Automatic Detection and Bypassing of Anti-Debugging Techniques for Microsoft Windows Environments

Juhyun PARK[1], Yun-Hwan JANG[2], Soohwa HONG[1], Yongsu PARK[1]
[1]*Department of Computer Science, Hanyang University, S. Korea*
[2]*Department of Information Security, Hanyang University, S. Korea*
**Corresponding author Yongsu PARK: yongsu@hanyang.ac.kr*

***Abstract*—In spite of recent remarkable advances in binary code analysis, adversaries are still using diverse anti-reversing techniques for obfuscating code and making analysis difficult. Unlike most of the previous work that relies on debugger-plugins for neutralizing anti-debugging techniques, we focus on the Pin, which is one of the most widely used DBI (Dynamic Binary Instrumentation) tools in 80x86 environments. In this paper, we present an automatic anti-debugging detection/bypassing scheme using the Pin. In order to evaluate the effectiveness of our algorithm, we conducted experiments on 17 most widely used (commercial) protectors, which results in bypassing all anti-debugging techniques automatically. Particularly, our experiment includes Safengine, which is one of the most complex commercial protectors and, to the best of our knowledge, it has not been successfully analyzed by academic researchers up to now. Also, experimental results show that the proposed scheme performs better than the most recent work, Apate.**

***Index Terms*—computer hacking, computer security, debugging, reverse engineering, software protection.**

I. INTRODUCTION

While a considerable amount of research efforts has been done on static and dynamic analysis for malicious code, anti-reverse engineering technology is still being developed to protect malware [1-3].

The protector is a (commercial) program that uses various anti-reverse engineering techniques such as anti-debugging, code encryption, code obfuscation, code virtualization, etc. Themida [4], VMProtect [5], and Safengine [6] are some examples of widely used commercial protectors for Microsoft Windows environments.

If malicious code is packed using the protector, malware analysts will have trouble in analyzing, which causes spending a significant amount of time, i.e., analysis/response can be slow down. E.g., the latest commercial protectors are still considered to be difficult to analyze: Themida's [4] latest code virtualization does not seem to have an easy way to analyze it yet, and to the best of our knowledge, the latest version of Safengine [6] has not been analyzed.

To neutralize anti-debugging techniques, most of previous works have been developed as plug-ins of debuggers [7-9]. Unlike these, we focus on the Pin [10], which is one of the most widely used DBI (Dynamic Binary Instrumentation) tool for 80x86 environment. DBI is the binary code analysis tool for dynamically analyzing the behavior of a target program at runtime. We chose the Pin because it emulates the binary code with high accuracy, and we found out that many anti-debugging techniques are automatically bypassed.

In this paper, we present a new strong anti-anti-debugging scheme using the Pin for 80x86 Windows environments. Our algorithm first finds suspicious code chunk. Then, it tries to match specific anti-debugging technique in the category (API-based, instruction-based, and others). If there is a match, it conducts bypassing work. After the bypassing work is done, it continues execution and analysis until the next anti-debugging code chunk is met.

To evaluate feasibility of our algorithm, we conducted experiments on (commercial) representative 17 protectors, which are considered as the most complex and hard to analyze. Experimental results show that our scheme successfully bypasses all anti-debugging routines in them. The tested protectors include Safengine [6], which is, to the best of our knowledge, one of the most complex protectors and has not been successfully analyzed. Also, experimental results show that our scheme outperforms the most recent work, Apate [9].

This paper is organized as follows. Section II explains the related work. In Section III, we describe our scheme to find the anti-debugging techniques using the Pin in the x86 Microsoft Windows environment. Section IV explains the experimental results and Section V concludes the paper.

II. RELATED WORK

For malware analysis and binary code analysis, a significantly large amount of research work has been done up to now [11-17]. However, topics relevant to advanced anti-reverse engineering techniques have not drawn strong interest from academic researchers (except for some specific topics such as code virtualization [8-9]). Anti-reverse engineering techniques can roughly be classified as anti-debugging, code encryption/self-modification, code obfuscation, and code virtualization [18-21]. For lack of space, we briefly explain recent research work on analysis of anti-debugging techniques.

Xu Chen *et al.*'s work [18] deals with anti-debugging and anti-reverse engineering techniques which are commonly used in malicious code. They first classified and characterized various anti-reversing techniques. Then, they described how to detect and avoid them. Also, they devised

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

Digital Object Identifier 10.4316/AECE.2019.02003

how to detect remote host VMs (virtual machines) using clock skew behaviors. They proposed the new scheme to protect the real machines by mimicking the monitoring/debugging systems such that malware misjudges that it is being monitored/debugged and terminates execution.

