Packer Identification using Grayscale Images of Binaries

Pierre Mondon
School of Computing
University of Kent, UK

Rogério de Lemos
School of Computing
University of Kent, UK

***Abstract*—Malware frequently uses packing techniques, making the analysis of packed executables essential for developing robust malware defences. This paper focuses on the identification of packers used in MS Windows binaries. Our approach first converts these binaries into grayscale images of 128 pixels in width and arbitrary height, and then applies convolutional neural networks (CNN), as a sliding window, along the vertical dimension of the image for extracting a sequence of high-level features. Finally, this sequence is fed into a recurrent neural network (RNN) that classifies the binary image. Using a CNN to extract features eliminates the need for manual feature engineering, which is time-consuming and resource-intensive. An advantage of our approach is that we do not rely on code analysis, hence removing dependencies on third-party software, which results in our method being able to handle all files. Another advantage of our approach, compared with traditional CNNs, is that it does not require image resizing, which prevents information loss. Overall, our results can improve the state-of-the-art by up to 23.506% while removing third-party dependencies on external software to analyse code.**

***Index Terms*—malware, packer, binary analysis, obfuscation, deep learning**

I. Introduction

Obfuscation hardens the understanding of a program while preserving its logic. The most common obfuscation technique is packing, and it has been estimated that 92% of malware is packed [6]. Three main research areas are tackling packed executables: detection, identification and unpacking. Packer detection is concerned with determining whether a binary is packed or not. Packer identification determines the specific packer used to pack an executable. Finally, unpacking aims to restore the original code prior to packing. This paper targets packer identification.

Packing is a type of obfuscation that embeds the original code of the program by either encrypting or compressing it. More precisely, a packer takes as input a binary A, and produces an intermediate executable A_p by encrypting, compressing it, or both. The packed executable A_p is then embedded in a new executable B that is composed of the packed binary and an unpacking stub U. The unpacking stub is responsible for unpacking the program A_p at run-time before A can be executed. Since A is no longer visible in binary B, it is impossible to statically analyse A without first unpacking it. The static analysis of the packed executable is restricted to analysing its unpacking stub U.

There are two main types of malware analysis techniques: static and dynamic. Static analysis is based on properties extracted or generated from an executable file, without performing execution. For example, by disassembling a binary and generating its control flow graph. On the other hand, dynamic analysis requires the execution of the binary and is dependent on the execution environment. Static analysis is computationally better than dynamic analysis, but is vulnerable to binary packing [20]. On the other hand, dynamic analysis is vulnerable to different obfuscation techniques such as process handle enumeration [9, 10]. It is currently accepted that both types of analysis can be complementary, and do not need to be seen as purely competitive [8].

Often, packers include additional obfuscation techniques in the unpacking stub, which can remove or alter features used in the identification of packers. For instance, packer identification techniques based on control flow graphs are vulnerable to code flattening [24]. This motivates the need for new packer identification techniques, rather than relying on analysis techniques known to be vulnerable to code obfuscation.

In this paper, we address the problem of packer identification based on static analysis of binary. We present an approach based on grayscale representation of binaries [21] from which we extract features with a convolutional neural network (CNN). We classify the extracted features with a recurrent neural network (RNN), specifically a bidirectional short-term memory (Bi-LSTM) layer.

Figure 1 shows six grayscale images representing the same binary packed with different packers. As this approach does not require execution of the binary, image based analysis of binaries can be classified as static analysis. Other approaches based on binary analysis tools [18] require manual feature engineering. Moreover, some of these binary analysis tools are unable to perform a full analysis of the binaries. In contrast, our approach can process 100% of the binaries. Additionally, compared with other image-based methods, our method does not require resizing the images [28], which ensures that no information is lost.

The evaluation of the proposed method is performed against two identification techniques, i.e., 2-spiff [18] and PEiD [25], and for that we rely on an online available dataset of packed binaries[1] that uses 25 different packers. This dataset is a

[1] https://github.com/packing-box/dataset-packed-pe

sanitised version removing false labels and duplicates from this paper [7].

The key contribution of this paper is a packer identification method based on grayscale representation of binaries, which uses a CNN to extract features from the binary image using a sliding window algorithm, and an RNN to classify the extracted features. The proposed approach is evaluated by comparing it against other packer identification techniques. Our approach, which allows 100% of the binaries to be processed, can improve state-of-the-art methods by up to 23.506%, while removing the need for feature engineering.

