JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 35, 887-901 (2019)
DOI: 10.6688/JISE.201907_35(4).0011

Efficient Automatic Original Entry Point Detection*

GYEONG-MIN KIM, JUHYUN PARK, YUN-HWAN JANG AND YONGSU PARK+
Department of Computer Science
Hanyang University
Seoul, 04763 Korea
E-mail: {casualab; hdhyun216; dbsghksdlwkd; yongsu}@hanyang.ac.kr

Malware authors employ sophisticated anti-reverse engineering techniques such as packing, encryption, polymorphism, *etc.* For a packed file, when launched, the packed executable will reconstruct the code of the original program. The OEP (Original Entry Point) is the address indicating the beginning point of the original code. Previous work or conventional unpacking tools provide a relatively large set of OEP candidates and sometimes OEP is missing among candidates. In this paper, we present an efficient OEP detection scheme for x86 Windows environments. This scheme is designed to find exact one OEP by using three methods. First, we enhanced Isawa *et al.*'s work by examining branch instructions. Our second method is to track the system parameters relevant to the main function in stack memory to refine OEP candidates. Our third method is that we track the startup function calls to find the installation routine for exception handling. To evaluate feasibility, we implemented our algorithm and then conducted experiments on 16 commercial representative packers and 6 previous unpacking tools/schemes. Experimental results show that even though our scheme produces a single OEP candidate for each packed file, accuracy is the highest (up to 14 times higher than the previous work).

Keywords: anti-reverse engineering, malicious code analysis, code obfuscation, program analysis, computer security

1. INTRODUCTION

Malware is becoming more and more advanced with deploying diverse anti-reverse engineering techniques. According to AV-TEST [1], more than 92% of malware is compressed, encrypted, or packed. Such protection techniques often cause a major inconvenience in malware analysis and response.

Packing is the obfuscation method that uses compression or encryption to hide the original code from analysts. Packed programs cannot be statically analyzed in detail because the original program is encrypted or compressed. Dynamic analysis, *e.g.*, analysis after decompressing or decrypting the packed program, can give more information. When the packed file is executed, first the unpacking routine reconstructs the original code in the memory and then transfers the control flow to the beginning point of the reconstructed code. We call this point (the address indicating the beginning point of the original code) OEP (Original Entry Point). If the analyst finds the OEP, he/she can dump the original program in the memory for further analysis.

The unpacker is an automatic tool to help find the OEP and dump the original program in the memory. However, commercial unpackers have their own limitations. For

Received March 9, 2018; accepted July 26, 2018.
Communicated by Meng Chang Chen.

+ Corresponding author.

* This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2017R1D1A1B03029550).

example, PEiD [2] gives various information for the program including entry point, packer name, import table address, *etc.* However, for the packed program it cannot find the OEP. Universal PE unpacker [3] is a plug-in module for IDA Pro for generic automatic unpacking. Shortcomings of Universal PE unpacker is that a prior knowledge about the possible range of the OEP should be given. PinDemonium [4] is a generic unpacker, which is based on Pin [5], finds OEP candidates and uses Scylla to dump the memory and to reconstruct the library function table used by the program. However, the number of OEP candidates provided by PinDemonium is large and the task of identifying OEPs depends entirely on the capabilities of the malware analysts.

In this paper, we present a new scheme for finding OEP for diverse packers. First, it tracks both WrittenAndExecuted (which will be explained in Section 2) addresses and branch instructions for finding the OEP candidates set. Then, it refines the OEP candidates set by looking up the command-line parameters of the main function. Then, it tracks the system startup function calls up to the main function to find the SE handler installation routine. Finally, it regards the address closest to the SE handler as the OEP.

To test feasibility of our scheme, we have implemented the proposed scheme and the previous work and then conducted experiments for OEP detection with various representative commercial packers. Our scheme produces a single OEP candidate for each packed executable and nonetheless accuracy is the highest (up to 14 times higher than the previous work).

The remainder of this paper is organized as follows. Section 2 explains related work and Section 3 describes the property of the OEP and the proposed scheme. We deal with implementation issues in Section 4 and show experimental results in Section 5. Section 6 concludes the paper.

2. RELATED WORK

For malware analysis and binary code analysis, a significantly large amount of research work has been done up to now [6-10]. However, topics relevant to generic unpacking and OEP detection techniques have not drawn strong interest from academic researchers. In this section, we briefly explain recent research work on generic unpacking and OEP detection techniques.

Generic Unpacking using Entropy Analysis [11]: In [11], Jeong *et al.* proposed a general unpacking mechanism for finding OEP using entropy analysis. Entropy is a measure of uncertainty in a series of numbers or bytes [12]. When a general executable is packed (encrypted or compressed), the degree of disorder in the executable increases, causing the entropy value to increase [12]. Entropy measurements are done by measuring the entropy of the memory section. Comparing with other sections having a high entropy value, the section having a relatively low entropy value is determined as unpacked, *i.e.*, original binary code is written. The equation for obtaining the entropy value of a specific section *x*, *H*(*x*), is as follows:

$$H(x) = -\sum_{i=0}^{255} p(i)\log_2 p(i) . \tag{1}$$

The entropy measurement of section x is done through a statistical experiment. Suppose that the section x is a byte array where the length is s and that we observe each byte value in the array. The sample space of the experiment is a set of values that 1 byte can have and the event i is one of the values from 0x00 to 0xff. $p(i)$ is the probability that event i occurs, that is, the probability of having the value i when observing a specific 1 byte. The base of the log is 2 according to [12].

