IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 13, 2020, accepted December 23, 2020, date of publication January 1, 2021, date of current version January 13, 2021.

Digital Object Identifier 10.1109/ACCESS.2020.3048848

Bypassing Anti-Analysis of Commercial Protector Methods Using DBI Tools

YOUNG BI LEE, JAE HYUK SUK, AND DONG HOON LEE, (Member, IEEE)
Graduate School of Information Security, Korea University, Seoul 02841, South Korea

Corresponding author: Dong Hoon Lee (donghlee@korea.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government Ministry of Science and ICT (MSIT) under Grant NRF-2017R1A2B3009643.

ABSTRACT As most malware is infectious, anti-analysis and packing techniques supported by commercial protectors are conventionally applied to hinder analysis. When analyzing to detect and block such protected malware, it is necessary to do so in a virtual environment to prevent infection. In terms of packing, it is necessary to analyze using dynamic binary instrumentation (DBI), a dynamic analysis tool, which is advantageous for unpacking because DBI inserts code at run time and analyzes it dynamically. However, malware terminates on its own when it detects a virtual environment or DBI due to anti-analysis techniques. Therefore, it is necessary to also bypass anti-VM and anti-DBI techniques in order to successfully analyze malware in a virtual environment using DBI. It is very difficult for analysts to bypass anti-VM and anti-DBI techniques that are used in commercial protectors because analysts generally have little information on what methods are used or how to even bypass these techniques. In this paper, we suggest guidelines to aid in easy analysis of malware protected by anti-VM and anti-DBI techniques supported by commercial protectors. We analyzed the techniques used by five of the most common commercial protectors, and herein present how to bypass anti-VM and anti-DBI techniques supported by commercial protectors via a detailed algorithm analysis. We performed a bypass experiment after applying each commercial protector to 1573 executable files containing vulnerabilities provided by the National Institute of Standards and Technology (NIST). To our knowledge, this is the first empirical study to suggest detailed bypassing algorithms for anti-VM and anti-DBI techniques used in commercial protectors.

INDEX TERMS Obfuscation, commercial protectors, anti-analysis, anti-VM, anti-DBI, DBI tool.

I. INTRODUCTION

There is a steady increase in attacks utilizing malware on computer systems. Most malware is distributed via application of anti-analysis techniques and packing, a code obfuscation technique, while still supported by commercial protectors, inhibiting antivirus programs from detecting the presence of such malware. However, if anti-analysis techniques are applied, the malware may be forcibly terminated depending on the existence of a virtual environment or a debugger. Packing is a technique that changes the control flow of program execution, making it difficult to analyze. As such, many analysts use dynamic binary instrumentation (DBI) tools to analyze the packing of commercial protectors [1]–[3]. DBI tools insert executable code at run time to help dynamically analyze program behavior and make it possible to effectively assess obfuscation and packing techniques. Previous studies using DBI tools have already suggested various methods for unpacking or effectively analyzing programs protected by commercial protectors.

It is important to use a virtual environment when analyzing actual malware with certain commercial protectors that use an analysis method coupled with a DBI tool. This is because the host OS may become inadvertently infected while performing the malware analysis. However, if an anti-virtual machine (anti-VM) technique is applied to the malicious code, analysis methods using DBI tools cannot be used in a virtual environment. In addition, even when anti-DBI techniques are applied, analysis methods using DBI tools still cannot be used. Therefore, it is crucial to bypass anti-VM and anti-DBI techniques in order to analyze malware used on commercial protectors using DBI tools more efficiently in the field.

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

Assessment in this context is time consuming, as analysts must bypass the protected techniques before using DBI tools to analyze the malware. The longer it takes to analyze the malware, the more damage can occur because antivirus programs will not be able to update. In light of this, analysts must also understand anti-VM and anti-DBI techniques used in commercial protectors. In order to analyze malware in relation to the anti-analysis technique applied, many researchers are now studying anti-anti-analysis techniques, that is, bypassing anti-analysis techniques.

Although anti-anti-analysis techniques have been studied to a degree in terms of a theoretical framework [4]–[6], these are not directly suitable for anti-analysis techniques provided by commercial protectors. Further analysis is necessary because analysts cannot know which anti-analysis technique to use for commercial protectors. Also, even if the analyst knows which anti-analysis technique to use, the theoretical content cannot be applied as is because the anti-analysis technique may be modified. Therefore, empirical studies showing the results of various anti-analysis techniques used in commercial protectors are needed.

The goal of this paper is to enable analysts to use DBI tools when analyzing malware employing anti-analysis techniques supported by commercial protectors in a virtual environment. Therefore, we propose guidelines to aid in the easy analysis of malware protected by anti-VM or anti-DBI techniques supported by commercial protectors. In addition, we present bypass algorithms for anti-VM and anti-DBI techniques used in commercial protectors along with our implementation and evaluation results.

Contributions: The following presents a detailed discussion of our contributions.

- We propose guidelines to aid in the easy analysis of malware protected by anti-VM and anti-DBI techniques supported by commercial protectors. We present the results of our detailed analysis for anti-VM and anti-DBI algorithms used in commercial protectors, and we also categorize and present the techniques used with each tool. Therefore, our findings can be helpful for analysts who want to analyze malware protected by commercial protectors.
- We present how to bypass anti-VM and anti-DBI techniques supported by commercial protectors through a detailed analysis of the algorithms used. This paper features the first empirical research results produced via a detailed analysis of anti-analysis techniques provided by commercial protectors, and the first to suggest actual algorithms accordingly.

The organization of this paper is as follows: Section II describes the background of existing anti-analysis techniques and their respective categories, and Section III presents related works about bypassing anti-analysis. Section IV categorizes and presents anti-analysis techniques provided by commercial protectors, which are the actual targets for analysis. Section V presents an algorithm that bypasses anti-analysis techniques used in commercial protectors. Section VI presents our implementation and evaluation. Finally, we give our discussion and conclusion in Section VII and VIII, respectively.

II. BACKGROUND

A. CODE OBFUSCATION

Code obfuscation, a technique used to transform a program, hinders analysis because it modifies the internal code while maintaining functionality of the program. There are four categories of code obfuscation [7]: layout, data, control flow, and preventive.

- **Layout Obfuscation**: This technique modifies or removes detailed elements that do not affect the execution of programs. It mainly corresponds to a technique that makes it difficult to transform variable names or remove comments.
- **Data Obfuscation**: This technique transforms data values in a program or reconstructs data structures. It mainly corresponds to a technique that transforms variable values in a complex way or reconstructs array structures.
- **Control Flow Obfuscation**: This technique inserts dummy codes or modifies the control flow from inside the program. It mainly corresponds to a technique that inserts meaningless branch statements or transforms the control flow graph.
- **Preventive Obfuscation**: This technique inserts analysis prevention routines or analysis prevention codes inside programs. It mainly corresponds to a technique that prevents or terminates the operation of an analysis tool (e.g., a debugger or a disassembler).

In actuality, preventive obfuscation is based on the same concept as anti-analysis techniques, and this paper aims to implement bypass anti-analysis technique modules.

B. DYNAMIC BINARY INSTRUMENTATION

DBI is a technique primarily used for the dynamic analysis of programs [8], and it can be analyzed by inserting instrumentation code during program execution. As such, DBI is used to measure program performance, and analysts can use DBI tools to observe memory and register values during program execution, detect memory allocation errors, and perform security threat analysis. Because of these advantages, it has also recently been used to analyze malware using DBI. Tools such as PIN [9], DynamoRIO [10], and Valgrind [11] typically provide a framework to perform DBI.

In this paper, we chose to use PIN from the numerous DBI tools. PIN is a DBI framework provided by Intel that can be used in IA-32, x86-64, and MIC instruction-set architectures. It is widely used in security research because it allows for measurement at the granular level and provides an intuitive application programming interface (API). We chose to use PIN for these reasons, but it should be noted that our proposed algorithm and results can still be used with other DBI tools.

C. ANTI-ANALYSIS

Anti-analysis is a technique that prevents or interferes with program analysis. To make program analysis difficult, a program in which anti-analysis is applied can detect the analysis environment or analysis tools and forcibly terminate itself. Commercial protectors also offer diverse anti-analysis techniques, among these the most representative being anti-debugging, anti-VM, anti-patching, and anti-dumping. Anti-debugging is a technique that prevents analysis using a debugger, and anti-VM techniques achieve this in a virtual environment. Anti-patching mainly detects whether a file has been patched using a checksum value, and finally, anti-dumping detects and defends memory dumps performed by dump tools. Note that anti-debugging techniques do not only detect debugging tools. Some also detect DBI tools, which are referred to as anti-DBI techniques. Although anti-DBI techniques is not a separate option provided by commercial protectors, we analyzed some techniques provided by anti-debugging options in commercial protectors to detect and prevent DBI tools.

D. ANTI-VM

Anti-VM is a technique that detects a virtual machine environment and terminates the program so that it cannot be analyzed in a virtual machine environment. This technique is currently provided by various commercial protectors and is used most commonly to prevent analysis in a virtual environment. The virtual environment detection methods currently used in commercial protectors can be classified into three basic categories as follows:

- **Registry-based**: When Windows is installed in a virtual environment, the registry is set as information about the virtual machine. Therefore, there is a difference between the registry value of the guest OS and that of the host OS, such that virtual environment detection can be performed using this.
- **Hardware-based**: In a virtual environment, logically separated H/W is used through the hypervisor, not the physical H/W used in the host PC. Accordingly, there is a difference between the H/W information of the guest OS and that of the host OS. Virtual environment detection is performed using the difference between the H/W information of the guest OS and the host OS.
- **Process/Service-based**: In a virtual environment, specific programs are installed to use each virtual machine more effectively. Therefore, there are certain programs that only exist in the guest OS and are absent from the host OS. For example, a virtual environment is detected using a specific program that exists only in the guest OS.

There has been limited research on anti-VM techniques up to this point, and only a limited offering of empirical research has explored which techniques are used in real commercial protectors and how to bypass them.

E. ANTI-DBI

Anti-DBI techniques detect the situation under analysis with a DBI tool and forcibly terminates the program, which consequentially means that dynamic analysis cannot be performed using a DBI tool. Anti-DBI performs DBI tool detection using the principles or features of a certain DBI tool. Currently, commercial protectors do not provide any standalone anti-DBI option. In fact, the anti-DBI techniques provided by commercial protectors simply overlap with the techniques provided in anti-debugging options. So far, few cases have been systematically studied to block DBI tools. The anti-DBI techniques currently used in commercial protectors can be classified into four basic categories.

- **Overhead-based**: The difference between the general program execution time and the execution time of a program being analyzed using the DBI tool is that the DBI tool can be detected through the difference in execution time of the analysis target program.
- **JIT complier-based**: Unlike general programs, the DBI tool uses a just-in-time (JIT) compiler to patch and compile certain instructions in the program to be analyzed in real-time to perform program analysis. A DBI tool is detected using patching and compiling the instructions in the DBI tool cache.
- **API-based**: Windows provides anti-debugging application programming interfaces (APIs). Some APIs that perform anti-debugging can detect not only debuggers but also DBI tools, and anti-DBI uses these APIs to detect DBI tools.
- **Exception-based**: In most general situations, programs perform exception handling where the exception occurs. During analysis using a DBI tool, programs operate without executing exception handling in the part where the exception occurs. Therefore, if an exception occurs, a routine is inserted into the general program and the normal operation code of the program is inserted into the exception handling part. Then, a DBI tool can be detected as it operates the program without processing the exception.

