# Memory Behavior-Based Automatic Malware Unpacking in Stealth Debugging Environment

Yuhei Kawakoya Makoto Iwamura
Mitsutaka Itoh

NTT Information Sharing and Platform Laboratories
9-11, Midori-Cho 3-Chome, Musashino-Shi, Tokyo 180-8585, Japan
{kawakoya.yuhei,iwamura.makoto,itoh.mitsutaka}@lab.ntt.co.jp

## Abstract

*Malware analysts have to first extract hidden original code from a packed executable to analyze malware because most recent malware is obfuscated by a packer in order to disrupt analysis by debuggers and dis-assemblers. There are several studies on automatic extraction of hidden original code, which executes malware in an isolated environment, monitors write memory accesses and instruction fetches at runtime, determines if the code under execution is newly generated, then dumps specific memory areas into a file as candidates for the original code. However, the conventional techniques output many dump files as candidates for the original code when experiments are conducted on malware in the wild. Thus, manual identification of the true original code is needed. In this paper, we present "memory behavior-based unpacking," an algorithm that automatically identifies the true original code from among many candidates depending on the change in the trend of accessed memory addresses before and after the dumping points. To achieve this algorithm, we have implemented Stealth Debugger, a virtual machine monitor for debugging and monitoring all memory accesses of a process without interruption by any anti-debug functions of the malware. We have evaluated our proposed system by using malware obfuscated by various common packers. The results show that our proposed system successfully finds the original entry points and obtains the original code of the malware.*

## 1. Introduction

Most recent malware is obfuscated and has anti-debug features to delay analysis connected to the development of antivirus signatures or removal tools. It has been reported that 80% of wild malware is obfuscated by some type of packer [7]. For example, when a packed malware starts execution, an extraction routine runs to de-obfuscate the packed codes into the original codes then expand them into some memory regions; this procedure is called unpacking. After the extraction routine, the instruction pointer of the CPU (i.e., eip) jumps to the expanded original code. The code is the true body of the malware; it has functions for sending spam mail, stealing confidential information, and performing other malicious activities. Usually, when malware analysts investigate a packed malware, they have to manually extract the hidden original code with a debugger. Manual unpacking is very time consuming and expensive. Thus, automatic and generic malware unpacking has been investigated [7][13][4][11]. Almost all the studies focus on the algorithm that monitors the execution and write access to memory of a malware. This algorithm generates dumped files when the malware is about to execute a self-extracted memory region. This region is the code expanded on a memory by the de-obfuscated routine of the malware. While the algorithm works effectively in the case of malware packed on a single stage, i.e., on a single layer, it does not work well in the case of malware packed on multiple stages. This is because the algorithm outputs many files as candidates for the true original code, but it does not automatically identify the true original code from among these candidates.

In this paper, we present an algorithm called "memory behavior-based unpacking" for identifying the true original code from among many dumped files. Our algorithm focuses on changes in the main accessed memory addresses between the extracting code and the original code. We express the rate of the change as a score and identify the dump point where the highest score is computed. The dumped file at the dump point is defined as the true original code. To achieve the algorithm, a mechanism for monitoring all memory accesses of a malware process is essential. There are two main techniques for monitoring all memory accesses: simulator-based or instrumentation-

978-1-4244-9356-2/10/$26.00 ©2010 IEEE

tool-based. Even though these techniques are effective for finding bugs or debugging in software development, they are not effective for analyzing malware because the anti-debug features of the malware easily detect them. Both simulator- and instrumentation-tool-based techniques have heavy overheads for capturing all memory accesses and writing the accessed information into a log file. In addition, some implementations depend on the debugging features of the CPU or OS. Therefore, we have developed an environment, Stealth Debugger, for capturing all memory accesses. Stealth Debugger can analyze malware without being disrupted by anti-debug features and time checking. Stealth Debugger uses original debugging functionalities embedded in a virtual machine monitor consisting of virtualized hardware, which hides the existence of a debugger from the malware running on a guest OS. In addition, Stealth Debugger has the timeline trick technique for avoiding a time-checking-type anti-debug feature. This technique controls the elapse of the ticks on a guest OS, i.e., it stops the ticks on a guest OS when Stealth Debugger captures memory accesses and writes the logs into files by issuing I/O. If we use timeline trick, it is possible to hide the overhead resulting from monitoring of all memory accesses.

