LETTER

Executable Code Recognition in Network Flows Using Instruction Transition Probabilities

Ikkyun KIM[†a], **Koohong KANG**[††], **Yangseo CHOI**[†], **Daewon KIM**[†], **Jintae OH**[†], *Nonmembers*, **Jongsoo JANG**[†], *Member, and* **Kijun HAN**[†††], *Nonmember*

SUMMARY The ability to recognize quickly inside network flows to be executable is prerequisite for malware detection. For this purpose, we introduce an instruction transition probability matrix (ITPX) which is comprised of the IA-32 instruction sets and reveals the characteristics of executable code's instruction transition patterns. And then, we propose a simple algorithm to detect executable code inside network flows using a reference ITPX which is learned from the known Windows Portable Executable files. We have tested the algorithm with more than thousands of executable and non-executable codes. The results show that it is very promising enough to use in real world.

key words: *executable code, malware detection, IA-32 Instruction*

1. Introduction

As the "zero-day" worm attacks are becoming more sophisticated in spreading across Internet and their damage to our society is getting heavier, malware detection is one of the cutting edge research topics in computer security [1]. In the process to detect the malicious codes on the network, the starting phase is to recognize the presence of the executable code within IP packet's payload [2], [3]. In this letter, we propose a practical method to recognize the executable codes inside network flows or any types of files without the emulation of the execution level. Whilst the semantic–aware schemes explained in [3], [4] give a nearly perfect detection of executable codes, it might be not applicable to real-time systems. For this purpose, we introduce the instruction transition probability matrix (ITPX) characterizing the features of the instruction transition patterns of executable codes. We use the ITPX of typical executable codes as a profile to detect whether there are some executable codes. Andersson et al. [5] proposed a search algorithm to detect the executable code transmitted in buffer overflow attacks. However, the algorithm only identified the operation of the buffer overflows attack by printing out the sequence of system calls used in the exploit. Moreover, Chinchani and Berg [6] proposed a fast static analysis approach to detect exploit code inside network flows, where they relied on the control and data flow analysis at instruction level. Unfortunately, it still not fast enough to handle very large network traffic as mentioned in the paper. In this letter, we also rely on instruction patterns of network flows, but we use the ITPX of typical executable codes as a profile. Our approach does not have to match any signatures or to analysis control or data flows.

Manuscript received March 7, 2008.

[†]The authors are with Information Security Research Division, ETRI, Gajeong-dong, Yuseong-gu, Daejeon, 305–700, South Korea.

[††]The author is with the Dept. of Information and Communications Engineering, Seowon University 231, Mochung-Dong, Chongju, 361–742, South Korea.

[†††]The author is with the Dept. of Computer Engineering, Kyungpook National University 1370 Sankyuk-dong, Buk-gu Daegu 702–701, South Korea.

a) E-mail: ikkim21@etri.re.kr

DOI: 10.1093/ietisy/e91–d.7.2076

2. Discrete Markov Model & ITPX

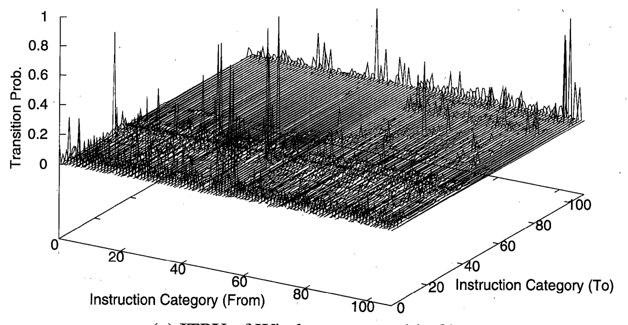
We introduce a discrete parameter Markov chain model to generalize the instruction transitions of executable codes. That is, we observe every instructions dissembled from the network flows and assume that the "future instructions" only depend on the "current instruction". This assumption is from that the atomic operations of machine codes would be independent on the past ones even if the contexts of high level-language programs written by C and C++ are a little dependent on the past ones.

First, we classify the several hundreds of IA-32 OP-codes into 109 instruction groups using the libdasm linear disassembler, which consist of arithmetic and logical instructions of the integer and floating point, privileged mode and NOP instructions, and so on. Now we drive an one-step homogeneous ITPX $P^{(1)}$ as follows,

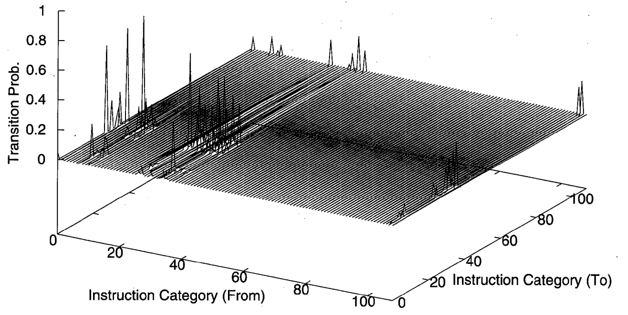
$$\mathbf{p}^{(1)} = \begin{pmatrix} p_{0,0} & \cdots & p_{0,108} \\ \vdots & \ddots & \vdots \\ p_{108,0} & \cdots & p_{108,108} \end{pmatrix}, \qquad (1)$$

where $p_{i,j} = P(X_n = j | X_{n-1} = i), i, j = 0, \ldots, 108, n > 0$ are the transition probabilities from instruction group i to j.