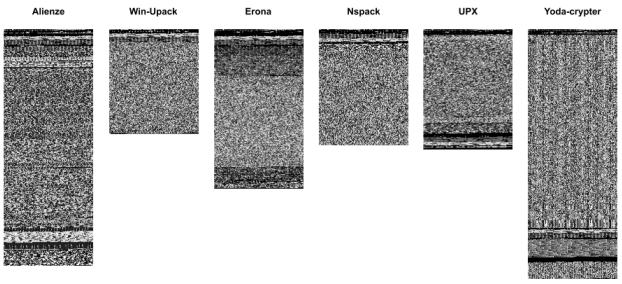

Fig. 1. Grayscale Images of ftp.exe for different packers.

II. Related work

Most work in packer identification focuses on features manually selected from static or dynamic analysis [20], without covering image-based approaches. We broaden our review to include image representation of binaries for malware classification, as well as existing packer identification methods.

A. Image representation of binaries for malware classification

Nataraj et al. [21] introduced the idea of representing binary executables as images, inspiring subsequent approaches [11, 27, 29]. A key challenge is that binary image size varies with the original file size, conflicting with the fixed input requirements of CNNs. Solutions include re-scaling [11, 27, 29] or extracting fixed-size feature vectors using wavelet, Gabor, or histogram-based features [2, 13, 19]. Re-scaling, however, risks information loss and image distortion leading to misclassifications. Yakura et al. [29] introduce an attention mechanism for malware family classification, enabling localization of relevant binary regions, but still rely on image rescaling.

B. Packer identification

Packer detection determines whether an executable is packed, while identification determines which packer was used. Bergenholtz et al. [4] proposed an RNN-based approach, though limited to detection only.

Two main dataset types exist: Lab and Wild [5]. Lab datasets offer full ground truth control, while Wild datasets use real-world malware labelled by existing tools, introducing mislabelling risks [3]. Biondi et al. [5] assign labels only when at least two of three tools agree, but this restricts the dataset to samples already handled well by existing tools, creating an artificially easier task.

1) Identification based on manually selected features: Feature-based static approaches are prone to engineering errors and may fail on heavily obfuscated binaries, as shown by the processing limitations of 2-spiff in Table I. Biondi et al. [5] use byte entropy, import functions, metadata, and entry bytes, though their datasets are heavily imbalanced. Noureddine et al. [22] extend this with online learning, though highly polymorphic engines risk cluster explosion. Liu et al. [18] propose 2-spiff, combining executable features with a function call graph (FCG), achieving 98.5% accuracy on 10 packers. We use this as our baseline.

2) Identification based on image representation: Kancherla et al. [16] present the only prior image-based study for packer identification, using byte and Markov plots with Wavelet and Gabor features fed into an SVM. Unlike our method, they still rely on feature engineering and do not process the full image directly.

III. Dataset

For this paper, we use a dataset from an online GitHub repository[2] which is a sanitised version of the dataset used by Choi et al. [7]. The data set consists of 25 different packers, and we named it as dataset $D25$.

In Table I, we present the two datasets: $D6$ and $D25$. For each packer in the datasets, we include: the mean size of files, the standard deviation of file sizes, and the number of files. We can see that the statistics are different for the same packer across the two datasets because the original size of the binaries prior to packing can vary, which produces different image sizes. Most methods based on images require the image to be resized to a fixed size. This fixed size can be empirically pre-determined, however, this still produces outliers (very small and very large images), which are very hard to classify because of distortion. For example, Figure 2 presents two images of 128×128 pixels, the left one is a large binary packed with UPX, and the original size of the image was 128×67986 pixels, the right one is a small binary packed using UPX with an original size of 128×144 pixels. Although both images have been packed with UPX, they visually look very different because of the resizing process. This example shows the importance of being able to handle images of any size when using image representation of binaries for packer identification.

The high standard deviation shows that even amongst the files packed with the same packer, the size of the file is spread out. For example, UPX has 309.47 and 859.01 Kilobytes for the average file size and its standard deviation in dataset $D25$, while for dataset $D6$, the corresponding values are 186.00 and 30.64 Kilobytes. As dataset $D25$ contains more packers and fewer files per packer than $D6$, the task is more difficult because the machine learning algorithms have fewer examples

[2]https://github.com/packing-box/dataset-packed-pe

TABLE I
STATISTICS FOR PACKERS IN DATASET $D25$.

