A Control Flow Graph-based Signature for Packer Identification

Moustafa Saleh
Threat Intelligence Center
Microsoft
Redmond, Washington
Email: mosale@microsoft.com

E. Paul Ratazzi
Cyber Assurance Branch
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*—The large number of malicious files that are produced daily outpaces the current capacity of malware analysis and detection. For example, Intel Security Labs reported that during the second quarter of 2016, their system found more than 40M of new malware [1]. The damage of malware attacks is also increasingly devastating, as witnessed by the recent Cryptowall malware that has reportedly generated more than $325M in ransom payments to its perpetrators [2]. In terms of defense, it has been widely accepted that the traditional approach based on byte-string signatures is increasingly ineffective, especially for new malware samples and sophisticated variants of existing ones. New techniques are therefore needed for effective defense against malware. Motivated by this problem, the paper investigates a new defense technique against malware.**

The technique presented in this paper is utilized for automatic identification of malware packers that are used to obfuscate malware programs. Signatures of malware packers and obfuscators are extracted from the CFGs of malware samples. Unlike conventional byte signatures that can be evaded by simply modifying one or multiple bytes in malware samples, these signatures are more difficult to evade. For example, CFG-based signatures are shown to be resilient against instruction modifications and shuffling, as a single signature is sufficient for detecting mildly different versions of the same malware. Last but not least, the process for extracting CFG-based signatures is also made automatic.

***Index Terms*—packer identification, malware detection**

I. Introduction

Zero-day malware detection is a persistent problem. Hundreds of thousands of new malicious programs 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 three types: 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 [3], [4] and around 50% of new malware are simply packed versions of older known malware [4], [5]. 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.

In this paper, we present a new method for packer or obfuscator detection and identification that builds upon our prior work on an instructions-based technique [6]. The work builds a recursive traversal disassembler that extracts the control flow graph of binary files, and then computes various statistical features of this graph to distinguish between obfuscated and normal files. Once the file is confirmed to be obfuscated, the new work constructs a compact signature of the control flow graph at the entry point which is resilient to dummy instructions insertion and a number of graph manipulation methods. This signature can be compared to a set of signatures in the database to determine the packer. If the packer is new or unknown, the signature can be automatically constructed from a set of packed files and used to update the database with no human intervention.

More specifically, the contributions of the paper are as follows:

- We introduce a compact signature for the control flow graph that can be used to identify the obfuscator or packer used in a file.
- The process of CFG signature construction is done automatically with no human intervention, so the system can store and be updated with new signatures to detect different types of obfuscators or packers.
- The signature is resilient to some instructions modifications and shuffling, which makes a single signature efficient against mildly different versions of the same code.
- We achieve a fast scanning speed of 0.5 ms per file on average, given the fact that our method encompasses disassembly, control flow graph extraction, signature creation and matching.
- There is a strong potential that the same technique can be used to detect malware variants.

Approved for Public Release; Distribution Unlimited: 88ABW-2016-4792, 28 Sep 2016.

978-1-5386-0595-0/17/$31.00 ©2017 IEEE

II. Related Work

Most of the current work of detecting obfuscated files is based on executable file structure characteristics, such as file entropy, signature, or file header analysis.

A. Entropy-based Detection

Lyda and Hamrock presented the idea of using entropy to find encrypted and packed files [3]. The method became widely used as it is efficient and easy to implement [7]. However, some non-packed files can have high entropy values and thus lead to false-positives. For example, the `ahui.exe` and `dfrgntfs.exe` files from Windows XP 32-bit have an entropy of 6.51 and 6.59, respectively for their `.text` section [8], [9] (our system detects these files correctly as non-packed). While entropy-based methods can be effective against encryption or packing obfuscation, it is ineffective against anti-disassembly tricks [6], and even against simple byte-level XOR encryption. Finally, the entropy score of a file can also be deliberately reduced to achieve an entropy value similar to that of a normal program [10].

B. Signature-based Detection

A popular signature-based tool to find packed files is *PEiD*, which uses around 620 packer and crypter signatures [11]. The obvious drawback of this tool is that it can identify only known packers, whereas sophisticated malware usually uses 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. Finally, it is a time consuming job to add a signature of a new packer since it usually requires manual analysis to extract a reliable signature.

