Automated Dynamic Analysis Scheme for Unpacking Malware

Minho Kim[1], Gwangyeol Lee[1], Haehyun Cho[1]*, and Jeong Hyun Yi[1]
[1]School of Software, Soongsil University, Seoul, 06978, Korea
mhkim37@soongsil.ac.kr, gwangyeal@gmail.com, haehyun@ssu.ac.kr, jhyi@ssu.ac.kr

Abstract

Malware is causing private information leaks and financial damages to numerous companies and users. To deal with increasing malware, malware analysts have been making tremendous efforts. Similarly, malware authors also use many techniques to evade detection from various automated analyses. Packing and obfuscation techniques are representative techniques that make analysis difficult. It is challenging to analyze malware that uses obfuscation or packing techniques. However, unfortunately, existing unpackers have a limitation that they cannot properly handle obfuscation techniques such as API obfuscation and original entry point (OEP) obfuscation. As a result, we are relying on manual analysis for analyzing malware equipped with such anti-analysis techniques.

In this paper, we first analyzed the OEP and API obfuscation techniques. We, then, design and implement a deobfuscation system, named Pinicorn, that deobfucate OEP and API obfuscation techniques, overcoming the limitation of previous approaches. Pinicorn analyzes and addresses the trampoline code used for the OEP and API obfuscation. Through the case study, we demonstrated that Pinicorn can be used to effectively deobfuscate each packer's OEP and API obfuscation techniques.

1 Introduction

In 2020, FireEye's M-Trends Report found that 31.28% of malware uses obfuscation techniques to evade detection [8]. Obfuscation techniques such the string encryption and API obfuscation make binaries difficult to understand for preventing reverse engineering [3]. In addition, attackers also employ packers that compress code and data and decompress them when a program executes, by which we cannot simply use any static analysis approach. Therefore, researchers have been working to resolve the issue of using packing and obfuscation techniques in malware [10, 19, 4, 13].

One of the main challenge for unpacking packed malware is to find the program's original entry point (OEP) [6, 9, 10]. Packers makes the OEP difficult to find by obfuscating it because unpackers firstly find the OEP to dump an unpacked executable file from memory [4]. In general, unpackers assume that a packed program is totally decompressed and loaded into the memory when they detected the OEP. Therefore, to detect the OEP, many studies have noted that a packed program executes a lot of memory access operations to decompress and load the compressed section [10, 1, 6]. Then, they monitor the execution of generated instructions to find the OEP. For instance, conventional approaches detected the OEP when a program executes a runtime-generated instruction within a compressed section [9, 10]. However, unfortunately, a recent study has reported that 9.5% of packed malware samples use the OEP obfuscation [4]. The OEP obfuscation technique used in packers such as Themida, VMProtect, and Enigma restores a packer's trampoline code instead of the original code into the OEP [16, 17, 21]. The trampoline code is to prevent dumping an executable files by tricking a control flow in the OEP of packed programs. when the OEP executes of a packed program, it first jumps to the

*Corresponding author: School of Software, Soongsil University, Seoul, 06978, Republic of Korea.

trampoline code because of the code inserted by packers. Next, the trampoline code jumps to the OEP of the program. Therefore, if we simply dump an executable file of which the OEP is obfuscated from the memory, the dumped binary cannot execute normally, causing a runtime error.

Another technical challenge for unpacking malware is to restore the program's import table which contains information APIs used in the program [2, 12]. In general, a portable executable (PE) file cannot call APIs of external libraries without using the import table [4]. Many packers such as Themida use the API obfuscation that basically removes the information stored in the import table for making reverse engineering very difficult [16]. During the runtime, similar to the OEP obfuscation, the API obfuscation technique stores the address of the trampoline code, not the actual memory addresses of APIs, in the code or import address table (IAT) for calling APIs at runtime [4, 19]. Therefore, during the unpacking process, if an unpacker dump an executable file from the memory without correctly restoring the import table, the unpacked program will be unexecutable. We can classify the trampoline code into the argument-sensitive one and argument-insensitive one. The latest research work, API-Xray, focuses on handling only the argument-insensitive trampoline code [4]. They categorized the API obfuscation into five types and propose a deobfuscation approach to rebuild the import table. However, handling the argument-sensitive trampoline code still remains a difficult problem for extracting executable files from packed malware.

In this paper, we aim to automatically deobfuscate the OEP and API obfuscation techniques. Especially, we focus on deobfuscate the two obfuscation techniques using the trampoline code. To this end, we propose an dynamic approach, named Pinicorn, for deobfuscating such obfuscation techniques and unpacking packed programs. Pinicorn first detects all cadidates of the OEP and collects the context information when a candidates of the OEP executes to emulate and analyze the trampoline code accurately by using Intel Pin, a dynamic binary instrumentation framework. At the same time, Pinicorn bypasses the anti-analysis techniques of packers while executing packed malware through Intel Pin [14]. Next, we analyze the obfuscated trampoline code, by emulating it with the recorded context information via Unicorn engine [15]. To be specific, we find code that branches from the OEP to the trampoline code and the trampoline code used for API calls in the IAT or text section. First, Pinicorn monitors the execution flow from the OEP. A packed program with the OEP obfuscation jumps to the trampoline code from the OEP and returns to the original code. Pinicorn record the location of the original code and then patches the OEP to branch to the next original code instead of the trampoline code. Next, Pinicorn analyzes the trampoline code used by the API obfuscation technique. In particular, Pinicorn monitors how a basic block where the argument-sensitive trampoline code is located executes. This is because the basic block calls different APIs depending on data passed from the other code blocks to the basic block. Therefore, Pinicorn analyzes the basic block that has the trampoline code to identify all the obfuscated APIs by emulating it. Finally, we dump an executable file from the memory, patch the binary based on the analysis results of the OEP and API obfuscation techniques, and rebuild the unpacked program.

Contributions. This paper makes the following contributions:

- We analzye state-of-the-art packers to investigate how they protect programs from reverse engineering.

- We propose an automated dynamic analysis method for deobfuscating and unpacking packed programs.

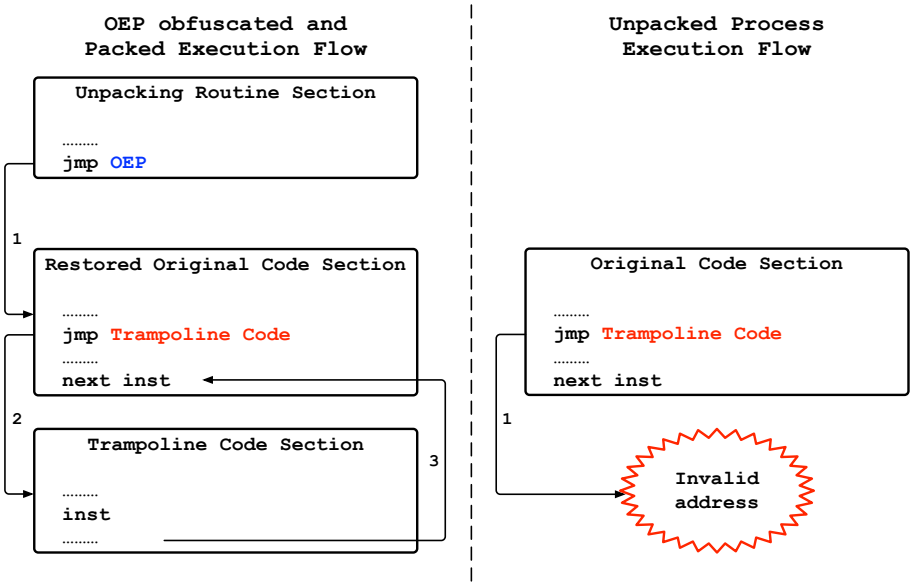


Figure 1: The control flow of the OEP obfuscated program.