	Mean size of files	Standard deviation of file sizes	Number of files	2-spiff file processed[1]
D25 dataset				
Alienyze	558.24	1147.30	126	125
Amber	595.25	1036.75	152	152
ASPack	343.81	982.30	118	118
BeRoEXEPacker	276.37	756.23	116	116
Enigma Virtual Box	1134.45	2056.20	123	123
Eronana Packer	271.19	1165.41	151	151
Exe32pack	26.51	6.76	129	4
EXpressor	197.47	239.14	121	121
FSG	210.86	306.40	119	119
JDPack	259.05	744.08	120	34
MEW	284.80	828.39	116	116
Molebox	225.85	162.71	119	119
MPRESS	206.83	351.59	116	116
Neolite	551.74	1032.01	116	116
NSPack	188.49	239.63	119	119
Packman	344.31	966.61	115	115
PECompact	169.67	227.07	124	21
PEtite	1131.67	4215.80	147	147
RLPack	290.81	757.26	115	115
TELock	184.78	959.39	120	120
Themida	3080.66	1050.01	123	123
UPX	309.47	859.01	122	54
WinUpack	169.75	191.48	119	119
Yoda-Crypter	267.51	210.86	118	76
Yoda-Protector	544.31	1262.83	115	115
Total			3079	2654

[1] 2-spiff file processed: Number of files that 2-spiff is able to process for each class

per packer to train with and more packers to differentiate. Moreover, the standard deviation of file sizes is higher in $D25$ than in $D6$. Hence, $D25$ challenges the resistance of the packer identification methods to spread out file sizes.

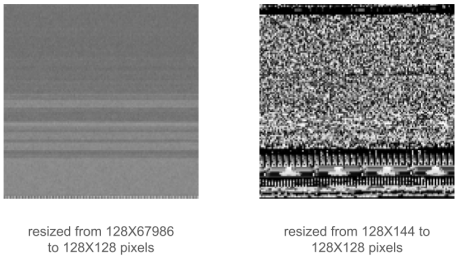

Fig. 2. Comparison of two image representations of UPX binaries when resized to 128*128 pixels.

IV. METHOD FOR IDENTIFYING PACKERS

This paper proposes a packer identification method using grayscale image representations of binaries, combining a convolutional neural network (CNN) and a recurrent neural network (RNN). Following Nataraj et al. [21], each binary is transformed into an image of fixed width (128 pixels per row), where 8-bit groups encode grayscale pixel values between 0 and 255, filled in reading order and zero-padded if necessary. Image heights vary with binary size, ranging from 128×128 to 128×567832 pixels.

A sliding window algorithm then extracts 128×128 windows, stepping downward by 64 pixels from top to bottom. These windows are fed into a CNN that produces compact high-level features, which are then passed to an RNN that predicts the packer used.

This approach offers three key advantages. First, using a CNN removes the need for manual feature engineering, which is time-consuming and prone to human bias; a CNN can be trained in hours compared to the weeks or months required for manual approaches. Second, since no binary analysis tools are needed for feature extraction, our method can process 100% of binaries, unlike code analysis approaches [18] that may fail on some inputs (as shown in Table I). Third, processing images without resizing preserves all information, while the RNN captures both short and long-range dependencies across the full image.

The method consists of the following steps:

Step 1 Binary to image - the binary is transformed into a grayscale image of 128 pixels in width, and the height is defined directly by the size of the binary. The last row is padded with black pixels, if necessary.

Step 2 Window sliding - window sliding is performed on the image vertically with a step of 64 pixels, from which a subimage of 128×128 pixels is extracted at each step. Figure 3 illustrates the result of our window sliding algorithm for an image representation of a binary.

Step 3 Feature extraction - the CNN is used for extracting a feature vector (128 floats) for each subimage.

Step 4 Image classification - the feature vectors are fed sequentially as input to the RNN that performs classification.

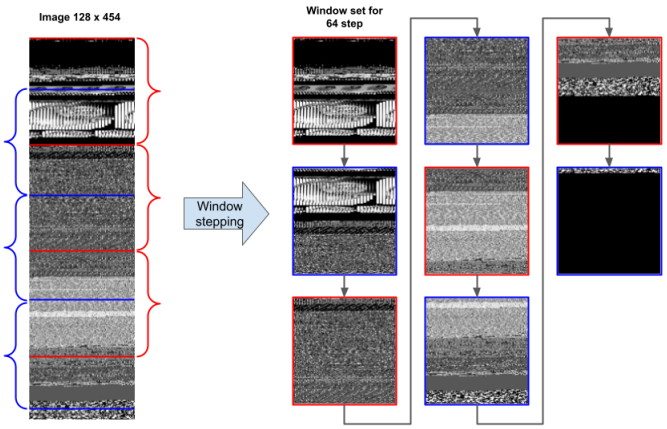

Fig. 3. Illustration of window sliding for a binary image.

