Contents lists available at ScienceDirect

Pattern Recognition Letters

journal homepage: www.elsevier.com/locate/patrec

# Classification of packed executables for accurate computer virus detection

Roberto Perdisci [a,*], Andrea Lanzi [b,c], Wenke Lee [a,b]

[a] *Damballa, Inc., 817 West Peachtree Street NW, Atlanta, GA 30308, USA*
[b] *Georgia Tech Information Security Center, Georgia Institute of Technology, Atlanta, GA 30332, USA*
[c] *Dipartimento di Informatica e Comunicazione, Universitá degli Studi di Milano, Milano 20122, Italy*

ARTICLE INFO

*Article history:*
Received 25 October 2007
Received in revised form 4 June 2008
Available online 3 July 2008

Communicated by N.S.V. Rao

*Keywords:*
Computer security
Pattern recognition
Packed executables
Computer virus detection

ABSTRACT

Executable packing is the most common technique used by computer virus writers to obfuscate malicious code and evade detection by anti-virus software. Universal unpackers have been proposed that can detect and extract encrypted code from packed executables, therefore potentially revealing hidden viruses that can then be detected by traditional signature-based anti-virus software. However, universal unpackers are computationally expensive and scanning large collections of executables looking for virus infections may take several hours or even days.

In this paper we apply pattern recognition techniques for fast detection of packed executables. The objective is to efficiently and accurately distinguish between *packed* and *non-packed* executables, so that only executables detected as *packed* will be sent to an universal unpacker, thus saving a significant amount of processing time. We show that our system achieves very high detection accuracy of packed executables with a low average processing time.

## 1. Introduction

As a consequence of the arms race between virus writers and anti-virus vendors, sophisticated code obfuscation techniques are commonly implemented in computer viruses. Executable code polymorphism, metamorphism, packing, and encryption, have been proven very effective in evading detection by traditional signature-based anti-virus software. Among these techniques, executable packing is the most common due to the availability of several open-source and commercial executable packers (Stepan, 2006). An *executable packing* tool (or packer, for simplicity) is a software that given a program $P$ generates a new program $P'$ which embeds an encrypted version of $P$ and a decryption routine. When $P'$ is executed, it will decrypt $P$ on the fly and then run it. Assuming $P$ contains known malicious code, signature based anti-virus would (likely) be able to detect it. However, if $P$ has been packed the anti-virus will try to match the signature of $P$ on $P'$. As the malicious code of $P$ is encrypted in $P'$, no match will be found. Therefore, $P$ will evade detection and infect the victim machine, if $P'$ is executed.

According to Lyda and Hamrock (2007) and Morgenstern and Brosch (2006), over 80% of computer viruses appear to be using packing techniques. Moreover, there is evidence that more than 50% of new viruses are simply re-packed versions of existing ones (Stepan, 2006). Although executable packing is very popular among virus writers, it is also applied for encrypting benign executables. Programmers of benign software apply packing to their applications mainly to make the resulting executables smaller in terms of bytes, and therefore faster to distribute through the network, for example. Also, packing makes reverse-engineering more difficult, thus making it harder for hackers to break the software license protections. As a matter of fact, there exist many commercial executable packing tools that have been developed mainly for protecting benign applications from software piracy. However, the percentage of packed benign executables is low (perhaps as low as 1%, although we were not able to find any study that can confirm this estimate, which is based solely on our experience).

