10-2014

Instructions-Based Detection of Sophisticated Obfuscation and Packing

Moustafa Saleh
University of Texas at San Antonio

Edward Paul Ratazzi
Syracuse University, epratazz@syr.edu

Shouhuai Xu
University of Texas at San Antonio

Recommended Citation

Saleh, Moustafa; Ratazzi, E.Paul; Xu, Shouhuai, "Instructions-Based Detection of Sophisticated Obfuscation and Packing," Military Communications Conference (MILCOM), 2014 IEEE , vol., no., pp.1,6, 6-8 Oct. 2014 doi: 10.1109/MILCOM.2014.9 keywords: {Electronic mail;Encryption;Entropy;Feature extraction;Malware;Reverse engineering}, URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6956729&isnumber=6956719

Instructions-based Detection of Sophisticated Obfuscation and Packing

Moustafa Saleh
Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas
Email: msaleh83@gmail.com

E. Paul Ratazzi
Information Directorate
Air Force Research Laboratory
Rome, New York
Email: edward.ratazzi@us.af.mil

Shouhuai Xu
Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas
Email: shxu@cs.utsa.edu

***Abstract*—Every day thousands of malware are released online. The vast majority of these malware employ some kind of obfuscation ranging from simple XOR encryption, to more sophisticated anti-analysis, packing and encryption techniques. Dynamic analysis methods can unpack the file and reveal its hidden code. However, these methods are very time consuming when compared to static analysis. Moreover, considering the large amount of new malware being produced daily, it is not practical to solely depend on dynamic analysis methods. Therefore, finding an effective way to filter the samples and delegate only obfuscated and suspicious ones to more rigorous tests would significantly improve the overall scanning process. Current techniques of identifying obfuscation rely mainly on signatures of known packers, file entropy score, or anomalies in file header. However, these features are not only easily bypass-able, but also do not cover all types of obfuscation.**

In this paper, we introduce a novel approach to identify obfuscated files based on anomalies in their instructions-based characteristics. We detect the presence of interleaving instructions which are the result of the opaque predicate anti-disassembly trick, and present distinguishing statistical properties based on the opcodes and control flow graphs of obfuscated files. Our detection system combines these features with other file structural features and leads to a very good result of detecting obfuscated malware.

I. Introduction

Zero-day malware detection is a persistent problem. Hundreds of thousands of new malware are produced and published on the Internet daily. Although conventional signature-based techniques are still widely relied upon, they are only useful for known malware. Many research efforts have aimed at helping flag and detect unknown suspicious and malicious files. All of these techniques can be categorized into sandbox analysis, heuristic static analysis or code emulation. Among the three, heuristic static analysis is the fastest, yet the weakest against obfuscation techniques. Code obfuscation includes packing, protecting, encrypting or inserting anti-disassembly tricks, and is used to hinder the process of reverse engineering and code analysis. About 80% to 90% of malware use some kind of packing techniques [1] and around 50% of new malware are simply packed versions of older known malware according to a 2006 article [2], and we believe it is more than that by now. While it is very common for malware to use code obfuscation, benign executable files rarely employ such techniques. Thus, it has become a common practice to flag an obfuscated file as suspicious and then examine it with more costly analysis to determine if it is malicious or not.

Most current work of detecting obfuscated files is based on executable file structure characteristics as we will show in Section II. Many public packers, indeed, exhibit identifiable changes in the packed PE file. However, this is not always the case with custom packers and self-encrypting malware. Moreover, packing is not the only obfuscation technique used by malware writers. Malware can use anti-analysis tricks that hinder the disassembly or analysis process. Such tricks can leave absolutely no trace in the header as it is based on obfuscating the instructions sequence and the execution flow of the program. Other methods depend on detecting the signature of known packers in the file. The drawback of this method is obvious as it does not work with unknown and custom packers and cryptors. It also fails if the signature is slightly modified. Calculating the entropy score of the file is another method of identifying packed and encrypted files. This method could be effective against encryption or packing obfuscation, but is ineffective against anti-disassembly tricks. In addition, the entropy score of a file can be reduced to achieve low entropy similar to those normal program.