- We demonstrate that our proposed "Emulation-based Obfuscated Code Execution" is a practical approach for identifying and analyzing the trampoline codes in packed programs.

2 Background and Related work

In this section, we introduce two obfuscation techniques used in packers, the terms used in the paper, and previously proposed approaches for the deobfuscation.

2.1 Packers

In general, a packer compresses the code and data of an executable file to prevent tampering and reverse engineering. The packed code and data is decompressed and loaded into a virtual memory space of a process during runtime. The restored code and data performed the same operation as the original executable file. In the early days, packers only provided the compression and decompression functions [20]. However, many packers started adding various anti-analysis techniques to make it more challenging to analyze packed programs. For example, packers provide anti-debugging and virtual environment detection techniques [16, 17, 7, 22, 18]. In addition, packer also provides sophisticated anti-analysis techniques such as the code virtualization, data obfuscation, and API obfuscation [16, 17]. Because packers provide such various anti-analysis techniques, malware authors frequently use them to evade detection. In order to effectively analyze packed malware, security researchers have been conducting various research to unpack the packed program. In this work, we focus on two obfuscation techniques: the OEP obfuscation and API obfuscation. The two obfuscation techniques have not received much attention from security researchers.

2.2 The OEP Obfuscation

A general OEP obfuscation process is as follows: In the compression function, a packer adds sections called unpacking routines. Unpacking routines include code that restores compressed code and data during runtime and various protection mechanisms. The packer then pollutes changes the program's entry point in the program header to the unpacking routine section. Finally, the unpacking routine restores the OEP of the program and jump to it as the last step of the restoration process.

The main challenge of the unpacking process is finding the OEP of packed programs. The most of previous approaches monitored how the unpacking routine restores the original code and data [10, 9]. Because the restoring process executes a lot of memory write operations, the unpackers detect it as the OEP when a process executes a newly generated instruction [9]. Another approach is to analyze the execution patterns of instructions during runtime. PinDemonium [6] is an open-source unpacker implemented based on Intel Pin. It detects the OEP when the memory address difference between the previous and current instructions is larger than the threshold (0x200). In addition, if the section of the previous and the current instructions is different, PinDemonium detects the current instruction is the OEP. Kim et al. used the characteristic that the system call sequence and parameters, which are used to call the main function, of the packed program and the original program are the same after executing an instruction at the OEP [11]. Their approach classifies the most recent "written and executed" instruction as the OEP when a known system call sequence and parameters executed.

Many packers implement the OEP obfuscation that can defeat the previous OEP detection approaches. When the unpacker detects the OEP, it dumps the entire process and creates an unpacked binary. Therefore, recent packers' OEP obfuscation use "trampoline code" to defeat the OEP detection approaches as follow. The OEP obfuscated program branches to the trampoline code after restoring the original code as shown in Figure 1. Then, the process executes the trampoline code and then returns to the next instruction. Hence, unpackers that find the OEP by using the memory addresses or sections of instructions executed cannot successfully dump the original code because it has jump instructions to the trampoline code. Consequently, as shown in Figure 1, when the unpacked program runs, the OEP instruction jumps to an invalid address, causing runtime errors. To the best of our knowledge, there are no unpackers that can analyze and restore the original code against such OEP obfuscation technique.

2.3 API Obfuscation Technique

A typical API obfuscation technique used in packers works as follows: First, a packer parses the original import table of a program to find virtual memory addresses to call APIs used in the program, and stores the information somewhere in the program after packing it. Then, the packer deletes the information stored in the import table of the program. The unpacking routine writes the virtual memory addresses to the import table instead of the operating system's loader when the program executes. The addresses that the packer writes in the import table can be either actual Windows APIs' addresses or the trampoline code address which jumps to APIs.

Previous approaches have recognized the importance of reconstructing the import table to deobfuscate the API obfuscation [12, 5]. To this end, they monitored the virtual memory addresses written by the unpacking routine to the import table and restored them. However, there are packers that write the trampoline code address of obfuscated APIs in the import table, and thus, the previous approach cannot restore the original import table [6]. As shown in Figure 2, the program that has the import table of which virtual memory addresses of APIs are incorrect

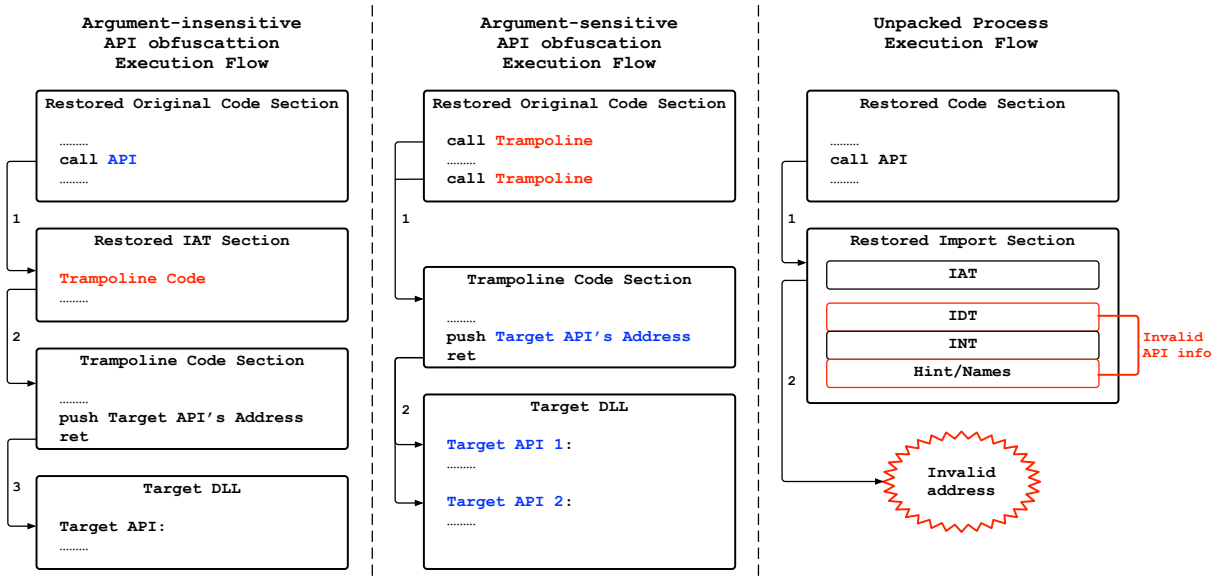


Figure 2: The control flow of the API obfuscated program.

cannot execute normally. To handle such API obfuscation technique, we first need to analyze how the trampoline code works. However, the trampoline code is difficult to figure out how it operates without using the dynamic analysis. We classified trampoline codes into two types: (1) argument-insensitive and (2) argument-sensitive trampoline code. Most packers use argument-insensitive trampoline code for implementing the API obfuscation [16, 17, 21, 7]. Also, packers store their trampoline code into the unpacking routine or into a dynamically allocated memory region.

