DETECTING PACKED EXECUTABLES BASED ON RAW BINARY DATA

Lakshmanan Nataraj[a], *Grégoire Jacob*[b], *B.S. Manjunath*[a]

[a]Dept. of Electrical and Computer Engineering, [b]Dept. of Computer Science,
University of California, Santa Barbara.

ABSTRACT

Packing an executable originally referred to the compression of the file to reduce its size on disk. Nowadays, packing also introduces encryption and anti-debug techniques to protect executables from reverse engineering. This explains why packers are extensively used in creating new malware variants which are not detected by traditional signature-based anti-malware tools. Although universal unpackers exist for extracting the executable code from packed files, they often rely on methods based on dynamic analysis, thus making them computationally expensive and time consuming. Hence, it is important to detect packed executables beforehand to avoid unnecessary computations so that only protected executables need be sent to the unpacker before further analysis.

In this paper, we propose a new technique for fast identification of packed executables by analyzing only the raw binary data. We extract bigram-based features on packed and unpacked executables and use a support vector machine for training and testing. Experimental results reveal that we are able to correctly identify packed executables with a high detection rate in the range of 95%-98% for a variety of packers and crypters.

Index Terms—Malware, Detecting Packed Executables, Bigrams-based Classification.

1. INTRODUCTION

Cyber Security is facing a serious threat in this new age. Everyday we observe many security breaches, a lot of them leading to the downloading of malware without the user's consent. The number of malware being generated currently is growing exponentially. At present, most prevalent malware are remotely controlled bots, spyware, adware which support organized world-wide criminal activity over the internet. The inability of anti-virus software to reliably protect computers and networks against the continuous stream of malware threats is well known. These software use traditional signature-based methods to identify if a program is malicious or not. However, they are not effective due to the huge bundle of newly generated malware. Further, most of the malware undergo packing (compression and/or encryption) which changes the appearance of the original malicious binaries thus evading detection. In Symantec's Global Internet Security threat report released in April 2010 [8], it is reported that Symantec generates thousands of new signatures every year- 2,895,802 in 2009 as compared to 169,323, in 2008. However, not all these signatures can be attributed to new malware. As mentioned in [9] more and more malware are being packed, thus degrading the performance of traditional anti-virus software, since they are not effective in unpacking a protected executable. An executable can be packed using methods such as compression, encryption or a combination of both. Packers transform the original executable's binaries to a different form and generate new executables by embedding the compressed, possibly encrypted code with an appended loading routine. This routine is responsible for the automated decompression and decryption of the code before executing them. There are hundreds of packers that exist today, both commercial and open source. This makes it very easy for malware writers to create new malware variants which are not detectable by traditional anti-virus software. Hence, these software resort to either creating a new signature for every new packed threat it encounters or try to emulate the executable code and then scan the image of the code in memory. In general, it is believed that nearly 80% of malware are packed [1], [9] and 50% of existing malware are packed versions of old malware [5].

In our work, we focus on identifying if a given executable is packed or not. We only use the raw binary information to extract features that can effectively distinguish between packed and unpacked executables. In our method, decoding an executable's instructions is not necessary to determine if the file is packed or not. When analyzing large collections of malware samples, our algorithm can quickly tell which samples are packed or encrypted. Based on this information, only packed/encrypted samples that have to be unpacked need be sent to the un-packer before further analysis.

The main contribution of this paper is that we propose a novel method to analyze executables to tell whether they are

packed or not. We explore the statistical co-occurrence of byte codes in executables to distinguish packed and unpacked files. We show that our method achieves higher accuracy than other current methods based on entropy analysis [1] and higher robustness than methods that explore the structural information of portable executables [2].

The rest of the paper is organized as follows. In Sec. 2, we briefly review some of the existing methods which detect packed executables without going inside the actual program structure. In Sec. 3, we describe our methodology to classify packed executables from unpacked executables. The experiments and results are discussed in Sec. 4 and the paper concludes in Sec. 5.