We implemented our proposed techniques, established an automated unpacking system, and then evaluated it by using malware obfuscated by ten common packers. Our experiment shows that for six malware our system accurately identified the true original code from among many candidates. If we included the second highest scores, our system found eight malware.

The contributions in our paper are as follows.

- The memory behavior-based unpacking algorithm is proposed for identifying the true original code of malware from among many dumped files. Previous studies have concentrated on the phase of generating candidates for the original code. How to find the true original code from among the candidates has not been discussed.

- Stealth Debugger, which is a debugger for avoiding some anti-debug features, is presented and its implementation reported.

- The timeline trick technique for defeating malware that has a time-checking anti-debug function is introduced, and its implementation in Stealth Debugger is reported.

- The implementation of our proposed techniques and integration of them as an automatic unpacking system is reported. The system is evaluated by using malware obfuscated by common packers.

## 2. Related Work

In this section, we explore previous studies related to unpacking methods. The most primitive unpacking method is manual unpacking with a debugger. It is time consuming, and special analytical skills of a certain level are required. If the packer used in a malware can be precisely identified, the unpacker corresponding to the packed binary could be found. However, even though there are several studies and tools[10] for identifying a specific packer used by a malware, the packers used for almost all malware are unknown or hard to identify. Therefore, many generic unpacking methods have been proposed.

Royal et al. have developed PolyUnpack, which is based on the observation that sequences of packed or hidden code in a malware can be made self-identifying when the malware runtime execution is checked against its static code model[14]. Kang et al. have proposed a generic unpacking system named Renovo[7], which is developed on TEMU[15]. Renovo monitors all memory write accesses by malware and marks the written memory areas as dirty. When the malware is about to execute dirty memory areas, i.e., the areas the malware has written by itself, Renovo creates a memory dump at that time. Azure[13], Ether[4], and Suffron[11] differ in their implementations and platforms, yet the algorithm they have adopted for judging when they should dump memory regions into a file is the same as that of Renovo. That is, all of these systems monitor written and executed memory areas. To observe memory accesses by malware, Azure is implemented with an Intel-VT[6], Ether is developed as a module for Xen hypervisor[2], and Suffron leverages Intel-PIN[8]. Martignoni et al. have proposed OmniUnpack, which captures dangerous system call invocations, defines the time of first invoking one of these dangerous system calls as completion of code extraction, and then dumps specific memory regions[9]. An analyst acquires an original code of a packed malware with standard forensic tools such as Volatility [1] by dumping physical memory after executing a malware and waiting for a minutes.

Many generic and automatic unpacking methods have been proposed in the past. However, these methods are not automatic in the true sense of the term. The reason is that these methods generate many dump files as candidates for the true original code; this requires analysts to manually identify the true original code from among the candidates. Table 1 shows how many candidates were generated when we conducted an experiment on a system by using an algorithm similar to that of the related works, i.e., monitoring written and executed memory, and then dumping specific memory regions when the malware is about to execute the code written by itself. These proposed methods would work well in the case of single-layered packed binaries. The methods would generate only one dump file, so it would be

unnecessary to identify one true original code from among a number of dumped files. However, such a case is very rare in the real world, as Table 1 shows. Almost all malware is obfuscated by multiple-layered packers, so to achieve automatic unpacking, it is necessary to identify the true original code from among dumped files.

Meanwhile, standard-forensic-tools-based unpacking cannot correctly stop the execution at the original entry point. This means the acquired dumped file is just a memory dump and cannot be executed again. In addition, there is a possibility that the functions around the original entry point might be erased or altered by the malware after executing them. Therefore, the best dumped file for an analyst is one which is acquired just when the original entry point is executed.

## 3. Our Proposals

As we mentioned in the previous section, there have been several studies on automatic malware unpacking. However, these studies did not focus on multi-layered packed malware. In other words, even though the previous studies tended to generate many candidates, a method was not proposed for identifying the true original code from among the candidates. Hence, malware analysts have to manually identify the true original code.

In this paper, we present memory behavior-based unpacking, which automatically identifies the true original code from among the candidates generated by a similar technique to that of previous studies. Our method focuses on the shift of the mainly accessed memory range. We suppose that the memory access behavior of the codes would be widely different depending on their roles. In malware, the purpose of extracting code is to create a body of malicious code in the memory. This malicious code aims to conduct malicious activities, such as stealing confidential information, sending spam mail, and establishing malicious servers. Based on this supposition, our proposed technique monitors the memory access around dumping points when a dumped file is created, i.e, the malware is about to execute a newly generated piece of code. Our technique also calculates the score, using the amount of change in the memory access trend before and after a dumping point, then identifies the true original code based on the score.

