CONTENT-AGNOSTIC MALWARE IDENTIFICATION AND BINARY DATA VISUALIZATION USING WAVELET TRANSFORMS

NATHANIEL FERNANDES, GRIFFITH THOMAS, AND BENJAMIN ALLIN

Abstract. When analyzing files without text content - such as executables - it is often necessary to create visualizations based on the file's binary content. These visualizations make it easy to quickly recognize otherwise indistinguishable patterns in the file. Such an approach may prove invaluable in the field of cybersecurity, where identifying malicious code embedded in an otherwise benign file is of utmost concern. In this work, we develop a content-agnostic method to visualize binary files as images using Haar wavelets and use computer vision techniques to perform malware detection on the industry-standard DikeDataset to determine whether or not a given file is malicious or benign. We achieve a remarkable binary classification accuracy, sensitivity, and specificity of $(91 \pm 2)\%$, 0.94 ± 0.06, 0.97 ± 0.00, respectively, with a false negative rate (FNR), area under the receiver operating characteristic curve (AUC-ROC), and F1-score of 0.12 ± 0.04, 0.88 ± 0.04, and 0.91 ± 0.02, respectively. This represents a 57%, 750%, and 45% improvement in accuracy, sensitivity, and specificity, respectively, over the open-source malware detection system "malware-dection" (*sic*). All source code to reproduce the following results can be found at `https://github.com/Nathaniel-Fernandes/math-414-final-project`.

1. Introduction

1.1. Project Scope.

In this work, we develop a useful, mathematically sound algorithm for visualizing binary data and apply it to an open problem in cybersecurity: malware detection. In its raw form, binary data represents an indecipherable string of 1s and 0s. Because binary content is not human-readable, bad actors can potentially embed malicious code into seemingly benign files. As the number of cyberattacks grows year over year [1], it is becoming increasingly important to develop smart systems that can vet incoming data for malicious content. While some malware-detection systems attempt to de-compile the binary data back into machine instructions or the original code [2], these approaches are inadequate for two main reasons. Firstly, it is not always possible to decode the bytes sensibly, especially if a bad actor is intentionally attempting to obfuscate the original source code. Secondly, and more importantly, this approach requires a semantic parser that can understand and flag suspicious content. Such a semantic parser would not only be costly to build but

practically challenging since the set of possible malicious lines of code is virtually infinite.

Therefore, in this work, we propose a content-agnostic, automated method to determine if a file is malicious or not that uses our binary data visualization technique. First, we lay the theoretical groundwork for a mathematically sound method to visualize binary data as an RGB-colored image using wavelets and the Hilbert curve. Second, we apply our visualization technique on the open-source DikeDataset [3] to generate images of malicious vs. benign files and use the resultant images to train a convolutional neural network that classifies the images (and files, by extension) as "malware" or "benign". Lastly, we compare our wavelet-based malware-detection model to the open-source malware detection system "malware-dection" (*sic*) [4]. While we only train and test our model on the DikeDataset, one could easily extend our work to additional datasets, with the performance likely to improve significantly as the neural networks train on more data.

1.2. **Related Works.**

Binary data visualization is a commonly used method for analyzing unknown files. Previous work has visualized Portable Document Format (PDF) files to quickly locate potentially malicious files [5]. The PDF specification supports embedded JavaScript execution, and this JavaScript might exploit vulnerabilities in the PDF viewer's execution engine. To detect these scripts, the bytes of the file were colored based on a pre-defined set of ranges, including ASCII control characters, null bytes, and characters commonly used in code, such as brackets. This made malicious files easy to identify, even with some obfuscation techniques applied.

Visualization has also been used to enhance machine classification of files. In work done at Hanyang University, instruction sequences were reformatted into visual data to calculate similarity scores for files [2]. Instead of taking the raw bytes of the file for visualization, this paper extracted only the sequence of the instruction codes used in the file. This sequence of instructions was then processed with a hash function to place data points on a two-dimensional (2D) grid. These grids, being mostly sparse to prevent hash collisions, are not easily human-readable. Instead, similarity scores were calculated based on this grid and used to classify the input

files. This method successfully classified different types of malware based on their similarity scores.

Wavelets have also frequently been used to extract visual data for classification or identification purposes. HMLET [6] extracted key content information from executables, processed the content data, and trained a machine-learning classification model on the resulting data. Here, wavelets were applied to the extracted content data to extract high-resolution data in both the frequency and time domains. This method also restricted content information to only a 256×256 size buffer but still had success classifying benign and malicious files. A similar approach was taken by Kancherla and Mukkamala, who mapped the bytes of the executable to a grayscale byteplot, applied a level three decomposition to the data, and trained a support vector machine (SVM) classification model on the outputs [7]. Using this approach, they were able to achieve extremely high classification accuracies of 95.95% and false negative rates of less than 2%.