A. Convolutional neural network

We now describe the architecture of the CNN [12] that performs feature extraction from an image representing a binary. The full CNN architecture is illustrated in Figure 4 (left), and begins with three identical blocks, where each block is composed of the same two layers. The first layer in each block is a convolutional layer with 1×1 strides and 3×3 kernel size, while the second layer is performing max pooling, with a 2×2 pooling size. After the three convolutional blocks, the extracted features are flattened, and a dropout layer with a drop rate of 0.1 is applied, i.e., 10% of the neurons' activations are set to zero. Dropout [12] is a regularisation technique that is known to reduce overfitting occurring when training large neural networks on small datasets [26], and it was beneficial

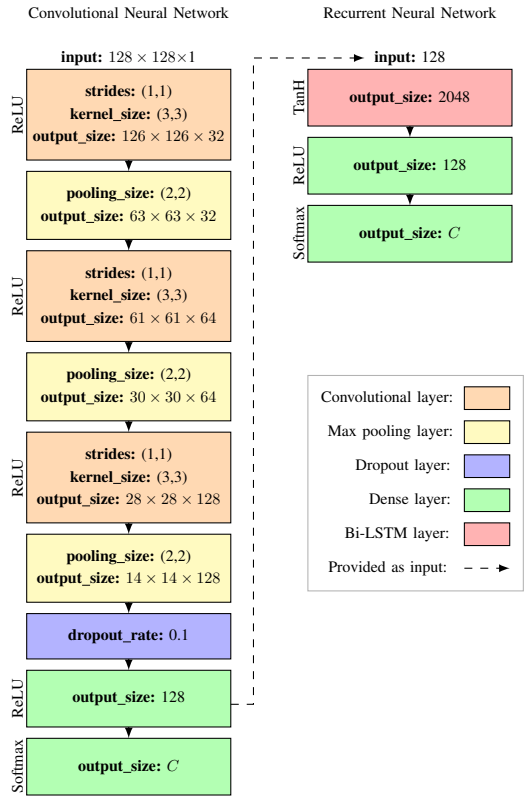

Fig. 4. This figure illustrates the architecture of our model. The CNN is shown on the left, while the RNN is on the right. Both neural networks output a vector of size C, which corresponds to the number of classes, i.e., C = 25 for the dataset $D25$.

when training our 3.3 million parameters CNN on 182K data points for both $D6$ and $D25$. Finally, the CNN ends with two fully connected layers (also known as dense layers). The first layer contains 128 neurons, while the second layer is composed of as many neurons as the number of classes in the dataset. Note, the last layer (with the softmax activation) is removed after training the CNN, and the output of the penultimate layer is provided as input to the RNN.

B. Recurrent neural network

Recurrent neural networks (RNN) have been designed to process time series data, such as text, audio recordings, and historical stock prices, which can be understood as a sequence of elements, e.g., text is a sequence of words. The elements of the sequence are provided as input to the recurrent layers sequentially. More precisely, the first element in the sequence is passed to the layer that outputs a value. Then, the output value is fed back to the layer alongside the second element in the sequence, and this process is repeated until the end of the sequence is reached.

However, the standard recurrent layer described above struggles with long range dependencies. The long short term memory (LSTM) layer addresses this problem by storing internal states containing the contextually relevant information. These internal states can be forgotten as information becomes irrelevant, and new information can be stored when deemed necessary.

Unfortunately, LSTM layers can only keep track of the context presented earlier in the sequence, i.e., the LSTM layers do not consider the elements coming after in the sequence. In terms of binary analysis, the LSTM layer can learn relationships from the header to the footer, but not in the other direction. The bidirectional LSTM (Bi-LSTM) layer solves this issue by combining two LSTM layers. The first LSTM processes the input sequence forward, while the second LSTM processes the sequence backwards, and the output of these two layers is combined to form the Bi-LSTM output [12].

Let's now focus on the architecture of the RNN, which identifies the packer that was used in the binary, c.f. Figure 4 (right). Importantly, the RNN takes as input a sequence of high-level features produced by the CNN when all image slices (i.e., windows) are provided as input. The first layer of the RNN is a Bi-LSTM layer, which consists of two LSTM layers, each composed of 1024 neurons, and whose outputs are concatenated to form the Bi-LSTM output, i.e., a vector of 2048 elements. The RNN ends with two dense layers, the first has 128 neurons, while the size of the second depends on the number of classes in the dataset. This is because, similarly to the CNN, the RNN was trained to identify the packer used in each binary. However, in contrast to the CNN, whose predictions were based on image patches, the RNN's predictions are based on the high-level features extracted by the CNN.

C. Training the convolutional and recurrent neural networks

This section describes the training process for the deep learning architecture described above. Using the notation in Table II, Algorithm 1 describes how the dataset of packed binaries is pre-processed before it is used for training the deep learning models.

The first part of Algorithm 1 — Training feature extractor — corresponds to the CNN training. First, each binary is turned into its image representation (line 4), then a sequence of windows is extracted from the image (line 5). Next, for each window in the sequence, a new training example is added to the CNN training set $\mathcal{D}_{CNN}$, where the window's label is the binary label (line 6). Additionally, the windows sequence is stored in the windows training set $\mathcal{D}_W$ alongside the binary label (line 7).