*Universal unpackers* (Royal et al., 2006; Kang et al., 2007) are able to detect and extract (part of) $P$ from $P'$ without specific knowledge about the encryption algorithm used to generate $P'$. The code of $P$ is dynamically extracted by running $P'$ in an isolated environment and monitoring the execution of instructions written in memory at run-time. After $P$ has been extracted, its code can be scanned using traditional anti-virus software. It has been shown that scanning the unpacked code using signature-based anti-virus software significantly improves virus detection accuracy (Royal et al., 2006; Martignoni et al., 2007). However, universal unpackers introduce a high computational overhead, and the processing time may vary from tens of seconds to several minutes per executable. For example, the average time it takes to unpack a packed virus using the Kang et al. (2007) unpacker is around 40 s. This greatly hinders virus detection, since without a priori knowledge on the nature of the executables to be checked for malicious code all of them would need to be run through the unpacker. As a

[*] Corresponding author. Fax: +1 4049617408.
*E-mail addresses:* roberto.perdisci@gmail.com, perdisci@damballa.com (R. Perdisci), andrew@idea.sec.dico.unimi.it (A. Lanzi), wenke@cc.gatech.edu (W. Lee).

0167-8655/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.patrec.2008.06.016

consequence, scanning large collections of executables looking for virus infections may take several hours or even days.

Pattern recognition techniques have recently been proven effective in solving problems of interest in the computer security field (see (Kolter and Maloof, 2006; Wright et al., 2006; Perdisci et al., 2006), for example). For this reason, researchers in both the pattern recognition and the computer security community are gaining interest in such kind of promising applications. In this paper we present an application of pattern recognition techniques for fast detection of packed executables. The objective is to accurately distinguish between *packed* and *non-packed* executables, so that only the executables detected as *packed* will be sent to a computationally expensive universal unpacker for hidden code extraction, before being sent to the anti-virus software.

Therefore, our classification system helps in improving virus detection while saving a significant amount of processing time. In this paper we do not focus on the improvements in virus detection accuracy achieved after unpacking, because this has already been studied in (Royal et al., 2006; Martignoni et al., 2007), for example. Instead, we focus on the accuracy and computational cost related to the classification of packed executables into the two classes *packed* and *non-packed*.

Signature-based detectors of packed executables are available on the internet. For example, PEiD (http://peid.has.it) is very well known and very likely the most used. However, although signature-based detectors are fast and have relatively low false positives, they suffer from a high number of false negatives. This is mainly due to the fact that executable packing tools can be easily modified by virus writers to avoid signature-based detection (Stepan, 2006).[1] On the other hand, we will show that our classification approach has a much better generalization ability than signature-based approaches and is able to distinguish between packed and non-packed executables with very low false positive and false negative rates.

We consider programs in portable executable (PE) format, which is the format used in 32-bit and 64-bit Microsoft Windows operating systems. In order to classify an executable program, we use binary static analysis to extract information such as, for example, the name of the code and data sections, the number of writable–executable sections, the code and data entropy, etc. (see Section 3). This information allows us to translate each executable into a pattern vector. We then apply pattern recognition techniques to distinguish between *packed* and *non-packed* executables.

Fig. 1 shows how our classifier may be used to improve virus detection accuracy with low overhead, compared to a system where all the executables are directly sent to the universal unpacker. Once a PE executable is received, our classification system performs a static analysis of the PE file in order to extract a number of features, as described in Section 3. This feature extraction process can be performed efficiently. After feature extraction, the obtained pattern vector representation of the PE executable is sent to the PE file classifier. If the executable is classified as *packed*, it will be sent to the universal unpacker for hidden code extraction, and the hidden code will then be sent to the anti-virus scanner. On the other hand, if the executable is classified as *non-packed*, it will be sent directly to the anti-virus scanner. It is worth noting that the PE file classifier may erroneously label a non-packed executable as packed. In this case the universal unpacker will not be able to extract any hidden code from the received PE file. Nonetheless, this is not critical because if no hidden code is extracted, the AV scanner will simply scan the original non-packed code. The only cost paid in this case is the time spent by the universal unpacker in trying to unpack a non-packed executable. On the other hand, the PE classifier may in some cases classify a packed executable as non-packed. In this case, the packed executable will be sent directly to the anti-virus scanner, which may fail to detect the presence of malicious code embedded in the packed executable, thus causing a *false negative*. However, we will show in Section 4 that our PE file classifier has a very high accuracy and is therefore able to limit the false negatives due to these cases.