In order to get the reference ITPX of executable codes, we scrutinized the sequences of 230,000 instructions in the execution code area (.txt section) of 80 Windows PE (Portable Executable) files stored at windows/system32 folder in Windows System using the libdasm linear disassembler, and then we determined the average transition probabilities from instruction group i to j. From Fig. 1 (a) and (b), we note that the ITPX of executable codes has its own correlation characteristic of instruction transitions compared with the one of ordinary text files.



(a) ITPX of Windows executable files



(b) ITPX of ASCII text files

Fig. 1 The visualization of instruction transition probability matrix on the executable codes and non-executable data.

3. Basic Idea and Experimental Results

We define two new terminologies as follows,

- *Definition 1.* MDR (Minimum Decision Range): The number of minimum instructions required for the determination in which the instruction sequence of the executable code exists.
- *Definition 2.* EES (Expectation of Executable Sequence):

$$EES_n = \Pr\left(y_n, y_{n+1}, \cdots, y_{n+MDR-1} | ITPX\right). \qquad (2)$$

where n is from 1 to $(length(payload) - MDR)$ and y_i's are the observed ith instruction transitions of nth chunk of an IP packet. From the assumption, individual observation y_i's are statistically independent of one another, and we estimate EES by the log function for computational convenience [7] as follows,

$$log\_EES_n = -\sum_{k=n}^{n+MDR-1} \ln p_{i,j}^k, \qquad (3)$$

where $p_{i,j}^k$ is the corresponding $p_{i,j}$ in Eq. (1) of the reference ITPX when the k-th instruction is i and the $(k+1)$-th instruction is j. We note that we choose $MDR = 50$ according to the general size of shellcodes of which very short execuuable codes.

Even if many threshold-based anomaly intrusion detections have the trouble do deal with their high false alarm rate, we also use this well-studied method in this letter.

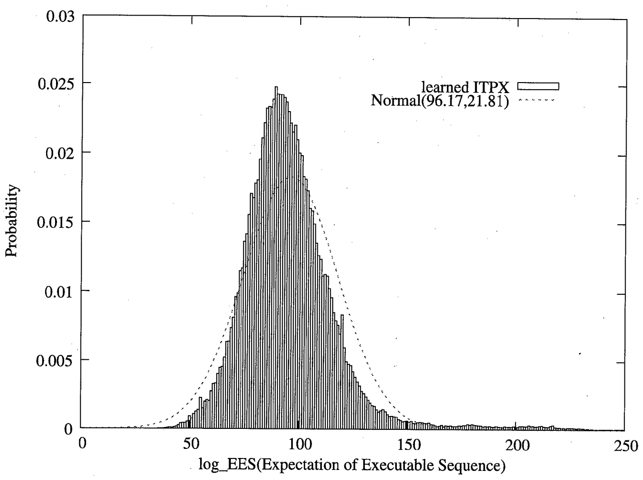


Fig. 2 Histogram of the Expectation of Executable Sequence (log_EES) for the learning Windows PE files.

Table 1 False negative and false positive for each file type when the threshold is based on 90% reliability.

File Type	False ± (%)	Number of Total Chunks	Number of Exe. Chunks
Windows PE	−8.12652%	268073	246288
Hangul Txt	+0.15011%	303104	455
ASCII Txt	+0%	61440	0
Windows DOC	+4.75188%	319242	15170
Windows PPT	+0.40478%	392314	1588
Windows XLS	+16.82454%	209557	35257
MP3	+0%	409600	0
JPEG	+0.03513%	352957	124
PDF	+0.04973%	406185	202

Table 2 False negative and false positive for each file type when the threshold is based on 95% reliability.

File Type	False ± (%)	Number of Total Chunks	Number of Exe. Chunks
Windows PE	−5.55297%	268073	253187
Hangul Txt	+0.25437%	303104	771
ASCII Txt	+0%	61440	0
Windows DOC	+7.49494%	319242	23927
Windows PPT	+0.83657%	392314	3282
Windows XLS	+23.03908%	209557	48280
MP3	+0%	409600	0
JPEG	+0.06176%	352957	218
PDF	+0.06820%	406185	277

Figure 2 shows the histogram of log_EESs of our learning Windows PE files, from which we obtain mean and variance and note that $Nor(m = 96.17, \sigma^2 = 21.81)$ matches well the histogram. In order to detect executable code, we adopt an acceptance region for a right-hand one-sided test of normal distribution. To evaluate the proposed approach, we tested nine different types of files, such as general ASCII text, Hangul (Korean character) text, JPG image, MP3 audio, PDF, Windows DOC, XLS, and PPT files. As experiment results, Tables 1 and 2 present the false positives and negatives for 90% and 95% confidence intervals of $Nor(96.17, 21.81)$, respectively. Unfortunately, we notice

