CISSE™

Open Access License Notice

This article is © its author(s) and is licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0). This license applies regardless of any copyright or pricing statements appearing later in this PDF. Those statements reflect formatting from the print edition and do not represent the current open access licensing policy.

License details: https://creativecommons.org/licenses/by/4.0/

The Colloquium for Information Systems Security Education (CISSE)

Efficient Machine Learning for Malware Detection

Thomas Andrew Koch
Department of Computer Science and Information Systems
University of North Georgia
Dahlonega, USA
takoch8198@ung.edu
0009-0000-2585-0202

Tamirat Abegaz
Department of Computer Science and Information Systems
University of North Georgia
Dahlonega, USA
tamirat.abegaz@ung.edu
0000-0003-1263-8469

Hyungbae Park
Department of Computer Science and Information Systems
University of North Georgia
Dahlonega, USA
hpark@ung.edu
0009-0007-9777-4211

Abstract—As the landscape of cyber threats continues to expand, malware detection has become increasingly crucial for maintaining robust cybersecurity. While standard malware detection techniques such as signature-based methods are very effective and widespread, they face certain challenges with zero-day and novel malware. The emergence of artificial intelligence in recent years has led to the development of alternative approaches to this issue, specifically through machine learning techniques. This research aims to analyze the effectiveness and viability of one such machine learning approach; the use of a Convolutional Neural Network (CNN) model for the classification of benign and malicious Windows executable binaries. To accomplish this, we gathered a substantial dataset of both benign and malicious Windows binaries and converted them into grayscale images to train several CNN models with slightly varying architecture for the classification task. Following the training of the models, they were evaluated on an unseen test dataset to compare label predictions against each other, as well as Windows Defender. This approach aims to achieve a definitive metric for determining the effectiveness of this type of malware detection for Windows-based antivirus applications. What we found is that certain CNN models are not only able to perform on par with Windows Defender, but in some cases even outperform them. In conclusion, our study demonstrated that utilizing CNN models with grayscale image conversion of Windows binaries is an effective and efficient approach to malware detection.

Keywords—*Machine Learning, Artificial Intelligence, Malware Detection, Convolutional Neural Network, ML, AI, CNN*

I. INTRODUCTION

Malware detection is a critical component of cybersecurity practice. Malware based cyber-attacks account for millions of security breaches every year [1], and having effective techniques to detect these attacks is critical to maintaining proper data security. The industry standard for malware detection uses database signatures of known malicious binaries for detection. This could include file hashing, file size and metadata, binary strings, network traffic patterns, YARA rules, or any other type of identifying feature that is characteristic of a known malware source [2]. This method is very effective for detecting against the rampant amount of known malicious binaries that are out there. Signature-based detection has been around for many years, and has a large established infrastructure, with huge databases containing hundreds of millions of known binaries. However, even though signature based detection has a lot of positive components and has been proven highly effective, it does face some challenges. The most obvious one is that it relies on a sample being previously analyzed and added to a database to be detected, this can cause problems in certain cases when dealing with novelty or zero-day malware attacks [3]. Because this type of malware is not yet known, it could slip by signature-based antivirus/anti-malware software. There is also a need to constantly update the database or have access to a newly updated database to keep up with the latest signatures. In recent years, some alternative approaches to malware detection have been developed. Some of these approaches, including the one analyzed in this project, utilize machine-learning algorithms and deep neural networks to train an artificial intelligence model on pattern recognition of malware [4]. This approach allows binaries to be analyzed based on different patterns and behaviors of executables completely statistically. Most importantly, it also allows a model to be trained on certain behaviors and detect these behaviors in binaries that might be novel, or not yet entered into one of the signature databases. There are several advantages to this type of machine learning approach, including accurate detection of novel malware, less of a need for constant updates, and scalability without the need for huge databases containing millions of every known piece of malware. This project aims to analyze the effectiveness of these machine-based deep learning models when compared to signature-based detection to determine the viability and accuracy of this approach, specifically on Windows systems dealing with executable binaries.

CNNs are a very effective type of machine-learning model for this kind of task. Windows binaries provide a very complex datatype for a machine learning model, and deep neural networks are more effective than other models when dealing with this type of complex data [5]. The primary objective of this research is to address the following questions: To what extent can a CNN differentiate between benign and malicious Windows executables when trained on a custom dataset, and

how accurate is it in predicting harmful Windows binaries? How does its performance compare to standard detection methods such as Windows Defender? When assessing the advantages and disadvantages of standard detection versus machine learning-based antivirus solutions, it is important to evaluate the accuracy of machine learning models in comparison to conventional detection methods. Even though they offer potentially significant advantages over signature-based detection, if their detection accuracy is compromised, it might not be worth implementing. This project aims to clarify the effectiveness of these different methods for detecting binaries on Windows systems. In order to achieve this, we trained several different CNNs on a substantial dataset of both benign and malicious software, then employed these models on an unseen dataset and compared their prediction scores to a Windows Defender scan to assess the effective accuracy of these methods.

