Mal-Xtract: Hidden Code Extraction using Memory Analysis

Charles Lim¹, Yohanes Syailendra Kotualubun², Suryadi³, Kalamullah Ramli⁴

¹,²Department of Information Technology, Swiss German University, Edutown BSD City, Tangerang, 15339 Indonesia
³Department of Mathematics, Universitas Indonesia, Kampus UI, Depok 16424, Indonesia
⁴Department of Electrical Engineering, Universitas Indonesia, Kampus UI, Depok 16424, Indonesia

E-mail: ¹charles.lim [at] sgu.ac.id, ²yohannes.kotualubun [at] student.sgu.ac.id, ³yadi.mt [at] sci.ui.ac.id, ⁴kalamullah.ramli [at] ui.ac.id

Abstract. Software packer has been used effectively to hide the original code inside a binary executable, making it more difficult for existing signature based anti malware software to detect malicious code inside the executable. A new method of written and rewritten memory section is introduced to detect the exact end time of unpacking routine and extract original code from packed binary executable using Memory Analysis running in an software emulated environment. Our experiment results show that at least 97% of the original code from the various binary executable packed with different software packers could be extracted. The proposed method has also been successfully extracted hidden code from recent malware family samples.

1. Introduction

Static Analysis through reverse engineering is one of the main techniques to analyze assembly codes inside a binary executable, especially for malicious software (malware). To prevent this, malware authors commonly use various obfuscation techniques to protect and hide their original malicious codes from being detected. With the rate of new malware now is more than 200 malware every second [1], malware analyst faces even more challenges to be able to analyze the huge volume of malware. Nevertheless, the new malware often is a “recycle” version of existing version, added with various obfuscation techniques to create a new signature to avoid from being detected. Obfuscation techniques include source code obfuscation [2, 3], Control-flow obfuscation [4], Instruction Emulation [5], and binary code packing [6]. Recent study [7] indicated up to 80% of detected malware is packed. Ugarte [8] found that common packers will run their unpacking routine and reveal their original body at some time in physical memory. First, packers write their original body in some memory sections and jump to their Original Entry Point, which contain the newly written memory section. Based on this characteristic, we define some pre-defined memory section size that each packer will write into it and use these pre-defined values as the threshold for memory section. With the increased use of software packer in malware, malware detection becomes less effective [9]. We believe that our method can help to increase the malware detection rate by unpacking the encryption and obfuscation from the packed malware.
In this experiment, we propose a method to detect the end of unpacking routine from common packer that runs inside QEMU [10]. We present a method called Mal-Xtract, a memory access analysis technique that determines the threshold of written section size to be used in extracting the original hidden-body by dumping the whole physical memory and extract its unpacked code. The contributions of this paper are as follows:

(i) We develop a memory analysis technique to determine the end of unpacking routine. We propose a method to detect the end of memory write section before program counter start to execute the original body of packed executable and to identify the memory address that has been written by the executable, and mark it as newly written or rewritten. Our method relies on the number memory section written or rewritten by the executable, the end of unpacking routine is indicated by the last written memory section.

(ii) We identify the threshold of written memory section size for every unpacking routine from the software packer. Normal distribution probability is used to determine dump sample probability threshold of written memory section.

(iii) We propose a method to validate the extracted code from physical memory dump with the original code. We use the similarity percentage to measure the percentage unpacked code from the binary executable under investigation.

2. Related Works

To avoid from being detected by anti-virus software, malware authors obfuscate the binary code using various software packers. By making the malicious code “stealth” from the presence of anti-virus, malware can accomplish the tasks that author has designed for. Rad et al [11] argued that several reasons to use “stealth” techniques in malware. i.e. (i) to have invincibility from all malware analysts (ii) to prevent it from reverse-engineered, (iii) to extend the lifetime of the malware, and (iv) to prevent it being modified or tampered.