After creating $\mathcal{D}_{CNN}$ (lines 3 to 8), the CNN is trained for 20 epochs (line 9), and the CNN corresponding to the epoch with the highest accuracy is stripped of its dropout layer, which means that the neurons' activation is no longer clamped to zero. Moreover, the last layer of the CNN is removed, consequently, the CNN output becomes a vector containing 128 high-level features.

The second part of Algorithm 1 — Training packer identifier — corresponds to the RNN training. For each window sequence in the windows dataset $\mathcal{D}_W$, the modified CNN is applied to all the windows to extract a sequence of high-level features (line 12), which is stored in the RNN training set

TABLE II
NOTATIONS FOR TRAINING ALGORITHM.

Symbol	Description
e, i, w, l	An arbitrary executable, image, window, and label, respectively.
W	A set of windows.
F	A set of feature vectors.
$\mathcal{D}_E$	Training set of labelled binaries, i.e., a set of (e, l) tuples.
$\mathcal{D}_{CNN}$	Training set used to train the CNN, i.e., a set of (w, l) tuples.
$\mathcal{D}_W$	Training set of labelled windows, i.e., a set of (W, l) tuples.
$\mathcal{D}_{RNN}$	Training set used to train the RNN, i.e., a set of (F, l) tuples.
CNN	The trained CNN able to extract features.
RNN	The trained RNN able to identify packers.

(line 13). Finally, the RNN is trained for 30 training epochs by performing stochastic gradient descent on the categorical cross-entropy (line 15).

Both models were trained using the Adam optimiser [17] to minimise the categorical cross-entropy defined as follows:

$$\mathcal{L}(y, l) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} l_i^c \ln y_i^c, \tag{1}$$

where N is the number of training samples, C is the number of classes, y_i^c is the probability that the i-th training sample belongs to the c-th class according to the model, lastly, l_i^c is one if the i-th data point is labelled as part of the c-th class and zero otherwise. Intuitively, the categorical cross-entropy is small when the predictions of the model y are close to the correct labels l.

V. EXPERIMENTS

To demonstrate the effectiveness of the proposed method for identifying packers, in the following, we describe the practical set up of the experiments, and present and analyse the results obtained.

A. Experimental setup

Experiments were conducted on two machines. The machine for training the CNNs has an Intel i9-10980XE processor and 128 GB RAM. The machine for training the RNNs and computing the metrics has an A100 GPU, an Intel Gold 5317 CPU and 8 GB of RAM. The experiments were conducted on the two datasets described in Section III, dividing 80% for training and 20% for testing. The division of the dataset for training the CNN and RNN is identical to avoid overlapping as the two models are trained separately (Subsection IV-C). We repeated the experiments on 10 models for both datasets, each with a different random seed (for random dataset division and TensorFlow random seed), and the results were calculated by averaging each metric and reporting the standard deviation.

Algorithm 1: Training the convolution and recurrent neural networks.

Input : Training set of labelled binaries $\mathcal{D}_E$.
Output: The trained CNN able to extract features;
The trained RNN able to identify packers.

===== *Training feature extractor* =====
1 $\mathcal{D}_{CNN} \leftarrow \emptyset$
2 $\mathcal{D}_W \leftarrow \emptyset$
3 **foreach** $(e, l) \in \mathcal{D}_E$ **do**
4 $\quad i \leftarrow executable\_to\_image(e)$
5 $\quad W \leftarrow create\_windows(i, 64)$
6 $\quad \mathcal{D}_{CNN} \leftarrow \mathcal{D}_{CNN} \cup \{(w, l) | w \in W\}$
7 $\quad \mathcal{D}_W \leftarrow \mathcal{D}_W \cup \{(W, l)\}$
8 **end**
9 $CNN \leftarrow train\_cnn(\mathcal{D}_{CNN})$

===== *Training packer identifier* =====
10 $\mathcal{D}_{RNN} \leftarrow \emptyset$
11 **foreach** $(W, l) \in \mathcal{D}_W$ **do**
12 $\quad F = \{CNN(w) | w \in W\}$
13 $\quad \mathcal{D}_{RNN} \leftarrow \mathcal{D}_{RNN} \cup \{(F, l)\}$
14 **end**
15 $RNN \leftarrow train\_rnn(\mathcal{D}_{RNN})$
16 **return** CNN, RNN

B. Implementation

A prototype of our method has been implemented using Pillow [3] for transforming binaries into images, and TensorFlow [1] for implementing the neural networks.

C. Evaluation metrics

We present our results in terms of recall, precision, F1 score and accuracy. These metrics were computed with the SKlearn metrics library [4]. They were calculated based on the number of true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN). Given a packer P, a true positive is a binary packed with P, and classified as P. A false positive is a binary not packed with P, but classified as P. A false negative is a binary packed with P, but classified as any packer but P. Finally, a true negative is a binary not packed with P and classified as such. The computation of the accuracy score was done with the default parameters of SKlearn. For the metrics recall, precision and F1, as the datasets $D6$ and $D25$ are balanced (see Table I), we calculate the average metrics for each label, and find their unweighted mean.

