5 Cybersecurity Technologies: Countermeasures Against Emerging Threats
5-1 Technology for Supporting Obfuscated-Malware Analysis

Ryoichi ISAWA and Tao BAN

Many malware[*1] specimens emerge from the Internet every day, making it necessary for analysts to capture those specimens, analyze them, and create countermeasures more effectively. Our research focuses on the second step, which means how to further effective malware analysis. To achieve this, one of the most important challenges is generic unpacking, which can automatically extract the original binary of packed (compressed and/or encrypted)[*2] malware without depending on the applied packing algorithms. Generic unpacking is a key research topic because most specimens are packed to shield them from code analysis. This paper presents an effective generic-unpacking system.

1 Introduction

Cyberattacks are taking place each and every day, and their frequency is growing steadily. Many of the cyberattacks make use of malware. For example, banking malware, a type of cyber threat, has become widely known in recent years. This malware gains unauthorized access to a PC and acquires online banking information for siphoning money from bank accounts. Ransomware is also rampant: it encrypts documents on a PC, preventing them from being read. A PC infected with ransomware may display a screen showing the steps to send money to the attacker, suggesting that the user may obtain the key to decode the encrypted files if he/she sends money. The victim, who desperately needs the correct key to enable access to the files, has no other option than to send money. However, the transmission of money is no guarantee at all that he/she will receive the correct key. To prevent infection by malware, or, to enable quick recovery from it, the introduction of effective measures is urgently needed to capture, analyze, and disinfect it. Successful analysis of malware sheds light on its infection vectors and behaviors, enabling appropriate measures to be taken. We have conducted study in this area, placing special focus on attaining greater analysis efficiency. For obstructing analysis[*3], many instances of malware are known to be packed (compressed and/or encrypted). Any attempt to analyze malware in code analysis, therefore, needs to identify the type of packing and devise procedures to unpack it, before extracting the malware code (hereafter referred to as the "original code"). The process may be quite time consuming. On the other hand, malware authors can pack the malware almost effortlessly using such packing tools (hereafter referred to as a "packer") as UPX[1] and Themida[2]. Because many packers are easily available, the type of packer used for the malware is usually unknown at the point when it is captured. Suffering from it, an analyst has to identify and unpack each time he/she captures an instance. Therefore, automatic extraction of the original code could contribute to reducing the time required before starting actual analysis, resulting in a drastic increase in efficiency of the entire process. We have undertaken research and development of a generic unpacking system, with a view to increasing analysis efficiency.

Generic unpacking systems have been an active area of research up to now[3]–[9], and they generally perform unpacking procedures taking advantage of the following characteristic behaviors common to packed programs: when a packed program is initiated, it first boots up an unpacking routine (decoding code), which reconstructs the original code from the packed format and writes it in

*1 Software developed with an intention to perform malicious acts
*2 Among the obfuscation techniques used in malware, this study focuses on those related to packing (encryption/compression).
*3 Analysis to draw relevant information from programming code

memory as the decoding process proceeds; then the reconstructed original code is executed. The point from where the reconstructed code is executed is called the OEP (Original Entry Point).

As described above, every packer starts execution after it writes the original code into memory, meaning that the original code remains either in written or executed code. Detection of the OEP enables the researcher to know the position from where the original code starts executing. Therefore, detection of the OEP provides a challenge for researchers.

Commonly used techniques for detecting written/executed code includes the single-stepping execution method and the data execution prevention method. The single-stepping execution method is a two-stage strategy: the program is run once followed by instruction-by-instruction[*4] execution, for which the written instruction and address are each recorded. For each execution of an instruction, the address on which it is written/executed is detected as an OEP candidate. One drawback of this technique is the very long execution time inevitably associated with instruction-wise execution. In the data execution prevention method, an execution prohibition setting is placed in the memory area where a write operation is executed. If an instruction in the execution-prohibited memory area is invoked for execution, an execution prohibition exception is thrown, enabling detection of the write/execution operation. Although this method provides much faster processing than the single-stepping execution method, information on CPU registers and other elements can be obtained only when an exception is thrown. A drawback to this technique is the paucity of information that can be used to identify the OEP-related instructions among those written or executed.

In view of the situation in which a large number of malware occurrences are being observed on a daily basis, we selected the data execution prevention method because of its potential to deliver faster processing. Based on this method, we propose a generic unpacking system featuring high accuracy for OEP detection. The proposed system provides the following two functions.