A detailed outline of the anti-VM and anti-DBI techniques used in commercial protector tools is provided in section IV.

III. RELATED WORKS

This section describes existing research related to malware analysis and unpacking using DBI tools, and it also explains how the bypassing anti-VM and anti-DBI techniques proposed in this paper can be used efficiently in the field only when they are provided together.

Suk *et al.* [3] used PIN to analyze Themida. Based on the analysis results, Themida's unpacking method was implemented by the algorithm, and the unpacking results were verified using a large data set. This is the first empirical study to analyze almost all Themida-based obfuscation options simultaneously while reconstructing them closely to the original program. However, Suk *et al.* treats the anti-VM option as out of scope even though Themida has an anti-VM option.

Suk *et al.* can remove in the unpacking process even if the anti-VM option is applied, but this is limited to the case in which unpacking is performed in the host OS. However, unpacking the malware from the host OS may lead to a situation where the host OS becomes infected by the malware. When analyzing malware, it is important to analyze malware in a virtual environment, making it is necessary to consider an anti-VM option. Therefore, by applying Themida's anti-anti-VM algorithm presented in this paper, the analysis results will be enhanced with a more empirical study that can be applied in the field.

M.Polino *et al.* [12] collected a large volume of malware, analyzed the anti-DBI techniques applied to them, and suggested countermeasures. The researchers classified anti-DBI techniques into four categories based on the 7,006 malware programs they collected and analyzed in terms of how many anti-DBIs were applied. The results of this analysis and bypass were also described, and in order to show that their results could be applied to commercial packers, unpackers were produced in prototype form. Subsequently, the authors also evaluated bypassing anti-DBI techniques and unpacking results for commercial packers. However, although this study described 1,093 of the 7,006 malware programs as applying an anti-DBI technique, the existence of a ground truth appears to be lacking. Therefore, the method lacks a definitive way to check whether or not anti-DBI techniques were applied to the 1,093 malware. Accordingly, there is a limitation in that the possibility of false positives or false negatives cannot be excluded. In fact, the researchers do not separately indicate that an anti-DBI technique has been applied in the 1,093 malware included in the study. In addition, anti-DBI bypassing and unpacking experiments were performed on commercial packers used by malware in the wild. During the experiment, there was a case in which unpacking was not performed correctly. This was because the experiment was performed in a virtual machine environment and an anti-VM technique was applied to the malware. Therefore, in order to empirically analyze malware using dynamic tools, not only anti-anti-DBI but also anti-anti-VM techniques must be provided.

Park *et al.* [13] suggested automatic anti-debugging technique detection and bypass methods using the PIN for a number of self-selected commercial protectors. The researchers performed DBI detection analysis along with anti-debugging techniques and suggested a bypass method for DBI detection. In addition, it was described that the PIN works normally for a program applying anti-analysis via each commercial protector. However, there is insufficient explanation about the anti-DBI algorithms used by each commercial protector and proposed bypassing methods. Therefore, there is a limitation in that sufficient information cannot be provided to analysts who want to assess the anti-analysis techniques of commercial protectors using the information presented in that study. In response to this, we present our results after analyzing detailed algorithms used by each commercial protector, which means that it is possible to provide sufficient information to the empirical analyst through the algorithm proposed in this paper.

Cheng, Binlin, *et al.* [14] proposed a new unpacking process called rebuilt-then-called, which differs from the existing written-then-executed process. Rebuilt-then-called uses the feature of redesigning the import address table (IAT) and calling the API just before the original execution code is executed in the packed program. It is also a technique of searching for the original entry point (OEP). This process can be analyzed without being detected by anti-debugging, anti-DBI, or other anti-analysis techniques because it uses kernel-level DLL hijacking without using dynamic analysis tools. For performance evaluation, the authors collected 238,835 packed malware and conducted experiments. A laptop was used as the environment in which the malware was unpacked, and as a result of the experiment, the tool described unpacking to be successful with a 97.3% probability. This experiment demonstrated that unpacking is possible even when anti-VM techniques are applied. However, there is no part that explains whether this success was due to the application of the anti-anti-VM technique, or if it was merely not necessary to bypass because the experiment was conducted in the host OS. When experimentation occurs in the host OS, if the part that the authors suggest would redesign the IAT is not found, the host OS can be infected with malware. In fact, the researchers state that unpacking has failed with a 2.7% probability. Also, owing to the custom packer of the malicious behavior payload that does not utilize IAT, there is a limitation in that it is difficult to prevent packing-based malware from affecting the host OS. Therefore, it is important to perform the analysis in a virtual environment to cope with various malware attack scenarios. In this paper, we propose an anti-anti-VM study to complement the limitations of the technique suggested by Cheng, Binlin, *et al.* Therefore, if the above study were to consider the results of this paper, analysis may be conducted in a safer environment.

D'Elia, Daniele Cono, *et al.* [15] proposed Bluepill, a human-centered dynamic analysis system to facilitate malware analysis. Bluepill is based on DBI tool, and the authors of the above mentioned study configured a rule set for automatic bypass by analyzing known anti-analysis techniques. The rule set is a form in which a bypass algorithm is built for each detailed anti-analysis technique, such that it can be automatically bypassed when using the framework. Therefore, when analyzing malware using Bluepill, analysts can assess only anti-analysis techniques that are not included in the existing framework and add them to the framework's rule set. However, in the case of the anti-analysis technique provided by commercial protectors, it is difficult to analyze anti-analysis technique algorithms and add them to the rule set because obfuscation is applied simultaneously. Therefore, it is possible to supplement the anti-analysis bypass methods of commercial protectors by updating Bluepill using the research results of this paper.

Choi, Seokwoo, *et al.* [16] proposed x64Unpack, which analyzes the packed executable file and unpacks it.

TABLE 1. Summary of anti-VM and anti-DBI techniques used in commercial protectors.

Protector	Anti-VM Techniques	Anti-DBI Techniques
Themida	- Registry - Hardware (*IN* Instruction)	- N/A
Enigma	- Hardware (*IN*, *CPUID* Instruction) - Process (*VBoxServiece.exe*)	- N/A
VMProtect	- Hardware (*CPUID* Instruction) - Hardware (Firmware Table Information)	- Exception (Single Step)
Obsidium	- Hardware (*IN* instruction) - Hardware (Disk Drive Value, Firmware Table Information) - Process (*VBoxGuest*)	- API (*ZwQueryInformationProcess* (0x1f)) - Process (*VBoxGuest*) - Exception (Single Step)
ACProtect	- N/A	- JIT-Compiler (Self-Modification)

In addition, the study presents analysis results on how the program packed with VMProtect 3.4 works using x64Unpack and which API is used. The researchers used x64Unpack to bypass and unpack VMProtect, Themida, UPX, and MPRESS anti-reversing techniques. x64Unpack is a form of running a packed program using a CPU emulator in a host environment. If anti-reversing techniques appear in the process of running, they are bypassed by having the authors' proposal follow a predefined API and exception handling routine. To extend x64Unpack so that other commercial protectors can be unpacked, an anti-reversing technique bypassing the API and exception handling routine must be added after additional analysis of the commercial protector. Therefore, the commercial protector bypass methods presented in this paper may improve the x64Unpack extension.

IV. ANTI-VM & ANTI-DBI ANALYSIS OF COMMERCIAL PROTECTORS

In this section, we describe the detailed algorithm after analyzing anti-VM and anti-DBI techniques provided by five commonly used commercial protectors (i.e., Themida, Enigma, VMProtect, Obsidium, and ACProtect). Upon analyzing the five commercial protectors, we summarized anti-VM and anti-DBI techniques provided by each commercial protector as shown in Table 1. Anti-VM techniques are provided as an option in all commercial protectors except one. All the tools that provided the anti-VM option also provided virtual environment detection techniques using the hardware features of the guest OS. Unfortunately, the anti-DBI technique has not yet been widely made available in commercial protectors. Tools either contain one or three anti-DBI techniques, which all provide an array of technique types. Later in this section, detailed algorithms are described related to anti-VM and anti-DBI used by each commercial protector. With the detailed algorithms of anti-VM, analysis was performed on VMware and VirtualBox, which are two of the most commonly used virtual machines. VMware and VirtualBox in a virtual environment are representative enough to be featured in two performance comparison papers spanning a wide window of time [17], [18].

A. THEMIDA

Themida is a protector tool that provides 21 obfuscation options, including anti-analysis techniques at the binary level [19], and it continues to operate with new versions being released. In this paper, we performed our analysis on Themida version 2.4.5. Note that Themida offers various anti-analysis options, but no anti-DBI techniques.

1) ANTI-VM OF THEMIDA

Themida detects a virtual environment using two types of anti-VM techniques. When detecting a virtual environment, a message box like in Fig.1 appears and the program is terminated.

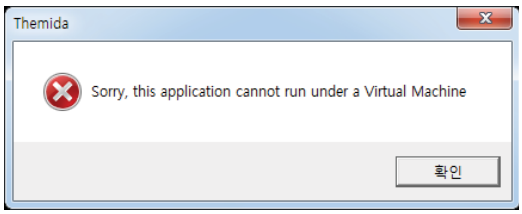

FIGURE 1. A pop-up message box that appears when executing the program applied Themida's anti-VM in a virtual environment.

a: DETECTION USING REGISTRY KEY VALUE

In each registry key value, there are various features that can be assumed to be virtual environments. However, features used by VMware and VirtualBox are utilized differently. When executing a program with Themida obfuscation applied in a VMware environment, it reads the DriverDesc[1] value *VMware SVGA 3D* among the values stored in the registry and performs virtual environment detection through string match. See the VMware part in Fig.2.

In VirtualBox, the virtual environment is detected through string match by reading the *VBOX −1* value of SystemBiosVersion and the *Oracle VM VirtualBox* value of

[1]HKEY_LOAL_MACHINE\SYSTEM\ControlSet001\Control\Class\{4d36e968-e325-11ce-bfc1-08002be10318}\0000

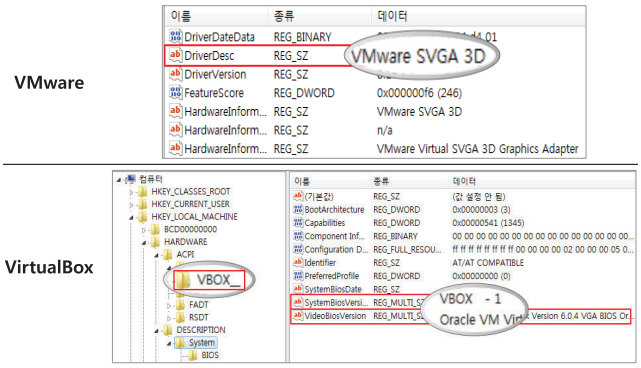

FIGURE 2. Registry of VMware and VirtualBox used by Themida.

VideoBiosVersion among the values stored in the registry[2] when executing the program with Themida obfuscation. In addition, detection is performed by using the *VBOX__* value, which is the folder name inside HARDWARE in HKLM.[3] See the VirtualBox part in Fig.2.

b: BYPASSING DETECTION USING REGISTRY KEY VALUE

In order to perform a DriverDesc value string match in the VMware registry, the memory value is read using the *memmove API*. When calling the *memmove API*, the third parameter, a source address value, is checked to see if it has the value as the *VMware SVGA 3D*. As shown in Fig.3, if the value exists, the anti-VM can be bypassed by modifying it.

