Received June 5, 2020, accepted June 29, 2020, date of publication July 13, 2020, date of current version July 23, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3008900

x64Unpack: Hybrid Emulation Unpacker for 64-bit Windows Environments and Detailed Analysis Results on VMProtect 3.4

SEOKWOO CHOI[1], TAEJOO CHANG[1], (Senior Member, IEEE), CHANGHYUN KIM[2], AND YONGSU PARK[2]

[1]The Affiliated Institute of ETRI, Daejeon 305-600, South Korea
[2]Department of Computer Science, Hanyang University, Seoul 04763, South Korea

Corresponding author: Yongsu Park (yongsu@hanyang.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIT) under Grant 2020R1F1A1048443.

ABSTRACT In spite of recent remarkable advances in binary code analysis, malware developers are still using complex anti-reversing techniques to make analysis difficult. To protect malware, they use packers, which are (commercial) tools that contain various anti-reverse engineering techniques such as code encryption, anti-debugging, and code virtualization. In this paper, we present x64Unpack: a hybrid emulation scheme that makes it easier to analyze packed executable files and automatically unpacks them in 64-bit Windows environments. The most distinguishable feature of x64Unpack compared to other dynamic analysis tools is that x64Unpack and the target program share virtual memory to support both instruction emulation and direct execution. Emulation runs slow but provides detailed information, whereas direct execution of the code chunk runs very fast and can handle complex cases regarding to operating systems or hardware devices. With x64Unpack, we can monitor major API (Application Programming Interface) function calls or conduct fine-grained analysis at the instruction-level. Furthermore, x64Unpack can detect anti-debugging code chunks, dump memory, and unpack the packed files. To verify the effectiveness of x64Unpack, experiments were conducted on the obfuscation tools: UPX 3.95, MPRESS 2.19, Themida 2.4.6, and VMProtect 3.4. Especially, VMProtect and Themida are considered as some of the most complex commercial packers in 64-bit Windows environments. Experimental results show that x64Unpack correctly emulates the packed executable files and successfully produces the unpacked version. Based on this, we provide the detailed analysis results on the obfuscated executable file that was generated by VMProtect 3.4.

INDEX TERMS Anti-forensics, code obfuscation, computer security, dynamic code analysis, reverse engineering.

I. INTRODUCTION

Nowadays, software developers are using a variety of obfuscation techniques to deter code analysis and to protect their copyright. Modern (commercial) obfuscation tools [1]–[3] contain strong anti-reverse engineering techniques and are actively utilized for anti-computer forensics to deter analysis [4]. Also, they are widely used for developing malware.

Even though each has a different behavior, basically, the obfuscation tools use a common technique called *code packing*, a method of compressing or encrypting the target program for protection: it transforms the target program into the packed one by compressing or encrypting the code into the packed data and associating this with the unpacking routine. Additionally, these tools put diverse anti-reverse engineering techniques (e.g., anti-debugging [5], self-modifying code [6], code-encryption [5]) in the unpacking routine to make analysis difficult.

To analyze packed software, static analysis, which conducts analysis without execution, has its limitation since the target code is encrypted or compressed. To overcome this, dynamic analysis can be used, which conducts code execution and analysis together. To analyze dynamically, debuggers [7] or DBI (Dynamic Binary Instrumentation) tools [8], [9] are

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

widely used, which have shortcomings in that execution environments are slightly different from that of actual execution and many anti-reverse engineering techniques can detect this difference.

To cope with new anti-reverse engineering techniques and to effectively analyze packing tools, we present x64Unpack: the hybrid application emulator that alternates emulating machine instructions and executing binary code chunks. x64Unpack runs in 64-bit Microsoft Windows environments and has the following distinctive features. Since the target program and x64Unpack share virtual memory, target code chunks can be directly executed, which provides efficiency and correctness. Alternatively, they can be emulated using the CPU emulator, which provides fine-grained analysis (including instruction-level emulation, tracking registers or memory I/O, and exception handling).

For CPU emulation, the Bochs emulator [10] is used to improve accuracy. We use [11] to automatically decode the obfuscated API (Application Program Interface) function calls. Moreover, x64Unpack provides diverse functionalities for analyzing packed software, including automatic finding the original entry point (OEP: the entry point of the original target program, refer to Section II-A.) [12], detecting anti-debugging routines, dumping the memory region, and automatically unpacking the packed file.

To verify the effectiveness of the proposed technique, we conducted experiments on widely used (commercial) packers in 64-bit Microsoft Windows environments: VMProtect 3.4 [2], MPRESS 2.19, Themida 2.4.6 [1], and UPX 3.95 [13]. Especially, VMProtect and Themida are considered as some of the most complex commercial protectors in Microsoft Windows environments. x64Unpack successfully unpacks all the test files that were packed with these protectors and has produced detailed logs. Based on this, we explain in detail the structure of obfuscated files that were generated from VMProtect 3.4. We also discuss how to unpack and obtain the original unprotected code. The following is a summary of the contributions of this paper.

- Unlike previous work/tools, x64Unpack supports both instruction emulation mode and direct execution mode to provide fine-grained analysis and fast execution.
- While the previous dynamic analysis tools [7]–[9] focus on accuracy and speed, x64Unpack aims at automatically evading anti-reverse engineering techniques to make the analysis environment as close as possible to the actual execution.
- x64Unpack is the specialized unpacking tool that analyzes the unpacking process, which resides in the beginning of the program execution. x64Unpack automatically (or manually) unpacks the target program when OEP is encountered, rather than analyzing the entire target program.
- We provide a detailed analysis of the latest version of VMProtect: Version 3.4. To the best of our knowledge, there has been no analysis results published for the recent version of VMProtect.

This paper is organized as follows. In Section II we describe x64Unpack, the proposed hybrid dynamic program analysis tool. Section III summarizes the experimental results and Section IV provides the detailed analysis results on the unpacking routine of the VMProtect 3.4. Section V explains how to perform unpacking to get the original file. Section VI deals with related work and we conclude in Section VII.

II. x64Unpack: HYBRID APPLICATION EMULATOR FOR 64-BIT WINDOWS ENVIRONMENTS

Section II-A shows the general unpacking procedure and Section II-B explains the overview of x64Unpack. In Sections II-C - II-I, we deal with details of x64Unpack, including memory management, CPU emulation, API function calls, exception handling, and program loading. In Section II-J, we explain implementation issues.

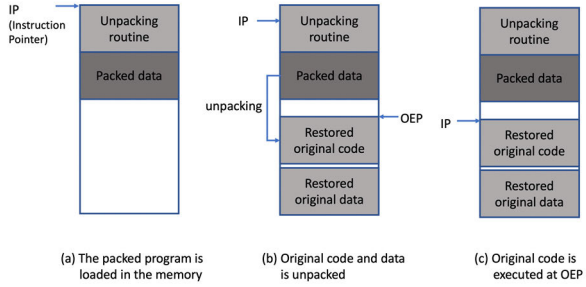

FIGURE 1. General unpacking procedure.

A. (SIMPLIFIED) UNPACKING PROCEDURE

Recall that code packing is the technique to transform the target program into a packed one such that it compresses or encrypts the code into the packed data and associates this with the unpacking routine. Fig. 1 shows the conventional unpacking procedure when the packed program is executed. After the packed program starts (Fig.1-(a)), the execution flow goes to the restoration routine (we call this the unpacking routine), which unpacks/decrypts the packed data to restore the original code and the original data. When this work is completed (Fig.1-(b)), it also restores the execution context for the original program code, including initialization of CPU registers. Then, it sets the program counter to the entry point of the unpacked original code region (we call this point OEP (Original Entry Point)), which is shown in Fig.1-(c). Finally, the restored original code is executed. In general, complex packers use code-packing multiple times while using diverse anti-reverse engineering techniques to deter analysis.

B. OVERVIEW OF x64Unpack

The x64Unpack runs as a single process, where Fig. 2 shows the simplified overall running procedure. First, as shown in Figure 2-(b), x64Unpack loads the packed target program into memory (which is described in Section II-I). Then, each instruction of the unpacking routine is emulated one by one using the CPU emulator and the original code and data are

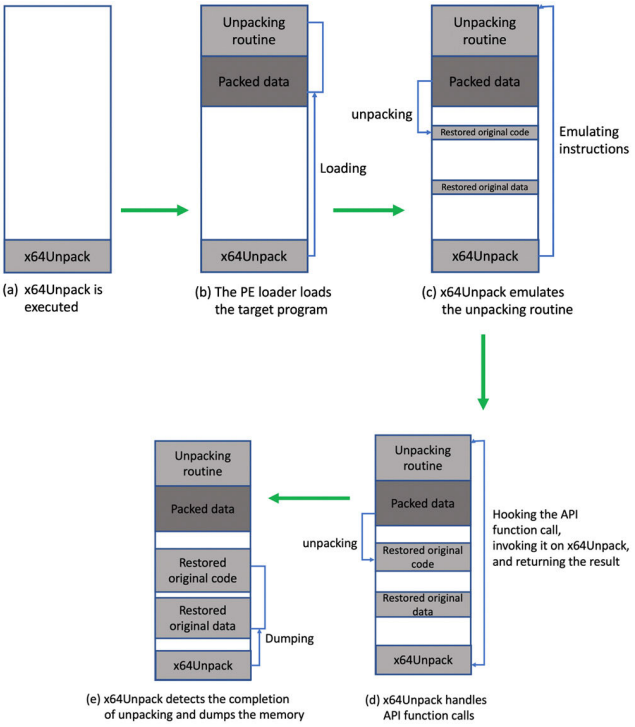

FIGURE 2. **(Simplified) overall execution procedure of x64Unpack.**

partially restored (Figure 2-(c), Section II-D). If the unpacking routine invokes the API function call, x64Unpack hooks it and returns the result after calling the function. (Figure 2-(d), Section II-E). When the original code and data are fully restored, x64Unpack automatically detects this (using [12]) and dumps the memory to produce the unpacked executable file (Figure 2-(e), Section II-G).

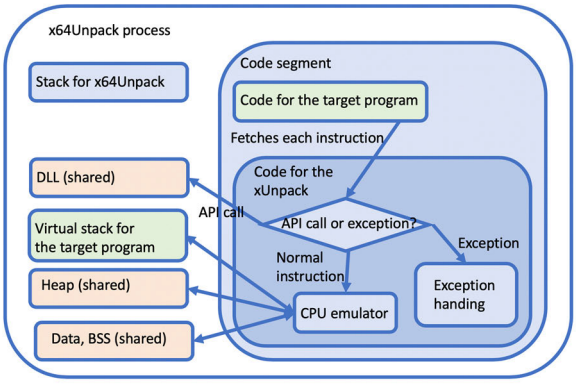

FIGURE 3. **Memory layout of x64Unpack.**

The x64Unpack's memory structure is shown in Fig. 3. x64Unpack manages two separate stacks: one is a real stack for the x64Unpack and the other is a virtual stack for the target program. Other segments, i.e., Heap, Data, BSS and DLL (Dynamic Link Library) are shared by the target program and x64Unpack. The code segment contains the target process code and the x64Unpack code.

Suppose that the target program has already been loaded (we will explain this in Section II-I). The emulation of the target program is conducted as follows. First, x64Unpack takes every machine instruction of the target program and emulates it using the CPU emulator. Sometimes, the instruction is related to the call/jump to the API (Application Program Interface) function (in DLL). Then, x64Unpack directly calls the corresponding function on behalf of the target program and returns the result to the target program. Additionally, x64Unpack handles diverse tricky cases including exceptions, interrupt-related instructions. In emulating machine instructions, all memory I/O operations are hooked to check permissions.