The method of finding the OEP in [11] is as follows. First, it runs the program until the branch (*e.g.*, JMP, JCC, CALL, or RET) instruction is executed. Then, the program execution is paused, and the entropy of the target section is analyzed. If it has a low entropy value but previously it had high one, this algorithm regards the target address as the OEP. However, if it is not, it continues to execute the next instruction. The entropy threshold value, which is determined to be an unpacked section, is 4.1-4.6 [11]. As in Section 5, experimental results show that this approach is not so accurate for finding OEP compared with other methods, *i.e.*, it has high false positives/negatives.

OEP Detection Method with Candidate-Sorting [13]: In [13] Isawa *et al.* proposed a new dynamic analysis scheme to identify OEP. It monitors program execution and memory writes and checks whether the code is generated at runtime or not. This method is divided into two parts: 'tracking the decoding routine' and 'sorting the OEP candidates.'

Suppose that memory is divided into pages (4K Bytes). [13] uses the following terms. A generating page is the memory page that writes some data into another memory page, where this data will be executed later, and a sharing page is a memory page that writes some data to the generating page or the sharing page.

This scheme has two array lists (W, X). Initially, it marks all the pages as R/X, where R/X means that it is possible to read and execute data on this page. First, the packed program is loaded into memory and then executed. A write page-fault exception occurs when writing data to the R/X page. Then, (src, dst) address pair for the write instruction is put to the array W. For this page, it changes the mark as W/NX, meaning that it is possible to write data but impossible to execute it. If an instruction on the W/NX page is subsequently executed, an executing page-fault exception will occur. For each exec page-fault, the corresponding address, e, is put to the array X. An exec page-fault exception occurred means that the instruction at e, which is being executed, was written before. Then, this page is changed back to R/X. We call this address (e) *WrittenAndExecuted* throughout this paper. Array X can be viewed as the list of WrittenAndExecuted addresses. This marking process is continued until the packed program stops executing.

'Tracking the decoding routine' is performed as follows. First, it sorts each pair of elements in the W array in chronological order. Then, it checks the time whether the instruction at the dst address is executed later by comparing the execution time of the element address in X. If so, the page with the src address paired with the dst address is regarded as the generating page, judging the page and src as part of the unpacking routine and marking them as U. Then, again, each pair of elements in the W array is sorted in chronological order. If the dst address is marked U, the page with the src address of the pair is the sharing page and we mark the page and src as U. After the packed program execution is finished, we list all the addresses of the X array and all the src addresses of W array in chronological order. Among the listed addresses, the address in the X array, which is closest to the last U marked address, is selected as the optimal OEP candidates.

'sorting the OEP candidates' sorts OEP candidates that are from 'tracking the decoding routine,' from the most likely to the least likely. We omit detailed algorithm for lack of space.

PinDemonium [4]: DBI (Dynamic Binary Instrumentation) is a way to analyze the behavior of binary applications by inserting executable code at runtime. The PinDemonium [4] is the unpacker using Intel's DBI tool, Pin. The PinDemonium uses Scylla [14] to dump the memory when WrittenAndExecuted occurs. It relies on the heuristic method for the dumped memory to judge whether unpacking has finished or not and to find the OEP. The heuristic methods include entropy analysis, LongJump, JumpOuterSection, and Yara rule.

The entropy analysis method is to find the OEP by measuring the entropy of the memory section, where entropy calculation is very similar to [11] or [12]. The program is loaded into memory and the entropy value is calculated from how many times the byte values from 0x00 to 0xff appear in the memory section. If the difference between the entropy value measured at WrittenAndExecuted and the firstly measured entropy value (when loading the program into memory) is greater than the threshold, the most recently executed WrittenAndExecuted address is regarded as the OEP.

LongJump is the branch such that the difference in between the current EIP and the previous one is above the threshold (*e.g.*, 0x200 [4]). JumpOuterSection is the branch where the memory section name of the previous EIP is different from that of the current EIP. If the target address of JumpOuterSection or LongJump is WrittenAndExecuted, PinDemonium regard this as the OEP candidate.

The Yara rules are predefined patterns for classifying malicious code, where YARA [15] is the tool to classify/identify malware. PinDemonium uses the Yara rules to find the OEP for the dumped files. Whenever WrittenAndExecuted address appears, it dumps the memory region and then matches the Yara rules. If there is a match found, PinDemonium regards this as the end of unpacking. However, this approach is unable to find the OEP if there is no match on unpacking stub.

PolyUnpack [16]: [16] provides a behavior-based approach to automate the process of extracting hidden-code from unpack-executing malware. Unpack-executing malware is program that has an obfuscation mechanism that makes malicious parts of code look like data at compile-time rather than instructions, and then transforms them into executable code at run-time. The scheme proposed in [16] is composed of static analysis and dynamic analysis. First, it performs a static analysis on malware to extract the static code view. Then, malware is executed to perform dynamic analysis. If a new instruction sequence that was not found in the static code view appears in dynamic analysis, PolyUnpack regards this as hidden-code and automatically extracts the corresponding code chunk for further analysis. This approach does not require a clear prior knowledge of unpacking techniques applied to malware, but it has a disadvantage in that it requires considerable resources for analysis compared with the static code analysis.