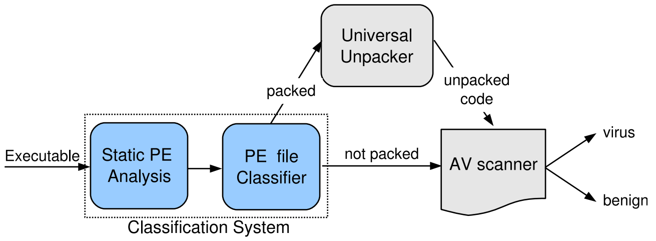

**Fig. 1.** Example of use of our classification system.

Our classification system is particularly useful in a number of applications. For example, web-sites hosting free software downloads usually receive many new executable files per day from software developers who want to make their applications available (often as either a free-ware or demo version of commercial software). Of course, these web-sites need to guarantee that the software they distribute do not contain computer viruses. To achieve this goal, before distribution every application is typically scanned using a number of different signature-based anti-virus software. However, if the virus is hiding in a packed executable, there is a good chance that it will not be detected and the web-site may become a source of infection. On the other hand, using the system proposed in Fig. 1 helps in improving virus detection with low computational overhead.

We performed experiments on 5498 executables. Of these, 2598 are *packed* computer viruses collected from the Malfease Project dataset (http://malfease.oarci.net), 2231 are benign executables extracted from a clean installation of Windows XP Home plus several common user applications, and 669 are packed benign executables obtained by manually packing applications selected from the start menu of Windows XP using 17 different executable packing tools. We show that our system achieves very high detection accuracy of packed executables with an average processing time per executable as low as 2.82 s.

The remainder of the paper is organized as follows. In Section 2 we present an overview of the related work. In Section 3 we briefly discuss the Portable Executable (PE) file format, and describe the features used for classifying PE executables. We then present and discuss the experimental results in Section 4, and summarize our work in Section 5.

## 2. Related work

In (Cohen, 1987), detection of computer viruses is shown to be undecidable both by *a-priori* and runtime analysis, and it can be shown that distinguishing between packed and non-packed executables is also undecidable (Royal et al., 2006). Although these results prove that no algorithm can detect packed executables and computer viruses with absolute precision, detection may still be performed with high accuracy, as we discuss in this paper.

In (Martignoni et al., 2007), Martignoni et al. propose OmniUnpack, an unpacking tool that monitors the execution of applications in memory and detects attempts of executing dynamically decrypted code. If such code is detected, OmniUnpack will scan it

[1] According to Stepan (2006) "modified packing tools are being created at a rate of about 10–15 per month".

using signature-based anti-virus software. If any malicious code is found the execution of the application will be stopped, otherwise it will continue until another attempt to execute decrypted code is detected. OmniUnpack is supposed to be integrated with the operating system kernel, and monitors every application executed on the machine. Similarly Royal et al. (2006) and Kang et al. (2007) present executable unpackers based on dynamic analysis of executables performed in an isolated environment (e.g. a virtual machine or an emulator). Our approach is different in that we do not need to execute an application to detect if it contains packed executable code. Our classification system extracts a number of features from executable files in PE format through static analysis, and can be used to classify packed executables without need to actually execute them.

To the best of our knowledge, the closest work to ours is Bintropy (Lyda and Hamrock, 2007), a tool for the detection of packed executables based on byte entropy analysis. Bintropy divides the PE file into blocks of 256 bytes. It then computes the entropy of each block, the average, and the maximum block entropy. Given a dataset of packed binaries, the authors use simple statistical inference to compute a 99.99% confidence interval on the value of the average block entropy and maximum block entropy (Lyda and Hamrock, 2007). Based on the lower bound of these two confidence intervals, they set two thresholds, one for the value of the average block entropy, and one for the maximum block entropy. During test, if a PE file is found to have average and maximum entropy above the respective thresholds, it will be classified as *packed*. In our work we do not limit the analysis to the PE file entropy. We introduce and motivate the use of additional features that help in distinguishing between packed and non-packed executables. Furthermore, instead of using simple statistical inference for computing confidence intervals on the entropy values, we apply statistical learning techniques to derive more accurate classification rules from a labeled dataset of packed and non-packed executables.