Adam J. Smith *et al.* developed REDIR [22], a tool for statically analyzing anti-debugging techniques in the binary code. In order to analyze the obfuscated code, binary code is first transformed into the intermediate language using BAP [23] and then optimized for easy analysis. Then, the suspicious code chunk is found by the predefined rule. If there is a pattern match, it dynamically analyzes the code chunk under the debugging environment in the BAP. If the execution result is different from the actual execution result, they judge that the corresponding chunk is the anti-debugging routine.

In [24], Ping Chen *et al.* investigated how much anti-debugging and anti-VM techniques are used in both general malware and targeted malware (i.e., it is designed for the specific target system). As a result, targeted malware does not use anti-debugging/anti-VM techniques more than general malicious code, unexpectedly. Moreover, they have observed that there is a decrease in the number of used anti-VM techniques over time in the targeted malware.

K. Yoshizaki *et al.* [25] focus on the property that malware uses some specific API functions for anti-debugging. They proposed the automatic scheme to detect anti-debugging techniques by hooking those API calls. After hooking the anti-debugging-related API calls, it changes the return values of API functions, which plays two roles: first, detecting the malicious code, and second, cheating the target malware such that malware execution is not considered to be in the debugging environments. The proposed scheme has two states. In the non-analysis state, malicious code detects the debugging/monitoring status and aborts execution. In the analysis state, malware conducts malicious actions because it cannot detect the debugging status. By comparing these two patterns of behaviors, they proposed the scheme to detect malware automatically.

For detecting/evading anti-debugging techniques, diverse tools have been developed, the most of which are implemented as plug-ins of debuggers. Representative tools include OllyAdvanced [8] and strongOD [7]. Recently, Hao Shi and Jelena Mirkovic proposed Apate [9], which detects and defeats various anti-debugging techniques for Microsoft Windows environments. Apate was implemented as a plug-in for 80x86 WinDbg 6.3. In their implementation, Apate can handle 79 anti-debugging-related attack vectors. They showed that Apate outperforms other debuggers (IDA Pro, OllyDbg, and Immunity Debuggers) for anti-debugging detection. Moreover, in their experiments Apate found all anti-debugging techniques for 4 complex malware samples and 10 (commercial) protectors. In Section IV, we will show experimental results: even though Apate is much better than other DBI/debuggers for detecting anti-debugging, our scheme outperforms Apate, especially for complex commercial protectors.

III. An Automatic Anti-debugging Detection/Bypassing Scheme using the PIN

First, in Section III-A, we explain major anti-debugging techniques which are widely used in Microsoft Windows environments. Then, in Section III-B we describe a new automatic scheme to evade anti-debugging techniques in 80x86 environments. This scheme relies on the Pin [10], which is one of the most widely used DBI tools for 80x86 environments.

Dynamic Binary Instrumentation (DBI) is a binary code analysis tool for dynamically analyzing the behavior of a target program at runtime. DBI emulates and analyzes the target program as follows. First, the plug-in code for dynamic analysis should be made in advance. By using this, DBI interleavingly executes the target code chunk and the plug-in code chunk such that various information such as memory, registry, and API (Application Program Interface)-call information can be analyzed. DBI was primarily designed for performance analysis such as finding slow/bottleneck points in less-optimized code or for finding bugs in erroneous code. Also, DBI can be used for analyzing malware or other diverse applications including [26-29].

A. Analysis of anti-debugging techniques for Microsoft Windows environments.

Generally, each DBI can bypass different anti-debugging techniques. Unlike other debuggers and DBI tools, the Pin can bypass many anti-debugging functionalities without using any plug-in tools. However, execution in the Pin is not exactly same as the real execution and some anti-debugging techniques can detect this difference. Hence, it is necessary to investigate which techniques can detect the Pin or not. We first manually implemented most of anti-debugging techniques [19-21] for Microsoft Windows environments and then conducted experiments with the Pin for checking whether each of them can be evaded by the Pin or not.