3. PROPERTY OF THE OEP AND PROPOSED SCHEME

In Section 1, we defined the OEP as the point at which the packer decompresses the original executable program and passes control to it. We have learned that the OEP has the following property through experiments using the programs that are packed with various (commercial) packers.

3.1 Property of the OEP

(A) The system startup function calls after the OEP

Generally, when programs written in high-level programming language are compiled to binary code, compiler-specific system startup functions are added. Startup functions initialize resources, set up the environment, and *etc.* for preparing the execution of the main function. Comparing these system functions for the packed program and the original one, we observed that the function callings of two programs are identical. Also, all these functions are called immediate after the OEP. Hence, we can track these function calls to find the OEP.

Fig. 1 shows the execution flow in the program, which was compiled with TDM-GCC and packed by UPX [17]. After the unpacking routine is completed, a series of system startup functions are called before the main function.

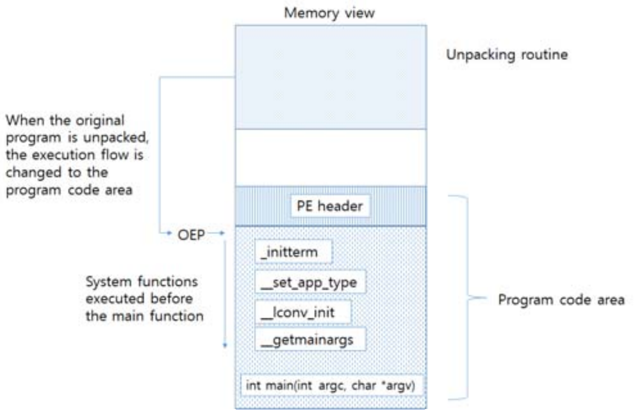

Memory view
Unpacking routine
When the original program is unpacked, the execution flow is changed to the program code area
PE header
OEP
_initterm
__set_app_type
System functions executed before the main function
__lconv_init
__getmainargs
Program code area
int main(int argc, char *argv)

Fig. 1. An example for the system startup functions calls after the OEP.

Although it is compiler/operating system dependent, system startup functions can be roughly classified as follows. First, getting system environment values (*e.g.*, process ID, current time, OS version, …), second, memory allocation and function pointer table initialization, third, setting application type, screen info, or file handles, forth, setting environment variables. Table 1 shows system startup functions for major compilers (after the OEP is executed). This data can be used to identify the compiler and to find the SE handler installation routine, which will be explained later.

There are two representative functions that handle command-line parameters of main() in Microsoft compilers: __getmainargs() and __wgetmainargs(). They invoke command-line parsing and copies the arguments to int main(int argc, char *argv[]) back through the passed pointers. The __getmainargs() has five parameters: int *_Argc, char ***_Argv, char ***_Env, int _DoWildCard, and _startupinfo *_StartInfo [18]. Among them, we focus on second argument, _Argv, which is an array of strings from command-line parsing. For example, we can track _Argv[0], which contains the full path name of the target program.

Table 1. System startup functions for major compilers.

Compiler name	SE Handler installation	ProcessID, Version, Time	MemoryAlloc, Initialize the table of function pointers	AppType, Screen Information	Handling command-line args, environment setting
MSVC 6.0	fs:[0]	GetVersion	HeapCreate RtlAllocateHeap VirtualAlloc	GetStartupInfoA GetStdHandle GetFileType SetHandleCount	GetCommandLineA GetEnvironmentStringsW FreeEnvironmentStringsW GetACP GetCPInfo MultiByteToWideChar LCMapStringW GetModuleFileNameA
MSVC 8.0	fs:[0]	GetVersion	HeapCreate RtlAllocateHeap VirtualAlloc	GetStartupInfoA GetStdHandle GetFileType SetHandleCount	GetCommandLineA GetEnvironmentStringsW WideCharToMultiByte FreeEnvironmentStringsW GetACP GetCPInfo GetStringTypeW MultiByteToWideChar LCMapStringW GetModuleFileNameA IsBadReadPtr
MSVC 2010	fs:[0]	GetSystemTimeAsFileTime GetCurrentThreadId GetCurrentProcessId QueryPerformanceCounter	_initterm_e	__set_app_type	RtlEncodePointer _CRT_RTC_INITW _controlfp_s _initterm_e __crtSetUnhandledExceptionFilter _initterm RtlDecodePointer __getmainargs
MSVC 2013	fs:[0]	GetSystemTimeAsFileTime GetCurrentThreadId GetCurrentProcessId QueryPerformanceCounter	_initterm_e _initterm	__set_app_type	RtlEncodePointer _controlfp_s _initterm_e _initterm __crtSetUnhandledException RtlDecodePointer __getmainargs
TDM-GCC	fs:[0]	GetSystemTimeAsFileTime GetCurrentProcessId GetCurrentThreadId GetTickCount QueryPerformanceCounter	_initterm	__set_app_type _initterm __lconv_init __gconv_init	_initterm __getmainargs