C. MEMORY MANAGEMENT

As mentioned in Section II-B, the memory sections are shared between the x64Unpack and the target program except for the stack (shown in Fig. 3). This makes it possible to set an emulation environment that is very close to real execution. When x64Unpack loads the target program into the memory, it allocates and loads the code and data exactly in the same area as in the case when the target program is normally executed (which will be explained in Section II-I). To do so, the code of x64Unpack should be loaded in other unused space. In 64-bit environments, the virtual address space is very large so finding the unused space is not difficult.

Because we alternate execution of the target program and x64Unpack, we should maintain separate stacks (as shown in Fig. 3). When emulating each machine instruction of the target program, we should hook to check permissions, e.g., memory I/O. If the operation is not allowed, we should raise an appropriate exception just as in the real execution. Sometimes, if the memory access is related with anti-debugging, we should modify content of the specific memory region for automatic bypass.

D. CPU EMULATION

x64Unpack has a CPU emulator, which emulates the target program's code on a per instruction basis. It fetches the instruction from the starting location in the (packed) target program and then checks the address of the instruction. If it is the starting address of an API function, x64Unpack calls the API function directly (which is explained in Section II-E). Otherwise, x64Unpack uses the CPU emulator to emulate the instruction and then changes the instruction pointer (RIP register) in the CPU emulator to process the next instruction. Fig. 4 shows the main emulation loop in x64Unpack to handle each instruction of the target program. If the instruction pointer (RIP register) is MAGIC_VALUE, x64Unpack handles the return of the vectored exception (which is explained in Section II-H). If the RIP points out the starting address of the API function, x64Unpack executes the corresponding API function directly. Otherwise, it emulates the instruction using the CPU emulator. Then, it outputs the status and writes the log message.

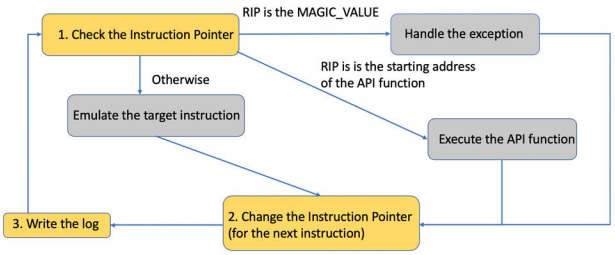

FIGURE 4. The main loop in x64Unpack (for handling each instruction in the target program).

E. API FUNCTION CALLS

Whenever a new DLL module (or the target program) is loaded, x64Unpack registers the list of all functions of the DLL module (or the target program) in the memory. Using this, when the instruction is emulated, x64Unpack checks its address: If the address of the instruction is within the API function code area, x64Unpack calls the corresponding the API function directly and the return value is passed to the CPU emulator.

```
void HaeAPI::hae_wcsnicmp(BX_CPU_C* hcpu, quint64 addr)
{
    typedef Bit64u (__fastcall *func_t)(Bit64u, Bit64u, Bit64u);
    func_t func = (func_t)addr;
    hRAX = func(hRCX, hRDX, hR8);
    hcpu->_LastError = GetLastError();
    MSG((hcpu->thread_index, OUTPUT_BOTH, "_wcsnicmp called: %S %S", (wchar_t*)hRCX, (wchar_t*)hRDX)


}
void HaeAPI::haeCreateFileW(BX_CPU_C* hcpu, quint64 addr)
{
    typedef Bit64u (__fastcall *func_t)(Bit64u, Bit64u, Bit64u, Bit64u, Bit64u, Bit64u, Bit64u);
    func_t func = (func_t)addr;
    hcpu->_LastError = GetLastError();

    Bit64u a4 = hcpu->readQword(hRSP+32);
    Bit64u a5 = hcpu->readQword(hRSP+32+8);
    Bit64u a6 = hcpu->readQword(hRSP+32+16);

    hRAX = func(hRCX, hRDX, hR8, hR9, a4, a5, a6);

    MSG((hcpu->thread_index, OUTPUT_BOTH, "CreatFileW: %S access: %x share: %x a4: %llx a5: %llx a6:!
        (Bit32u)hR8, a4, a5, a6, hRAX));

}
```

FIGURE 5. Wrapper functions for _wcsnicmp() and CreateFileW.

Most of Windows API functions can be implemented simply by calling them directly. For example, Fig. 5 shows the code chunk that directly executes the _wcsnicmp() and CreateFileW() API functions. In the case of the _wcsnicmp(), since it receives three parameters, x64Unpack copies the parameters into the RCX, RDX, and R8 registers. After calling _wcsnicmp(), x64Unpack stores the result in the RAX register. Similarly, CreateFileW() is an API function that takes seven arguments so x64Unpack uses the RCX, RDX, R8, and R9 registers and the stack to set parameters and then calls the CreateFileW() function.

Some specific functions require manipulations before/after the function call or we should reimplement the function. For example, for the timeGetTime() function, we adjust the return value (tick counter) because the target program may detect that the tick counter value goes too fast (i.e., emulation is slow). To make the tick go slow, x64Unpack maintains the virtual tick counter variable. First, x64Unpack reads the system time at startup and sets the virtual tick counter as the real value. After that, x64Unpack manipulates the virtual counter to go much slower than the real one as emulation proceeds. (For detail of the change of the virtual tick counter, refer to Section II-F.) Similarly, x64Unpack uses virtual values for the process ID and thread ID (for evading anti-reverse engineering techniques). Related major API functions are GetCurrentProcessID(), GetCurrentProcess(), GetCurrentThread(), and GetCurrentTrheadId(), in all which x64Unpack modifies the return values.

The advantage of treating API functions as direct calls rather than emulating their code is that the execution speed is much faster than that of emulation. Moreover, direct execution is more accurate for complex code. On the downside, there are a large number of API functions in the Windows environments, which are too much work for handling all of them (parameter conversion and passing, function calls, and return value conversion). Note that our objective is not emulating/executing the entire target program, but instead focusing on only the unpacking work until the OEP (Original Entry Point) is met. When execution reaches the OEP, x64Unpack notifies this event and then automatically dumps the memory and reconstructs the unpacked version of the target system. Hence, we can focus on dealing with widely used API functions in unpacking routines in protectors, not all the API functions in Windows systems, which relieves burden of work.

When unregistered functions from the external DLL are called, x64Unpack produces a warning message, and emulates the instructions of the function body just like in the case for the target program emulation.

F. THREAD HANDLING

In x64Unpack, the main thread is responsible for handling the emulator GUI (for controlling x64Unpack). Thread 0 is responsible for CPU emulation, which is for the target program. Since we want to make emulation environments very close to those of real execution, x64Unpack shares PEB (Process Environment Block) and TEB (Thread Environment Block) information between the main thread and thread 0, which is shown in Fig. 6.

When the CreateThead() API function is called in the target code, x64Unpack calls CreateThread() to the operating system to create a new thread (Thread 1). This thread has its own TEB while PEB is shared with the main thread (since all threads belong to the single process).

Later, if CreateThread() is called again in the target program, another thread (Thread k, $k > 1$) is created with its own TEB. PEB of Thread k is the same as that of the main thread.

x64Unpack hooks every memory access to monitor PEB/TEB access (to detect and avoid anti-debugging routines) and take appropriate action as needed.

In x64Unpack, the time stamp value is derived from the virtual tick counter. This counter is incremented by 1 whenever a single instruction in Thread 0 is emulated. For time synchronization between threads, we use the round-robin

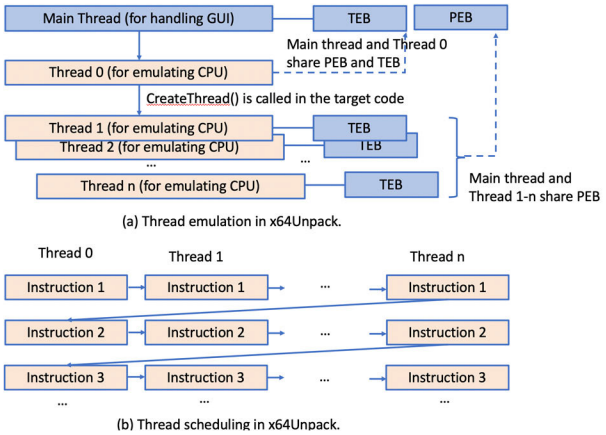

FIGURE 6. Thread emulation and thread scheduling in x64Unpack.

scheduling. First, we execute one instruction in Thread 0 and then run one instruction of all other threads (to do this we use thread-interlocking for synchronization). After finishing emulation of all other threads, x64Unpack increments the virtual tick counter by 1 and emulates the next instruction in Thread 0.

When the Sleep() function is called in the thread, x64Unpack reduces the number of running threads (nRunningThread) by one. Then, it excludes the corresponding thread in the ready thread list. Likewise, if the WaitForSingleObject() function is called on the thread, we exclude the corresponding thread in the list.

G. FINDING OEP (ORIGINAL ENTRY POINT)

We suppose that the target program was packed using the protector. Recall that OEP (Original Entry Point) is the address indicating the beginning point of the original code which is unpacked and loaded into the memory.

We have implemented a routine for finding OEP, which is based on the algorithm in [14]. x64Unpack finds the case where the instruction pointer jumps to the memory region which has been written after the start-up. If found, x64Unpack regards this address as an OEP candidate. Generally, x64Unpack finds many OEP candidates because packers use the self-modifying code technique. Hence, we use the heuristic algorithm [12] to refine OEP candidates.

H. EXCEPTION HANDLING

Since many protectors use anti-debugging techniques related to exceptions or interrupts, emulation of exception handling routines is essential. In 64-bit environments, for handling exceptions, Microsoft Windows uses VEH (Vectored Exception Handling), which is an extension of Structured Exception Handling (SEH) (that is used in 32-bit Windows).[1] VEH is similar to SEH, with the following differences: First, handlers are not tied to specific functions nor to stack frames. Second, exception handlers are explicitly added by calling the API function (AddVectoredExceptionHandler()) rather than as a byproduct of try/catch statements. After calling this function, NTDLL adds a new handler to the vectored exception handler list, which has a circular linked structure in the heap.

If an exception is raised, the operating system catches and forwards it to KiUserExceptionDispatcher() (in NTDLL), which calls RtlDispatchException Then, RtlCallVectoredExceptionHandlers() is called, which finds the appropriate handler in the handler list and executes it. (internals are very undocumented.)

We have implemented exception-related routines for VEH: exception registration, exception handling, return after exception handling, context saving and context restoration. Further, x64Unpack can automatically detect and log exception-based anti-debugging techniques.

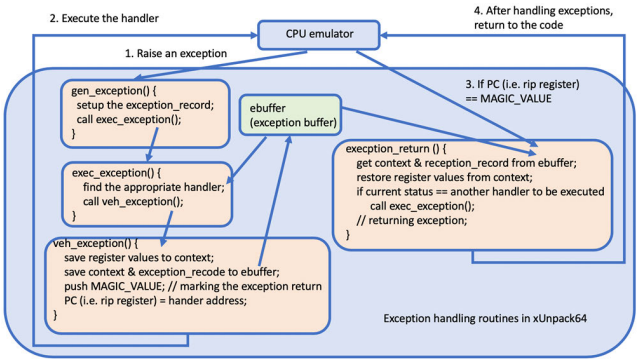

FIGURE 7. Exception handling routines in x64Unpack.