Wavelets have also been applied to other image identification tasks. Work by Wolter et al. created a deepfake detection model by using wavelets to extract spacial and frequency data [8]. They tested several different wavelets, including Haar, Daubechies, and Symlets, extending them to apply to two-dimensional data. They used wavelet packets to analyze both the low- and high-frequency data and collected the mean and standard deviations of the wavelet coefficients. A linear regression model was trained on the resulting data, achieving up to 96% accuracy. Similar efforts from Srivastava and Khare apply wavelets to content-based image retrieval [9]. They use a discrete wavelet transform combined with local binary patterns to extract texture features from images at different resolutions, as well as some other feature extraction techniques. These feature vectors were then used to query images from a database. This method was better than other models with smaller images but needed improvement for larger ones.

2. Mathematical Background

2.1. **1D to 2D Data Mapping.**

Every file, regardless of format, can be represented as a sequence of bytes, or a discrete, one-dimensional (1D) signal. To treat the file data as a two-dimensional

(2D) image, some method must be used to reshape the 1D signals into a 2D format. The most natural method is to define a width and simply wrap data each time it reaches the width:

$$X_{2D}[i, j] = X_{1D}[i + j * \text{width}]$$

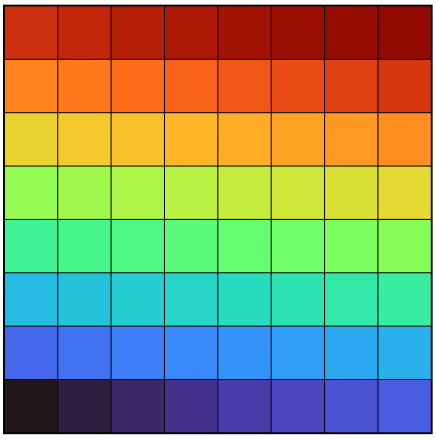

FIGURE 1. A naïve mapping that does not preserve data locality.

This method has the advantage of generalizing to any width and height easily. However, it does not preserve the locality of data properly. As Figure 1 demonstrates, there is a discontinuity between the start and end of each row. This is undesirable for visualization, as it causes large jumps between values in the 2D data where there is no jump in the source data. A more complex approach might mirror every other row to create a snaking pattern, ensuring that a continuous path is taken across the image:

$$X_{2D}[i, (-1 * j\%2) * j] = X_{1D}[i + j * \text{width}]) \tag{1}$$

However, this does not fix the large differences between rows on the side opposite the transition. Instead, a much better approach to map a 1D to 2D signal that preserves data locality would be to use a space-filling curve such as the Hilbert Curve described below.

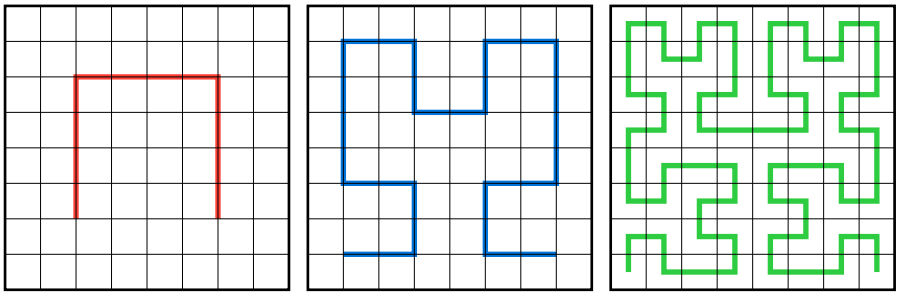

FIGURE 2. The first three levels of the Hilbert Curve, left to right.

As seen in Figure 2, a N-order Hilbert Curve can be constructed by connecting four $N-1$ order Hilbert Curves and reflecting them so no connecting lines cross. An infinite recursion is unnecessary in the case of a finite, discrete signal and one only needs a high enough resolution to reach all data points. One limitation of using the Hilbert Curve for 2D data representation is that the Hilbert Curve only fits a square, and a curve of level l only reaches points on a square of width and height 2^l. This means the data must be reformatted into a square with a side length of 2^x. The parameter x can be chosen to fit all of the data or to contain as much of the data as possible without extending the signal. We select the non-extended version to avoid padding the signal, and thus the remapping is:

$$\begin{gathered} \text{width} \leftarrow 2^{\left\lfloor \log_2 \sqrt{\text{len}(X_{\text{1D}})} \right\rfloor} \\ X_{\text{2D}}[\text{Hilbert}(i + j * \text{width})] = X_{\text{1D}}[i + j * \text{width}] \end{gathered} \tag{2}$$

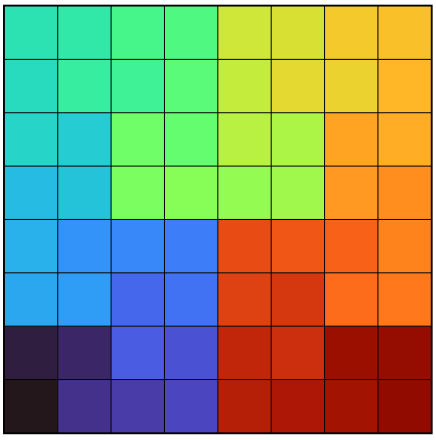

FIGURE 3. The Hilbert Curve mapping applied to the prior visualization.