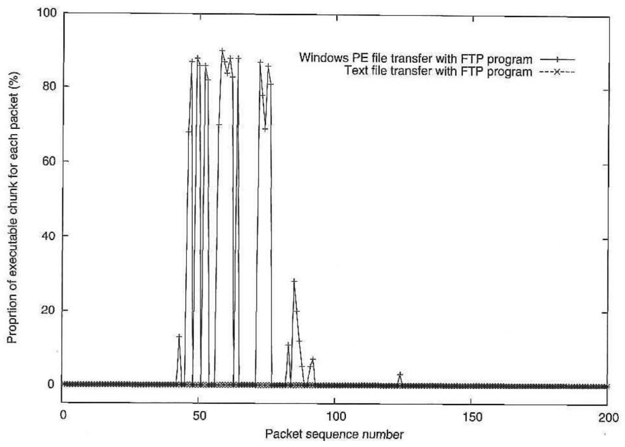


Fig. 3 The proportion of executable chunks in the network packets of FTP connections.

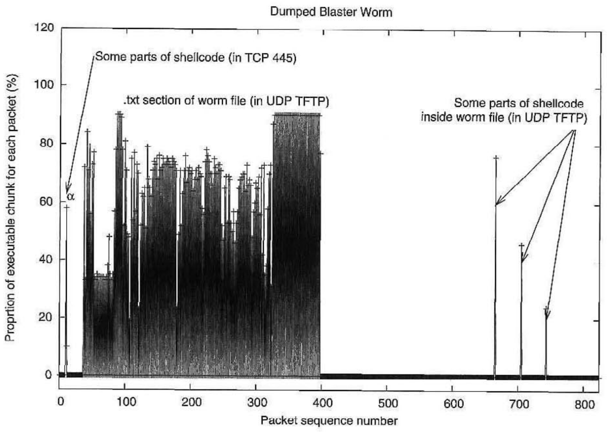


Fig. 4 The proportion of executable chunks in the network packets of the Blater worm. ((α) indicates recognition of the shellcode in the exploit.)

that the false positive of Windows XLS files is very high because the instruction transition patterns of the files might be very similar to executable codes.

4. Packet-Based Experiment

The final goal of this letter is to recognize any executable parts in the network packets. So we tried two types of experiments with dump files captured bi-directional network packets. The first experiment used two FTP traffic dump files in which one delivered a Windows executable file and another delivered a text file, respectively. From Fig. 3, we note that there is no executable chunk in the FTP stream for text file, while we can see about 80% the executable chunks in the packets of the sequence number between 48 and 80 of the FTP stream delivering Windows PE file. We also verified that the packet sequence numbers of executable part of the graph in Fig. 3 are identical to the .txt section of PE executable file. In the second experiment, we used a real worm dump file – the Blaster worm – which exploits the RPC DCOM vulnerability in Windows OS. These captured packets are composed of bidirectional 824 packets, where the first shellcodes of the exploit are delivered to the target system on TCP 445 port, and then the worm file – blaster.exe – is delivered to victim system using a UDP TFTP service like a typical propagation scenario of a malware. As shown in Fig. 4, we can confirm that our proposed mechanism can recognize not only the executable part of the PE file but also the executable part of shellcode of the exploits in the network packet.

5. Conclusion

We have presented a fast method to detect executable codes in network traffic that can be implemented with simple and practical. The proposed novel ITPX-based approach shows very promising results for the detecting of the most current executable codes. Moreover this method can be applied to malware detection system without network session tracking and any packet reassembling.

References

[1] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," OSDI, pp.45–60, 2004.

[2] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," SSYM'04: Proc. 13th Conference on USENIX Security Symposium, pp.255–270, USENIX Association, Berkeley, CA, USA, 2004.

[3] X. Wang, C.C. Pan, P. Liu, and S. Zhu, "Sigfree: A signature-free buffer overflow attack blocker," USENIX-SS'06: Proc. 15th Conference on USENIX Security Symposium, pp.225–240, USENIX Association, Berkeley, CA, USA, 2006.

[4] M. Christodorescu, S. Jha, S.A. Seshia, D.X. Song, and R.E. Bryant, "Semantics-aware malware detection," IEEE Symposium on Security and Privacy, pp.32–46, 2005.

[5] S. Andersson, A. Clark, and G. Mohay, "Network-based buffer overflow detection by exploit code analysis," Information Technology Security Conference 2007, pp.39–53, 2007.

[6] R. Chinchani and E. van den Berg, "A fast static analysis approach to detect exploit code inside network flows," 8th International Symposium, RAID 2005, pp.284–301, Seattle, WA, USA, 2005.

[7] I.J. Myung, "Tutotial on maximum likelihood estimation," J. Math. Psychol., vol.47, pp.90–100, 2003.