Generic Unpacking using Entropy Analysis

Guhyeon Jeong*, Euijin Choo**, Joosuk Lee*, Munkhbayar Bat-Erdene*, and Heejo Lee*

*Div. of Computer & Communication Engineering, Korea University, Seoul, Republic of Korea.
Email: {ghjeong, jupiter7, munkhbayar, heejo}@korea.ac.kr
**Department of Computer Science, North Carolina State University.
Email: {echoo@ncsu.edu}

Abstract

Malwares attempt to evade AV scanners using various obfuscation techniques. Packing is a popular obfuscation technique used by 80% of malwares. In this paper, we propose a generic unpacking mechanism to find the original entry point (OEP) using entropy analysis. The experiment using 110 packed executables demonstrates the proposed mechanism can locate the OEPs of 72% of the packed executables. Furthermore, we show how the mechanism could be applied to packed malwares.

1 Introduction

Packing is one obfuscation technique. A packed executable includes a compressed original executable, and an unpacking module. The unpacking module decompresses the compressed executable and runs the decompressed one. As packing uses a compression algorithm, a packed executable is naturally obfuscated. The increasing number of variants and unknown packers gives malware writers many choices. Moreover, experts are capable of building their own packers to evade signature-based AV scanners effectively. The study [8] presented by Symantec Research Laboratories showed, that over 80% of malwares is packed.

Nevertheless, AV programs still respond to packed malwares by updating signatures to detect newly packed malwares. The time to analyse packed malwares, and find their signature takes much longer compared to the time to create new packers. Consequently, some AV scanners simply report all executable files compressed by the same packer as viruses, causing false alarms [16].

In this paper, we propose a generic unpacking mechanism to find the original entry point (OEP) using entropy analysis. Robert Lyda *et al.* showed that binary files with a higher entropy score tend to be correlated with the presence of encryption or compression [11]. Applying this fact, the concept arises that an entropy score of memory space is continuously changed while packed instructions are unpacked into memory.

Contributions of our research follow. First, with the help of entropy analysis, we can determine the moment at which unpacking is completed. The proposed mechanism effectively finds the OEP. Second, our approach does not rely on signatures and therefore, it can locate the OEP, even packed with an unknown packer.

The remainder of paper is organized as follows. In section 2, we review related work. The proposed mechanism is described in section 3. We introduce the concept of packed executables and their entropy using information theory, and then our approach to unpacking. Section 4 shows our experimental results and analysis. In this section, the results indicate that our approach can solve the unpacking problem. Finally,

978-1-4244-9356-2/10/$26.00

we conclude in section 5.

2 Related Work

Most of unpacking research run a packed executable in a specific environment, such as a debugger or a virtual machine, to control its execution flow. Accordingly, it is possible that finding hidden instructions generated during an unpacking process. The examples include PolyUnpack [12], OmniUnpack [13], Renovo [9], Justin [8], and Hump-and-Dump [15]. In contrast to the formers, a static approach [6] was proposed.

PolyUnpack [12] performs static analysis over a packed executable to acquire a model of what its execution would look like if it did not generate and execute code at runtime. When the first instruction of a sequence not found in the static model is detected, the unknown instruction sequence is written and the execution of the packed executable is halted.

OmniUnpack [13] monitors the program execution and tracks written, as well as written-then-executed, memory pages. When the program makes a potentially damaging system call, OmniUnpack invokes a malware detector on the written memory pages. If the detection result is negative, execution is resumed. If new type of malware appears, the dangerous system calls they defined on their paper could not match.

Renovo [9] utilizes a virtual machine. By using a virtual machine, they run a packed executable and record memory writing operations on shadow memory. When execution flow reaches one of checked bits of the shadow memory, all the checked memory bits are dumped. Shadow memory is changed to extract hidden code from packed executables with multiple hidden layers. With this mechanism, Renovo can find hidden layers as well.