As mentioned above, this method has the disadvantage of requiring the input data to have size $2^l * 2^l$, where $l \in \mathbb{Z}$. While this does severely restrict the shape of input

data, it may not be an issue as HMLET [6] achieved good results while similarly having a square 256×256 map.

2.2. **2D Wavelet Application.**

In signal processing, it is often desirable to analyze both the time domain and frequency domain of a signal. A common tool for analyzing the frequency components of a signal is the Fourier transform, which converts a signal into its frequency representation. However, the Fourier transform generally does not provide sufficient information about the relationship between the time and frequency domains, such as when a particular frequency enters a signal. The short-time Fourier transform can help by only analyzing one region of a signal at a time, but as the time resolution increases, the frequency resolution decreases.

Wavelets provide a method of analysis that can be performed at multiple time and resolution scales. Similar to the Fourier transform, wavelet analysis projects a signal f into a space represented by a particular class of functions. Unlike Fourier analysis, this projection can then be decomposed into values at different time resolutions using a scaling property.

For demonstration on discrete signals, the Haar wavelets are useful because of their simplicity. The Haar system is composed of a scaling function $\varphi(x)$ and a wavelet function $\psi(x)$:

$$\varphi(x) = \begin{cases} 1 \text{ if } 0 \leq x < 1 \\ 0 \text{ otherwise} \end{cases}$$

$$\psi(x) = \begin{cases} 1 & \text{if } 0 \leq x < \frac{1}{2} \\ -1 \text{ if } & \frac{1}{2} \leq x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Using this equation, we can define a projection of a signal f into the space V_j, which is spanned by the set of functions $\{\varphi(2^{-j}x + k), k \in \mathbb{Z}\}$.

$$\text{proj}_{V_j}(f) = \sum_{k=-\infty}^{\infty} a_k^j \varphi(2^{-j}x + k),$$

$$a_k^j = \frac{\langle f(x), \varphi(x) \rangle}{\|\varphi(x)\|} = 2^{\frac{j}{2}} \int_{-\infty}^{\infty} f(x)\varphi(x)\,\mathrm{d}x$$

However, for the case of a discrete signal $y[m]$, this can be simplified by taking j such that 2^{-j} is the distance between elements in the signal. Then, a_k^j becomes $y[2^{-j}k]$, meaning the projection is:

$$\text{proj}_{V_j}(y)[m] \approx \sum_{k=-\infty}^{\infty} y[2^{-j}k]\varphi(2^{-j}m+k) = y[m]$$

Multi-resolution analysis (MRA) can then be applied to separate the lower and higher frequency components of the signal. In particular, the signal above can be separated into a component contained within V_{j-1} and W_{j-1}, where W_j is the space spanned by the wavelet function $\{\psi(2^{-j}x+k), k \in \mathbb{Z}\}$. This can be done using the L and H filters, defined using the scaling coefficients p_k, which for Haar MRA are $p_0 = p_1 = 1$ and 0 otherwise.

$$a_k^{j-1} = L[a^j]_{2k} = (h * a^j)_{2k} \qquad l = \frac{1}{2}\overline{p_{-k}} = \left(-1 : \frac{1}{2}, 0 : \frac{1}{2}\right)$$

$$b_k^{j-1} = H[a^j]_{2k} = (l * a^j)_{2k} \qquad h = \frac{1}{2}(-1)^k p_{k+1} = \left(-1 : -\frac{1}{2}, 0 : \frac{1}{2}\right)$$

For the case of Haar wavelets, this gives a simple equation that finds the averages and differences of the a^j components. This operation can be repeated multiple times to decompose the signal into components at the j to $j+N$ levels.

The wavelet transform discussed above only applies to 1D signals. To work on 2D signals, such as images, we need a method to generalize these wavelet transforms to higher dimensions. The most commonly used method for images, applied in the JPEG 2000 standard [10], is to apply the high-pass and low-pass filters on each dimension. This gives four quadrants as shown in Figure 4, where L and H represent the results of the high pass and low pass filters in the horizontal and vertical directions. Each remaining quadrant represents the remaining coefficients for each filter combination. The LL component, also called the approximation, is then decomposed again into sub-quadrants. This can be repeated to obtain lower-resolution components of the data.