II. RELATED WORKS

A. Machine Learning for Malware Analysis

CNNs and other machine learning models have recently become state-of-the-art methods for solving complex computing problems, as they have proven effective for these tasks. One example is how they were employed to detect vulnerabilities between different programming languages [6]. As far as malware goes, some have taken similar approaches, such as training machine learning models on similarities between bit-sequences [7]. Others have reviewed and proven the benefits of machine learning malware detection, such as its effectiveness on novel malware [5], [8].

B. Efficiency and Computational Cost

One of the drawbacks of many machine learning methods involving binary analysis is that they can come with very heavy computational costs. To address this issue, researchers have implemented various efficient techniques, such as converting binaries to grayscale images and training deep learning networks on these images [9]. Due to their high level of accuracy and low computational cost, this is the approach we adopted for this project. In [10], the authors have done things such as using lightweight dynamic models only using assembly strings rather than using larger pre-trained models to achieve similar results with less resources. Finally, some researchers have employed language processing techniques with machine learning to improve static AI analysis of malware [11].

C. Alternative Approaches

In [12], the authors have made tools for android, for example, that extract program functions and API calls to determine if a file is malicious. Others have developed very precise computational tools for detecting binary similarities [13]. To supplement these findings, we aim to demonstrate that the efficiency of machine-learning approaches to malware detection is comparable to that of Windows Defender. While it might not be feasible yet to fully replace the conventional malware detection methods, a potential application of this approach could involve integrating a machine learning framework with existing malware detection methods. This combined strategy could be used to examine unmarked files for potential instances of novel malware.

III. METHODOLOGY

For the design of this experiment, the specific process will involve employing a CNN to classify binaries in the form of grayscale images. This will include five major steps, outlined in the following subsections.

A. Collecting the Dataset

Windows binaries were collected and organized into two distinct datasets. The first dataset comprises benign software. A substantial number of executables in this dataset are sourced from Windows system applications, as these are known to be harmless. This collection is supplemented with additional binaries confirmed to be harmless and deemed appropriate for the project. Secondly, malware binaries will be collected. GitHub has many open source “educational” malware repositories, and this project will use a lot of these as examples of malware to train the model. This section will involve executables that are directly known to be malicious. Our initial goal is to compile a dataset containing at least 50 benign and 50 malicious executables to train the model with, potentially increasing these numbers based on the results of the initial test runs. The justification for using a custom dataset rather than a large publicly available one is as follows:

- Customizability – A custom dataset allows for the selection of specific samples that are relevant to our research and better meets the project’s needs.
- Size – A smaller size will provide more insight into the machine learning models used, as well as allow for a greater detail of preparation when preprocessing the data.
- Training Time – A smaller dataset enables quicker training and turnaround, allowing us to fine-tune hyperparameters effectively and apply data augmentation techniques.

We acknowledge that one of the potential problems with a smaller dataset is a higher chance of overfitting, and possibly a relatively low validation accuracy. This issue will be mitigated with substantial data augmentation to artificially increase the size of the dataset. Low validation accuracy is accepted as a part of the experiment, assuming that it still reaches acceptable levels (above 70%). Experimentation with parameters will be done to increase the validation accuracy as high as possible.

B. Preparing the Dataset

For this step, all the binaries that had been collected were converted to grayscale images. To do this, an executable was read and saved as a binary using Python, then preprocessed to fit a standard size (determined during testing). After that had been done, the NumPy and OpenCV libraries for Python

were used to convert the data into an array format, and print it as a grayscale image. The exact dimensions of the preprocessed data will need to be determined in the following steps, once the dataset collection is complete. Once all the binaries are converted and saved as grayscale images, the training and test dataset will be prepared off these images. The preparation will involve sorting the images into test and training sets and labeling them either as malicious or benign.

As shown in Figure 1, the binaries were preprocessed using a Python script, which employed the previously mentioned NumPy and OpenCV libraries to trim and pad the files to a consistent array size and convert them into 128x128 pixel grayscale images.

```
def convert_to_image(binary, image, imageWidth, imageHeight):

    with open(binary, 'rb') as file:
        binary_data = file.read()

    array = np.frombuffer(binary_data, dtype=np.uint8)

    # Preprocessing step: Add padding if the file is too small or trim if it is too large
    if len(array) < imageHeight * imageWidth:
        padding = np.zeros(imageHeight * imageWidth - len(array), dtype=np.uint8)
        array = np.concatenate((array, padding))
    elif len(array) > imageHeight * imageWidth:
        array = array[:imageHeight * imageWidth]

    image_data = array.reshape((imageHeight, imageWidth))

    cv2.imwrite(image, image_data)
    print(f"image saved as {image}")

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Convert binary to greyscale image")
    parser.add_argument("folder", help="The folder you want to convert")
    parser.add_argument("--output_folder", help="The output folder")
    parser.add_argument("--width", type=int, default=128, help="Image width")
    parser.add_argument("--height", type=int, default=128, help="Image Height")

    args=parser.parse_args()
    folder = args.folder
    output_folder = args.output_folder
    imageWidth = args.width
    imageHeight = args.height

    if output_folder and not os.path.exists(output_folder):
        os.makedirs(output_folder)

    for filename in os.listdir(folder):
        if filename.endswith(".exe"):
            binary = os.path.join(folder, filename)
            output_filename = f"{os.path.splitext(filename)[0]}_IMAGE.png"
            image = os.path.join(output_folder, output_filename) if output_folder else output_filename
            convert_to_image(binary, image, imageWidth, imageHeight)
    if output_folder:
        print(f"Images saved in the folder: {output_folder}")
    else:
        print("Images saved in the same folder as the binaries")
```

Fig. 1. Preprocessing Script.