A recently proposed deobfuscation approach tried to solve the problem of using the argument-insensitive trampoline code used in the API obfuscation [5, 4]. API-Xray classified the API obfuscation into five types (IAT Redirection via SEH, Anti-debugging Routine, ROP Redirection, Stolen Code, and Rewrite API Callsite). They analyzed dumped executable files from programs packed by various packers to identify the trampoline code. API-Xray executed and analyzed all trampoline code contained in the dumped files by using the hardware-assisted API micro execution. Based on the information obtained from the dynamic analysis, they identified all the obfuscated APIs and reconstructed the original import table. However, they limited the problem scope to only the argument-insensitive trampoline code.

Some packers such as ASProtect [18] use the argument-sensitive trampoline code for API obfuscation. The argument-sensitive trampoline code jumps to multiple APIs based on the context information of a process. Specifically, by manually analyzing ASProtect, we found that stack-related instructions within the same basic block affect trampoline code's behavior. Therefore, in order to analyze the argument-sensitive trampoline code, we should consider the program's context when it executes the trampoline code. We demonstrate how we analyze the argument-sensitive trampoline code by using the information Section 4.

3 Overview

For unpacking packed (and obfuscated) programs, in this work, our goal is to design and implement an automated frame work, Pinicorn, that (1) detects and patches the OEP instructions, and (2) analyzes the trampoline code used in the API obfuscation for restoring the original import table. Especially, we focus on unpacking packed Windows programs and malware. In this section, we present the goal of this work and overview how we achieve the goal.

3.1 Goal

1. **Finding and patching the OEP instructions:** Existing unpackers generate unpacked programs through dumping a program from its virtual memory space after detecting the OEP [9, 6]. In this process, if an unpacker does not properly deobfuscate the OEP obfuscation, the unpacked program will access an invalid memory area because of the trampoline code inserted by packers. Therefore, we propose a new approach for deobfucasting such OEP protection techniques, overcoming the limitation of the previous approaches.

2. **Detecting API calls from the trampoline code:** Packers use the trampoline code to prevent import table reconstruction. An unpacked program with an incorrect import table cause a runtime error when calling an API. The previously proposed approaches identified hidden APIs by monitoring memory write operations and executing the trampoline code inserted in a packed program [19, 4]. However, such approaches have a limitation that can analyze only the argument-insensitive trampoline code. In this work, to tackle the limitation, we aim to analyze the hidden APIs called by the argument-sensitive trampoline code. To this end, we dynamically analyze packed programs from a basic block that contains stack-related instructions affecting the execution of the trampoline code.

3.2 Our Approach

The Pinicorn first parses a program's sections and DLL information. Pinicorn uses the parsed information for monitoring memory write operations and finding the trampoline code. Next, Pinicorn detects and bypasses the packer's anti-analysis techniques such as anti-debugging and anti-vm techniques. Without bypassing the anti-analysis techniques, we cannot execute and analyze how the packed program restores the original code and data with dynamic analyzers. We discuss anti-analysis techniques used by packers in detail in Section 4.1.2.

For obfuscating the OEP, packers modifies the program's control flow to branch from the OEP to the trampoline code. Hence, obfuscated programs jumps directly to the trampoline code when it enters the OEP. This is because unpackers are simply dumping code from the virtual memory, when they detect newly generated instructions written in memory. Therefore, if unpackers do not find and remove the trampoline code, unpacked programs will terminate with a runtime error because of using an invalid memory address. To address this problem, Pinicorn monitors the program's control flow to detect the OEP and instructions that jumps to the trampoline code. After finding the OEP and such instructions, we patch the instructions that jumps to the trampoline code with instructions that jumps to the instructions in the OEP. We discuss this deobfuscation mechanism in detail in Section 4.2.1.

On the other hand, the API obfuscation techniques write the trampoline code' address in the import table, not the Windows API address. Therefore, employing the trampoline code neutralize the previous approaches which rebuild the import table by simply monitoring memory addresses used when packed programs call APIs. We explore all the trampoline code in packed

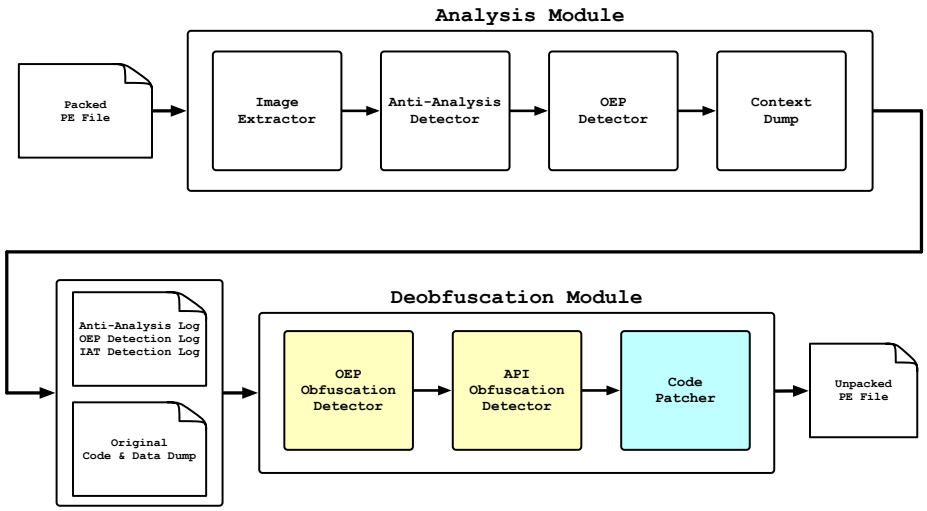


Figure 3: The overview of Pinicorn.

programs and discover the actual addresses of APIs obfuscated by the trampoline code. We discuss our deobfuscation mechanism in detail in Section 4.2.2.

In summary, Pinicorn have the following advantage over the previous work: it can reliably deobfuscates the OEP obfuscation methods so that unpacked programs can run without causing runtime errors; and it addresses the issue which can be occurred by incorrectly rebuilt import tables.

4 Design

In this section, we describe the design of Pinicorn. Pinicorn consists of two modules: (1) DBI-based program execution module (Section 4.1) and (2) Emulation-based obfuscated Code execution module (Section 4.2). Figure 3 illustrates the design of Pinicorn. First, Pinicorn extract all context information needed for deobfuscating the OEP obfuscation and API obfuscation techniques. Next, it find all the trampoline code inserted in a packed program. Then, it executes the trampoline code independently to figure out what it obfuscates. Finally, Pinicorn deobfuscate and unpack the program, reconstructing the program by using the identified information. The reconstructed program contains the deobfuscated OEP and import table.

4.1 DBI-based Program Execution

Pinicorn dynamically executes the input program to obtain information for deobfuscating it. The DBI-based program execution module works based on Intel Pin to analyze the target program dynamically. It performs the following operations while executing the target program.

4.1.1 Image Extraction

Pinicorn extracts the following information that are loaded into the virtual memory space by the target program.

1) **Process and Thread Information.** Pinicorn records information regarding the context of the process and threads to completely restore the context information including the

Table 1: A summary of information that the anti-analysis techniques check.