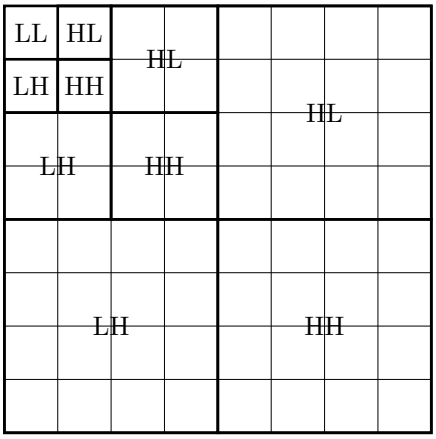

FIGURE 4. A 2D generalization of wavelet filters as used in JPEG 2000

Alternatively, this decomposition model can be applied recursively on all quadrants, rather than limiting it to the approximation component. This method will result in a grid of evenly sized signal components, all decomposed to the same level. However, in the case of visualization, we are interested in extracting features from different resolution levels. Therefore, we want to keep the wavelet coefficients of the lower resolutions as non-decomposed values to preserve the features of those levels.

2.3. **A Brief Introduction to Neural Networks.**

When trying to classify sets of data (i.e., assign each input datum a categorical label), there exists a plethora of methods. While statistical features can be used, neural networks are gaining in popularity due to their wide-scale accessibility and high accuracy. This type of model relies on using a perceptron as a building block, which is a weighted sum of inputs:

$$y = b + \sum_{i=1}^{n} x_i w_i$$

$$\text{ReLU}(y) = \begin{cases} y \text{ if } y > 0 \\ 0 \text{ otherwise} \end{cases}$$

The value, y, can then be passed through an activation function (e.g., ReLU) that introduces nonlinearities into an otherwise linear system. In a fully connected neural network, the output of each layer of perceptrons is fed into every single perceptron in the subsequent layer. This structure allows for neural networks to be capable of representing a wide variety of functions. For example, a neural network with at

least one hidden layer, a non-linear activation function, and a sufficient number of neurons can approximate any continuous function arbitrarily well [11].

However, for scenarios such as training an image classification model where there are many parameters, these networks can become incredibly costly to train. To address this issue, convolutional neural networks were developed specifically for image-related problems. In this model, each pixel in the image H is treated as an input, but instead of passing these images directly to perceptrons, a filter F consisting of a matrix of random numbers is passed over the image. Convolving the image with multiple of these filters, as shown by operator G, allows for the model to extract features from the image without manual oversight.

$$G[i,j] = (F * H)[i,j] = \sum_u \sum_v H[u,v]F[i-u, j-v]$$

To ensure that the model does not grow too large from using multiple filters, a pooling layer is commonly used to reduce complexity. These layers take a small portion of the image (e.g., a 4×4 patch) and either average or max the inputs into one data point. This helps avoid overfitting and ensures that the compute time does not increase exponentially. Additionally, in order to combine all of the information found in the previous layers, the several layers of data created by the various filters are flattened into a 1D vector. This vector can be used as input to a fully connected neural network, which can be used to complete the classification model and obtain an output.

A loss function is used to train the model by quantifying the current performance of the model on the task. In our case, since we are performing binary classification, we use the binary cross-entropy loss function, L_{BCE}. Here, the average difference between the correct results and actual predictions is calculated on a logarithmic scale, which mirrors entropy calculations:

$$L_{BCE} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \cdot \log(y_i) + (1-y_i) \cdot \log(1-y_i)]$$

Once this value is found, gradient descent and backpropagation can be used to calculate the necessary updates to all perceptrons' weights and filter values.

3. Application

At a high level, our system works by reading bytes from a file and remapping the 1D signal into a two-dimensional grid called a byteplot, and then applying a 2D Haar wavelet transform to decompose the byteplot into N levels of wavelet components. These results are colorized for human use and the resulting image is saved as a "png" and fed into a convolutional neural network which classifies the original file as "malware" or "benign". Note that all source code to reproduce these results can be found at `https://github.com/Nathaniel-Fernandes/math-414-final-project`.

3.1. **Preprocessing.**

3.1.1. *System Requirements.*

A laptop or desktop computer is required capable of running Python. It is required to use Python 3.7 in order to run the "malware-dection" software for benchmarking. In addition, the following third-party packages and exact versions are needed. Please consult the ReadMe on GitHub for any additional help setting up the environment and running our code.

3.1.2. *Data Processing.*

The processing step begins by reading in a user-specified file as a variable-sized array of 8-bit bytes unsigned integers. This array is then transformed into a 2D array of the same values using either the row-column (Equation 1) or Hilbert curve (Equation 2) approaches discussed above, depending on which flag the user passes via the command line. To simplify the algorithm, we restrict this 2D array to a square grid with equal side lengths of 2^l, where l is an integer selected to be as large as possible while still being filled entirely by the file data.

Once the data is in a 2D grid, it is processed using a wavelet transform selected by the user from a list of supported families (i.e., Haar and Daubechies). Then, a new grid is constructed using the low and high pass components of the transform. This is repeated for the higher levels of decomposition, with each level filling the space of the approximation component in the previous level. The approximation component is only written for the last level of the transform, producing a grid similar to Figure 4. This grid is restricted to the same size as the original data. Some wavelets (which contain more than two non-zero scaling coefficients p_k) pro-

duce results that are too large to fit into the quadrant of the grid that is assigned to them, so the extra coefficients are dropped. Each quadrant or sub-quadrant is normalized to the region $[0, 1]$ to ensure that the data, especially the approximation component, does not overwhelm the others during visualization.

