Renovo: A Hidden Code Extractor for Packed Executables

Min Gyung Kang, Pongsin Poosankam, and Heng Yin [*]
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213
{mgkang@, ppoosank@, hyin@ece.}cmu.edu

ABSTRACT

As reverse engineering becomes a prevalent technique to analyze malware, malware writers leverage various anti-reverse engineering techniques to hide their code. One technique commonly used is code packing as packed executables hinder code analysis. While this problem has been previously researched, the existing solutions are either unable to handle novel samples, or vulnerable to various evasion techniques. In this paper, we propose a fully dynamic approach that captures an intrinsic nature of hidden code execution that the original code should be present in memory and executed at some point at run-time. Thus, this approach monitors program execution and memory writes at run-time, determines if the code under execution is newly generated, and then extracts the hidden code of the executable. To demonstrate its effectiveness, we implement a system, Renovo, and evaluate it with a large number of real-world malware samples. The experiments show that Renovo is accurate compared to previous work, yet practical in terms of performance.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Security

Keywords

Reverse Engineering, Dynamic Analysis, Code Obfuscation, Malware Analysis

1. INTRODUCTION

Reverse engineering is one of the main techniques used for malware analysis. To make the analysis more difficult, malware writers usually have their programs heavy-armored with various anti-reverse engineering techniques. Such techniques include binary and source code obfuscation [13, 22], control-flow obfuscation [20], instruction virtualization [10], and binary code packing [27]. This paper focuses on identifying and extracting the hidden code generated using binary code packing, one of the most common anti-reverse engineering methods. Code packing transforms a program into a packed program by compressing or encrypting the original code and data into packed data and associating it with a *restoration routine*. A restoration routine is a piece of code for recovering the original code and data as well as setting an execution context to the original code when the packed program is executed. This technique is available as commercial products [12, 14, 26, 28, 29] and open-source tools. According to the anti-virus (AV) program test results of AV-Test GmbH [15], the detection rates of 8 major AV programs varied from 10% to 80% when known malware binaries have been packed.

Various tools have been developed to identify and extract the hidden code in packed executables. Commonly known tools such as PEiD [6] employ a simple pattern matching approach. These tools check an executable with a signature database to determine what kind of packing tool is used to create the executable. Then, using a priori knowledge about the packing tool, it is possible to extract the hidden binary from the executable [9]. Although this approach is usually fast and accurate for known packing tools, it is unable to detect novel and modified packing techniques. For example, a variant of the Bagle worm employed its own compression engine which is not known to the public [19]. In fact, by modifying the open source anti-reverse engineering tools like YodaProtector [11], it is easy for malware writers to implement new anti-reverse engineering algorithms and tricks.

Dynamic analysis is a promising solution to the problem of hidden code extraction because it does not depend on signatures. Regardless of what packing technique might be applied to the original program, the original code or its equivalent must eventually be present in memory and get executed at some point at run-time. By taking advantage of this intrinsic nature of packed executables, one could potentially extract the hidden binary code or its equivalent as a raw memory dump. However, it is not clear which regions in the memory contain the hidden binary and when is the right time to dump such regions, i.e., when the execution context

[*] Heng Yin is also affiliated with Department of Computer Science at The College of William and Mary, Williamsburg, Virginia 23187

WORM'07, November 2, 2007, Alexandria, Virginia, USA.
Copyright 2007 ACM 978-1-59593-886-2/07/0011 ...$5.00.

jumps to the hidden original code. In addition to the hidden code, other information such as the *original entry point* (OEP) is also crucial for further analyses of the malware. The original entry point is the first hidden instruction being executed when the program control flow is transferred from the restoration routine to the hidden code. Several approaches, such as Universal PE Unpacker [17] and PolyUnpack [27], have shown that extracting packed binaries and finding the OEP using dynamic analysis is feasible. These approaches either rely on some heuristics or require disassembling the packed program. However, heuristics about packed code may not be reliable in all cases and can be easily evaded. In addition, correctly disassembling a binary program itself is challenging and error-prone, as demonstrated in [25]. To overcome the disassembly challenge required for packed code extraction, a tool like PolyUnpack needs to perform a series of static and dynamic analysis which leads to performance overhead.

In this paper, we present a fully dynamic approach for extracting the original hidden code and additional information useful for further analysis of the extracted malware binary. We capture an intrinsic nature of packed programs that is independent of the packing techniques applied on the programs. That is, the original code will be dynamically generated and then executed.