Justin [8] is a generic unpacking solution. It is designed to detect the end of unpacking of a packed binary's run and invoke AV scanning against the process image at that time. The difference to other research is that Justin incorporates *Dirty Page Execution*,*Unpacker Memory Avoidance*, *Stack Pointer Check* and *Command-Line Argument Access* for accurate end-to-unpacking detection.

Hump-and-Dump [15] is a different approach from other research. Hump-and-Dump tries to find the OEP. Using a characteristic of unpacking, it counts the number of loops used in unpacking. When the number of loops is greater than a threshold and no more big loops are used for the period of a threshold, the address of the loop end point is the OEP.

Recently, Kevin Coogan *et al*. proposed an automatic static unpacking mechanism [6]. It uses static analysis techniques to identify the unpacking code that comes with a given malware binary, then uses this code to construct a customized unpacker for that binary. This customized unpacker can then be executed or emulated to obtain the unpacked malware code.

Unpacking tools include VMUnpacker [7], QuickUnpack [3] and RL!Depacker [10]. These tools can unpack executables packed with what they have analysed. However, this *case-by-case* approach is inefficient. First, it costs too much to analyse all the packers and the analysis should be done manually. Second, it will not work for variants.

Suggested unpacking mechanism can effectively cooperate with malware detection systems such as BitBlaze [14] and SplitScreen [5] because, as aforementioned, 80% of malwares are packed before having distributed.

3 A Generic Mechanism to Find the OEP

Finding the OEP is a primary requirement for unpacking. An automated unpacking mechanism is also necessary to unpack lots of packed malwares. An automated unpacking mechanism requires *a generic characteristic* because automated but not generic mechanisms are limited to respond to unknown packers. Therefore, these two requirements should be satisfied by an unpacking mechanism.

In this section, a generic mechanism satisfying the above two requirements is described. We first explain fundamental knowledge of packed executables. Next, entropy analysis is explained. It is a core concept making our approach meet both requirements. Finally, our approach is described based on this background knowledge.

3.1 Packed Executables

A packed executable is built with two main parts during a two phase packing process. First, the orig-

inal executable is compressed and stored in a packed executable as data. Second, a decompression module is added to the packed executable. The decompression module is used to restore the original executable.

Unpacking is the reversal of packing. Decompression is first conducted and the execution flow jumps to the first instruction of the unpacked code. After restoring the original executable, execution flow jumps from the end point of the decompression module to an entry point of the original executable.

3.2 Entropy Analysis

In information theory, entropy is a measure of uncertainty in a series of an information unit. Information is compressed by following a logical sequence. First, some repeated patterns are found in the information, and then the redundancies of the patterns are used to reduce the size of the information. That is, the number of patterns of the information is reduced by compression and a series of bits becomes more unpredictable, which is equivalent to uncertainty. Therefore, the measured entropy of compressed information is higher than of the original information.

Shannon's formula is devised to measure information entropy, as follows:

$$H(x) = -\sum_{i=1}^{n} p(i) \cdot log_b^{p(i)},$$

where H(x) is the measured entropy value and p(i) is the probability of an i^{th} unit of information in event x's series of n symbols. The base number of the logarithm can be any real number greater than 1. However, 2, 10, and Euler's number e are chosen in general.

3.3 Proposed mechanism

The main concept is derived from the difference of measured entropy values between packed and unpacked instructions informing analysers an unpacking process is being conducted. Basically, we execute a given packed executable, and let it conduct unpacking process. During an unpacking process, packed instructions are unpacked by a decompression module, and intuitively, measured entropy of the memory space will be changed. Eventually, the end of unpacking can be detected by monitoring the cessation of entropy changes.

In our approach, an executable is given as an input and the approach locates the OEP as a result if the executable is packed. Additionally, we note an assumption that we can assure if an executable is whether packed, based on [11]. The remainder of this section details the proposed approach.