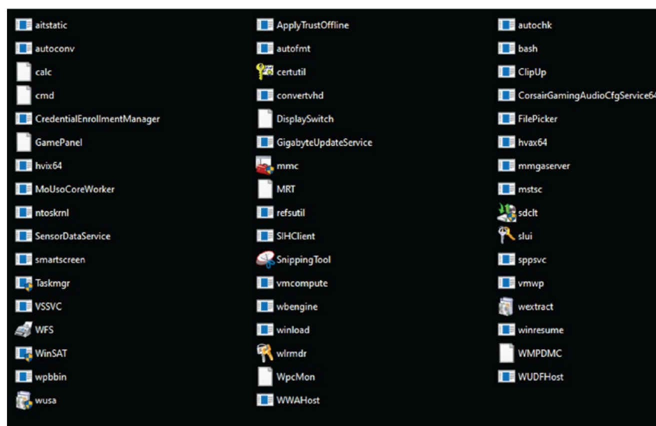


Fig. 2. Original Binaries.

Figures 2 and 3 display a sample set of the benign executables sourced from Windows system apps, illustrating their state before and after preprocessing. Figure 2 shows the original executables prior to preprocessing, and Figure 3 shows the grayscale images after they have undergone the preprocessing script shown in Figure 1.

Fig. 3. Preprocessed Binary Images

C. Building a CNN Model

We use a CNN model rather than some other image processing models. The justification for choosing this model is as follows:

- Image processing – CNNs have been proven very effective at image classification tasks, which this project involves [14].
- Accuracy of Images – Comparing executables as grayscale images has been found to be highly effective for accurate pattern recognition [15].
- Wide use – Convolutional Networks appear to be the standard for this type of machine learning problem, which means there will likely be lots of available support and troubleshooting documentation for any problems encountered.

For the actual CNN that will be employed, it will be built using Keras on top of TensorFlow API in Python. The exact architecture of the model will be determined as the training goes along, adjusting hyperparameters, layers, and loss functions during testing. That being the case, the testing will begin first with a simple model provided in the TensorFlow documentation [16], involving a stack of convolutional (Conv2d) and maxpooling (Maxpool2d) layers, with flattening and dense layers on top.

The testing will initially start with the ReLu activation function and the size of the tensors will be determined based off the grayscale images generated when preparing the dataset. Instead of using RGB color channel like other models, this experiment should be able to just use one channel because the images will be grayscale. The justification for choosing ReLu as the initial activation function is that it is very efficient while also providing accurate results in many neural network models.

D. Training the Model

This is the step where the model is trained, evaluated, architecture adjusted, parameters changed, retrained, evaluated, and etc. until a desired result is achieved. Once all variables have been appropriately adjusted and the architecture finalized, the model will be saved and subsequently evaluated.

All of our datasets utilized an 80/20 training/validation split, which meant that 80% of the dataset was used to train the model and the other 20% was used for tracking the validation accuracy across epochs. In Section IV, you will see a progression in both graphs (accuracy and loss) across multiple training iterations and model versions, starting with the initial, less accurate results.

E. Evaluate the Accuracy and Loss of the CNN model

Once the training is finished, the results will be compiled and documented. We use accuracy and loss as the primary metrics for determining the effectiveness of the model. The end goal will be to determine how accurate the model is at detecting malicious code in a binary, and if it would be effective enough to potentially compete with more traditional signature-based detection methods. To test this on a smaller scale, a handpicked set of 50 malicious files and 50 benign files will be run through a Windows Defender scan to represent a signature-based program, and the results of that will be documented. The same set will be fed to the machine learning model, and the accuracy directly compared to the results of the signature-based method. Overall, for this project we expect that the model will be trained to an accuracy of at least 80%. This expectation is based off similar papers and experiments conducted in [4], [5]. 80% and above will be considered successfully accurate. In comparison, an accuracy of 70-79% will be considered poor, and if the model is not able to be trained to an accuracy higher than 70% it will not be considered usable by these metrics.

IV. EXPERIMENTS AND RESULTS

The initial dataset was gathered successfully as outlined above. The benign files came from Windows apps on a daily computer, specifically the Windows/System32 folder, as well as the SystemApps folder. The malicious files were sourced from a variety of malware repositories on GitHub including 'theZoo' and VX-Underground (both Educational Repositories) [17], [18]. We downloaded these repositories, extracted all of the relevant Win32 archives that contained Windows malware. From there, we began preprocessing as described in Section III-D.