2. RELATED WORK

Generic unpacking is a very active field of research [9], [10]. The main solution that these methods adopt is performing dynamic execution in a virtual environment. But our method focuses on determining if a given executable is packed or not. In our method, there is no need to execute the code beforehand. We extract features only from the raw binary data to tell packed from unpacked executables. Hence, our method falls under "blind" methods since no actual disassembling of executable code is necessary. To the best of our knowledge, only a few such "blind" methods exist in literature and they are also recent. We will review some of them below.

Perhaps the first work that was published to distinguish packed executables from unpacked ones was by Lyda and Robert [1]. They use entropy analysis on blocks of raw byte codes to determine the statistical variation between packed and unpacked executables. Their method works by dividing an executable into blocks of 256 bytes. They then use the entropy of the blocks to find variations. Since many blocks contain a lot of zeros, the entropy in these blocks reduces. Hence, the authors only consider blocks in which at least half the bytes are non-zero. The entropy is then computed for the valid blocks from which the average entropy of all these valid blocks and the maximum block entropy are found. Based on these two parameters, the authors perform simple statistical tests to obtain two thresholds to distinguish packed from un-packed files. In [4], Ebringer et al also use entropy analysis to classify different family of packers.

In contrast to [1], Perdisci et al [2] use a pattern recognition based approach to classify packed executables from unpacked executables. Similar to the above papers, they consider executables in portable executable (PE) format, which is the format used in most Microsoft Windows operating systems. They perform binary static analysis of a PE file and extract nine features: number of standard and non-standard sections, number of executable sections, number of readable/writable/executable sections, number of entries in the Import Address Table (IAT), and the entropies of PE header, code section, data section and the entire PE file. They then train and test these features on a labeled dataset using various machine learning classifiers and show that they obtain an accuracy of close to 95%. In [3], Shafiq et al use the above technique as a first step to classify packed and unpacked executables and then use this output for malware detection. Although this method works fairly well, the problem with PE feature is that they can be easily modified to mimic that of a normal executable. For example, one can mimic standard section names, modify the protection of sections at runtime or also build a fake Import Address Table. However, for our method, we neither need any PE information nor do we need to compute entropy. Our method relies on statistical properties of the data which is harder to tamper with.

3. DISTINGUISHING PACKED EXECUTABLES

In order to classify packed executables from unpacked executables, we use features based on n-grams of the executable's byte codes. Detecting malware based on n-grams analysis of bytes has been shown to be effective in telling malware from clean files. In one of the seminal papers [6], Koetler and Maloof gathered 1,971 benign and 1,651 malicious executables and extracted n-grams of byte codes as features. On various machine learning techniques, they show that they obtained very high detection rates to classify malware from clean files. Similarly, Moskovitch et al [7] also use n-gram based features to classify malware from benign files. However, in these works, they do not separate the packed files from unpacked files before processing.

3.1. Feature Extraction

Motivated by the above approaches, we also use n-grams based features to distinguish packed from unpacked executables. The reason for using n-grams is that packing often shifts the bits or transforms the binaries to a different form which in turn alters the n-grams of byte codes. Further, packing also usually reduces the size of an executable, due to which the number of distinct n-grams present in the executable reduces after packing. Instead of computing the actual n-grams vector which depends on the size of a file, we compute the distribution of the n-grams which is vector of constant size. Further, there is no need to compute the actual n-grams in order to compute the distribution. To illustrate better, consider an example where an executable file is read in hexadecimal form in to a vector 'hex', whose length will be equal to the total number of bytes. Let the first few entries in the byte sequence of 'hex' be '0a 1b c4 8a'. Then, the corresponding bigrams will be 0a1b, 1bc4, c48a. Let 'disbn' be the zero vector corresponding to the distribution of the bigrams. Since the total number of possible bi-grams varies

from '0000' to 'ffff', the size of this vector is 65,536. As shown in Fig.1, the computation of the distribution does not need the actual computation of bigrams vector.

```
hex // hex string of chars of input file
disbn // zero array of size 1x65536
for i=0:2:# of bytes -4
   temp = hex(i:i+3);  // Example 'ffff'
   dec = hex2dec(temp) ; // Example 65535
   disbn(dec) +=1; // increment the count
end
```