3.1.3. *Visual Identification.*

The visualization step takes in the normalized wavelet coefficients from the processing step. An image is created with each pixel representing one coefficient, where the colors are selected from a red-purple color gradient based on the coefficient's normalized value. To ensure that the image is easily displayable, it is scaled by a factor of four ($w_{\text{new}} = 4 \cdot w_{\text{old}}$). This prevents the linear interpolation commonly used in photo display programs from blurring the data.

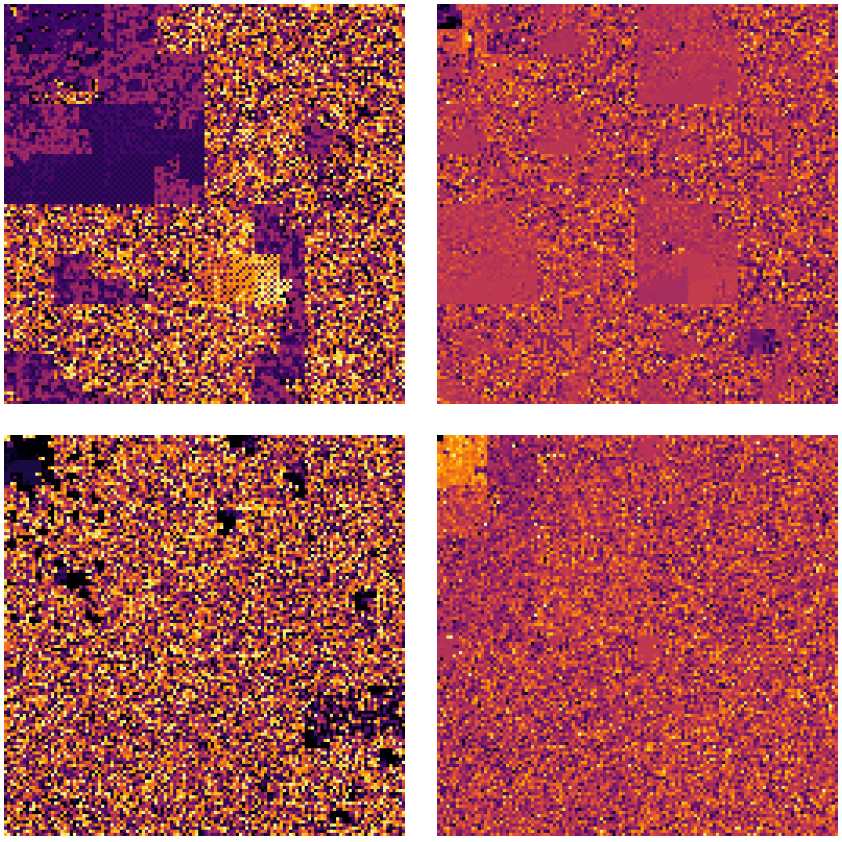

FIGURE 5. On top, a PDF file with no decomposition applied (left) and the same with level 3 decomposition (right). On bottom, a JPEG image

with no decomposition and level 3 decomposition. All use Hilbert curve mapping and Haar wavelets

The resulting images are quite striking, but can often be challenging to interpret. Recognizable patterns are easily noticeable in both the row-column layout and in the Hilbert curve layout but are more distinguishable in the Hilbert curve representation, likely due to data locality being preserved. Large blocks of similar values are easily recognizable in the results, but more detailed patterns tend to look like noise in the high-frequency components. This is especially apparent in file types that use compression, as much of the original data has a noisy appearance to begin with (Figure 5). Due to the noisy behavior of the extracted detail, we found it easier to recognize patterns in the binary file before the wavelet decomposition was applied using the Hilbert curve transform.

3.2. **Machine Learning Classification.**

This step takes as input the image files generated from the wavelet transforms and attempts to classify them as "malware" or "benign". We chose to use the DikeDataset which offers pre-annotated malicious and benign executables because it is open-source and fits our needs. After completing the above preprocessing steps on 2164 executables (with 50% benign and 50% malicious), we train and test a custom-built convolutional neural network using the procedures below.

Note: For training purposes, we consider "malware" the positive label, meaning that a true positive will properly identify a malicious file as malware (see Table 1).

3.2.1. *Dataset Description.*

We use the open-source DikeDataset [3] because it is one of the few, high-quality compilations of pre-annotated benign and malicious executable files. The dataset consists of 1,082 benign files and 10,841 malicious ones. To correct the class imbalance, we used all 1,082 benign files and randomly selected exactly 1,082 malicious files. In addition to labeling each file as "malware" and "benign", the DikeDataset also contains the type(s) of malware that is contained in malicious files (i.e., generic, trojan, ransomware, worm, backdoor, spyware, rootkit, encrypter, downloader). To limit complexity, we did not use this information but instead left this multiclass classification problem for future work.