To achieve the technique noted above, a mechanism for obtaining all memory accesses by a malware is necessary. Mainly two types of techniques for capturing all memory accesses have been proposed: simulator-based or binary-instrumentation-tool-based. However, neither type of technique can be adopted for malware analysis because of the anti-debug functions of malware. While the simulator-based technique takes a lot of overhead to execute a block of code and collect an all-memory-accesses log, the instrumentation-tool-based type often utilizes the debugging functionalities of the OS and CPU. This means that a malware can easily discover the existence of those functionalities. Therefore, we have proposed Stealth Debugger, a virtual machine monitor that hides the existence of debugging functionalities and the overhead accompanied by logging memory accesses. Stealth Debugger can avoid debugger detection because it does not depend on a debugging support function of the CPU and OS, and it is also able to avoid the time-checking type anti-debug through the timeline trick technique, which stops the ticks on a guest OS while information on the memory accesses is obtained.

Using these proposed techniques, we can acquire an all-memory-accesses log of a malware without being disrupted by the malware, and thus it becomes possible to establish the system of memory behavior-based unpacking.

### 3.1. System Design

Figure 1 shows our whole proposed system. There are two main components in our system. First is "Stealth Debugger", which is a virtual machine monitor embedded with debugging functionalities and specialized for malware analyzing. Stealth Debugger collects all memory access information and obtains memory dumps when malware running in a guest OS is about to execute some self-expanded piece of code. Second is the component for computing the score around dumping points, which are the points at which dumped files are created, and for identifying the true original code from among the dumped files based on the score.

### 3.2. Stealth Debugger

Stealth Debugger is a debugger for malware analysis and is established on QEMU, which is an open source x86 emulator[3]. Most recent malware has anti-debug functionalities, such as detecting the existence of a debugger in the same environment or timing its own execution on a specific code path and comparing that time with the time previously taken in another environment. If the malware detects the existence of a debugger or searches the overhead to execute a specific code path, it disrupts the debugging process by forcibly killing the debugger process or changes its behaviors by jumping to an irregular code path.

In Stealth Debugger, we have achieved debugging functionalities without the usual supports of the CPU and OS. We have embedded debugging functionalities, such as breakpoint, single step, memory access trace, and others, into a virtual machine layer so that it becomes possible to hide the existence of the debugger from the malware (Fig. 2). In addition, Stealth Debugger is able to control the tick of time in a guest OS to avoid the malware detecting the larger amount of time taken to execute a specific block of code; this is called "timeline trick." More accurately, timeline trick stops the elapse of time in a guest OS during the time that it debugs and analyzes a malware running on the guest OS, such as interactively issuing commands through a controller or writing memory access/execution trace logs

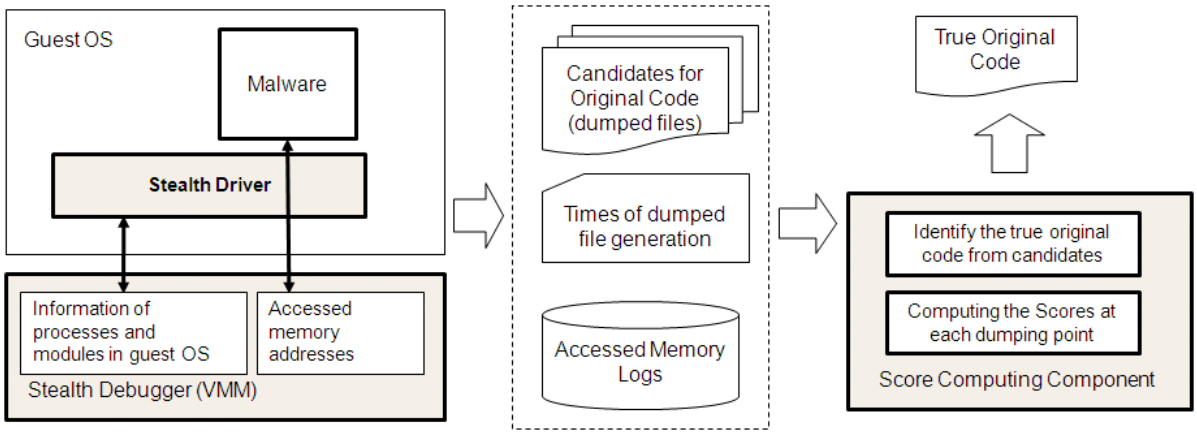


**Figure 1. Overview of proposed system**

into files on a host OS (Fig. 3). Thus, Stealth Debugger is capable of debugging and analyzing malware without being affected by various types of anti-debug functionalities.