D. Experimental results

Table III contains the results for our method (CNN-BI-LSTM) on both datasets ($D6$ and $D25$). We will discuss the results for each dataset individually, then compare them to conclude.

[3]https://pypi.org/project/Pillow/
[4]https://scikit-learn.org/stable/modules/model_evaluation.html#confusion-matrix

TABLE III
RESULTS FOR 2-SPIFF METHOD, OUR METHOD AND PEID OVER THE TWO DATASETS.

	Dataset	Statistical metric	Recall	Precision	F1	Accuracy	Percentage of treated files
2-spiff	partial D25	avg[1]	89.726	91.364	89.843	94.783	86.2
		std[2]	2.040	1.789	1.933	0.762	
	entire D25	avg	75.800	86.017	76.463	52.635	100.0
		std	0.382	2.133	0.461	0.438	
CNN-BI-LSTM	D25	avg	96.191	96.384	96.173	96.188	100.0
		std	1.023	1.015	1.051	0.983	
PEiD	D25	NA[3]	55.657	61.538	56.409	56.999	100.0

[1]avg: average; [2]std: standard deviation; NA[3]: PEiD does not require training so we use the entire datasets for calculating the metrics.

1) Results for $D25$ :: On the dataset $D25$, when applying our method, we obtain excellent results on all metrics. On average, we obtain above 96% across F1 score, precision, recall and accuracy. The standard deviation is stable across all four metrics with about 1.0. Again, we note that we can classify 100% of the dataset.

From this experiment, we conclude that our method can handle a dataset composed of 25 packers with good stability when generating models with a different random seed, as shown by the standard deviation.

2) Results and discussion :: The results of the experiments show excellent results on both datasets. Nevertheless, we have about a 3.5% decrease on all metrics when generating models from $D25$ compared with $D6$. This shows that our method can handle different sizes of datasets with different numbers of packers while remaining efficient across all metrics. This decrease is most likely due to an increase in the number of classes, i.e, the number of packers in the dataset, since $D6$ contains only 6 different packers, where $D25$ is composed of 25 packers, making the classification task way more difficult. Between the two datasets, there are some common packers as both $D6$ and $D25$ contain binaries packed with UPX, Amber and Upack. Moreover, these experiments show that changing the random seed does not affect the results obtained. Last, these experiments show that our method is capable of handling 100% of the files.

VI. EVALUATION

For evaluating our packer identification method based on deep learning models, we perform a comparison against two packer identification approaches: PEiD[5] – a signature based approach, and 2-spiff [18] – a machine learning based method.

In the following, we start by describing PEiD and 2-spiff, as well as their implementation. Next, we discuss the results obtained by the comparative approaches on both datasets used. Then, we compare these results with our method's results presented in Section V. Finally, we discuss the limitations of the proposed method.

A. Signature based packer identification method

PEiD is a signature based tool that aims at detecting and identifying packers, and for that, it relies on a signature database. The tool scans the target binary for a matching signature from the database, and if there is a match, the binary is identified as packed by a particular packer. In our case, we use the Python implementation of PEiD [6]. This implementation uses 5500 signatures, unlike the original tool [7] that only uses 470 signatures.

Since PEiD does not require training, the entire datasets were used for testing. PEiD sometimes returns an empty label if no signature is matched, and for that, we introduced a class "no label" with zero data points in the dataset to perform the metrics calculation.

B. Machine learning based packer identification method

The machine learning method chosen for comparison is 2-spiff [18]. Overall, since 2-spiff provides excellent results, we have decided to use this method as a baseline comparison for our experiments.

This method is a two step process that first detects whether a binary is packed, then identifies which packer is used. As our datasets are only composed of packed binaries and we only focus on identification, we only implemented the identification part of 2-spiff. Their solution also allows for classification of unseen packers by defining a confidence threshold empirically set to 65%. For each dataset, the model was evaluated by averaging the metrics over 10 random seeds and reporting the standard deviation.

2-spiff uses features extracted from the function call graph (FCG) and file attributes to identify packers. The FCG of a binary is built as follows. First, the binary is disassembled and the functions are identified. Then, return instructions are taken as vertices, and directed edges are added to represent the process of jumping to another function. In essence, this graph represents the possible path from function to function in a binary.

A full description of all the individual features is available in their paper [18]. They leverage the Support Vector Machine (SVM) algorithm to generate a machine learning model. We re-implemented 2-spiff using the following packages:

- IDA Pro [15] for generating the function call graph (FCG).