(B) The command-line parameters that are used by the main function

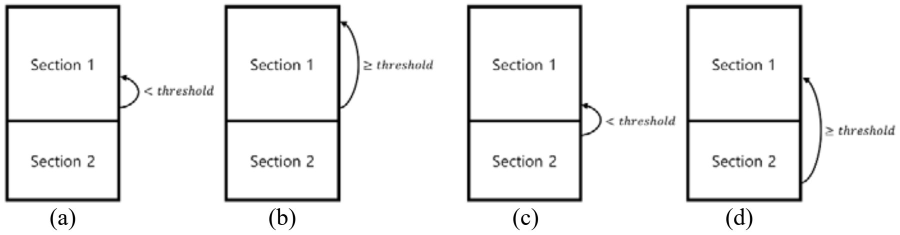
Since the main function is called after the OEP execution, we can rule out all the OEP candidates after the main() execution. To do so, we track the functions which handle command-line parameters of main() because the parameters of system startup functions for the packed file and original one are identical.

(C) OEP w.r.t. LongJump/JumpOuterSection

Recall that the OEP is one of the destination addresses at which the program execution flow has changed, *i.e.*, a branch has occurred. We identify LongJump and JumpOuterSection to find out the OEP, as follows.

LongJump is the branch where the case where the difference value between the current EIP and the previous EIP is more than threshold (0x200) [4]. JumpOuterSection is the branch with the case where the memory section name of the current EIP is different from the memory section name of the previous EIP.

Based on these two, there are four cases of branch, as shown in Fig. 2. In case (a), the difference between current EIP and previous EIP is less than the threshold value and memory section name is not changed. In case (b), the difference between current EIP and previous EIP is greater than or equal to the threshold value, so this is LongJump. Memory section name is not changed. Case (c) is not LongJump but JumpOuterSection. Case (d) is both LongJump and JumpOuterSection. From our experience, for cases (b), (c), and (d), there is a possibility that the target address of the jump is OEP, especially case (d) has the highest probability. However, for case (a), we have found no occurrence.



(a) (b) (c) (d)

Fig. 2. LongJump and JumpOuterSection.

(D) WrittenAndExecuted

In Section 2, we already defined WrittenAndExecuted as the memory address at which the instruction is executed, where the instruction was previously written. Let WrittenAndExecuted page denote the page containing this WrittenAndExecuted. Since the original code of the packed program is decrypted/decompressed by the unpacking routine, then executed, the OEP is necessarily located in the WrittenAndExecuted page set. Moreover, if we consider that the original code is fully unpacked before it gets executed, then the OEP is the first execution in a WrittenAndExecuted page that happens after the last write (end of unpacking) in this WrittenAndExecuted page set [19].

(E) Entropy of memory region containing OEP

As we already explained in Section 2, we can measure entropy values for memory regions to judge whether they are packed or not. When a general executable file is packed (encrypted or compressed), the degree of disorder in the file increases, causing the entropy value to increase [12]. Entropy measurements are performed by measuring the entropy of the memory region. To find the OEP, for each branch we can measure the entropy value of the target memory region, *i.e.*, measuring the difference between the initially measured entropy value and currently measured one. If the difference is above the threshold value, we can judge that unpacking is completed and that the target address can be regarded as the OEP candidate.

3.2 Proposed Scheme

(A) Overview of the proposed scheme

Fig. 3 shows the overview of the proposed scheme. The proposed scheme has 8 steps, which is explained as follows.