The contributions of this paper are as follows:

Propose a fully dynamic approach for extracting the original hidden code of packed executables: A considerable effort has been made to come up with practical solutions for identifying compressed executables and restoring their original hidden code and data. Previous work relies on either heuristics of known packing tools or the accuracy of the disassembler. However, as we see in the Bagle case, malware writers can apply modified binary compression techniques to evade heuristic-based tools [19]. In addition, disassembling binary executables as being done in [25] and [27] is an arduous task. In this paper, we present a binary extraction technique which is fully dynamic and thus does not depend on the program disassembly or the known signatures of packing techniques. We also show that our proposed technique can extract the original hidden code and data, and find the entry point of the original program that enables efficient code analysis.

Provide additional information for the next-step analysis: In addition to extracting the hidden code, our proposed method can provide additional information on the packed binaries:

- Identify the exact regions of memory where the hidden code and data reside: by tracking the newly-written memory areas of the program, we can distinguish newly-generated code and data at run-time from the packed binary, and thus obtain the exact regions of them.

- Extract information on multiple *hidden layers* : even in the case that the original program is hidden through multiple rounds of compression and encryption, we can keep track of intermediate code and data for each round. This provides valuable information on what kind of packing methods are in use and what kind of data is generated at each round.

Implement and evaluate *Renovo,* an automated framework for extracting hidden code: Applying our proposed technique, we build a framework for automatically examining executable binaries and extracting their original hidden code. Since this is a fully automated process, it could be used by anti-virus programs and on-line malware binary analysis services [1,3]. We also present the evaluation results of Renovo, demonstrating that it is both highly effective and efficient compared to previous approaches.

2. RELATED WORK

Extracting and re-building the original program from a compressed or encrypted binary has been one of the major challenges for software reverse engineers and the security community. For known packing techniques, there exist corresponding *unpackers* [9]. However, given an arbitrary packed executable, which unpacker to use is still a problem. PEiD [6] is a tool for identifying compressed Windows PE binaries. Using the database of the signatures for known compression and encryption techniques, it identifies the packing method employed and, thus, suggests which unpacker can be applied. However, despite their ability to perfectly restore the original program, executables packed with unknown or modified methods are beyond the scope of this approach.

Universal PE Unpacker [17] and OllyBonE [4] are attempts to develop a comprehensive solution to this problem. As plug-in modules for IDA Pro [18] and Olly Debugger [5], both tools identify packed executables and their original entry points by using several heuristics. For example, Universal PE Unpacker assumes that `GetProcAddress` is always called to setup the import table after the original program is unpacked and before the program counter reaches the OEP. Also, it is not intended to be an automated unpacking tool because it must be given *a priori* knowledge about the the possible range of the OEP. OllyBonE sets the "Break-on-Execution" flag on the reserved memory sections used to accommodate unpacked code and data. When the CPU accesses these execution-protected pages, OllyBonE detects it and enables the extraction of the hidden code executed on OllyDbg. Although the OSes do not always enforce the assumptions where these heuristics work, in most of the cases, it produces correct results quickly. However, as shown in [27] and in our results, some malware can evade this heuristic-based approach.

PolyUnpack [27] is a general approach for extracting the original hidden code without any heuristic assumptions. PolyUnpack takes advantage of the intrinsic nature of packed executables where the hidden code is generated and executed at run-time, and thus it is not present in the code section of the packed executable. As a pre-analysis step, PolyUnpack disassembles the packed executable to partition it into the code and data sections, Then it executes the binary instruction by instruction, checking whether the instruction sequence from the current point is in the code section identified in the pre-analysis step. The authors have implemented this approach and have shown that it can successfully identify and extract the hidden code in malware samples in the wild. However, in terms of performance, disassembling a program and single-step executing a binary significantly increase the computational complexity of its analysis.

Christodorescu et al. proposed several normalization techniques that transform obfuscated malware into a normalized form to help malware analysis [16]. Their unpacking normalization is similar to our approach. Its basic idea is to detect the execution of newly-generated code by monitoring memory writes after the program starts. We independently propose and implement our approach, and conduct more extensive experiments using various packed malware samples.

There has also been a commercial effort to enhance the detection rate against packed malware. McAfee applies the Generic Decryption Engine (GDE) technique to its antivirus products [24]. GDE analyzes the decryption (decompression) algorithm in the malware code and uses this algorithm to extract the hidden code before applying its detection engine. Ewido Networks employs an emulation-based technique to extract the hidden code of malware [19]. The details of how these mechanisms work are not present in [19, 24], but some malware in the wild are still shown to be able to evade these commercial virus scanners [27].