The first function tries to detect the unpacking routine, which consists of the parent instructions that regenerate the code to be written to memory and executed. It takes advantage of the behavior of the unpacking routine of writing the regenerated code in memory. The address of the instruction executed immediately after the unpacking routine has finished its operations constitutes the most probable candidate for the OEP. The second function sorts the OEP candidates in the order of likelihood, starting from the most probable one. This likelihood-based sequence facilitates the routine to find the authentic OEP in the shortest steps: when the most probable candidate is found to be false, then the routine tries the next in the sequence.

The generic unpacking method of our own development performs a packer identifying routine[10] as the initial step of the whole unpacking procedure. If the packer identification routine successfully finds a known packer, and if the unpacking algorithm corresponding to the packer has already been implemented, then the implementation is used preferentially. If the packer is unknown — i.e. not registered in the system – the generic unpacking system is used to locate the OEP.

In view of the ever increasing occurrences of malware incidents in recent years, we consider that improving the efficiency of malware analysis is a challenge of significant importance that NICT should face seriously. Research and development of a system capable of locating the OEP accurately, combined with the use of conventional systems from the viewpoint of practicality, will reduce the burden of the analysts who have to cope with malware incidents.

This report consists of the following sections. Section **2** provides the basic knowledge for understanding the basic operations of the packer and fundamentals of the data execution prevention method. Section **3** mainly describes how a generic unpacking system works. The results of the evaluation experiments are given in Section **4**, and summarized in Section **5**. See also references [10] and [11] for further details of the achievements reported here.

2 Basic knowledge

2.1 Packer

The packer is a software tool used for packing (encrypting and/or compressing) programs. When it packs a target program, it also appends an unpacking routine to it, enabling the program to self-extract for subsequent execution. Figure 1 illustrates the file structure of a program packed with UPX (Hello_upx.exe). This packed file, Hello_upx.exe, consists of the following sections: PE header, empty section, packed program (Hello.exe), and unpacking routine. When put into execution, the unpacking routine has the highest priority to run. It unpacks the compressed program and

*4 “Instruction” in this paper means machine language instructions such as MOV and JMP[18].

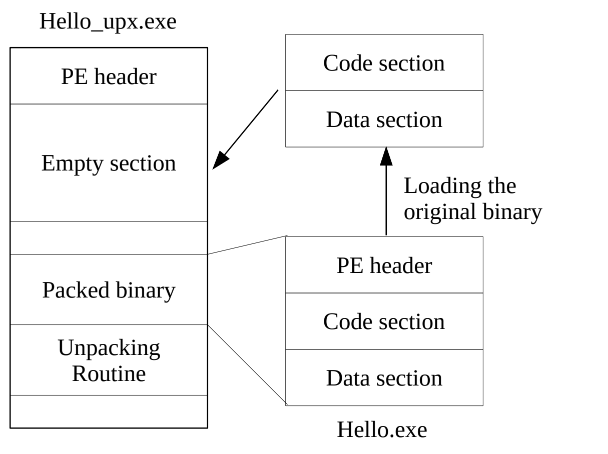

Fig. 1 File structure and behavior of a program packed through the use of UPX

lays it out in memory. In this process, the unpacked executable, Hello.exe, is written in the empty section for execution.

While UPX is an example of a very simple packer, some others, such as Themida and ASProtect[12], employ a very complex packing algorithm. The unpacking process may differ depending on the selection of the packer. Note, however, irrespective of the choice of packer, the original program is invariably unpacked in memory, followed by writing and then execution.

The key challenge for generic unpacking is to devise an efficient method to detect the OEP among the array of written/executed code, or to extract the original code.

2.2 Basis of analysis

When the single-stepping execution method is used, the program proceeds in a step-by-step fashion: one instruction is executed at a time, followed by a halt and disassembly[*5]. This procedure enables instruction-by-instruction acquisition of such information as: the type of instruction executed, execution of the sequence of instructions, how memory is modified, and the contents of CPU registers. The single-stepping execution method can be realized either in a virtual environment (e.g., Xen[13], KVM[14], and QEMU[15]) or by deploying a Dynamic Binary Instrumentation (DBI) tool, notably PIN[16] and Valgrind[17].