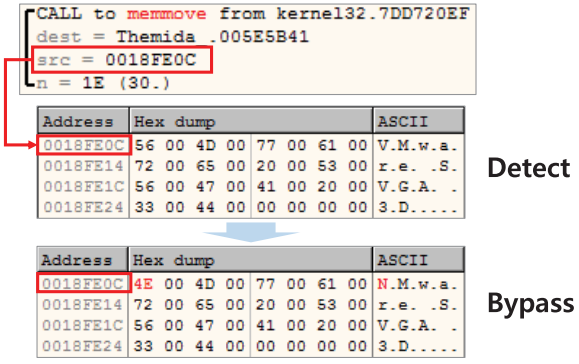

FIGURE 3. Bypassing a detection technique through modification of the *memmove API*.

In VirtualBox, *memmove API* is used to read *VBOX-1* and *Oracle VM VirtualBox* values from the memory to perform a string match. Therefore, bypass is possible through the detection and modification of the corresponding value.

Additionally, VirtualBox is detected by checking whether the *VBOX__* folder exists in the registry using the *RegOpenKeyExA API*. Therefore, as shown in Fig.4, it is possible to bypass detection by modifying the value of the registry folder name existing in the second argument.

[2]HKEY_LOAL_MACHINE \HARDWARE\DESCRIPTION\Control \System

[3]HKEY_LOCAL_MACHINE \HARDWARE\ACPI\DSCT\VBOX__

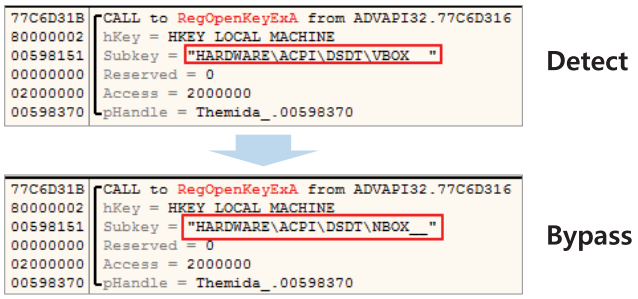

FIGURE 4. Bypassing a detection technique through modification of the *RegOpenKeyExA API*.

c: DETECTION USING IN INSTRUCTION

The VMware virtual machine has an I/O port communication channel, through which it exchanges data between the host OS and the guest OS. If analysts use the *IN* instruction, they can obtain information about I/O port. Additionally, if analysts execute the *IN* instruction by inserting the value 0x5658 (i.e.,VX) in the DX register (i.e., the communication channel), the value containing the information of the virtual machine is stored in the EAX or EBX register. Therefore, it is possible to detect the virtual machine using the *IN* instruction.

d: BYPASSING DETECTION USING IN INSTRUCTION

Themida uses the *IN* instruction twice to detect the VMware environment. In the first case, 0x564D5868 (i.e.,VMXh), which indicates a magic number, is put in the EAX register, and 0x14 (memory size request) is put in the ECX register. The host OS cannot execute the *IN* instruction, so a 0x0 value is entered in the EAX register. However, the *IN* instruction is executed in the guest OS, and a value other than 0x0 is received. Therefore, the anti-VM can be bypassed by modifying the EAX register value to 0x0 after the *IN* instruction executes as shown in Fig.5.

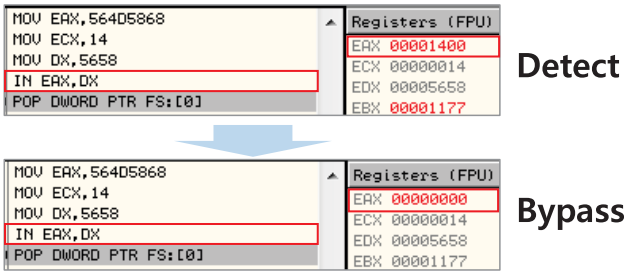

FIGURE 5. Bypassing a detection technique through modification of the register (*IN*1 instruction).

In the second case, 0x564D5868 (i.e.,VMXh) is put in the EAX register, and 0xA (a request VMware Version type) is put in the ECX register. Subsequent execution of the *IN* instruction in the guest OS leaves a unique value of 0x564D5868 (i.e.,VMXh) in the EBX register. Therefore, the anti-VM can be bypassed by setting the EBX register value to 0x0 after the *IN* instruction executes as shown in Fig.6.

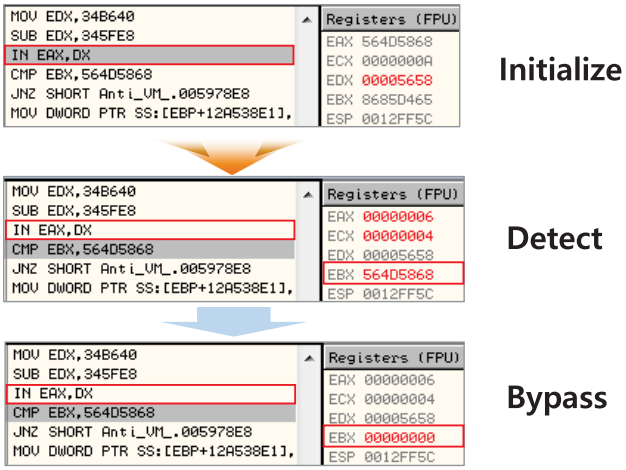

FIGURE 6. Bypassing a detection technique through modification of the register (*IN*2 instruction).

B. ENIGMA

Companies still operate and manage Enigma as a commercial protector, which provides anti-VM but not anti-DBI. In this paper, we performed our analysis on the Enigma version 6.00.

1) ANTI-VM OF ENIGMA

Enigma provides an anti-VM option to other environments besides VMware and VirtualBox as shown in Fig.7 Because the target of this paper is either VMware or VirtualBox, we did not perform any analysis for other environments. Despite this, we made sure to verify all options before performing the experiment. Among the anti-VM options provided by Enigma, there are three virtual environment detection techniques used in both VMware and VirtualBox. The three options are VMware, VirtualBox, and Hyper-V (CPU feature enabled). Each anti-VM option detection message box is shown in Fig.8. The Hyper-V (CPU feature enabled) option can detect both VMware and VirtualBox through the CPU information of the virtual environment.

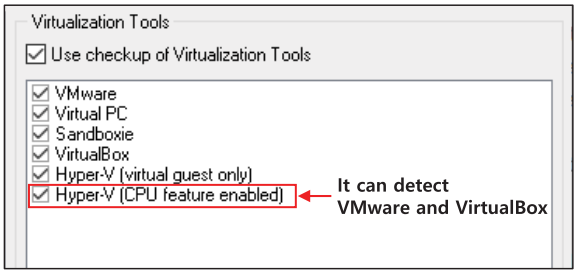

FIGURE 7. Anti-VM option provided by Enigma.

a: DETECTION AND BYPASSING USING IN INSTRUCTION

Programs applying Enigma detect VMware using the *IN* instruction. Similarly, Enigma uses the Themida 2nd *IN* instruction method, which executes 0xA in ECX register.

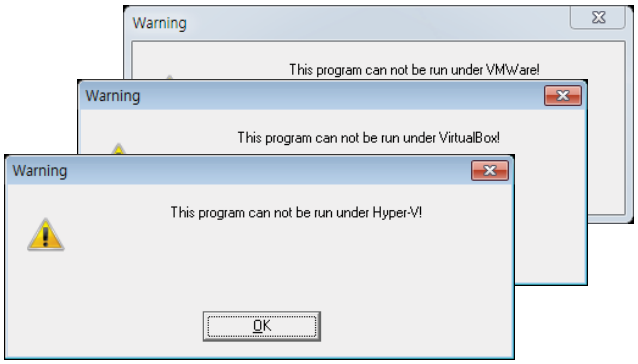

FIGURE 8. A pop-up message box that appears when executing a program applying Enigma's anti-VM option in a virtual environment.

Therefore, the bypass method is identical to that used in Themida.

b: DETECTION USING RUNNING PROCESS

Enigma detects VirtualBox by detecting the use of *VBoxService.exe*, an additional program provided by VirtualBox, which is a file created by installing VirtualBox Guest Additions. By installing VirtualBox Guest Additions, VirtualBox users can conveniently adjust the window size and resolution. Therefore, most users who use VirtualBox will also install and use VirtualBox Guest Additions.

c: BYPASSING DETECTION USING A RUNNING PROCESS

To detect *VBoxService.exe*, Enigma uses *processs32next API* to obtain information on currently running processes, and this information is stored in Unicode in the memory. After this, Enigma uses *WideCharToMultiByte API* to compare against the predefined *VBoxSevice.exe* ASCII value. Note that this API converts Unicode values stored in the memory to ASCII. After that, it performs a string comparison using the process name changed to ASCII and performs detection. As shown in Fig.9, the analyst can modify the process name value in ASCII format stored in the memory, or they can bypass it by modifying the comparison routine.

d: DETECTION USING CPUID INSTRUCTION

The *CPUID* instruction returns various processor information, such as the serial number and manufacturer ID according to the EAX register value. Since most guest OS run on the hypervisor, processor information that is distinct from the host OS appears. Enigma uses these differences to detect virtual environments.

e: BYPASSING DETECTION USING CPUID INSTRUCTION

If 0x1 is put in EAX and the *CPUID* instruction is executed, information about the model and type of the processor is returned. After executing the *CPUID* instruction, the 31st bit of the ECX register indicates the existence of a hypervisor. Therefore, by using this bit, the host OS and the guest OS can be differentiated. If the bit is 1, it is deemed to be the

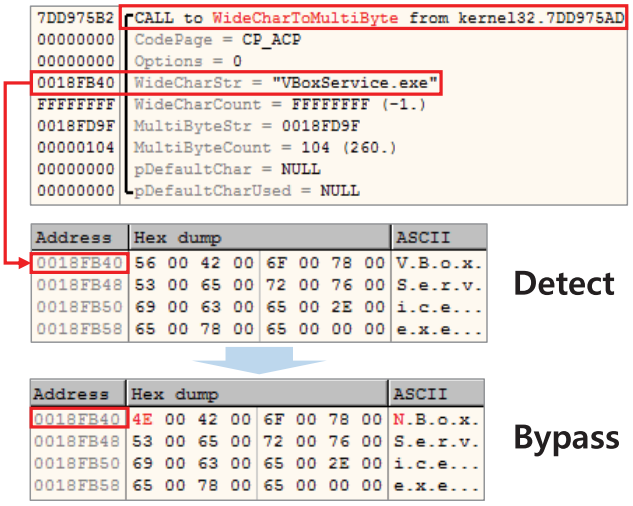

FIGURE 9. Bypassing a detection technique through modification of the *WidecharToMultibyte API* (*VBoxService.exe*).

guest OS CPU, and Enigma determines that it is a virtual environment. To bypass the virtual environment detection using the *CPUID* instruction, the anti-VM can be bypassed by changing it to 0 through an eXclusive OR (XOR) operation when the 31st bit of the ECX register is set to 1 as shown in Fig.10.

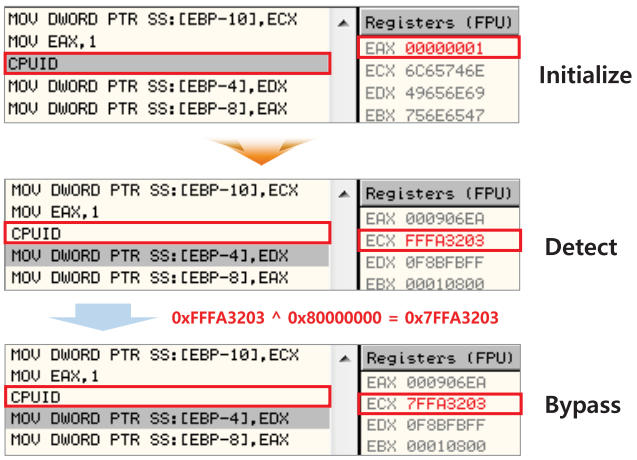

FIGURE 10. Bypassing a detection technique through modification of the register (*CPUID* instruction).