3. PROBLEM STATEMENT AND OUR APPROACH

In this paper, we devise a mechanism to automatically identify packed executables and extract their original hidden code and data. Specifically, given an arbitrary executable binary, we want to verify whether it executes the original program code that is generated from the packed data in the binary. In addition, when observing this behavior, we extract the whole newly-generated code and data with its entry point address (OEP).

Packed Executable: Figure 1 shows how a typical packed executable work. After the packed executable starts, its attached restoration routine performs transformation procedures on the packed data, and then recovers the original code and data. When the restoration completes, the restoration routine prepares the execution context for the original program code to execute, which includes initializing the CPU registers and assigning the program counter to the entry point of the newly-generated code region. Hereafter in this paper, we refer to this restoration step as *hidden layer*. Note that a packed executable may have multiple hidden layers, making it even more difficult to analyze. As will be shown in Section 5, about 80% of the malware samples used in our experiments have more than one hidden layer.

Scope: Besides the packing techniques, there are other kinds of anti-reversing techniques. Linn et al. [22] proposed obfuscation techniques to thwart the disassembly procedure. In response, Kruegel et al. [21] presented a defense against these techniques. Recently, sophisticated software protection tools, like Themida [10], convert the original x86 instructions into virtual instructions in its own randomized instruction set, and then interpret these virtual instructions at run-time. Dealing with these anti-reversing techniques are beyond the scope of this paper.

Our Approach: No matter what packing methods or how many hidden layers are applied, the original program code and data should eventually be present in memory to be executed, and also the instruction pointer should jump to the OEP of the restored program code which has been written in memory at run-time. Taking advantage of this inevitable nature of packed executables, we propose a technique to dynamically extract the hidden original code and the OEP from the packed executable by examining whether the current instruction has been generated at run-time, after the program binary was loaded. For this purpose, we monitor if the instruction pointer jumps to the memory region which has been written after the program start-up. When a program is loaded in memory, we generate a memory map and initialize the map as *clean.* Whenever the program performs a memory write instruction, *e.g.,* `mov %eax, [%edi]` and `push %eax`, we mark the corresponding destination memory region as *dirty*, which means it is newly generated. Meanwhile, when the instruction pointer jumps to one of these newly-generated regions, we determine that there is a hidden layer hiding the original program code, and identify the newly-generated memory regions to contain the hidden code and data, and the address pointed by the instruction pointer as the original entry point (OEP). To handle the possible hidden layers that may appear later on, we initialize the memory map as clean again, after storing all the information extracted from the current hidden layer. Then, we repeat the same procedure until the time-out.

Advantages: The advantages of this approach are threefold: First, we assume nothing about the packing methods except the inevitable fact that the original hidden code should eventually be written and executed at run-time. Therefore, our approach is able to handle any sort of packing techniques applied to the binaries. Second, the approach can determine the exact memory regions accommodating the code or data generated at run-time. Since we keep information about memory writes at byte-level, it is possible to efficiently extract the newly-generated code and data. Lastly, Our approach does not rely on any information on the code and data sections of the binary. Unlike previous approaches [17, 27] which employ disassembly techniques, our approach depends solely on the origin of the instructions being executed.

4. SYSTEM DESIGN AND IMPLEMENTATION

To demonstrate the effectiveness of our approach, we design and implement a system, Renovo, to automatically identify packed executables and extract their hidden code. Figure 2 depicts an overview of Renovo. Renovo is built on top of TEMU [8], which is a dynamic analysis component of the BitBlaze [2] binary analysis platform. When analyzing an executable, we run it in an *emulated environment.* The *execution monitor* observes its execution from the outside (i.e., the host system), consults the *shadow memory* of that process, and determines if any hidden code is currently executed. If so, it extracts the hidden code and obtains the OEP which is useful for further code analysis. We will present these components in turn, and discuss the limitations of the current implementation of Renovo.

4.1 Emulated Environment

When analyzing an executable, we run it in an emulated environment. This emulated environment facilitates instrumenting CPU instructions in a fine-grained manner. In particular, we need to instrument the instructions that perform memory writes. We also need to instrument the instructions that change the execution flow (e.g., `jmp` and `call` in x86), in order to identify the original entry point.

Moreover, this emulated environment provides isolation between the extraction engine and the malicious programs under analysis. Therefore, it is difficult for malicious code to interfere with the extraction engine and affect the analysis results.