Fig. 7. shows exception handling procedures in x64Unpack. First, the CPU emulator raises an exception. Then, x64Unpack setups the exception record and finds the appropriate handler. It calls veh_exception(), where it saves all registers to the context structure and adds the context and the exception record into ebuffer. This buffer has a circular linked list in the heap to maintain all exception handlers information. Then, x64Unpack pushes MAGIC_VALUE into the virtual stack to specify the return to the code after handling the exception. Finally, it changes the program counter (RIP register) as the starting address of the handler.

Second, control flow goes back to the CPU emulator, which emulates the appropriate exception handler code.

Third, if the RIP register is equal to MAGIC_VALUE, this means the end of execution of the exception handler code and x64Unpack calls exception_return It restores the context and exception record from ebuffer and restores the saved register context. If there is another exception handler to be executed, it calls exec_execption

Forth, after handling exceptions, control flow goes back to the instruction (in the target code) that originally invoked the exception.

[1] Precisely speaking, 64-bit Windows operating systems uses 64-bit SEH and VEH. For implementation of 64-bit SEH, we omit explanation for lack of space.

I. LOADING THE TARGET PROGRAM INTO THE MEMORY

In Microsoft Windows, the file format for the executable binary program is PE (Portable Executable), which contains the information necessary for the loader.

Our PE loader works as follows. 1. It parses the PE file. 2. From the PE header, important information is obtained (the entry point, heap sizes, and stack sizes, etc.). 3. It iterates through each section and copies it from the file to the memory. 4. It adjusts entries in the symbol tables. 5. After that, x64Unpack starts the emulation process by creating a main emulation thread (Thread 0) and by fetching the instruction from the starting address of the target program.

Because (in Windows environments) the executable and DLL have the same structure, this PE loader is also used for loading external DLL files. When the target program explicitly loads the DLL by calling functions such as LoadLibrary(), we hook the function call and use this PE loader to load the corresponding DLL file.

J. IMPLEMENTATION

We have implemented x64Unpack using Qt 5.6 [15] and Visual Studio 2015 Professional on Microsoft Windows 10 64-bit environment. For CPU emulation, we have adopted CPU emulation part from the Bochs [10].

For handling API functions, we have implemented 196 API functions (4 GUI-related functions: MessageBoxA(), MessageBoxW(), MessageBoxExA(), GetWindow()), which are widely used by the (commercial) protectors for unpacking, decompressing, or anti-reverse engineering in 64-bit environments.

x64Unpack has the GUI interface to output analysis results, to dump memory, or to produce the unpacked executable file. Further, the user can set the specific memory region for tracing the instruction execution for deep analysis. Fig. 8 is an example for the screenshot of x64Unpack, where the complete log file is available at https://drive.google.com/open?id=11Pt9Y1kUSoasxleJsDwsqIsBPioCl31r. The log contains the following information: executed instructions, memory reads/writes, API function calls, memory region information, exception, and thread information.

III. EXPERIMENTAL RESULTS

To verify the effectiveness of the proposed scheme, we selected the following protectors.

- VMProtect: Version 3.4 [2]
- Themida: Version 2.4.6 [1]
- UPX: Version 3.95 [13]
- MPRESS 2.19

UPX is well-known and one of the most widely used packers in Microsoft Windows environments. MPRESS is another free high-performance executable packer for 32-bit/64-bit environments. VMProtect and Themida are considered as some of the most advanced, complex protectors in 64-bit Microsoft Windows environments. To the best of our knowledge, there has been no analysis results published for the recent version of VMProtect.

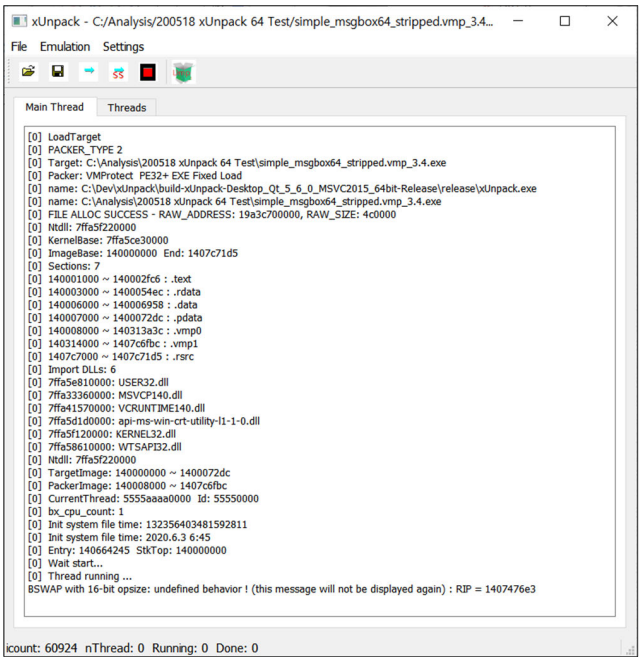

FIGURE 8. The graphical user interface of x64Unpack.

For VMProtect, we have used the following options: Memory Protection, Import Protection, Resource Protection, and Pack the Output File. Memory Protection is the feature that detects memory tampering. Import Protection is the feature that obfuscates import regions. Resource Protection encrypts the resource area. Pack the Output File is the feature that compresses executable files.

The test program uses some simple API functions related to the message box and strings: the program generates a random number, outputs a message box, and then exits. We have compiled it with Visual Studio 2015 in the release mode and default options.

After packing the above test program with 4 packers, we have tested x64Unpack with the packed files. We have confirmed that x64Unpack successfully executes them: bypassing anti-reverse engineering techniques, entering the OEP, displaying the message box, and then terminating the program.

We conducted two experiments for measuring the average elapsed time. In the first experiment, we chose a simple test program that runs 1000*1000 simple loops and then calls puts() function 1000 times. For comparison, we executed the test program (i.e., real execution) and then ran it under Pin [8], x64Unpack, and x64dbg [17]. For fair comparison, we set Pin, x64Unpack, and x64Dbg such that they produce instruction traces. We repeated the work for 30 times to get the average value. Fig. 9-(a) shows the average execution time. In this experiment, x64dbg runs the slowest because x64Dbgscript is slow. Pin is faster than x64Unpack since it uses caching for efficiency. We also measured memory usage where that of raw execution, x64Unpack, Pin, and x64dbg are 0.478 MBytes, 23.910 MBytes, 73.952 MBytes, and 8.1742 MBytes, respectively.

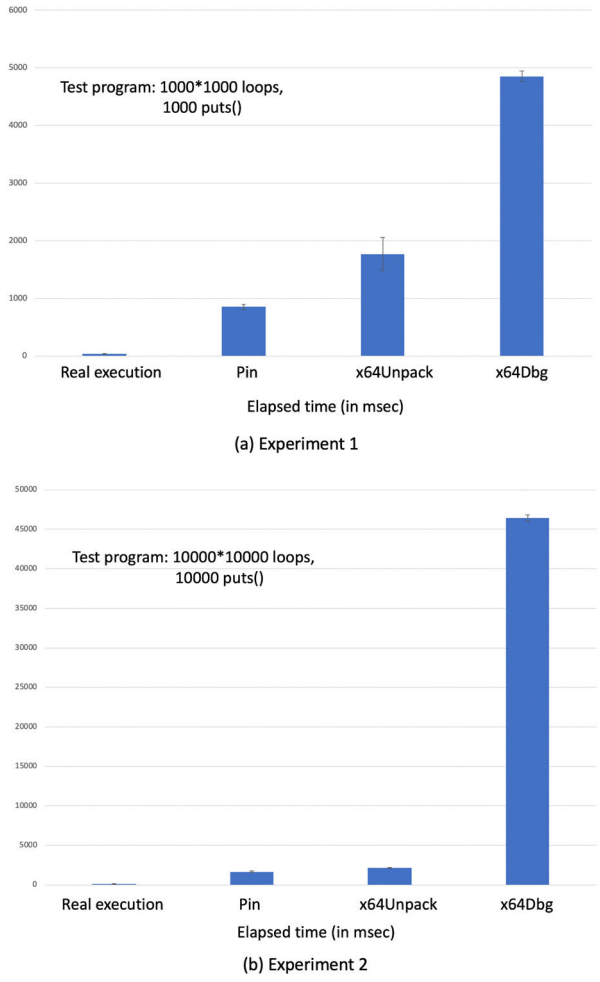

FIGURE 9. Elapsed time comparison for 2 Experiments.

In the second experiment, we used the modified test program, which runs 10000*10000 simple loops and then calls puts() function 10000 times. As seen in Fig. 9-(b), x64Dbg runs the slowest because x64Dbgscript inserts the break point for each instruction in the target program, which is very slow. Similar to Experiment 1, x64Unpack is slower than Pin. Note that x64Unpack focuses on analyzing the unpacking routine only (not the entire program). Hence, we believe that performance is the second issue compared with the correct execution of the unpacking routine having diverse anti-reverse engineering techniques. We also measured memory usage where that of raw execution, x64Unpack, Pin, and x64dbg are 0.474 MBytes, 24.221 MBytes, 73.800 MBytes, and 8.195 MBytes, respectively.

IV. DETAILED ANALYSIS RESULTS ON THE PACKED FILES BY VMProtect 3.4

This section summarizes the analysis results on the packed files by VMProtect 3.4. First, we have obtained the trace information using x64Unpack and then have analyzed the trace information, e.g., to find which anti-reverse engineering techniques are used, and when the compressed code is uncompressed.

We use the following notations throughout this section. PEHdr means the PE file header, Pker means memory region w.r.t. the packer, and Target means the memory region w.r.t. the target program. Rx and Wx denote the memory read and write, respectively where x is either b (8bits), w (16bits), d (32bits), or q (64bits). Mod means a Windows module (exe, dll, ocx, sys files, etc.). Heap_addr means the allocated heap memory area and addr means the postfix four digits of the memory address.

Section IV-A compares the section structures of the executable files for the original version and the packed version. Section IV-B explains API obfuscation techniques and Section IV-C describes the unpacking procedure of the obfuscated file.

A. COMPARISON RESULTS ON THE SECTIONS IN THE PE FILE

For the original file and the packed file, we compare section information as follows. Fig. 10 shows the section information for the original file and the packed version (VMProtect 3.4). Depending on option settings, VMProtect splits the target file into multiple sections or keeps as a single section. In the default settings, VMProtect adds 2 sections (.vmp0 and .vmp1) for the unpacking routine.

Original version	Packed version (by VMProtect 3.4)
Sections: 6 140001000 ~ 140002fc6 : .text 140003000 ~ 1400054ec : .rdata 140006000 ~ 140006958 : .data 140007000 ~ 1400072dc : .pdata 140008000 ~ 1400081e0 : .rsrc 140009000 ~ 14000907c : .reloc	Sections: 8 140001000 ~ 140002fc6 : .text 140003000 ~ 1400054ec : .rdata 140006000 ~ 140006958 : .data 140007000 ~ 1400072dc : .pdata 140008000 ~ 14031f2f3 : .vmp0 140320000 ~ 1407bf660 : .vmp1 1407c0000 ~ 1407c0094 : .reloc 1407c1000 ~ 1407c11d5 : .rsrc

FIGURE 10. The section structure of the original executable file and the packed version.