There are several type of binary encryption such as oligomorphism [12], polymorphism [13] [14] [15], metamorphic encryption [16], and also a new encryption approach, mimimorphism [17]. According to Sun et al [18], a packer is an executable program that compresses or encrypts the original body of executable file. The key method in extracting a hidden-code from packed executable is to find the exact time when the unpacking routine is completed. The idea is, when a packed executable being run, all of its instructions and data will be loaded into memory. All of the unpacking process will be done at the memory level, with the original body instructions will also be shown after all unpacking process is completed.

PolyUnpack [19] compared the instruction from the static analysis with the state instruction from memory that can be used to acquire the original body of unpacked code by creating the complete executable version of the hidden-code extracted. Renovo [20] focused on the memory write instruction execution at memory level. They used ‘shadow memory’ to taint all the newly written memory section as ‘dirty’. When the instruction pointer jumped to the ‘dirty’ memory, then the executable dump procedure is called. OmniUnpack [21] observed the execution tracking and using a page-level protection mechanism that can monitor both user and kernel space to flag when the code is executed from a page that was modified. Eureka [22] developed a kernel driver that performs a memory dump of the executable and disassemble it whenever the malicious payload has been fully unpacked. Rotalume [5] focused on the emulator based packer, and identified the fundamental characteristic of emulation decoding: iterative main-loop, bytecode fetches using virtual program counter and opcodes monitoring based on the bytecode fetches to create the control flow graph from each emulation packer. Our previous study, Mal-unpack [23], described method of using Binary Instrumentation to extract instruction trace of the running executables at the user space. Some drawbacks from these previous studies is that all of the unpacking engine only can apply for [Type I] up to [Type IV] from Ugarte's Packer
classifications [8]. They cannot detect the modern anti analysis technique, such as metamorphic obfuscator and emulation based packer that can translate or re-encrypt the memory section after its executions as mentioned by Ugarte et al [8] as [Type V] and [Type VI] packers classification. In this paper, Our Mal-Xtract manages to extract the hidden body from [Type V] and [Type VI] packers classification, as defined by Ugarte et al, by tracking memory access (read, write and rewrite) for memory write section size. Comparison between our approach, Mal-Xtract and other methodologies is shown in the table 1.

| Key Items            | Poly Unpack | Renovo       | Eureka       | Mal-unpack   | Mal-Xtract  |
|----------------------|-------------|--------------|--------------|--------------|-------------|
| Memory Access Space  | User Space  | User and Kernel Space | User space   | User Space and Kernel Space | Memory Access Write & Read |
| Features             | Instruction Trace | Runtime Memory Instruction | Runtime System Call API | Instruction Trace | Memory Access Write & Read |
| Tracking End of Unpacking Methods | Comparing the static instruction with runtime memory instruction | Detecting the memory write instruction (mov %eax, [%edi] and push %eax) | Detecting the memory start and termination System Call API and Program counter | Detecting the memory write instruction | Using memory Analysis to find the end of unpacking routine using memory write section threshold |

There are several techniques can be used to perform dynamic binary analysis such as system emulator, system call hooking, dynamic taint analysis, dynamic binary instrumentation and system emulation. Tools such as cuckoo sandbox [24] uses Out-VM Application Programming Interface (API) Hooking as its main analysis. One of the major drawback of hooking technique is that it runs on a user memory space and can not monitor kernel memory space. Another dynamic analysis technique, dynamic taint analysis, e.g. used in Panorama [25], tracks tainted data flow and program behavior. Unfortunately, taint tracking method is not strong enough when dealing with unpacking routine since the analysis need to go deeper into memory and instruction level access. Eureka [22] developed a custom kernel driver to inspect all instruction pointer. Our previous work, mal-unpack [23] uses binary instrumentation to observe the instruction trace at the user memory space only and can not trace kernel memory space. Dynamic binary analysis using system emulation, i.e. QEMU [26], has been applied by various researchers, includes Renovo [20], Ether [27], CWSandbox [28] and Lindorfer et al [29]. One of the major advantage is that QEMU perform instruction translation that enables more detail information tracking regarding instruction, control flow and register content. In this research, we use PANDA [30], an open source Platform for Architecture-Neutral Dynamic Analysis that built upon QEMU [26], a whole system emulator, in which the analysis has access to all data and the executing code in the guest. In addition, PANDA is also capable of recording and replaying executions - enabling iterative, deep the whole system analysis.
3. Mal-Xtract

