International Journal of Security and Its Applications
Vol.7, No.5 (2013), pp.257-268
http://dx.doi.org/10.14257/ijsia.2013.7.5.24

# A Heuristics-based Static Analysis Approach for Detecting Packed PE Binaries

Rohit Arora, Anishka Singh, Himanshu Pareek and Usha Rani Edara

*Centre for Development of Advanced Computing,*
*Hyderabad, India*
*{rarora, anishkas, himanshup, ushae}@cdac.in*

***Abstract***

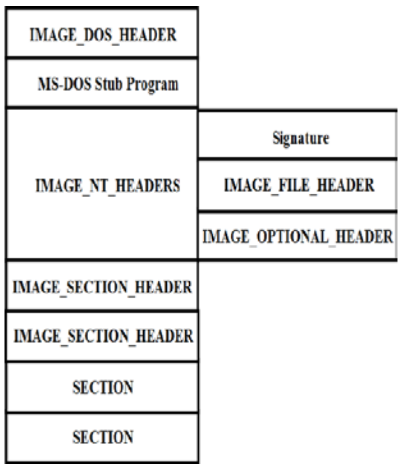
*Malware authors evade the signature based detection by packing the original malware using custom packers. In this paper, we present a static heuristics based approach for the detection of packed executables. We present 1) the PE heuristics considered for analysis and taxonomy of heuristics; 2) a method for computing the score using power distance based on weights and risks assigned to the defined heuristics; and 3) classification of packed executable based on the threshold obtained with the training data set, and the results achieved with the test data set. The experimental results show that our approach has a high detection rate of 99.82% with a low false positive rate of 2.22%. We also bring out difficulties in detecting packed DLL, CLR and Debug mode executables via header analysis.*

***Keywords****: Obfuscated, packed, static analysis, heuristics*

## 1. Introduction

To evade widely deployed and trusted signature-based detection malware authors obfuscate the code using various techniques [1]. The obfuscation of a binary file can be achieved either by packing the original file or trying to create new versions of the old malware program by modifying syntax of code without changing semantics. Our focus in this paper is to detect the binary executables that are packed. This method of obfuscation refers to encrypting the original file or code and embedding it as data in another file. For this malware authors may rely on existing packer tools such as UPX [2], Themida [3], AS Pack [4] *etc* or write their own custom packers. Packed executable detection on static parameters can be done either by using signature based approach or by using static heuristics based approach. PEiD [5] is a commonly used packed executable detector based on signatures. These approaches will be of little help to identify executables packed with previously unknown packers.

Figure 1(a) shows the format of a PE executable file [6]. PE file contains essential headers and various sections containing code and data. Each section has its corresponding section header. File starts with IMAGE_DOS_HEADER which provides location of IMAGE_NT_HEADERS. IMAGE_OPTION_HEADER contains fields like number of sections, size of code, address of entry point, size of initialized data and un-initialized data *etc*. To construct an executable which unpacks itself into memory, malware authors first encrypt the executable and create a new executable with the encrypted code as data and the code to decrypt the data. This is depicted in Figure 1(b). Our tool extracts various section header characteristics from the PE file and also calculates entropy of all sections. The extracted characteristics are quantified and a score is computed based on power distance method. This score is used in classifying a binary as packed or not-packed. The above mentioned contributions apply essentially to Windows PE executable files.



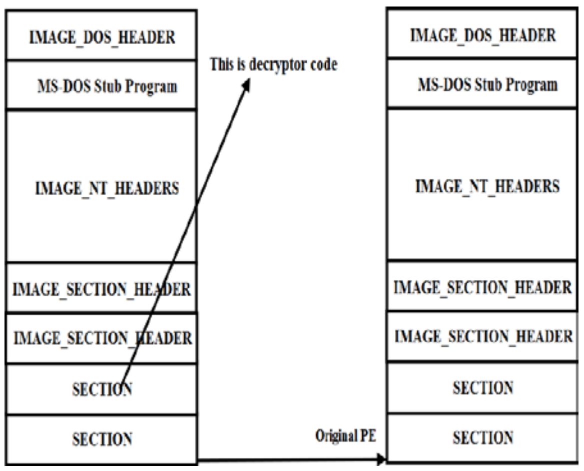


**Figure 1(a). PE File Structure** **Figure 1(b). Unpacking process of a Packed PE**

The rest of this paper is organized as follows. Section 2 brings out the related work. Section 3 presents our approach towards packed executable detection. Section 4 brings out the results and Section 5 concludes the paper with future work.

