	Total
CVE-2006-2389	4				4
CVE-2006-2492	13				13
CVE-2006-6456	2				2
CVE-2007-0671			1		1
CVE-2008-2244	5				5
CVE-2008-4841	1				1
CVE-2009-0556			1		1
CVE-2009-0563	1				1
CVE-2009-3129		24		5	29
CVE-2010-0822		2			2
CVE-2010-1901				1	1
CVE-2010-3333				6	6
CVE-2011-1269	1		2		3
CVE-2012-0158	13			2	15
CVE-2014-1761				1	1
UNKNOWN	3				3
Total	43	26	4	15	88

TABLE III: Observed Features

Feature	Number
(1)Self-modifying,(2)PEB access, (3)API call	55
(1)Self-modifying, (2)PEB access	2
(2)PEB access, (3)API call	17
None	14

TABLE IV: Maximum Step

Feature	Step
Start to (1)Self-modifying	35,847
Start to (2)PEB access or (1)Self-modifying to (2)PEB access	857
(2)PEB access to (3)API call	2,772,706

TABLE V: Average of Ratio of Emulation Trials

Size	Higher entropy byte sequence	Larger difference in entropy
128	0.561	0.33
192	0.581	0.317
256	0.578	0.288
384	0.554	0.268
512	0.593	0.27
1,024	0.715	0.305
1,536	0.817	0.403
2,048	0.882	0.55

The ratio of emulation trials is smallest where the size is 384 bytes and the algorithm is "Larger difference in entropy". Fig. 4 shows the distribution of the ratio of emulation trials when we used the most appropriate parameter and algorithm. The efficiency is higher where the ratio is smaller. It is more efficient than random choice if the ratio is less than 1.

IV. Experiments

As described in Section III-B, our system searches only the Stream area. As described in Section III-B, our system uses 16,384 as the step limit for the emulator if it observes self-modifying code and 4,194,304 if it observes PEB access. As described in Section III-D, our system is implemented so that the byte-sequence size for entropy calculations is 384 and the algorithm is "Larger difference in entropy".

A. CFB/RTF Analysis and Narrowing down by Pre-disassembly

The Stream area occupies 30.83% of all CFB files in the sample set and binary data occupies 11.95% of all RTF files. Disassembly was possible for 95.92% of the byte sequences

in all files.

B. False Positives

In addition to the sample set described in Section III, 125 benign files (doc:50, xls:25, ppt:25, rtf:25) were prepared. Our system did not attempt to extract shellcode from these benign files.

C. Shellcode Extraction

Using our system in the experimental environment of Table VI, we attempted to extract shellcode from 88 samples in the sample set. Shellcode could be extracted in 74 of these 88 cases.

Our system required 9,468 seconds for all processes. Shellcode extraction required 5,389 seconds, except for samples that did not include shellcode. However, only 4,399 seconds were required in one case. The average execution time for the booting emulator was 1.638 milliseconds. Fig. 5 shows the distribution of times to extract shellcode, omitting the 4,399-second case.

TABLE VI: ENVIRONMENT

	Shellcode Extraction	Dynamic Analysis
CPU	Pentium M 1.20GHz	Core i7 3.40GHz
Memory	1GB	512MB
OS	Ubuntu 10.04 LTS	Windows XP SP3 (Virtual Machine)

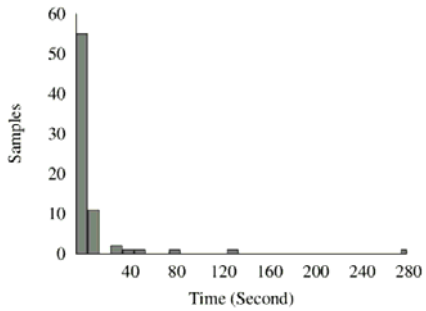

Fig. 5. Time for shellcode extraction.

D. Results of Dynamic Analysis

Table VII shows the results of executing the 74 executable files extracted by our system on a 32-bit Windows virtual machine. We regarded a document as malicious if the shellcode wrote a file and executed it (Drop) or accessed the network (Communication). We regarded cases as failures if the shellcode could not continue the execution because of an illegal instruction or a memory leak. There were also cases where execution was stuck in an infinite loop.