C. VMProtect

VMProtect is a powerful protector for virtualization obfuscation that provides various functions. In this paper, we analyzed VMProtect version 3.0.9, which includes both anti-VM and anti-DBI techniques.

1) ANTI-VM OF VMProtect

VMProtect uses three types of anti-VM techniques to detect virtual environments. When detecting a virtual environment, a message box like that shown in Fig.11 appears and the program is terminated.

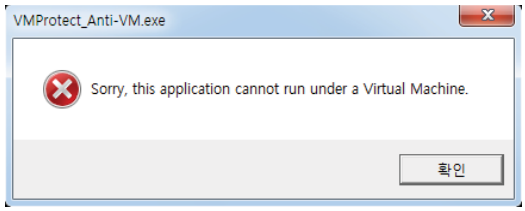

FIGURE 11. A pop-up message box that appears when executing a program applying VMProtect's anti-VM option in a virtual environment.

a: DETECTION USING FIRMWARE TABLE

VMProtect detects a virtual machine environment via a string match routine from the firmware table information obtained using *GetSystemFirmwareTable API*.

b: BYPASSING DETECTION USING THE FIRMWARE TABLE

The program applying VMProtect's anti-VM option has a routine to string match VMware and VirtualBox strings using information in the firmware table. VMware is detected by comparing the information of the firmware table by 1 byte sequentially from the front of 0x564D7761 (i.e., VMwa), and VirtualBox is detected by comparing 1 byte sequentially from the front of 0x56697274 (i.e., Virt). As shown in Fig.12, the routines that VMProtect detects using firmware table information are notably unique, such as *CMP BYTE PTR DS:[EDX],* 0x56. Therefore, the analyst can locate a routine to check the firmware table using a unique comparison routine instruction. To bypass this, the value stored in the memory can be altered as shown in Fig.12 since the first byte of both VMware and VirtualBox is 0x56.

c: DETECTION AND BYPASSING USING CPUID INSTRUCTION

Like Enigma, VMProtect puts 0x1 in the EAX register when executing *CPUID* and detects that it is a virtual environment with the 1 and 0 of the 31st bit of ECX after execution. The bypass method is identical to the one described in the Enigma portion of this paper.

The program did not operate normally when analyzed using PIN, and the anti-DBI was applied.

2) ANTI-DBI OF VMProtect

VMProtect does not have an anti-DBI option, but it is applied through an anti-debugging option. Anti-debugging is provided in two modes: the user mode and the user mode + kernel mode. For this paper, we conducted research on both modes. The anti-DBI technique provided by VMProtect is included in both modes. In addition, even when the anti-debugging option was not set and only the anti-VM option was set, it was confirmed that the anti-DBI effect also occurred. The analyst can see that the program does not work normally and terminates as shown in Fig.13 when using

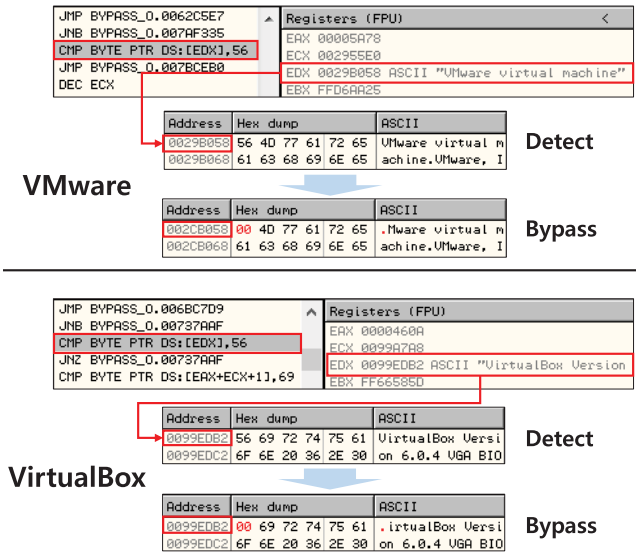

FIGURE 12. Bypassing a detection technique through modification of the memory (firmware table information).

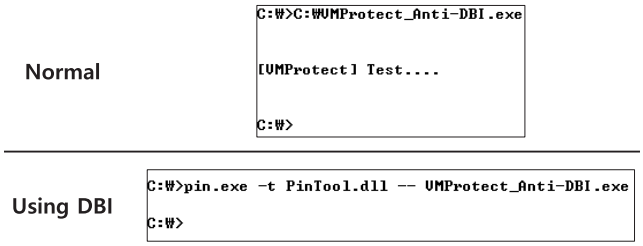

FIGURE 13. A program that has applied VMProtect anti-VM and anti-debugging options is terminated during analysis using DBI.

DBI for the programs to which the VMProtect anti-debugging and anti-VM are applied. There is one anti-DBI technique provided by VMProtect, but the bypass method of this technique is different depending on the anti-debugging and anti-VM options.

a: DETECTION USING SINGLE STEP (ANTI-VM OPTION)

In VMProtect's anti-VM option, a technique that forcibly terminates DBI uses a single step exception handling technique. Single step is a technique that forcibly handles exceptions using the trap flag, which is activated by setting 0x100 in the *EFLAGS* register. Therefore, if 0x100 is put on the stack and the *POPFD* instruction is executed, exception handling occurs. In a normal program, exception processing occurs by single stepping, but exception processing does not occur in the state of analysis using a debugger or a DBI tool. Therefore, programs not being analyzed and using DBI work normally when the two anti-VM techniques are bypassed in a virtual environment, but those being analyzed using a DBI tool are abnormally terminated by single step.

b: BYPASSING DETECTION USING SINGLE STEP (ANTI-VM OPTION)

In order to bypass the anti-DBI technique in the anti-VM option, the program operates normally by performing XOR after checking whether or not the stack address is a trap flag (0x100) before the *POPFD* instruction, as shown in Fig.14.

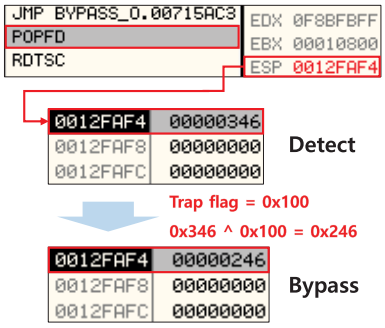

FIGURE 14. Bypassing a detection technique by a trap flag XOR operation (single step (anti-VM)).

c: DETECTION USING SINGLE STEP (ANTI-DEBUGGING OPTION)

This technique is the same as the single step technique applied in the VMProtect anti-VM option. Though the detection process is similar with that used in anti-VM, there are differences in the bypassing process.

d: BYPASSING DETECTION USING SINGLE STEP (ANTI-DEBUGGING OPTION)

With single step, exception handling occurs during program execution. However, with debugging or DBI, it is executed normally without any exception handling. This means that single step can be used to detect the DBI tool by using this principle. When using a DBI tool, exception handling does not occur and the program executes, resulting in abnormal termination of the program, thus revealing the DBI tools and debuggers. Therefore, if a trap flag is set in the execution timing of the *POPFD* instruction, as shown in Fig.15, it is possible to bypass the program, thus allowing the programmer to operate it as intended by forcing exception handling using the API provided by the DBI tool.

D. OBSIDIUM

Obsidium, like previously developed tools, is a protector that continues to release new versions. In this paper, we analyzed Obsidium version 1.6.7, which provides both anti-VM and anti-DBI techniques.

1) ANTI-VM OPTION OF OBSIDIUM

The anti-VM technique provided by Obsidium includes three types of techniques to detect virtual environments, during which a message box like in Fig.16 appears and the program is terminated.

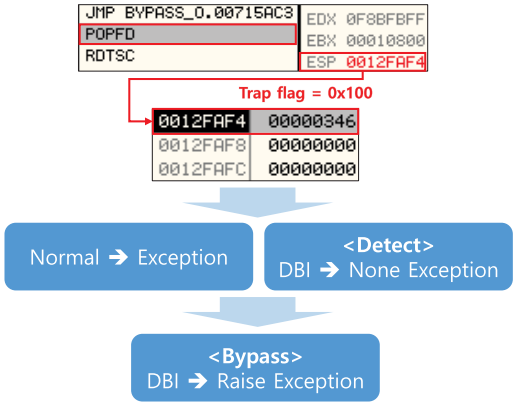

FIGURE 15. Bypassing a detection technique by raise exception (VMProtect single step (anti-debugging)).

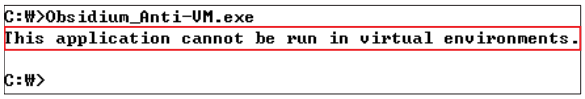

FIGURE 16. A program applying Obsidium's anti-VM option is terminated in a virtual environment.

a: DETECTION AND BYPASSING USING IN INSTRUCTION

Programs applying Obsidium's anti-VM option detect the VMware environment by using a feature that keeps the VMXh unique value in the EBX register when *IN* instruction is used in VMware. This technique is the same as the second *IN* instruction technique used in Themida. As such, the bypass method is also identical to that used in Themida.

b: DETECTION USING DISK DRIVE VALUE

The hardware contains various information such as model names and serial numbers. Since the guest OS uses virtual hardware, different features from the host OS environment appear. Obsidium performs detection using the disk drive model name information. The disk drive model names of VMware and VirtualBox include the VMware and VBOX strings, which Obsidium uses to detect virtual environments.

c: BYPASSING DETECTION USING DISK DRIVE VALUE

As in Fig.17, we found a routine in the program with Obsidium and checked the virtual environment using the disk drive model name.

In fact, this program has a routine to string match VMware and VirtualBox strings from the disk drive model name. Note that the first letter of both VMware and VBOX is 0x56 (i.e., V). Therefore, analysts can see that there is a comparison routine such as *CMP BYTE PTR DS: [EAX],* 0x56 as shown in Fig.17. It is possible to bypass the anti-VM technique by modifying the memory data in the EAX register value before execution or by modifying the instruction to a value other than 0x56.

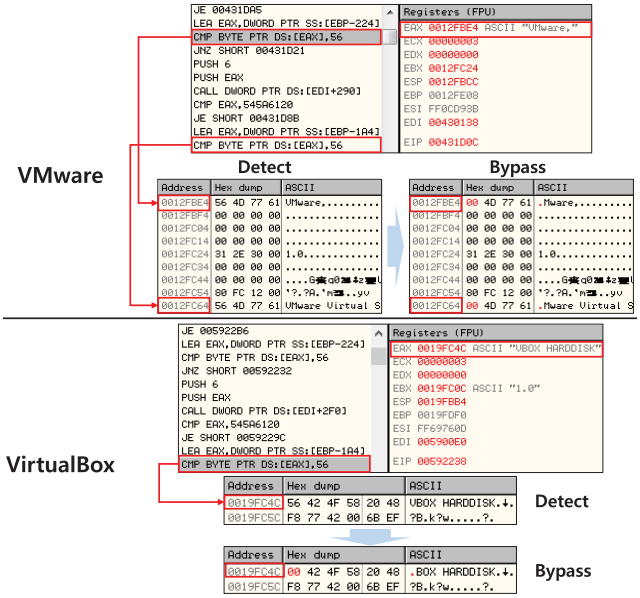

FIGURE 17. Bypassing a detection technique through modification of the memory (disk drive value).

d: DETECTION USING VBoxGuest FILE (ANTI-VM OPTION)

Obsidium detects the existence of the *VBoxGuest* name file in order to detect the VirtualBox. *VBoxGuest* files are created by installing VirtualBox Guest Additions in VirtualBox. Regardless of whether it is VirtualBox or not, if VirtualBox Guest Additions is not installed, it is certain that the*VBoxGuest* file does not exist, and as such, the virtual environment cannot be detected by the corresponding technique.

e: BYPASSING DETECTION USING VBoxGuest FILE (ANTI-VM OPTION)

In Obsidium, *CreateFileW API* is used to check the existence of the *VBoxGuest* file. As shown in Fig.18, the program inserts \\.*VBoxGuest* as the file name argument value and *OPEN_EXISTING* as the mode argument value in *CreateFileW API*.