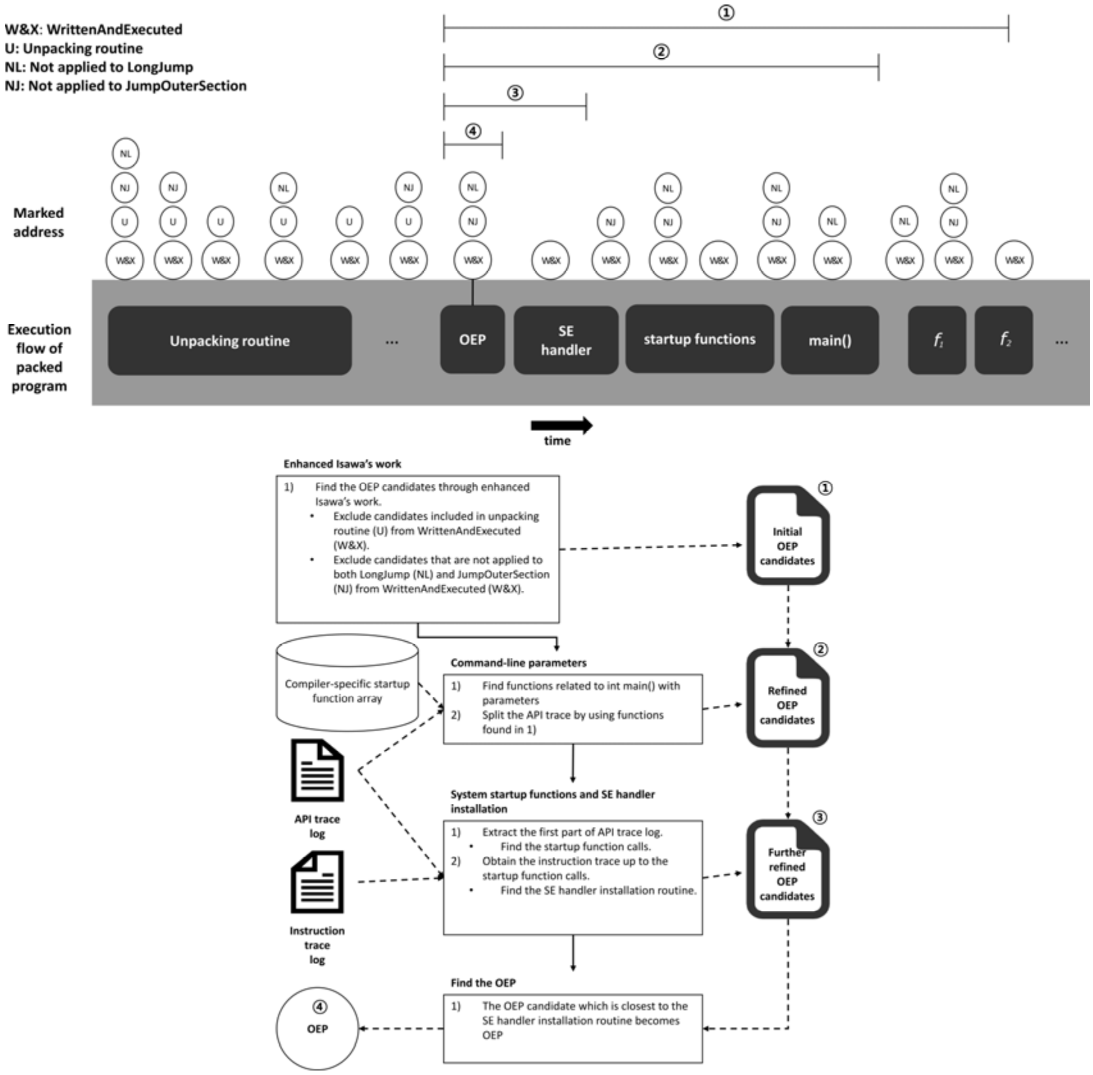

W&X: WrittenAndExecuted
U: Unpacking routine
NL: Not applied to LongJump
NJ: Not applied to JumpOuterSection
①
②
③
④
Marked address
NL
NJ
U
W&X
Execution flow of packed program
Unpacking routine
...
OEP
SE handler
startup functions
main()
f_1
f_2
...
time
Enhanced Isawa's work
1) Find the OEP candidates through enhanced Isawa's work.
• Exclude candidates included in unpacking routine (U) from WrittenAndExecuted (W&X).
• Exclude candidates that are not applied to both LongJump (NL) and JumpOuterSection (NJ) from WrittenAndExecuted (W&X).
① Initial OEP candidates
Compiler-specific startup function array
Command-line parameters
1) Find functions related to int main() with parameters
2) Split the API trace by using functions found in 1)
② Refined OEP candidates
API trace log
Instruction trace log
System startup functions and SE handler installation
1) Extract the first part of API trace log.
• Find the startup function calls.
2) Obtain the instruction trace up to the startup function calls.
• Find the SE handler installation routine.
③ Further refined OEP candidates
Find the OEP
1) The OEP candidate which is closest to the SE handler installation routine becomes OEP
④ OEP

Fig. 3. Overview of the proposed scheme.

1. First, it uses the enhanced Isawa *et al.*'s work to get the initial OEP candidates (①). The detailed procedure is described in Section 3.2.2.

2. It records the API trace log of the target program.
3. It splits the API trace log into 2 parts: the first part is from the beginning to just before the main() call and the second one is from the main() call to the end. The procedure for finding the main() call is described in Section 3.2.3.
4. From the OEP candidates set, it excludes the candidates after the main() call that was found in Step 3 (②).
5. It finds the system startup function calls (which is described in Section 3.2.4) from the first part of the log in Step 3.
6. From the OEP candidates set, it excludes the candidates after the system startup function calls found in Step 5 (③).
7. It records the instruction trace log up to the beginning of the system startup function calls of the target program.
8. It finds the address of the SE handler installation routine (which is described in Section 3.2.4) in the instruction trace log.
9. From the OEP candidates set, it excludes the candidates after the SE handler installation routine found in Step 8.
10. Among the OEP candidates filtered through the steps above, it selects the address which is closest to the address found in Step 8 and regards it as the OEP (④).

(B) Enhanced Isawa *et al.*'s work

We have improved Isawa *et al.*'s work [13] to correctly identify the OEP candidates. This method is based on the property of the OEP in Sections 3.1.3 and 3.1.4. Our method has two steps. The first step is to find the unpacking routine and the destination addresses of branch instructions. The second step is to rule out the all addresses relevant to the unpacking routine.

We track every write operation and WrittenAndExecuted to find all OEP candidates, as follows.

(a) For each memory write operation, we put (src, dst), the source and destination addresses in W.
(b) For each WrittenAndExecuted, we put the address in X.
(c) For each branch instruction which is not applied to LongJump, we mark the destination address as NL.
(d) For each branch instruction which is not applied to JumpOuterSection, we mark the destination address as NJ.

As explained in Section 3.1, the OEP is in WrittenAndExecuted address set. However, this set is too large, and we need to refine it to reduce the size by detecting the unpacking routine.