In this paper, we present a new method for detecting obfuscated programs. We build a recursive traversal disassembler that extracts the control flow graph of binary files. This allows us to detect the presence of interleaving instructions, which is typically an indication of the opaque predicate anti-disassembly trick. Our detection system uses some novel features based on referenced instructions and the extracted control flow graph that clearly distinguishes between obfuscated and normal files. When these are combined with a few features based on file structure, we achieve a very high detection rate of obfuscated files.

More specifically, our contributions of the paper are:

- We leverage the fact that some advanced obfuscated malware use opaque predicate techniques to hinder the process of disassembly, and describe a technique that turns this strength into a weakness by detecting its presence and flagging the file as suspicious (Section III).
- We identify distinguishing features between obfuscated and non-obfuscated files by studying their control flow graphs. These features help detect obfuscated

Approved for Public Release; Distribution Unlimited: 88ABW-2014-1216 Dated 20140326

files while avoiding drawbacks of the other methods that rely on file structure.

- We achieve a fast scanning speed of 12 ms per file on average, despite the fact that our method encompasses disassembly, control flow graph creation, feature extraction, and file structure examination.

The rest of the paper is structured as follows: Section II briefly review the related work. Section III discusses the opaque predicate technique that can hinder the process of disassembly. Section IV reveals the statistical characteristics we identified for distinguishing obfuscated files from non-obfuscated ones. Section V describes our experiments and results. Section VI discusses the results and potential limitations. Section VII concludes the paper.

II. Related work

A. Entropy-based detection

Lyda and Hamrock presented the idea of using entropy to find encrypted and packed files [1]. The method became widely used as it is efficient and easy to implement. However, some non-packed files could have high entropy values and thus lead to false-positives. For example, the `ahui.exe` and `dfrgntfs.exe` files have an entropy of 6.51 and 6.59 respectively for their `.text` section [3], [4]. (These two example files exist in Windows XP 32-bit and are detected by our system as non-packed.) In addition to entropy-based evasion techniques mentioned in [5], simple byte-level XOR encryption can bypass the entropy detection as well.

B. Signature-based detection

A popular tool to find packed files is PEiD, which uses around 470 packer and crypter signatures [6]. A drawback of this tool is that it can identify only known packers, while sophisticated malware use custom packing or crypting routines. Moreover, even if a known packer is used, the malware writer can change a single byte of the packer signature to avoid being detected as packed. In addition, the tool is known for its high error rate [7].

C. File header anomaly detection

Other research such as [7]–[10] use the PE header and structure information to detect packed files. These techniques can get good results only when the packer changes the PE header in a noticeable way.

Besides the shortcomings of every technique, notably, none of the aforementioned techniques can statically detect the presence of anti-disassembly tricks or other forms of control flow obfuscation, yet these are now commonly used by a wide range of advanced malware. In addition, our proposed system does not depend on a coarse-grained entropy score of the file or section, signature of packers, or file header features. Thus, it is able to overcome these shortcomings.

III. Anti-disassembly tricks used by malware writers

Malware writers use a variety of anti-analysis tricks to protect against all kinds of analyses. One class of these is anti-disassembly tricks. Anti-disassembly tricks hinder the process of disassembly and hence reduce the effectiveness of static analysis-based detection of malware. One of the most common techniques is to use an Opaque Predicate. Although there are legitimate reasons for including opaque predicate tricks, such as watermarking [11] and to hinder reverse engineering, they are commonly used in malware to prevent analysis.