Fig.1 Pseudo code to compute distribution of bi-grams

values in descending order. Due to sorting, we lose the location of bi-grams, although the variation between different bi-grams is captured. This helps us to distinguish between unpacked and packed files as shown in Fig.2 which shows the feature vectors for unpacked, packed and encrypted executable. While pruning the feature, the top few values are not considered since they are usually the same for all files. After discarding the top few values, the size of the feature vector is reduced to a manageable size.

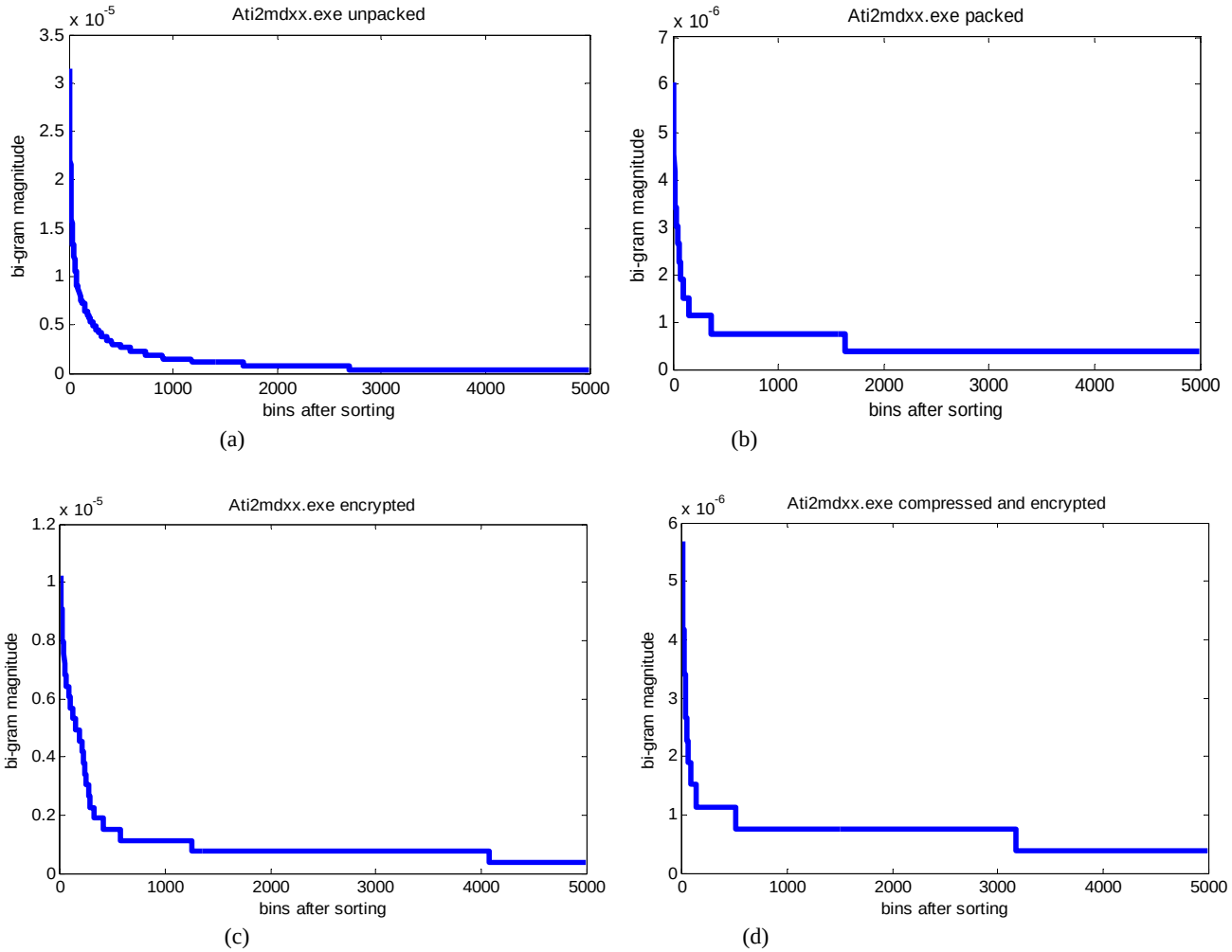

Fig. 2 Bigram-based features computed on different versions of an executable: (a) unpacked (b) packed (UPX) (c) encrypted (Yoda) (d) compressed and encrypted (Telock)

3.2. Feature Pruning

As mentioned earlier, we compute the distribution of the n-grams of byte codes. We choose bigrams since unigram does not capture all the variations and higher n-grams not only takes time to compute but their feature vector size is also large. Once we find the distribution of bi-grams, we sort the

3.3. Algorithm and Classification

The algorithm to compute the feature is as follows:

1. Read the .exe file in hexadecimal format.
2. Compute the distribution of the bi-grams to obtain a vector of size 1x65536.

3. Sort the distribution in descending order.
4. Remove the top few entries.
5. Truncate the size of the vector to a reduced size to obtain the feature vector.

We compute the features on a labeled dataset of unpacked and packed executables. We then use a support vector machine (SVM) with radial basis kernel for training and testing.

4. EXPERIMENTS

In our experiments, we use 300 unpacked files taken from the "systems" folder of a Windows XP Service Pack 2 operating system and pack/encrypt them using a host of packers (*UPX, NsPack, Upack, PeCompact, FSG, MEW, AsPack*) and crypters (*Telock, Polyene, Yoda*). For every packer/crypter, we get an additional set of 300 files; thus the total number of packed/encrypted files is 3000. We compute the feature vector for every executable as described in Sec. 3. We exclude