Packer	Version	Anti-Debug	Anti-VM
Themida	2.4.5	PEB (BeingDebugged)	Register Key Exists
Themida	2.4.5	NtQueryInformationProcess	Register Key Value Checks
Themida	2.4.5	-	IN Instruction
Themida	3.0.7	PEB (NtGlobalFlag)	Register Key Exists
Themida	3.0.7	NtQueryInformationProcess	Register Key Value Checks
Themida	3.0.7	NtUserGetForegroundWindows	GetSystemFirmwareTable
Enigma	5.2	PEB (BeingDebugged)	IN Instruction
Enigma	5.2	-	CPUID Instruction
Enigma	7.0	PEB (BeingDebugged)	IN Instruction
Enigma	7.0	PEB (HeapFlags)	-
Enigma	7.0	NtQueryInformationProcess	-
VMProtect	3.0.9, 3.4.0	PEB (BeingDebugged)	CPUID Instruction
VMProtect	3.0.9, 3.4.0	NtQueryInformationProcess	-
VMProtect	3.0.9, 3.4.0	NtClose	-
VMProtect	3.0.9, 3.4.0	Single-Step Exception	-
Obsidium	1.6.7	PEB (BeingDebugged)	IN Instruction
Obsidium	1.6.7	NtQueryInformationProcess	GetSystemFirmwareTable
Obsidium	1.6.7	NtClose	-
ASProtect	2.7.8	PEB (BeingDebugged)	-
Yodas's Protector	1.03.3	PEB (BeingDebugged)	-
Yodas's Protector	1.03.3	NtQueryInformationProcess	-

process environment block (PEB), thread environment block (TEB), registers, data stored in the stack, and each thread's stack location of them for emulating the process later.

2) **Program Sections.** Pinicorn records the program sections' names, addresses, and permissions. In addition, Pinicorn checks if each section's names and permissions and monitors change while executing the target program.

3) **DLL Information.** Pinicorn parses and records each DLL's API names and addresses. Pinicorn uses this information to detect obfuscated API and rebuild the import table.

4.1.2 Anti-Analysis Detector

Packers provide evasion techniques such as anti-debugging and virtual machine detection techniques. Without bypassing them, we cannot analyze packed programs with tools using debugging mechanisms and on a virtual machine. Pinicorn employs Intel PIN and we use Pinicorn on a virtual machine to analyze malware. Therefore, Pinicorn should be able to detect and bypass anti-debugging and virtual machine detection techniques for executing and analyzing target programs. To this end, we first analyzed anti-analysis techniques used in packers. In Table 1, we summarized the analysis results of such anti-analysis techniques. By using the information, we implemented the anti-analysis detector that dynamically detects instructions and bypasses the anti-analysis techniques. To be specific, the anti-analysis detector meet known patterns of instructions that check the use of a debugger or virtual machine while executing the target program, it dynamically instruments the instructions so that the target program cannot be aware of the analysis environment.

4.1.3 Finding Candidates of the OEP

PinDemonium detects OEP with four strategies (Long Jump, Jump Outer Section, Entropy Analysis, Yara Rule) [6]. Among them, Pinicorn uses Long Jump and Jump Outer Section. To detect OEP, Long Jump detect OEP if the difference between the previous EIP and the current EIP is more significant than the threshold (0x200). Jump Outer Section detects when the previous EIP and the current EIP belong to different sections. More specifically, the previous EIP must belong to the section added during the packing process, and the current EIP must belong to the restored section. Also, the current EIP instruction must be a "Written-then-Execute" instruction for both approaches. In both ways, Pinicorn finds a list of one or more OEP. It is then used to emulate all OEP lists during the OEP deobfuscation process.

4.1.4 Context Dump

The deobfuscation process needs the context of the program to emulate a packed program. In this process, we dump the CPU context, the program's code, data, stack, and dynamically allocated memory regions when Pinicorn detected a candidate of the OEP. Specifically, dumping dynamically allocated memory regions is really important because many packers use dynamically allocated memory regions for loading their trampoline code. We use the VirtualAlloc API to monitor dynamically allocated memory regions and dump them.

4.2 Emulating the Execution of Obfuscated Code

We designed the emulation-based obfuscated code execution module based on Unicorn engine [15]. The Unicorn engine is a light-weight CPU emulator framework, based on QEMU, with recorded context information of the target program. To emulate the execution of obfuscated code, Pinicorn uses the information observed through the DBI-based program execution module. By emulating the execution, we dynamically analyze the OEP obfuscation and API obfuscation techniques used in the target program to deobfuscate them.

4.2.1 OEP Obfuscation Detector

Pinicorn analyzes using the context information recorded when we detected candidates of the OEP. When the packer obfuscates the OEP of the target program, it patches the OEP instructions to forcibly jump to the trampoline code. In the trampoline code, the packed program jumps to an instruction of the original code so that the program can execute normally. In the other words, the OEP obfuscation prevents executions of executable files directly dumped from the memory by inserting instructions that jumps to the trampoline code in the OEP. Also, because the trampoline code is loaded into a dynamically allocated memory region, existing unpackers cannot successfully restore the original program as far as they do not know how the obfuscated OEP instructions execute. Therefore, to deobfuscate the OEP obfuscation technique, we should find all memory addresses returned from the trampoline code. To do this, Pinicorn searches for the trampoline code's presence when we sequentially run the OEP list. We classify the program as OEP obfuscated if Pinicorn found the trampoline code. Then, we emulate the trampoline code to monitor the control flow. The trampoline code in OEP returns to the original code when all executions are finished. Finally, when the control flow changes from the trampoline code to the original code, the OEP obfuscation detector captures the memory address and the captured memory addresses are used to patch the OEP later.

Table 2: A example of ASProtect's basic block with argument-sensitive trampoline code.

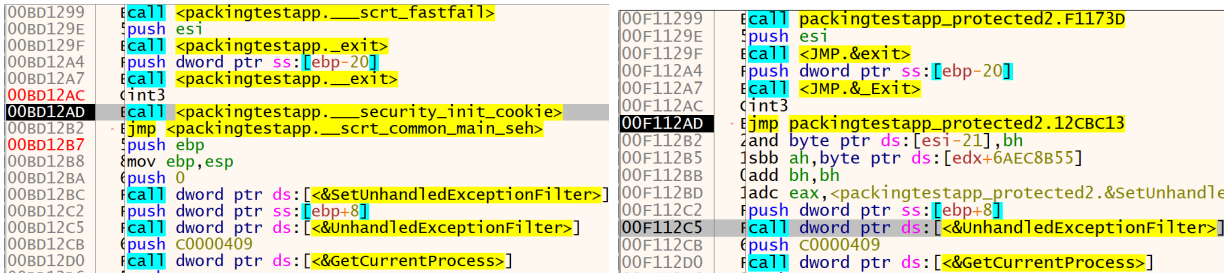
API	GetSystemTimeAsFileTime	QueryPerformanceCounter	IsProcessorFeaturePresent	IsDebuggerPresent
Basic Block	and dword ptr ss:[ebp-C], 0 lea eax, dword ptr ss:[ebp-C] and dword ptr ss:[ebp-8], 0 push eax **call 3240000**	xor dword ptr ss:[ebp-4], eax lea eax, dword ptr ss:[ebp-14] push eax **call 3240000**	push ebx push 17 **call 3240000**	mov dword ptr ss:[ebp-58], 40000015 mov dword ptr ss:[ebp-54], 1 mov dword ptr ss:[ebp-4C], eax **call 3240000**