Opaque predicate tricks [12] insert conditional statements (usually control flow instructions) whose outcome is constant and known to the malware author, but not clear in static analysis. Thus, a disassembler will follow both directions of the control flow instruction, one of which leads to the wrong disassembly and affects the resulting control flow graph. As an example, listing 1 shows an opaque predicate trick inserted on lines 6 and 7 of the code snippet. Since the compare on line 6 will always evaluate to true, the `fake` branch will never be taken at runtime. However, to a disassembler, this fact is not apparent and it will evaluate both paths.

In this example, the disassembler will follow the target of the `jne` instruction on line 7, which leads to a byte of data on line 11. The disassembly will continue starting with this byte, `0F`, resulting in decoding an instruction with opcode **`0F`**`9090 8BC9BA44`. This incorrect instruction is `SETO BYTE PTR DS:[EAX+44BAC98B]` as shown in figures 1 and 2 for two common disassemblers, IDA Pro and OllyDbg, respectively.

We developed a recursive traversal disassembler that is able to detect interleaving code and flag the corresponding basic block as problematic, so an analyst could easily know where to find these tricks. Figure 3 shows a portion of the control flow graph output from our disassembler for this example. Two blocks are shown in red to indicate that they are interleaving and only one of them is correct.

```
1         xor eax, eax
2         nop
3         nop
4   L1:
5         push eax
6         cmp eax, eax
7         jne fake
8         add ecx, 333h
9         jmp skip
10  fake:
11        DB 0Fh
12  skip:
13        nop
14        nop
15        mov ecx, ecx
16        mov edx, 444h
17        push offset ProcName
18        push eax
19        call GetProcAddress
```

Listing 1. Opaque predicate trick snippet.

IV. Instructions-based detection: Turning attackers' strength into weakness

Due to obfuscation techniques such as opaque predicate, the control flow graph (CFG) and the sequence of instructions

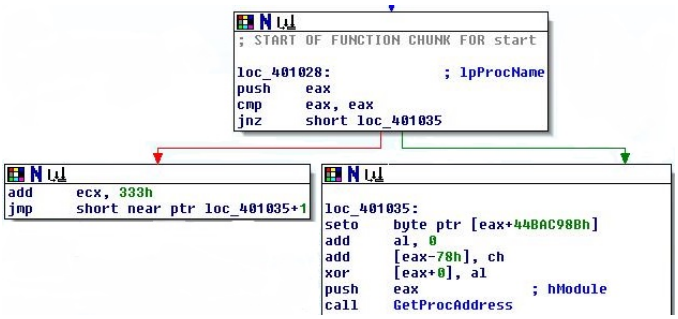

Fig. 1. Portion of IDA Pro graph for the example in listing 1.

```
33C0          XOR EAX,EAX
90            NOP
90            NOP
50            PUSH EAX
3BC0          CMP EAX,EAX
75 08         JNE SHORT 00401035
81C1 3303000( ADD ECX,333
EB 01         JMP SHORT 00401036
0F9090 8BC9B( SETO BYTE PTR DS:[EAX+44BAC98B]
04 00         ADD AL,0
0068 88       ADD BYTE PTR DS:[EAX-78],CH
3040 00       XOR BYTE PTR DS:[EAX],AL
50            PUSH EAX
E8 32000000   CALL <JMP.&kernel32.GetProcAddress>
```

Fig. 2. Portion of OllyDbg disassembly for the example in listing 1.

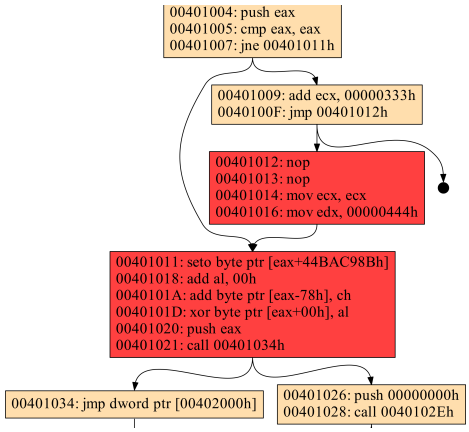

Fig. 3. Part of our disassembler's output for the example in listing 1.

extracted from obfuscated programs are usually convoluted, resulting in different sizes of basic blocks compared to a normal program, a greater percentage of sink vertices of all basic blocks, and other telltale features. In the following subsections, we introduce interesting features that can effectively identify an abnormal control flow graph and sequence of instructions. We show how each of these features differs in case of obfuscated and clean files. The illustrative statistical distributions presented in this section are from representative file sets that are also used in the experiments of Section V.