In this paper, we develop Mal-Xtract, that monitors memory write access both user and kernel memory space for the running binary executable under PANDA [30]. Figure 1 shows the Mal-Xtract framework that allows us to track physical memory access of the running binary executable. The goal is to uncover an indicator of full or partial original hidden-code on the physical memory has been written. To do this, PANDA runs the binary executable and record whole system emulation in a log file [31].

To track memory written by the binary executable, the result recording log of PANDA is replayed with the tracking of memory write and its adjacent memory address selected. All written address and its adjacent memory address will be marked as memory section written by the running binary executable. Since there is no ground truth about how many written bytes in memory that can determine the memory section, we use an empirical approach, assuming the normal distribution of each dump of memory, to extract the revealed code from the memory space. The number of instruction count is recorded during each dump and based on the experiments an upper value of number of instructions is set as the threshold value. Section 3.1 discusses the process of determining memory section threshold.

Once the physical memory dump is available, Volatility memory forensic tool is used to extract the binary executable and the result is validated. This binary executable resulting from memory dump cannot be executed directly in the Windows environment due to missing or damaged Import Address table (IAT). Most software packers manipulate import address table to protect its code obfuscation and to perform anti-analysis. The pseudo code of the proposed Mal-Xtract is listed in Algorithm 1.

3.1. Finding Written Memory Section Threshold

To determine written memory section threshold, we track the memory address for writing, especially in contiguous manner, e.g. starting at memory address 0x00100101 and next memory address would be 0x00100102, 0x00100103, etc. Whenever the memory address written started to jump into different memory address such as 0x70089030, the memory section size is counted and the memory section is marked. Each memory write/rewrite section are stored in a log and marked the above threshold section as instruction section. The advantage of this method is that it does not depend on Virtual Address Descriptor (VAD) section, but on the actual memory...
Algorithm 1 Memory Analysis Algorithm
1: procedure Main Memory Analysis
2: Perform Recording from the Executables
3: Replay the Recording using ASIDSTORY ▷ Get PID from the executables
4: Replay the Recording using BUFMON ▷ Get all Memory Write/Read Log from PID
5: ▷ Parsing Raw Memory Write/Read Log to Memory Address Write / Rewritten
6: i = 1
7: while Line(i) <> End Of File Memory Logs do
8: if Address(i) == Written Address then
9: AddressWrittenCount + 1
10: else
11: New Written Address ▷ add New Written Address into Memory Address List
12: i ++ ▷ Parsing the Memory Address List to memory section by capturing all adjacent Address
13: j = 1.
14: while Address(j) <> End Of Memory Address List do
15: if Address(j + 1) == Address(j) then
16: Address Section Size + 1
17: else
18: New Written Memory Section (Section Size)
19: j ++ ▷ Parsing the Memory Section to the Memory section Threshold
20: k = 1.
21: while Memory Section(k) <> End Of Memory Address Section List do
22: if Memory Section(k) > Memory Dump Threshold then
23: Mark the Memory Section as Memory Dump Threshold
24: j ++ ▷ Parsing The percentage of each dump threshold
25: for each $T \in \text{DumpThresholdS}$ do
26: Get Instruction Count from the latest Memory Section of Threshold $T$
27: Percentage of Threshold $T = \text{Instruction Count (T)} / \text{Instruction Total of Recording}$