The data execution prevention method is characterized by its ability to prohibit the code in a specified memory area from running. Some CPUs — typically Intel64 and IA-32 architecture[18] — virtually partition the memory space into 4096-byte memory pages, and manage the behavioral attributes of each page — i.e. write

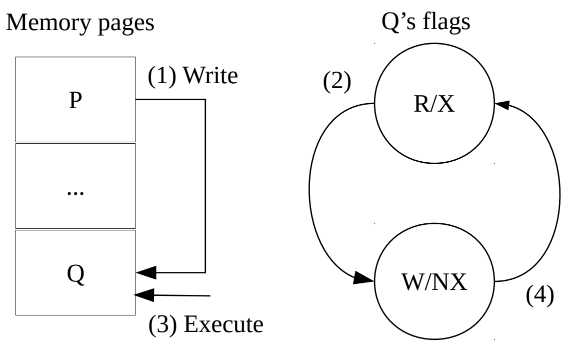

Fig. 2 Flag state transitions (R/X: Read-only(write-inhibited)/eXecutable, W/NX:Writable/non-eXecutable)

enabled/disabled and execution enabled/disabled. Figure 2 illustrates the mechanism to detect written/executed instructions. The entire memory is initially set to the R/X (Read-only/eXecutable, namely write inhibited) state. Then, as shown in the Figure, a write operation is performed to the memory page Q. This write attempt throws a write-inhibited exception, informing the system that the attempt was made to the memory page Q. Then the system assigns the W/NX (Writable/None-eXecutable, i.e. write-enabled but execution-disabled) attribute to Q. When an attempt is made to execute an instruction in the memory page Q, an execution-inhibited exception is thrown, informing the system that the write/execution instruction was made to the particular page. Upon acknowledging the execution-inhibited exception, the system sets the R/X attribute to the memory page Q again. The system can obtain the address of an OEP candidate instruction by following these steps. Note, however, as this is a page-by-page memory management scheme, it does not permit inhibiting any particular instruction from executing. Because acquisition of information (CPU register and others) is triggered only by the throw of an exception, the amount of information obtainable using this method is necessarily smaller than that accessible by the single-stepping execution method. On the other hand, thanks to smaller frequencies of execution halts, this method is capable of faster processing than the single-stepping execution method.

*5 Process to convert binary data into an instruction

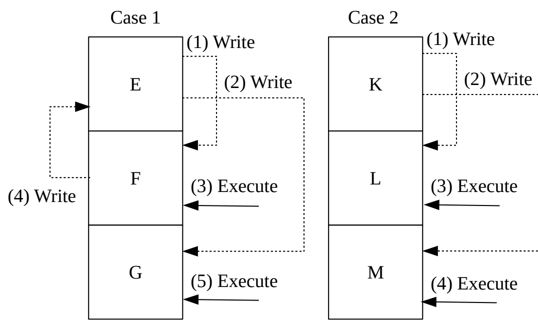

Fig. 3 A case study of OEP identification (E-G, K-M: memory page)

3 Generic unpacking system

3.1 Basic idea

The generic unpacking system does not target all the packed programs: rather, it is a selective method — it runs a packer identification routine[10] in an early stage to find out the type of packer used. If it is a known packer and a corresponding unpacker is available, the unpacker should be used. In general, the unpacker specifically designed to meet the characteristics of a packer has better accuracy to locate the OEP than a generic unpacking system.

Our generic unpacking system uses the data execution prevention method with a view to locating the OEP faster. This method, in addition to outputting the address associated with the instruction that performed a write/execution operation, is capable of outputting a set of OEP candidates with order of priority (determined by using the following two functions). The first of these two functions, after running of the packed program, examines the written/executed instructions to identify those that constitute the unpacking routine. The role of the unpacking routine, as has been explained in Subsection **2.1**, is to generate instructions that are written/executed. The technique used here is to detect the parental instruction that generated the written/executed instruction using the data execution prevention method, and use it as an unpacking routine instruction. The address of the instruction executed immediately after the unpacking routine is assumed to be the most probable OEP candidate.