A. Percentage of sink vertices in CFG

The CFG of a given program is a digraph where each vertex represents a basic block. Sink vertices in this context refer to those vertices with zero out-degree. Sink vertices are usually the exit point of the program, and since a typical program has few exit points in the code, the number of sink vertices is very small compared to other vertices. Obfuscated malware that employ anti-analysis techniques lead to inaccurate static disassembly of the file. Thus, the ratio of sink vertices to the

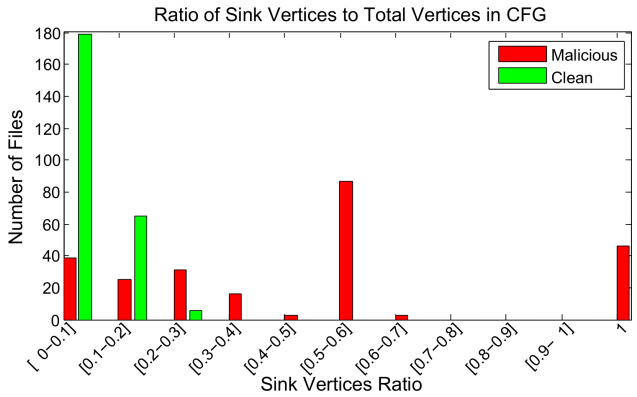

Fig. 4. Distribution of sink vertices to all vertices ratio for malicious and clean file sets.

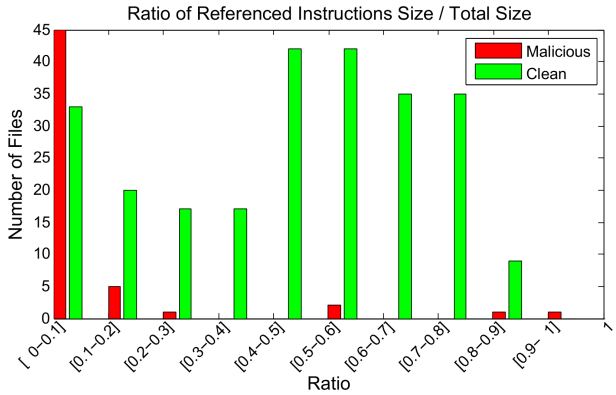

Fig. 5. Distribution of referenced instruction size to section size ratio for malicious and clean file sets.

total vertices becomes different from a normal file. Figure 4 compares the ratio of sink vertices in both clean non-obfuscated and malicious obfuscated files, respectively.

B. Percentage of the size of referenced instructions to the entire size of the section

Due to code obfuscation, encryption or packing, the size of referenced instructions compared the size of the section is relatively smaller than that of clean files. The decryption or unpacking routine that exists in the same section of the encrypted or packed code occupies a much smaller size than the actual payload of the file. This fact represents a distinguishable feature between packed and non-packed files. Figure 5 shows these values in different files in clean and malicious dataset.