TABLE VII: RESULT OF EXECUTION

Success	Drop	50
	Communication	1
Failure	Memory	6
	Instruction	3
	Unknown	1
Infinite Loop		13

V. CONCLUSIONS

Shellcode is usually versatile. Neither the initial register contents nor the starting address of the loaded shellcode have to be specific values. However, our system cannot extract shellcode in some cases, and some executable files did not behave as malicious documents. Because some examples of shellcode need specific addresses or register values, they are not considered versatile.

Our system stops its emulation after the first API invocation, even if the executable file continues on the 32-bit Windows virtual machine. Therefore, the executable file did not behave as a malicious document, despite shellcode being extracted. We conclude that impersonation and reproduction such as specifying the address of shellcode or memory allocation by the application software was an insufficient condition.

A. Performance

For the narrowing down described in Section IV.A, about 30% of the byte sequences were shellcode candidates. However, the effect of narrowing down based on pre-disassembly was smaller than we expected. If the booting process for the emulator was enhanced or the examination environment was different, pre-disassembly might not be necessary.

B. Order of Priority

Our system determined the priority order for shellcode candidates using entropy. As shown in Fig. 4, the number of emulation trials was considerably smaller than expected. There were a few samples that required more time than expected (random choice would be better).

The difference between entropies for byte sequences such as that shown in Table I depend on the specification of the listed document formats described in Section II.A. If there are document formats that do not exhibit differences between entropies, the method used by our system would not be appropriate.

C. Broken Samples

As shown in Table VII, there were cases involving infinite loops because the file size was smaller than that expected by the shellcode. If such a document were to be opened by vulnerable application software, it would stick in an infinite loop and therefore not behave as a malicious document.

If we were to analyze the code statically for infinite loops before forming the sample set, we would be able to exclude these cases. However, this proof was due to the shellcode being executed by our system. Without using our system, we would not know that some of the samples were broken.

VI. RELATED WORK

A. Network Communication Analysis

Polychronakis *et al*. [4] proposed a method for extracting shellcode from network communications. In that paper, the shellcode candidates are determined on the basis of code that refers to its own address. The method observes self-modifying code via an emulator, as does our system. However, shellcode that does not modify itself is not extracted, even if it has other features such as PEB access or API invocation.

B. Structure Analysis of Document Files

Li *et al*. [5] proposed a method for detecting malicious Microsoft Word documents by statistical analysis that

involves entropy. However, the purpose was not shellcode extraction. They prepared many application software environments for opening documents to enable dynamic analysis. This is different from our system, because our system does not use application software to open documents.

C. Shellcode Analysis

Cova *et al*. [10] proposed a method for detecting malicious JavaScript. Shellcode was extracted using the method described in their paper. Our system, which extracts shellcode from document files, is different because their method extracts shellcode generated by JavaScript. Fratantonio *et al*. [9] proposed a tool for analyzing shellcode that is extracted via the method proposed by Cova *et al*. [10]. Our system is assumed to utilize dynamic analysis environments other than the 32-bit Windows environment used in our system. Consequently, our system does not have the capacity to list API invocations. Our system assumes that the issues for dynamic analysis discussed by Fratantonio et al. [9] can be resolved by other systems. However, the 32-bit Windows executable file output by our system disguises the name of the executable file, as does the application software that would open the document file. Their tool might be able to analyze the 32-bit Windows executable file output by our system, even though we expect that the output executable file would be analyzed via the environment proposed by Inoue and Yoshioka [11]–[13].

VII. Future Work

Our system can extract shellcode without requiring use of the vulnerable target application. We conclude that our system is effective and useful for analyzing malicious document files.

Our system focuses on the file formats listed in Section II.A. The malicious document file must include shellcode capable of being executed without use of the target application software. Our system has definite limits, although we do not know the proportion of malicious document files that can be analyzed by our system. The analysis of Return-oriented Programming documents or malicious documents that depend on their application software is an unresolved issue.

On the other hand, it would be possible to increase the proportion of analyzable documents by including additional acceptable formats. This might not be difficult because, for example, the Office Open XML format in Microsoft Office 2007 (and later) is CFB-based. If the original binary image could be extracted from compressed or encoded data, our proposal would also be effective for such document files.