(a) We sort W in chronological order.
(b) For each pair (src, dst), we check the followings.
    i. We compare dst with the addresses in X to see if it was executed. If the dst address is executed, src address is marked as U.
    ii. If dst is marked as U, the src address is also marked as U.
(c) Repeat (b) until there is no more change.

(d) List all the elements in X in chronological order.
(e) If any of the listed addresses are marked as both NL and NJ, they are excluded from X.
(f) If any of the listed addresses are marked as U, they are also excluded from X.
(g) Now, X contains the OEP candidates.

Note that this algorithm is similar to [13], Isawa *et al.*'s work. The difference is that [13] manages memory in the unit of page whereas our algorithm processes each address independently, which provides fine grained information. Also, [13] does not track every write instruction since its algorithm is based on the page fault handling, *e.g.*, the first write on page A is tracked but subsequent writes on the same page cannot be tracked. Another point is that, unlike [13], our algorithm tracks LongJump and JumpOuterSection, which produces a smaller OEP candidates set.

(C) The method for tracking command-line parameters

This method is based on the property of OEP in Section 3.1.2. From instruction trace log, we detect system startup function calls related with the command-line parameters. For example, __getmainargs(), __wgetmainargs(), or getCommnadLineA(). In these functions, one of the arguments contains the point to the string, the full path name of the target program. If we find startup function calls through theses parameters, we split the API trace log based on the first invoke of the calls. Then, we rule out the OEP candidates after this call.

(D) Pattern-matching on the startup functions and SE handler installation routine

The goal of this step is to find the address of SE handler installation routine, which is the last write instruction on fs:[0] (explained in Table 1) in the instruction trace log. The following shows this algorithm, which is based on the property that system startup function calls of the packed programs (after the OEP) are same as those of the original program.

1. Suppose that database w.r.t. startup functions (described in Section 3.1.1) has already been built. In the database, each table contains information of startup functions for each compiler. In the table for each compiler, the record consists of 'name' to specify the name of the startup function and 'count' that is initialized to be 0.
2. Read the first part in API trace log, which was generated from the method in Section 3.2.3. For each API call in the trace, if there is a match in the database, we increase the count value by one.
3. Once the API trace log has been read, calculate the sum of all count values in each table and find the table with the largest value. We regard that the program has been compiled by this compiler.
4. Find the address of the first executed function of the identified compiler in the instruction trace log.
5. Read the instruction trace log and search for SE handler installation routine which is closest to the address found in Step 4.
6. Regard the WrittenAndExecuted address closest to the SE handler installation routine as the OEP value.

4. IMPLEMENTATION

We implemented the proposed scheme using the unpacker, PinDemonium [4], which relies on the Pin [5]. Pin is a DBI framework and supports the android, Linux, OS X, Windows operating systems and executables for IA-32, x86-64 and MIC instruction-set architectures. Pin allows a plugin tool to insert arbitrary code, which is written in C/C++, in arbitrary places in the executable. For further information, refer to [5].

For LongJump/JumpOuterSection, PinDemonium already has the relevant code and we omit the implementation. In Section 4.1, we describe implementation of enhanced Isawa *et al.*'s work. In Sections 4.2 and 4.3, we explain implementation of finding SE handler installation routine and tracking command-line parameters, respectively.

4.1 Implementation of Enhanced Isawa *et al.*'s Work

This section describes the implementation of enhanced Isawa *et al.*'s work in Section 3.2.2. In PinDemonium, the function to find the OEP is IsCurrentInOEP(). First, we prepared five markers: W_src, W_dst, X, NL, and NJ. We used Pin's INS_IsMemoryWrite() function to find out whether the current instruction is memory write or not. For each write, the address of instruction is marked as W_src and target address of memory write is marked as W_dst. Then, we update X that contains the address for WrittenAndExecuted. Also, we used PinDemonium's LongJumpHeuristic() and JumpOuterSection() function to find out whether the target address of branch instruction can be OEP candidate or not. For each branch instruction, if the difference between the previous EIP and the current EIP is less than threshold, the target address of the branch instruction is marked as NL, and if the memory section name of the previous EIP and the current EIP is same, the target address of the branch instruction is marked as NJ.

4.2 Implementation of Finding SE Handler Installation Routine

First, we have prepared information for system startup functions for major compilers, which is in Table 1 of Section 3.1.1. Then, we have implemented instruction trace routine and API trace routine. We used Pin's RTN_FindNameByAddress() for implementing API trace routine and INS_InsertCall() for implementing instruction trace routine. For pattern-matching of the function call, we compare only the function name and ignore parameter values.

For finding SE handler installation routine, we should track the write operation on fs:[0]. To do so, we used Pin's INS_IsMemoryWrite() for tracking the write operation and we have recorded changes in the register values in execution. From this and instruction trace logs, we can find the instruction to write on fs:[0]. Because there are many cases for this during execution, we carefully select such operation between the unpacking routine and the main function call.

4.3 Implementation of Tracking Command-Line Parameters

This section describes the implementation of tracking command-line parameters which is explained in Section 3.2.3. We used the Pin to hook up read/write on ESP-10h when __getmainargs() function is called because it contains the pointer to the full path name of the target program. Generally, __getmainargs() is called several times because it reads each byte of the path at a time. In this implementation, we copy the path string to another array and then find out whether it is the same as the name of the target program.