C. Average number of instructions in basic blocks

After constructing the control flow graph of the program, each basic block will represent a set of instructions with a single entry and a single exit instruction. The exit instruction, in most cases, is a control transfer that affects the flow of the execution. If the disassembly was wrongly redirected into disassembling packed or encrypted data due to anti-disassembly tricks, false instructions will be decoded, which will result in different characteristics of a typical control flow graph of a normal application. One of these characteristics is the average size of instructions in basic block. Figure 6

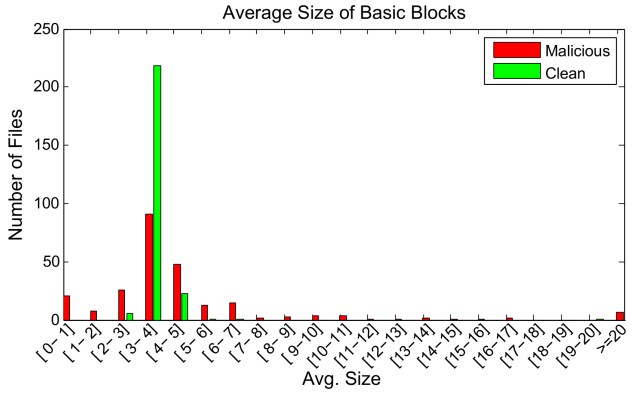

Fig. 6. Distribution of basic block average size for malicious and clean file sets.

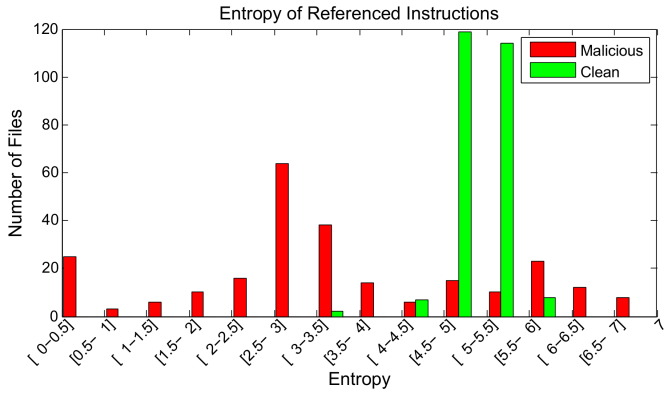

Fig. 7. Distribution of file entropy for referenced instructions only.

shows the average number of instructions in a basic block in malicious and benign dataset respectively.

D. Entropy of referenced instructions opcodes

As discussed in Section II, entropy is a measure of randomness which can sometimes be used to detect packed files. Almost all techniques that use entropy to detect packed files calculate the entropy of the entire file, a section, or the file header. However, as explained earlier, an encrypted or packed data can still exhibit low entropy if an entropy reduction method is used. In addition, a normal program could contain data of high entropy within the code. In this case, the entropy of this data will be incorporated in the total entropy. This is a major source of false positives.

On the other hand, if the program employs an anti-disassembly technique that is able to deceive the disassembler into decoding false instructions, the resulting opcodes of the false instructions will have different statistical distribution than those of real ones. If the entropy of only referenced instructions is computed, we would have a more specific and accurate use of the entropy metric. Thus, even if a normal program contains data of high entropy within the code, the entropy of this data will not be incorporated in the total entropy calculation, because the flow of execution of a normal program ensures jumping over this data during execution. Figure 7 shows the distribution of file entropy when only referenced instructions are considered, for both non-obfuscated clean and obfuscated malicious files, respectively.

E. Existence of interleaving instructions

In our system we flag any file with interleaving instructions as obfuscated, since unobfuscated applications do not usually intentionally employ opaque predicate. Existence of such interleaving instructions is a clear flag of obfuscation, unless it is a bug or an artifact in an unobfuscated program. In Section V, we show how our system found an artifact in non-obfuscated Windows files when interleaving instructions were detected in them.

F. Existence of unknown opcodes

If an unknown opcode is encountered while disassembling the file, it means that the disassembly process is diverted from the normal execution path, and the file is flagged as obfuscated. It's worth mentioning that our disassembler does not make assumptions about indirect addressing. Assumption in this case would be uncertain and following uncertain paths would definitely lead to high false-positive rate. Therefore, since the disassembler covers almost all opcodes of the x86 and IA-64 architecture, even some of undocumented instructions, finding an unknown opcode in the extracted instructions would be a strong evidence of obfuscation.