write. According to Sharif et al [22], any memory section could contain data or code, and when unpacking process occurred, the packer will unpack the original instruction and data to the physical memory [8]. To determine the threshold of memory section, an empirical approach is used, assuming normal distribution statistics of all memory dumps of our experiments. In our early experiments, written memory section tends to be larger and gets smaller as the unpacking routine completes it tasks. Figure 2 shows the dump performed in the range of threshold from 50,000 bytes to 100 bytes per written memory section. Thus, a minimum of 100 bytes of memory section is used as the lower level threshold value since all of the software packers investigated in our early experiments are already unpacked at this value. On the other hand, memory section of size 50,000 bytes are chosen to the upper level threshold value for any memory sections found above 50,000 bytes or above. For example, for some software packers, there is a write up to 200k bytes for 1 memory section, and they are considered as part of 50,000 byte range.
3.2. Hidden-Code Extraction Validation

To validate the code extracted from each dump of the running executable, process dump is performed after each dump completed, using Volatility [32] with procdump feature. Since the executables extracted from this process which include slack space between the Portable Executable (PE) sections are not page-aligned, further extraction is performed to cover all the expanded section written after the executables run. While Renovo [20] extract only from memory sections that had been expanded from the executables, our extraction method ensures a more complete code coverage from the memory dump. The binary executables resulted from the dump are further processed using IDA Pro [33], a debugger and static analysis tool, to provide an assembly ASM file. The ASM file, which consists of all the code and data from the sections of dumped executables, are then parsed using python script to obtain all the mnemonic instructions or x86 Opcode Instructions [34]. Only the mnemonic instructions are retrieved since the data and assembly variables part are already translated for every packer as shown in figure 4 and 5. Figure 4 shows the comparison the API-string translation from the dumped packed executable (left) and the original executable (right) while figure 5 presents assembly variable translation from the dumped packed executable (left) and the original executable (right).
Figure 5. Assembly Variable Translation (Left: Dumped Packed Executable, Right: Original Executable)

The same process is also applied to the original binary executable (no software packer is applied), the parsed instructions from original binary executable is then compared using context diff comparison technique, which first align the whole line of each executable’s parsed ASM file and then search for diff line for each aligned line. All the diff line are totaled and similarity percentage is then calculated using the following equation:

\[
\text{Similarity} = \frac{\sum \text{Diff}}{\sum \text{Line}} \times 100\%
\] (1)

Other than similarity percentage, file entropy, is used to determine whether the binary executable is packed. Thus the combination of similarity percentage value and file entropy are used in our research whether the binary executable file extracted is packed.

4. Experiment Data
A similar sample binary executables as [36], two generic Windows applications in Windows 7 (calc.exe and notepad.exe) and one custom standalone program (helloworld.exe) that print string in the command prompt, are used in this research. These binary executables are packed with a collection of third party binary packers, both free and commercial packers, i.e. UPX [37], PECompact2 [38], FSG [39], Armadillo [40], Molebox [41], WinUPack [42], Themida [43] and VMProtect [44]. We also evaluate our method on 20 malware samples, obtained from from VirusShare and MALHEUR [45] dataset, which contain of 4 different malware families, Lipier (2009), domaiq (2014), Sality (2015), and Uparte (2015). For all malware samples, only packed malware samples are selected and used in this research, standard file entropy [35] is used to check for packed executables.

5. Experiment Setup
All experiments are performed using an Apple Macintosh i7 2.7 Ghz with 16 GB RAM Mac OS X 10.12, VMWare Fusion 8.0.2 that runs Ubuntu 14.04 with 6 GB RAM and 4 Core Processor. PANDA QEMU is installed inside Ubuntu VM and runs Windows 7 32bit SP1 with 2 GB RAM for its QEMU VM Emulation and all samples executed inside the Windows 7 OS. All the sample binary executables also gathered from Windows 7 32bit SP1 Portable Executables. We are using Windows XP SP2 VM for running IDA PRO to extract ASM file from extracted binary executables.
6. Evaluation and Discussion

For all the memory dumps, the lower and upper memory section threshold values of 100 bytes and 50,000 bytes respectively, are used in this research. In addition, based on Lyda [35], file entropy threshold value of 6.67 is used to indicate whether a file is packed. Table 2 shows the memory dump results of various packed sample binary executables.