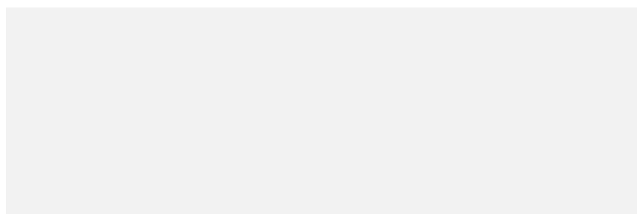
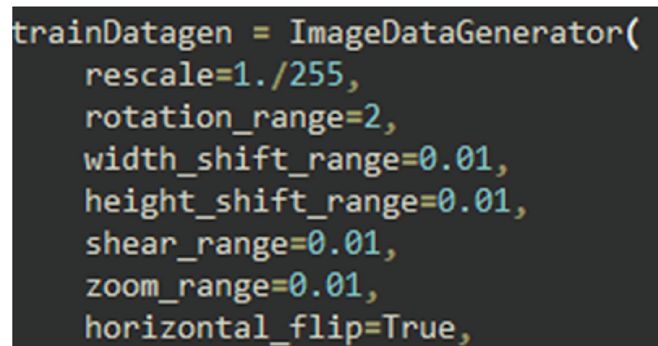


Fig. 4. Interesting Malware Images.

In Figure 4, we have a few examples of very interesting malicious programs displayed as grayscale images. While a lot of the patterns appear similar to human eyes, some examples such as the ones shown here look very unique when compared to the benign samples. The first set of training models were built using a mixture of TensorFlow documentation and GPT-4 AI assistance. The first runs had problems with the model overfitting to the smaller dataset as predicted. These models had no data augmentation, so we added the data augmentation accordingly until the overfitting was resolved to a satisfactory level.



```
trainDatagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=2,
    width_shift_range=0.01,
    height_shift_range=0.01,
    shear_range=0.01,
    zoom_range=0.01,
    horizontal_flip=True,
```

Fig. 5. Data Augmentation.

Figure 5 shows the exact augmentation values used throughout the majority of the training. The data augmentation was added as stated to reduce overfitting, the rescaling and rotation range had the biggest impact on this. The rest of the augmentation involved very small shifts to the width, height, shear and zoom. Based on our experience, larger values than these resulted in the model failing to learn effectively.

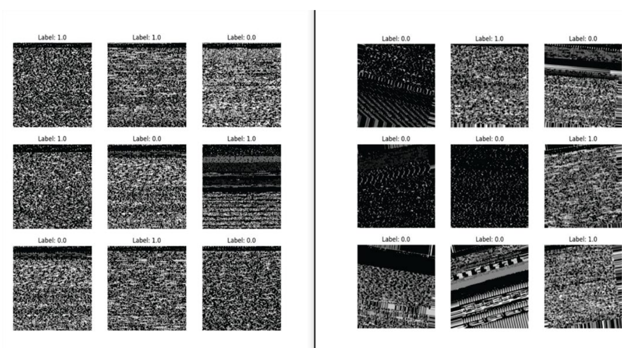


Fig. 6. Data Augmentation Visualization.

Figure 6 shows sample sets with and without augmentation. Each image is labeled, with images marked as 1.0 representing benign samples and those labeled as 0.0 indicating malicious samples.

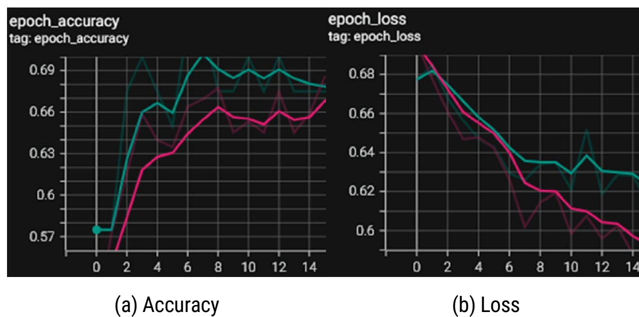
```
optimizer = Adam(learning_rate=custom_learning_rate)

model = models.Sequential()
model.add(layers.DepthwiseConv2D(kernel_size=(3, 3), activation='relu', kernel_regularizer=l2(0.01), padding='same', input_shape=(128, 128, 3)))
model.add(layers.DepthwiseConv2D(kernel_size=(3, 3), activation='relu', kernel_regularizer=l2(0.01), padding='same', input_shape=(128, 128, 3)))
model.add(layers.Conv2D(10, (3, 3), activation='relu', groups=1, input_shape=(128, 128, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(Dropout(0.5))
model.add(layers.Conv2D(10, (3, 3), activation='relu', groups=1, input_shape=(128, 128, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(Dropout(0.5))
model.add(layers.Dense(128, activation='relu'))
#model.add(Dropout(0.5)) - this layer was frequently included and excluded to analyze overfitting
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])

model.fit(trainGenerator, epochs=num_epochs, validation_data=testGenerator, callbacks=[tensorboard_callback])
```

Fig. 7. Initial Model Architecture.