## 2. Related Work

Rober Lyda and James Hamrock presented an entropy analysis approach for detecting the encrypted and packed malware [8]. Their analysis is based on the fact that encrypted bytes possess a high degree of randomness. Roberto et al. proposed a packed executable detection method based on feature extraction. PE header features are extracted, analysed, and finally given to the classifier for packed executable classification. The experiments were performed using various machine learning algorithms [9]. Choi *et al.*, proposed an approach to detect packed executables based on header analysis. Eight characteristic values are selected from the attributes of the PE file header. Their approach uses Euclidean distance as a quantifying measure [10]. S. Treadwell and M. Zhou proposed a heuristic approach for detection of obfuscated malware [11]. Their approach utilizes a risk analysis matrix and a risk score is computed to determine if file under analysis is malicious. Igor Santos *et al.*, presented a collective-learning-based packed executable detection system [12]. Various PE header characteristics and entropy are used and collective classifier algorithms are used to classify the packed executable files. S. Han *et al.*, also proposed a static analysis approach to detect packed executable files [13]. On the other hand, dynamic analysis techniques such as code emulation and dynamic translation also exist for packed executable detection but it may be difficult for deployment at host level or gateway level [14].

Our approach is based on the analysis of various characteristics pertaining to the PE such as section headers, section entropy *etc*. Our study is based on an insight that a sensibly chosen set of PE characteristics could yield tangible results in the process of packed executable detection.

## 3. Our Approach

Firstly, we define various static parameters for analysis and associate a weight and risk with each defined parameter. Then we analyse a given binary to find if these parameters are exhibited. A score is calculated and PE file is classified as packed or not packed. Figure 2 shows the flow chart depicting the process followed for packed executable detection.

## 3.1. PE Heuristics

We started with experiments to identify attributes which can be used to classify the packed executables. These attributes are explained in subsections outlined below. We also present taxonomy of PE header heuristics that could be used in general with any heuristic analysis approach based on the PE file format. Essentially, the heuristics pertaining to the PE file format could be classified based on the scheme depicted in Figure 3. The categories include viz. Entry point checks, Permissions checks, checks on Import table, and Section name checks. The heuristic related to entry point for *e.g.*, "Is entry point pointing to the executable section" goes into the Entry point category of checks. Likewise, the heuristics pertaining to the import table goes into the Import checks category and so on. This kind of classification provides ease of use and facilitates to add new heuristics pertaining to any of the categories. In addition, it enables to derive a mechanism for final score computation based on weights assigned to each category.

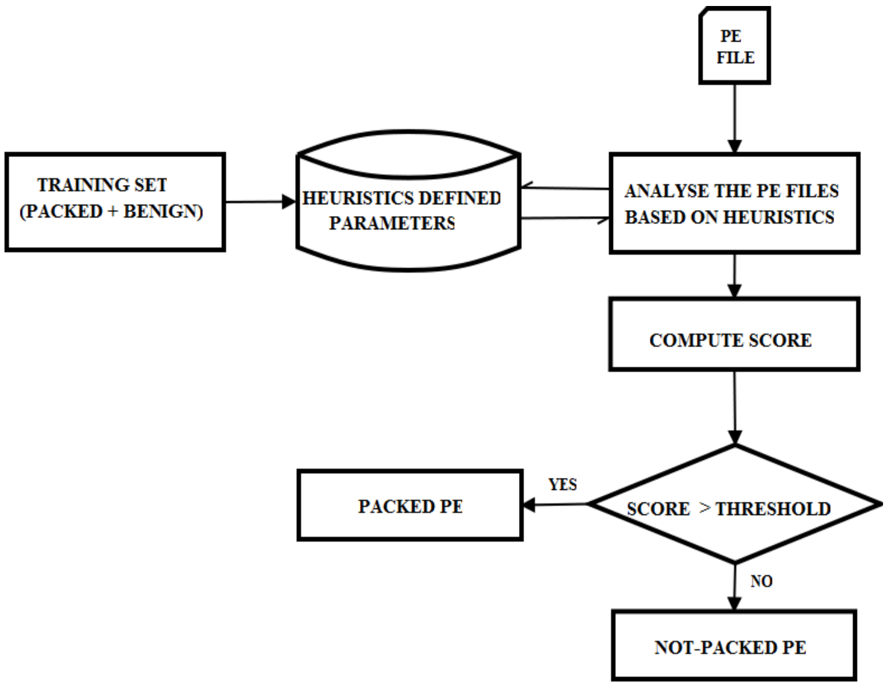

PE FILE
TRAINING SET (PACKED + BENIGN)
HEURISTICS DEFINED PARAMETERS
ANALYSE THE PE FILES BASED ON HEURISTICS
COMPUTE SCORE
SCORE > THRESHOLD
YES
PACKED PE
NO
NOT-PACKED PE