Even tough Armadillo v8.6, categorized as Type VI packer [8], implements numerous anti-reverse-engineering technique, our method could extract the whole body from original body in the Dump 5000 with similarity of 99.6%. Similarly, for UPX and WinUpack as Type I packer, FSG, PECompact2 and Molebox that categorized as Type III packer can be extracted with 99% similarity. Thus, for all type of common software packer unpack their original body into physical memory at the threshold of 5000 bytes as the memory write section threshold that indicate the end of unpacking routine before it continue to execute the unpacked instructions.

| Packers   | D   | E  | S    | D   | E  | S    | D   | E  | S    |
|-----------|-----|----|------|-----|----|------|-----|----|------|
| Armadillo | 5000| 5.96| 99.6%| 5000| 5.61| 98.8%| 5000| 5.72| 99% |
| FSG       | 50000| 4.8 | 99.8%| 50000| 5.8 | 99.8%| 50000| 5.1 | 99.8%|
| Molebox   | 15000 | 6.2 | 99.6%| 25000 | 6.2 | 99.8%| 15000 | 6.0 | 99% |
| UPX       | 25000 | 3.7 | 99.6%| 25000 | 5.3 | 99.8%| 25000 | 5.2 | 99.1%|
| PECompact2 | 8000 | 3.56| 99.6%| 25000 | 6.12| 99.8%| 25000 | 5.96| 99.2%|
| WinUpack  | 50000 | 3.5 | 99.6%| 50000 | 4.5 | 99.8%| 50000 | 4.2 | 99.6%|
| Themida   | 50000 | 7.6 | 97.1%| 50000 | 7.91| 97.4%| 50000 | 7.3 | 97.5%|
| VMProtect | 50000 | 6.2 | 97.5%| 50000 | 6.1 | 97.2%| 50000 | 6.3 | 97.3%|

Remark:
D Memory Section Threshold
E Max Entropy from Executable Sections
S Similarity Percentage

For software packer that uses virtualization which implement advanced code obfuscation using emulation and bytecode translation [10], our method successfully extract 97% the original body at dump 50,000 bytes. From the statistical view in Figure 6, UPX, FSG, PE2 and WinUpack, reached maximum similarity at dump 50,000, while some other packers only reach the same similarity level at 5000 bytes. This is due to UPX, FSG, PE2 and WinUpack are categorized as compression packers which don’t have any code protection feature. However, memory dump with the lower memory section value than 50,000 bytes, the similarity was significantly decreased to 89% for VMProtect and no executables could be extracted for Themida packed executables. In fact, in our experiments, no file extraction is possible for memory dump with the lower memory section value at 30,000 bytes. Themida starts to apply its memory protection features from memory dump 30,000 bytes to memory dump 100 bytes. Based on this observation, we can conclude that Themida and VMProtect write all original body in physical memory before any emulation and anti analysis routine are applied. In addition, there are also differences between all the unpacked dump on each executable due to the complexity of instructions for each binary executable; the more complex binary executable, the more more protection, encryption and obfuscation in the code and memory level.
Figure 6. Similarity Level for Dumped Packed Executables

Figure 7 shows the memory section distribution for all binary executable, Benign (not packed) and Packed binary executables, used in our research. Based on the graph in figure 7, there is a large memory section access (write and rewrite) at memory dump of 100 bytes, but we have either the value of 0 or 1 for memory section access at the memory dump of 50,000 bytes. It is also interesting to note that all binary executables, including benign binary executable, demonstrate a large memory section usage at the memory dump 100 bytes. Thus, for all software packers the unpacking process range from memory dump 50,000 bytes up to memory dump 100 bytes, and the unpacked binary executable (both from benign and packed binary executable) run normally when the memory section usage hits 100 bytes.