V. Experiment and Evaluation of our Detection Method

We ran our proposed system on two sets of files. The first set consists of 250 clean files taken from a clean Windows XP 32-bit machine, all of them non-packed. The second set consists of 250 malicious packed files, packed by both commercial and custom packers. The ranges of values for the features introduced in Section IV are shown in Tables I and II for both clean and malicious sets, respectively. Based on the result in Table I, we established six criteria to test if the file is packed:

1) The sink vertices ratio lies outside the range of non-packed files.
2) The referenced instructions ratio lies outside the range of non-packed files.
3) Average number of instructions in basic block lies outside the range of non-packed files.
4) Entropy of referenced instructions lies outside the range of non-packed files.
5) The code has one or more anti-disassembly tricks.
6) The code references an unknown opcode.

TABLE I. Value ranges of statistical features in Windows XP clean file set.

Property	Min	Max
Sink vertices ratio	0.0260047	0.254545
Referenced instructions ratio	0.000544839	0.884181
Average number of instructions in basic block	2.3125	19.6357
Entropy of referenced instructions	3.38	5.61
Files with a referenced unknown opcode	0	
Files with anti-disassembly trick	0	

TABLE II. VALUE RANGES OF STATISTICAL FEATURES IN MALICIOUS FILE SET.

Property	Min	Max
Sink vertices ratio	0.0	1.0
Referenced instructions ratio	1.89169 x 10^{-6}	0.92139
Average number of instructions in basic block	1	2142
Entropy of referenced instructions	0	6.81
Number of referenced unknown opcode	0	5
Files with anti-disassembly trick	63	

Based on these criteria, we achieved 100% correct detection of the clean files as non-packed and 98.8% of the malicious files as packed or obfuscated. This result is shown in Table III. In addition, we found that by adding an extra criterion by measuring the entropy of the entry point section and marking files with entropy greater than 6.5 as packed, we could achieve 100% detection of malicious files as packed, i.e., 0% false negatives. However, this introduced a 14.8% false positive rate as some of the clean non-obfuscated files were flagged as obfuscated.

Although structural features of the files were not our main concern in this paper, we added a few checks on the file structure which further improved the result. The following list of structural features were used to help identify obfuscated files:

1) The entry point is in file header before any section.
2) There is no `.text` or `CODE` section in the file.
3) The entry point is in the last section while it is neither `.text` nor `CODE` section.
4) `SizeOfRawData` $= 0$ and `VirtualSize` > 0 for some sections.
5) Sum of `SizeOfRawData` field of all sections is greater than the file size.
6) Two or more sections overlap.
7) The file has no imports at all or the import table is corrupted.

The scanning result when using each detection features is shown in Table III where FN and FP refer to false negative and false positive rates, respectively.

We collected 423 clean non-obfuscated files from a clean Windows 7 32-bit Home Basic Edition and scanned them using only our instructions-based criteria. We used the ranges mentioned in Table I as detection conditions. There was 7 out of the 423 files (1.64%) detected as obfuscated. Since we know that those files are not obfuscated, we considered, at the beginning, the result as a false-positive. However, after manually reverse engineering the files, it turned out there is an artifact of some incomplete code generation [13] in six of them. The files have overlapped instructions that, if executed, would likely crash the programs under certain conditions. Although these conditions were not clear to us, based on the instructions' location in the file, we feel that is unlikely that the execution of these faulty instructions would ever take place [13]. The seventh file is `sppsvc.exe` that has `Referenced Instruction Ratio` $= 0.000273778$, which is less than the minimum boundary set in Table I. Therefore, since the first six files contain an artifact, we could safely exclude them from the set and consider that there was no false-positive in our results except that corresponding to `sppsvc.exe`. Finally, we note that when we used entropy for detection, it led to the worst result as some non-obfuscated files show high entropy in the code section. The full results of this analysis of the file sets are shown in Table III.