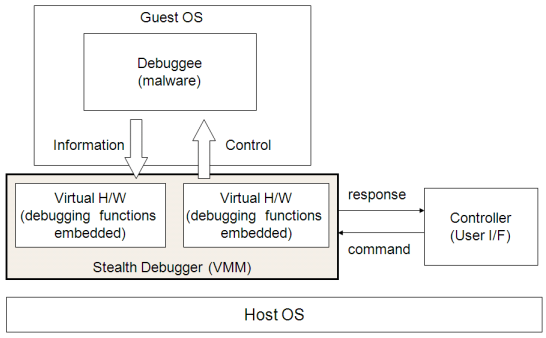


**Figure 2. Architecture of Stealth Debugger**

### 3.3. Bridging Semantic View Gap

As often discussed in virtual machine-based IDS research, there is a semantic view gap between a guest OS and host OS. The guest OS information acquired in a host OS is too primitive to accurately grasp the situation in a guest OS. Stealth Debugger has the same problem, and it is hard to grasp the situation of a malware running in a guest OS from the information only acquired at a virtual machine layer. To bridge the semantic view gap, we install a kernel module, named Stealth Driver, in a guest OS to collect the guest OS specific information, process IDs, virtual memory addresses of modules, and PE header information of a process running in a guest OS. The kernel driver communicates with a virtual machine by a special interrupt and passes information to a virtual machine monitor. Stealth Driver is implemented with Windows callback functions registered by "PsSetCreateProcessNotifyRoutine" and "PsSetLoadImageNotifyRoutine" APIs. These callback functions are invoked whenever processes are created and destroyed, and modules are loaded or unloaded. Thus, Stealth Debugger is able to know all processes running and all modules loaded in a guest OS. However, we have to protect Stealth Driver from malware attacks because Stealth Driver is installed on a guest OS, i.e., it runs in the same environment as the malware. To protect Stealth Driver from a malware attack, we adopt a rootkit-like technique, Direct Kernel Module Manipulation[16]. This technique de-chains the LDR_DATA_TABLE_ENTRY data structure of the target module from a loaded module linked list managed by a Windows kernel. The linked list is referenced by many Windows system tools for enumerating loaded modules. We de-chain the data structure of Stealth Driver from the linked list, which hides it from malware process.

### 3.4. Output Dumped Files

In our system, to obtain dumped files as candidates for the true original code, we have adopted a technique similar to that in a previous work[9], which observes the memory access of a malware at the page level and then detects that the malware is about to execute a self-extracted piece of code. Our system memorizes written memory addresses based on the page level (i.e., 4 KB) because the performance is more effective than that on an instruction level, even though more dumped files would be generated (discussed in Ref. [9]). Our system captures all memory accesses for both writing and instruction fetching at the MMU and registers all memory pages that include the written addresses. When a malware fetches the instructions in the registered memory pages, our system stops the ticks of the guest OS and dumps specific memory regions into the files.

### 3.5. Computing Scores

In this section, we introduce the algorithm for identifying the true original code from among the candidates. We calculate the score of all dump points by the following algorithm. After that, our algorithm decides that the dumped file generated at the dump point with the highest score is the