4.2.2 API Obfuscation Detector

The API obfuscation techniques injects the trampoline code to hide which API is used in a target program. Pinicorn runs each trampoline code to figure out what it means. The trampoline code used in the API obfuscation techniques can be implemented in various forms. In general, there are two types of the trampoline code: (1) argument-sensitive trampoline code and (2) argument-insensitive trampoline code. The argument-sensitive trampoline code calls all obfuscated APIs by using a single code block based on the context information. The execution result of the argument-sensitive trampoline code depends on the parameters passed by a process before executing the trampoline code. On the other hand, the argument-insensitive trampoline code consists of a lot of code blocks as many as the number of obfuscated APIs and they are should be stored in the import table. Therefore, for detecting obfuscated APIs when the argument-insensitive trampoline code is used, we just need to execute each trampoline code block independently. To our best, all previous unpacking approaches has considered only the argument-insensitive trampoline code [19, 4, 5]. In addition, they suggested employing the symbolic execution as a solution to analyze the argument-sensitive trampoline code. In this work, we analyze the argument-sensitive trampoline code by emulating a basic block's first instruction that affects the execution result of the trampoline code. Table 2 is an example of a basic block that calls an API obfuscated by ASProtect. 'call 3240000' can call all obfuscated APIs using the same trampoline code. We run the basic block that the trampoline code belongs to identify all the obfuscated APIs. The trampoline code makes many API calls during execution. The obfuscated API can be found through the 'pop reg' and 'ret' instruction patterns. Before executing the 'ret' instruction, the trampoline code writes the obfuscated API's address in the stack area pointed to by the ESP register. We classified the API addresses stored on the stack as obfuscated APIs. Similarly, argument-insensitive trampoline codes can identify obfuscated APIs through this pattern.

4.2.3 Patching the Target Program

In this paper, we cover two obfuscation techniques: OEP and API obfuscation. First, we patch the instruction executed by OEP obfuscation. The patch target is an instruction that directly branches to the trampoline code. If the presence of OEP obfuscation remains in the unpacked program code, the program's execution flow executes the unmapped memory region. A runtime error occurs because the unpacked program does not contain any sections added by the packer or areas of dynamically allocated memory. To solve this problem, Pinicorn monitors the control flow after the OEP trampoline code, as in Algorithm 1. prevIP is the patch target, and curIP is the first instruction in the following original control flow. We patch the prevIP to branch to curIP. Next, we solve the incorrect import table creation by API obfuscation. Some packers store random values in the IAT at runtime, making it impossible to identify the order of APIs within the IAT accurately. Therefore, Pinicorn creates a new IAT for generic unpacking. We sorted identified APIs by DLL and recorded them in IAT. Next, we reconstruct the IDT, INT, and



(a) Original EntryPoint (b) Obfuscated Original EntryPoint

Figure 4: OEP code comparison between Original program and OEP obfuscated program: shows that the OEP obfuscation program jumps directly to the trampoline code immediately.

```
 >>Exploring -- 0x00ec1023 jmp 0x20ad
 >>Exploring -- 0x00ec20d0 jmp 0x39e6e4
[Jmp to Other Section]  0x00ec20d0 -> 0x0125f7b4 push ebp
[Jmp to Origin Section] 0x00f13902 -> 0x00ec1d30 push ebp
------------------ [not Patched Instruction] -----------------
 >>Exploring -- 0x00ec1023 jmp 0x20ad
 >>Exploring -- 0x00ec20d0 jmp 0x39e6e4
ERROR: Invalid memory mapping (UC_ERR_MAP)
```

Figure 5: Execution result of unpatched program: show that unpatched program accesses invalid memory and exits.

Hint/Names sections using the newly created IAT. We create sections based on the PE Format document published by MSDN. As mentioned earlier, due to the limitation of not being able to utilize the original IAT, the API call instructions need to be patched. Patched instructions will point to the newly created IAT. Finally, we modify the DataDirectory structure's value to point to the newly created IAT and IDT in the PE header. This process allows us to identify the location of the newly unpacked program's import table.

5 Evaluation

In this section, we demonstrate the effectiveness of Pinicorn by deobfuscating and unpacking a program packed by Themida that uses the OEP and API obfuscation techniques.

5.1 OEP Deobfuscation for Themida.

As shown in Figure 4, the EntryPoint-Virtualization option of Themida (v3.0.7) modifies the program's OEP or the basic block's instruction executed after the OEP. Figure 5 shows that the OEP obfuscated code accesses an unmapped memory region after simply dumping an executable file from the memory, resulting in a runtime error. Pinicorn, as in Algorithm 1, detects the trampoline code used for the OEP obfuscation. Pinicorn then patches the obfuscated code to branch to the original code. As a result, Figure 6 shows the same execution results for the original and patched binary. As this case study shows, Pinicorn can solve the OEP obfuscation by analyzing the trampoline code.

Algorithm 1 Deobfuscation algorithms for Themida OEP Obfuscation.

Input: Current Instruction (curINS)
Output: Obfuscated OEP's Address

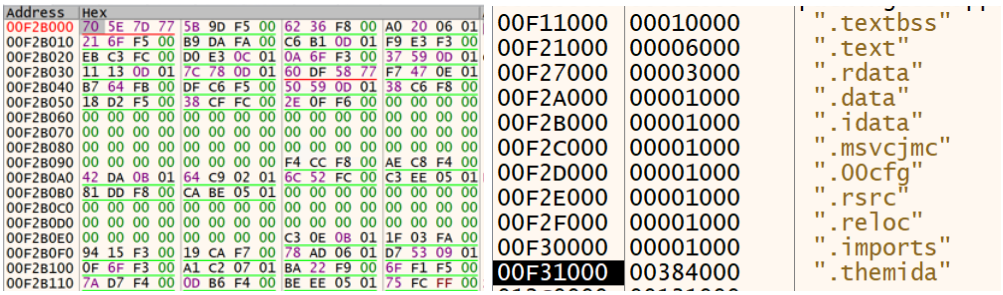
```
1: OEP_TrampolineCode_List ← Trampoline Code's address in OEP
2: Trace_Flag ← False
3: if curINS = OEP_TrampolineCode_List then
4:     prevIP ← curINS
5:     Trace_Flag ← True
6: end if
7: if curINS ⊂ OrigianlCodeSection and Trace_Log = True then
8:     Trace_Flag ← False
9:     curIP ← curINS
10: end if
```

```
 >>Exploring -- 0x00ec1023 jmp 0x20ad
 >>Exploring -- 0x00ec20d0 jmp 0x39e6e4
[Jmp to Other Section]  0x00ec20d0 -> 0x0125f7b4 push ebp
[Jmp to Origin Section] 0x00f13902 -> 0x00ec1d30 push ebp
 >>Exploring -- 0x00ec1d30 push ebp
 >>Exploring -- 0x00ec139d jmp 0x2eb3
 >>Exploring -- 0x00ec3250 push ebp
 >>Exploring -- 0x00ec327c call 0xf44
 >>Exploring -- 0x00ec31c0 push ebp
 >>Exploring -- 0x7607f390 jmp dword ptr [0x760e1878]
[Call API] 0x7607f390 GetSystemTimeAsFileTime
------------------ [Patch Instruction] ----------------------
 >>Exploring -- 0x00ec1023 jmp 0x20ad
 >>Exploring -- 0x00ec20d0 jmp 0xc60
 >>Exploring -- 0x00ec1d30 push ebp
 >>Exploring -- 0x00ec139d jmp 0x2eb3
 >>Exploring -- 0x00ec3250 push ebp
 >>Exploring -- 0x00ec327c call 0xf44
 >>Exploring -- 0x00ec31c0 push ebp
 >>Exploring -- 0x7607f390 jmp dword ptr [0x760e1878]
[Call API] 0x7607f390 GetSystemTimeAsFileTime
```

Figure 6: Execution result of patched program: show that patched program executes the same as the original program.