The second function searches for the next probable OEP candidate under an assumption that the likelihood is higher if it is located nearer to the most probable OEP in the execution sequence. Based on this idea, the function sorts and outputs all of the instruction addresses that were written/executed. Figure 3 shows two diagrams illustrating case studies to identify the most probable OEP candidate. In case 1, the instruction in memory page E is written to F, followed by writing it to G as well. Next, based on the fact that the instruction in F is executed, all of the instructions in E are judged to constitute the unpacking routine. Subsequently, an instruction in F performs a write operation to E. In this case, a page is also considered to belong to the unpacking routine if its instruction makes a write operation to the page that has been judged to be a part of the unpacking routine. The idea behind this judgement is that the unpacking routine pages share data between each other. Lastly, an instruction in G is executed. Here, because E and F have been judged to belong to the unpacking routine, steps (1) to (4) are considered to be the instructions executed by the unpacking routine. From all these, the address of the next executed instruction, step (5), is assumed to be the most probable OEP candidate. In case 2, an instruction in K performs a write operation to L and M, and an instruction in L is executed subsequently. Therefore, K is judged to contain the unpacking routine. In case 2, because K is the only page that is judged to contain the unpacking routine, the address of the instruction executed immediately after the completion of instructions in K, i.e. step (3), presents the most probable OEP candidate.

3.2 Generic unpacking algorithm

Two types of memory page are defined. Suppose an instruction A writes data[*6] to an arbitrary location in a page, and a certain instruction in the page is executed. Then the memory page to which the instruction A belongs is defined as a “code generating page.” Suppose an instruction B writes data in a code generating page. Then the memory page to which the instruction B belongs is defined as a “data sharing page.” In case 1 of Fig.3, E is a code generating page and F is a code sharing page. By the same token, K in case 2 is a code generating page. Based on the reasons described in Subsection **3.1**, a code generating page and data sharing page are identified as being an unpacking routine.

In order to determine the type of a memory page, the following steps are taken following the execution of the packed program. Each time a write inhibit exception is thrown, a pair of addresses — the address of the instruction that executed the write operation (hereafter “*src*”) and the

*6 Generically called “data,” because it is often difficult to draw a clear distinction between an instruction and a value of a program variable at the moment of writing.

destination address of the operation (hereafter "*dest*") — are saved in an array W. In parallel to this, when an execution inhibit exception is thrown, the address of the corresponding instruction is saved in an array X. These steps are repeatedly carried out until the execution of the packed program comes to an end, or for the duration of the pre-configured timeout period. The next step is to check for each pair in W, if the memory page to which *dest* belongs coincides with one of the memory pages to which any one address in X belongs. If the page is determined, the memory page to which the *src* (which is paired with the *dest*) belongs is assumed to be the code generation page. Then a check is made to confirm if the *dest* writes data, for each pair in W, to the code generation page. If true, the memory page to which the *src* belongs is assumed to be the data sharing page. Next, a check is made if each address in X belongs either to the code generation page, or to the data sharing page. If true, the address is assumed to be the address of the unpacking routine.

Lastly, the addresses in X and all the *src*'s in W are simultaneously sorted to determine the address of the unpacking routine. The address in X that immediately follows it is assumed to be the most probable OEP candidate.

To define the priority order for OEP candidates, all the addresses in X are arranged in the order of the execution sequence, creating an array X=(X_0, X_i,…,X_i,…,X_p,…,X_{l-1},…), where Xp is the address of the most probable OEP candidate, p is its index, X_{l-1} is the address of the last executed instruction, l is the address number in x, and i=0,1,2,…, l-1.

Then x is sorted using the following formula:

$$j = \begin{cases} 0 & (i = p) \\ 2|p-i| - (1 + sign(p-i))/2 & \text{(otherwise)} \end{cases} \qquad (1)$$

where j is the index for xi after sorting, |･| represents an absolute value, and sign (･) represents a sign function that returns -1 if the input is negative, and +1 otherwise. In summary, x is sorted to form an array (X_p, X_{p-1}, X_{p+1}, X_{p-2}, X_{p-2},…). For example, if p is set to 2 when x is represented by an array (x0, x1, x2, x3, x4, x5, x6), the sorted result would look like (x2, x1, x3, x0, x4, x5, x6).

In case 1 of Fig.3, the address of the step (5) instruction is assumed to be the address of the most probable OEP candidate. This comes from the fact that step (4) is the last address of the unpacking routine, and step (5) represents the instruction executed immediately following it. In case 2 of Fig.3, the address of the step (3) instruction is assumed to be the address of the most probable OEP candidate.