First, the executable is executed and keeps running unless the instruction is one of instructions such as JMP, JCC, CALL, or RET. If one of the instructions is encountered, execution is paused and entropy analysis for that instant of the process is conducted because when unpacking is completed, those kinds of instructions should be used to change an execution flow from the end of the decompression module to the beginning of the unpacked original code, the OEP.

Entropy analysis is conducted by measuring memory spaces. It decides whether or not unpacking process is complete by measuring entropy of the each section of a packed executable, and checking if an instruction jumps to an address in a section where unpacked code is written. The next step is determined by the result of the entropy analysis. If the packed executable is unpacked, the OEP is located. Conversely, if the unpacking process is incomplete, the paused process continues to execute the next instruction.

Another issue about execution-flow-changing instructions is the number of the instructions in a program. An unpacking module consists of several iterations, which uses those instructions; thus, increasing analysis time. To solve this problem, we cached a number of addresses of JMP and JCC instructions that are met during analysis; if a cached address is reached again, entropy analysis at that point of time is skipped.

Using the formula explained in the *Entropy Analysis* section, the entropy of the filtered data can be measured. We use the logarithm to the base Euler's number e, and the unit of data is a byte. During an unpacking process, instructions are unpacked in a section. A value of the measured entropy implies the data state (e.g. packed, unpacked, or being unpacked) in each section at that moment. Our experimental results show that the entropy values change while a packed executable is unpacked. As a consequence, we can determine if unpacking is complete using entropy analysis.

Different from related works such as PolyUnpack [12], OmniUnpack [13], or Renovo [9], our approach locates OEP. Consequently, we are able to restore packed malwares to original ones to analyse them. It means that we can reuse the original malwares to analyse, instead of conducting redundant dynamic analysis every time we need to analyse. When it comes to Hump-and-dump [15], it relies on a threshold in order to find OEP, and it could make false positives. Although Justin [8] shows an impressive result on their paper, there is a possibility that it could be bypassed because it uses heuristics. In contrast, our approach is based on a theory, which makes our approach more reliable. A flaw exists in ours that an entropy could be modified by adding garbage data. It needs to be solved by optimising monitored memory areas in the future work.

4 Experimental Results and Analysis

In this section, we show experimental results with analyses. We show that the measured entropy of unpacked code is similar to the original executable's entropy. Next, the result for unpacking is given; it demonstrates a 72% success rate. Patterns of packers are also presented with graphs of packed executables' measured entropies. The patterns are used to categorise packers. Finally, effectiveness of a cache, which is used to accelerate the experiment, is presented. Some packers are chosen for the experiments in consideration of related works [12] [13] [9] [8] [15] [6].

Table 1. An experimental set of executables and packers

Executables	Packers
freecell, notepad, msiexec, telnet, calc, winmine, mshearts, mspaint, spider, dxdiag	alter_exe, aspack, fsg, molebox, morphine, mpress, nPack, nSpack, RLPack, UPX iT, upxn

4.1 Entropy of Unpacked Code

Although entropy scores for each type of data is given in [11], entropy scores of executables and memory space could be different. Thus, we measure entropy scores of unpacked code sections. It would be the best case if all the unpacked code is measured as same entropy, unfortunately it is not. Hence, we need to set a range of possible entropy values for being unpacked. We use two constants, termed E_{min} and E_{max} to determine if unpacking is complete.

To set the constants, we first run packed executables and dump code sections. Garbage values can exist in the dumped data, and should be erased. There are too many consecutive zeros in bytes, which are garbage values in most cases, in memory dumps. If twelve consecutive bytes exist in the memory dumps, we erase the data because there are rarely instructions with the same twelve consecutive bytes.

Although this idea might look nonsense, it has two reasons. One is that the maximum length of an instruction is fifteen because if it exceeds fifteen, an x86 system would generate an exception. The other is that disassembling dumped data to delete garbage values costs too much. However, these are still limitations to be improved later in the future work.