TABLE I. ANTI-DEBUGGING TECHNIQUES FOR DETECTING THE PIN

Classification	Anti-debugging technique	Pin Detected
API-based	IsDebuggerPresent	X
	CheckRemoteDebuggerPresent	X
	OutputDebugString	X
	FindWindow	X
	QueryInformationProcess (ProcessDebugPort)	X
	SetInformationThreadDebuggerDetaching	X
	OllyDbg OutputDebugString() Format String	X
	SeDebugPrivilege OpenProcess	X
	QueryInformationProcess (ProcessDebugFlags)	O
	QueryInformationProcess (DebugObjectHandle)	X
	QueryPerformanceCounter	O
	GetTickCount	X
	timeGetTime	X
	CloseHandle	X
	Hardware Breakpoints	X
	Control-C Vectored Exception	X
Instruct	RDTSC	O

ion-based	INT 3 Exception (0XCC)	X
	INT 2D (Kernel Debugger Interrupt)	O
	ICE Breakpoint	X
	Single Step Detection	O
	Unhandled Exception Filter	X
	VMware STR Register Detection	X
	VMware LDT Register Detection	X
	Prefix Handling	O
	CMPXCHG8B and LOCK	X
	VMware Magic Port	X
Others	Memory Breakpoint	O
	Self-Modification	O

Table I shows the experimental results on the Pin for each anti-debugging technique in Microsoft Windows environments. We classify 29 techniques as three categories: the anti-debugging techniques that are performed by calling the API are classified as "API-based," and the technique performed by a specific instruction are classified as "Instruction-based." Techniques that are not in these categories are classified as "Others." As shown in Table I, most of the techniques (except 8) cannot detect the Pin correctly. We briefly describe the 8 techniques that can detect the Pin as follows.

• **QueryInformationProcess** (a.k.a. ProcessDebugFlags): This is one of the ntdll.dll APIs, which has five arguments. The second argument represents the data type of the target program. Suppose that a constant value, 0x1f, is used as this argument and then this API is called. After the return, we check the memory cell pointed by the third argument. if 0 is stored, it is judged to be debugged.

• **QueryPerformanceCounter:** This is the time-related API of kernel32.dll. This API returns the current data value of the hardware performance counter to the specified memory. Because DBI execution is slower than the real execution, we can judge that the program is being analyzed when the difference between the two data values exceeds a predetermined threshold value after this API is called twice.

• **RDTSC:** This is the instruction that returns the processor timestamp value, which records the number of elapsed clock cycles from the last reset time. Due to the slow execution property of DBI, the anti-debugging technique judges that the program is being debugged when the difference between two timestamp values exceeds a certain threshold value after executing this instruction twice.

• **Int 0x2d:** In this technique, we register the desired exception handler in fs: [0x0] where SEH (Structured Exception Handler, the exception handling routine of Windows operating system) is located. Then, we generate the breakpoint exception using int 0x2d instruction. The DBI tools such as the Pin, usually ignore the execution of this handler routine. Hence, this technique checks whether the handler has been executed or not to judge for being in the debug mode.

• **Single Step Detection:** The debugger or the DBI tool can be detected by using the fact that it ignores or incorrectly handles the exception when a single-step exception occurs. The trap flag in the EFLAGS register is flipped to invoke a single-step exception. In normal execution, the SEH handler is executed but in DBI/debugging environments, usually it is not executed.

• **Prefix Handling:** In some 80x86 instructions, there is a prefix byte, which gives a different meaning when interpreting the opcode. For example, rep prefix means to repeat the subsequent instruction until the ecx register becomes zero. If an interrupt is invoked when the instruction is repeatedly executed, the debugger or the DBI program may ignore or incorrectly handle the exception, which can be used to detect the debugger.

• **Memory Breakpoint:** Some DBI programs can execute instructions in the protected region while there is no exception occurred. However, in the real execution an exception should occur in the access of the protected region. This difference is used to detect the debugger/DBI. Typically, we set protection region at a specific area in the runtime and then access it using jmp/call/ret instructions.

• **Self-Modification:** Some DBI/debuggers execute code chunks in the assumption that the code has not been modified during the run-time. If the code is modified in the run-time, they should carefully flush the instruction cache memory to reflect the modification. Otherwise, the original instructions (before the modification) are executed. Note that self-modifying code is not common in the normal program. Hence, many DBIs including the Pin do not flush the caches. This difference is used to detect the debugger/DBI.

B. Automatic detection and neutralization of anti-debugging techniques using the Pin

In this subsection, we present a new automatic detection and neutralization scheme for anti-debugging techniques using the Pin tool. This scheme conducts dynamic analysis, in which it executes the code chunk in the target program and detects/bypasses anti-debugging techniques at the run-time. It relies on Intel's Pin DBI in x86 environments. As mentioned in Section III-A, the Pin is detected in 8 anti-debugging techniques out of the 29. Hence, we focused on these 8 cases: detecting and evading them automatically.