C. File Header-based Detection

File header-based techniques include those proposed by [12]–[16]. These techniques can get good results only when the packer changes the PE header in a noticeable way. Indeed, many public packers exhibit identifiable changes in the packed PE file. However, this is not always the case with custom packers and self-encrypting malware. Moreover, if the packing avoids the PE file header completely, and instead focuses on instruction sequence or program execution flow, there will be absolutely no trace in the header. Lastly, even if a packer introduces some identifiable artifacts in the header, some of these can be easily removed by the malware author without affecting the integrity of the executable. For example, section names and some strings could be manually restored to match the original file header.

Besides the specific shortcomings of each technique described here, none of them can statically detect the presence of anti-disassembly tricks or other forms of control flow obfuscation. Unfortunately, these tricks are now commonly used in a wide range of advanced malware. This major shortcoming is the first motivation for our work to automatically detect obfuscated files. Because our proposed system does not depend on a coarse-grained entropy score of the file or section, byte signature of packers, or file header features, it overcomes the techniques' individual shortcomings while also addressing the need for detecting files containing anti-disassembly tricks and control flow obfuscation.

III. Background

A. Packer Identification

If a program is determined to be obfuscated, it is helpful to determine the obfuscator or packer that was used. Identifying the packer will accelerate the automatic unpacking process, since the file can then be directly unpacked using the correct packer. If the unpacker is not determined, a heuristic/universal unpacker must be used to emulate the instructions and guess the original entry point of the unpacked file. Universal unpackers are not as reliable as those unpackers that were made specifically for known packers and obfuscators. A universal unpacker has to make some guesses and assumptions to determine the end of unpacking process that might deem wrong. Because of the shortcomings with packer detection, reliable packer identification is also problematic. This fact represents the second motivation for our work for automatic packer identification.

B. Control Flow Graphs

For a control flow graph (CFG) G consists of a set of nodes, where each node represents a basic block. The following function maps a node to its 2-tuple of right and left children:

$$Children(x) = (x_r, x_l) \,|\, x, x_r, x_l \in G$$

In addition, parents of a node x are those nodes that have a direct connection to x. The set of parents is defined as:

$$Parents(x) = \{y \in G\} \,|\, x \in Children(y)$$

A node x is an exit node $\iff Children(x) = \varnothing$, and x is the entry node $\iff Parents(x) = \varnothing$. Each node in a CFG can have up to two children.[1] In case the node ends with a control transfer instruction to an indirect address, we consider the node as having no children, because there is almost no way to statically know the target address of the instruction. Thus, switch tables or C++ polymorphic method invocation when using a JMP or CALL instruction with indirect address, it will be treated as a childless node. Beside that, unconditional or conditional jumps with direct addresses will have one or two children, respectively.

1) CFG Manipulation: CFGs can be manipulated in a number of ways without affecting the overall logic of the program. These manipulations can be due to normal compiler optimization processes or, as in the case of malware, polymorphism. The shape of the CFG changes while the original logic does not. For example, a control flow instruction such as `JE` can be changed to `JNE`. In this case,

[1]In this work, we only consider programs composed of Intel® 64 and IA-32 instructions.

the left and right children will be swapped, resulting in a mirrored version of the original CFG. Figure 1 shows a CFG and its mirrored version, whereby each control flow instruction with two children in Figure 1(a) has been negated so as to transform it into the mirrored version shown in Figure 1(b).

Another way that CFGs can be manipulated without affecting program logic is by prepending a node at the entry node. For example, inserting a `JMP` instruction as a new entry point with an address of the first instruction of the code will create an extra basic block at the beginning of the CFG while keeping the logic unchanged. Similarly, nodes can be appended to exit nodes or inserted in the middle of the graph. Finally, a node may be split into two to increase the number of nodes.

The rest of the paper is structured as follows: Section IV explains our methodology of constructing a reliable signature from the control flow graph. Section V describes our experiments and results, while Section VI discusses the results and potential limitations. Finally, Section VII concludes the paper.