As seen in Fig. 10, the packed version has the same address and name for the following sections: .text, .rdata, .data and .pdata sections. As mentioned previously, .vmp0 and .vmp1 contain code and data for the unpacking routine and program protection. .reloc contains data that is not related to the original program, and .rsrc has the same name and content as the original .rsrc section (unless resource encoding is set in the options). Before execution, the .text, .rdata, .data, .pdata, and .vmp0 sections are empty and they are filled during the unpacking procedure.

If we execute the packed file, after unpacking is finished, the obfuscated code and data are restored to their original sections and then the original code is executed.

B. ANALYSIS OF THE API OBFUSCATION TECHNIQUES

Generally, (commercial) protectors use diverse API obfuscation techniques to deter analysis on API function calls, where major 3 of them [11] are as follows.

- Obfuscation of call/jmp instructions that direct to the beginning of the API functions. (1st method)
- Obfuscation for the Import Address Table (IAT). (2nd method)
- Obfuscation for either some part or the whole code of the API function body. (3rd method)

Among them, VMProtect uses the first one only. In VMProtect, each call or jmp instruction for the API function call is converted into several instructions having the same semantics (1st method). For this case, jmp/call goes across the different sections, which is seen in Fig. 11.

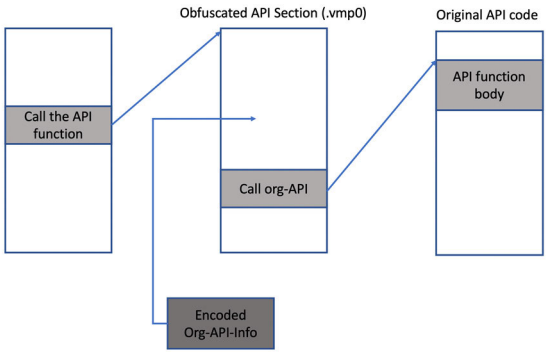

FIGURE 11. Obfuscated API function calls in VMProtect.

In the figure, 'Call the API function' means the instruction for calling the API function. The 'Obfuscated API Section (.vmp0)' contains the obfuscation-related instructions, which are added by VMProtect. This section contains many branch instructions for code obfuscation. 'Encoded Org-API-Info' indicates that the encoded address of the API function body, where the original API function address is encoded. In calling the obfuscated API function, this value is decoded and then used to jump to the beginning point of the original API function body.

C. ANALYSIS RESULTS ON THE EXECUTION OF THE PACKED FILE (VMProtect 3.4)

When we execute the obfuscated executable file that was packed with VMProtect 3.4, the unpacking routine conducts the following work until the OEP (Original Execution Point) is met (after that, code of the original file will be executed). These functionalities are arranged in order of execution.

1) CALLING GetModuleHandleA() TO GET THE HANDLES FOR Kernel32.DLL AND NTDLL.DLL

First, VMProtect calls the GetModuleHandleA() function, which is shown in Fig. 12. The packed executable file invokes API functions, which are not listed in the import section (because import section is obfuscated). To retrieve API function addresses, in the run-time, the unpacking routine first calls GetModuleHandleA(), which returns the module handles of Kernel32.dll and NTDLL.dll. Then, it computes the addresses of the API functions using these handles.

```
[0]  Wait start...
[0]  Thread running ...
BSWAP with 16-bit opsize: undefined behavior ! (this message will not
be displayed again) : RIP = 1407476e3
[0]  First API at 1274c (75596)
[0]  LocalAlloc: Flags = 0  Bytes =  160: HLocal = 2130ae4a710
[0]  GetModuleHandleA called kernel32.dll: 7ffc97990000
[0]  GetModuleHandleA called ntdll.dll: 7ffc994a0000
```

FIGURE 12. Calling GetModuleHandleA.

```
[0]  read Qword GS:[60]=b8ba969000
[0]  PEBrw: b8ba969120 = 47ba
[0]  PEBrw: b8ba969120 = 47ba
```

FIGURE 13. Checking the build number of the operating system.

2) CHECKING THE BUILD NUMBER OF THE OPERATING SYSTEM

The unpacking routine guesses the correct build number of the operating system, as follows. (This is because the anti-debugging techniques are very sensitive with the version of the target operating systems.) In the execution trace of Fig. 13, the unpacking routine gets the address of PEB by reading the memory cell at GS:[60] and then reads the content at PEB + 0×120 to retrieve the build number. In this figure, 0×47ba represents that the build number is Windows 10 1809.

3) CHECKING THE INTEGRITY OF THE PE HEADER

The unpacking routine reads the PE Header and checks whether the content of the PE header has been modified or not.

4) CHECKING THE INTEGRITY OF THE CONTENT OF THE EXECUTABLE FILE

The unpacking routine reads and checks whether the content of the executable file has been modified or not, i.e., it reads the file in the disk and checks whether the file contents are equal to what are currently loaded in memory or not. By calling GetModuleFileNameW(), it gets the name of the executable file. It opens the file by calling ZwOpenFile Then, the entire file contents are read into memory through ZwCreateSection() and ZwMapViewOfSection These contents are used to check whether the loaded code/data in the memory is tampered or not. If it has been tampered with, a message box would pop up for a warning (Fig. 16) and the program terminates.

5) TARGET CODE RESTORATION

The unpacking routine decodes the target code and the obfuscated API function code and then writes them to the memory, i.e., conducting the unpacking procedure. Before the unpacking, the .text and .vmp0 sections are filled with zeroes. Because the memory sections for the target code are prohibited for the write operation, the unpacking routine gives the write permission into each memory section using ZwProtectVirtualMemory Then, it reads the encoded area in the file and writes each decoded byte to .text and .vmp0 section.

```
[0]  PEHdr: Rb: 140000040 = e
[0]  PEHdr: Rb: 140000041 = 1f
[0]  PEHdr: Rb: 140000042 = ba
[0]  PEHdr: Rb: 140000043 = e
[0]  PEHdr: Rb: 140000044 = 0
[0]  PEHdr: Rb: 140000045 = b4
[0]  PEHdr: Rb: 140000046 = 9
[0]  PEHdr: Rb: 140000047 = cd
[0]  PEHdr: Rb: 140000048 = 21
[0]  PEHdr: Rb: 140000049 = b8
[0]  PEHdr: Rb: 14000004a = 1
[0]  PEHdr: Rb: 14000004b = 4c
[0]  PEHdr: Rb: 14000004c = cd
[0]  PEHdr: Rb: 14000004d = 21
```

FIGURE 14. Reading and checking the PE header.

```
[0]  GetModuleFileNameW: hmod = 140000000 size = fe
[0]  GetModuleFileNameW: 140000000, C:\Dev\xUnpack\build\simple_msgbox64_stripped.vmp
[0]  ZwOpenFile: 13ffffb28 80100080 13ffffb38 13ffffd48 3 60
[0]  File Name from Handle 4a4: \\?\C:\Dev\xUnpack\build\simple_msgbox64_stripped.vmp
[0]  NTSTATUS: 0, HANDLE:4a4
[0]  ZwCreateSection: 13ffffb08 4 13ffffb38 0 2 8000000 4a4
[0]  File Name from Handle 4a4: \\?\C:\Dev\xUnpack\build\simple_msgbox64_stripped.vmp
[0]  NTSTATUS:0, SECTION HANDLE:49c
[0]  ZwMapViewOfSection: 49c ffffffffffffffff 13ffffa30 0 0 13ffffb00 13ffffb18 1 0 2
[0]  BaseAddress:220fd2a0000
[0]  pBaseAddress:13ffffa30
[0]  BaseAddress:220fd2a0000
[0]  ViewSize:4c0000
[0]  NTSTATUS:0
[0]  LastError:0

[0]  File: Rb: 220fd2a0001 = 5a
[0]  File: Rb: 220fd2a0002 = 90
[0]  File: Rb: 220fd2a0003 = 0
[0]  File: Rb: 220fd2a0004 = 3
[0]  File: Rb: 220fd2a0005 = 0
[0]  File: Rb: 220fd2a0006 = 0
[0]  File: Rb: 220fd2a0007 = 0

[0]  File: Rb: 220fd352a97 = 35
[0]  File: Rb: 220fd352a98 = 44
[0]  File: Rb: 220fd352a99 = 8f
[0]  File: Rb: 220fd352a9a = 66
[0]  File: Rb: 220fd352a9b = 32
[0]  File: Rb: 220fd352a9c = 82
[0]  File: Rb: 220fd352a9d = 6f
[0]  File: Rb: 220fd352a9e = f6
[0]  ZwUnMapViewOfSection: ffffffffffffffff 220fd2a0000
[0]  NTSTATUS:c0000019, LastError:0
```

FIGURE 15. Checking the integrity of the content of the executable file.

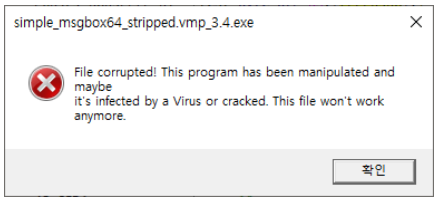

FIGURE 16. A pop-up message box showing that the file has been corrupted.

6) CHECKING CODE-INJECTION

Reverse engineers often inject their code chunks for analysis in the new memory section. VMProtect checks this code-injection as follows. It first gets the size of the image from the PE header and then calls ZwQueryVirtualMemory() with the end address that is calculated by adding the base address and the size of the image. If there is any memory region that is already allocated at the end of the program, it regards this is the code-injection, pops up the message box and exits the program.

```
[0]  haeZwProtectVirtualMemory: ffffffffffffffff 13ffffaa8 13ffffc18 40 13ffffa78
[0]  BaseAddress: 140008000, NumberOfBytesToProtect:30c000, NewAccessProtection:40,
[0]  haeZwProtectVirtualMemory: ffffffffffffffff 13ffffaa8 13ffffc18 40 13ffffa78
[0]  BaseAddress: 140001000, NumberOfBytesToProtect:2000, NewAccessProtection:40, O
[0]  haeZwProtectVirtualMemory: ffffffffffffffff 13ffffaa8 13ffffc18 4 13ffffa78
[0]  BaseAddress: 140003000, NumberOfBytesToProtect:3000, NewAccessProtection:4, Ol
[0]  haeZwProtectVirtualMemory: ffffffffffffffff 13ffffaa8 13ffffc18 4 13ffffa78
[0]  BaseAddress: 140007000, NumberOfBytesToProtect:1000, NewAccessProtection:4, Ol
[0]  haeZwProtectVirtualMemory: ffffffffffffffff 13ffffaa8 13ffffc18 4 13ffffa78
[0]  BaseAddress: 140038000, NumberOfBytesToProtect:6000, NewAccessProtection:4, Ol
[0]  LocalAlloc: Flags = 0  Bytes =  3e6c: HLocal = 220f462eff0
[0]  Target: Wb: 140001000 = 48
[0]  Target: Wb: 140001001 = 8b
[0]  Target: Wb: 140001002 = c4
[0]  Target: Wb: 140001003 = 55
[0]  Target: Wb: 140001004 = 48
[0]  Target: Wb: 140001005 = 8d
[0]  Target: Wb: 140001006 = 68
[0]  Target: Wb: 140001007 = 88
[0]  Target: Wb: 140001008 = 48
[0]  Target: Wb: 140001009 = 81
[0]  Target: Wb: 14000100a = ec
[0]  Target: Wb: 14000100b = 70
[0]  Target: Wb: 14000100c = 1
[0]  Target: Wb: 14000100d = 0
[0]  Target: Wb: 14000100e = 0
[0]  Target: Wb: 14000100f = 48
[0]  Target: Wb: 140001010 = c7
[0]  Target: Wb: 140001011 = 44
[0]  Target: Wb: 140001012 = 24
```