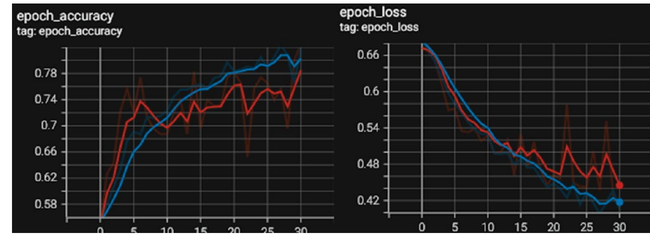
Figure 7 shows the improved architecture developed as a result of the initial test runs. Adam being utilized as the optimizer was experimented with and yielded the best results. After that, the depthwiseConv2D layers were used in addition to normal Conv2d layers in order to resolve some TensorFlow errors that were preventing the model from running, they also yielded a higher validation accuracy than without them. We then added several Conv2d, Maxpooling, Flatten, and Dense layers. While we had originally planned on only using ReLu, a combination of dense layers with ReLu and Sigmoid yielded the best results. The dropout layers were added to reduce the original concerns of overfitting. The results of the first runs on this dataset are shown below in Figure 8:



(a) Accuracy (b) Loss

Fig. 8. Initial Model Accuracy and Loss.

As seen in Figure 8, this model reported an accuracy of 70% and a validation accuracy of 67.5% and was trained across 18 Epochs. Although overfitting was largely mitigated, we considered that this validation accuracy was insufficient for comparison with Windows Defender. Despite adjustments to hyperparameters, data augmentation and model architecture, these changes did not consistently yield a validation accuracy above 70%. As a result, we decided to increase the size of the dataset. To achieve this, additional benign samples were collected from the 'Program Files' folders, while additional malicious files were sourced from Cryptware Malware Database [19] and another GitHub repository titled Malware-Database [20]. These files were preprocessed in the same manner as before, and then the model was trained again on this dataset. Because of the danger of working with live malware samples, this step was done on a separate airtight Ubuntu virtual machine. The second dataset contained 400 benign and 400 malicious samples.



(a) Accuracy (b) Loss

Fig. 9. Second Dataset Model Accuracy and Loss.

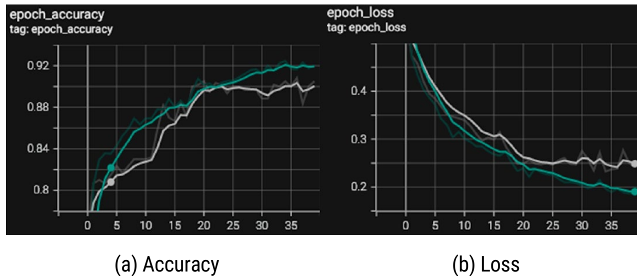
As shown in Figure 9, the second set of test runs achieved an accuracy of 80.2% with a validation accuracy of 78.5% across 30 epochs. This result was more along the metrics of accuracy that we were hoping to achieve. At this point in the testing, we considered moving on to the comparison step. However, given that the increased dataset resulted in nearly a 20% improvement in validation accuracy, we decided to experiment with further increasing the dataset size.

For the third dataset, we sourced additional benign samples from the same sources, roughly 1500 samples. The malware was a little bit different, it appeared as though we had exhausted the main GitHub repositories, and we did not want to include duplicate malware or sources that were not as credible as the ones we used so far. As a result, we ended up sourcing the majority of the rest of the executables from MalwareBazar malware database [21], a large online database with daily contributions used by IT professionals and Cybersecurity researchers for malware analysis. They provide the option to export daily batches of malware containing hundreds of malicious executables. We downloaded a few of these more recent daily batches, and performed the same method before of transferring all the executables, preprocessing them, and saving the images in a third dataset. We ended with 1500 more malicious samples, combining to a total dataset of 3000 images. We used a training and test split of 80/20 again, which meant that 2400 were used for training and the other 600 were used as the validation set.

Once we had gathered this dataset, we trained the same CNN models on the new data, with slightly adjusted batch sizes, epochs and learning rates to account for the increased dataset size. While this did increase the accuracy, it was not as big as the first jump. These models trained to an accuracy solidly in the upper 80th percentile, some of them flattening out at around 90% validation accuracy.

As shown in Figure 10, this model evened out at 91% accuracy and 90% validation accuracy, with .189 training loss and .24 validation loss across 40 epochs. This was deemed acceptable for the rest of the experiment, yet we decided to try increasing the dataset size one more time in an attempt to squeeze out a little more validation accuracy. To do this, we

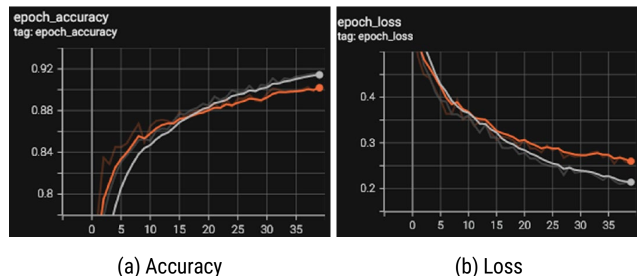
obtained even more benign files from the same sources as before, and more malicious binaries from MalwareBazar daily malware batches. We then sorted through, copied over all the executables, and ran them through the preprocessing script to convert them into the 128x128 grayscale images. At this point, we had increased the size of the dataset to its final version of 4,188 images. Of these, 1,821 were benign, and the remaining 2,367 were malicious. This imbalance was justified by the fact that false negatives are significantly worse for malware detection than false positives, so we would rather the model be slightly biased in favor of detecting malware. Instead of an exact 80/20 split, we used 400 benign and 400 malicious files for the validation set, in order to have an equal balance of both images in the test set.