If we use getCommnadLineA() instead of __getmainargs(), this function copies the address of the program execution path from the data segment to the EAX register. That means, we should hook up EAX and follow this address. This function is also called multiple times because it accesses each byte of the full path name at a time. We hook this, store it in the array and find whether there is the string that is the same as the target program name. If we have found the target program name, we can split the API trace log.

5. EXPERIMENTAL RESULTS

In Section 5.1 we briefly explain environments for our experiments. Then, experimental results are shown in Section 5.2.

5.1 Environments

In the experiment we chose the most widely used (commercial) 16 packers: UPX 1.02, PECompact, ASPack 2.0, ASProtect 1.2, Obsidium 1.3, Themida 2.3, VMProtect 2.0, WWPack, Packman, petite, MEW, Enigma 4.03, mpress, nspack, yoda 1.3, and Safengine 3.09. For each packer, we used default option settings.

The test program used in our experiment was compiled with TDM-GCC, where it displays a simple text to the console window. The experiment was conducted under the following environments. OS: Windows 7 SP1, CPU: Intel i7 3.4 GHz and Memory: 14 GB DDR3 DRAM. The version of Pin was 2.14, and Pin's plug-in was compiled with Visual Studio 2010. We have implemented our scheme using the source code of PinDemonium [4], whose release date is July 2016.

5.2 Experimental Results

Compared unpacking schemes are as follows: PinDemonium [4], Isawa *et al.*'s scheme [13], Jeong *et al.*'s scheme [11], PEiD, QuickUnpack2.2, and AbstersiverA. For [11] and [13], since our experimental environments are different from them, we have also implemented [11, 13] on PinDemonium. Table 2 shows the experimental results. In this table, column A means the number of the OEP candidates that are produced from each scheme and column B indicates correctness, *i.e.*, whether the OEP is in the OEP candidates or not. In the bottom row, cell A has the average value of the OEP candidates for each scheme while cell B has the detection rate, which is calculated by the number of packers for which the scheme has successfully found the OEP, divided by the number of all packers.

As for correctness (column B), for all packers PEiD cannot find the correct OEP. Quickunpack and AbstersiverA successfully find the OEP only for some packers. Isawa *et al.*'s work successfully finds the OEP for 13 packers among 16, which means the detection rate is 81.25%. For PinDemonium and the proposed scheme, they find the OEP for all packers except for yoda and Safengine where the detection rate is the highest, 87.5%. This is because the yoda and Safengine have anti-reverse engineering techniques for detecting the Pin and they abort execution.

Table 2. Experimental results.

Packer	Pin-Demonium		[13]		[11]		Our Implementation of [11]		PEiD		Quick Unpack 2.2		AbstersiverA (ASPack Unpacker)		Proposed scheme	
	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B
UPX 1.02	2	O	2	O	1	▲	many	O	1	X	1	O	0	X	1	O
PECompact	5	O	2	O	–	n/a	many	O	1	X	1	O	0	X	1	O
ASPack 2.0	13	O	2	O	1	O	many	O	1	X	1	O	1	O	1	O
ASProtect 1.2	92	O	16	O	–	n/a	many	O	1	X	1	X	0	X	1	O
Obsidium 1.3	68	O	31	X	–	n/a	many	X	1	X	1	X	0	X	1	O
Themida 2.3	592	O	26	O	–	n/a	many	X	1	X	1	X	0	X	1	O
VMProtect 2.0	4	O	2	O	–	n/a	many	O	1	X	1	X	0	X	1	O
WWPack	4	O	3	O	–	n/a	many	O	1	X	1	X	0	X	1	O
Packman	2	O	2	O	–	n/a	many	O	1	X	1	O	0	X	1	O
petite	11	O	5	O	–	n/a	many	O	1	X	1	X	0	X	1	O
MEW	3	O	3	O	–	n/a	many	O	1	X	1	O	0	X	1	O
Enigma 4.03	10	O	9	O	–	n/a	many	X	1	X	1	X	0	X	1	O
mpress	4	O	3	O	1	▲	many	O	1	X	1	X	0	X	1	O
nspack	4	O	2	O	1	▲	many	O	1	X	1	X	0	X	1	O
yoda 1.3	0	X	0	X	–	n/a	0	X	1	X	1	X	0	X	0	X
Safeingine 3.09	0	X	0	X	–	n/a	0	X	1	X	1	X	–	n/a	0	X
Average value/ Detection rate (%)	50.87	87.5	6.75	81.25	0.25	–	many	68.75	1	0	1	31.25	0.06	6.25	0.875	87.5

A: the number of OEP candidates, B: correctness, O: have found the OEP,
X: cannot find, ▲: have found the OEP only for some cases)

As for the number of the OEP candidates, (our implementation of) [11] has the largest number of OEP candidates (for most of cases, more than 1000). Also, PinDemonium and Isawa *et al.*'s work output a relatively large number of OEP candidates. For all packers, the proposed scheme outputs only 1 OEP candidate. Nonetheless, for 14 of 16 packers the proposed scheme correctly answers the OEP value.