4 Evaluation experiment

4.1 Data sets and environment for the experiment

Malware samples for evaluation experiments, 35 in all, were selected in the following fashion. Analysis of the hundreds of thousands of malware samples we have collected in the past few years using the "Antivirus software" (Symantec) revealed that they can be classified into any one of 135 categories. The categories include such species as "Backdoor.IRC.Bot" and "Unknown." Among them, we selected 35 categories in the order of malware population (note, however, that "unknown" is excluded). The next step was the examination to determine if the samples of each category were packed. We used PEID and the method proposed by Lyda et al.[19] in this procedure. PEID is a signature based tool that determines if a program is packed. On the other hand, Lyda et al.'s method uses the entropy principle to determine if a program is packed or not. Subsequently, random selection was made to extract one — excluding packed ones — from each category, resulting in a set of 35 samples. The hash value (SHA256[20] of each sample was calculated to make sure that none coincide with others. Then all 35 samples were packed using 25 types of packers, and the samples that were found to be inoperative were discarded. The remaining packed samples — 753 in all — served for the experiment. Table 1 lists the names of the packers used in this experiment. We made sample selection from the un-packed ones for reliable OEP determination.

To provide the environment of the experiment, Windows XP was installed as a guest OS in VirtualBox[21]. Each of the samples was unpacked on this guest OS to guarantee accuracy of the generic unpacking system. Note that the packer identification function described in Subsection **3.1** "Basic Idea" was not used. Unpacking was carried out solely by applying the algorithms described in Subsection **3.2**.

4.2 Experimental Results

Table 1 summarizes the results of the data execution prevention method: column 4 shows the average of the number of addresses corresponding to the written/executed instructions, and column 5 shows the standard deviation. For example, ASPack 2.33 was used to pack 35 samples, and generated on average 9.23 instructions that were written/executed. The standard deviation was 5.14.

Our generic unpacking system sorts the addresses of

Table 1 Results from the experiments ('#' designates the number of samples, '−' indicates 100%)

	Packer		OEP candidates		Recall (%)						
No.	Name	#	AVG	SD	n = 1	2	4	6	8	16	32
1	ASPack 2.33	35	9.23	5.14	94	97	100	−	−	−	−
2	ASProtect 1.70	35	67.20	12.66	37	40	43	43	43	43	86
3	exe32pack 1.42 trial	10	7.00	2.86	90	100	−	−	−	−	−
4	Exe Stealth 2.73 trial	35	9.43	6.23	97	97	97	100	−	−	−
5	Ezip 1.0	35	10.00	6.03	94	97	97	100	−	−	−
6	FSG 2.0	34	8.76	5.37	94	97	100	−	−	−	−
7	Mew11SE 1.2	33	11.42	7.50	94	97	97	100	−	−	−
8	MoleBoxPro 2.6.4 trial	35	23.29	5.16	0	3	6	37	97	100	−
9	mpress 2.19	31	9.87	4.46	94	97	100	−	−	−	−
10	nPack 1.1.300	33	9.39	5.27	97	100	−	−	−	−	−
11	NsPack 3.7 trial	34	10.00	6.15	94	97	97	100	−	−	−
12	Packman 1.0	35	9.29	5.08	97	100	−	−	−	−	−
13	PECompact 2.79 trial	34	10.94	6.05	94	97	97	100	−	−	−
14	PESpin 1.33	34	13.12	5.51	94	97	97	97	97	97	100
15	Petite 1.4	18	11.56	7.10	56	61	94	100	−	−	−
16	PKLITE32 1.1	14	9.57	5.43	14	100	−	−	−	−	−
17	RLPack 1.20	34	10.00	5.20	94	97	100	−	−	−	−
18	SimplePack 1.0	33	10.00	6.14	94	97	97	100	−	−	−
19	tElock 0.99	19	8.42	5.32	95	100	−	−	−	−	−
20	Themida 2.2.7.0	32	300.47	16.57	0	0	3	44	75	97	97
21	Upack 0.399	26	11.73	6.18	88	92	96	100	−	−	−
22	UPX 3.08	34	10.09	6.05	94	97	97	100	−	−	−
23	WinUpack 0.31	33	10.21	6.17	94	97	97	100	−	−	−
24	WWPack32 1.20 trial	22	9.18	4.41	95	100	−	−	−	−	−
25	yoda's protector 1.02	35	14.71	5.58	97	97	97	97	97	100	−
	Total	753		Average	81	85	87	92	96	97	99

written/executed instructions in the order of OEP likelihood. We considered the unpacking was successful if the authentic OEP fell within the n-th on the list arranged in the order of likelihood. We examined the ratio of successfully unpacked samples, with varied n values, for each packer' samples. This evaluation index is called Recall[22][23]. The Recall values for each packer are listed in Table 1. For example, in case n=2 for ASPack 2.33, the Recall value becomes 97%, indicating successful unpacking for 34 samples (97% of the total sample number #).