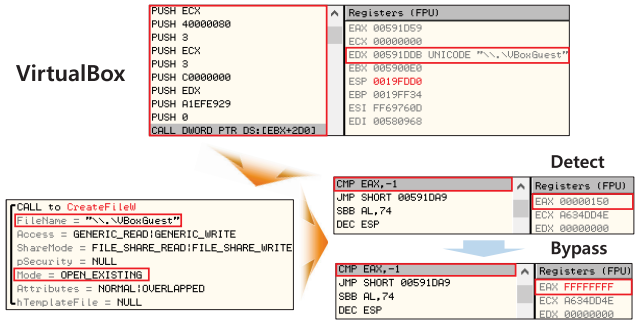

FIGURE 18. Bypassing a detection technique through modification of the register (*VBoxGuest* file).

If the return value of the EAX register is 0xFFFFFFFF (−1) after executing the API, it means that the *VBoxGuest*

file does not exist, and it is not a VirtualBox. Conversely, if the return value of the EAX register returns with a value other than 0xFFFFFFFF, it means that the*VBoxGuest* file exists, and it is a VirtualBox. Therefore, it is possible to bypass the anti-VM technique by changing the EAX register value to 0xFFFFFFFF in the comparison routine *CMP EAX, 0xFFFFFFFF* immediately after *CreateFileW API*.

f: DETECTION USING SMBIOS INFORMATION ON THE FIRMWARE TABLE

System Management BIOS (SMBIOS) is a standard for data structures that is used to read information stored in the BIOS on a computer. It contains a variety of information, such as BIOS information and system information. The guest OS has BIOS and system features that are distinct from the host OS, which makes it detectable using these features.

g: BYPASSING DETECTION USING SMBIOS INFORMATION FROM THE FIRMWARE TABLE

Obsidium uses *GetSystemFirmwareTable API* to obtain information about SMBIOS. In the information obtained through the API in the virtual environment is system information, including VMware and VirtualBox strings. In Obsidium, string matching is performed on the obtained system information string. First, check the first character 0x56 (i.e., V) through the *CMP BYTE PTR [ESI],* 0x56 instruction as shown in Fig.19. As shown in Fig.19, the first character 0x56 (i.e., V) is checked through the instruction *CMP BYTE PTR [ESI],* 0x56 and then the routine to check the rest of the string is executed. After that, a virtual environment is detected using the signature stored in the EAX register (*CMP EAX, 0x117A8875*(=signature)), which is calculated through the operation in advance. Signatures are configured differently for each virtual environment.

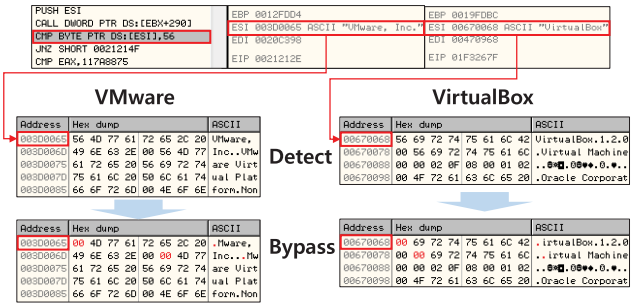

FIGURE 19. Bypassing a detection technique through modification of the memory (firmware table information).

In the instruction to check the first character, the ESI register value as the address has a value of 0x564D7761 (i.e., VMwa) in the case of VMware and 0x56697274 (i.e., Virt) in the case of VirtualBox. Therefore, to bypass this, when the first character check routine is performed, the ESI register value as the address is checked to see if it has a value of 0x564D7761 (i.e., VMwa) or 0x56697274 (i.e., Virt). If it indeed has that value, it can be bypassed by modifying 0x56, the first character in the memory, to 0x0.

2) ANTI-DBI OF OBSIDIUM

Obsidium's anti-DBI technique is available when the anti-debugging option is applied. There are three type of anti-DBI techniques used in Obsidium. If an analyst attempts to perform an analysis of the program applying Obsidium's anti-DBI technique using a DBI tool, the program will terminate as shown in Fig.20.

```
C:₩>pin.exe -t PinTool.dll -- Obsidium_Anti-DBI.exe
Debugger detected - please disable it and restart the application.

C:₩>
```

FIGURE 20. A program applying the Obsidium anti-debugging option is terminated during analysis using DBI.

a: DETECTION USING ZwQueryInformationProcess API

ZwQueryInformationProcess API is an API that allows you to search for information in the process, and it can also detect debugging. If an analyst enters 0x7 (ProcessDebugPort) as the second argument to this API, the address value of the third argument returns a value indicating whether or not the debugger is running. If debugging is in progress, 0xFFFFFFFF (−1) is returned. This technique detects debugging through *ZwQueryInformationProcess API* as documented in MSDN. However, there is an additional undocumented debugging detection option. If an analyst calls the API with 0x1F as the second argument, the analyst can check whether debugging is being performed. If 0 is returned for the third address value, debugging is in progress; if 1 is returned, debugging is not in progress. The method of detecting by way of positioning the documented 0x7 as an argument does not detect DBI tools, but if 0x1F is inserted, the API can detect them.

b: BYPASSING DETECTION USING ZwQueryInformationProcess API

Obsidium detects DBI by inserting 0x1F in *ZwQueryInformationProcess API*. Generally speaking, to bypass this, the DBI tool is used to track the *ZwQueryInformationProcess API*. If the second argument is 0x1F, it can be bypassed by changing the value stored in the third address to 1 after API execution. However, in order to inhibit analysis in programs applying Obsidium, some APIs are not directly called. So even after using the DBI tool to trace all the *ZwQueryInformationProcess API*s used in the program, it cannot be traced well. Therefore, it is necessary to locate the *ZwQueryInformationProcess API* that performs the anti-DBI technique via a feature of argument values before calling *ZwQueryInformationProcess API*. As shown in Fig.21, the nearest call is found using the argument value as a feature. As such, if the value stored in the third address value after call is 0, it can be bypassed by changing it to 1.

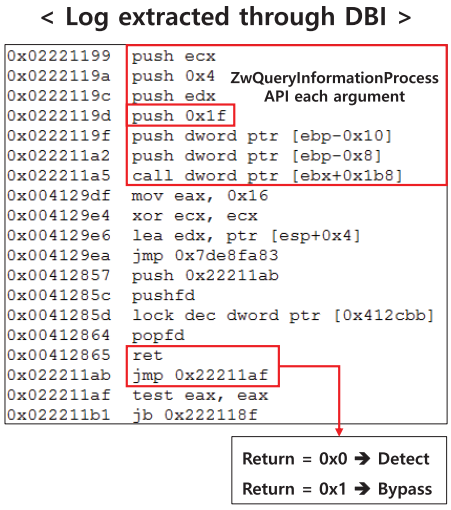

FIGURE 21. Bypassing a detection technique by modifying the return value (*ZwQueryInformationProcess(0x1F) API*).

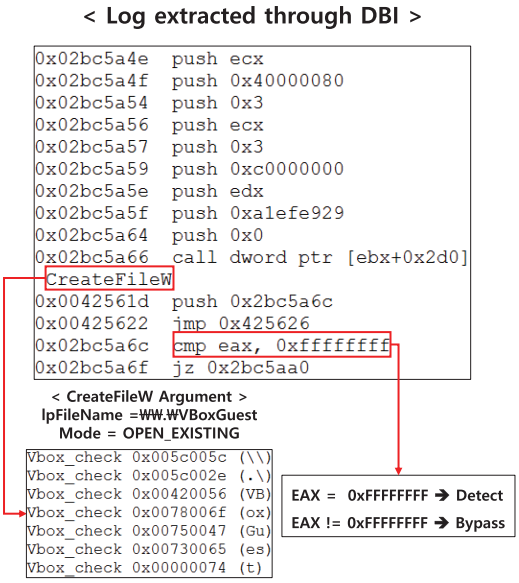

FIGURE 22. Bypassing a detection technique by modifying the return value (*VBoxGuest* file).

c: DETECTION USING VBoxGuest FILE (ANTI-DEBUGGING OPTION)

If the *VBoxGuest* file does not exist while the program that has applied Obsidium is being analyzed through DBI, the process is forcibly terminated as shown in Fig.20. If the program is not being analyzed, it executes normally. However, when the program is being analyzed using DBI, the DBI is detected using the existence of the *VBoxGuest* file. Therefore, when the *VBoxGuest* file is installed in the VirtualBox environment, it can be analyzed without being detected if the program is analyzed using DBI. The difference from *VBoxGuest* detection in the anti-VM option is that, in the anti-debugging option, the *VBoxGuest* file must exist to bypass it.

d: BYPASSING DETECTION USING VBoxGuest FILE (ANTI-DEBUGGING OPTION)

In order to bypass DBI detection using *VBoxGuest*, the analyst must do the opposite of what is necessary for the anti-VM option. When the *CreateFileW API* is executed, the analyst checks the argument to verify whether or not *VBoxGuest* is set as the file name, and *OPEN_EXISTING* is set as the mode. After that, *CMP EAX, 0xFFFFFFF* instruction appears as shown in Fig.22. If the EAX register value is 0xFFFFFFFF, program debugging is detected, so the analyst can bypass the anti-DBI technique by changing 0xFFFFFFFF to another value.

e: DETECTION USING SINGLE STEP

Obsidium's anti-debugging option uses a single step technique to forcefully terminate the program during analysis using DBI. This single step technique is performed sporadically and not every time the program is executed, which means that sometimes the analysis of the program can be completed using DBI without bypassing the single step. However, cycles in which single step techniques appear are quite frequent and must be bypassed.

f: BYPASSING DETECTION USING SINGLE STEP

To bypass a single step, the *AND EAX,* 0x7 instruction is performed after the *CALL DWORD PTR [EBX+0x12C]* instruction as shown in Fig.23. After executing the *AND EAX,* 0x7 instruction, we confirmed that various values were entered in the EAX register. The single step routine proceeds only when the EAX register is 0x5 or 0x2. Therefore, bypass is possible by changing the EAX register to a value other than 0x5 or 0x2 after executing the *AND EAX,* 0x7 instruction.