**Figure 2 Steps Involved in Packed Executable Detection**

### 3.1.1. Section Names

**IF_PACKED_NAMES:** By analysing a large dataset of packed malwares, we construct a list of section names which are commonly used by packers. If a section name used in PE file belongs to this list then this property is set. For example, .UPX, .PECOMPACT *etc*.

**NUM_NOT_KNOWN:** We construct a list of standard section names as per the Microsoft PE specification [7]. If a section name is not found neither in this list nor in the list of names used by packers, then this property is set. The usage of previously unseen section names is suspicious.

**NUM_SECTION_WITH_NON_PRINTCHAR:** This property defines the number of sections containing non printable characters. Standard practice of defining section names does not contain any non-printable characters. Thus, while examining a PE file if any section name appears with non-printable characters then this property is set.

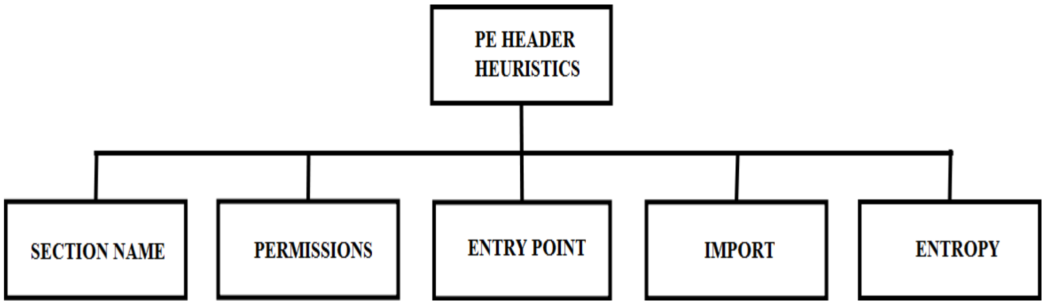


**Figure 3. Taxonomy of PE Heuristics**

**NUM_NO_NAME:** This property is set when a section is found with no name. Many packed executables are found with no section name. This heuristic also provides an effective contribution in identifying a packed executable.

### 3.1.2. Permissions

**NO_CODE_SECTION:** An executable which does not contain code section gives a direct indication in packed executable detection, as there is a higher probability of code section being hidden.

**NUM_EXE_CODE_MISMATCH:** If a section contains code, it is essential to have executable permissions for executing that particular code and vice versa. If any mismatch is found in these conditions then this property is set.

**NUM_MEM_EXE_WRITE:** The packed executable program should first run an unpacking routine in order to unpack the packed program. The unpacking process entails writing unpacked code in an executable section of the memory image of running program. Therefore, a packed program needs to include at least one section which is writeable as well as executable at the same time. On the other hand, the executable sections in the non-packed PE file need not be writeable and so the MEM_EXE_WRITE flag is not set. Consequently, counting the number of sections which are writeable and executable makes a significant contribution in concluding whether an executable is packed.

**NUM_DATA_EXECUTE:** The .data section contains the initialized data. In case of non-packed files read and write permissions are desirable for the .data section. However .data section can be used to store the unpacking routine in case of a packed executable. In such a case .data section needs to be given executable permissions. Therefore, a data section having executable permissions gives an upswing.

**NUM_DATA_CODE:** If an executable is packed or encrypted then it may contain the code and data in the same section in order to run the unpacking routine. If any such section exists this property is set.

### 3.1.3. Entry Point

**FAKE_ENTRY_POINT:** Entry point of an executable is where the execution starts. The section where the entry point lies should have the code and execute flags. If entry point of an executable violates this condition, then it is considered as fake entry point.

**IS_TLS_CODE:** An executable can optionally contain the entry point function in Thread Local Storage. This is considered as hidden entry point.