Figure 1 shows the overall structure of the proposed scheme. First, the Pin loads the target program (e.g., malware containing anti-debugging techniques) on the memory and fetches the first instruction. Then, it calls our scheme (plug-in) for detecting anti-debugging techniques. At 'Check Current IP' step, our scheme first checks whether the current instruction belongs to the DLL (Dynamic Link Library) or to the target program code area. If it belongs to the DLL area, we should check the API-based anti-debugging techniques since API code resides in DLL. Otherwise, we go to 'Check the Instruction' step. In this step, we examine whether the current instruction has some specific functionalities w.r.t. anti-debugging. If so, we perform pattern matching to find the corresponding instruction-based anti-debugging techniques. Finally, in "Check other Anti-Debugging Techniques" step, we check other anti-debugging techniques. After finishing execution of our scheme (plug-in code), execution goes back to the Pin. Optionally, the Pin can execute other plug-ins (e.g., for malware analysis). Then, the Pin emulates (decodes and executes) the current instruction. Finally, it fetches the next instruction of the target program.

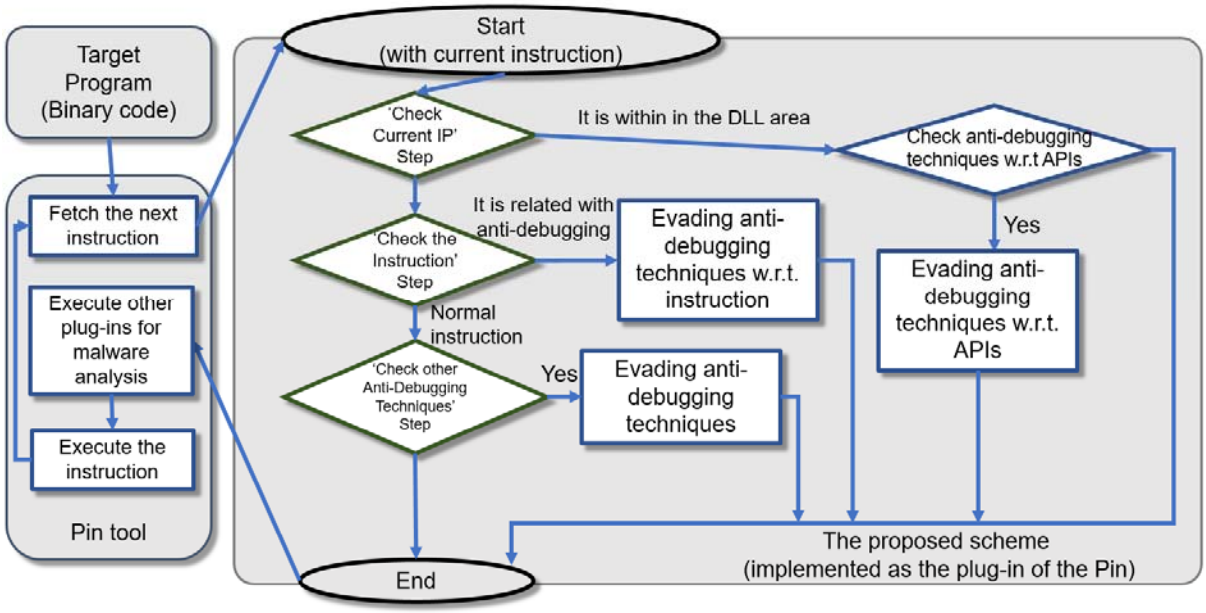

Figure 1. Overall structure of the proposed scheme for detecting and bypassing anti-debugging techniques using the Pin.

Algorithm 1 Algorithm for 'Check Current IP' Step

Input current instruction (**curIP**) to be executed, previously executed instruction (**preIP**), current API (**curAPI**) to be or being executed, and the current status.
Output the flag indicating presence of anti-debugging technique, the (changed) status.

```
static timecount = 0
while curIP > 0x10000000 and preIP < 0x10000000   // Check DLL Area
  if curAPI = QueryInformationProcess() then      // Check API Name
    if ProcessDebugPort = ProcessDebugFlags then  // Check API's argument value
      ProcessInformation = 1                      // Change the return value
  else if curAPI = QueryPerformanceCounter() then // Check API Name
    lpPerformanceCount = timecount                // Change the API's argument value
    timecount <- timecount + 1                    // Increment for next API call
ENDWHILE
```

Figure 2. Pseudo-code of 'Check current IP' Step in Figure 1.