FIGURE 17. Decoding procedure in VMProtect.

```
[0]  PEHdr: Rd: 14000003c = 80
[0]  PEHdr: Rd: 1400000d0 = 7c8000 SizeOfImage
[0]  ZwQueryVirtualMemory: Handle = ffffffffffffffff BaseAddr =
```

FIGURE 18. Checking the injected code chunks.

```
[0]  GetProcessAffinityMask: Handle = ffffffffffffffff ProcessAM = fffff
[0]  SetThreadAffinityMask: hThread = fffffffffffffffe Mask_ptr = 1: fff
[0]  Sleep: 0
[0]  SetThreadAffinityMask: hThread = fffffffffffffffe Mask_ptr = fffff
[0]  SetThreadAffinityMask: hThread = fffffffffffffffe Mask_ptr = 2: fff
[0]  Sleep: 0
[0]  SetThreadAffinityMask: hThread = fffffffffffffffe Mask_ptr = fffff
[0]  SetThreadAffinityMask: hThread = fffffffffffffffe Mask_ptr = 4: fff
[0]  Sleep: 0
[0]  SetThreadAffinityMask: hThread = fffffffffffffffe Mask_ptr = fffff
[0]  SetThreadAffinityMask: hThread = fffffffffffffffe Mask_ptr = 8: fff
[0]  Sleep: 0
[0]  SetThreadAffinityMask: hThread = fffffffffffffffe Mask_ptr = fffff
[0]  SetThreadAffinityMask: hThread = fffffffffffffffe Mask_ptr = 10: ff
[0]  Sleep: 0
[0]  SetThreadAffinityMask: hThread = fffffffffffffffe Mask_ptr = fffff
[0]  SetThreadAffinityMask: hThread = fffffffffffffffe Mask_ptr = 20: ff
[0]  Sleep: 0
```

FIGURE 19. Setting the thread affinity.

7) SETTING THE THREAD AFFINITY

The routine sets thread affinity to use all CPU cores (in Fig. 19). This may prevent or make it difficult analysis by fixing thread affinity.

8) ANTI-DEBUGGING TECHNIQUE FOR CHECKING THE ELAPSED TIME

It calls time-related API functions. This is for measuring the elapsed time for detecting debuggers/analysis tools. If the elapsed time is too long, it regards as being analyzed and it terminates execution.

```
[0]  PEBrw: 9dea5df120 = 47ba
[0]  GetSystemTimeAsFileTime: 1d5b6434a68190b RIP: 14012902d
[0]  GetCurrentProcessId: 66660000
[0]  GetCurrentThreadId: ret = 55550000
[0]  GetTickCount: 2eee
[0]  QueryPerfomanceCount: b7543d
```

FIGURE 20. Checking the elapsed time.

9) USING FIBER-LOCAL STORAGE TO KEEP THE ALLOCATED HEAP ADDRESS

A fiber is a particularly lightweight thread of execution. In Microsoft Windows, fibers run in the context of the threads where each thread can schedule multiple fibers. The unpacker routine calls fiber-related functions such as FlsGetValue(), FlsSetValue(), and FlsAlloc Also, it calls FlsSetValue(), which may mislead us into believing that an FLS based anti-debugging technique is used [5]. If properly used, FlsSetValue() can alter the control flow such that the debugger loses its control. However, VMProtect does not create any fiber at all. Instead, it uses FLS (Fiber Local Storage) as a storage to keep the pointer to the allocated memory region (which was allocated by calling RtlAllocateHeap() function), which is seen in Fig. 21-(a).

```
[0]  FlsAlloc: Callback: 140023c74 : index = 8
[0]  PEImg: Wd: 140038790 = D8
[0]  #RtlAllocateHeap: 27912880000, 8, 2c8: 27912880860
[0]  FlsSetValue: index = 8 val = 27912880860
```

(a) Using FLS to keep the pointer (pointing to the heap memory).

```
[0]  FlsGetValue: index = 8: ret = 27912880860
[0]  GetLastError: 0
[0]  Heap_0860 Wd: 27912880928 = 3
[0]  Heap_0860 Wd: 27912880928 = 1
[0]  GetLastError: 0
[0]  FlsGetValue: index = 8: ret = 27912880860
[0]  GetLastError: 0
[0]  Heap_0860 Wd: 27912880928 = 3
[0]  Heap_0860 Wd: 27912880928 = 1
[0]  GetLastError: 0
[0]  FlsGetValue: index = 8: ret = 27912880860
[0]  GetLastError: 0
[0]  Heap_0860 Wd: 27912880928 = 3
[0]  Heap_0860 Wd: 27912880928 = 1
[0]  GetLastError: 0
[0]  FlsGetValue: index = 8: ret = 27912880860
[0]  GetLastError: 0
[0]  Heap_0860 Wd: 27912880928 = 3
[0]  Heap_0860 Wd: 27912880928 = 1
```

(b) Accessing the pointer using FlsGetValue().

FIGURE 21. Using Fiber Local Storage (FLS) to keep allocated heap address.

Then, this memory address is retrieved many times by FlsGetValue() function (which is seen in Fig. 21-(b)).

10) ENCODING THE IMPORTANT POINTERS

VMProtect hides the important addresses by calling RtlEncodePointer() and RtlDecodePointer() functions. We think that VMProtect hides the starting addresses to API functions and to allocated memory blocks. E.g., in Fig. 22, addresses $0 \times 1400244c$ and 0×1400268.4 are the starting points of obfuscated calls to the API functions GetLastError() and RtlAllocateHeap(), respectively.

```
[0]  RtlEncodePointer: 14002444c ret = b897d5148000000
[0]  PEImg: Wq: 14003ba98 = b897d5148000000
[0]  RtlEncodePointer: 1400268a4 ret = b897c3608000000
[0]  PEImg: Wq: 140038920 = b897c3608000000
```

FIGURE 22. Encoding the pointers for the API functions.

```
[0]  #RtlAllocateHeap: 2122ac70000, 8, 100: 2122ac77800
[0]  RtlEncodePointer: 2122ac77800 ret = 403108e800001097
[0]  PEImg: Wq: 14003db30 = 403108e800001097
[0]  PEImg: Wq: 14003db28 = 403108e800001097
[0]  Heap_7800 Wq: 2122ac77800 = 0
[0]  PEImg: Wd: 14003d900 = 200
[0]  #RtlAllocateHeap: 2122ac70000, 8, 1000: 2122ac70b30
[0]  PEImg: Wq: 14003c8e8 = 2122ac70b30
[0]  Heap_0b30 Wq: 2122ac70b30 = 140038fa0
[0]  Heap_0b30 Wq: 2122ac70b38 = 140038fd0
[0]  Heap_0b30 Wq: 2122ac70b40 = 140039000
[0]  Heap_0b30 Wq: 2122ac70b48 = 140039030
[0]  Heap_0b30 Wq: 2122ac70b50 = 140039060
[0]  Heap_0b30 Wq: 2122ac70b58 = 140039090
[0]  Heap_0b30 Wq: 2122ac70b60 = 1400390c0
[0]  Heap_0b30 Wq: 2122ac70b68 = 1400390f0
[0]  RtlEnterCriticalSection called: 14003bd30
[0]  RtlDecodePointer: 403108e800001097 ret = 2122ac77800
[0]  RtlDecodePointer: 403108e800001097 ret = 2122ac77800
[0]  RtlSizeHeap: hHeap = 2122ac70000 Flags = 0 Mem = 2122ac77800: 100
[0]  RtlEncodePointer: 140027268 ret = 16195ba80000000c
[0]  Heap_7800 Wq: 2122ac77800 = 16195ba80000000c
```

FIGURE 23. Encoding the pointer for the heap memory region.

```
[0]  CreateToolhelp32Snapshot: PID = 0
[0]  Thread32First: Handle = 698 buffer = 13ffff4a0 : 1
[0]  dwSize=1c, th32ThreadID=0, th32OwnerProcessID=0, tpBasePri=0, tpDeltaPri=0
[0]  GetCurrentProcessId: 66660000
[0]  GetCurrentThreadId: ret = 55550000
[0]  Thread32Next: Handle = 698 buffer = 13ffff4a0 : 1
[0]  dwSize=1c, th32ThreadID=0, th32OwnerProcessID=0, tpBasePri=0, tpDeltaPri=0
[0]  Thread32Next: Handle = 698 buffer = 13ffff4a0 : 1
[0]  dwSize=1c, th32ThreadID=0, th32OwnerProcessID=0, tpBasePri=0, tpDeltaPri=0
[0]  Thread32Next: Handle = 698 buffer = 13ffff4a0 : 1
[0]  dwSize=1c, th32ThreadID=0, th32OwnerProcessID=0, tpBasePri=0, tpDeltaPri=0
[0]  Thread32Next: Handle = 698 buffer = 13ffff4a0 : 1
[0]  dwSize=1c, th32ThreadID=0, th32OwnerProcessID=0, tpBasePri=0, tpDeltaPri=0
[0]  Thread32Next: Handle = 698 buffer = 13ffff4a0 : 0
[0]  dwSize=1c, th32ThreadID=0, th32OwnerProcessID=0, tpBasePri=0, tpDeltaPri=0
[0]  CloseHandle: 698: 1
```

FIGURE 24. Traversing the threads.

Fig. 23 indicates that RtlEncodePointer() is used for encoding the address $0 \times 2122ac77800$, which is the starting address of the heap memory region created by calling RtlAllocateHeap() beforehand.

11) THREAD SNAPSHOT

VMProtect traverses the threads in the thread list using the following API functions: CreateToolhelp32Snapshot(), Thread32First() and Thread32Next If the target process is being debugged, traversing threads with Thread32Next() goes into the infinite loop.