### 3.1.4. Import Tables

**IS_LESS_IMPORTS:** Packed executables contains less number of imports as actual functionality is encrypted and code which decrypts the encrypted executable is the only one in original form. If number of import is less than 20, then this property is set.

**IT_NON_STD_SECTION:** The Import Address Table (IAT) holds the addresses of the imported functions. In case of normal files, this IAT is stored in the standard sections such as .text section. When an executable is packed, as the standard section names are encapsulated into the newly added sections, apparently the IAT appears in a non-standard section.

### 3.1.5. Entropy

**SECTION_ENTROPY:** Entropy is a method for measuring the uncertainty in the series of bytes. Frequency of each byte (00h-FFh) is used to calculate the entropy. Entropy is calculated as

$$entropy = -\sum_{i=0}^{N} p(i)\log_2 p(i)$$

where $p(i)$ is the probability of the occurrence of a byte in that particular section. Higher entropy score reflects more uncertainty in a series of bytes and indicates towards encrypted data. Thus, we calculate the entropy of each section and the highest entropy among all the sections is considered as the final entropy measure of the executable [8].

## 3.2. Score Computation

Secondly, we perform executable analysis using the heuristics defined above. This analysis essentially involves computing a score using power distance method based on weights and risk score associated with each heuristic. Table 1 shows the weight and risk score associated with each of the heuristics. Weight given to each heuristic represents how better it is an indication that the executable under analysis could be packed. The risk score is assigned based on how risky a particular property is if found in a binary file. Both weight and risk score together are used in computing a score value, which further determines whether a PE has been PACKED or NOT. Initially, we assigned a predefined weight and risk score for each heuristic based on our study and experience with respect to packed executables. Likewise, for few heuristics such as IF_PACKED_NAMES, NUM_SECTION_WITH_NONPRINTCHAR and NUM_NO_NAME we observed that the actual value of the heuristic stands more appropriate for weight rather than an assigned number. Therefore, for those heuristics mentioned the weight is considered as the '*count*' obtained for that heuristic. After carrying out the tests with the training data set, we have stabilized the values for weight and risk score against each heuristic. The stabilized values are shown in Table 1.

For computing the score, our approach uses power distance. The formula is mentioned below.

$$score = \sqrt{\sum_{i=0}^{N} W_i^{R_i}}$$

Where $W_i$ represents weight of the ith parameter and $R_i$ is the corresponding risk.

## 3.3. Classify Executable

Finally, we classify the executable as PACKED or NOT PACKED based on the score value obtained in score computation above. The score value is compared against a threshold value, and if the score value is above the threshold then the executable is said to be PACKED else NOT PACKED. The threshold value is evolved based on the results obtained with the training data set. The next subsection under the Results section showcases the results obtained with the training data set and the evolved threshold value.

**Table 1. PE Heuristics with Weights and Risk Score**

| S. No | Type | Parameters | Weight | Risk |
|---|---|---|---|---|
| 1 | Section Name | IF_PACKED_NAMES | Count | 5 |
| 2 | Section Name | NUM_NOT_KNOWN | 0.5 | 1 |
| 3 | Section Name | NUM_SECTION_WITH_NONPRINTCHAR | Count | 2 |
| 4 | Section Name | NUM_NO_NAME | Count | 2 |
| 5 | Permission | NO_CODE_SECTION | 5 | 3 |
| 6 | Permission | NUM_EXE_CODE_MISMATCH | 5 | 4 |
| 7 | Permission | NUM_MEM_EXE_WRITE | 5 | 4 |
| 8 | Permission | NUM_DATA_EXECUTE | 5 | 4 |
| 9 | Permission | NUM_DATA_CODE | 5 | 2 |
| 10 | Entry point | FAKE_ENTRY_POINT | 5 | 3 |
| 11 | Entry point | IS_TLS_CODE | 3 | 2 |
| 12 | Import | IS_LESS_IMPORTS | 3 | 2 |
| 13 | Import | IT_NON_STD_SECTION | 7 | 3 |
| 14 | Entropy | SECTION_ENTROPY | 3 | 2 |

## 4. Results

### 4.1. Training Data Set and Test Environment

Our test lab environment has a honey-pot setup using Dionaea tool at a broadband internet connection. This tool emulates the vulnerabilities, invites the attacks and logs the malware. In addition, we have collected malware from other sharing websites like offensivecomputing.net, and contagiodump.blogspot.com. Out of the collection from our lab, we extracted the training set consisting of 2078 packed executables. For identifying the packed executables, we used the emulation approach [14]. A set of 4677 benign executables is also collected from System32 folder of a freshly installed Windows machine.