Fig. 2 shows the pseudo-code of the algorithm executed at the 'Check Current IP' step of Fig. 1. In Fig. 2, we check whether the control flow goes from the target program area to the DLL region or vice versa, where the threshold address value is 0x10000000; in Microsoft Windows (32-bit) environment, the memory region below 0x10000000 is for code, global data, stack, and heap while the area above that is used for DLL, memory-mapped file, etc. If the execution flow goes to DLL area, this means that the API has been called. At this point, we check appropriate API-based anti-debugging techniques, as follows.

If the name of currently called API is QueryInformationProcess(), as explained in Section III-A, we should check whether the 2nd argument of the call is ProcessDebugFlags (0x1f). If true, we change the value of ProcessInformation, the return value of QueryInformationProcess(), to 1 (true) for bypassing.

If there is an API call for QueryPerformanceCounter(), instead of returning the hardware timestamp value, we return the virtual time counter value (timecount). Then, timecount is increased by 1. If the same API is called again, it will return timecount+1, and the difference between the return values of two calls becomes small (=1), which makes anti-debugging technique get neutralized.

If the address of the currently executed instruction is 0x10000000 or less, we regard that the instruction is in the target program area (code, data, stack and heap). Fig. 3 shows the pseudo-code of the algorithm executed at the 'Check the Instruction' step of Fig. 1. In Fig. 3, we find instruction-based anti-debugging techniques by pattern-matching.

Algorithm 2 Algorithm for 'Check the Instruction' Step

Input current instruction (**curIP**) to be executed and the current status.
Output the flag indicating presence of anti-debugging technique, the changed status.

```
static timecount = 0
while curIns                                          // Check instruction exists
  if curIP = rdtsc then                               // Check instruction name
    eax = timecount                                   // Change the register value
    timecount <- timecount + 1                        // Increment for next API call
  else if curIP = int 0x2d then                       // Check instruction name
    PIN_RaiseException(EXCEPTCODE_DBG_BREAKPOINT_TRAP)  // Raise breakpoint exception
  else if curIP = popfd then                          // Check instruction name
    trapflag = 1                                      // Set trap flag (8th bit of EFLAGS) to 1
    PIN_RaiseException(EXECPTCODE_DBG_SINGLE_STEP_TRAP)  // Raise single step exception
  else if curIP = int and sizeof(curIP) = 3 then      // Check instruction name
    curIP <- curIP + 2                                // Increase current instruction pointer
    PIN_ExecuteAt()                                   // Execute instruction again
ENDWHILE
```

Figure 3. Pseudo-code of 'Check the Instruction' Step in Figure 1.

If the current instruction is RDTSC, we change the value of eax register to the virtual time counter (timecount) and then increment timecount value by 1. Since subsequent executed RDTSC will return timecount+1, the difference between the return values of two RDTSC calls becomes small (=1), which makes the anti-debugging technique get neutralized.

If the program encounters the instruction 'int 0x2d' when executed normally, it will raise a breakpoint exception whereas the Pin does not raise exception at all. Therefore, if the current instruction is 'int 0x2d,' breakpoint exception should be forcibly raised using the Pin library function, PIN_RaiseException().

If current instruction is 'popfd,' we should check the EFLAGS that is stored in the stack before executing instruction. The trap flag, which is the 8th bit of EFLAGS, raises the single step exception. Since the Pin cannot handle single step exception automatically, if the trap flag bit is 1, it has to be cleared to 0. After that, single step exception should be forcibly raised using the Pin library function, PIN_RaiseException().

Finally, if the current opcode is 'int' and the size of instruction is 3 bytes, this instruction has some prefix bytes because the size of the instruction 'int' is originally 1 byte. Generally, the Pin cannot correctly execute this instruction. In order to manually handle this, current instruction pointer is incremented by 2 to skip prefix bytes for bypassing this anti-debugging technique.