5.2 API Deobfuscation for Themida.

API-Wrapping option in Themida (v3.0.7) obfuscates all APIs in the packed program's original IAT. All trampoline codes of Themida are argument-insensitive. Also, unlike previous versions and other packers, Themida keeps all trampoline codes in a ".themida" section. Unpacking routines of Themida write the wrapped API's trampoline codes into the original IAT locations, as shown in Figure 7. We parse all addresses and run them independently to identify obfuscated APIs. As shown in Algorithm 2, to solve API obfuscation, Pinicorn runs all the trampoline code blocks. Figure 8 shows Pinicorn's report and the original IAT.



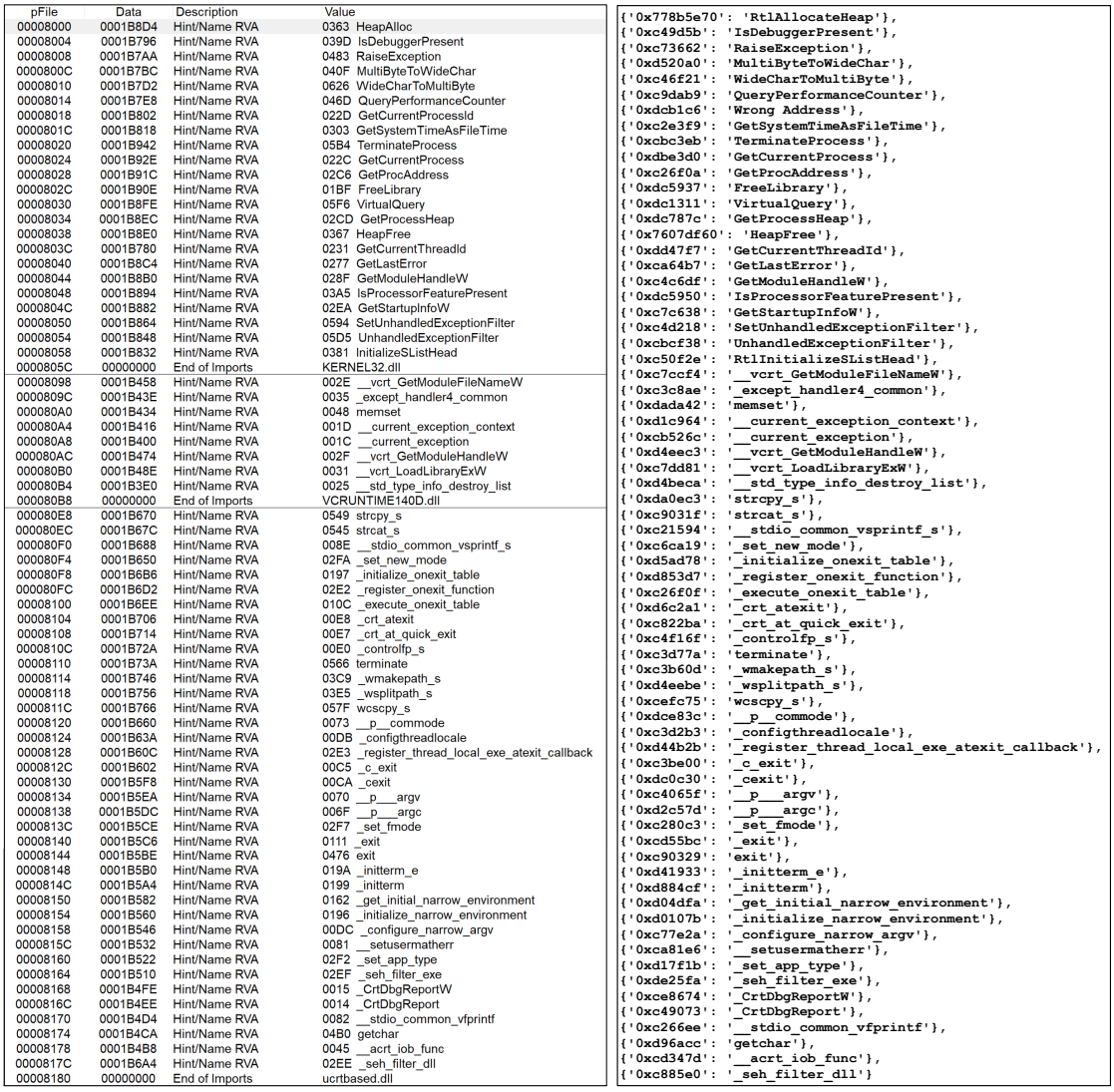
Address	Hex
00F2B000	70 5E 7D 77 5B 9D F5 00 62 36 F8 00 A0 20 06 01
00F2B010	21 6F F5 00 B9 DA FA 00 C6 B1 0D 01 F9 E3 F3 00
00F2B020	EB C3 FC 00 D0 E3 0C 01 0A 6F F3 00 37 59 0D 01
00F2B030	11 13 0D 01 7C 78 0D 01 60 DF 58 77 F7 47 0E 01
00F2B040	B7 64 FB 00 DF C6 F5 00 50 59 0D 01 38 C6 F8 00
00F2B050	18 D2 F5 00 38 CF FC 00 2E 0F F6 00 00 00 00 00
00F2B060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F2B070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F2B080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F2B090	00 00 00 00 00 00 00 00 F4 CC F8 00 AE C8 F4 00
00F2B0A0	42 DA 0B 01 64 C9 02 01 6C 52 FC 00 C3 EE 05 01
00F2B0B0	81 DD F8 00 CA BE 05 01 00 00 00 00 00 00 00 00
00F2B0C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F2B0D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F2B0E0	00 00 00 00 00 00 00 00 C3 0E 0B 01 1F 03 FA 00
00F2B0F0	94 15 F3 00 19 CA F7 00 78 AD 06 01 D7 53 09 01
00F2B100	0F 6F F3 00 A1 C2 07 01 BA 22 F9 00 6F F1 F5 00
00F2B110	7A D7 F4 00 0D B6 F4 00 BE EE 05 01 75 FC FF 00

(a) Import Address Table

00F11000	00010000	".textbss"
00F21000	00006000	".text"
00F27000	00003000	".rdata"
00F2A000	00001000	".data"
00F2B000	00001000	".idata"
00F2C000	00001000	".msvcjmc"
00F2D000	00001000	".00cfg"
00F2E000	00001000	".rsrc"
00F2F000	00001000	".reloc"
00F30000	00001000	".imports"
00F31000	00384000	".themida"

(b) Memory Map

Figure 7: IAT and memory map of API obfuscated program: shows that API obfuscation techniques write the address of trampoline code in IAT.