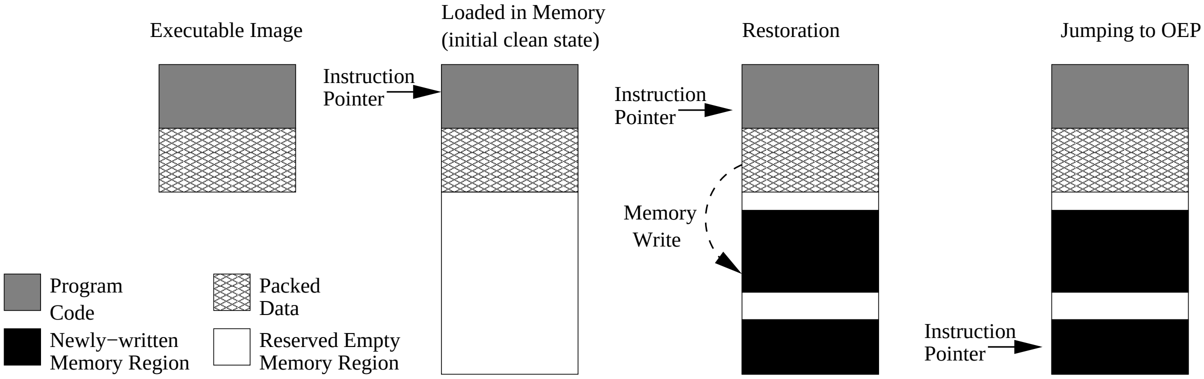

Figure 1: How a packed executable works.

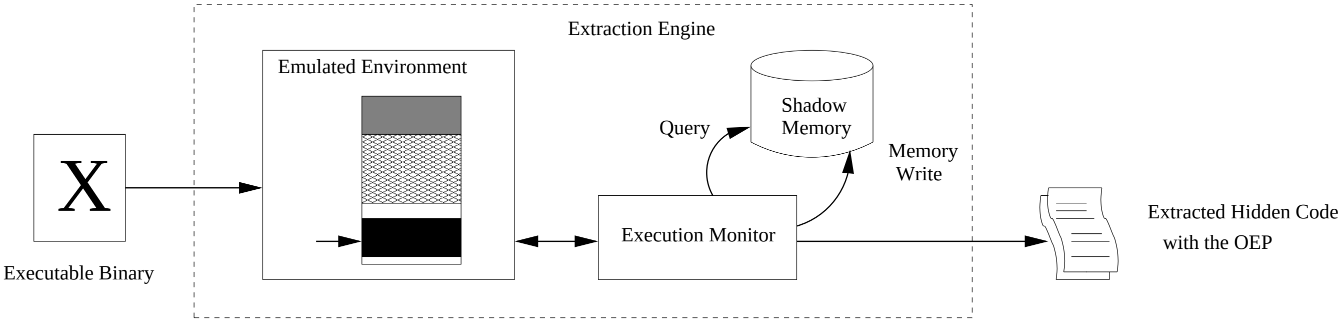

Figure 2: Renovo Overview

4.2 Shadow Memory

We maintain a shadow memory for the memory space of the observed process. Considering that x86 instructions have variable sizes, and that data and instructions can be interleaved in x86 executables, we associate each byte of the memory space with a flag. Since this flag has only two states: clean and dirty, it can be represented as 1 bit. 0 stands for "*clean*", and 1 for "*dirty*". To minimize the memory consumption, we employ a page-table-like structure to store the shadow memory.

4.3 Extraction Engine

The extraction engine observes the program execution from the host system. When the program is about to execute, we set the whole memory space as clean. In this case, the page table of the shadow memory is empty. During program execution, the extraction engine instruments memory writes within the observed process, and updates the shadow memory. Meanwhile, it queries the shadow memory, and checks if any byte of the memory region that the current instruction occupies is dirty. If so, it can determine the instruction has been newly generated. A special case is that the program may load a dynamic linked library after it has been executed for a while. This library may occupy a memory region that has previously been dirty bytes. In this case, we set this region to clean states.

Usually a whole-system emulator only provides hardware-level view of the emulated system. For our analysis, we need to know which process is running and which process should be observed. We make use of a mechanism provided by TEMU to reason about OS-level semantics. Technically, a kernel module is inserted into the emulated system to obtain necessary process information. This kernel module registers several call-back functions. Thus the module will be notified whenever a process is created or destroyed, or a module (DLL or Executable) is loaded into the process. Hence, when a new process is created, the kernel module records the process name and the value of CR3. In an x86 system, the CR3 register stores the physical address of the page table for the current process, and thus it is unique for each process. Then, we only need to perform instrumentation when the current CR3 corresponds to the monitored process. If we identify a module is loaded after the program starts to execute, we know which memory region this module occupies, and clean the states within the region.