12) CHECKING INTEGRITY OF THE MEMORY REGION

VMProtect installs a trampoline at the start of the ZwProtectVirtualMemory() function to monitor the .vmp0 section of the obfuscated API calls. Every time ZwProtectVirtualProtect() is called, it jumps to a chuck of code, which checks the integrity of .vmp0 section and then executes the original ZwProtectVirtualProtect() function and returns to the caller. To install a trampoline, VMProtect allocates a heap memory region and overwrites the original two instructions in ZwProtectVirtualMemory() function body.

```
[0] VirtualAlloc: addr = 7ff8f6a90ab0 size = 1000 type = 3000 prot = 40: Er
[0] Heap_6650 Wq: 1b062346660 = 7ff8f6a90000
[0] Heap_7920 Wq: 1b062347998 = 7ff8f6a90000
[0] Heap_0000 Wb: 7ff8f6a90002 = d1
[0] Heap_0000 Wb: 7ff8f6a90001 = 8b
[0] Heap_0000 Wb: 7ff8f6a90000 = 4c
[0] Heap_0000 Wb: 7ff8f6a90007 = 0
[0] Heap_0000 Wb: 7ff8f6a90006 = 0
[0] Heap_0000 Wb: 7ff8f6a90005 = 0
[0] Heap_0000 Wb: 7ff8f6a90004 = 50
[0] Heap_0000 Wb: 7ff8f6a90003 = b8
[0] Heap_6650 Wq: 1b062346650 = 8
[0] GetCurrentProcess: 7777000000000000
[0] WriteProcessMemory: Handle = 7777000000000000 BaseAddr = 7ff8f6a90008 B
[0] E9 AB CA EA FF
[0] Heap_0000 Wb: 7ff8f6a90014 = 0
[0] Heap_0000 Wd: 7ff8f6a90010 = 50b8
[0] Heap_0000 Wb: 7ff8f6a9000f = d1
[0] Heap_0000 Wb: 7ff8f6a9000e = 8b
[0] Heap_0000 Wb: 7ff8f6a9000d = 4c
[0] GetCurrentProcess: 7777000000000000
[0] WriteProcessMemory: Handle = 7777000000000000 BaseAddr = 7ff8f6a90015 B
[0] FF 25 00 00 00 00
[0] Heap_0000 Wq: 7ff8f6a9001b = 14000de00
[0] GetCurrentProcess: 7777000000000000
[0] WriteProcessMemory: Handle = 7777000000000000 BaseAddr = 7ff8f693cab0 B
[0] E9 60 35 15 00
[0] Wq: 1b062346658 = 7ff8f693cab0 ZwProtectVirtualMemory
```

FIGURE 25. Installing a trampoline at ZwProtectVirtualMemory.

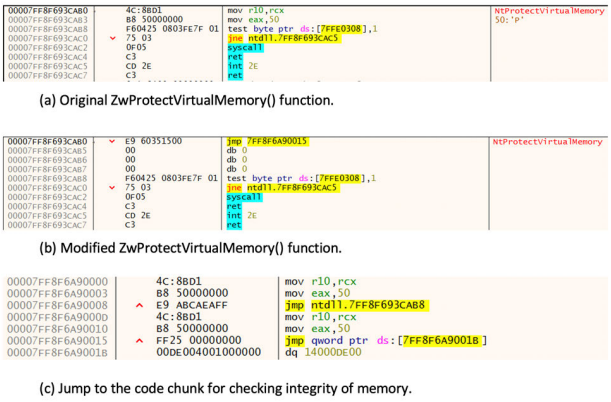

FIGURE 26. After modifying ZwProtectVirtualMemory() function, Jumping to the code chunk for checking integrity of memory.

Fig. 25 shows how trampoline code is written during execution. Fig. 26 shows the modified code for the trampoline. As seen in Fig. 26-(a) and Fig. 26-(b), ZwProtectVirtualMemory() is patched to redirect to $0 \times 7ff8f6a90015$ at the newly allocated heap address that contains a jump instruction to the integrity check code. Fig. 26-(c) shows the newly allocated memory region (which was written in the trace with the tag: Heap_0000 in Fig. 25) using VirtualAlloc This region contains the first two instructions copied from ZwProtectVirtualMemory When ZwProtectVirtualMemory() is called, it executes 'jmp 7ff8f6a90015; jmp [7ff8f6a9001b]' and the integrity-checking code starting at $0 \times 14000de00$. After that, it returns to the $0 \times 7ff8fa9000$ – the start address of the Heap_0000 – and executes the copied two instructions from ZwProtectVirtualMemory() and then returns to $0 \times 7ff8f693cab8$ that is the third instruction of the original ZwProtectVirtualMemory

If we trace the memory reads, we can see that execution iterates over the checksum of memory in the obfuscation API area as in Fig. 27. (It scans the .data section and .vmp0 section and then checks the integrity of the program.)

```
[0] Patched API Call at 1406f9ec5 func: 220fc5d2630 RIP: 7ff8f693cab0
[0] Rq: 7ff8f6a9001b = 14000de00
[0] Rd: 14006f61f = 9f0cf523
[0] Rb: 14006f61e = a1
[0] Rd: 14006f61a = 37d1b9c3
[0] Rb: 14006f619 = 89
[0] Rd: 14006f615 = dedd6127
[0] Rb: 14006f614 = c8
[0] Rd: 14006f610 = 534f3630
[0] Rd: 140107689 = b4d8089b
[0] Rb: 14010768d = 76
[0] Rd: 14010768e = 4b270c10
[0] Rb: 140107692 = 34
[0] Rd: 140107693 = 35c915b0
[0] Rb: 140107697 = 81
[0] Rd: 140107698 = 35c18a68
[0] Rb: 14010769c = 80
[0] Rd: 14010769d = 34dbff8a
[0] Rb: 1401076a1 = 38
[0] Rd: 1401076a2 = ca3f9f47
[0] break: StopExec: 7ff8f693cac0 ticks = 2dd1ee35a icount = 494fe389
```

FIGURE 27. The integrity-checking routine in the trace.

13) BRANCH TO OEP

Finally, it restores memory access permission of each section to the original value and then branches to OEP, which is shown in Fig. 28. (after that, code of the original file will be executed).

```
[0] haeZwProtectVirtualMemory: ffffffffffffffff 13ffffbf8 13ffffc20 20 13fffa68
[0] BaseAddress: 140008000, NumberOfBytesToProtect:30c000, NewAccessProtection:20,
[0] haeZwProtectVirtualMemory: ffffffffffffffff 13ffffbf8 13ffffc20 20 13fffa68
[0] BaseAddress: 140001000, NumberOfBytesToProtect:2000, NewAccessProtection:20, (
[0] haeZwProtectVirtualMemory: ffffffffffffffff 13ffffbf8 13ffffc20 2 13fffa68
[0] BaseAddress: 140003000, NumberOfBytesToProtect:3000, NewAccessProtection:2, O
[0] haeZwProtectVirtualMemory: ffffffffffffffff 13ffffbf8 13ffffc20 2 13fffa68
[0] BaseAddress: 140007000, NumberOfBytesToProtect:1000, NewAccessProtection:2, O
[0] haeZwProtectVirtualMemory: ffffffffffffffff 13ffffbf8 13ffffc20 4 13fffa68
[0] BaseAddress: 140038000, NumberOfBytesToProtect:6000, NewAccessProtection:4, O
[0] !!Target fetch: OEP (Next) = 14000216c icount: 49488e12, stack = 140000000
[0] break: StopExec: 14000216c ticks = 2dcd58cb4 icount = 49488e12
[0] Wait start...
```

FIGURE 28. Branch to OEP.

V. RESTORING PROCESS FOR THE PACKED FILES

This section describes the process of restoring packed files from VMProtect 3.4 to obtain the unpacked version. Section V-A describes how to restore the import sections and Section V-B explains how to create the PE file. x64Unpack can automate these processes to get the unpacked version.

A. DEOBFUSCATING OBFUSCATED API FUNCTION CALLS

VMProtect deletes IAT section in the packing process. Fig. 29-(a) and Fig. 29-(b) describe the differences between the IAT of the original executable file and the corresponding memory region when OEP is met. This region will be used for reconstruing the IAT table.

In VMProtect, API function calls are also obfuscated. Fig. 30 shows an example for how an original API function call is obfuscated. The instruction at 0×1400011.7 'call cs:MessageBoxA' is obfuscated to 'call sub_140150aa3'.

The target address of the obfuscated call instruction points to the .vmp0 section. The instructions from the address $0 \times 140150aa3$ to $0 \times 14007.21d$ in Figure 31 are equivalent to the instruction 'call MessageBoxA'.

The obfuscated instructions can be optimized using code optimization techniques such as dead code elimination, constant propagation, constant folding and peephole optimization. We explain the procedure on the code in Fig. 31.

```
.idata:0000000140003168 ; Imports from USER32.dll
.idata:0000000140003168 ;
.idata:0000000140003168 ; int __stdcall MessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType)
.idata:0000000140003168                 extrn MessageBoxA:qword ; CODE XREF: sub_140001000+1A71p
.idata:0000000140003168                                         ; DATA XREF: sub_140001000+1A71r ...
.idata:0000000140003170
.idata:0000000140003178 ;
.idata:0000000140003178 ; Imports from VCRUNTIME140.dll
.idata:0000000140003178 ;
.idata:0000000140003178 ; void __stdcall __noreturn _CxxThrowException(void *pExceptionObject, _ThrowInfo *pThrowInfo)
.idata:0000000140003178                 extrn __imp__CxxThrowException:qword
.idata:0000000140003178                                         ; DATA XREF: _CxxThrowException↑r
.idata:0000000140003178                                         ; .rdata:0000000140004520↓o
.idata:0000000140003180                 extrn __imp___std_exception_destroy:qword
.idata:0000000140003180                                         ; DATA XREF: __std_exception_destroy↑r
.idata:0000000140003188 ; void *__cdecl memset(void *, int Val, size_t Size)
.idata:0000000140003188                 extrn __imp_memset:qword
.idata:0000000140003188                                         ; DATA XREF: memset↑r
```

(a) IAT of the original executable file for some functions.

```
.rdata:0000000140003168                 dq 0E8F31130E1E24F8Fh
.rdata:0000000140003170                 dq 0
.rdata:0000000140003178                 dq 23FF83F10CDA41A8h
.rdata:0000000140003180                 dq 4B62D9D7CDDD725Dh
.rdata:0000000140003188                 dq 0A6861A0B9599256Ch
```

(b) IAT in the memory at OEP.

FIGURE 29. Differences between the IAT of the original executable file and the corresponding memory at OEP.

```
.text:0000000140001187 lea     rdx, [rsp+170h+lpText]
.text:000000014000118C cmp     qword ptr [rsp+170h+var_138+8], 10h
.text:0000000140001192 cmovnb  rdx, [rsp+170h+lpText] ; lpText
.text:0000000140001198 mov     r9d, 114h       ; uType
.text:000000014000119E lea     r8, Caption     ; "Lucky Day"
.text:00000001400011A5 xor     ecx, ecx        ; hWnd
.text:00000001400011A7 call    cs:MessageBoxA
.text:00000001400011AD mov     ebx, eax
```

(a) Original MessageBoxA() function call.

```
.text:0000000140001187                 lea     rdx, [rsp-8+arg_20]
.text:000000014000118C                 cmp     qword ptr [rsp-8+arg_30+8], 10h
.text:0000000140001192                 cmovnb  rdx, [rsp-8+arg_20]
.text:0000000140001198                 mov     r9d, 114h
.text:000000014000119E                 lea     r8, aLuckyDay   ; "Lucky Day"
.text:00000001400011A5                 xor     ecx, ecx
.text:00000001400011A7                 call    sub_140150AA3
.text:00000001400011AC                 int     3               ; Trap to Debugger
.text:00000001400011AD ; ---------------------------------------------------------------------------
.text:00000001400011AD                 mov     ebx, eax
```

(b) Obfuscated API function call.

FIGURE 30. Original MessageBoxA() function call vs. obfuscated function call.

First, meaningless jmp and nop instructions can be eliminated. The obfuscated instructions excluding nop and jmp are 'push rax; cdqe; lahf; mov rax, [rsp+8]; lea rax, [rax+1]; mov [rsp+8], rax; movsx eax, r15w; movsx ax, r1b; lea rax, 0 × 14000182d; mov rax, [rax+0 × 7.9f1]; lea rax, [rax+0 × 648.390a], xchg rax, [rax]; ret.'