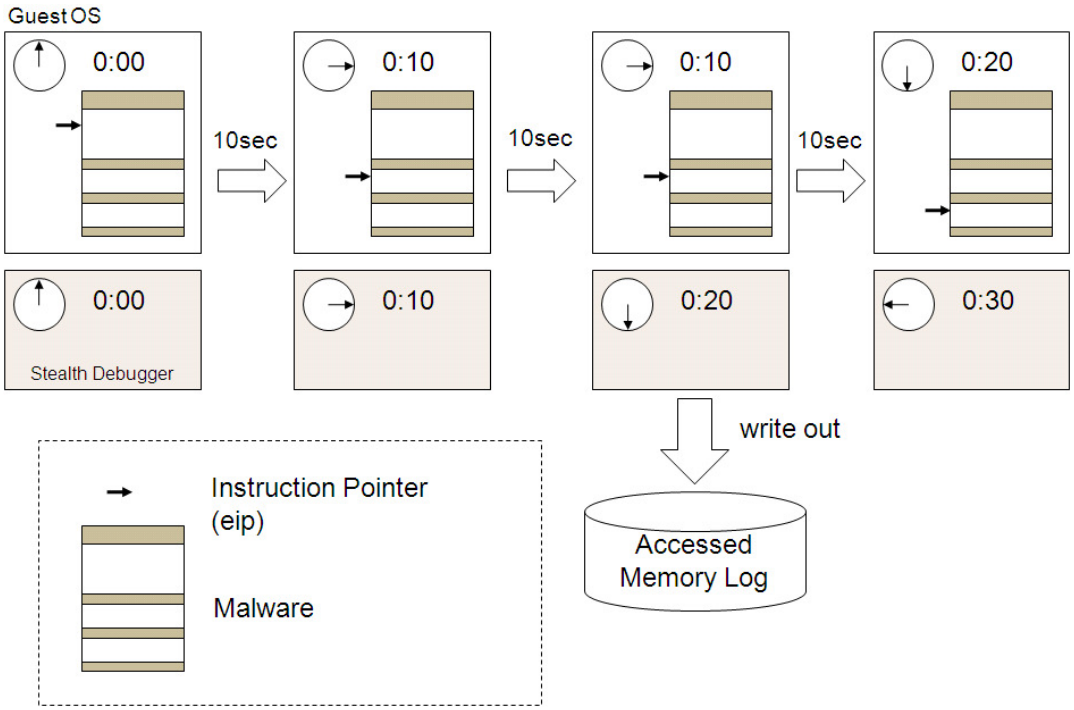


**Figure 3. Timeline Trick**

true original code. All computation and selection of files is processed at the Score Computing Component in Fig. 1.

First, the component receives types of memory access data, i.e., read, write, and fetch, and the time of each dumping point from Stealth Debugger. We define the set of the addresses of accessed memory during $T$ as $A(T) = (a_1, a_2, ..., a_n), 0 < i < n$, where $a$ is a memory address. $N$ is the number of dumped files, where the set $t_j = (t_1, t_2, ..., t_j), 0 < j < N$ is the time at which the dumped files are created. We define the average of accessed memory addresses during $T$ as the characteristic of the term. Thus, the value of this characteristic is calculated by the below formula.

$$F = \frac{1}{n}\sum^{n} A(T) \tag{1}$$

The value of the characteristic at several seconds, $\alpha$, before and after dump points, $t_i$, is calculated as follows.

$$F(i)_{before} = \frac{1}{n}\sum^{n} A(T_i) \quad t_i - \alpha < T_i < t_i \tag{2}$$

$$F(i)_{after} = \frac{1}{n}\sum^{n} A(T_i) \quad t_i < T_i < t_i + \alpha \tag{3}$$

From the above expressions, we define the value of the $i-th$ dump points as below.

$$v(i) = |F(i)_{before} - F(i)_{after}| \tag{4}$$

Then, we normalize the value by the next expression.

$$v(i)' = \frac{v(i)}{v_{max}} \tag{5}$$

We calculate the scores in all cases of read, write, and fetch, and then compute the score of $i-th$ dump points by adding all the scores at $i-th$.

$$S_i = v(i)'_{read} + v(i)'_{write} + v(i)'_{fetch} \tag{6}$$

We determine the highest of the scores of each dump point and suggest that the file dumped at the time with the highest computed score is the true original code.

## 4. Experimental

To evaluate the effectiveness of our proposed method, we implemented Stealth Debugger with timeline trick and the component for computing scores, thus establishing our proposed system. On the system, we conducted the experiment as follows. First, we prepared one malware that was not obfuscated. This malware is a bot type and tries to send query packets to the DNS server in order to communicate with C & C servers after being executed. We packed the malware by using ten common packers, i.e., UPX, Aspack, MEW, PECompact, ASProtect, Yoda's Protector, FSG, nPack, WinUpack, and SDProtector, and then we prepared the ten malware for which the original entry points and the original code were already known.