Instrumenting memory writes is straightforward. TEMU has a centralized module to instrument memory references. Thus, we simply mark the destination (in virtual address) as dirty if the current memory reference is a write and comes from the observed process.

When checking newly generated instructions, we do not have to check every instruction. To optimize the performance, we check every basic block in the observed process. A basic block is a sequence of instructions with only one entry and one exit. Thus a basic block is a contiguous code region. At the block entry, we record its address. Then at the block exit, we check if there is any dirty memory locations within the region covering this block. If so, this block entry is the OEP, and we dump the pages containing dirty memory bytes.

In order to extract hidden code from packed executables with multiple hidden layers, we clean the dirty states in the

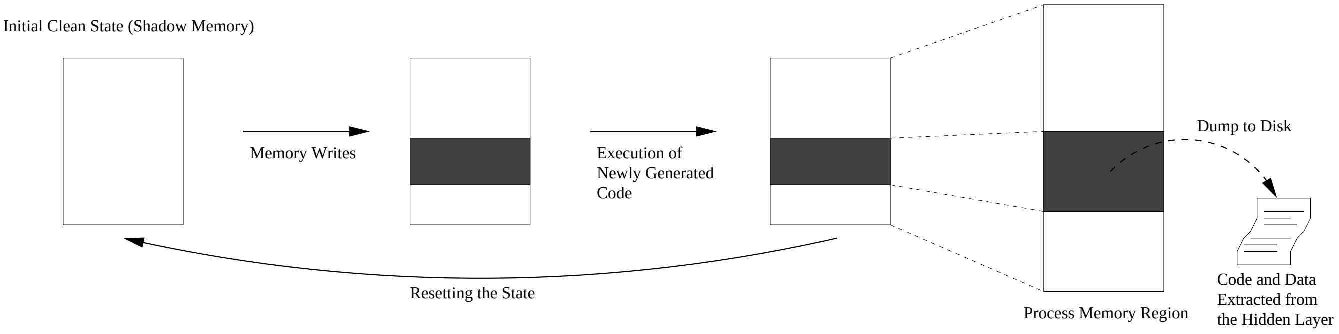
Initial Clean State (Shadow Memory)
Memory Writes
Execution of Newly Generated Code
Dump to Disk
Resetting the State
Process Memory Region
Code and Data Extracted from the Hidden Layer

Figure 3: Extracting hidden layers by using the shadow memory

shadow memory, and then repeat the extraction procedure as shown in Figure 3. Note that determining whether a program has hidden code or not is an *undecidable* problem [27]. Thus, we introduce a configurable time-out parameter into the system. If we do not observe any hidden code being executed within this time-out, we terminate the extraction procedure. In the experiments, we set this parameter to be 4 minutes.

4.4 Discussion

Here we discuss potential evasion techniques that malware writers can employ to thwart Renovo.

Circumventing the emulated environment. As we run the binaries in an emulated environment, an obvious evasion is to detect the presence of this emulated environment and stay inactive. For example, the malicious code may measure elapsed time for certain instructions, because emulating these instructions incurs high overhead [30], or check the results of certain instructions (e.g., `sidt`), because the results they generate are different under real and emulated environments (e.g., *redpill* test [7]). While there may be no comprehensive solution to this problem, we may apply defenses to specific detection techniques. For example, in the current implementation, we instrument several instructions like `sidt`. If the binary under analysis executes one of these instructions, we may return a fake result to make it believe that it is running under the real environment. In our experiments, we have successfully deceived the redpill [7]. We may also instrument `rdstc` to delude detection techniques using the elapsed time for certain instructions.

Exploiting the time-out. Since determining whether an executable contains hidden code or not is an undecidable problem as shown in [27], we employ a time-out mechanism. The malicious programs may exploit this feature to evade detection by staying inactive for a sufficiently long period. A better metric to determine when to terminate the extraction procedure is to count how many different instructions from the binary have been executed. Thus, the malicious programs cannot evade detection by simply sleeping or busy looping. We will explore this metric in the future release of Renovo.

5. EVALUATION

In this section, we describe two experiments and present the evaluation results, demonstrating that Renovo is an accurate and practical solution for extracting the original hidden code of packed executables.