Algorithm 3 Algorithm for 'Check other Anti-debugging Technique' Step

Input current instruction (**curIP**) to be executed, previously executed instruction (**preIP**) and the current status.
Output the flag indicating presence of anti-debugging technique, the (changed) status.

```
while curIns                                          // Check instruction exists
  if INS_IsBranchorCall(curIP) and preIP < 0x10000000 then   // Check curIns is branch or call from target area
    targetAddr <- target address of curIP             // Store the target address of curIP
    if !PIN_CheckReadAccess(targetAddr) then          // Check Page Guard
      PIN_RaiseException(EXCEPTCODE_ACCESS_INVALID_PAGE)   // Raise invalid page access exception
  else if INS_IsMemoryWrite(curIP) then               // Check curIns is memory write
    targetAddr <- target address of curIP             // Store the target address of curIP
    if addrnow < targetAddr ≤ addrlast                // Check write operation is self-modification
      PIN_ExecuteAt()                                 // Execute again to clean the cache memory
ENDWHILE
```

Figure 4. Pseudo-code of 'Check other Anti-debugging Technique' Step in Figure 1.

Fig. 4 shows the pseudo-code of 'Check other Anti-Debugging Techniques' step of Fig. 1. At this step, it is necessary to check the current status to identify anti-debugging techniques such as memory breakpoints and self-modification. Memory breakpoints occur only when the current instruction is a branch or call. For each branch or call instruction, in order to find memory breakpoints, we check whether access violation is invoked or not by using the Pin library function, PIN_CheckReadAccess(). If the return value is false, it means that PAGE_GUARD is applied to the target address. Since the Pin cannot handle this automatically, invalid page access exception should be manually raised using PIN_RaiseException().

Since self-modification occurs only when the current instruction is the memory write operation, we hook every memory write operation using INS_IsMemoryWrite(). For each memory write operation, we check self-modification as follows. When executing self-modified code, the Pin overwrites instructions in the cache area. We can check this case by utilizing the fact that the Pin copies the code chunk from the memory to the cache area all at once and then executes each instruction in the cache. Let "addrlast" denote the last address of the cached chunk of code and "addrnow" denote the current instruction pointer. We hook every write operation and check whether addrnow < writing_address ≤ addrlast or not. If it is true, we judge that self-modification occurs, and we call PIN_ExecuteAT() function to clean the cache memory in the Pin.

IV. Experimental Results

In Section IV-A we explain the experimental results for each anti-debugging technique. Then, in Section IV-B, we explain the experimental results on our scheme for the major (commercial) protectors. Experimental environments are as follows. CPU: Intel i5 3.7GHz (80x86), Operating system: Microsoft Windows 7 (32bit), DBI tool: Pin v3.2.

A. Experimental results for each anti-debugging technique.

We have implemented the proposed scheme and conducted experiments for 29 anti-debugging techniques [19-21]. The following table summarizes the experimental results. Unlike the Pin, which has simple instruction tracing code, the proposed scheme successfully evades all of them.

Table II. Experimental Results on Anti-Debugging Techniques

Classification (number of anti-debugging techniques)	**The number of successfully by-passed anti-debugging techniques**	
	Pin	**Proposed scheme**
API-based anti-debugging techniques	14	16
Instruction-based anti-debugging techniques	7	11
Others	0	2
Total: 29	21	29

B. Experimental results for major protectors.

Recall that the Protector (a.k.a. the packer) is the program that uses various anti-reverse engineering techniques to deter program analysis. In this experiment, we chose 17 major protectors, which are being widely used in Microsoft Windows environments. The protector list includes ASProtect 2.56, Enigma Protector 4.40, Themida 2.2.7 [4], ACProtect 2.0.0, VMProtect 3.09, Safengine 2.3.9.0 [6], and, Obsidium 1.5 [30], which can be considered the most complex and difficult to analyze.

Experiments were conducted as follows. First, we used each protector to pack the unprotected program. Second, we conducted simple dynamic analysis (extracting the instruction trace) using the previous tools: WinDbg 10.0, OllyDbg 1.1, QuickUnpack 2.2, AbstersiverA, the vanilla Pin (it has simple instruction tracing code and does not contain any anti-debugging detection methods), OllyAdvanced [8] and Apate [9]. Third, we conducted dynamic analysis using the proposed scheme.