pFile	Data	Description	Value
00008000	0001B8D4	Hint/Name RVA	0363 HeapAlloc
00008004	0001B796	Hint/Name RVA	039D IsDebuggerPresent
00008008	0001B7AA	Hint/Name RVA	0483 RaiseException
0000800C	0001B7BC	Hint/Name RVA	040F MultiByteToWideChar
00008010	0001B7D2	Hint/Name RVA	0626 WideCharToMultiByte
00008014	0001B7E8	Hint/Name RVA	046D QueryPerformanceCounter
00008018	0001B802	Hint/Name RVA	022D GetCurrentProcessId
0000801C	0001B818	Hint/Name RVA	0303 GetSystemTimeAsFileTime
00008020	0001B942	Hint/Name RVA	05B4 TerminateProcess
00008024	0001B92E	Hint/Name RVA	022C GetCurrentProcess
00008028	0001B91C	Hint/Name RVA	02C6 GetProcAddress
0000802C	0001B90E	Hint/Name RVA	01BF FreeLibrary
00008030	0001B8FE	Hint/Name RVA	05F6 VirtualQuery
00008034	0001B8EC	Hint/Name RVA	02CD GetProcessHeap
00008038	0001B8E0	Hint/Name RVA	0367 HeapFree
0000803C	0001B780	Hint/Name RVA	0231 GetCurrentThreadId
00008040	0001B8C4	Hint/Name RVA	0277 GetLastError
00008044	0001B8B0	Hint/Name RVA	028F GetModuleHandleW
00008048	0001B894	Hint/Name RVA	03A5 IsProcessorFeaturePresent
0000804C	0001B882	Hint/Name RVA	02EA GetStartupInfoW
00008050	0001B864	Hint/Name RVA	0594 SetUnhandledExceptionFilter
00008054	0001B848	Hint/Name RVA	05D5 UnhandledExceptionFilter
00008058	0001B832	Hint/Name RVA	0381 InitializeSListHead
0000805C	00000000	End of Imports	KERNEL32.dll
00008098	0001B458	Hint/Name RVA	002E __vcrt_GetModuleFileNameW
0000809C	0001B43E	Hint/Name RVA	0035 _except_handler4_common
000080A0	0001B434	Hint/Name RVA	0048 memset
000080A4	0001B416	Hint/Name RVA	001D __current_exception_context
000080A8	0001B400	Hint/Name RVA	001C __current_exception
000080AC	0001B474	Hint/Name RVA	002F __vcrt_GetModuleHandleW
000080B0	0001B48E	Hint/Name RVA	0031 __vcrt_LoadLibraryExW
000080B4	0001B3E0	Hint/Name RVA	0025 __std_type_info_destroy_list
000080B8	00000000	End of Imports	VCRUNTIME140D.dll
000080E8	0001B670	Hint/Name RVA	0549 strcpy_s
000080EC	0001B67C	Hint/Name RVA	0545 strcat_s
000080F0	0001B688	Hint/Name RVA	008E __stdio_common_vsprintf_s
000080F4	0001B650	Hint/Name RVA	02FA _set_new_mode
000080F8	0001B6B6	Hint/Name RVA	0197 _initialize_onexit_table
000080FC	0001B6D2	Hint/Name RVA	02E2 _register_onexit_function
00008100	0001B6EE	Hint/Name RVA	010C _execute_onexit_table
00008104	0001B706	Hint/Name RVA	00E8 _crt_atexit
00008108	0001B714	Hint/Name RVA	00E7 _crt_at_quick_exit
0000810C	0001B72A	Hint/Name RVA	00E0 _controlfp_s
00008110	0001B73A	Hint/Name RVA	0566 terminate
00008114	0001B746	Hint/Name RVA	03C9 _wmakepath_s
00008118	0001B756	Hint/Name RVA	03E5 _wsplitpath_s
0000811C	0001B766	Hint/Name RVA	057F wcscpy_s
00008120	0001B660	Hint/Name RVA	0073 __p__commode
00008124	0001B63A	Hint/Name RVA	00DB _configthreadlocale
00008128	0001B60C	Hint/Name RVA	02E3 _register_thread_local_exe_atexit_callback
0000812C	0001B602	Hint/Name RVA	00C5 _c_exit
00008130	0001B5F8	Hint/Name RVA	00CA _cexit
00008134	0001B5EA	Hint/Name RVA	0070 __p___argv
00008138	0001B5DC	Hint/Name RVA	006F __p___argc
0000813C	0001B5CE	Hint/Name RVA	02F7 _set_fmode
00008140	0001B5C6	Hint/Name RVA	0111 _exit
00008144	0001B5BE	Hint/Name RVA	0476 exit
00008148	0001B5B0	Hint/Name RVA	019A _initterm_e
0000814C	0001B5A4	Hint/Name RVA	0199 _initterm
00008150	0001B582	Hint/Name RVA	0162 _get_initial_narrow_environment
00008154	0001B560	Hint/Name RVA	0196 _initialize_narrow_environment
00008158	0001B546	Hint/Name RVA	00DC _configure_narrow_argv
0000815C	0001B532	Hint/Name RVA	0081 __setusermatherr
00008160	0001B522	Hint/Name RVA	02F2 _set_app_type
00008164	0001B510	Hint/Name RVA	02EF _seh_filter_exe
00008168	0001B4FE	Hint/Name RVA	0015 _CrtDbgReportW
0000816C	0001B4EE	Hint/Name RVA	0014 _CrtDbgReport
00008170	0001B4D4	Hint/Name RVA	0082 __stdio_common_vfprintf
00008174	0001B4CA	Hint/Name RVA	04B0 getchar
00008178	0001B4B8	Hint/Name RVA	0045 __acrt_iob_func
0000817C	0001B6A4	Hint/Name RVA	02EE _seh_filter_dll
00008180	00000000	End of Imports	ucrtbased.dll