Second, the dead code elimination algorithm can remove the instructions which are meaningless because registers are overwritten: 'cdqe; lahf; mov rax, [rsp+8]' instructions can be eliminated because 'lea rax, [rax+1]' overwrites rax. 'movsx eax, r15w; movsx ax, r1b' instructions are eliminated because 'lea rax, 0 × 14000182d' overwrites rax. The instructions after dead code eliminations are 'push rax; lea rax, [rax+1]; mov [rsp+8], rax; lea rax, 0 × 14000182d; mov rax, [rax+0 × 7.9f1]; lea rax, [rax+0 × 648.390a], xchg rax, [rax]; ret.'

Third, constant propagation is applied to 'lea rax, 0 × 14000182d; mov rax, [rax+0 × 7.9f1]', which are transformed into 'mov rax, [0 × 14007.21e]'. The value of [0 × 140007.21e] is 0 × 7ff891c8e426 and it is considered as a constant. Hence, the instruction is transformed into 'mov rax, 0 × 7ff891c8e426' where the address 0 × 7ff8f6571d30 is the start address of MessageBoxA() function.

Fourth, constant folding is applied to the instructions 'mov rax, 0 × 7ff891c8e426; lea rax, [rax+0 × 648.390a]' and

```
.vmp0:0000000140150AA3
.vmp0:0000000140150AA3
.vmp0:0000000140150AA3
.vmp0:0000000140150AA3 sub_140150AA3 proc near
.vmp0:0000000140150AA3
.vmp0:0000000140150AA3 var_8= qword ptr -8
.vmp0:0000000140150AA3
.vmp0:0000000140150AA3 ; FUNCTION CHUNK AT .vmp0:0000000140090354 SIZE 00000009 BYTES
.vmp0:0000000140150AA3 ; FUNCTION CHUNK AT .vmp0:00000001400D2D1A SIZE 00000010 BYTES
.vmp0:0000000140150AA3 ; FUNCTION CHUNK AT .vmp0:00000001401398AF SIZE 00000021 BYTES
.vmp0:0000000140150AA3 ; FUNCTION CHUNK AT .vmp0:000000014018B028 SIZE 0000000E BYTES
.vmp0:0000000140150AA3
.vmp0:0000000140150AA3 nop
.vmp0:0000000140150AA4 jmp     loc_14018B028
.vmp0:0000000140150AA4 sub_140150AA3 endp
.vmp0:0000000140150AA4
```

```
.vmp0:000000014018B028 ; START OF FUNCTION CHUNK FOR sub_140150AA3
.vmp0:000000014018B028
.vmp0:000000014018B028 loc_14018B028:
.vmp0:000000014018B028 push    rax
.vmp0:000000014018B029 cdqe
.vmp0:000000014018B02B lahf
.vmp0:000000014018B02C mov     rax, [rsp+8]
.vmp0:000000014018B031 jmp     loc_140090354
.vmp0:000000014018B031 ; END OF FUNCTION CHUNK FOR sub_140150AA3
```

```
.vmp0:0000000140090354 ; START OF FUNCTION CHUNK FOR sub_140150AA3
.vmp0:0000000140090354
.vmp0:0000000140090354 loc_140090354:
.vmp0:0000000140090354 lea     rax, [rax+1]
.vmp0:0000000140090358 jmp     loc_1401398AF
.vmp0:0000000140090358 ; END OF FUNCTION CHUNK FOR sub_140150AA3
```

```
.vmp0:00000001401398AF ; START OF FUNCTION CHUNK FOR sub_140150AA3
.vmp0:00000001401398AF
.vmp0:00000001401398AF loc_1401398AF:
.vmp0:00000001401398AF mov     [rsp+8], rax
.vmp0:00000001401398B4 movsx   eax, r15w
.vmp0:00000001401398B8 movsx   ax, r14b
.vmp0:00000001401398BD lea     rax, loc_14000182D
.vmp0:00000001401398C4 mov     rax, [rax+7B9F1h]
.vmp0:00000001401398CB jmp     loc_1400D2D1A
.vmp0:00000001401398CB ; END OF FUNCTION CHUNK FOR sub_140150AA3
```

```
.vmp0:00000001400D2D1A ; START OF FUNCTION CHUNK FOR sub_140150AA3
.vmp0:00000001400D2D1A
.vmp0:00000001400D2D1A loc_1400D2D1A:
.vmp0:00000001400D2D1A lea     rax, [rax+648E390Ah]
.vmp0:00000001400D2D21 xchg    rax, [rsp+8+var_8]
.vmp0:00000001400D2D25 jmp     nullsub_227
.vmp0:00000001400D2D25 ; END OF FUNCTION CHUNK FOR sub_140150AA3
```

```
.vmp0:000000014007D21D
.vmp0:000000014007D21D
.vmp0:000000014007D21D
.vmp0:000000014007D21D nullsub_227 proc near
.vmp0:000000014007D21D retn
.vmp0:000000014007D21D nullsub_227 endp
.vmp0:000000014007D21D
```

FIGURE 31. Obfuscated instructions equivalent to 'call MessageBoxA.'

they are transformed into 'mov rax, 0 × 7ff8f6571d30 (= the starting address of MessageBoxA).'

Finally, a peephole optimization is applied into 'push rax; lea rax, [rax+1]; mov [rsp+8], rax': These instructions increase the value of the stack top by 1 and push the value of rax on the top of the stack, which is equivalent to 'inc [esp]; push rax.' We use peephole optimization because this pattern is not common in compiler optimization techniques. 'push rax; mov rax, 0×7ff8f6571d30; xchg rax, [rax]' are optimized into 'push 0 × 7ff8f6571d30' and 'push 0 × 7ff8f6571d30; ret' are optimized into 'call 0 × 7ff8f6571d30' by peephole optimization.

Sometimes VMProtect uses indirect call/jmp for invoking API functions. E.g., first, instruction 'call A' is invoked, second, instruction 'jmp B' at the address A is executed, and then the function body at the address B is executed. We track the execution flow to analyze all indirect call/jmps to the API functions and then convert them as direct call/jmps.

To find obfuscated API calls, we disassemble the whole .text section and collect every call instruction that targets

into .vmp0 section. Instead of using code optimization technique, we use Run-until-API method [11] to find the original API function efficiently. We use x64Unpack to emulate each obfuscated call instruction until it reaches the API function body. After that we can deobfuscate API function calls and can reconstruct the IAT table.

B. CREATING THE PE FILE

We copy the original sections into the unpacked version. Then, we create an import section from the IAT information. We copy resource and TLS if present. All sections need to be repositioned. Finally, the PE header checksum is calculated using the CheckSumMappedFile() function.

VI. RELATED WORK

In the case of malware detection [18]–[21] and binary code analysis [22], [23] relevant studies have been actively conducted. However, anti-reversing techniques have not attracted special interest from researchers except for specific topics such as code virtualization [24], [25]. This section summarizes related research/tools on the detection and bypass of anti-reversing techniques, which can be classified into four categories: debugger, CPU emulator, DBI and unpacking. The details of each are as follows.

Debugger: One of the most widely used tools for dynamic analysis of binary code is the debugger. The debugger supports run-time disassembly for the target program and single-step execution of instructions. Execution context can also be monitored and immediately manipulated. The debugger allows the analyst to easily check how status is being changed as each instruction is executed. However, the debugging environment is quite different from the actual execution environment, and there are various anti-debugging techniques using this difference.

Ollydbg [7] is one of the most widely used debuggers for binary code analysis in Microsoft Windows environments, but it supports only 32-bit environments. WinDbg is the debugger made by Microsoft and has the advantage of being able to debug the kernel. However, WinDbg has the disadvantage of relatively inconvenient interface compared to other debuggers. WinDbg Preview has a significant improvement over the interface compared to WinDbg, and supports Time Traveling that performs reverse single-step execution.

Recently, Hao Shi and Jelena Mirkovic [16] proposed Apate, a framework for hiding the debugger from anti-debugging techniques. They have designed and implemented various bypassing techniques for anti-debugging in 32-bit Windows environments. Apate was implemented as a plug-in of 32bit-version of WinDbg. They claim that Apate outperforms other debugger-hiding schemes by a wide margin. In our experiment, even though Apate works well on simple packers (e.g., UPX, MPRESS, or ASPack), it fails on complex packers (e.g., VMProtect, Safengine, or Themida).

CPU emulator/virtualization: The CPU emulator is the tool that emulates and executes binary code. However, using this tool alone we cannot execute operating system-related code, so we need to build a virtual environment for correct execution.

QEMU [26] uses a CPU emulator to provide virtualization environments. This allows us to install an operating system on a virtual machine. Bochs [10] is an IA-32/x86-64 emulator while QEMU supports a variety of platforms, including IA-32 and ARM. Generally, the running speed is much slower than that of the real execution. If we use hardware-supported virtual environments, the execution speed on the virtual machine is very similar to that on the real execution. The tools supporting hardware virtualization include VMWare ESXi [27], Xen [28], and Microsoft Hyper-V.

Using virtualization tools, we can run the target code in a completely isolated environment. Since code execution in a virtualized environment is almost similar to that of a non-virtualized environment, anti-reversing techniques are automatically bypassed [29].

However, since these tools are not designed for analyzing the binary code, it is very inconvenient for code analysis. E.g., when extracting the instruction trace, not only the target process/thread information but also other processes/threads information or kernel information are included, which makes the analysis inconvenient. Further, it is difficult to combine the virtualization tool and the analysis tool such that only the desired part is analyzed. Since there are relatively a few virtualization tools exist, recently techniques for detecting their virtualization environments are actively being developed [6].

[30] presents HybridEmu, the dynamic analysis scheme for investigating the internal structure of malicious code in Microsoft Windows 32-bit environments. Similar to xUnpack64, HybridEmu can directly call or emulate various API functions in malware while emulating instructions using the 32-bit CPU simulator. However, it is designed only for 32-bit environments.

Dynamic Binary Instrumentation (DBI): DBI tools can run the analyst's code in arbitrary or specific parts of the target code at runtime. It operates in such a way that the target code and the analysis code are interleaved and executed. Pin [8] provides various APIs for analyzing the target code and is famous for correct execution and fastness. However, it is difficult to identify the cause when an error occurs because it is not open source.

Valgrind [31] and DynamoRIO [9] also provide a variety of APIs for analyzing the binary code and are open source. Valgrind only supports the Linux operating system, while DynamoRIO has a disadvantage in that for a large/complex program it often fails for analysis. Detours [32] hooks the major Win32 API functions for analysis. It is well known for efficiency and correctness. However, there is a disadvantage in that detailed analysis (e.g. at the instruction level analysis) is difficult.