the top 10 terms after sorting the bigrams distribution and then take the next 5000 terms as the feature vector as shown in Fig. 2. For our experiments, we train and test these features for every packer family. Hence for every packer, there are totally 600 executables of which 300 are packed and 300 are unpacked. Among these files, we take 350 files for training and 250 files for testing and use an SVM with radial basis kernel for classification. The classification accuracies obtained for training were in the range 0.98-1.0. The testing results are tabulated in Tab. 1. We observe that we obtain very high accuracy in the range of 0.95-0.98. Among packers, the best results were obtained for executables that were packed using *NsPack* packer and among crypters, the best results were obtained for *Telock*, although the results for other packers/crypters were not far behind. We compared our results with the entropy tests as mentioned in [1]. We re-implemented their method in which they divide the data stream into blocks of 256 bytes and compute the entropy for every block. Since many blocks contain a large number of zeros, they consider only blocks in which at least half the number of bytes are non-zero. Once this is done, they compute the average entropy over all the blocks and the maximum entropy. Based on these parameters, they conduct statistical tests to obtain thresholds which can distinguish unpacked files from packed or encrypted files. In [1], the authors state that executables with average block entropy greater than 6.677 and maximum entropy greater than 7.199 are statistically likely to be packed or encrypted. However, when we re-implemented their method, we observed that executables whose average block entropy is greater than 5.9 and maximum block entropy is greater than 7 are statistically likely to be packed or encrypted and the detection rates for the thresholds used in [1] were lower. The reason for this difference could be the fact that the authors in [1] used a smaller dataset with just two packers and one crypter. In Fig. 3 we show that our method outperforms the entropy based method of [1].