(a) Original Import Address Table

```
{'0x778b5e70': 'RtlAllocateHeap'},
{'0xc49d5b': 'IsDebuggerPresent'},
{'0xc73662': 'RaiseException'},
{'0xd520a0': 'MultiByteToWideChar'},
{'0xc46f21': 'WideCharToMultiByte'},
{'0xc9dab9': 'QueryPerformanceCounter'},
{'0xdcb1c6': 'Wrong Address'},
{'0xc2e3f9': 'GetSystemTimeAsFileTime'},
{'0xcbc3eb': 'TerminateProcess'},
{'0xdbe3d0': 'GetCurrentProcess'},
{'0xc26f0a': 'GetProcAddress'},
{'0xdc5937': 'FreeLibrary'},
{'0xdc1311': 'VirtualQuery'},
{'0xdc787c': 'GetProcessHeap'},
{'0x7607df60': 'HeapFree'},
{'0xdd47f7': 'GetCurrentThreadId'},
{'0xca64b7': 'GetLastError'},
{'0xc4c6df': 'GetModuleHandleW'},
{'0xdc5950': 'IsProcessorFeaturePresent'},
{'0xc7c638': 'GetStartupInfoW'},
{'0xc4d218': 'SetUnhandledExceptionFilter'},
{'0xcbcf38': 'UnhandledExceptionFilter'},
{'0xc50f2e': 'RtlInitializeSListHead'},
{'0xc7ccf4': '__vcrt_GetModuleFileNameW'},
{'0xc3c8ae': '_except_handler4_common'},
{'0xdada42': 'memset'},
{'0xd1c964': '__current_exception_context'},
{'0xcb526c': '__current_exception'},
{'0xd4eec3': '__vcrt_GetModuleHandleW'},
{'0xc7dd81': '__vcrt_LoadLibraryExW'},
{'0xd4beca': '__std_type_info_destroy_list'},
{'0xda0ec3': 'strcpy_s'},
{'0xc9031f': 'strcat_s'},
{'0xc21594': '__stdio_common_vsprintf_s'},
{'0xc6ca19': '_set_new_mode'},
{'0xd5ad78': '_initialize_onexit_table'},
{'0xd853d7': '_register_onexit_function'},
{'0xc26f0f': '_execute_onexit_table'},
{'0xd6c2a1': '_crt_atexit'},
{'0xc822ba': '_crt_at_quick_exit'},
{'0xc4f16f': '_controlfp_s'},
{'0xc3d77a': 'terminate'},
{'0xc3b60d': '_wmakepath_s'},
{'0xd4eebe': '_wsplitpath_s'},
{'0xcefc75': 'wcscpy_s'},
{'0xdce83c': '__p__commode'},
{'0xc3d2b3': '_configthreadlocale'},
{'0xd44b2b': '_register_thread_local_exe_atexit_callback'},
{'0xc3be00': '_c_exit'},
{'0xdc0c30': '_cexit'},
{'0xc4065f': '__p___argv'},
{'0xd2c57d': '__p___argc'},
{'0xc280c3': '_set_fmode'},
{'0xcd55bc': '_exit'},
{'0xc90329': 'exit'},
{'0xd41933': '_initterm_e'},
{'0xd884cf': '_initterm'},
{'0xd04dfa': '_get_initial_narrow_environment'},
{'0xd0107b': '_initialize_narrow_environment'},
{'0xc77e2a': '_configure_narrow_argv'},
{'0xca81e6': '__setusermatherr'},
{'0xd17f1b': '_set_app_type'},
{'0xde25fa': '_seh_filter_exe'},
{'0xce8674': '_CrtDbgReportW'},
{'0xc49073': '_CrtDbgReport'},
{'0xc266ee': '__stdio_common_vfprintf'},
{'0xd96acc': 'getchar'},
{'0xcd347d': '__acrt_iob_func'},
{'0xc885e0': '_seh_filter_dll'}
```

(b) Deobfuscated Import Table

Figure 8: Comparison of original program's IAT and deobfuscated program's IAT.

Algorithm 2 Deobfuscation algorithms for Themida API Obfuscation.

Input: Current Instruction (curINS)
Output: Obfuscated API's Address

```
1: API_TrampolineCode_List ← Trampoline Code's address in IAT
2: DllInfo_List ← DLL's information
3: Trace_Flag ← False
4: if curINS = API_TrampolineCode_List then
5:     Trace_Flag ← True
6: end if
7: if Trace_Log = True and curINS_Opcode = "ret" then
8:     if *ESP ⊂ DllInfo_List then
9:         Trace_Flag ← False
10:        API_TrampolineCode_List ← API_TrampolineCode_List + 1
11:        curINS ← API_TrampolineCode_List
12:    end if
13: end if
```

6 Conclusion

In this paper, we propose Pinicorn that dynamically analyze and deobfuscate the OEP and API obfuscation techniques for unpacking packed programs. Pinicorn first executes the packed binary and detects candidates of the OEP. When Pinicorn metts each candidate, it dumps the program's context. Then, it dynamically analyzes the binary by emulating it with the recorded context information. After the analysis, Pinicorn patches the program according to the detection results. Through the evaluation, we showed that Pinicorn can successfully deobfuscate the OEP and API obfuscation techniques. In our future work, we will evaluate our approach in large-scale to show and discuss the effectiveness and limitations of Pinicorn in detail.

7 Acknowledgments

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government, Ministry of Science and ICT (MSIT) (No.2017-0-00168, Automatic Deep Malware Analysis Technology for Cyber Threat Intelligence).

References

[1] Munkhbayar Bat-Erdene, Taebeom Kim, Hyundo Park, and Heejo Lee. Packer detection for multi-layer executables using entropy analysis. *Entropy*, 19(3):125, 2017.

[2] Binlin Cheng and Pengwei Li. Bareunpack: Generic unpacking on the bare-metal operating system. *IEICE TRANSACTIONS on Information and Systems*, 101(12):3083–3091, 2018.

[3] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 395–411, 2018.

[4] Binlin Cheng, Jiang Ming, Erika A Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean-Yves Marion. {Obfuscation-Resilient} executable payload extraction from packed malware. In *30th USENIX Security Symposium* (*USENIX Security 21*), pages 3451–3468, 2021.

[5] Seokwoo Choi, Taejoo Chang, Changhyun Kim, and Yongsu Park. X64unpack: Hybrid emulation unpacker for 64-bit windows environments and detailed analysis results on vmprotect 3.4. *IEEE Access*, 8:127939–127953, 2020.

[6] STEFANO D'ALESSIO and SEBASTIANO MARIANI. Pindemonium: a dbi-based generic unpacker for windows executables. 2016.

[7] Ashkbiz Danehkar. *Yoda's Protector*, 2013. https://sourceforge.net/projects/yodap/, last viewed July 2022.

[8] FireEye. M-trends 2020, 2020. https://content.fireeye.com/m-trends/rpt-m-trends-2020, last viewed July 2022.

[9] Ryoichi Isawa, Daisuke Inoue, and Koji Nakao. An original entry point detection method with candidate-sorting for more effective generic unpacking. *IEICE TRANSACTIONS on Information and Systems*, 98(4):883–893, 2015.

[10] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 46–53, 2007.

[11] Gyeong Min Kim and Yong Su Park. Improved original entry point detection method based on pindemonium. *KIPS Transactions on Computer and Communication Systems*, 7(6):155–164, 2018.

[12] David Korczynski. Repeconstruct: reconstructing binaries with self-modifying code and import address table destruction. In *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–8. IEEE, 2016.

[13] Young Bi Lee, Jae Hyuk Suk, and Dong Hoon Lee. Bypassing anti-analysis of commercial protector methods using dbi tools. *IEEE Access*, 9:7655–7673, 2021.

[14] Osnat Levi. Pin - a dynamic binary instrumentation tool, 2007–2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html. Accessed on: July 2, 2022.

[15] Osnat Levi. Unicorn - the ultimate cpu emulator, 2015–2022. https://www.unicorn-engine.org/, last viewed July 2022.

[16] Oreans Technologies. Themida overview - oreans technologies, 2004–2022. https://www.oreans.com/Themida.php, last viewed July 2022.

[17] VMProtect Software. VMProtect Software Protection, 2003–2022. https://vmpsoft.com/, last viewed July 2022.

[18] StartForge. *ASProtect*, 2007–2022. http://www.aspack.com/asprotect32.html/, last viewed July 2022.

[19] Jae Hyuk Suk, Jae-Yung Lee, Hongjoo Jin, In Seok Kim, and Dong Hoon Lee. Unthemida: Commercial obfuscation technique analysis with a fully obfuscated program. *Software: Practice and Experience*, 48(12):2331–2349, 2018.

[20] The UPX Team. *UPX - the Ultimate Packer for eXecutables.*, 1996–2022. https://upx.github.io/, last viewed July 2022.

[21] The Enigma Protector. Enigma protector, 2004–2022. https://enigmaprotector.com/, last viewed Aug 2022.

[22] Yaldex. Acprotect standard, 2006–2022. http://www.yaldex.com/Bestsoft/Utilities/acprotect.htm, last viewed July 2022.