These DBIs focus on fast and accurate code execution and flexible code instrumentation of target binaries. Since the original code is modified in the process of code instrumentation, the anti-reversing detection techniques for detecting

Resource and Handle	
WaitForSingleObject	DuplicateHandle
CloseHandle	GetModuleFileNameA
GetModuleFileNameW	
Process and Thread Enumeration	
Module32First	Process32First
Thread32First	Module32Next
Process32Next	Thread32Next
File	
WriteFile	GetCurrentDirectoryA
SetCurrentDirectoryA	GetMappedFileNameW
SHGetFolderPathW	
Service	
EnumServicesStatusExW	
GUI	
MessageBoxW	MessageBoxExA
GetWindow	FindWindowA
Memory	
VirtualAlloc	AllocateHeap
FreeHeap	RtlAllocateHeap
RtlFreeHeap	VirtualProtect
VirtualFree	LocalAlloc
LocalFree	HeapCreate
HeapDestroy	HeapFree
WriteProcessMemory	ZwQueryVirtualMemory
ZwProtectVirtualMemory	ZwMapViewOfSection
Process and Thread	
ZwQueryInformationProcess	FatalExit
exit	FinalExit
GetCurrentProcessId	GetCurrentProcess
OpenThread	GetExitCodeThread
GetCurrentThread	GetCurrentThreadId
ZwSetInformationThread	GetThreadContext
SetThreadContext	CreateThread
ZwQueryInformationThread	SuspendThread
ResumeThread	RtlGetCurrentPeb
GetProcessAffinityMask	SetThreadAffinityMask
Synchronization	
LocalLock	RtlEnterCriticalSection
Library	
LoadLibraryA	LoadLibraryExW
GetModuleHandleA	GetModuleHandleW
Time	
GetLocalTime	GetTickCount
QueryPerformanceCounter	GetSystemTimeAsFileTime
CRT	
__initterm	__initterm_e
puts	_CrtDbgBreak
_CrtDbgRepor	_CrtSetDbgBlockType
__dllonexit	GetStartupInfoW
GetStartupInfoA	__set_app_type
__lconv_init	GetCommandLineA
GetCommandLineW	__crtSetUnhandledExceptionFilter
Error and Exception	
SetLastError	SetUnhandledExceptionFilter
RtlAddVectoredExceptionHandler	RtlRemoveVectoredExceptionHandler
RtlAddFunctionTable	RaiseException
Debugging	
CheckRemoteDebuggerPresent	IsDebuggerPresent
ZwQuerySystemInformation	

this modification cannot be avoided. As DBI tools become popular for analyzing anti-reversing code, techniques for identifying individual DBIs have been developed [33], [34].

Unpacking tools: Unpacking tools have been developed for various packers. One example is the UPX unpacker [13]. Renovo [13] provides a general way to find the unpacking procedures. It finds written-and-execute behaviors which are common in the unpacking work. However, Renovo does not deal with various anti-reverse engineering techniques, which makes difficult to use in reality. VMAttack [35] focuses on automatic deobfuscation (i.e., intensive code-stripping) of code-virtualization techniques in VMProtect (version 2) whereas our work deals with automatic analysis of unpacking procedure of packers (including VMProtect 3.4). Pindemonium [6], one of the most famous open-sourced unpacking tools, relies on Pin for analyzing the packed program and for dumping the unpacked code. To the best of our knowledge, it cannot unpack recent version of sophisticated (commercial) packers, including VMProtect, Themida, and Safengine. Recently, UnThemida [36] was developed as a plug-in for the Pin tool to analyze and to unpack the structure of files packed with Themida 2.4.5. Unlike previous tools including UnThemida or UPX unpacker, x64Unpack is designed to cope with diverse packers: it can handle both finding general unpacking routines and evading various anti-reverse engineering techniques.

VII. CONCLUSION

In this paper, we proposed x64Unpack, a hybrid application emulator for 64-bit Windows environments. Using this tool, analysts can collect and analyze unpacking-related information through interception of API calls, memory I/O, etc. It can be used to analyze obfuscation tools, to detect and bypass anti-debugging techniques, and to restore packed files. To verify the effectiveness of the proposed method, experiments were conducted on widely used obfuscation tools: MPRESS 2.19, Themida 2.4.6, VMProtect 3.4 and UPX 3.95. For the obfuscated files that were packed with the default settings, x64Unpack unpacks them successfully. Based on this, we provide detailed analysis results on the obfuscated executable file by VMProtect 3.4. Additionally, we explain how to restore the unpacked version from the obfuscated binary code. Since x64Unpack currently cannot run 32-bit applications, the future work includes supporting 32-bit programs. Further, hooking API-functions or monitoring memory I/O can be implemented using filter drivers, which we leave as a future work.

APPENDIX A
NAMES OF MODIFIED API FUNCTIONS

The following list shows the names of the modified API functions in x64Unpack (grouped by functionalities), some of which are described in Section II-E.

REFERENCES

[1] Orleans Technology. *Themida: Advanced Windows Software Protection System*. Accessed: Jan. 19, 2020. [Online]. Available: https://www.oreans.com/themida.php

[2] VMSoft. *VMProtect Software: VMProtect Virtualizes Code*. Accessed: Jan. 19, 2020. [Online]. Available: http://vmpsoft.com/products/vmprotect/

[3] Safengine. *Safengine Protector*. Accessed: Jan. 19, 2020. [Online]. Available: http://www.safengine.com/en-us/

[4] S. Berinato. (Jun. 1, 2007). *The Rise of Anti-Forensics CSO from IDG*. Accessed: Jan. 19, 2020. [Online]. Available: https://www.csoonline.com/article/2122329/the-rise-of-antiforensics.html

[5] P. Ferrie. (2011). *The Ultimate Anti-Debugging Reference*. Accessed: Jan. 19, 2020. [Online]. Available: https://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf

[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 *Proc. IFIP Int. Conf. ICT Syst. Secur. Privacy Protection*, Ghent, Belgium, vol. 471, 2016, pp. 323–336.

[7] OllyDbg. *OllyDbg V1.10: 32-Bit Assembler Level Analyzing Debugger for Microsoft Windows*. Accessed: Jan. 19, 2020. [Online]. Available: http://www.ollydbg.de/

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Chicago, IL, USA, 2005, pp. 190–200.

[9] T. Garnett, "Dynamic optimization of IA-32 application under DynamoRIO," M.S. thesis, Dept. Elect. Eng. Comput. Sci., MIT, Cambridge, MA, USA, 2003.

[10] K. P. Lawton, "Bochs: A portable PC emulator for unix/X," *Linux J.*, vol. 1996, no. 29, p. 7, Sep. 1996.

[11] S. Choi, "API deobfuscator: Resolving obfuscated API functions in modern packers," presented at the BlackHat, Las Vegas, NV, USA, Aug. 2015.

[12] G. M. Kim, J. Park, Y.-H. Jang, and Y. Park, "Efficient automatic original entry point detection," *J. Inf. Sci. Eng.*, vol. 35, no. 4, pp. 887–902, Jul. 2019.

[13] Heaventools. *UPX Unpacker Plug-In: Automatic UPX Unpacking*. Accessed: Jan. 19, 2020. [Online]. Available: http://www.heaventools.com/PE_Explorer_plug-ins.htm

[14] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proc. WORM*, Alexandria, VA, USA, 2007, pp. 46–54.

[15] Qt Group. *Qt: Cross-Platform Software Development Framework for Native Embedded, Desktop and Mobile Applications*. Accessed: Jan. 19, 2020. [Online]. Available: https://www.qt.io/

[16] H. Shi and J. Mirkovic, "Hiding debuggers from malware with apate," in *Proc. Symp. Appl. Comput. (SAC)*, Marrakech, Morocco, 2017, pp. 1703–1710.

[17] x64dbg. *An Open-Source X64/X32 Debugger for Windows*. Accessed: Jan. 19, 2020. [Online]. Available: https://github.com/x64dbg/x64dbg

[18] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, Y. Wang, and Y. Xiang, "A3CM: Automatic capability annotation for Android malware," *IEEE Access*, vol. 7, pp. 147156–147168, 2019.

[19] Y.-S. Liu, Y.-K. Lai, Z.-H. Wang, and H.-B. Yan, "A new learning approach to malware classification using discriminative feature extraction," *IEEE Access*, vol. 7, pp. 13015–13023, 2019.

[20] S. Shen, H. Zhou, S. Feng, J. Liu, and Q. Cao, "SNIRD: Disclosing rules of malware spread in heterogeneous wireless sensor networks," *IEEE Access*, vol. 7, pp. 92881–92892, 2019.

[21] R. Kumar, X. Zhang, W. Wang, R. U. Khan, J. Kumar, and A. Sharif, "A multimodal malware detection technique for Android IoT devices using various features," *IEEE Access*, vol. 7, pp. 64411–64430, 2019.

[22] R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachadran, and S. Venkatraman, "Robust intelligent malware detection using deep learning," *IEEE Access*, vol. 7, pp. 46717–46738, 2019.

[23] Z. Fang, J. Wang, B. Li, S. Wu, Y. Zhou, and H. Huang, "Evading anti-malware engines with deep reinforcement learning," *IEEE Access*, vol. 7, pp. 48867–48879, 2019.

[24] S. Bardin, R. David, and J.-Y. Marion, "Backward-bounded DSE: Targeting infeasibility questions on obfuscated codes," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Jose, CA, USA, May 2017, pp. 633–651.

[25] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the semantics of obfuscated code," in *Proc. USENIX Secur. Symp.*, 2017, pp. 643–659.

[26] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf.*, Anaheim, CA, USA, 2005, pp. 41–46.

[27] D. Mishchenko, *VMware ESXi: Planning, Implementation, Security, Course Technology*, 1st ed. Boston, MA, USA: Course Technology, 2011, pp. 1–23.

[28] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. SOSP*, New York, NY, USA, 2003, pp. 164–177.

[29] C. Miller, D. Glendowne, H. Cook, D. Thomas, C. Lanclos, and P. Pape, "Insights gained from constructing a large-scale dynamic analysis platform," in *Proc. DFRWS*, Austin, TX, USA, vol. 22, 2017, pp. S48–S56.

[30] S. Choi, T. Chang, S.-W. Yoon, and Y. Park, "Hybrid emulation for bypassing anti-reversing techniques and analyzing malware," *J. Supercomput.*, Apr. 2020. [Online]. Available: https://link.springer.com/journal/11227/onlineFirst/page/5, doi: 10.1007/s11227-020-03270-6.

[31] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. PLDI*, San Diego, CA, USA, 2007, pp. 89–100.

[32] G. Hunt and D. Brubacher, "Detours: Binary interception of Win32 functions," in *Proc. 3rd USENIX Windows NT Symp.*, Seattle, WA, USA, 1999, p. 14.

[33] J. Kirsch, Z. Zhechev, B. Bierbaumer, T. Kittel, "PwIN–Pwning Intel piN: Why DBI is unsuitable for security applications," in *Proc. ESORICS*, in Lecture Notes in Computer Science, vol. 11098, 2018, pp. 363–392.

[34] M. Polino, A. Continella, S. Mariani, S. D'Alessio, L. Fontana, F. Gritti, and S. Zanero, "Measuring and defeating anti-instrumentation-equipped malware," in *Proc. DIMVA*, Dagstuhl, Germany, vol. 10327, 2017, pp. 73–96.

[35] A. Kalysch, J. Götzfried, and T. Müller, "VMAttack: Deobfuscating virtualization-based packed binaries," in *Proc. ARES*, 2017, pp. 1–10.

[36] 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, Jul. 2018.

SEOKWOO CHOI received the B.S., M.S., and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 1998, 2000, and 2009, respectively. He is currently a Senior Researcher with The Affiliated Institute of ETRI, South Korea. His research interests include malware analysis and code deobfuscation.

TAEJOO CHANG (Senior Member, IEEE) received the B.S.E. degree in electrical engineering from Ulsan University, in 1982, and the M.S.E. and Ph.D. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), in 1990 and 1998, respectively. In February 2000, he joined The Affiliated Institute of ETRI, South Korea. His current research interests include digital forensics and binary analysis.

CHANGHYUN KIM is currently pursuing the bachelor's degree in computer science with Hanyang University, Seoul, South Korea. His main research interests include network security analysis, software security, and middleware security.

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