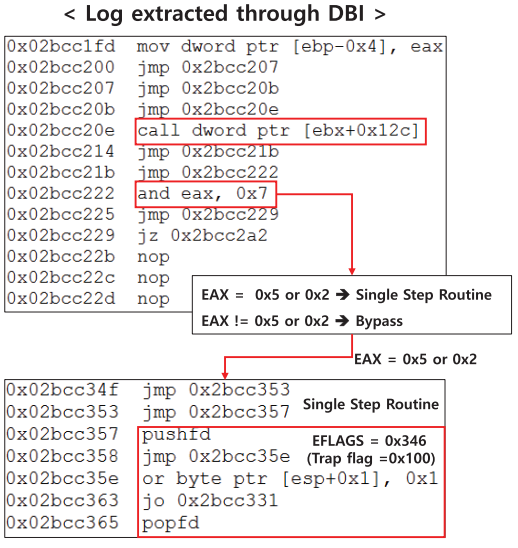

FIGURE 23. Bypassing a detection technique through modification of the register (*single step*).

E. ACProtect

ACProtect, unlike other tools, is a protector that does not continue to release new versions. In this paper, we analyzed the ACProtect version 2.0, which offers a variety of options like anti-debugging. However, ACProtect does not provide an anti-VM option.

1) ANTI-DBI OF ACProtect

Unlike previously developed protectors, even if the anti-debugging option is not applied in ACProtect, the anti-DBI technique is applied. ACProtect has one anti-DBI technique, and when the ACProtect-applied program is analyzed by a DBI tool, the program does not operate normally and is terminated as shown in Fig.24.

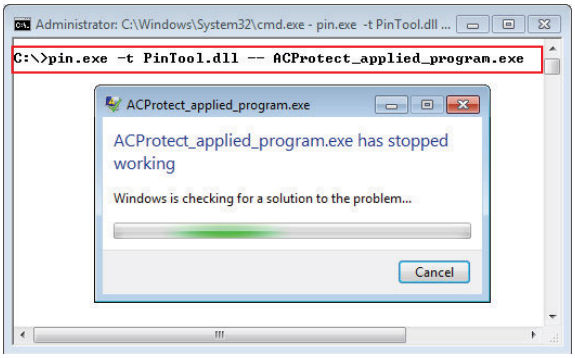

FIGURE 24. A program applying ACProtect is terminated during analysis using DBI.

a: DETECTION USING SELF-MODIFICATION CODE

ACProtect uses self-modification code to detect DBI tools. This code modifies its own instruction code while the program is running. As shown in Fig.25, this poses no problems for normal program execution, but when an analyst executes a program with DBI tool, the program terminates. This is because DBI tools use a just-in-time (JIT) compiler to compile and run programs in real time. When a DBI tool executes, instructions for programs of a certain size are put in the cache, and the program is executed through real-time JIT compilation. When self-modification is applied, the instruction appropriately self-modifies during program execution. If the modified code is placed in DBI cache before modification, the modified code is not applied. Therefore, when using a DBI tool, the instruction is executed before modification so that the program does not execute, indicating that, as a result, the DBI tool can be detected using this principle.

b: BYPASSING DETECTION USING SELF-MODIFICATION CODE

In Fig.25, the *MOV DWORD PTR SS: [EBP+F], EAX* instruction is a memory write instruction. If the instruction to write the memory is executed, *MOV EAX, 5A* is changed to *MOV EAX, ACP_NONE.0041B462*. However, when a program is executed using a DBI tool, a certain amount of

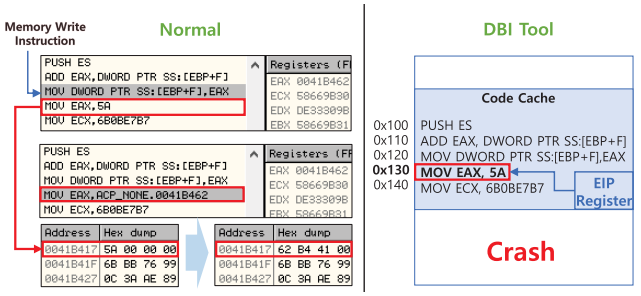

FIGURE 25. A program applying ACProtect crashes due to self-modification during analysis using DBI.

instructions are put into the cache and executed line by line as shown in Fig.25. When the memory write instruction *MOV DWORD PTR SS: [EBP+F], EAX* is executed in DBI cache, the memory area of the program is altered, but not the instruction in the DBI cache. Therefore, the DBI tool causes a crash by executing an instruction that is not self-modifying. To bypass this, the analyst should check the instructions that write the memory region in the DBI cache. Once this is located, the analyst should check if the instruction to write the memory modifies the instructions in the DBI cache. If the instruction to be modified is an instruction included in the DBI cache, it can be bypassed by clearing the cache after executing the instruction to write memory and collect the changed instruction again.

V. ANTI-ANTI VM & ANTI-ANTI DBI OF COMMERCIAL PROTECTORS

This section presents an algorithm that can bypass anti-VM and anti-DBI techniques for each tool by using a DBI tool based on the preceding analysis.

In all algorithms, the current instruction (*curINS*), the current instruction address (*curAddr*), and the current API routine (*curRTN*) are entered as arguments. Input options *curINS*,*curRTN*, or *curAddr* helps the algorithm determine whether or not an anti-analysis technique is applied based on which is received.

A. ANTI-ANTI-VM OF THEMIDA

Themida detects the virtual environment using two types of techniques. The algorithm to bypass Themida's anti-VM option using a DBI tool is shown as Alg. 1.

The program applying Themida's anti-VM technique detects a virtual environment using the *IN* instruction. If the EDX register contains value 0x5658 (VX) in the execution order of the *IN* instruction, it means that information about the virtual machine is obtained from the port. The algorithm then removes the *IN* instruction to bypass the anti-VM technique and checks if the value of the ECX register is 0x14 or 0xA. If the ECX register value is 0x14, the EAX register changes to 0. Also, if the ECX register is 0xA, the EBX register value changes to 0. By changing the EAX and EBX register

Algorithm 1 Anti-Anti-VM Algorithm for Themida

Input: Current Instruction (curINS), Current API (curRTN)

```
if (curINS = IN_ins) AND (EDX = "VX") then
    INS_Delete(curINS)
    if ECX = Memsize_Request (= 0x14) then
        EAX ← 0
    else if ECX = Version_Request (= 0xA) then
        EBX ← 0
    end if
end if
if curRTN = memmove_API then
    if Src = ("VMware_SVGA_3D" OR "Oracle_VM") then
        Src ← 0
    end if
end if
if curRTN = RegOpenKeyA_API then
    if hKey = HKLM then
        if Subkey = "VBOX__" then
            Subkey ← 0
        end if
    end if
end if
```

Algorithm 2 Anti-Anti-VM Algorithm for Enigma

Input: Current Instruction (curINS), Current API (curRTN)

```
CPUID_Flag ← False
GuestOS_Sig ← 0x80000000
if CPUID_Flag = True then
    if (ECX & GuestOS_Sig) ≠ 0 then
        ECX ← (ECX xor GuestOS_Sig)
    end if
    CPUID_Flag ← False
end if
if curINS = CPUID_ins then
    CPUID_Flag ← True
end if
if (curINS = IN_ins) AND (EDX = "VX") then
    INS_Delete(curINS)
    if ECX = Version_Request (= 0xA) then
        EBX ← 0
    end if
end if
if curRTN = WideCharToMultiByte_API then
    if WideCharStr = "VBoxService.exe" then
        WideCharStr ← 0
    end if
end if
```

values to 0, the analyst can bypass the anti-VM using the *IN* instruction.

The program applying the Themida anti-VM option detects a virtual environment using registry values with an algorithm that saves each signature for three special registry values used by Themida. The algorithm checks for the existence of *VMware SVGA 3D* and *Oracle VM* through *memmove API* tracking, and if it exists, it changes to 0. To check the VBOX registry key of Virtualbox, the algorithm traces *RegOpenKey_A API*. When opening HKLM (HKEY_LOCAL_MACHINE), if the *VBOX__* is included, it is possible to bypass anti-VM by changing the value to 0.

B. ANTI-ANTI-VM OF ENIGMA

Enigma detects a virtual environment using two types of techniques. The algorithm to bypass Enigma's anti-VM option using a DBI tool is shown as Alg. 2.

The program applying Enigma's anti-VM option detects a virtual environment using the *CPUID* instruction, which is tracked by the algorithm. After executing the *CPUID* instruction to check whether the 31st bit of the ECX register is set to 1, the algorithm performs an AND operation of 0x80000000 (GuestOS_Sig) to the value of the ECX register. If the 31st bit is set to 1, a bypass is possible by changing the bit value to 0 through an XOR operation.

The program applying to Enigma's anti-VM option uses the *IN* instruction to detect a virtual environment. The difference from Themida is that only the *IN* instruction is used when the ECX register is 0xA. The algorithm traces the *IN* instruction and removes it if the value of the EDX register is 0 x 5658 (VX). If the ECX register is 0xA, an analyst can bypass the anti-VM using the *IN* instruction by changing the EBX register value to 0.

Enigma checks whether or not *VBoxSevices.exe* has been executed to detect VirtualBox, but the string match determines whether or not *VBoxSevices.exe* executes. In a program to which Enigma is applied, *WideCharToMultiByte API* must be used to match string format, which allows a string match of a *VBoxSevices.exe* string to be performed. Therefore, when changing the string format of *VBoxSevices.exe* by tracking *WideCharMultiByte API*, the anti-VM option can be bypassed by changing the string to 0.

C. ANTI-ANTI-VM OF VMProtect

VMProtect detects virtual environments using two types of techniques. In order to bypass VMProtect's anti-VM option using DBI, one of the anti-DBI techniques (single step) needs to be bypassed. The algorithm used to bypass using a DBI tool in this case is shown as Alg. 3.

The program applying the VMProtect anti-VM option uses the *CPUID* instruction to detect a virtual environment. The bypass method for the *CPUID* instruction is the same as the method used in Enigma.

The program applying the VMProtect anti-VM option uses firmware table information to detect a virtual environment. In the firmware table information, the string "VMware" exists for VMware and the string "Virtual" exists for VirtualBox. There is a unique instruction called *CMP BYTE PTR DS:[EDX],* 0x56 that is used for virtual environment

Algorithm 3 Anti-Anti-VM Algorithm for VMProtect

Input: Current Instruction (curINS), Current API(curRTN)

VM_CheckRoutine ← "CMP BYTE PTR DS:[EDX], 0x56"
CPUID_Flag ← *False*
GuestOS_Sig ← 0x80000000
if *CPUID_Flag* = *True* **then**
  if (*ECX* & *GuestOS_Sig*) ≠ 0 **then**
    ECX ← (*ECX* **xor** *GuestOS_Sig*)
  end if
  CPUID_Flag ← *False*
end if
if *curINS* = *CPUID_ins* **then**
  CPUID_Flag ← *True*
end if
if *curINS* = *VM_CheckRoutine* **then**
  if ∗*EDX* = ("*VMware*" OR "*Virtual*") **then**
    ∗*EDX* ← 0
  end if
end if
if *curINS* = *POPFD_ins* **then**
  if (∗*ESP* & *TrapFlag*(= 0x100)) ≠ 0 **then**
    ∗*ESP* ← (∗*ESP* **xor** *TrapFlag*(= 0x100))
  end if
end if

Algorithm 4 Anti-Anti-DBI Algorithm for VMProtect

Input: Current Instruction (curINS), Current API (curRTN)

if *curINS* = *POPFD_ins* **then**
  if (∗*ESP* & *TrapFlag*(= 0x100)) ≠ 0 **then**
    DBI_RaiseException(*EIP* + 1, *ESP* + 1))
  end if
end if

detection with the corresponding firmware table information. With this, a *CMP BYTE PTR DS:[EDX],* 0x56 instruction trace is performed and the value is checked against the value of the EDX register as an address before the instruction is executed. If a the string "VMware" or "Virtual" value exists, it can be bypassed by changing the value to 0.