Packer / Crypter	Detection Rates	True Positives	False Positives
UPX	0.9769	0.9630	0.0093
Nspack	**0.9815**	0.9722	0.0093
Upack	0.9698	0.9341	0
Pecompact	0.9773	0.9643	0.0093
FSG	0.9773	0.9643	0.0093
MEW11	0.9772	0.9772	0.0278
Aspack	0.9722	0.9630	0.0185
Telock	**0.9857**	0.9714	0
Polyene	0.9454	0.9000	0.0093
Yoda	0.9495	0.9273	0.0278

Tab.1 Packer Detection results for various packers

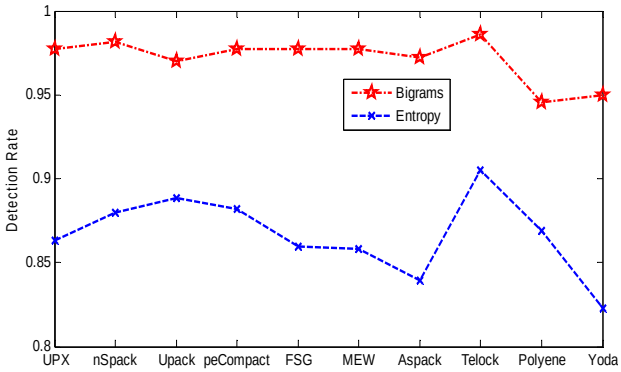

Fig. 3 Detection rates for bigrams-based method and entropy method [1]

In our previous experiments, we trained and tested the features for every family of packers and crypters. This means that we have to store support vectors for every family which is not preferable in a realistic scenario. To see if our method is more general, we trained with one packer at a time and tested against the packed executables generated by other packers/crypters and computed a confusion matrix as shown in Tab. 2. We observe that even if we train our feature on packed executables generated by one family of packers, the classifier is able to detect packed executables from other family of packers. Among the different packers/crypters, the classification results were comparatively low only for executables encrypted using Yoda Crypter. This is because Yoda does not pack the data and only performs encryption on the executable. Hence, files encrypted by Yoda do not show strong variations when compared with other packed

files. Next, we mixed packed files from various packers to obtain a "mixed bag" of 300 packed samples which we used for training. We tested it on all the packed executables and were able to obtain high accuracies (Tab. 2). For all the results reported in Tab. 2, the false positive was as low as 0.01. We can also see from Tab. 2 that the best results were obtained for the classifier which uses packed samples from Polyene.

We also evaluate our method on a small set of non-system executables and malware. In particular, we take 261 unpacked executables that are programs, applications, etc. and 207 unpacked botnets obtained from Anubis [11]. We packed them using UPX to generate 468 packed samples and run our classifier on the 936 unpacked and packed samples. The overall detection rate we obtained was 0.9542 with a false positive of 0.015. We believe that the results would be similar if we used other packers/crypters.

Finally, we apply our algorithm to malware in the "wild". We analyzed 8192 malware collected from Anubis [11]. We do not know beforehand if these malware were packed or not. On applying our algorithm, we found that 4649 malware were likely to be packed and 3543 are likely unpacked. This corresponds to a packing percentage of 56.75%.

5. CONCLUSION

In this paper, we propose a novel method to distinguish packed executables from unpacked executables by just using the raw binary data. We show that features based on bigrams of byte codes can effectively distinguish packed from unpacked executables. In contrast to the current methods that detect packed files without decoding the data, our results indicate that we are able to obtain higher accuracy when compared to the entropy based method and also better robustness when compared to the method that uses PE information, which can be easily tampered with. With more and more malware being packed, this method will particularly be useful in computer security applications as a first block in telling whether an executable is packed or not and based on this information, further analysis can be carried out. In future, we will focus on classifying various packers that generate packed executables.

6. ACKNOWLEDGEMENTS

We would like to thank Prof. Giovanni Vigna and Prof. Christopher Kruegel for their timely help and support.