IV. Control Flow Graph-based Signature

A. Graph Preprocessing

As discussed earlier, polymorphic codes usually insert dummy instructions to evade detection by string signature. These instructions could be either instructions that do not affect the execution flow of the program, such as arithmetic or memory operations. In the first case, the control flow graph of the program will not be affected, and the CFG signature will thus stay the same. In the second, inserting control flow instructions will add more basic blocks to the CFG and change the CFG signature. For instance, a malicious program could add a series of `JMP` instructions between basic blocks, with each `JMP` pointing to the next. The malicious program could also divide each basic block into smaller pieces and connecting them with `JMP` instructions in between.

In all these cases, the logic of the program will be intact, but the presence of dummy `JMP`s will change the CFG and affects the signature unless these superfluous nodes are minimized. Thus, we preprocess the CFG in order to alleviate the effects.

First, in order to eliminate the effect of extra nodes, we normalize the graph by combining nodes with only one child into one, as follows:

$$Merge(x, y) = v \iff x_r = x_l = y \,|\, x, y, v \in G \\ and\ x_r, x_l \in Children(x)$$

Second, to mitigate the effect of mirroring, we traverse the graph, and for each node, determine the shortest distance to root and assign it this value. Then every input CFG is topologically sorted so that for each node, the deepest node on its right branch has a shorter distance to root than the deepest node on the left branch. This sorting operation is defined as follows:

$$\forall x, x_r, x_l \in G, swap(x_r, x_l) \iff x_{rd} > x_{ld}\ ,$$

where x_{rd}, x_{ld} are the depth of the right and left branches of x, respectively.

Figure 2 shows the normalized and sorted version of the graph in Figure 1(a).

Once preprocessed in this way, two graphs may be compared using two types of signatures, *exact* and *approximate*. An exact signature match identifies two CFGs that are identical. However, if one was subjected to minor manipulation, the signatures will fail to match. In this case, an approximate signature may be used in an attempt to find a match in spite of minor manipulations, although doing so may come at the cost of potential false-positives.

B. Exact Signature

There are a number of known representations of graphs. In this paper, we represent the CFG signature as a series of integer values. Nodes are visited in breadth-first fashion, and each node is given a sequential ID. Since each node in the control flow graph can have a maximum of two children, an ID is followed by two IDs for the children, then the child with the least ID is listed followed by its children, etc. Each node is represented as 3-elements tuple of $(NodeID, Child1ID, Child2ID)$. If a node has only one child, the second nonexistent child is represented by zero. In case of an exit node that has no children, both children are represented by two zeros. For example, the following signature represents the graph in Figure 2.

```
1 2 3 2 4 5 3 0 0 4 0 0 5 0 0
```

The number of nodes in the graph equals the length of the signature divided by 3. With this representation, the original graph can be fully restored from the signature.

Obviously, the node ID is repeated when it is listed as a parent and as a child. Thus, the signature can be shortened by removing $NodeID$, and then each two consecutive IDs represent the two children of a node. The previous signature can then be rewritten as:

```
2 3 4 5 0 0 0 0 0 0
```

In this version, the number of nodes equals the length of the signature divided by 2 and still the original graph shape can be fully restored with this exact signature.

C. Approximate Signature

As mentioned earlier, it is possible for the CFG to be modified in various ways without affecting the original program logic. Thus, exclusively relying on finding exact signature matches will miss many isomorphic graphs and increase the rate of false negatives. In order to overcome this problem and match similar CFGs, we construct an approximate representation of the graph, as follows.