### 4.2. Fallouts Obtained with the Training Data Set

This section presents the results obtained from analysis carried out on the training data set. Figure 4 shows the true positive rate i.e. the number of input packed executables classified correctly as PACKED. It represents the true positive rate for different threshold values. For the threshold values of 1, 2 and 3 there is no change in the true positive rate.

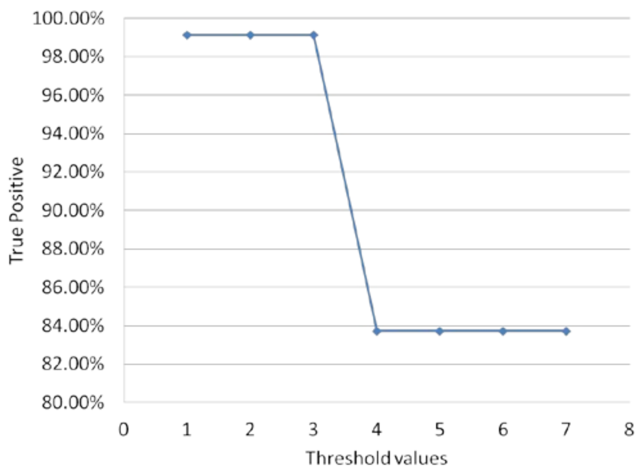


**Figure 4. True Positive Rate with Various Threshold Values**

Figure 5 represents the false positive rate (FPR) with various threshold values. FPR is percentage number of benign executables classified as PACKED. The false positive rate decreases as we increase the threshold value. False positive rate for threshold values of 1, 2 and 3 are 2.26%, 2.22% and 2.22% respectively. As threshold value is increased to 4 false positive rate falls considerably but it also reduces the true positive rate which is detrimental and optimized threshold value is 3.

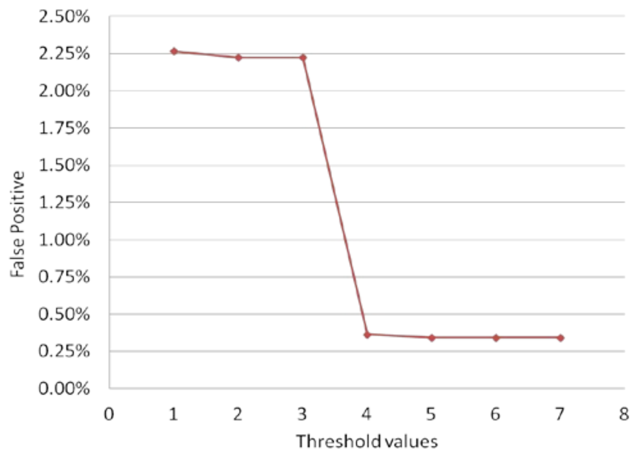


**Figure 5. False Positive Rate for Various Threshold Values**

Figure 6 shows the false negative rate. It is number of input packed executables that are incorrectly classified as NOT PACKED. Our approach shows a very low false negative rate of 0.086% on the training data set.

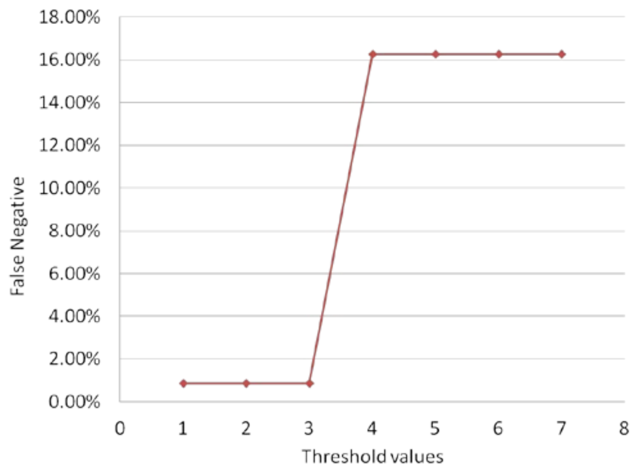


**Figure 6. False Negative Detection Rate with Varying Thresholds**

**Table 2. Results Obtained with the Training Data Set**

| Results with Training Data Set | Threshold Value = 1 | Threshold Values = 2 & 3 |
|---|---|---|
| Total Test Set (Packed/Not Packed) | 6755 (2078/4677) | 6755 (2078/4677) |
| Detected | 4571 | 4571 |
| Not Detected | 2060 | 2060 |
| Detection Rate | 99.13% | 99.13% |
| False Positive Rate | 2.26% | 2.22% |
| False Negative Rate | 0.08% | 0.08% |