7. REFERENCES

[1] R. Lyda and J. Hamrock, "Using entropy analysis to find Encrypted and packed malware", IEEE Security and Privacy, v.5 n.2, pp. 40-45, March 2007.

[2] R. Perdisci, A. Lanzi, and W. Lee, "Classification of packed executables for accurate computer virus detection," *Pattern Recognition Letters*, Elsevier, v. 29, n.14, pp. 1941-1946, 2008.

[3] M.Z. Shafiq, S.M. Tabish, and M. Farooq, "PE-Probe: Leveraging Packer Detection and Structural Information to Detect Malicious Portable Executables," *Virus Bulletin Conference*, 2009.

[4] T. Ebringer, L. Sun, and S. Boztas, "A fast randomness test that preserves local detail," *Virus Bulletin Conference*, Oct. 2008.

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

[6]. J.Z. Kolter, M.A. Maloof, "Learning to detect and classify malicious executables in the wild". Journal of Machine Learning Research v. 7, pp. 2721–2744, 2006.

[7]. Moskovitch et al, "Unknown malcode detection and the imbalance problem", Jour. in Computer Virology, v.5, pp. 295-308, 2009.

[8]. Symantec Global Internet Security Threat Report, April 2010, <http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xv_04-2010.en-us.pdf>,2010.

[9]. F. Guo, P. Ferrie, T. Chiueh, "A study of the packer problem and its solutions," *Recent Advances in Intrusion Detection (RAID)*, 2008.

[10]. P. Royal, M. Halpin, D. Dagon, R. Edmonds and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," *ACSAC 2006*,pp. 289-300, 2006.

[11]. Anubis: Analyzing Unknown Binaries, < http://anubis.iseclab.org/>

	UPX	Nspack	Upack	Pecompact	FSG	MEW	Aspack	Telock	Polyene	Yoda
UPX	**0.9628**	0.9740	0.9598	0.9654	0.9406	0.9462	0.9519	0.9790	0.9167	0.8789
Nspack	0.9405	**0.9759**	0.9537	0.9581	0.9147	0.9165	0.9407	0.9790	0.9074	0.8661
Upack	0.9405	0.9722	**0.9638**	0.9654	0.9258	0.9239	0.9463	0.9809	0.9093	0.8569
Pecompact	0.9517	0.9759	0.9658	**0.9709**	0.9239	0.9239	0.9537	0.9809	0.9167	0.8624
FSG	**0.9684**	0.9629	0.9658	0.9617	**0.9647**	0.9629	0.9481	0.9733	0.9389	0.8642
MEW	0.9610	0.9740	**0.9718**	0.9672	0.9536	**0.9703**	0.9500	0.9714	0.9352	0.8587
Aspack	0.9424	0.9722	0.9577	0.9559	0.9239	0.9239	**0.9630**	0.9771	0.9111	0.8697
Telock	0.9052	0.9443	0.9437	0.9308	0.8813	0.8757	0.9242	**0.9847**	0.8926	0.8367
Polyene	0.9554	**0.9814**	0.9658	**0.9709**	0.9388	0.9462	0.9519	0.9809	**0.9426**	0.8826
Yoda	0.9257	0.9536	0.9457	0.9472	0.9109	0.9054	0.9519	0.9790	0.9204	**0.9358**
Mixed	**0.9405**	**0.9647**	**0.9598**	**0.9526**	**0.9221**	**0.9128**	**0.9444**	**0.9790**	**0.9148**	**0.8807**

Tab.2 Confusion Matrix showing detection rates of packed executables from different family of packers/crypters. Each row corresponds to training using the packer in column 1 and testing with the different families of packers. The rates in 'bold' indicate the best detection rate for a packer family or the rates obtained using same packers for training and testing. It can be seen that for some families (UPX, Nspack, Upack), features trained using a different family gives higher rates. The last row corresponds to training using a mixed bag of packed sampled and testing against all the packed executables.