5.1 Extracting from Synthetic Samples

To verify that Renovo generates accurate results, we have tested Renovo and two other extraction techniques, Universal PE Unpacker [17] and PolyUnpack [27], against the synthetic sample programs generated by using 14 different packing tools. These tools apply different packing techniques as well as encryption, code obfuscation, debugger detection, and instruction virtualization to thwart reverse engineering.

Samples: We use Microsoft notepad as an original binary to generate synthetic packed program samples. For all tools but Themida [10], the samples are created using the tools' default configuration. In the case of Themida, we generated two samples with slightly different configurations: one with instruction virtualization ("VM option") and one without it. Other than that, both options still use the same compression, encryption, and other techniques to protect the program from reverse engineering. We tested and ensured that none of these synthetic samples contains the binary string found in the `.text` section of the original notepad program. With the knowledge that these packing tools usually restore and execute the original binary instructions at run-time, we could verify the correctness of our extraction technique by comparing the extracted hidden code regions with the `.text` section of the original binary.

Renovo: As shown in Table 1, Renovo fully extracted the original binaries processed by all but 3 packing tools, which are Armadillo, Obsidium, and Themida(w/ VM). But in the first two cases, the samples terminated before reaching the original program code, likely because the executables are not compatible with the Renovo's emulation engine. Nevertheless, Renovo still identified these two samples as packed executables because it successfully extracted hidden code and data from several initial hidden layers, which seem to be its restoration routines. In the case of a sample generated using Themida(w/ VM), Renovo extracted some hidden regions which do not match the original notepad binary. We believe this is the VM virtualization code equivalent to the original notepad instructions since we successfully extracted those from a sample generated using Themida(w/o VM).

Universal PE Unpacker (UUnP): Although UUnP requires *a priori* knowledge about the possible range of the OEP, it can run automatically without such input from a user. By default, it assumes that the OEP locates in the first program segmentation as identified by IDAPro and uses this contiguous memory segmentation as the possible range of the OEP. We ran UUnP using this default heuristic and

Tool	Size	Renovo		UUnP		PolyUnpack	
	(KB)	result	time	result	time	result	time
			(sec)		(sec)		(sec)
None	68	no	N/A	no	N/A	no	N/A
Armadillo	564	error	44	error	1	part	1617
ASPack	53	yes	35	yes	3	part	181
ASProtect	153	yes	48	error	6	yes	62
FSG	46	yes	38	yes	3	yes	92
MEW	44	yes	36	yes	139	yes	739
MoleBox	108	yes	47	error	242	no	757
Morphine	72	yes	36	yes	1	yes	174
Obsidium	143	error	61	error	1	no	457
PECompact	49	yes	37	error	2	no	39
Themida(w/ VM)	1342	part	60	no	9	timeout	1800
Themida(w/o VM)	1067	yes	70	error	10	timeout	1800
UPX	47	yes	35	yes	3	yes	94
UPXS	47	yes	37	yes	4	yes	92
WinUPack	44	yes	38	error	12	part	33
YodaProtector	64	yes	36	error	1	part	62

Remark:

no A tool identified a binary as not being packed.

yes A tool extracted the whole original notepad binary.

part A tool identified an incorrect entry point or could only extract parts of the original binary.

timeout A tool did not terminate within the time-out period of 30 minutes.

error A tool encountered errors or terminated prematurely.

Table 1: Extracting the Hidden Code in Synthetic Samples

	Renovo	UUnP	PolyUnpack
Extracted results	366	186	171
IRC pattern found	363	176	86
Avg. time (sec.)	40.9	15.7	365.8

Table 2: The comparison of Renovo and previous work on malware samples.

found UUnP successfully extract the original notepad code from 6 out of 15 samples (Table 1). It failed on the sample generated by Themida(w/ VM) as the executable detected the presence of IDA's debugger. For the rest of the samples, UUnP encountered the exception handler routine and was unable to proceed to later execution steps. Nevertheless, note that UUnP is very efficient as it can extract most hidden code in less than 10 seconds.

PolyUnpack: We obtained the analysis results of PolyUnpack [27] by submitting samples to the Malfease website [23] of which PolyUnpack operates as its sub-module. We also asked the PolyUnpack authors to run our samples against a version of PolyUnpack that handles some forms of structured exception handling in addition to the functionalities presented on the Malfease website. PolyUnpack identified 10 samples to be packed and extracted the full original notepad code from 6 of them. It partially extracted packed code in the other 4 samples as some of the original binary is still hidden. PolyUnpack extracted only part of these samples because it can not handle multiple hidden layers without reprocessing the partially unpacked executable. PolyUnpack reached the 30-minute time-out in the samples generated using Themida. For the rest of the samples, PolyUnpack determined them to be not packed.