In summary, some previous unpacking tools/schemes with high detection rates output many OEP candidates while the others with less OEP candidates have low accuracy. Compared with the previous schemes, the proposed scheme pinpoints the OEP with the highest level of accuracy.

6. CONCLUSIONS

In this paper, we presented a new efficient scheme for finding the OEP for diverse packers. The demerit of the previous work is that either they cannot find the OEP correctly or they produce too many OEP candidates for packers. For latter case, analyst should manually analyze further to find the correct OEP, which takes a considerable amount of time.

To find the OEP, our scheme first uses enhanced Isawa *et al.*'s work to finds the OEP candidates. Then, we hook the parameters used in the main function to determine whether execution has passed the OEP or not, by which we can minimize the candidate set. Then, we match the compiler-specific startup function calls for refining the candidate set and find the SE handler installation routine to select the OEP value from the set. We conducted experiments on 16 (commercial) packers. Experimental results show that compared with the previous schemes including PinDemonium and Isawa *et al.*'s work, our scheme pinpoints the correct OEP whereas others produce many OEP candidates. Nonetheless, accuracy is the highest (up to 14 times higher than the previous work).

REFERENCES

1. T. Brosch and M. Morgenstern, "Runtime packers: The hidden problem?" *BlackHat*, https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf, 2006.
2. ALDEID, PEiD, http://www.peid.info.
3. Hex-Rays, Universal PE Unpacker, https://www.hex-rays.com/products/ida/support/tutorials/unpack_pe/index.shtml.
4. S. D'Alessio and S. Mariani, "PinDemonium: a DBI-based generic unpacker for Windows executables," *BlackHat*, 2016, pp. 1-56.
5. Intel, "Pin user manual," https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/.
6. S. Bardin, R. David, and J. Y. Marion, "Backward-bounded DSE: Targeting infeasibility questions on obfuscated codes," in *Proceedings of IEEE Symposium on Security and Privacy*, 2017, pp. 633-651.
7. T. Blazytko, M. Contag, M. Aschermann, and T. Holz, "Syntia: Synthesizing the semantics of obfuscated code," in *Proceedings of USENIX Security Symposium*, 2017, pp. 643-659.
8. R. David, S. Bardin, T. D. Ta, J. Feist, L. Mounier, M. L. Potet, and J. Y. Marion, "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis," in *Proceedings of Software Analysis*, *Evolution*, *and Reengineering*, 2016, pp. 653-656.
9. X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 24-35.
10. S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code," in *Proceedings of Network and Distributed System Security Symposium*, 2016, pp. 21-24.
11. G. Jeong, E. Choo, J. Lee, M. Bat-Erdene, and H. Lee, "Generic unpacking using entropy analysis," in *Proceedings of International Conference on Malicious and Unwanted Software*, 2010, pp. 98-105.
12. R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security and Privacy Magazine*, Vol. 5, 2007, pp. 40-45.

13. R. Isawa, D. Inous, and K. Nakao, "An original entry point detection method with candidate-sorting for more effective generic unpacking," *IEICE Transactions on Information and Systems*, Vol. E98-D, 2015, pp. 883-893.
14. NtQuery, "Scylla – x64/x86 imports reconstruction," https://github.com/NtQuery/Scylla.
15. Virustotal, "YARA: The pattern matching swiss knife for malware researchers (and everyone else)," https://virustotal.github.io/yara/.
16. P. Royal, M. Halpin, and D. Dagon, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *Proceedings of the 22nd Annual Computer Security Applications Conference*, 2006, pp. 289-300.
17. UPX Team, "UPX –the ultimate packer for eXecutables," https://upx.github.io/.
18. Microsoft, "__getmainargs, __wgetmainargs," MSDN (Microsoft Developer Network), https://msdn.microsoft.com/ko-kr/library/ff770599.aspx.
19. J. Lenoir, "Implementing your own generic unpacker," in *Proceedings of HITB GSEC*, 2015, pp. 1-19.
20. N. Talekar, "Practical reversing II – unpacking EXE," https://repo.zenk-security.com Reversing%20.%20cracking/Reversing%20Malware%20Analysis%20Training/Presentations/Reversing%20&%20Malware%20Analysis%20Training%20Part%207%20-%20Unpacking%20UPX.pdf.

Gyeong-Min Kim received the B.E. degree in Computer Science from ChungAng University, South Korea, in 2015. He currently is an M.S. student in Computer Science from Hanyang University, Seoul, Korea. His main research interests include malware analysis, software security, and binary code analysis.

Juhyun Park received the B.E. degree in Computer Science from Sangmyung University, South Korea, in 2018. He currently is an M.S. student in Computer Science from Hanyang University, Seoul, Korea. His main research interests include malware analysis, software security, and binary code analysis.

Yun-Hwan Jang received the B.E. degree in Computer Science from Hongik University, South Korea, in 2015. He currently is an M.S. student in Information Security from Hanyang University, Seoul, Korea. His main research interests include android security, web security, malware analysis, and network security.

Yongsu Park received the B.E. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 1996. He received the M.E. degree and the Ph.D. degree in Computer Engineering from Seoul National University in 1998 and 2003, respectively. He is currently a Professor in the Department of Computer Science at Hanyang University, Seoul, Korea. His main research interests include computer system security, malware analysis, operating system security, and network security.