In (Kolter and Maloof, 2006), Kolter et al. use *n*-gram analysis to distinguish between viruses and benign executables. However, they do not distinguish between packed and non-packed executables. Also the output of their n-gram analysis classifier is not able to detect to what family of viruses a malicious executable belongs to (e.g. it cannot distinguish between different kinds of bots, like AgoBot (CA. Win32.agobot, 2006) and SpyBot (CA. Win32.sdbot, 2005). Our work is different because we focus on distinguishing between packed and non-packed executables, and then rely on the unpacker and signature based anti-virus software for distinguishing between specific kinds of viruses and benign executables.

## 3. Feature extraction

In this section we present a simplified overview of the Portable Executable format, and the intuition behind the features chosen for translating executable programs into pattern vectors suitable for classification.

### 3.1. The portable executable format

The Portable Executable (PE) format is a file format used in 32-bit and 64-bit versions of Microsoft Windows operating systems for executables, object code, and DLLs (Pietrek, 2002a,b). PE files encapsulate the information necessary for the operating system loader to manage the executable code. This includes dynamic library references for linking, Application Programming Interface (API) export and import tables, resource management data, etc. A simplified view of the PE file format is reported in Fig. 2.

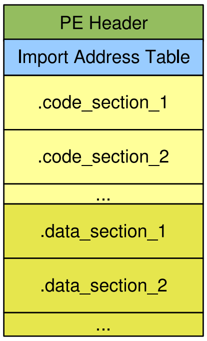

**Fig. 2.** PE file format.

The PE header instructs the operating system on how to map the executable in memory. Each code and data section in the PE file is identified by a name and marked as Readable, Writable or Executable. Usually, code sections are marked as Readable/Non-Writable/Executable, which tells the operating system that the corresponding memory locations contain executable code and write operations should be forbidden. On the other hand, data sections are usually marked as Readable/Writable/Non-Executable, and therefore the Program Counter[2] should never point to memory locations in the range of data sections.[3] For example, most PE executables contain a Readable/Executable code section named `.text` and a Readable/Writable data section named `.data`.

During execution, when a process calls a basic operating system function its Import Address Table (IAT) is used as a lookup table to resolve the address of the function to jump to. For example, when an applications calls the function `CreateWindowEx`, the operating system will lookup the IAT to find the address of the function in memory, and then execute it to create a new application window on the screen.

### 3.2. From PE file to pattern vector

Here we describe the feature extraction process we use to translate a PE file into a pattern vector. We measure the following nine features:

#### 3.2.1. Number of standard and non-standard sections

The PE file of non-packed applications usually contains a well defined set of standard sections. For example, applications compiled using Microsoft Visual C++ usually contain at least one code section named `.text`, and two data sections named `.data`, and `.rsrc` (the complete list of "standard" section names is reported in (Pietrek, 2002b)[4]). On the other hand, packed executables often contain code and data sections which do not follow these standard names. For example, the UPX executable packing tool (http://upx.sourceforge.net) usually creates PE files that contains two sections named `.UPX0` and `.UPX1`, respectively, and a section named `.rsrc`. The two sections `.UPX0` and `.UPX1` are not standard and may be used to distinguish an executable *packed* using UPX from *non-packed* executables. Besides UPX, a number of other packers usually generate PE files which contain code and data sections having non standard names. Therefore, counting how many standard

[2] The Program Counter is the CPU register that points to the next instruction to be executed.

[3] The Non-Executable flag is enforced using the NX bit in most of the 64-bit CPUs. 32-bit x86 processors do not implement the NX bit. In this cases the NX bit can be emulated by the operating system.