5.2 Extracting from Malware Samples

In this experiment, we test Renovo with the real malware samples which are protected by known and unknown packing techniques. We also used Universal PE Unpacker (UUnP) and PolyUnpack for comparison analysis like in the previous experiment.

Samples: To select the most-likely packed executables, we briefly examined the malware samples provided by Korea Information Security Agency (KISA) using PEiD [6]. From these samples, we collected 374 malware samples which are identified either to be packed by known tools like PECompact and UPX, or to contain *overlay* sections in their PE headers. (The samples with the overlay sections are likely to be packed executables.) According to the Norton Anti-Virus scan results, 7 of these samples are downloaders, and the rest are bot programs.

Overall: As shown in Table 2, Renovo identified most of the samples to be packed executables; only 8 out of total 374 samples were identified as normal executables. However, these 8 samples seem to have crashed or terminated before reaching the original hidden code. In comparison, both UUnP and PolyUnpack identified only about half of the samples to be packed executables. Like in the previous experiment, we also encountered exception handler problem when running UUnP on some of the samples. The average time for hidden code extraction is 40.9 seconds for Renovo, 15.7 seconds for UUnP, and 365.8 seconds for PolyUnpack. Considering that the system boot time of Renovo is about 30 seconds, the sheer code extraction time of Renovo is approximately 10 seconds which is less than that of UUnP. This is also a promising result when compared to the performance of Norton Anti-Virus. For the same set of malware samples, Norton Anti-Virus took 17 seconds per sample in average.

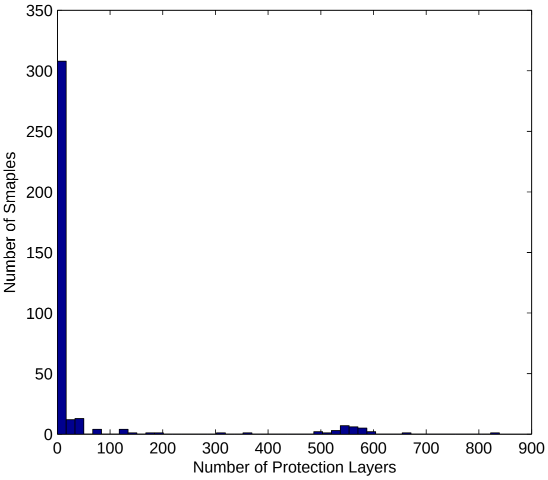

Figure 4: Hidden Layers in Malware Samples

Unlike the evaluation using the synthetic samples where we have the original program binaries, it is difficult to verify the correctness of extracted code and data. Therefore, we examined extracted code and data to see if they contain any of the IRC commands that common bot programs use to communicate with control servers. Considering the fact that most of the samples (367 out of 374) are bot programs, the extracted code and data are likely to contain some of these IRC commands which are not present in the packed executables. As we see in the second row of Table 2, most of the extracted code and data extracted by Renovo contain these IRC command strings which have not been found in the packed malware samples.

Multiple Hidden Layers As described in Section 3, Renovo can handle multiple hidden layers and thus, can extract the code and data at each hidden layer. Figure 4 shows the number of hidden layers found by Renovo and the number of corresponding samples. While most of the malware samples apply less than 20 hidden layers, some of the samples are found to use more than 500 hidden layers. Most of these highly-layered samples are applying unknown packing techniques which are not in the PEiD signature list. We conjecture that they might be a new type of packing technique which generates and executes only some parts of the original code on the fly to protect itself from dynamic analysis techniques at run-time. We leave this for future research.

5.3 Performance Overhead

We measured the performance overhead of Renovo by running a sample program on both Renovo and normal environment. The sample program is a small test binary which outputs simple text messages and it was packed using the UPX packing tool. We found that the current version of Renovo shows a performance slowdown of 8 times on average compared to the normal execution environment. Considering that Renovo is aiming to provide hidden code extraction environment for malware analysis which usually takes several hours to days, this degree of slowdown in initial execution time is tolerable.