Table 3 shows the result of our evaluation of the malware samples using Mal-Xtract framework. Static analysis tools such as PEStudio [47] is then used to gather all static information from file extracted from memory dump of running malware binary samples. Due to the lack of ground truth to validate the unpacked malware samples, a combination of parameters are used in this research, i.e. Indicator of Compromise (IoC) [48], Blacklisted string and Suspicious API Import as comparative features from before malware unpacking process and after malware unpacking itself. The results show that our proposed method successfully unpack all the packed malware indicated by much lower entropy value after unpacked [35].

During the unpacking process, malware samples evaluated exhibit a larger body size before and after the process, as the binary executable writes a new section or rewrite the existing section. All the comparative features show a positive results except for Upatre family. On Upatre family no obfuscation is found, i.e. on the suspicious string or API, but a larger size for each unpacked dump malware can be observed. This type of packer can be correlated with Ugarte [8] as Packer type I,II, and III. On the other hand, Obfuscator such as Armadillo and Molebox, which can be correlated with Ugarte [8] as packer type IV, V, and VI. In the malware experiments, Lipler, Domoiq and Sality malware family demonstrated obfuscation functionality, the malware samples have a lot of suspicious strings and API calls based on Indicator of Compromise static analysis.

The limitation of dump executable resulted from Mal-Xtract is not a runnable executable inside Windows Operating System, due to the Import Address Table that already tampered by
Figure 7. Memory Section Distribution for Packers

Table 3. Malware Unpacking Result

|   | Indicator of Compromise | Suspicious API Import | Blacklist String |
|---|--------------------------|-----------------------|------------------|
|   | Families | Result | Before | After | Before | After | Before | After |
| Lipler | Full | 5 | 7 | 49 | 109 | 57 | 218 |
| Domaiq | Full | 15 | 10 | 72 | 180 | 227 | 568 |
| Upatre | Full | 7 | 6 | 52 | 52 | 90 | 82 |
| Sality | Full | 5 | 19 | 1 | 1 | 3 | 30 |

the packer. There are also huge overhead in the analysis logs because we are using recording and memory access capture.

7. Conclusion

Mal-Xtract, a memory analysis based on system emulation to extract hidden-body code has been presented. Our experiments shows that memory analysis technique is effective to handle common [Type I] to [Type IV] software packers [8]. Depending on type of packers, memory write section may differ: memory write section for most software packers can be found at the threshold of dump 5,000 bytes and memory write section for advanced software packer that utilizes virtualization is found at dump 50,000 bytes. Our experiment results show that at least 97% of the original code from the various binary executable packed with different software packers could be extracted. Our proposed method has also proved to be successful in extracting hidden code from recent malware family samples tested in this research. In the future, we are planning to include instruction and register tracing as part of the features to determine the completion of unpacking routine.
References

[1] AVTEST Av test - the independent it security institute URL [https://www.av-test.org/en/statistics/malware/]

[2] Ceccato M, Di Penta M, Falcantoni A, Falcarin F, Torchio M and Tonella P 2014 Empirical Software Engineering 19 1040–1074 URL [https://dx.doi.org/10.1007/s10664-013-9248-x]

[3] Viticchié A, Regano L, Torchio M, Basile C, Ceccato M, Tonella P and Tiella R 2016 Assessment of source code obfuscation techniques In Proceedings of the 16th IEEE International Working Conference on Source Code Analysis and Manipulation (IEEE Computer Society) pp 11–20 URL [http://dx.doi.org.ezproxy.bibli.polito.it/10.1109/SCAM.2016.17]

[4] Balachandran V and Emmanuel S 2013 IEEE Transactions on Information Forensics and Security 8 669–681 URL [https://dx.doi.org/10.1109/TIFS.2013.2250964]