[4] The standard section names reported in (Pietrek, 2002b) are related to Microsoft compilers. However, the same section names are commonly found in PE files generated using other compilers.

and non standard section names are present in a PE file gives us a clue on whether the executable is packed or not.

### *3.2.2. Number of executable sections*

While analyzing the output of executable packing tools, we noticed that the PE file of some packed executables do not carry any executable section. This is obviously anomalous, because if the operating system does not allow the Program Counter to point to non-executable sections in memory the program will of course crash. This has become true since the introduction of memory protection techniques in Windows XP Service Pack 2 (Andersen, 2004). However, packed executable that do not contain any executable sections may still run correctly in older version of Windows. On the other hand, the `.text` section (i.e., the standard code section) of non-packed executables is always correctly marked as Executable.[5] Therefore counting the number of executable sections in the PE file helps in distinguishing between packed and non-packed executables.

### *3.2.3. Number of readable/writable/executable sections*

Assume we execute a packed executable $P'$ which hides an encrypted program $P$. When executed, $P'$ will first activate an unpacking routine in order to decrypt $P$, and then will execute it. The unpacking process entails writing decrypted code (i.e. $P$) in an executable section of the memory image of $P'$. Therefore, the PE file of $P'$ needs to include at least one section which is Readable/Writable/Executable at the same time. On the other hand, the executable sections (usually the `.text` section) in the PE file of non-packed applications do not need to be writable, and the Writable section flag is not set. Therefore, counting the number of sections which are writable and executable at the same time adds a piece of evidence to the conclusion whether the executable is packed.

### *3.2.4. Number of entries in the IAT*

The Import Address Table (IAT) of a PE executable contains the address of the external functions called by the application. These external functions are imported from Dynamic Linked Libraries (DLL). Each imported function has an address in the IAT which is written by the operating system loader after the application is launched and the PE file is mapped into memory. Every time the application calls an external function, the IAT is queried in order to resolve its address in memory.

Most non-packed executables import many external functions. For example, they usually import many functions from the native Windows API, which are used to read/write form/to files, open new windows on the screen, manage a network connection, and so on. Therefore, the IAT will usually contain many entries, one per each imported function. On the other hand, packed executable often import very few external functions. The main reason is in that the unpacking routine does not need many external functions. The basic operations the unpacking routine performs are read and write memory locations in order to decrypt the code of the packed application on the fly. For example, no window on the screen or network operation is usually needed. This is reflected in a small number of entries in the IAT of a packed executable.[6]

### *3.2.5. PE header, code, data, and file entropy*

The encrypted code of an application $P$ packed (i.e. hidden) into $P'$ is usually stored in a code or data section of the PE file (we identify a section as a code section if the Executable section flag is set, otherwise we consider the section to be a data section). As the code of $P$ is usually somehow encrypted, it will look like "random", loosely speaking. On the other hand non encrypted code sections contain well "structured" information, namely the opcode of executable instructions and the memory location of the operands. Non-encrypted data sections also contain somehow structured information. Following this observation, we measure the byte entropy of the code and data sections in the PE file. If the entropy of a section is close to 8 bits, which is the maximum byte entropy, the section likely contains encrypted code.

The code and data sections are not the only places where the executable packing tool may hide the code of the original application. There are parts of the PE header dedicated to optional fields that are not necessary for the correct loading of the program into memory by the operating system. Some packing tools may therefore hide encrypted code in those unused portions of the PE header. For this reason we measure the byte entropy of the PE header as well. Considering that the PE file is quite complex and contains other such unused spaces (for example, portions of the header of each section), the encrypted code may be hidden in several other locations (Lyda and Hamrock, 2007). Therefore, we also measure the entropy of the PE file as a whole to take into account these cases.