Table III. Experimental Results on Major Protectors

Protector name	**WinDbg**	**OllyDbg [31]**	**QuickUnpack**	**AbstersiverA**	**Pin**	**OllyAdvanced**	**Apate [9]**	**Our scheme**
UPX 1.02	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
PECompact	No	No	Yes	No	Yes	Yes	No	Yes
ASPack 2.0	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
WWPack	No	No	No	No	Yes	No	Yes	Yes
Packman	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Petite	Yes	Yes	No	No	Yes	Yes	Yes	Yes
MEW	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Mpress	Yes	Yes	No	No	Yes	Yes	Yes	Yes
Nspack	No	Yes	No	No	Yes	Yes	Yes	Yes
yoda 1.3	No	No	No	No	No	No	No	Yes
ASProtect 2.56	No	Yes	No	No	Yes	Yes	Yes	Yes
Enigma 4.40	No	No	No	No	No	No	Yes	Yes
Themida 2.3.5	No	No	No	No	Yes	No	No	Yes
ACProtect 2.0.0	No	No	No	No	No	Yes	Yes	Yes
VMProtect 3.09	Yes	No	No	No	No	No	Yes	Yes
Safengine 2.3.9.0	No	No	No	No	No	No	No	Yes
Obsidium 1.5	No	No	No	No	No	No	No	Yes

Success rate	41.2%	47.1%	29.4%	0.06%	64.7%	58.8%	70.5%	100 %

Table III shows the experimental results on the 17 protectors. The success rate is defined as the number of successfully analyzed cases (i.e., successfully bypassed all anti-debugging techniques) divided by the number of all protectors. As shown in Table III, the success rates of WinDbg, OllyDbg, QuickUnpacker, AbstersiverA, OllyAdvanced are relatively low: from 0.06% – 58.8%. (We omit StrongOD [7] because its result is identical to OllyAdvanced.) Compared with these tools, the success rate of the vanilla Pin is much higher: 64.7%. The vanilla Pin successfully executes ASProtect, Enigma protector, and Themida until the OEP (original entry point, the starting address of the original program) is met. However, it cannot evade anti-debugging techniques and execution is aborted before the OEP on ACProtect, VMProtect, Safengine and Obsidium. The success rate of Apate, 70.5%, is slightly higher than that of the vanilla Pin. Apate can successfully analyze for relatively simple protectors whereas execution aborts for some complex commercial protectors. We confirmed that for all 17 protectors, our implementation of the proposed scheme can successfully detect and evade all anti-debugging techniques. Among them, to the best of our knowledge, SafeEngine 2.3.9 is the protector that can be considered extremely difficult to analyze and has not been successfully analyzed by academic researchers up till now.

V. CONCLUSION

We have implemented 29 anti-debugging engineering techniques in Microsoft Windows environments and conducted experiments to check whether they can detect the Pin tool, or not. The results show that 8 of them can detect the Pin. From this, we devised an automatic scheme that can detect and evade anti-debugging techniques for Microsoft Windows environments. We conducted experiments on 17 major protectors. Experimental results show that the proposed method can detect and evade all anti-debugging techniques used in the protectors. Among them, to the best of our knowledge, SafeEngine 2.3.9 is the protector that has not been successfully analyzed in the public until now.

REFERENCES

[1] W. Yan, Z. Zhang, N. Ansari, "Revealing packed malware," IEEE Security and Privacy, Vol. 6, No. 5, pp. 65-69, 2008. doi:10.1109/msp.2008.126

[2] D. Devi, S. Nandi, "Detection of packed malware," in Proc. of the First International Conference on Security of Internet of Things, pp. 22-26, 2012. doi:10.1145/2490428.2490431

[3] G. N. Barbosa, R. R. Branco, "Prevalent characteristics in modern malware," in Proc. of Black Hat'2014, USA, 2014.

[4] Orleans Technology, "Themida: advanced windows software protection system," https://www.oreans.com/themida.php, 2014.

[5] VMSoft. "VMProtect software: VMProtect virtualizes code," http://vmpsoft.com/products/vmprotect/, 2018.

[6] Safengine, "Safengine protector," http://www.safengine.com/en-us/, 2017.

[7] StrongOd, StrongOD 0.4.8.892 – Make your OllyDbg Strong, https://tuts4you.com/download.php?view.2028, 2012.

[8] OllyAdvanced, OllyAdvanced – OllyDbg plugin for a number of advancements and anti-debug features, https://www.aldeid.com/wiki/OllyDbg/OllyAdvanced, 2013.

[9] H. Shi, J. Mirkovic, "Hiding debuggers from malware with Apate," in Proc. of ACM SAC'2017, pp. 495-508, 2017. doi:10.1145/3019612.3019791

[10] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Raddi, K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in Proc. of the 2005 ACM SIGPLAN Conference on PLDI, pp. 190-200, 2005. doi:10.1145/1064978.1065034

[11] S. Bardin, R. David, J. Marion, "Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes," in Proc. of 2017 IEEE Symposium on Security and Privacy, pp 633-651, 2017. doi:10.1109/sp.2017.36

[12] T. Blazytko, M. Contag, C. Aschermann, T. Holz, "Syntia: Synthesizing the Semantics of Obfuscated Code," in Proc. of USENIX Security Symposium 2017, pp. 643-659, 2017.