Warning: the malware files included in DikeDataset are real malware. Do not execute them at the risk of damaging your system.

3.2.2. *Performance Metrics & Benchmarks.*

Since we are performing a classification machine learning task, we use the industry-standard performance metrics of accuracy, sensitivity (aka. true positive rate or TPR), specificity, area under the receiver operating characteristic curve (AUC-ROC), and F1-score. Moreover, for our specific task, it is very important to know if malicious files are incorrectly being predicted as benign (whereas the opposite scenario is less important). Therefore, we have also decided to calculate the false negative rate (FNR). All these metrics can be derived from the confusion matrix, as seen in the tables below. (Note: $\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$.)

	Predicted Malware	Predicted Benign
Actually Malware	TP	FN
Actually Benign	FP	TN

TABLE 1. A Standard 2x2 Confusion Matrix.

Metric	**Equation**	**Range**	**Higher Is...**
Accuracy	$\frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}$	[0%, 100%]	Better
Sensitivity	$\frac{\text{TP}}{\text{TP} + \text{FN}}$	[0%, 100%]	Better
Specificity	$\frac{\text{TN}}{\text{TN} + \text{FP}}$	[0%, 100%]	Better
AUC-ROC	$\int_0^1 \text{TPR}(\text{FPR}) d\text{FPR}$	[0, 1]	Better
F1-Score	$\frac{\text{TP}}{\text{TP} + \frac{1}{2}(\text{FP} + \text{FN})}$	[0, 1]	Better
FNR	$\frac{\text{FN}}{\text{TP} + \text{FN}}$	[0%, 100%]	Worse

TABLE 2. List of Metrics Used.

To benchmark our performance, we compare our wavelet-based malware detection model to the open-source malware detection system "malware-dection" (*sic*) [4]. While we attempted to compare against [12], we found it impossible to install the package due to its dependence on many legacy packages that had been deprecated.

As per the DeepReflect GitHub repository, an attempt to contact the author was made on July 24, 2022, to no avail. We chose not to compare our model to paid malware detection software due to constraints on the project budget ($0). Lastly, we defer comparison to other open-source malware detection systems [6] [2] [7] for the future since doing so would fall outside the scope of our work: our goal is to demonstrate our method of using wavelet transforms is viable for malware detection, not necessarily prove our system is the best with comprehensive tests.

3.2.3. *Convolution Neural Network Architecture.*

The CNN used in this classification has two convolution stages with 16 3x3 filters for the first stage and 32 filters for the second stage. Max pooling steps were performed after each convolution stage to decrease overfitting and reduce the memory footprint and training costs. After this, the data was flattened and connected to a 120-neuron layer followed by an 84-neuron layer, and finally an output layer. The ReLU activation function was used throughout the entire model.

Before the data could be inputted into the model, the images, which ranged from 32×32 to 2048×2048, needed to be converted to a universal size. To stay consistent with the processing step, only sizes of $2^l \times 2^l$, where $l \in \mathbb{Z}^+$ were tested. This also reduced the number of images that would have to be resized.

3.2.4. *Training & Testing Protocol.*

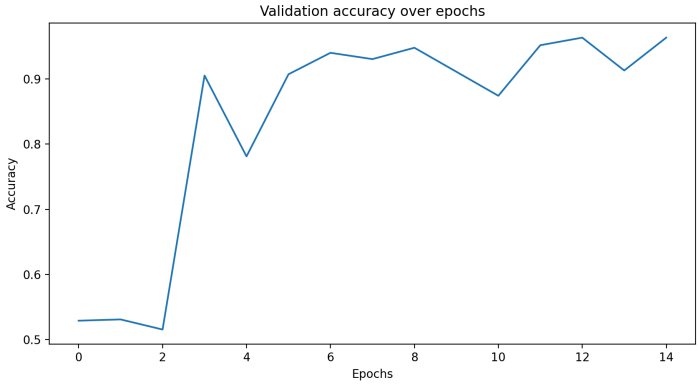

FIGURE 6. Accuracy Increases With Training

To train our model, all data was randomly split into 80% training data and 20% testing data. The model was trained in batches of 16 samples, with a total of 5-15 epochs being completed. The accuracy of the model was captured after each epoch

graphed in Figure 6. The final accuracy of 91% demonstrates how the classification model is capable of reasonably differentiating between malware and benign files and considering features that were not immediately observable.

4. Conclusions

4.1. **Results.**

Before training the network, we skimmed through the samples of benign and malicious samples to see if there were any observable differences. As seen in Figure 7, we noticed that some of the malicious samples had well-defined, square, uni-colored patches while the benign samples seemed more randomized. This may be a limitation of the DikeDataset and how they generated the malicious binaries, however, it is difficult to tell since the binary data is indecipherable and hence unreadable. To remedy this problem, it would be good to find additional data sets to train our ML model on in future use cases. However, this was only a general observation and there were still many malicious samples that did not follow this trend that the model was able to correctly predict.