(a) Accuracy (b) Loss

Fig. 10. Third Dataset Model Accuracy and Loss.

The final set of training runs that we did used the same architecture as before, with again slight adjustments to learning rate and epoch amounts. This set of runs resulted in the following model which we ended up using for the remainder of the experiment, referred to as Model 1:



(a) Accuracy (b) Loss

Fig. 11. Model 1 Accuracy and Loss.

Seen in Figure 11, Model 1 reported an accuracy of 91%, and a validation accuracy of 90% like the previous model, but this time with a training loss of .211 and a validation loss of .253 across 39 epochs. While the loss was slightly higher, we felt that this model showed the least amount of overfitting, with a closer loss gap from training to validation than the previous models. The increase in dataset size did not significantly improve the validation accuracy, and we determined that a validation accuracy of 90% was sufficient for the remainder of the experiment. We acknowledge that with datasets of substantially larger scale, it is certainly possible to achieve validation accuracy exceeding the 90% in this project. After we attempted tweaking hyperparameters, data augmentation, and learning rate, this model continued to yield the best overall validation accuracy with the lowest loss deviation across the validation set.

In a final effort to explore further improvements to the machine learning process and enhance accuracy, we proceeded to train an additional CNN model, with GPT-4 managing the training process entirely unassisted [22]. We prompted it with the dataset and task at hand, and it built a model with the following architecture:

```
# Model Building
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 1)),
    MaxPooling2D(2, 2),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Flatten(),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])
```

Fig. 12. GPT Model Architecture.

Figure 12 shows the GPT-4 model architecture. This model will be referred to as Model 2. One thing that is interesting to note about Model 2, is that the architecture is much simpler. GPT-4 decided to replace the depthwise Conv2d layers with normal Conv2d layers, utilizing fewer nodes per layer. It also did a couple of interesting things as noted below in Figure 13:

```
# Callbacks
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=3,
    verbose=1,
    restore_best_weights=True
)
```

```
reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.2,
    patience=2,
    verbose=1,
    min_lr=0.00001
)
```

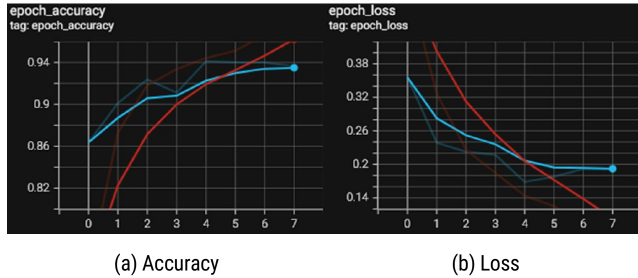
(a) Accuracy (b) Loss

Fig. 13. GPT Model Callbacks.

Seen in Figure 13, GPT-4 implemented two different techniques that were not originally included in Model 1: automated early stopping, and a dynamic learning rate adjustment. The automated early stopping callback led the model to stop training after the model had sufficiently converged, in order to prevent the model from overtraining and overfitting on the training data. The dynamic learning rate adjustment would lead to a decrease in the learning rate as the model converged, resulting in finer adjustments and higher accuracy. We ran a test run, and fed the results back into GPT-4. As a prompt, we fed it both the loss and accuracy graphs, as well as the full TensorFlow console logs from the training. After an adjustment of slightly increasing the dropout from 0.5 to 0.6, we were left with the final version of Model 2. While this was originally just done to see if any further improvements could be made to Model 1, Model 2 ended up yielding significantly more accurate results as shown in Figure 14:

Figure 14 shows the accuracy and loss graphs for this model. While on first glance it appears as though the model is showing signs of overfitting, the early stopping callback

effectively yielded stable results. This model achieved a stable accuracy of 94.6% and a validation accuracy of 93.5%, reflecting a notable 3% improvement over Model 1. Although the initial plan was to compare Model 1 to Windows Defender, we decided to compare Model 1, Model 2, and Windows Defender together. This approach not only provides insight into the accuracy and viability of machine-learning based malware detection but also offers some insight into the feasibility of using generative AI to train its own machine-learning models, with a potential use case in antivirus software.



(a) Accuracy (b) Loss

Fig. 14. Model 2 Accuracy and Loss.