[5] Sharif M, Lanzi A, Giffin J and Lee W 2009 Automatic reverse engineering of malware emulators 2009 30th IEEE Symposium on Security and Privacy (IEEE) pp 94–119 URL [http://dx.doi.org/10.1109/SP.2009.27]

[6] Kim J, Lee J Y, Chang H Y, Cho S, Park Y, Park M and Wilsey P A 2010 Design and performance evaluation of binary code packing for protecting embedded software against reverse engineering 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (IEEE) pp 80–86 URL [http://dx.doi.org/10.1109/ISORC.2010.23]

[7] McAfee Labs 2015 McAfee Labs URL [http://www.mcafee.com/us/resources/white-papers/wp-good-bad-the-unknown.pdf]

[8] Ugarte Pedrero X, Balzarotti D, Santos I and Bringas P G 2015 SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers SSP 2015, IEEE Symposium on Security and Privacy, May 18-20, 2015, San Jose, CA, USA (San Jose, UNITED STATES) URL [http://dx.doi.org/10.1109/SP.2015.48]

[9] Moser A, Kruegel C and Kirda E 2007 Limits of static analysis for malware detection Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual (IEEE) pp 421–430

[10] QEMU Qemu - open source processor emulator URL [http://wiki.qemu.org/Main_Page]

[11] Padir B B, Masrom M and Ibrahim S 2012 International Journal of Computer Science and Network Security 12 74–83 URL [http://paper.ijcns.org/07_book/201208/20120813.pdf]

[12] Aycock J 2006 Computer viruses and malware vol 22 (Springer Science & Business Media) URL [http://dx.doi.org/10.1007/0-387-34188-9]

[13] O’Kane P, Sozer S and McLaughlin K 2011 Security & Privacy, IEEE 9 41–47 URL [http://dx.doi.org/10.1109/SP.2011.98]

[14] Szor P 2005 The art of computer virus research and defense (Pearson Education)

[15] Li X, Loh P K and Tan F 2011 Mechanisms of polymorphic and metamorphic viruses Intelligence and Security Informatics Conference (EISIC), 2011 European (IEEE) pp 149–154 URL [http://dx.doi.org/10.1109/EISIC.2011.77]

[16] Christodorescu M and Jha S 2003 Static analysis of executables to detect malicious patterns USENIX Security Symposium

[17] Wu Z, Gianvecchio S, Xie M and Wang H 2010 Mimimorphism: A new approach to binary code obfuscation Proceedings of the 17th ACM conference on Computer and communications security (ACM) pp 536–546 URL [https://doi.org/10.1145/1866307.1866368]

[18] Sun L, Versteeg S, Boztaş S and Yann T 2010 Pattern recognition techniques for the classification of malware packers Information security and privacy (Springer) pp 370–390 URL [http://dx.doi.org/10.1007/978-3-642-14081-5_23]

[19] Royal P, Halpin M, Dagon D, Edmonds R and Lee W 2006 Polyunpack: Automating the hidden-code extraction of unpack-executing malware Computer Security Applications Conference, 2006. ACSAC’06. 22nd Annual (IEEE) pp 289–300 URL [http://dx.doi.org/10.1109/ACSAC.2006.38]

[20] Kang M G, Poosankam P and Yin H 2007 Renovo: A hidden code extractor for packed executables Proceedings of the 2007 ACM workshop on Recurring malware (ACM) pp 46–53 URL [https://doi.org/10.1145/1314389.1314399]

[21] Martignoni L, Christodorescu M and Jha S 2007 Omnimapack: Fast, generic, and safe unpacking of malware Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual (IEEE) pp 431–441 URL [http://dx.doi.org/10.1109/ACSAC.2007.15]

[22] Sharif M, Yegneswaran V, Saidi H, Porras P and Lee W 2008 Eureka: A framework for enabling static malware analysis Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security (Springer-Verlag) pp 481–500 URL [http://dx.doi.org/10.1007/978-3-540-88313-5_31]