## 4. Experiments

We performed experiments on 5498 executables in PE format. We collected 2598 packed viruses[7] from the Malfease Project dataset (http://malfease.oarci.net), and 2231 non-packed benign executables[8] collected from a clean installation of Windows XP Home plus several common user applications. Also, we generated 669 packed benign executables by applying 17 different executable packing tools freely available on the Internet to the executables in the Windows XP start menu. Of the 3267 packed executables in our collection, PEiD (http://peid.has.it), (very likely) the most used signature-based detector for packed executables, was able to detect only 2262 of them, whereas 1005 remained undetected. This means that PEiD had a false negative rate of 30.8%. Among the 1005 undetected packed executables, there were 604 packed viruses and 401 of our "manually" packed benign.

We developed a PE format analysis tool written in Python, which extracts the 9 features described in Section 3 and summarized in Table 1.

We applied our tool to each of the executables in our collection, therefore obtaining a labeled dataset with 5498 entries. We divided our dataset in two parts: (1) a training dataset containing 2231 patterns related to the non-packed benign executable and 2262 patterns related to the packed executables detected using PEiD; (2) a test dataset containing 1005 patterns related to the packed executables that PEiD was not able to detect. These datasets and the feature extraction tool we used to generate them are available at http://roberto.perdisci.googlepages.com/code.

In order to perform experiments with different learning algorithms we used Weka (http://www.cs.waikato.ac.nz/ml/weka). Table 2 reports the results obtained with 6 different classifiers, namely

[5] This is true unless a "non standard" compiler is used to generate the PE file.

[6] The original (packed) application will likely need to perform operations on the screen and network. Therefore, the unpacking routine will usually overwrite the IAT in memory on the fly to add the missing IAT entries.

[7] Royal et al. (2006), a universal unpacker, was used to confirm that the collected viruses were actually packed.

[8] Strictly speaking, determining whether an executable is packed or not is an undecidable problem (Royal et al., 2006). However, we checked the 2231 benign executable with PEiD to verify that none of them were packed. Although PEiD suffers from false negatives, given the benign nature of the executables we assume none of them has been packed with unknown or modified executable packing tools that cannot be detected using PEiD's signatures.

**Table 1**
Summary of the features extracted from PE files

| Feature | Range of values |
|---|---|
| Number of standard sections | integer $\geqslant 0$ |
| Number of non-standard sections | integer $\geqslant 0$ |
| Number of Executable sections | integer $\geqslant 0$ |
| Number of Readable/Writable/Executable sections | integer $\geqslant 0$ |
| Number of entries in the IAT | integer $\geqslant 0$, or $-1$ if the PE file has no IAT |
| Entropy of the PE header | [0,8] |
| Entropy of the code sections | [0,8], or $-1$ if the PE file has no code section |
| Entropy of the data sections | [0,8], or $-1$ if the PE file has no data section |
| Entropy of the entire PE file | [0,8] |

**Table 2**
The first two columns report the average and standard deviation of the AUC and accuracy computed over 10 rounds of 10-fold cross-validation

| Classifier | CV AUC | % CV accuracy | % Test accuracy |
|---|---|---|---|
| Naive Bayes | 0.9917 (0.0038) | 98.42 (0.4913) | 97.11 |
| J48 | 0.9958 (0.0034) | 99.57 (0.3032) | 97.01 |
| Bagged-J48 | 0.9994 (0.0010) | 99.59 (0.3086) | 96.82 |
| IBk | 0.9994 (0.0008) | 99.43 (0.3486) | 95.62 |
| MLP | 0.9995 (0.0008) | 99.42 (0.3966) | **98.91** |
| Entropy threshold | *0.9766* | *96.57* | 91.74 |

The test accuracy (last column) refers to the percentage of packed executables not detected by PEiD that were correctly detected by each classifier.