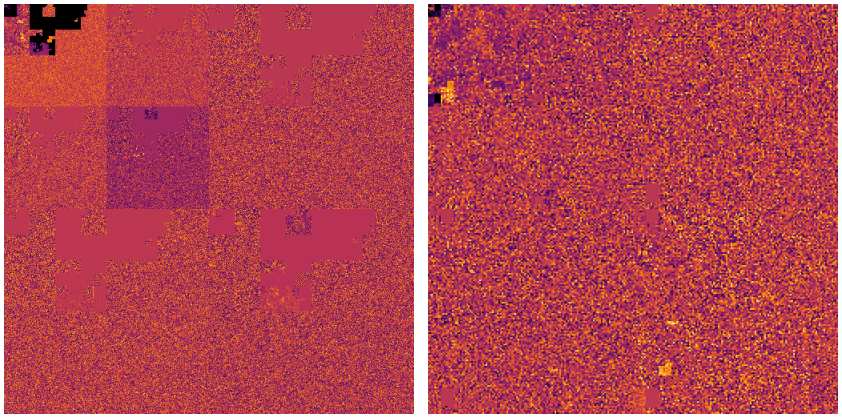

Figure 7. Side-by-side comparison of example malware File (Left) and benign (Right). The malware has square patches of uniform color whereas the benign file's colors seem more random.

After testing out various sizes, we discovered that 64×64 was the smallest image size that produced viable results. Images any smaller would result in the larger images losing too many features and a general decrease in the performance of the trained model. On the other hand, increasing the size of the image only produced increases in the accuracy of the model and stability of its predictions, with benefits plateauing after 512×512. Since the model began taking increasingly longer times to train after this point, we decided to settle on preprocessing images to 128×128 as a good balance between accuracy benefits and training costs.

System	Accuracy	Sensitivity	Specificity
Ours	**0.91 (0.02)**	**0.94 (0.06)**	**0.97 (0.00)**
malware-dection	0.58	0.11	0.67

System	FNR	AUC-ROC	F1-Score
Ours	0.12 (0.04)	0.88 (0.04)	**0.91 (0.02)**
malware-dection	**0.00**	**1.00**	0.72

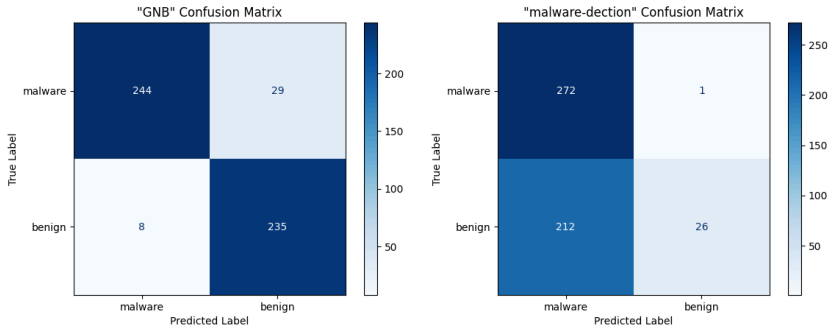

FIGURE 8. Comparison of binary classification confusion matrices

Overall, we are ecstatic about our results. By using a relatively simple wavelet transform to visualize the binary data in a novel way, we were successfully able to detect malicious files. Since we balanced the datasets (including exactly 1082 benign files and 1082 malicious ones), the baseline accuracy for randomly guessing is exactly 50%. Thus, we have reported an $\frac{91\%-50\%}{50\%} = 82\%$ improvement in malware detection over random guessing. Moreover, our sensitivity, specificity, and false negative ratings of 94%, 97%, and 10.6% are remarkable, given that the model is content-agnostic and only looks at the binary visualization images. Compared to "malware-dection", we achieved a 57%, 750%, and 45% improvement in accuracy, sensitivity, and specificity, respectively. While their software has a 0% false negative rate and perfect AUC-ROC of 1.00, it is easy to see why when looking at the confusion matrix in Figure 8: the "malware-dection" model (almost) always outputs the label of malware, regardless of if the file is actually benign! (This explains why their sensitivity is so low: they simply classify every file as malware.)

4.2. **Future Work.**

Perhaps the most immediate next step moving forward with our model would be to modify the loss function to penalize false negatives more harshly. This is

due to the fact that in most applications, labeling a malicious file as benign (false negative) is much worse than labeling a benign file as malicious (false positive). Since there is always a tradeoff between these two types of wrong classifications, favoring the less harmful one would help our model become more useful in a real-world scenario.

Another feature that could be used to extend the usefulness of the project would be to identify the features that the CNN extracts from the processed input images. This would allow for the capability to see what characteristics in the inputs are most useful in identifying malware. Various methods and software programs facilitate this analysis such as the SHAP Python package [13]. Researching and adding one of these programs to our model could help build a better understanding of what is happening within the model.