By reducing the garbage values, entropy of only unpacked code can be measured. Table 2 shows measured entropies of the code sections of executables. It shows unpacked instructions have similar values to their entropy, though they are not exactly same. 45 executables randomly chosen in the Windows system directory are analysed to measure entropy. This experiment determines E_{min} and E_{max} to be 4.1 and 4.6, respectively.

Table 2. Measured entropy scores of ten unpacked code used in the unpacking experiment

	UPX	aspack	mpress	nspack	Original
calc.exe	4.26	4.30	4.26	4.26	4.30
dxdiag.exe	4.26	4.27	4.26	4.26	4.27
freecell.exe	4.33	4.38	4.23	4.33	4.37
mshearts.exe	4.28	4.32	4.25	4.32	4.32
msiexec.exe	4.34	4.35	4.34	4.34	4.35
mspaint.exe	4.38	4.41	4.38	4.38	4.40
notepad.exe	4.30	4.37	4.30	4.29	4.37
spider.exe	4.61	4.63	4.61	4.61	4.63
telnet.exe	4.46	4.49	4.46	4.46	4.50
winmine.exe	4.41	4.45	4.40	4.40	4.45

Table 3. Correctness of analysed OEPs for 110 packed executables

	alter_exe	aspack	fsg	molebox	morphine	mpress	nPack	nSpack	RLPack	UPX iT	upxn
calc.exe	C	C	C	C	F	C	C	C	C	C	C
dxdiag.exe	F	C	F	C	F	F	C	F	F	F	F
freecell.exe	C	C	C	C	F	I	C	C	C	C	C
mshearts.exe	C	C	C	C	F	I	C	C	C	C	C
msiexec.exe	C	C	C	C	F	I	C	C	C	C	C
mspaint.exe	C	C	C	C	F	I	C	C	C	C	C
notepad.exe	C	C	C	C	F	I	C	C	C	C	C
spider.exe	C	C	C	C	F	I	C	C	C	C	C
telnet.exe	C	C	C	C	F	I	C	C	C	C	C
winmine.exe	F	C	F	C	F	F	C	F	F	F	F

C:CORRECT (72%), I:INCORRECT (6%), F:Failure (22%)

4.2 Found OEP

We generate 110 packed executables by packing each of the eleven executables using each ten packers to conduct an experiment indicating the generic characteristics of our approach. As we are aware of the original executables, we found the OEPs of packed executables from the experiment can be compared to the real OEPs to evaluate our approach. Table 3 illustrates the accuracy of the analysed results. Three cases may occur in this experiment. Each result obtains either the correct or incorrect OEP. Correct and incorrect results are written as C and I, respectively. The other case is a failure to analyse a packed executable, and is written as F.

This unpacking result indicates our approach locates 72% of exact OEPs. Traditional unpacking research is not able to find the OEP. Similar to our work, in Hump-and-Dump [15], they also try to find the OEP in a packed executable. However, their algorithm depends on some thresholds, whose values are not given in their paper. It indicates their approach to solve this problem could be evaded by modifying the number of loops or each of the iterations. In contrast, our approach utilizes the nature of information, which cannot be modified. Therefore, our approach is more reliable than [15].

Some incorrect (6%) or failed (22%) results exist in Table 3. The first case of incorrect results occurs for a packer, *mpress*. It unpacks code and jumps to the unpacked code at the very last part of unpacking. However, the jump is not directed to the OEP. As it is almost unpacked at the moment, entropy is also in the range between E_{min} and E_{max}. This is the cause of the imprecision. The packer does the remainder of work before jumping to the OEP. In this case, the OEP found is very similar to the real OEP as parts of original code have been restored.

22% of failures occurs under the following condition. When execution flow reaches the OEP, if the measured entropy value at that moment is less than E_{min} or greater than E_{max}, the analysis tool cannot find the OEP, and just ignores it. Due to our strong assumptions about the memory space to be monitored and the range between E_{min} and E_{max}, this failure occurs a few times.