[5]PEiD is a signature based tool to detect and identify packers

[6]https://github.com/packing-box/peid

[7](https://www.aldeid.com/wiki/PEiD)

- NetworkX [14] for extracting features from the FCG.
- PEfile[8] for generating the file attribute based features.
- SKlearn [23] for implementing the machine learning algorithm (SVM).

For the Support Vector Machine (SVM), we settled on a linear kernel as it produces the best results among all the kernels implemented by SKlearn.

Importantly, 2-spiff is not able to process all the files in the dataset $D25$. This is due to IDA Pro failing in the analysis of some binaries, making it impossible to extract the features associated with the function call graph (FCG). The number of files processed by 2-spiff is available in Table I column "2-spiff file processed". Because not all the binaries in the dataset are classified, we present the results in two different ways in Table III:

- partial D25: Metrics are calculated only for the files that 2-spiff can process.
- entire D25: All the files in the dataset are considered, and we introduced a class "no label" with zero data points. It was considered that 2-spiff labels a binary as "no label" if a file cannot be processed.

Adding the "no label" class allows us to compare 2-spiff to our approach even when 2-spiff cannot process part of the files, and highlights the important ability of our model to handle all the files. Also, presenting the results only on the subset of files that 2-spiff can handle allows us to evaluate 2-spiff in a hypothetical scenario where 2-spiff is capable of handling all the files.

C. Results for dataset $D25$

For 2-spiff, both partial D25 and entire D25 (Table III) are considered. Indeed, entire D25 reflects the current performance of 2-spiff, while partial D25 provides an estimation of the best achievable performance.

2-spiff, on the partial D25, performs well achieving with all metrics above 89%. This shows that 2-spiff is an excellent method for packer identification on the binaries that it can handle. Those results are consistent with the results reported on their paper which shows that our implementation is correct.

2-spiff, on the entire D25, exhibits a drop in performance. This highlights the limitation of 2-spiff, which relies on code analysis that sometimes fails, and demonstrates the need for identification methods that can handle all binaries.

PEiD performs poorly on this dataset with metrics not exceeding 62%. The low performance is caused by the inability of PEiD to classify multiple classes, which leads to 0% recall for these classes. However, this tool is capable of handling all the files in the dataset.

D. Experimental conclusion and comparison of results

There are several conclusions that can be drawn from these experiments. First, the machine learning based method 2-spiff outperforms PEiD on both $D6$ and $D25$. Despite achieving lower performance, PEiD could be advantageous in some settings as it can handle all binary files. Second, we have shown that if 2-spiff was able to process any file, it could produce good performance, i.e., all metrics are around 90% on the partial D25. However, all the metrics significantly drop when considering all the files in $D25$, which indicates the need for methods that can handle any files.

Additionally, in an ideal setup where 2-spiff can handle all the files, i.e., partial D25, our approach improved upon 2-spiff by 4.805% on average over all metrics. In particular, we obtained an improvement of 1.405% in accuracy and 6.465% in recall. This shows that our proposed method outperforms the state-of-the-art even if a perfect implementation of 2-spiff was available.

More importantly, when considering a more realistic scenario where all the files are considered, i.e., entire D25, our method outperformed 2-spiff on all metrics by an average margin of 23.506%.

Nevertheless, 2-spiff still outperforms the signature based solution on 3 metrics out of 4 on entire D25, and outperforms PEiD on partial D25. This set of experiments shows that the capability of handling 100% of the dataset files is fundamental for packer identification, highlighting the superiority of our approach that does not rely on feature engineering and code analysis.

VII. Conclusions and future work

This paper proposes a new static method for identifying packers based on the image representation of binaries. We use convolutional neural networks (CNN) with a sliding window algorithm along the vertical axis to extract a sequence of high level features, which is then fed into a recurrent neural network (RNN) for packer identification. Unlike other approaches, our method does not require resizing the image, eliminating information loss, and handles any binary without relying on manual feature engineering or code analysis. We have demonstrated that our method outperforms the state-of-the-art method [18] on all datasets, with an average improvement of 23.506% on the $D25$ dataset and 4.805% when using only the binaries that 2-spiff can handle.

As future work, we will investigate creating a dataset of wild malware with accurate packer labels, requiring extensive manual reverse engineering and cross-validation. Such a dataset would also allow evaluation against adversarial drift by filtering files by time of appearance. Additionally, we plan to explore applying our method to other binary classification tasks, such as malware family identification, since our approach removes the image resizing limitation.

References

[1] Martín Abadi and et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] AN Akansu, WA Serdijn, and IW Selesnick. 2010. Wavelet transforms in signal processing: a review of

[8] https://pypi.org/project/pefile/

emerging applications. *Physical Communication* 3, 1 (2010), 1–18.