TABLE III. FILE SET ANALYSIS RESULTS.

	FN	FP	Correctly detected	Percentage %
Instructions-based features only				
Clean files, WinXP	0%	0%	250 / 250	100%
Clean files, Win7	0%	0.1%	416 / 417	99.9%
Malicious files	1.2%	0%	247 / 250	98.8%
Instructions-based features with checking entropy of entry point section				
Clean files, WinXP	0%	14.8%	213 / 250	85.2%
Clean files, Win7	0%	13.2%	362 / 417	86.8%
Malicious files	0%	0%	250 / 250	100%
Structural features only				
Clean files, WinXP	0%	0%	250 / 250	100%
Clean files, Win7	0%	0%	417 / 417	100%
Malicious files	36.8%	0%	158 / 250	63.2%
Instructions-based with structural features				
Clean files, WinXP	0%	0%	250 / 250	**100%**
Clean files, Win7	0%	0.1%	416 / 417	**99.9%**
Malicious files	0%	0%	250 / 250	**100%**

We ran another test on a larger set of 10,171 malicious files. The set is a collection of live malware given to us by a security firm. Unfortunately, we have not been given details about the set in terms of packing/obfuscation. Although we admit that scanning result of this set is not a concrete measure of the effectiveness of the system since we cannot give a confirmed value of false-positive or false-negative, we opted to show the result for the sake of illustration. Table IV shows the value ranges of each criteria, while Table V shows the result of scanning the large malware set.

For all of our experiments, we observed that the system was able to process files at an average rate of 12 ms each.

TABLE IV. VALUE RANGES OF STATISTICAL FEATURES IN LARGE SET OF MALICIOUS FILES.

Property	Min	Max
Sink vertices ratio	0.0	1.0
Referenced instructions ratio	0.62 x 10^{-6}	1.0
Average number of instructions in basic block	1	69600
Entropy of referenced instructions	0	7.23
Number of referenced unknown opcode	0	163
Files with anti-disassembly trick	1835	

TABLE V. ANALYSIS RESULTS FROM THE LARGE SET OF MALWARE.

	Detected as packed	Percentage %
Instructions-based features only	9982 / 10171	98.1418%
Instructions-based with structural features	10161 / 10171	**99.9%**

VI. Discussion and limitations

We can summarize the features used in our system into two categories. Instructions-based statistical features, and structural features. The file structure features have the same advantages and limitations of the previous research discussed in Section II. The major contribution of the paper are the instructions-based method of detection.

All the features mentioned in Section IV except the one in Subsection IV-B are useful metrics when the file under consideration has features to intentionally deceive the disassembler into decoding wrong execution paths. This is due to existence of anti-disassembly tricks or other control flow obscuring techniques.

On the other hand, if the file is packed or encrypted with no control flow obfuscation, the feature discussed in Subsection IV-B (`Referenced Instruction Ratio`) comes into play. It can detect that just a small portion of the section is executed, which is typically the case when a small routine is responsible for unpacking or decrypting the relatively large remainder of the file. However, the limitation in this case is when this small routine exists in a separate section from the code to be unpacked. In this case, the section containing the unpacking routine would contain just the referenced instructions of the unpacking routine, and thus its ratio will be high. Hence, our detection would likely be evaded if a file is packed such that the unpacked routine and the packed code exist in two different sections, the file does not affect the header in a distinguishable way, and does not have anti-disassembly tricks.

Finally, the proposed system cannot yet analyze .NET and Java files because these are represented by an intermediate language which needs other methods of disassembly. The system was developed in C++ and it uses the BeaEngine library [14]. The experiment was conducted under Windows 7 64-bit on a notebook with Intel Core i5 processor and 8GB of RAM. The average execution time was observed to be around 12 ms per file.