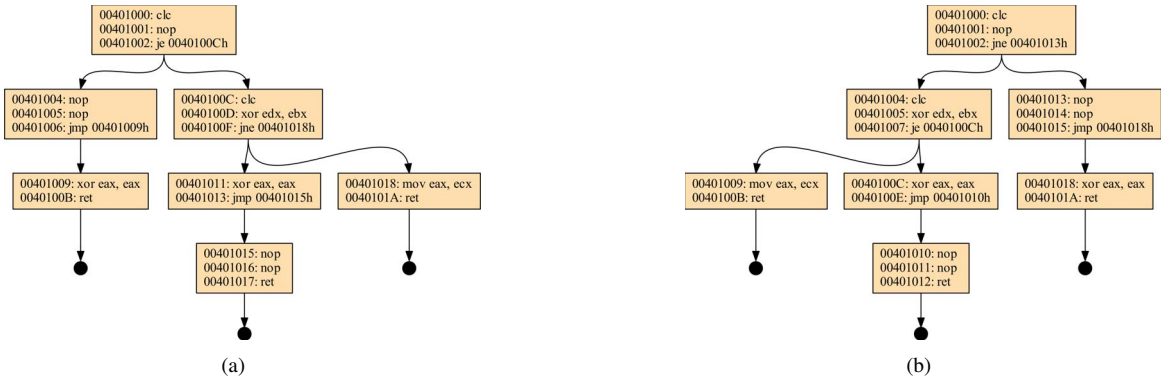

Fig. 1. (a) An example CFG, (b) a mirrored version of the same CFG.

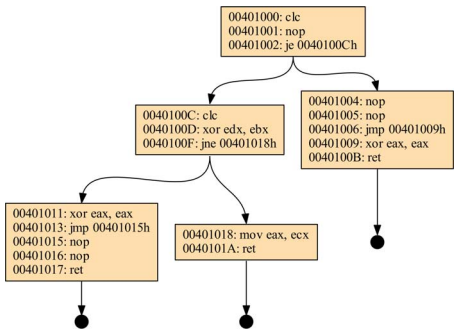

Fig. 2. A normalized and sorted version of the CFG in Figure 1(a).

Each node is visited in a breadth-first fashion and labeled with two values, x, y, representing the number of parents and the number of children of each node, respectively. Since each node cannot have more than two children, the possible values for y are 0, 1, 2, and thus y can be represented with 2 bits. On the other hand, the possible number of parents is unlimited. However, our sampling of 300,000 files found that the average number of parents for all CFG nodes in these files is 1.3. Moreover, only 130 files in this sample set have a CFG containing at least one node with more than 63 parents.

Since the number of children can be represented in 2 bits, there are 6 remaining bits available for the number of parents, assuming the entire label is limited to one byte. Thus, a one-byte label can represent a maximum of 63 parents, which we believe to be sufficient for the vast majority of CFGs, based on our sampling.

To illustrate this, we again refer to the example of Figure 2. For this example, the following hexadecimal sequences represent the number of children, parents, and the combined value, respectively. For each byte in the combined sequence, the two low order bits represent the number of children, and the high order six bits represent the number of parents.

 # children: `02 02 00 00 00`
 # parents: `00 01 01 01 01`
Combined label: `02 06 04 04 04`

There are many properties that each node in the CFG can have, such as, number of parents, number of children, shortest distance to exit node, shortest distance from root, number of instructions and level. However, with the exception of the number of children and parents, examining these properties shows that they are susceptible to change even if a single node is inserted or removed. For example, if we define a signature based on the distance of each node to the root, and then a new node is inserted as a new root, the entire signature will be completely different. The same can happen if we used a shortest distance to exit node measurement. Although the number of instructions in each node will not be affected by CFG manipulation, it will change if dummy instructions are inserted or removed, something very common in polymorphic code, and usually easier to accomplish than changing the CFG.

Overall for a given CFG, the number of direct children and parents are resilient to these modifications. Only the node that is a direct children or parent of a modified node will have its value in the signature changed. The rest of the CFG is not affected in terms of parents or children and thus the rest of the signature will be the same.

This makes the choice of these two properties a good fit to detect mildly different or optimized code when the signature of a similar variant is known.

D. Signature Matching

To compare two signatures, we use a difference score based on edit distance as a metric for comparison. The difference score between two signatures x and y is defined as the number of insertion, deletions and substitutions needed to convert one string to another. We define a value δ as the number of allowed distance between two signatures to establish a match. Each signature x in the database has its own value of δ to make sure we match variants of x correctly. Hence, a match between two signatures x and y is defined as:

$$x \text{ and } y \text{ are matched} \iff 0 <= EditDist(x, y) <= \delta_x,$$

where x is the signature stored in the database and y is the signature of the input file.

V. Experiment and Evaluation

To evaluate the utility of our signature and matching algorithm, we conducted an experiment using 7 publicly-available packers commonly abused by malware authors to obfuscate their code: *UPX*, *Execryptor*, *Themida*, *FSGv1.33*, *FSGv2.0*, *eXpressor* and *Yoda's Protector*. For each packer, one obfuscated file was sufficient to get a CFG signature able to detect all files in each test set of 20 obfuscated files for each packer.