After we had trained Models 1 and 2, we determined the training phase was complete and moved on to the comparison phase of the experiment. For this step, a dataset of 50 benign and 50 malicious files was gathered for a small-scale assessment of both of the models' accuracy in relation to Windows Defender. The 50 benign files were sourced from a mixture of Windows system apps and benign freeware. The 50 malicious files were sourced from MalwareBazar [21] daily batches all within the past few months, with the aim of including more recent and novel malware. The initial set was scanned through Windows defender, which yielded the following results:

TABLE I. Windows Defender Scan Results

File Type	Total Files	Flagged
Benign	50	0
Malicious	50	46

Table I presents the results of the Windows Defender scan, which yielded expected results. One advantage to Windows Defender is that it has a very low number of false positives on average, due to the need for specific signatures to exist in a malware database in order to flag a binary. In this case, it correctly flagged 0 out of 50 benign samples, which aligns with our expectations. Additionally, Windows Defender effectively detected 46 out of 50 malicious files, achieving a 92% detection rate. Overall, considering both benign and malicious files, Windows Defender demonstrated a total accuracy of 96%.

Now, to compare this to Model 1 and Model 2, this same 100 file dataset was converted through the same preprocessing technique as before, but this time we loaded the saved models to plot a prediction on each of the 100 test files. First, we plotted the predictions for Model 1 on the malicious and benign samples:

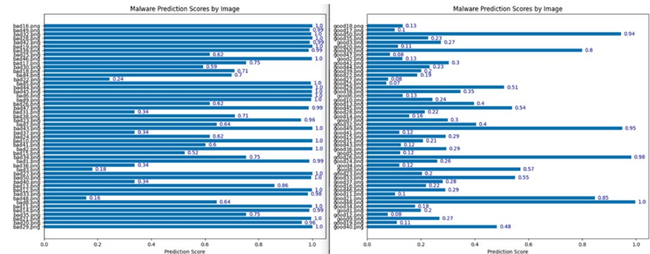


Fig. 15. Model 1 Prediction Score.

As shown in Figure 15, these graphs represent the prediction labels of Model 1 for both the malicious and benign test sets. The bar graphs span from 0 to 1, where 0 and 1 indicate benign and malicious, respectively. The left graph shows the malicious files, and the right graph represents the benign files. Scores above 0.5 were flagged as malware, and any files labeled at or below 0.49 were classified as benign. Overall, this model yielded the following results:

TABLE II. Model 1 Accuracy Score

File Type	Total Files	Flagged
Benign	50	10
Malicious	50	43

Table II presents the detection rates achieved by Model 1 on the same dataset. While the malicious file detection was somewhat comparable to that of Windows Defender, Model 1 yielded a significant increase in false positives. This outcome was somewhat expected as previously mentioned, as Windows Defender requires specific malware signatures to flag a binary. Model 1 flagged 43 out of 50 malicious files, achieving a 86% detection rate. Including the benign samples, the overall accuracy of Model 1 was 83%.

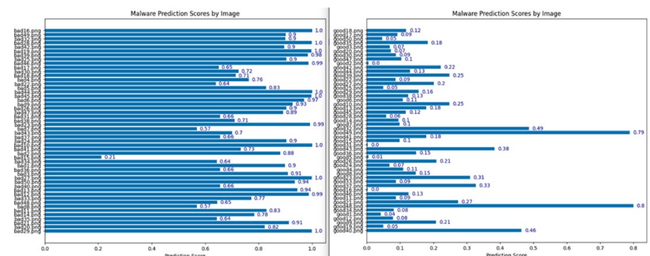


Fig. 16. Model 2 Prediction Score.

After this, we used the exact same test set to make predictions with GPT-4 generated Model 2. Figure 16 displays these graphs representing the prediction scores for both the

malicious and benign sets, measured on a scale of 0 to1, where scores above 0.5 indicate malicious and scores at or below 0.49 indicate benign. The results were exceptional and are summarized in the following table:

TABLE III. Model 2 Accuracy Score

File Type	Total Files	Flagged
Benign	50	2
Malicious	50	49

Table III presents that the GPT-4 generated model achieved remarkable accuracy, identifying 49 out of 50 malicious files, achieving a 98% detection rate. In comparison to Model 1, it demonstrated significantly fewer false positives, flagging only 2 out 50 benign files as malicious. Overall, Model 2 achieved a total accuracy of 97%. This performance not only surpassed Model 1 but also exceeded that of Windows Defender by a notable margin.

TABLE IV. Final Accuracy Results

	Model 1	Model 2	Windows Defender
Benign %	80%	96%	100%
Malicious %	86%	98%	92%
Total Acc %	83%	97%	96%

The final results of all three methods are shown in Table IV. Although our initial hypothesis was that the machine learning approach could yield results comparable to Windows Defender, we did not expect the CNN models to outperform it. It is also noteworthy that GPT-4 is not only capable of building and training a machine-learning model but it can do so with a nuanced and complex dataset, such as binary grayscale images, and achieve a level of accuracy higher than traditional methods.

V. CONCLUSION

At the beginning of this project, our primary objectives were to address the following questions: To what extent could a CNN differentiate between benign and malicious Windows executables when trained on a custom dataset, and how accurate is it in predicting harmful Windows binaries? How does its performance compare to that of conventional malware detection such as Windows Defender?