VII. Conclusion

Due to the high number of malware being produced every day, the need for a fast and efficient system detection persists. If there is an efficient, fast way to detect the presence of obfuscation in a sample and then move it to a more rigorous test, this would reduce some of the burden on the more costly methods and help keep up with the big number of samples.

This paper presents a generic heuristic method to detect obfuscation based on both the structural and instructions-based features of the file. We built a complete recursive traversal disassembler for x86 and IA-64 binary files. We were able to detect the instructions overlapping trick and presence of unknown opcodes, which are mainly symptoms of opaque predicate or a bug in the code. In addition, a number of statistical features based on the control flow graph and the instructions that help distinguish malicious and benign files have been presented. When measuring those features combined with structural features of a sample, we achieve very high detection result of obfuscated files with a very fast scanning time of 12 ms on average per file.

A key advantage of our method is that it is not limited to a certain type of packers or a specific obfuscation technique. In our future work, we plan to add more instructions-based features and incorporate machine learning techniques to classify different packers. We believe that if these future goals are accomplished and the limitations mentioned in VI are overcome, they would lead to more accurate results with less margin of error.

Acknowledgment

We thank Peter Ferrie, principal anti-virus researcher at Microsoft, for answering our questions as well as his comments and feedback on our Windows files analysis. We thank Xabier Ugarte-Pedrero, a security researcher, for his explanations and informative discussion of his paper [5]. We thank Qingji Zheng for useful discussion.

This paper was partly supported by NSF under Grant No. 1111925 and ARO under Grant No. W911NF-12-1-0286. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the position of the NSF, the US Army, or the US Air Force.

References

[1] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *Security Privacy, IEEE*, vol. 5, no. 2, pp. 40–45, 2007.

[2] A. Stepan, "Improving proactive detection of packed malware," *Virus Bulletin*, pp. 11–13, March 2006.

[3] VirusTotal.com, "`ahui.exe`." http://goo.gl/kbbJKi. Accessed: Feb. 7th, 2014.

[4] VirusTotal.com, "`edfrgntfs.exe`." http://goo.gl/XCqUcF. Accessed: Feb. 7th, 2014.

[5] X. Ugarte-Pedrero, I. Santos, B. Sanz, C. Laorden, and P. Bringas, "Countering entropy measure attacks on packed software detection," in *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*, pp. 164–168, 2012.

[6] aldeid.com, "PEiD." http://www.aldeid.com/wiki/PEiD. Accessed: Feb. 8th, 2014.

[7] M. Shafiq, S. Tabish, and M. Farooq, "PE-probe: leveraging packer detection and structural information to detect malicious portable executables," in *Proceedings of the Virus Bulletin Conference (VB)*, pp. 29–33, 2009.

[8] R. Perdisci, A. Lanzi, and W. Lee, "Classification of packed executables for accurate computer virus detection," *Pattern Recogn. Lett.*, vol. 29, pp. 1941–1946, Oct. 2008.

[9] S. Treadwell and M. Zhou, "A heuristic approach for detection of obfuscated malware," in *Intelligence and Security Informatics, 2009. ISI '09. IEEE International Conference on*, pp. 291–299, June 2009.

[10] I. Santos, X. Ugarte-Pedrero, B. Sanz, C. Laorden, and P. G. Bringas, "Collective classification for packed executable identification," in *Proceedings of the 8th Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference*, CEAS '11, (New York, NY, USA), pp. 23–30, ACM, 2011.

[11] G. Myles and C. Collberg, "Software watermarking via opaque predicates: Implementation, analysis, and attacks," *Electronic Commerce Research*, vol. 6, no. 2, pp. 155–171, 2006.

[12] E. Eilam, *Reversing: Secrets of Reverse Engineering*. Wiley & Sons, 2008.

[13] P. Ferrie, "Principal anti-virus researcher at Microsoft." Personal Communication. Jan. 22nd, 2014.

[14] BeaEngine, "BeaEngine." http://www.beaengine.org/. Accessed: Apr. 3rd, 2014.