[23] Lim C, Sulistyan D Y, Ramli K et al. 2015 Advanced Science Letters 21 3333–3336 URL [https://doi.org/10.1166/asl.2015.6487]
[24] Oktavianto D and Muhardianto I 2013 Cuckoo Malware Analysis (Packt Publishing Ltd)

[25] Yin H, Song D, Egele M, Kruegel C and Kirda E 2007 Panorama: capturing system-wide information flow for malware detection and analysis Proceedings of the 14th ACM conference on Computer and communications security (ACM) pp 116–127 URL https://doi.org/10.1145/1315245.1315261

[26] Bartholomew D 2006 Linux Journal 2006 3

[27] DINABURG A, ROYAL P, SHARIF M and LEE W 2008 Ether: malware analysis via hardware virtualization extensions Proceedings of the 15th ACM conference on Computer and communications security (ACM) pp 51–62 URL https://doi.org/10.1145/1455770.1455779

[28] WILLEMS C, HOLZ T and FREILING F 2007 IEEE Security and Privacy 5 32–39 URL https://doi.org/10.1109/MSP.2007.45

[29] LINDBERG M, KOLBITSCH C and COMPARETTI P M 2011 Detecting environment-sensitive malware International Workshop on Recent Advances in Intrusion Detection (Springer) pp 338–357 URL https://dx.doi.org/10.1007/978-3-642-23644-0_18

[30] DOLAN-GAVIT B F, HODOSH J, HULIN P, LEEK T and WHELAN R 2014 Repeatable reverse engineering for the greater good with panda Tech. rep. Department of Computer Science, Columbia University URL http://dx.doi.org/10.7916/D8MNIC1P

[31] PANDA Panda - open-source architecture-neural dynamic analysis URL https://github.com/panda-re/panda

[32] VOLATILITY: open source memory forensics URL http://www.volatilityfoundation.org

[33] IDA is a windows, linux or mac os x hosted multi-processor disassembler and debugger URL https://www.hex-rays.com/products/ida/

[34] x86 opcode and instruction reference URL http://ref.x86asm.net/coder32.html

[35] Lyda R and Hamrock J 2007 IEEE Security & Privacy 5 40–45 URL https://doi.org/10.1109/MSP.2007.48

[36] BÖHNE L 2008 Pandorabochs: Automatic unpacking of malware Ph.D. thesis University of Mannheim URL http://2014.back.lu/archive/2009/PandoraBochs.pdf

[37] UPX, a free and open source, cross-platform runtime packer URL http://upx.sourceforge.net/

[38] PECompact2, windows executable compressor URL http://pecompact2.software.informer.com

[39] FSG 2.0, [fast] [small] [good] perfect compressor for executable files URL http://www.downloadpcsoft.com/Windows/Development/Other/FSG_24767.html

[40] ARMADILLO, overlays packer and obfuscator URL http://the-armadillo-software-protection-system.software.informer.com

[41] Molebox, a free executable compression and encryptor URL https://molebox.en.softonic.com

[42] Winupack, a freeware runtime packer URL http://www.softpedia.com/get/PORTABLE-SOFTWARE/Compression-Tools/Windows-Portable-Applications-Portable-WinUpack.shtml

[43] Themida, a commercial executable protector URL http://www.oreans.com/

[44] Vmprotect, a commercial executable virtualization URL http://vmprotect.com/

[45] RIECK K, TRINIS P, WILLEMS C and HOLZ T 2011 Journal of Computer Security 19 URL https://dx.doi.org/10.3233/JCS-2010-0410

[46] Rolles R 2009 Unpacking virtualization obfuscators 3rd USENIX Workshop on Offensive Technolo-gies (WOOT) URL http://dl.acm.org/citation.cfm?id=1855877

[47] Pestudio URL https://www.winitor.com

[48] COHEN M, BILBY D and CARONNI G 2011 digital investigation 8 S101–S110