(a) Naive Bayes classifier.
(b) J48 decision tree (Weka's implementation of the well known C4.5).
(c) Ensemble of unpruned J48 decision trees constructed using Bagging and the Laplace correction for probability estimates.
(d) k-nearest-neighbors classifier (IBk is Weka's implementation of the kNN algorithm) with $k = 3$ and instance weights equal to the inverse of the distance from the neighbors.
(e) Multi Layer Perceptron (MLP) classifier constructed using 1 input layer with 9 nodes (one for each feature), 1 hidden layer with 5 nodes, and 1 output layer with 2 nodes (one for each class). All the hidden and output nodes use a sigmoidal activation function, whereas the input nodes use a linear activation function. The backpropagation algorithm was used for training, with 20% of training patterns reserved for validation.
(f) *Entropy Threshold* classifier, which classifies an executable according to a simple threshold on the value of the PE file entropy. That is, given a PE file *F*, if its file entropy is greater than a predetermined threshold *F* will be classified as *packed*, otherwise it will be classified as *non-packed*.

The first two columns of Table 2 were computed by repeating 10-fold cross-validation 10 times on the training dataset and then averaging over the obtained 100 results (the standard deviation is reported between parentheses). For the *Entropy Threshold* classifier we did not apply cross-validation. We computed the AUC by simply considering the value of the file entropy as a score and by computing the Wilcoxon–Mann–Whitney statistic (Cortes and Mohri, 2004). The accuracy (second column) was computed by finding the threshold $t_h$ that gives maximum separation between the packed and non-packed classes in the training dataset. We found that $t_h = 6.707$ gives maximum separation, and the related accuracy reported in the table is therefore the maximum value of the percentage of correctly classified patterns in the training dataset.

The *Test Accuracy* (the last column) reports the results on the test dataset, i.e., on the 1005 packed executables that were not detected by PEiD. We first trained each classifier on the training dataset, and then computed the accuracy on the test dataset. For the *Entropy Threshold* classifier, we used the threshold $t_h = 6.707$.

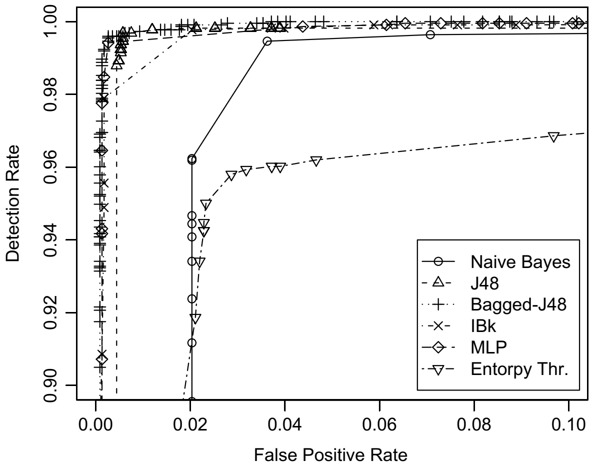

**Fig. 3.** ROC curves. The scale has been adjusted to highlight differences among classifiers.

Fig. 3 shows the ROC curves computed by 10-fold cross-validation. The range of false positives and false negatives has been cropped in order to highlight differences among the classifiers.

As we can see, Bagged-J48, IBk and MLP give nearly perfect classification results, according to the cross-validation experiments on the training dataset. A comparison of these three algorithms using paired *t*-test on the values of the AUC showed no statistically significant difference among them. On the other hand, according to the paired *t*-test the Naive Bays and the J48 classifiers performed significantly worse (although still very well in absolute terms), compared to Bagged-J48, IBk, and MLP. It easy to see that the Entropy Threshold classifier has much poorer performance, compared to all of the other classifiers. We conclude that although the file entropy is a discriminant feature, using all the nine features described in Section 3 significantly improves the classification of packed executables.

We can consider the *Test Accuracy* (last column in Table 2) as an estimate of the generalization ability of our classifiers, compared to the signature-based approach implemented by PEiD. As we can see, all the classifiers (excluding the Entropy Threshold classifier) correctly detected more than 95% of the packed executables that were not detected by PEiD. Even though the Naive Bayes and J48 algorithms performed slightly worse than Bagged-J48 and IBk in terms of AUC, they perform better on the test dataset. The best results were obtained by using the MLP classifier, which performs as well as Bagged-J48 and IBk in terms of AUC, and gives the best accuracy (reported in bold in Table 2) on the test dataset.