For each file in the data set, the control flow graph is extracted from the function at the entry point. The extraction is stopped at 200 basic blocks limit. In the case of Yoda's protector, the test set needed 3 signatures in PEiD to detect all of them, as the test set was packed by 3 different versions of the packer. However, we only needed one signature of the CFG to detect files packed by the 3 versions. Each signature successfully detected 100% of the test set. Table I shows the δ of each signature, where each δ in the table is obtained by scanning the test set and obtaining the maximum difference score.

Table II shows the distance of each signature from the other 6 packers. Note that the distance between a signature of packer x and the other 6 packers, is higher than δ_x, which indicates that files belong to a packer x will not be misidentified as packer y, for any two different packers x and y.

To measure the false-positive rate for each signature, we scanned 324 non-packed files taken from a clean Windows 7 machine. We had zero false-positive rate with all the signatures. In fact, the difference score was much greater than the δ value of each signature. Table III shows the lowest difference score for each signature when scanning the non-packed files.

VI. Discussion and limitations

During the experiment, we noted that UPX has a number of slightly different CFGs for some files even if the same version is used. We found that during packing, UPX will add code blocks in the unpacking stub based on the input file's structure.

For example, if the file has a relocation section, UPX unpacking stub will contain some code that deal with unpacking the relocation section, and this code will not be present in the stub of other files that do not have a relocation section. This is in fact the reason UPX have higher value of δ than the other packers. That is, the more difference among the CFGs belong to the same packer, the higher the value of δ needed to match all of them. Table IV shows the different edit distance scores we obtained when scanning UPX file set.

A typical PEiD signature and CFG signature for UPX are as follows (`??` in the PEiD signature denotes a wildcard, matching any byte value):

PEiD
```
60 BE ?? ?? ?? ?? 8D BE ??
?? ?? ?? C7 87 ?? ?? ?? ??
?? ?? ?? ?? 57 83 CD FF EB
0E ?? ?? ?? ?? 8A 06 46 88
07 47 01 DB 75 07 8B
```

CFG
```
01 09 0A 05 05 0E 0E 0A 05
06 0A 06 0A 06 0A 05 05 05
0A 05 0D 0A 05 0A 0A 0A 0E
06 05 05 0A 05 06 06 05 09
06 0A 09 05 05 05 05 0A 0A
05 04 06 05 05
```

Yoda's Protector file set is obfuscated using three versions of the packer which are v1.02b, v1.03.2 and v1.03.3. PEiD is using three signatures to detect all the files in the set, one signature for each version. Nevertheless, only one CFG signature with δ value of 2 was needed to detect the three versions. On the other hand, we chose to separate the signatures for FSG packer, so we have two different signatures to detect the FSG v1.33 and FSG v2.0, this is why we have δ of value 1 for each. However, if one signature is needed to match both versions, the δ is determined by experiment to be 6. Therefore, it is up to the system administrator to choose whether to have lower value of δ and more signatures, or higher value of δ with less number of signatures, considering the risk of false-positive with high δ.

We set a minimum and maximum size of nodes of 30 and 200 nodes, respectively. Thus, an input CFG with a number of nodes less than the minimum is skipped, as considering a CFG with size less than this minimum could produce high number of signature mismatches. On the other hand, the maximum number of 200 is thought to be enough to detect different CFGs, and so if the input file has more nodes than the maximum, it is trimmed down to 200 nodes.

Regarding the speed of the system, the process of scanning a file involves disassembly, control flow graph extraction, signature generation and matching. This process takes an average of 0.5 ms per file on an Intel® Xeon® processor with four cores and 8GB main memory.

A limitation of the system is imperfect disassembly due to indirect addressing. However, this problem exists in all disassemblers since the problem of perfect disassembly is an undecidable problem [17]. The use of indirect addressing at or near the entry point will prevent our disassembler from extracting the CFG afterwards. Also, the use of some obfuscation techniques such as multi-nodes junk code insertion or code flattening could hinder the comparison if they were applied to some samples in a set belongs to a certain packer. Therefore, as a future publication, we are currently working on using an emulator in an optimized way to get an accurate representation of the CFG in a near realtime speed. This will enable us to get a good signature that overcomes these obfuscation techniques.