As described above, an anti-DBI technique must be bypassed to circumvent the anti-VM option. To bypass a single step, the*POPFD* instruction must be traced. Before executing the *POPFD* instruction, the analyst should check whether 0x100 (trap flag) is set in ESP. If 0x100 (trap flag) exists, it can be bypassed by removing the trap flag through an XOR operation.

D. ANTI-ANTI-DBI OF VMProtect

There is only one technique that can detect DBI tools in VMProtect. The algorithm that bypasses VMProtect's anti-debugging option using a DBI tool is shown as Alg. 4.

In order to bypass the anti-DBI technique provided by VMProtect's anti-debugging option, an analyst must bypass

Algorithm 5 Anti-Anti-VM Algorithm for Obsidium

Input: Current Instruction (curINS), Current API (curRTN)

VM_Diskdirve ← "CMP BYTE PTR [EAX], 0x56"
VM_firmware ← "CMP BYTE PTR [ESI], 0x56"
VGuest_Cmp ← "CMP EAX, 0xFFFFFFFF"
VGuest_Flag ← *False*
if (*curINS* = *IN_ins*) AND (*EDX* = "*VX*") **then**
  INS_Delete(*curINS*)
  if *ECX* = *Version_Request* (= 0x*A*) **then**
    EBX ← 0
  end if
end if
if *curINS* = *VM_Diskdirve* **then**
  if ∗*EAX* = ("*VMware*" OR "*VBOX*") **then**
    ∗*EAX* ← 0
  end if
end if
if (*curINS* = *VGuest_Cmp*) AND *VGuest_Flag* **then**
  if *EAX* ≠ 0xFFFFFFF **then**
    EAX ← 0xFFFFFFF
    VGuest_Flag ← *False*
  end if
end if
if *curRTN* = *CreateFileW_API* **then**
  if *FileName* = \\.*VBoxGuest* **then**
    if *Mode* = *OPEN_EXISTING* **then**
      VGuest_Flag ← *True*
    end if
  end if
end if
if *curINS* = *VM_firmware* **then**
  if ∗*ESI* = ("*VMware*" OR "*Virtual*") **then**
    ∗*ESI* ← 0
  end if
end if

the single step. There is a slight difference from the method used in the anti-VM option of VMProtect. The algorithm performs a *POPFD* instruction trace and checks whether 0x100 (trap flag) is set in ESP before executing the *POPFD* instruction. If 0x100 (trap flag) is set, a bypass is possible by forcibly making an exception using the exception API provided via the DBI tool.

E. ANTI-ANTI-VM OF OBSIDIUM

Obsidium detects a virtual environment using three types of techniques. The algorithm that bypasses Obsidium's anti-VM option using a DBI tool is shown as Alg. 5.

The program applying Obsidium's anti-VM option uses the *IN* instruction to detect virtual environments. The *IN* instruction technology used in Obsidium is the same as the technique that inserts 0xA into the ECX register and executes it in Enigma. The bypass method is similar to that of the previously discussed method for Enigma.

The program applying the Obsidium anti-VM technique detects the virtual environment using disk drive information. In the disk drive information, the string ''VMware'' exists for VMware and the string ''VBOX'' exists for VirtualBox. There is a unique detection instruction called *CMP BYTE PTR DS:[EAX],* 0x56, which uses the corresponding disk drive information. Therefore, the *CMP BYTE PTR DS:[EAX],* 0x56 instruction trace is performed, and the value is checked against the value of the EAX register as an address before the instruction is executed. If the string ''VMware'' or ''VBOX'' value exists, it can be bypassed by changing the value to 0.

The program applying the Obsidium's anti-VM option detects VirtualBox by the existence of *VBoxGuest* files. *CreateFileW API* is executed in order to determine the existence of a *VBoxGuest* file, and when executing the *CreateFileW API*, \\.*VBoxGuest* exists in the *FileName* value. After executing the *CreateFileW API*, it is determined whether or not the *VBoxGuest* file exists through the result value that is stored in the EAX register. If the file does not exist, 0xFFFFFFFF is stored in the EAX register, and if the file exists, a value other than 0xFFFFFFFF is stored. Therefore, to bypass this, an analyst must check if \\.*VBoxGuest* exists in the *FileName* value in the *CreateFileW API*. After that, the algorithm traces whether or not the *CMP EAX, 0xFFFFFFFF* instruction appears. If the EAX register is any value other than 0xFFFFFFFF, it can be bypassed by changing it to 0xFFFFFFFF.

The program applying the Obsidium anti-VM option uses the firmware table information to detect virtual environments. In the firmware table information, the string ''VMware'' exists for VMware and the string ''Virtual'' exists for VirtualBox. There is a unique instruction *CMP BYTE PTR DS:[ESI],* 0x56 that is used for virtual environment detection with the corresponding firmware table information. Therefore, a *CMP BYTE PTR DS:[ESI],* 0x56 instruction trace is performed, and the value is checked against the value of the ESI register as an address before the instruction is executed. If either the string ''VMware'' or ''Virtual'' value exists, it can be bypassed by changing the value to 0.

F. ANTI-ANTI-DBI OF OBSIDIUM

There are three types of techniques that can be used to detect DBI tools in Obsidium. The algorithm that bypasses Obsidium's anti-debugging option using a DBI tool is shown as Alg. 6.

In order to bypass the anti-DBI provided by Obsidium's anti-debugging option, an analyst must bypass the *ZwQueryInformationProcess(0x1F) API*. In general, the analyst can bypass the anti-DBI option by tracking the *ZwQueryInformationProcess(0x1F) API* using the API tracking function of the PIN and modifying the return value, but in the case of Obsdium, API tracking is not possible because the *ZwQueryInformationProcess(0x1F) API* is not called directly. Therefore, Obsidium's anti-DBI technique should be tracked and bypassed using the argument value pattern inserted when calling the *ZwQueryInformationProcess(0x1F) API* found through the analysis. Obsidium's anti-DBI technique sequentially executes instructions for *push* 0x4, *push EDX*, and *push 0x1F* in order to insert argument values and call *ZwQueryInformationProcess(0x1F) API*. In the algorithm, this series of processes is expressed as ZwQueryInformation_Argument_Check. After that, Obsidium's anti-DBI

Algorithm 6 Anti-Anti-DBI Algorithm for Obsidium

Input: Current Instruction (curINS), Current API (curRTN)

```
Debug_Check_Address ← 0
ZwQuery_Flag1 ← False
ZwQuery_Flag2 ← False
VGuest_Cmp ← "CMP EAX, 0xFFFFFFFF"
VGuest_Flag ← False
Single_Sig1 ← "CALL DWORD PTR [EBX+0x12C]"
Single_Sig2 ← "AND EAX, 0x7"
Single_Flag ← False
if ZwQuery_Flag2 then
  if *Debug_Check_Address = 0x0 then
    *Debug_Check_Address ← 0x1
  end if
end if
if (curINS = RET_ins) AND ZwQuery_Flag1 then
  ZwQuery_Flag2 ← True
end if
if ZwQueryInformation_Argument_Check then
  if curINS = CALL_ins then
    Debug_Check_Address ← *(ESP + 12)
    ZwQuery_Flag1 ← True
  end if
end if
if (curINS = VGuest_Cmp) AND VGuest_Flag then
  if EAX ≠ 0xFFFFFFFF then
    EAX ← 0xFFFFFFFF
    VGuest_Flag ← False
  end if
end if
if curRTN = CreateFileW_API then
  if FileName = \\.\VBoxGuest then
    if Mode = OPEN_EXISTING then
      VGuest_Flag ← True
    end if
  end if
end if
if (curINS = Single_Sig2) AND Single_Flag then
  if (EAX & 0x7) = (0x2 OR 0x5) then
    EAX ← 0x0
    Single_Flag ← False
  end if
end if
if curINS = Single_Sig1 then
  Single_Flag ← True
end if
```

Algorithm 7 Anti-Anti-DBI Algorithm for ACProtect

Input: Current Address (curAddr), Current Instruction (curINS)

```
DBI_Head ← TRACE_Address(trace)
DBI_Tail ← DBI_Head + TRACE_Size(trace)
FirstWritten ← 0x0
if INS_MemoryOperandIsWritten(curINS) then
    Write_addr ← MemoryWriteAddress
    if DBI_Head ≤ Write_addr < DBI_Tail then
        FirstWritten ← Write_addr
    end if
end if
if curAdd < FirstWritten ≤ curAdd + ins_size then
    FirstWritten ← 0x0
    DBI_Cache_Collection(curAdd)
end if
```

technique indirectly calls the *ZwQueryInformationProcess(0x1F) API* using the CALL instruction. Since these patterns appear, the algorithm tracks them to locate which part calls the *ZwQueryInformationProcess(0x1F) API*. The return value of the *ZwQueryInformationProcess(0x1F) API* is stored in value with the 3rd parameter value (ESP+12) as the address. Therefore, Alg. 6 stores the value of ESP +12 in the *Debug_Check_Address* variable in advance. When the *ZwQueryInformationProcess(0x1F) API* is terminated, the *RET* instruction is called. Therefore, after the *RET* instruction, if it is in a debugging environment, 0 is stored in the value addressed to *Debug_Check_Address* such that it can be bypassed when changed to 1.

Obsidium's anti-DBI option detects DBI using the existence of *VBoxGuest* files. The algorithm performs a trace of the *CreateFileW API* and checks if the \\.*VBoxGuest* is included as an argument. After that, the algorithm traces whether or not the *CMP EAX, 0xFFFFFFFF* instruction appears. If the EAX register is 0xFFFFFFFF, it can be bypassed by changing that value to anything other than 0xFFFFFFFF.

Obsidium's anti-DBI option uses a single step technique to detect DBI and then forcibly terminates the program. However, the single step routine does not appear every time a program is executed, and its appearance is dependent on the state of the EAX register after the *CMP DWORD PTR [EBX*+0 x *12C*] instruction and *AND EAX,* 0x7 instruction. If the EAX register is 0x2 or 0x5 after the *AND EAX,* 0x7 instruction, the single step routine proceeds so it can be bypassed by changing the EAX register to 0 x 0.

G. ANTI-ANTI-DBI OF ACProtect

There is one type of technique for ACProtect to detect DBI tools. The algorithm to bypass ACProtect's anti-DBI option using a DBI tool is shown as Alg. 7.

In order to bypass ACProtect's anti-DBI, self-modification must be bypassed. The algorithm stores the start address value and the last given value in the DBI cache. The algorithm stores which address was written when each instruction that writes memory is executed. If the newly written address value is included between the DBI cache start address value and the last address value, a bypass is possible through the DBI API that collects the cache again. Through this process, the instructions altered during execution are newly collected and executed in the DBI cache, so that the program operates normally.

VI. IMPLEMENTATION & EVALUATION

We implemented a tool using the DBI framework to experiment with the proposed algorithm as shown in Fig. 26. The input of the tool is malware, into which the anti-VM and anti-DBI techniques of commercial protectors are applied. When the malware goes through the three modules, it bypasses the anti-VM or anti-DBI option, and the program runs normally without interruption. Therefore, malware can be analyzed using DBI in a virtual environment.

The following describes the three aforementioned modules.

API Trace Module traces and bypasses the API used by anti-DBI and anti-VM techniques by using the API trace function of a DBI tool. This module traces *memmove*, *RegOpenKeyA*, *CreateFileW API*, and others to bypass anti-VM and anti-DBI techniques.