The column headed by "n=1" in Table 1 indicates that 19 packers achieved a ≧90% success ratio of unpacking. The value n=1 means that the most probable candidate was indeed the OEP. The column headed by "n=8" in Table 1 indicates that 23 packers achieved a ≧97% success ratio of unpacking. That is, the authentic OEP could be reached within the 8th unpacking attempt on the sorted candidate list. See reference [13] for comparison with other systems. It also proposes a Recall improvement method applicable to ASProtect 1.7.

A measurement was made to evaluate the level of performance overhead produced by this system using a PC installed with an Intel Core i7 3.4 GHz CPU. A sample was run on this system without using the generic unpacking system, and the time required from the start to end of the process was measured. This process was repeated 5 times, and the average was calculated. Subsequently, the same sample was run on this system using the generic unpacking system, and the time required from the start to end of the process was measured. This process was also repeated 5 times, and the average was calculated. The sample selected for this experiment was Trojan.Panddos. Note, however, for the experiment that used PKLITE32, a different sample, Trojan.Usugelgen 3, was selected instead, because the

Table 2 Overhead associated with generic unpacking systems

Packers used for samples	Size (KB)	Original time (msec)	Time margine (msec)	(%)
ASPack	29	103	5	4.9
ASProtect	176	153	5	3.3
exe32pack	30	103	-3	-2.9
Exe Stealth	69	99	13	13.1
Ezip	69	101	4	4.0
FSG	24	100	-1	-1.0
Mew11SE	24	102	5	4.9
MoleBoxPro	94	119	12	10.1
mpress	26	109	-1	-0.9
nPack	29	98	9	9.2
NsPack	25	100	12	12.0
Packman	24	94	9	9.6
PECompact	26	107	-2	-1.9
PESpin	47	130	3	2.3
Petite	31	99	4	4.0
PKLITE32	111	36	6	16.7
RLPack	24	97	4	4.1
SimplePack	25	98	0	0.0
tElock	38	107	5	4.7
Themida	1184	1220	41	3.4
Upack	22	107	5	4.7
UPX	26	93	8	8.6
WinUpack	22	103	12	11.7
WWPack32	36	103	5	4.9
yoda's protector	44	6399	33	0.5

packer failed to unpack Trojan.Panddos.

The third column in Table 2, "Required time without using the system" (hereafter referred to as "original time"), shows the measured time without using the system. The fourth column "Time difference: With and without using the system" lists the measured time differences between the two modes of running the process, i.e. with or without using the system. The fifth column lists the increase ratios (%) of the running time. It is well understood from the table that time elongation due to the use of the generic unpacking system is commonly very small. For several packers, such as exe32.pack, the time required to complete the process was even reduced by using the system.

The time difference between the two operation modes hit the largest value for Themida, indicating an increase by 41 msec. This can be ascribed to Themida's characteristics: the unpacking routine very frequently throws write/execution exceptions, creating 27 msec of overhead for our system to extract the written/executed instructions. A residual 14msec was consumed for sorting the written/executed instructions (including the time required to determine the most probable OEP candidate). For other packers, the time required to complete the sorting of written/executed instructions fell invariably within 2 msec. From these results, it can be concluded that the overhead generated due to the use of the generic unpacking system is very small. With the use of the system, the sum of the time required to complete the process for 753 samples amounted to 1,061 seconds, thus, the per-sample average was 1.41 seconds. In conclusion, the researcher/engineer has to wait only 1 or 2 seconds before he/she can obtain the OEP candidate for one sample.

5 Conclusion

We have undertaken research and development of generic unpacking systems, an overview of which is presented here. In the face of the ever-increasing number and variety of malware attacks, stepping up the efficiency of malware analysis is urgently needed. The generic unpacking system provides an enabling technology to attain significantly higher efficiency in the analysis.