Second, we put one of the packed malware into a guest OS on Stealth Debugger and executed it. Stealth Debugger monitors the memory access behaviors of the malware and then writes the memory access logs into a file. At the same time, Stealth Debugger outputs specific memory regions into files, i.e., creates dumped files when the malware is about to execute the self-extracted memory areas. After starting the execution, we kept monitoring the above behaviors until the malware began to send packets. This was based on the observation that functions which send packets are usually in the body of a malware, so the extraction routine has completed when we observe any packets from the malware.

Third, we calculated the score at each dumping point and then compared these scores in order to determine the dump point with the highest score, which indicates that the biggest change of memory addresses has happened at that dump point. We define the dump point chosen by our algorithm as the point at which the original entry point of the malware is executed, and the dumped file generated at the point as the true original code.

Last, we evaluated our system by comparing the hashed value of a code section of the dumped file our system picked out with that of the original malware. The way which we compared the original code with the dumped file is as follows. We calculate the hash value of all local functions of the original malware with IDA pro[5]. And then, we check whether the dumped file has these functions of the original malware. unpacking is considered successful if almost all functions of the original malware are included in the dumped file. In addition, we checked if the execution point of the original entry point our algorithm computed was the same as that of the original entry point we obtained from the unobfuscated malware.

Our experiment had the following conditions.

- To calculate the scores at dumping points, the target term is five seconds before and after the dumping point ($\alpha = 5$).
- The target monitoring address range is that at which the image of a malware is loaded and the heap area used by the malware.
- The stack area is not included in our monitoring range.

The experiment was conducted in the following environment: Dell Precision T5500, Intel Xeon 2.27 GHz, 12G RAM, 64bit Windows7, with Stealth Debugger implemented on QEMU-0.9.1.

## 5. Results

Table 1 shows the results of our experiment. Our proposed system succeeded in finding the original entry point of six types of packed malware. If we include $\triangle$ matches, it is successful at finding eight malware. We failed to find the remaining two. In addition, we succeeded in unpacking the nine types of packed malware. We will next show the accessed memory address graphs of three types of cases: successful, nealy, and failure.

Figure 4 shows a graph of memory addresses accessed by the malware packed by UPX. In the case of UPX, our system succeeded in identifying the true original entry point from among 16 candidates. In addition, for all the types of memory accesses, i.e., read, write, and fetch, it indicates the highest score at the true original entry points. The dump point at 143.830 seconds is a correct point, i.e., the time when the original entry point of the malware is executed. The graphs make understanding where the true original point is easy because the range of memory accesses around the true original entry point is the most widely changed.

Figure 5 shows the results for WinUpack. We failed to find the true original entry point of the malware packed by WinUpack. However, the true dump point was the second highest score obtained by our algorithm. Therefore, it is possible to greatly reduce the time taken to identify the true original code from among the candidates. Our proposed system identified the dump point at 428.925 seconds, the point around which the mainly accessed addresses of read and write access changed. Around the true original entry point, we can see the biggest change of the fetch addresses, but there is almost no shift of the write address range (Table 5). Therefore, the total score failed to indicate the correct entry point.

**Table 2. Top three scores of WinUpack**

| RANK | TIME | READ | WRITE | FETCH | TOTAL |
|---|---|---|---|---|---|
| 1 | 428.925 | 0.966 | 0.894 | 0.351 | 0.737 |
| 2 | 853.096 | 0.754 | 0.263 | 1.000 | 0.672 |
| 3 | 853.098 | 0.762 | 0.684 | 0.444 | 0.630 |

Figure 6 shows the results for ASProtect. In this case, we had 54 dump points. The ranking of the read memory access that our algorithm computed as an original entry point was 17th, the ranking of the write was 35th, and the fetch was 2nd. The resultant total ranking of the score was 10th. Thus, we failed to discover the true original entry point. The reason our system failed to identify the true dump point at 225.053 seconds is that 12 dump points are located close together between 220 and 222 seconds, so the scores of the points differ only a little. As a result, the true original code is mis-identified. In addition, ASProtect uses the Stolen Bytes technique[17], which modifies a few bytes of the original functions. So, the original functions of the original code included in the dumped file is only 60%.