Instruction Trace Module tracks and bypasses instructions used by anti-DBI and anti-VM techniques using the instruction tracking function of a DBI tool. This module detects and bypasses special instructions that are used to detect virtual environments, such as *CPUID*, *IN* instructions.

Memory Write Check Module bypasses self-modification technique, which employs the JIT compiler characteristics among anti-DBI techniques. This module detects instructions that write memory, and then it checks whether the newly written memory address is included in the DBI cache. If it is included, DBI collects a new cache to bypass the anti-DBI technique.

For our evaluation, 1,573 test execution files were generated using Juliet Test Suite version 1.3 provided by the National Institute of Standards and Technology (NIST). The Juliet Test Suite code is categorized into a set of 118 security weaknesses based on common weakness enumeration (CWE). Each set has one or more flaw types that cause security weaknesses, and the Juliet Test Suite code contains 1,617 vulnerability types. Since Juliet Test Suite code can trigger vulnerabilities, it can also be used to create malware. Therefore, in this experiment, the experiment was performed assuming that the Juliet Test Suite code was malware.

The evaluation environment is as follows. The virtual environment used in the experiment was performed in VMware Workstation 15 Player and Oracle VirtualBox 6.0, both of which were configured in Windows 7. The experiment was performed using PIN version 2.7 of a DBI tool. Experiments were performed by applying anti-VM and anti-DBI techniques to 1,573 test portable executable (PE)

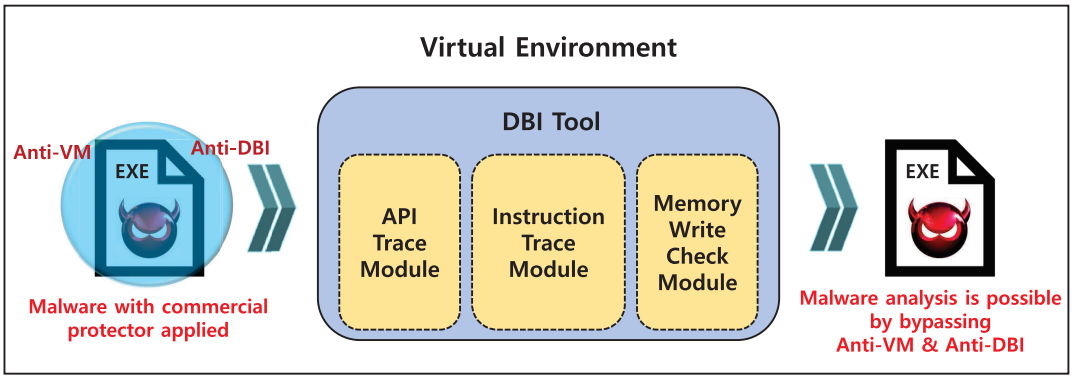

FIGURE 26. Overview of implementation.

files using five commercial protector tools (Themida 2.4.5, Enigma 6.00, VMProtect 3.0.9, Obsidium 1.6.7, and ACProtect 2.00). ACProtect does not provide any function that obfuscates large data sets automatically, and as such, no obfuscation options provided by ACProtect were applied to every test set. As a result, 30 test sets were manually generated and evaluated.

In terms of our experimental method, it was confirmed that the test set with an anti-VM technique in a virtual machine environment was bypassed using PIN, and the program executed successfully. We also performed an anti-anti-DBI experiment to check whether the test set with an anti-DBI technique was bypassed, and the program was successfully executed using PIN. The evaluation results are shown in Table 2. Test data sets obfuscated through each protector were all bypassed using the proposed algorithm. The experimental results indicate that if the malware uses an anti-VM technique and an anti-DBI technique from each commercial protector, the proposed algorithm makes it possible to bypass.

TABLE 2. Proposed anti-anti-VM & anti-anti-DBI algorithm evaluation.

Protector	Anti-anti-VM	Anti-anti-DBI
Themida	1,573 / 1,573 (100%)	N/A
Enigma	1,573 / 1,573 (100%)	N/A
VMProtect	1,573 / 1,573 (100%)	1,573 / 1,573 (100%)
Obsidium	1,573 / 1,573 (100%)	1,573 / 1,573 (100%)
ACProtect	N/A	30 / 30 (100%)

VII. DISCUSSION

At first glance, the method presented in this paper may appear to be applicable only to anti-DBI and anti-VM techniques for five commercial protectors. However, from the results of previous studies, the types of anti-DBI and anti-VM options are somewhat limited [5], [6], [12], [20], [21]. Branco, Rodrigo Rubira, Gabriel Negreira Barbosa, and Pedro Drimel Neto [20] mentioned that *IN* instruction is the most used anti-VM technique found in malware. Polino, Mario, *et al.* [12] mentioned that the Self-modification technique is the most used when investigating the anti-DBI technique applied to malware. We are presenting a bypass algorithm for both techniques. Therefore, by finely adjusting the bypass algorithm proposed in this paper, it will be possible to cover most cases. As such, we believe it would be inaccurate to claim that the general applicability of our proposal is insufficient in that the experiment was performed on five typical commercial protectors.

There are many options for any one commercial protector. In this paper, since only anti-VM and anti-DBI techniques were analyzed, it is difficult to present experimental results on whether the proposed bypass algorithm works properly even when other obfuscation options are applied. However, in the case of Themida, it is a structure in which unpacking is executed if it is not detected after the anti-analysis technique is performed, and these two parts can be considered to be separate [3]. Even when other protection techniques are applied, anti-analysis techniques are performed first and followed by other techniques. Therefore, even if other options are applied, deobfuscation should be applied after bypassing the anti-analysis technique, according to Suk *et al.* [3]. Since other commercial protectors have a similar structure, the bypass algorithm presented in this paper can work normally even when other obfuscation options are applied.

VIII. CONCLUSION

In order to analyze malware protected by commercial protectors in a virtual environment using DBI, we needed to additionally analyze anti-VM and anti-DBI techniques. We presented a detailed analysis and proposed bypass algorithms for anti-VM and anti-DBI techniques for commercial protectors in this paper, which is the first empirical study to propose detailed bypass algorithms of this capacity. The results of this study can serve as guidelines for easy analysis of malware protected by an anti-VM or an anti-DBI option supported by commercial protectors. In addition, other recent research has focused on unpacking DBI tools, and we believe that our bypass algorithm will aid in achieving higher success rates in future research. However, our paper

only addresses solutions for current techniques used in commercial protectors, and malware using customized protectors along with new versions of commercial protectors using anti-analysis techniques are emerging constantly. Considering these developments, anti-anti-analysis techniques must be studied further, which will form the basis of our future research.

REFERENCES

[1] S. D'Alessio and S. Mariani, "PinDemonium: A DBI-based generic unpacker for Windows executables," Black Hat USA, Tech. Rep., 2016.

[2] D. Reynaud and J.-Y. Marion, "Dynamic binary instrumentation for deobfuscation and unpacking," in *Proc. In-Depth Secur. Conf. Eur. (Deepsec)*, 2009.

[3] J. H. Suk, J.-Y. Lee, H. Jin, I. S. Kim, and D. H. Lee, "UnThemida: Commercial obfuscation technique analysis with a fully obfuscated program," *Softw., Pract. Exper.*, vol. 48, no. 12, pp. 2331–2349, Dec. 2018.

[4] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging Behavior in modern malware," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. FTCS DCC (DSN)*, 2008, pp. 177–186.

[5] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro, "SoK: Using dynamic binary instrumentation for security (And how you may get caught red Handed)," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Jul. 2019, pp. 15–27.

[6] P. Chen, C. Huygens, L. Desmet, and W. Joosen, "Advanced or not? A comparative study of the use of anti-debugging and anti-vm techniques in generic and targeted malware," in *IFIP Int. Conf. ICT Syst. Secur. Privacy Protection*. Springer, 2016, pp. 323–336.

[7] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Dept. Comput. Sci., Univ. Auckland, Auckland, New Zealand, Tech. Rep. 148, 1997.

[8] G.-R. Uh *et al.*, "Analyzing dynamic binary instrumentation overhead," in *Proc. WBIA Workshop ASPLOS*, 2006.

[9] *Pin*. Accessed: Aug. 13, 2020. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[10] *Dynamorio*. Accessed: Aug. 13, 2020. [Online]. Available: https://dynamorio.org/

[11] *Valgrind*. Accessed: Aug. 13, 2020. [Online]. Available: https://valgrind.org/

[12] M. Polino *et al.*, "Measuring and defeating anti-instrumentation-equipped malware," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, 2017, pp. 73–96.

[13] J. Park, Y.-H. Jang, S. Hong, and Y. Park, "Automatic detection and bypassing of anti-debugging techniques for microsoft windows environments," *Adv. Electr. Comput. Eng.*, vol. 19, no. 2, pp. 23–29, 2019.

[14] B. Cheng, J. Ming, J. Fu, G. Peng, T. Chen, X. Zhang, and J.-Y. Marion, "Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with Orders-of-Magnitude performance boost," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 395–411.

[15] D. C. D'Elia, E. Coppa, F. Palmaro, and L. Cavallaro, "On the dissection of evasive malware," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 2750–2765, 2020.

[16] S. Choi, T. Chang, C. Kim, and Y. Park, "X64Unpack: Hybrid emulation unpacker for 64-bit windows environments and detailed analysis results on VMProtect 3.4," *IEEE Access*, vol. 8, pp. 127939–127953, 2020.

[17] P. Li, "Selecting and using virtualization solutions: Our experiences with VMware and virtualbox," *J. Comput. Sci. Colleges*, vol. 25, no. 3, pp. 11–17, 2010.

[18] D. T. Vojnak, B. S. Eordevic, V. V. Timcenko, and S. M. Strbac, "Performance comparison of the type-2 hypervisor VirtualBox and VMWare workstation," in *Proc. 27th Telecommun. Forum (TELFOR)*, Nov. 2019, pp. 1–4.

[19] *Themida*. Accessed: Aug. 13, 2020. [Online]. Available: https://www.oreans.com/themida.php

[20] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies," *Black Hat*, vol. 1, pp. 1–27, Jul. 2012.

[21] H. Shi, J. Mirkovic, and A. Alwabel, "Handling anti-virtual machine techniques in malicious software," *ACM Trans. Privacy Secur.*, vol. 21, no. 1, pp. 1–31, Jan. 2018.

YOUNG BI LEE received the B.S. degree in information security engineering from Soonchunhyang University, Asan, South Korea, in 2019. He is currently pursuing the M.S. degree in information security with the Graduate School of Information Security, Korea University. His research interests include software protection, program obfuscation, program deobfuscation, reverse engineering, malware analysis, and digital forensic.

JAE HYUK SUK received the B.S. degree in electrical and computer engineering from the University of Seoul, Seoul, South Korea, in 2012, and the M.S. degree in information security from Korea University, Seoul, in 2014, where he is currently pursuing the Ph.D. degree in information security with the Graduate School of Information Security. His research interests include software protection, program obfuscation, program deobfuscation, reverse engineering, and malware analysis.

DONG HOON LEE (Member, IEEE) received the B.S. degree from Korea University, Seoul, South Korea, in 1985, and the M.S. and Ph.D. degrees in computer science from The University of Oklahoma, Norman, OK, USA, in 1988 and 1992, respectively. Since 1993, he has been with the Faculty of Computer Science and Information Security, Korea University. He is currently a Professor with the Graduate School of Information Security, Korea University. His research interests include cryptographic protocol, applied cryptography, functional encryption, software protection, mobile security, vehicle security, and ubiquitous sensor network security.