[3] Jim Alves-Foss and Varsah Venugopal. 2022. The Inconvenient Truths of Ground Truth for Binary Analysis. *arXiv preprint arXiv:2210.15079* (2022).

[4] Erik Bergenholtz, Emiliano Casalicchio, Dragos Ilie, and Andrew Moss. 2020. Detection of metamorphic malware packers using multilayered LSTM networks. In *Information and Communications Security: 22nd International Conference, ICICS 2020, Copenhagen, Denmark, August 24–26, 2020, Proceedings 22*. Springer, 36–53.

[5] Fabrizio Biondi, Michael A Enescu, Thomas Given-Wilson, Axel Legay, Lamine Noureddine, and Vivek Verma. 2019. Effective, efficient, and robust packing detection and classification. *Computers & Security* 85 (2019), 436–451.

[6] Tom Brosch and Maik Morgenstern. 2006. Runtime packers: The hidden problem. *Black Hat USA* (2006).

[7] Mi-Jung Choi, Jiwon Bang, Jongwook Kim, Hajin Kim, and Yang-Sae Moon. 2019. All-in-one framework for detection, unpacking, and verification for malware analysis. *Security and Communication Networks* 2019 (2019), 1–16.

[8] Michael D Ernst. 2003. Static and dynamic analysis: Synergy and duality.

[9] Francisco Falcón and Nahuel Riva. 2012. Dynamic binary instrumentation frameworks: I know you're there spying on me. In *Reverse Engineering Conference*.

[10] Ailton Santos Filho, Ricardo J Rodríguez, and Eduardo L Feitosa. 2022. Evasion and countermeasures techniques to detect dynamic binary instrumentation frameworks. *Digital Threats: Research and Practice (DTRAP)* 3, 2 (2022), 1–28.

[11] Daniel Gibert, Carles Mateu, Jordi Planes, and Ramon Vicens. 2019. Using convolutional neural networks for classification of malware represented as images. *Journal of Computer Virology and Hacking Techniques* 15 (2019), 15–28.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[13] Simona E Grigorescu, Nicolai Petkov, and Peter Kruizinga. 2002. Comparison of texture features based on Gabor filters. *IEEE Transactions on Image processing* 11, 10 (2002), 1160–1167.

[14] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). Pasadena, CA USA, 11 – 15.

[15] hex rays. 2007. IDA pro. https://hex-rays.com/ida-pro/

[16] Kesav Kancherla, John Donahue, and Srinivas Mukkamala. 2016. Packer identification using Byte plot and Markov plot. *Journal of Computer Virology and Hacking Techniques* 12 (2016), 101–111.

[17] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[18] Hao Liu, Chun Guo, Yunhe Cui, Guowei Shen, and Yuan Ping. 2021. 2-SPIFF: a 2-stage packer identification method based on function call graph and file attributes. *Applied Intelligence* 51, 12 (2021), 9038–9053.

[19] Francesco Mercaldo and Antonella Santone. 2020. Deep learning for image-based mobile malware detection. *Journal of Computer Virology and Hacking Techniques* 16, 2 (2020), 157–171.

[20] Trivikram Muralidharan, Aviad Cohen, Noa Gerson, and Nir Nissim. 2022. File Packing from the Malware Perspective: Techniques, Analysis Approaches, and Directions for Enhancements. *Comput. Surveys* 55, 5 (2022), 1–45.

[21] Lakshmanan Nataraj, Sreejith Karthikeyan, Gregoire Jacob, and Bangalore S Manjunath. 2011. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*. 1–7.

[22] Lamine Noureddine, Annelie Heuser, Cassius Puodzius, and Olivier Zendra. 2021. SE-PAC: A Self-Evolving PAcker Classifier against rapid packers evolution. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*. 281–292.

[23] F. Pedregosa and et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[24] Moustafa Saleh, E Paul Ratazzi, and Shouhuai Xu. 2017. A control flow graph-based signature for packer identification. In *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*. IEEE, 683–688.

[25] Jibz Snaker, Qwerton and XineohP. 2008. PE iDentifier (PEiD).

[26] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[27] Danish Vasan, Mamoun Alazab, Sobia Wassan, Hamad Naeem, Babak Safaei, and Qin Zheng. 2020. IMCFN: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networks* 171 (2020), 107138.

[28] Danish Vasan, Mamoun Alazab, Sobia Wassan, Babak Safaei, and Qin Zheng. 2020. Image-Based malware classification using ensemble of CNN architectures (IMCEC). *Computers & Security* 92 (2020), 101748.

[29] Hiromu Yakura, Shinnosuke Shinozaki, Reon Nishimura, Yoshihiro Oyama, and Jun Sakuma. 2018. Malware analysis of imaged binary samples by convolutional neural network with attention mechanism. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. 127–134.