Some improvements can be made with how our model utilizes the DikeDataset. For example, we only used 1082 of the 10841 available malicious files. Using all of the data could potentially help the model become more accurate and reduce overfitting. However, doing this would create large class imbalances in the training and testing data. To fix this, we can change the performance metrics we use such as using "balanced accuracy" ($BA = \frac{\text{Sensitivity + Specificity}}{2}$) instead of accuracy or the area under the precision-recall curve (AUC-PRC) instead of the AUC-ROC, which is more informative on binary classification tasks with heavily imbalanced data [14].

Additionally, the DikeDataset also contains annotations for what each malicious file is: generic, trojan, ransomware, worm, backdoor, spyware, rootkit, encrypter, and/or downloader. Therefore, we could extend our base classification model to classify the type of malware that a sample file contains. This would require increasing the number of output labels used in the model as well as using a sigmoid output function to allow for multiclass label prediction.

Lastly, while our current colorization scheme is useful for visualizing 1-channel, gray-scale images, we could modify our colorization scheme to take full advantage of the RGB color spectrum. Extensions to our approach could try colorization by byte class, similar to what was used by Mavric et al. [5]. For the wavelet components, the values are often centered around zero before normalization, so a negative/positive color scheme could be used to differentiate from the approximation data.

4.3. **Closing Thoughts.**

Overall, we were successfully able to use various 2D wavelet transforms to visualize binary files. Using the Haar wavelet, we trained a classification model to recognize the differences between malware and benign files in the DikeDataset. The final classification model had a remarkable binary classification accuracy, sensitivity, and specificity of $(91 \pm 2)\%$, 0.94 ± 0.06, 0.97 ± 0.00, respectively. This is an incredible improvement over the similar malware classification software "malware-dection", and achieves similar accuracy to other wavelet-based classification systems [7]. While we have outlined several areas for future improvements, we are greatly satisfied with our results and have successfully developed a mathematically sound algorithm for visualizing binary data and demonstrated its usefulness in the domain of malware detection.

References

1. Sayce, S.: 3 trends set to drive cyberattacks and ransomware in 2024. (2024)
2. Han, K., Lim, J. H., Im, E. G.: Malware analysis method using visualization of binary files. In: Proceedings of the 2013 Research in Adaptive and Convergent Systems. pp. 317–321. Association for Computing Machinery, Montreal, Quebec, Canada (2013)
3. iosifache: DikeDataset. GitHub repository. https://github.com/iosifache/DikeDataset. (2023)
4. kvu228: malware-dection. GitHub repository. (2023)
5. Mavric, S. H. T., Yeo, C. K.: Online binary visualization for Pdf documents. In: 2018 International Symposium on Consumer Technologies (ISCT). pp. 18–21 (2018)
6. Park, S., Jeon, S., Kim, H. K.: HMLET: Hunt Malware using Wavelet Transform on Cross-Platform. IEEE Access. 1–2 (2022). https://doi.org/10.1109/ACCESS.2022.3225223
7. Kancherla, K., Mukkamala, S.: Image visualization based malware detection. In: 2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS). pp. 40–44 (2013)
8. Wolter, M., Blanke, F., Heese, R., Garcke, J.: Wavelet-packets for deepfake image analysis and detection. Machine Learning. 111, 4295–4327 (2022). https://doi.org/10.1007/s10994-022-06225-5
9. Srivastava, P., Khare, A.: Integration of wavelet transform, Local Binary Patterns and moments for content-based image retrieval. Journal of Visual Communication and Image Representation. 42, 78–103 (2017). https://doi.org/10.1016/j.jvcir.2016.11.008
10. Skodras, A., Christopoulos, C., Ebrahimi, T.: The JPEG 2000 Still Image Compression Standards and Beyond. Signal Processing Magazine, IEEE. 18, 36–58 (2001). https://doi.org/10.1109/79.952804
11. Sahay, M.: Neural Networks and the Universal Approximation Theorem. (2020)
12. Downing, E., Mirsky, Y., Park, K., Lee, W.: DeepReflect: Discovering Malicious Functionality through Binary Reconstruction. In: 30th ${$USENIX$}$ Security Symposium (${$USENIX$}$ Security 21) (2021)
13. Lundberg, S. M., Lee, S.-I.: A Unified Approach to Interpreting Model Predictions, http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf, (2017)
14. Saito, T., Rehmsmeier, M.: The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets. (2015). https://doi.org/10.1371/journal.pone.0118432

Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843
Email address: njfernandes24@tamu.edu
URL: www.linkedin.com/in/nathanielfernandes/

Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843
Email address: cedric1008@tamu.edu
URL: griffiththomas.pages.dev

Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843
Email address: btallin@tamu.edu
URL: www.linkedin.com/in/benjamin-allin-470b711b4/