### 4.3. Results with the Test Data Set

We have arrived at the threshold value of 3 for our approach on basis of the detection rates mentioned in Table 2. For the test data set, we have collected 8568 samples from virussign.com. Out of that, our approach classified 4500 binaries as PACKED and 4068 as NOT PACKED with a False Negative rate of 0.17%. Figure 7 represents the final scores calculated for test data.

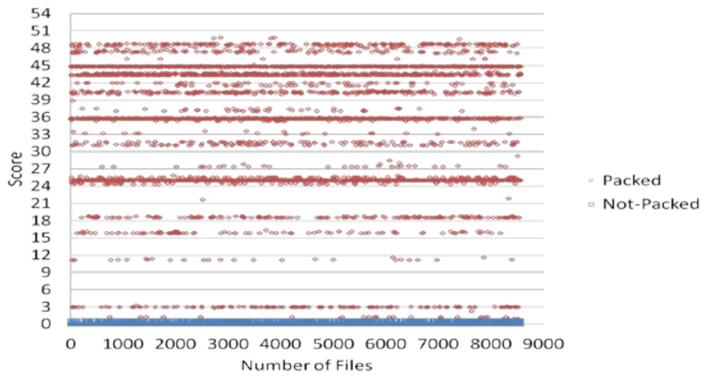


**Figure 7. Test Set Result**

### 4.4. Analysis of Common Packers

In Table 3 shown below, we present the score ranges of some common packers analysed.

**Table 3. Score Range of known Packers**

| S. No. | Packer Name | Score Range |
|---|---|---|
| 1 | UPX | 16.10 – 48.86 |
| 2 | PECompact | 25.17 – 44.91 |
| 3 | VMProtect | 31.81 – 48.21 |
| 4 | Armadillo | 01.00 – 48.68 |
| 5 | MingWin32 | 25.00 – 43.59 |
| 6 | AsPack | 16.37 – 49.94 |
| 7 | NsPack | 40.32 – 48.12 |
| 8 | Yoda | 19.02 – 48.21 |

We also present scores of some common malwares analysed with our solution in Table 4.

**Table 4. Score Range of known Malwares**

| Malware Name | MD5 Hash | Score | Peid Status |
|---|---|---|---|
| W32/Conficker | 83c52b56b1ecbe23183bae5e05474e3e | 48.68 | UPX |
| Stuxnet | b4429d77586798064b56b0099f0ccd49 | 3.08 | Unknown |
| Backdoor.Win32.Evilbot | 54faf63f7833cfad9c1422087e9f767e | 9.89 | Unknown |
| Additional Flamer | ee4b589a7b5d56ada10d9a15f81dada9 | 45.01 | UPX |
| Flamer | 37c97c908706969b2e3addf70b68dc13 | 3 | Unknown |

## 4.5. Observations

### 4.5.1. Exceptions with DLL, Debug Mode, and CLR files

With our tests we observed that some of the heuristics cannot be applied to files such as DLLs, PEs built in debug mode, and PEs built with CLR. Previous work to detect packed executables fails to make this observation. The details indicated by these files are given below.

**(a) DLL:** When our solution has been tested over DLL's, exceptional results were recognized. Separate evaluation was done for the DLL files to see the unusual behaviour. While evaluating them we encountered that it is setting NUM_DATA_EXECUTE property. Analysis on the characteristic IMAGE_DLLCHARACTERISTICS_NO_SEH was done to check whether the DLL files set this flag. When this flag is found set in the DLL files, it will also have data section with executable permissions. Therefore, NUM_DATA_EXECUTE heuristic cannot effectively contribute towards detection in case of such DLL files.