[13] R. David, S. Bardin, T. D. Ta, J. Feist, L. Mounier, M. L. Potet, J. Y. Marion. "BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-level Analysis," In Proc. of 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER) 2016, pp. 653-656, 2016. doi:10.1109/saner.2016.43

[14] X. Meng, B. P. Miller. "Binary code is not easy," in Proc. of the 25th International Symposium on Software Testing and Analysis, pp. 24-35, 2016. doi:10.1145/2931037.2931047

[15] S. Eschweiler, K. Yakdan, E. Gerhards-Padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code," in Proc. of The Network and Distributed System Security Symposium (NDSS 2016), 2016. doi:10.14722/ndss.2016.23185

[16] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, T. Holz. "Cross-Architecture Bug Search in Binary Executables," in Proc. of the 2015 IEEE Symposium on Security and Privacy 2015, pp. 709-724, 2015. doi:10.1109/sp.2015.49

[17] J. Lee, H. Chang, S. Cho, S. Kim, Y. Park, W. Choi, "Integration of Software Protection Mechanisms against Reverse Engineering Attacks," Journal of Information, Vol. 15. No. 4, pp. 1569-1578, 2012.

[18] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, J. Nazario, "Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware," in Proc. of IEEE Conference on Dependable Systems and Networks (DSN 2008), pp. 177-186, 2008. doi:10.1109/dsn.2008.4630086

[19] J. Tully, "Introduction into Windows anti-debugging," http://www.codeproject.com/Articles/29469/Introduction-Into-Windows-Anti-Debugging/, Sep. 2008.

[20] P. Ferrie, "The ultimate anti-debugging reference," http://www.anti-reversing.com/the-ultimate-anti-debugging-reference/, 2011.

[21] T. Shields, "Anti-debugging – a developers view," 2011.

[22] A. J. Smith, R. F. Mills, A. R. Bryant, G. L. Peterson, M. R. Grimaila, "REDIR: Automated static detection of obfuscated anti-debugging techniques," in Proc. of 2014 International Conference on Collaboration Technologies and Systems 2014, pp. 173-180, 2014. doi:10.1109/cts.2014.6867561

[23] D. Brumley, I. Jager, T. Avgerinos, E. J. Schwartz, "BAP: A Binary Analysis Platform," in Proc. of International Conference on Computer Aided Verification 2011, pp. 463-469, 2011. doi:10.1007/978-3-642-22110-1_37

[24] P. Chen, C. Huygens, L. Desmet, W. Joosen, "Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware," in Proc. of IFIPSEC'2016 Conference, pp. 323-336, 2016. doi:10.1007/978-3-319-33630-5_22

[25] K. Yoshizaki, T. Yamauchi, "Malware Detection Method Focusing on Anti-debugging Functions," in Proc. of Computing and Networking (CANDAR) 2014, pp. 563-566, 2014. doi:10.1109/candar.2014.36

[26] V. Oduguwa, A. Tiwari, R. Roy, "Evolutionary computing in manufacturing industry: an overview of recent applications," Applied Soft Computing, vol. 5, no. 3, pp. 281-299, 2005. doi:10.1016/j.asoc.2004.08.003

[27] C. Pozna, F. Troester, R. E. Precup, J. Tar, S. Preitl, "On the design of an obstacle avoiding trajectory: method and simulation," Mathematics and Computers in Simulation, vol. 79, no. 7, pp. 2211-2226, 2009. doi:10.1016/j.matcom.2008.12.015

[28] J. Saadat, P. Moallem, H. Koofigar, "Training echo state neural network using harmony search algorithm," International Journal of Artificial Intelligence, vol. 15, no. 1, pp. 163-179, 2017.

[29] S. Vrkalovic, E. Lunca, I. Borlea, "Model-free sliding mode and fuzzy controllers for reverse osmosis desalination plants, International Journal of Artificial Intelligence," vol. 16, no. 2, pp. 208-222, 2018.

[30] Obsidium Software. "Obsidium Software Protection System," http://www.obsidium.de/, 2016.

[31] OllyDbg. "OllyDbg v1.10: 32-bit assembler level analyzing debugger for Microsoft Windows," http://www.ollydbg.de/, 2014.