The average time to extract the features used for classification on a 2 GHz Dual Core AMD Opteron processor was about $t = 2.82$ s per executable. We believe *t* can be made even much smaller by developing an optimized feature extraction tool written in C, instead of using Python. The time needed for the classification of a pattern vector is in the order of $10^{-3}$ s, and therefore negligible compared to *t*.

## 5. Conclusion

Executable packing tools are extensively used by computer virus writers to hide malicious code into packed executables that may not be detected using traditional signature-based anti-virus software. Universal unpackers may be used to extract the hidden code from packed executables. After unpacking, the hidden code can be checked against signatures of known viruses, thus improving the capability of detecting hidden malicious code. However, universal unpackers introduce a very high computational overhad, because the analysis of any executable (packed or not) may take from tens of seconds to several minutes. This means that scanning large collections of executables looking for virus infections may take several hours or even days.

In this paper we proposed a technique for fast detection of packed executables. Only the executables that are classified as packed by our classifier will be analyzed using a universal unpacker for extracting the hidden code, and afterwards scanned by an anti-virus software, therefore improving virus detection accuracy while saving a significant amount of processing time. We described how to extract discriminant features from executable files in Portable Executable format, and we showed that our classification system achieves very high accuracy while keeping the processing time per executable low.

## Acknowledgments

We would like to thank Dr. Giorgio Giacinto and the anonymous reviewers for their helpful comments. This material is based upon work supported in part by the National Science Foundation under Grant No. 0716570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

Andersen, S., 2004. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. <http://technet.microsoft.com/en-us/library/bb457155(d=printer).aspx>.

CA. Win32.sdbot Family (2005). <http://ca.com/us/securityadvisor/virusinfo/virus.aspx?ID=12411>.

CA. Win32.agobot Family (2006). <http://www.ca.com/us/securityadvisor/virusinfo/virus.aspx?id=37776>.

Cohen, F., 1987. Computer viruses: theory and experiments. Comput. Security 6 (1), 22–35.

Cortes, C., Mohri, M., 2004. Confidence intervals for the area under the roc curve. In: NIPS 2004: Advances in Neural Information Processing Systems.

Kang, M.G., Poosankam, P., Yin, H., 2007. Renovo: A hidden code extractor for packed executables. In: WORM'07: Proceedings of the 5th ACM Workshop on Recurring Malcode.

Kolter, J.Z., Maloof, M.A., 2006. Learning to detect and classify malicious executables in the wild. J. Mach. Learn. Res. 7, 2721–2744.

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

Martignoni, L., Christodorescu, M., Jha, S., 2007. Omniunpack: fast, generic, and safe unpacking of malware. In: ACSAC'07: Proceedings of the 23rd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference.

Morgenstern, M., Brosch, T., 2006. Runtime Packers: The Hidden Problem? Presented at Black Hat USA 2006.

Perdisci, R., Gu, G., Lee, W., 2006. Using an ensemble of one-class SVM classifiers to harden payload-based anomaly detection systems. In: ICDM'06: Proceedings of the Sixth International Conference on Data Mining.

Pietrek, M., 2002a. An In-depth Look into the Win32 Portable Executable File Format. <http://msdn.microsoft.com/msdnmag/issues/02/02/PE/>.

Pietrek, M., 2002b. An In-depth Look into the Win32 Portable Executable File Format, part 2. <http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/>.

Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W., 2006. Polyunpack: automating the hidden-code extraction of unpack-executing malware. In: ACSAC'06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference.

Stepan, A., 2006 Improving Proactive Detection of Packed Malware. <http://www.virusbtn.com/virusbulletin/archive/2006/03/vb200603-packed.dkb>.

Wright, C.V., Monrose, F., Masson, G.M., 2006. On inferring application protocol behaviors in encrypted network traffic. J. Mach. Learn. Res. 7, 2745–2769.