**(b) PE built in Debug mode**: While running our solution on the executables built in debug mode, we observed that many of the heuristic check are true resulting in high scores. By analysing the executables, we found that properties viz. NUM_EXE_CODE_MISMATCH, NUM_MEM_EXE_WRITE, NUM_DATA_EXECUTE and NUM_DATA_CODE are present inside the executable. This is due to .textbss section which is included in all debug built executables. So we have kept a check on whether executable is built with debug mode or not by checking IMAGE_DEBUG_TYPE_CODEVIEW type. If it is a debug built executable then above mentioned heuristics are not considered.

**(c) PE built with CLR**: Executables built with CLR i.e. Common Language Runtime defined for .net framework, imports only the mscoree.dll. This increases the score due to the IS_LESS_IMPORT heuristic. So, if an executable built with .net framework is found, then IS_LESS_IMPORT heuristic does not contribute in the detection process.

## 4.6. Performance Analysis

Performance analysis of our implementation has been carried out for calculating the time taken by a PE file to get analysed. A set of 2000 already classified packed executables is taken and tested on both windows and Linux platform. Similar analysis has been done for a set of classified benign executables and a set of mixed files (Benign + Packed files). Average time taken per file is presented in Table 5.

**Table 5. Average Time Taken to Analyse PE Files**

| | Packed | Benign | Packed and Unpacked |
|---|---|---|---|
| Average Time on Windows (ms) | 0.51 | 0.34 | 0.41 |
| Average Time on Linux (ms) | 1.7 | 1.91 | 1.82 |

Average time taken to analyse any file on Microsoft's Windows 7 platform is 0.41 milliseconds and for Ubuntu 11.10 it is 1.82 milliseconds. This analysis was carried out on an Intel Core i7-3770 CPU with a clock frequency of 3.40GHz.

## 5. Conclusion and Future Work

In this paper, we presented a static analysis approach for packed executable detection. We also presented taxonomy of heuristics that provides ease of use and facilitates accumulation of additional heuristics in a structured manner. Our approach used power distance for score computation based on weights and risks assigned to the defined heuristics and classified the packed binaries based on the threshold observed with the training data set. We also emphasized that heuristics applied in such a manner has to be optimized for executables built in debug mode, DLLs and CLR executables. Our future direction extends to detect packed files based on combination of static analysis and emulation based approaches.

## References

[1] F. Guo, P. Ferrie and T. Chiueh, "A study of the packer problem and its solutions", Recent Advances in Intrusion Detection, **(2008)**.
[2] M. Oberhumer, "Ultimate Packer for Executables", http://upx.sourceforge.net/. 2007.
[3] Oreans Technology, "Themida Packer", http://www.oreans.com/themida.php. 2008.
[4] ASPack Software, "ASPACK Tool", http://www.aspack.com. 2007.
[5] X. P. Snaker, "PEiD", Available in: http://www.peid.info/. 2008.
[6] M. Pietrek, "Peering inside the PE: A Tour of the Win32 Portable Executable File Format", Microsoft Systems Journal-US Edition, **(1994)**, pp. 15-38.
[7] Microsoft, MSDN, "Microsoft PE and COFF Specification", http://msdn.microsoft.com/en-us/library/windows/hardware/gg463119.aspx. 2013.
[8] R. Lyda, and J. Hamrock, "Using entropy analysis to find encrypted and packed malware", Security & Privacy, IEEE, vol. 5, no. 2, **(2007)** March-April, pp. 40-45.
[9] R. Perdisci, A. Lanzi and W. Lee, "Classification of packed executables for accurate computer virus detection", Pattern Recognition Letters, vol. 29, no. 14, **(2008)**, pp. 1941-1946.
[10] Y. Choi, I. Kim, J. Oh and J. Ryou, "Encoded Executable File Detection Technique via Executable File Header Analysis", International Journal of Hybrid Information Technology, vol. 2, no. 2, **(2009)** April.
[11] S. Treadwell and M. Zhou, "A heuristic approach for detection of obfuscated malware", IEEE International Conference on Intelligence and Security Informatics, **(2009)** June.
[12] I. Santos, X. Ugarte-Pedrero, B. Sanz, C. Laorden and P. G. Bringas, "Collective classification for packed executable identification", Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti- Abuse and Spam Conference (CEAS), **(2011)**, pp. 23-30.
[13] S. Han, K. Lee and S. Lee, "Packed PE File Detection for Malware Forensics", 2nd International Conference on Computer Science and its Applications, **(2009)** December.
[14] A. E. Stepan, "Defeating polymorphism: beyond emulation", Proceedings of the Virus Bulletin International Conference, **(2005)**.