These two exceptions are caused by the strong assumptions in this paper. The 28% fail-to-find-OEP would become lower with additional research on two issues. First, determining memory space to be monitored. Second, minimizing the range between E_{min} and E_{max}. These exceptions will be considered in future work to improve the performance of our approach.

4.3 Patterns of packers

Another potential capability of our approach is categorising packers by their unpacking patterns. Not only variants of packers, but also some packers behaving in a similar way to each other, could be categorised using this approach. We draw graphs to show patterns of unpacking processes. The graphs are of the code section. Figure 1 shows what happens during an unpacking process in memory space in terms of entropy. The order of execution-flow-changing instructions and measured entropy values are shown on the X and Y

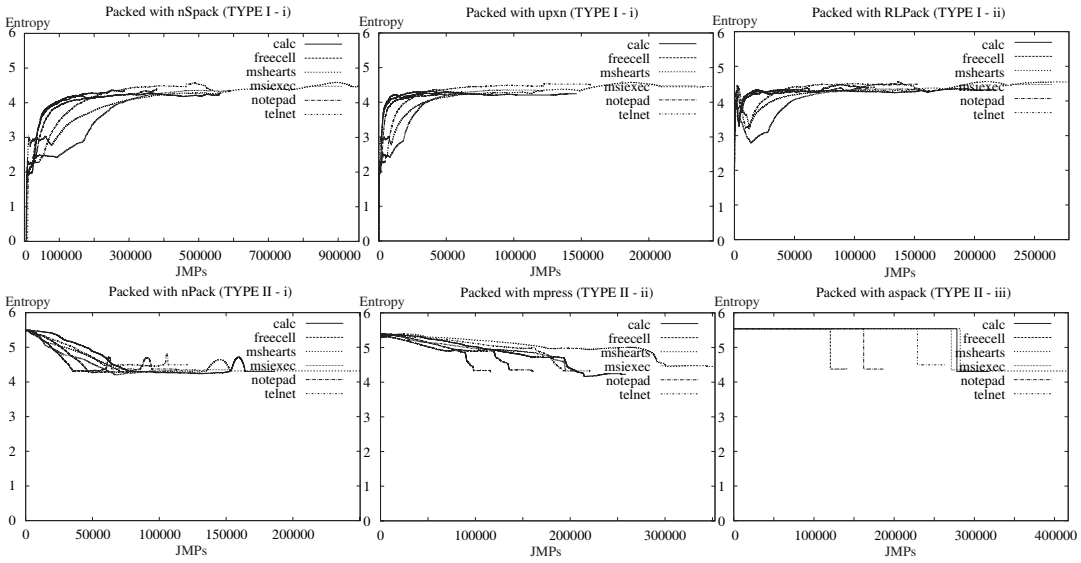

Figure 1. Patterns of unpacking processes

axes, respectively. The concept of cache is applied to decrease the experimental time due to the many execution-flow-changing instructions. In Figure 1, the order of execution-flow-changing instructions are the instant of each entropy analysis, execution-flow-changing instructions in a cache are skipped. In this experiment cache size is 15. Detailed explanation of cache is given in the next section.

Figure 1 shows patterns of changes of entropy values during the unpacking process for each packer. Scales of patterns distinguish between packed executables. Although the scales are not proportional to the size of packed executables, interestingly, the executables packed with the same packer still make the same patterns during unpacking processes. Thus, a pattern of an unpacking process could be applied to find the same packer family. Packer patterns fall into two types.

Packers of type I initialize memory space, where unpacked code will be written, as zeros; it starts with zero entropy values. As packed code is unpacked, written code causes the increase. Finally, it stops changing when unpacking is complete. Classifying in more detail, Type I packers could be categorised based on the pattern at the beginning part of changes. Type I-i (alter_exe, fsg, nSpack, UPX iT, and upxn) shows a continuous increase of the entropy in contrast to type I-ii (RLPack), an increase followed by the decrease in entropy.