We are planning to continue research to cope with the threats of malware. In addition to aiming at highly efficient analysis, we will also identify other challenges to solve through collecting malware instances, detailed analysis, and derivation of optimal measures. Major contents and achievements described in this report have been published elsewhere: see reference [10] (packer identification) and [11] (generic unpacking).

References

1 M. F. Oberhumer, L. Molnar, and J. F. Reiser, "UPX: Ultimate Packer for eXecutables," available at http://upx.sourceforge.net/, (Last access: April 21st, 2016).

2 Oreans Technologies, "Themida," available at http://www.oreans.com/themida.php, (Last access: April 21st, 2016).

3 P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack- Executing Malware," Proceedings of the 22nd Annual Computer Se- curity Applications Conference (ACSAC'06), pp.289–300, 2006.

4 M.G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code ex- tractor for packed executables," Proceedings of the 5th ACM work- shop on Recurring Malcode (WORM'07), New York, NY, USA, pp.46–53, ACM, 2007.

5 A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware Analysis via Hardware Virtualization Extensions," Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08), New York, NY, USA,

pp.51–62, ACM, 2008.

6 Y. Kawakoya, M. Iwamura, and M. Itoh, "Memory behavior-based automatic malware unpacking in stealth debugging environment," Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE'10), pp.39–46, 2010.

7 H.C. Kim, T. Orii, K. Yoshioka, D. Inoue, J. Song, M. Eto, J. Shikata, T. Matsumoto, and K. Nakao, "An Empirical Evaluation of an Unpacking Method Implemented with Dynamic Binary Instrumentation," IEICE Trans. Inf. & Syst., vol.94-D, no.9, pp.1778–1791, 2011.

8 L. Martignoni, M. Christodorescu, and S. Jha, "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware," Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07), pp.431–441, 2007.

9 F. Guo, P. Ferrie, and T.C. Chiueh, "A Study of the Packer Problem and Its Solutions," Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08), Berlin, Hei- delberg, pp.98–115, Springer-Verlag, 2008.

10 R. Isawa, T. Ban, S. Guo, D. Inoue, and K. Nakao, "An Accurate Packer Identification Method using Support Vector Machine," IEICE Trans. Fundamentals, vol.E97-A, no.1, pp.253–263, Jan. 2014.

11 R. Isawa, D. Inoue, and K. Nakao, "An Original Entry Point Detection Method with Candidate-Sorting for More Effective Generic Unpacking," IEICE Trans. on Info. & Syst., vol.E98-D, no.4, pp.883–893, April 2015.

12 StarForce Technologies Ltd., "ASPack Software," http://www.aspack.com/asprotect32.html, (Last access: April 21st, 2016).

13 XenProject, "TheXenProject," available at http://www.xenproject.org/, (Last access: April 21st, 2016).

14 R.H.O.S. Community, "Kvm: Kernel-based virtual machine," available at http://www.linux- kvm.org/, (Last access: April 21st, 2016).

15 F. Bellard, "QEMU." http://www.qemu.org/, (Last access: April 21st, 2016).

16 Intel Corporation, "Pin - a dynamic binary instrumentation tool," available at http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, (Last access: April 21st, 2016).

17 N. Nethercote and J. Seward, "Valgrind: A framework for heavy-weight dynamic binary instrumentation," SIGPLAN Not., vol.42, no.6, pp.89–100, June 2007.

18 Intel Corporation, "Intel 64 and ia-32 architectures software developer's manual," available at http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf, 2014.

19 R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," IEEE Security and Privacy, vol.5, no.2, pp.40–45, March 2007.

20 P.G. John Bryson, "Secure hash standard (shs) (federal information processing standards publication 180-4)," available at http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf, 2012.

21 Oracle, "Oracle VM VirtualBox," available at https://www.virtualbox.org/, (Last access: April 21st., 2016).

22 J. Han, M. Kamber, and J. Pei, "Data Mining: Concepts and Techniques," Third Edition, Morgan Kaufmann, 2011.

23 B. Croft, D. Metzler, and T. Strohman, "Search Engines: Information Retrieval in Practice," 2009.

Ryoichi ISAWA, Ph.D.

Senior Researcher, Cybersecurity Laboratory,
Cybersecurity Research Institute
Malware Analysis, Network Security

Tao BAN, Dr. Eng.

Senior Researcher, Cybersecurity Laboratory,
Cybersecurity Research Institute
Cybersecurity, Network Security