## 6. Discussion

In this section, we discuss the limitation of our proposal method. As we mentioned in previous sections, our proposal method focuses on the difference in memory access

**Table 1. Results of our proposed method**

| PACKER | VERSION | ♯ FILES | COMPUTED OEP | CORRECT OEP | MATCH | UNPACK | RANK |
|---|---|---|---|---|---|---|---|
| UPX | 3.04 | 16 | 142.830 | 142.830 | ◯ | ◯ | 1 |
| Aspack | 2.2 | 16 | 66.706 | 66.706 | ◯ | ◯ | 1 |
| MEW | 11 SE v1.2 | 17 | 168.262 | 168.262 | ◯ | ◯ | 1 |
| PECompact | 3.02 | 18 | 168.887 | 168.887 | ◯ | ◯ | 1 |
| ASProtect | 1.5 | 54 | 225.053 | 220.623 | × | △ | 10 |
| FSG | 2.0 | 16 | 66.906 | 65.159 | × | ◯ | 8 |
| nPack | 2.0 | 16 | 205.827 | 205.827 | ◯ | ◯ | 1 |
| WinUpack | 0.39final | 17 | 428.925 | 853.096 | △ | × | 2 |
| Yoda's Protector | 1.03.3 | 18 | 72.919 | 72.916 | △ | ◯ | 2 |
| SDProtector | BE v1.12 | 21 | 348.986 | 348.986 | ◯ | ◯ | 1 |

The ♯ FILES is the number of dumped files when we execute a malware obfuscated by a packer. The COMPUTED OEP is the time of the highest score computed by our proposed system. The CORRECT OEP column is the time when the correct original entry point is executed. If the COMPUTED OEP and CORRECT OEP match, ◯ is set in the MATCH column. If not, × is set. △ is set when the correct time is included in the top three scores calculated by our system. UNPACK is ◯ when the dumped file for which our system computed the highest score has almost all functions(90%) of the original malware. Under 90% we set △ and if we fail to almost all, × is set.

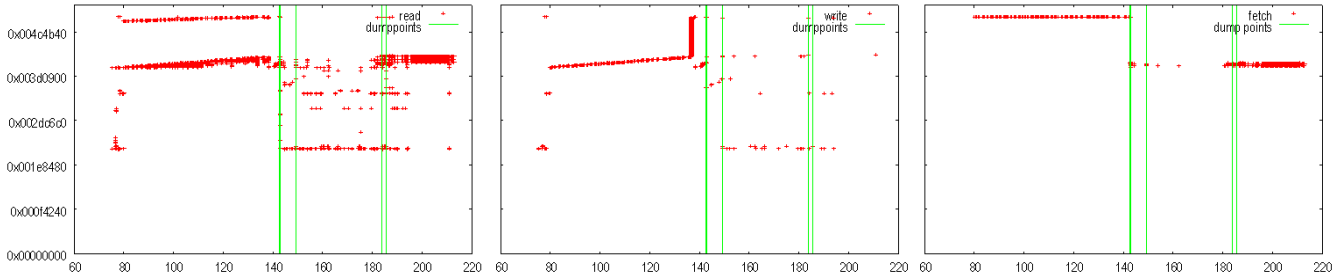


**Figure 4. UPX**

behaviors between an extracting code and an original code. Therefore, it is difficult to detect an original entry point of packed malware, in which the border is not clear. ASProtect is such a case. It modifies some functions of an original code, including the entry function, using the Stolen Bytes. This makes the memory access behave like an extracting code after executing the original entry point. Therefore, our proposed method may fail to detect the original entry point of such types of malware. It is also difficult to detect the original entry point of malware packed by a virtualization protector packer and unpack. This type of packer transforms several functions of an original code into a packer-specific intermediate language and executes them on its original virtual machine. Almost all virtualization protector packers may erase most of the memory access behaviors of an original code by transforming the instructions into their original language so that we cannot distinguish between the extracting code and the original code. To unpack the malware packed by a virtualization protector requires dedicated unpacking procedures and tools, as [12] reported.

## 7. Conclusion

The problem of identifying the true original code of malware from among dumped files that conventional automatic unpacking methods generate has been unsolved for several years. In this paper, we described the memory behavior-based unpacking method, an approach to identifying the true original code of malware from among the candidates. Our approach is based on the observation that the accessed memory address range around the true original entry point is extensively changed. To achieve the algorithm, an environment that is capable of collecting all memory accesses without being disrupted by malware anti-debug functionalities is necessary. Thus, we implemented Stealth Debugger, a virtualized hardware with embedded debugging functions and the timeline trick technique, and our proposed algorithm for identifying the true original code from among multiple dumped files. We evaluated our proposed system by using malware obfuscated by ten different common packers. The results of the experiment showed that our proposed system succeeded in collecting a memory access log of all malware while avoiding anti-debug tricks. Our system also found the