Type II packers overwrite unpacked code onto existing code, which could be already used and are never going to be needed, or garbage values. Type II shows decreasing patterns of entropy values even though they can be divided into dramatic or gradual changes. As in the way type I packers work, it stops changing when unpacking is eventually completed. This type of packers can be also categorised further like type I. Type II-i (nPack) shows a gradually decreasing pattern in the very first part of unpacking, and then a fluctuation of entropy. Type II-iii (aspack, molebox) shows dramatically decreasing entropy values. Type II-ii (mpress, morphine) is in the middle between type II-i and II-iii, and depicts a gradual decrease of entropy to the end point of unpacking.

4.4 Cache

Execution-flow-changing instructions play a role of a decision point at which to measure entropy. However, it also degrades analysis performance in terms of time because some of those instructions could be used for iterations and branches. Hence, reducing unnecessary entropy measurement is important to improve the time performance; this can be achieved by reducing iterations.

As the purpose is not an optimization, but just a reduction, a simple mechanism, caching, is used in

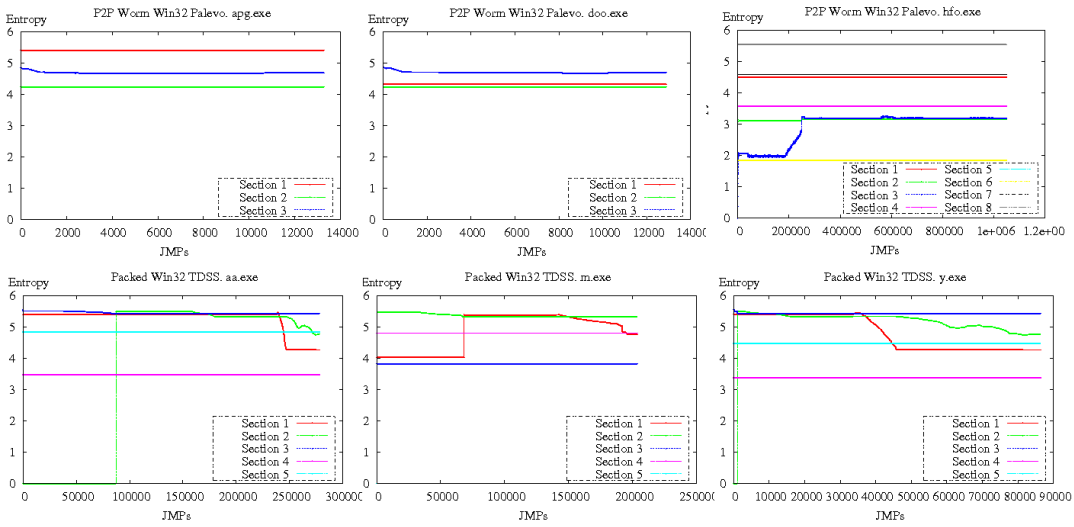

Figure 2. Analysis of packed malwares

our approach. To simplify caching, recent n JMP and JCC instructions are cached, where n is the maximum number of addresses that can be stored in the cache. Caching helps make analysis about twice as fast than before, as can be seen in Table 4.

Table 4. Analysis time in seconds vs. Cache size

n	0	5	10	15
freecell	392	337	360	331
calc	1296	974	899	687
mspaint	7149	6542	5625	4207
spider	17532	77423	12002	9703

4.5 Packed malwares in the wild

Another important point we should investigate is if it really works for packed malwares in the wild. We analyse unpacking process of packed malwares using our approach, and locate the candidate OEP from the analysis. Palevo and TDSS are chosen for packed malwares to be analysed in this experiment from VX Heavens [2]. Three variants for each packed malware are given. Variants of Palevo and TDSS are named as Palevo.apg, Palevo.doo, Palevo.hfo, TDSS.aa, TDSS.m, TDSS.y, respectively.