VII. Conclusion

Hundreds of thousands of malware are produced every year, with the vast majority of them obfuscated and packed.

TABLE I
VALUE OF δ FOR EACH SIGNATURE.

Packer	UPX	Execryptor	eXpressor	Themida	FSG v1.33	FSG v2.0	Yoda
δ	10	1	1	1	1	1	2

TABLE II
THE DISTANCE OF EACH PACKER SIGNATURE FROM THE OTHER 6 PACKERS.

Packer	UPX	Execryptor	eXpressor	Themida	FSG v1.33	FSG v2.0	Yoda
UPX	0	37	156	38	30	29	31
Execryptor	37	0	184	12	31	24	17
eXpressor	156	184	0	182	159	167	175
Themida	38	12	182	0	35	25	18
FSG v1.33	30	31	159	35	0	12	31
FSG v2.0	29	24	167	25	12	0	24
Yoda	31	17	175	18	31	24	0

TABLE III
MINIMUM SCORE FOR EACH PACKER AGAINST NON-PACKED FILES.

Packer	UPX	Execryptor	eXpressor	Themida	FSG v1.33	FSG v2.0	Yoda
δ	29	12	103	12	24	17	17

TABLE IV
EDIT DISTANCE SCORES OF UPX FILE SET.

Score	0	3	6	10
Number of files	7	11	1	1

Unpackers are used to reveal the malicious code of the file by unpacking its contents.

It is important to identify the packer used in the malware in the triage process, so the appropriate packer can operate on the file. Current techniques of packer identification rely mainly on byte signature, which can be easily evaded by introducing extra nodes and mirroring. In this paper, we introduce a new technique for packer and obfuscator identification based on a normalized and sorted CFG of the program. Exact and approximate signatures are then extracted from the preprocessed CFG and used to search for matches against a database of known packer signatures. The use of the approximate signature enables the matching algorithm to withstand the minor code modifications usually introduced by attackers in order to evade detection.

The method was tested using a dataset of 20 files for each of seven common packers. Performance was good, with CFG normalization, sorting and signature generation requiring an average of only 0.5 ms per file, and no false positives. We believe the technique has the potential to efficiently detect malware variants and different versions of the same code, which will be the focus of our future work.

Acknowledgement. This research was supported in part by ARO Grant #W911NF-13-1-0141, NSF Grant #1111925, and AFRL project RIGIHIDR.

REFERENCES

[1] M. Labs, "Mcafee labs threats report for september 2016." https://www.mcafee.com/us/resources/reports/rp-quarterly-threats-sep-2016.pdf. Accessed: April. 23th, 2017.

[2] C. T. Alliance, "Cryptowall version 3 sequel: An analysis to one of the most lucrative ransomeware cryptowall version 4 threat." https://www.cyberthreatalliance.org/pdf/cryptowall-report.pdf. Accessed: April. 23th, 2017.

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

[4] E. O. Osaghae, "Classifying packed programs as malicious software detected," *analysis*, vol. 20, no. 10, p. 19, 2016.

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

[6] M. Saleh, E. P. Ratazzi, and S. Xu, "Instructions-based detection of sophisticated obfuscation and packing," in *2014 IEEE Military Communications Conference*, pp. 1–6, Oct 2014.

[7] G. Jeong, E. Choo, J. Lee, M. Bat-Erdene, and H. Lee, "Generic unpacking using entropy analysis," in *2010 5th International Conference on Malicious and Unwanted Software*, pp. 98–105, Oct 2010.

[8] VirusTotal.com, "`ahui.exe`." http://goo.gl/QYKW22. Accessed: Aug. 29, 2016.

[9] VirusTotal.com, "`dfrgntfs.exe`." http://goo.gl/XCqUcF. Accessed: Aug. 29. 2016.

[10] 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.

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

[12] D. Devi and S. Nandi, "Pe file features in detection of packed executables," *International Journal of Computer Theory and Engineering*, vol. 4, no. 3, p. 476, 2012.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] R. N. Horspool and N. Marovac, "An approach to the problem of detranslation of computer programs," *The Computer Journal*, vol. 23, no. 3, pp. 223–229, 1980.