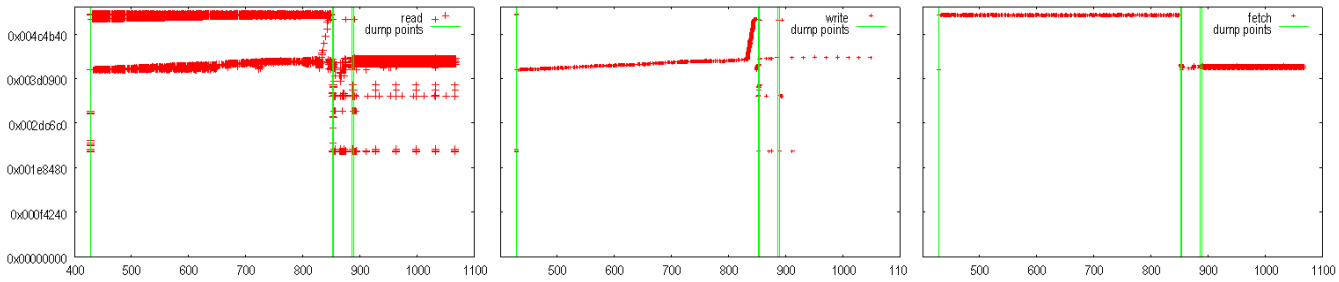


**Figure 5. WinUpack**

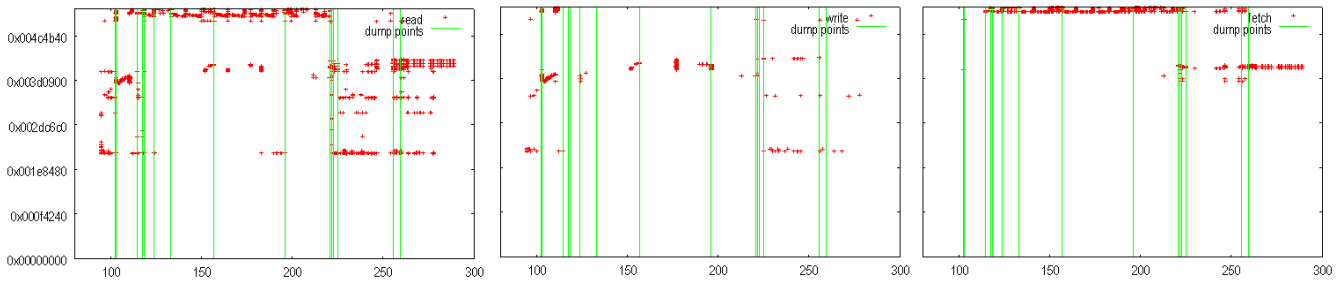


**Figure 6. ASProtect**

true original code of six malware. Our proposed method can reduce the cost for malware analysts since it is not necessary to manually identify the original code.

## References


[1] The volatility framework. `https://www.volatilesystems.com/`.

[2] P. Barham, B. Dragovic, K. Fraser, S. H, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles(SOSP)*, pages 164–177, 2003.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–46, April 2005.

[4] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security*, pages 51–62. ACM, November 2008.

[5] HexRays. `http://www.datarescue.com/`.

[6] Intel. Intel 64 and ia-32 architectures software developers manual. Specification, 2007.

[7] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM'7)*, October 2007.

[8] C. keung Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[9] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007.

[10] PEiD. `http://www.peid.info/`.

[11] D. Quist and Valsmith. Covert debugging circumventing software armoring techniques. In *Black Hat USA 2007*, August 2007.

[12] R. Rolles. Unpacking virtualization obfuscators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies*, August 2009.

[13] P. Royal. Alternative medicine: The malware analyst's blue pill. In *Black Hat USA 2008*, August 2008.

[14] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. *Computer Security Applications Conference, Annual*, 0:289–300, 2006.

[15] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, Dec. 2008.

[16] W.-J. Tsaur, Y.-C. Chen, and B.-Y. Tsai. A new windows driver-hidden rootkit based on direct kernel object manipulation. In *ICA3PP '09: Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing*, pages 202–213, 2009.

[17] M. V. Yason. The art of unpacking. In *Black Hat USA 2007*, August 2007.