Experimental results are illustrated in Figure 2. Graphs of Palevo.apg and Palevo.doo looks similar. Both of them use the third section to unpack instructions. However, the entropy scores of the first sections are different from each other. Two variants could include packed instructions in the third section, and it also seems possible that Palevo.apg has packed instructions in the first section according to the entropy score. Palevo.hfo shows a totally different pattern from the former variants. It is quite obvious that the third section is initialized, and used to unpack instructions. Its 1st, 7th, and 8th sections are possible areas packed instructions could be placed. Palevo.apg, Palevo.doo, and Palevo.hfo can be categorised into II-i, II-i, and I-i, respectively.

A graph of TDSS.a shows that the second section is used to unpack not native instructions but packed instructions. After writing packed instructions in the second section, it is used again to unpack packed instructions with the first section at the same time. Nevertheless, we can conclude the first section is the one holding the real OEP because its entropy score is in a range of native instructions. TDSS.m writes packed instructions in the first section, and uses it again to unpack packed instructions. TDSS.y can be said similar to TDSS.aa in terms of its unpacking procedure. Initial unpacking packed instructions process comes earlier, and unpacking packed instructions after the initial unpacking takes longer time in TDSS.y. TDSS.aa,

TDSS.m, and TDSS.y can be categorised into II-iii, II-ii, and II-i, respectively.

These experimental results for these notorious packed malwares imply that the proposed unpacking mechanism is useful to analyse packed malwares. It is a meaningful result that the proposed mechanism is applicable to packed malwares.

5 Conclusion

Malware writers use packing as an obfuscation technique to hide potential signatures that exist in their malwares and to evade signature-based malware detection systems. In this paper, we propose a generic mechanism finding OEPs of packed executables. Experiments show it satisfies two necessary requirements (locating OEP and being generic) of unpacking mechanisms. We categorise packing techniques based on patterns of entropy changes. Moreover we show how this mechanism could be applied to the analysis of packed malware. Our mechanism has the following features.

6 Acknowledgments

This research was supported by the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2010-(C1090-1031-0005)), and this work was supported by the IT R&D program of MKE/KEIT. [KI001863, The Development of Active Detection and Response Technology against Botnet] Additionally, this research was sponsored in part by the MSRA(Microsoft Research Asia).

References

[1] Bochs: The open source ia-32 emulation project. `http://bochs.sourceforge.net`.

[2] VX heavens. `http://vx.netlux.org`.

[3] AHTeam. Quickunpack. `http://qunpack.ahteam.org`.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[5] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen. SplitScreen: Enabling efficient, distributed malware detection. In *Proc. 7th USENIX NSDI*, San Jose, CA, Apr. 2010.

[6] K. Coogan, S. Debray, T. Kaochar, and G. Townsend. Automatic static unpacking of malware binaries. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, pages 167–176, Washington, DC, USA, 2009. IEEE Computer Society.

[7] D. S. W. L. (DSWLab). Vmunpacker. `http://www.dswlab.com/d3.html`.

[8] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 98–115, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *WORM '07: Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 46–53, New York, NY, USA, 2007. ACM.

[10] R. Labs. Rl!depacker. `http://ap0x.jezgra.net`.

[11] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy*, 5(2):40–45, 2007.

[12] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *ACSAC '07: Proceedings of the 23rd Annual Computer Security Applications Conference*, pages 431–441, 2007.

[13] 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*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.

[14] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *ICISS '08: Proceedings of the 4th International Conference on Information Systems Security*, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] L. Sun, T. Ebringer, and S. Boztas. Hump-and-dump: efficient generic unpacking using an ordered address execution histogram. 2nd Int'l CARO Workshop, May 2008.

[16] W. Yan, Z. Zhang, and N. Ansari. Revealing packed malware. *IEEE Security and Privacy*, 6(5):65–69, 2008.

[17] R. W. Yeung. *A first course in Information Theory*. Kluwer Academic/Plenum Publishers, 2002.