6. CONCLUSION

To thwart reverse engineering, malware writers often try to hide their original programs by transforming them into packed executables. In this paper, we propose a dynamic approach to extract the hidden code and data from these packed executable, and the contributions are three-fold:

First, we propose a fully dynamic method which monitors currently-executed instructions and memory writes at run-time. This approach maintains a shadow memory of the memory space of the analyzed program, observes the program execution, and determines if newly generated instructions are executed. Then it extracts the generated code and data. Assuming nothing about the binary compression and encryption techniques, we provide a means to extract the hidden code and information, which is robust against anti-reverse-engineering techniques.

Second, our approach provides additional information useful for further code analysis. Since it monitors the run-time memory writes at byte-level, we can extract the exact memory regions with newly-generated code and data. Moreover, even in the case that multiple hidden layers are applied to the binary, we can keep track of the restoration routines and extract information at each layer.

Finally, to demonstrate its effectiveness, we implement a system, Renovo. By evaluating it with synthetic samples and over 370 real-world malware samples, our experiments show that Renovo provides more accurate results than all previous approaches, and incurs acceptable performance overhead.

7. ACKNOWLEDGMENTS

We would like to thank Dawn Song for her invaluable insights and Juan Caballero, Sihyung Lee, Paul Royal, and anonymous reviewers for their insightful feedback. We also would like to thank Korea Information Security Agency (KISA) for kindly providing malware samples for our experiments. Support for this material was provided by the National Science Foundation under Grants No. 0433540 and 0448452. Partial support was also provided by the U.S. Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316 and under Grant DAAD19-02-1-0389 through CyLab at Carnegie Mellon. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of ARO, CMU, the U.S. Government or any of its agencies. Min Gyung Kang performed this research under support of Samsung Scholarship.

8. REFERENCES

[1] Anubis. `http://analysis.seclab.tuwien.ac.at`.

[2] BitBlaze Binary Analysis Platform. `http://bitblaze.cs.berkeley.edu/`.

[3] Norman SandBox Information Center. `http://www.norman.com`.

[4] OllyBonE. `http://www.joestewart.org/ollybone/`.

[5] OllyDbg. `http://www.ollydbg.de/`.

[6] PEiD. `http://www.secretashell.com/codomain/peid/`.

[7] Red Pill. `http://invisiblethings.org/papers/redpill.html`.

[8] TEMU: The BitBlaze Dynamic Analysis Component. http://bitblaze.cs.berkeley.edu/temu.html.

[9] The Unpacker Archive. http://www.woodmann.com/crackz/Tools/Unpckarc.zip.

[10] Themida. http://www.oreans.com/.

[11] Yoda Protector. http://sourceforge.net/projects/yodap/.

[12] ASPack Software. ASPack and ASProtect. http://www.aspack.com/.

[13] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[14] Bitsum Technologies. PECompact2. http://www.bitsum.com/pec2.asp.

[15] T. Brosch and M. Morgenstern. Runtime packers: The hidden problem? https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.p%df, 2006.

[16] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, Nov. 2005.

[17] Data Rescue. Universal PE Unpacker plug-in. http://www.datarescue.com/idabase/unpack_pe.

[18] DataRescue SA. IDA Pro disassembler: Multi-processor, Windows hosted disassembler and debugger. http://www.datarescue.com/idabase/.

[19] T. Graf. Generic unpacking: How to handle modified or unknown PE compression engines. http://www.virusbtn.com/pdf/conference_slides/2005/Graf.pdf, 2005.

[20] Y. L. Huang, F. S. Ho, H. Y. Tsai, and H. M. Kao. A control flow obfuscation method to discourage malicious tampering of software codes. In *ASIACCS '06: Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 362–362, New York, NY, USA, 2006. ACM Press.

[21] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[22] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, New York, NY, USA, 2003. ACM Press.

[23] Project Malfease. http://malfease.oarci.net/.

[24] McAfee. Advanced virus detection scan engine and DATs. http://www.mcafee.com/us/local_content/white_papers/wp_scan_engine.pdf.

[25] S. Nanda, W. Li, L. Lam, and T. Chiueh. BIRD: Binary interpretation using runtime disassembly. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 358–370, Washington, DC, USA, 2006. IEEE Computer Society.

[26] Obsidium Software. Obsidium. http://www.obsidium.de/show.php?home.

[27] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.

[28] Silicon Realms Toolworks. Armadillo. http://siliconrealms.com/index.shtml.

[29] Teggo. MoleBox Pro. http://www.molebox.com/download.shtml.

[30] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, fidelity and containment in the potemkin virtual honeyfarm. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.