We must increase the accuracy of the emulator, because the emulator did not observe features for some of the samples. Moreover, some executable files output by our system did not behave as malicious documents. Finally, the environment provided by our system must be improved to reproduce the environment of real application software.

References

[1] Mandiant. Mandiant Apt1: Exposing one of china's cyber espionage units. [Online]. Available: http://intelreport.mandiant.com/Mandiant_APT1_Report.pdf

[2] M. Prandini and M. Ramilli, "Return-oriented programming," *Security Privacy*, vol. 10, no. 6, pp. 84-87, 2012.

[3] Microsoft. Vulnerabilities in Microsoft Office Could Allow Remote Code Execution [Online]. Available: http://www.microsoft.com/en-us/download/details.aspx?id=10725

[4] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Network-level polymorphic shellcode detection using emulation," *Journal in Computer Virology*, vol. 2, no. 4, pp. 257-274, 2007.

[5] W. Li, S. Stolfo, A. Stavrou, E. Androulaki, and A. D. Keromytis, "A study of malcode-bearing documents," *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 231-250, 2007.

[6] Microsoft. Compound File Binary File Format. [Online]. Available: http://download.microsoft.com/download/9/5/E/95EF66AF-9026-4BB0-A41D-A4F81%802D92C/%5BMS-CFB%5D.pdf

[7] D. Rentz. The Microsoft Compound Document File Format. [Online]. http://www.openoffice.org/sc/compdocfileformat.pdf

[8] Microsoft. Word 2007: Rich Text Format (RTF) Specification. [Online]. Available: https://technet.microsoft.com/library/security/ms11-073

[9] Y. Fratantonio, C. Kruegel, and G. Vigna, "Shellzer: a tool for the dynamic analysis of malicious shellcode," in *Proc. the Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011, pp. 61-80.

[10] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proc. the World Wide Web Conference (WWW)*, 2010, pp. 281-290, Raleigh, NC.

[11] D. Inoue, K. Yoshioka, M. Eto, Y. Hoshizawa, and K. Nakao, "Malware behavior analysis in isolated miniature network for revealing malware's network activity," in *Proc. IEEE International Conference on Communications*, 2008, pp. 1715-1721.

[12] D. Inoue, K. Yoshioka, M. Eto, Y. Hoshizawa, and K. Nakao, "Automated malware analysis system and its sandbox for revealing malware's internal and external activities," *IEICE Transactions on Information and Systems*, vol. 92, no. 5, pp. 945-954, 2009.

[13] K. Yoshioka, D. Inoue, M. Eto, Y. Hoshizawa, H. Nogawa, and K. Nakao, "Malware sandbox analysis for secure observation of vulnerability exploitation," *IEICE Transactions on Information and Systems*, vol. 92, no. 5, pp. 955-966, 2009.

Kazuki Iwamoto received the B.S. degree in information science from Tokyo Denki University at Tokyo, Japan, in 1998. He received the M.S. degree in information engineering from Shinshu University at Nagano, Japan, in 2008. He entered the doctoral course in the Graduate School at Shinshu University in 2010.

He joined Japan Computer Security Research and started analyzing malicious software in 1998. He served on the IPA: Information-technology Promotion Agency in Japan as a visiting researcher in 2012. He is a senior software engineer, Advanced Research Laboratory at SecureBrain Corporation in Tokyo, Japan, where he joined in 2013.

Mr. Iwamoto is a member of Anti Virus Asia Researchers (AVAR) since 1999, a director in 2011 and 2012.

Katsumi Wasaki received the M.S. and Ph.D degrees in information engineering from Shinshu University at Nagano, Japan, in 1993 and 1997, respectively.

He is presently a professor of computer science, Department of Information Engineering at Shinshu University, where he initially joined in 1998. During occasional leave of absence from Japan, he was invited to visit Department of Computing Science, University of Alberta at Edmonton, Canada, in 2003 and 2005. He has served on the NEDO: New Energy and Industrial Technology Development Organization in Japan, as a member of technological evaluation committee in 2008. His current research interests include modeling and analysis of concurrent, parallel and/or distributed processing systems, mathematical model and formal verification of asynchronous circuits, and hardware compiler for model checking systems.

Prof. Wasaki is a member of IEEE, IEICE, IPSJ, IEEJ and JSiSE.