Based on the results obtained throughout this experiment, we conclude that CNN models are highly effective at distinguishing between benign and malicious Windows executables. They not only achieve results comparable to Windows Defender but also, in some respects, surpass it. The comparable accuracy of both methods, combined with the advantages that machine learning models have such as the ability to detect novel malware and the lack of constant database updates, leads us to the conclusion that machine learning-based malware detection methods, specifically CNNs, are capable of detecting malware at a comparable rate to conventional malware detection methods while providing several additional advantages

VI. FUTURE WORK

In addition to our findings, we also determined that generative AI is capable of building and self-training machine learning models without human intervention. While this was not originally one of the research goals, it was interesting enough to include in our results. Further work could be done in determining the effectiveness of generative AI in malware detection, such as possibly the ability for it to consistently retrain itself on newly scanned malware.

While the current industry around signature-based detection is well established and successful, it is understandable that machine learning approaches might not be so easy to implement. One proposed application of this research would include implementing a mixture of signature-based detection and a CNN model. This would potentially provide the opportunity to take advantage of the accuracy of CNNs while still maintaining the advantages of signature-based detection.

REFERENCES

[1] SonicWall. (2023) Mid-year 2023 cyber threat report. [Online]. Available: https://www.sonicwall.com/medialibrary/en/white-paper/mid-year-2023-cyber-threat-report.pdf

[2] Ö. A. Aslan and R. Samet, “A comprehensive review on malware detection approaches,” *IEEE access*, vol. 8, pp. 6249–6271, 2020.

[3] J. Scott, “Signature based malware detection is dead,” *Institute for Critical Infrastructure Technology*, 2017.

[4] M. E. Eren, M. Bhattarai, K. Rasmussen, B. S. Alexandrov, and C. Nicholas, “Malwaredna: Simultaneous classification of malware, malware families, and novel malware,” in *2023 IEEE International Conference on Intelligence and Security Informatics (ISI)*. IEEE, 2023, pp. 1–3.

[5] A. F. Alshmarni and M. A. Alliheedi, “Enhancing malware detection by integrating machine learning with cuckoo sandbox,” *arXiv preprint arXiv:2311.04372*, 2023.

[6] K. Hanifi, R. F. Fouladi, B. G. Unsalver, and G. Karadag, “Software vulnerability prediction knowledge transferring between programming languages,” *arXiv preprint arXiv:2303.06177*, 2023.

[7] W.-C. Lin and Y.-R. Yeh, “Efficient malware classification by binary sequences with one-dimensional convolutional neural networks,” *Mathematics*, vol. 10, no. 4, p. 608, 2022.

[8] M. Sakhai and M. Wielgosz, “Modern cybersecurity solution using supervised machine learning,” *arXiv preprint arXiv:2109.07593*, 2021.

[9] S. Anand, B. Mitra, S. Dey, A. Rao, R. Dhar, and J. Vaidya, “Malite: Lightweight malware detection and classification for constrained devices,” *arXiv preprint arXiv:2309.03294*, 2023.

[10] C. Huang, G. Zhu, G. Ge, T. Li, and J. Wang, “Fastbcsd: Fast and efficient neural network for binary code similarity detection,” *arXiv preprint arXiv:2306.14168*, 2023.

[11] R. Mehta, O. Jurečková, and M. Stamp, “A natural language processing approach to malware classification,” *Journal of Computer Virology and Hacking Techniques*, vol. 20, no. 1, pp. 173–184, 2024.

[12] A. Muzaffar, H. Ragab Hassen, H. Zantout, and M. A. Lones, “Droid-dissector: A static and dynamic analysis tool for android malware detection,” in *International Conference on Applied CyberSecurity*. Springer, 2023, pp. 3–9.

[13] Z. Liu, Z. Zhang, S. Ma, D. Liu, J. Zhang, C. Chen, S. Liu, M. E. Ahmed, and Y. Xiang, “Semdiff: Binary similarity detection by diffing key-semantics graphs,” *arXiv preprint arXiv:2308.01463*, 2023.

[14] A. Sharma and G. Phonsa, “Image classification using cnn,” in *International Conference on Innovative Computing and Communication (ICICC)*, 2021.

[15] H. Nguyen, F. Di Troia, G. Ishigaki, and M. Stamp, “Generative adversarial networks and image-based malware classification,” *Journal of Computer Virology and Hacking Techniques*, vol. 19, no. 4, pp. 579–595, 2023.

[16] Convolutional neural network (cnn) tutorial. [Online]. Available: https://www.tensorflow.org/tutorials/images/cnn

[17] ytisf.thezoo github repository. [Online]. Available: https://github.com/ytisf/theZoo

[18] vxunderground. vxunderground github repository. [Online]. Available: https://github.com/vxunderground

[19] Cryptware malware database github repository. [Online]. Available: https://github.com/topics/malware-database

[20] Dharkonsk, “Dharkonsk malware database GitHub repository,” GitHub, [Online]. Available: https://github.com/DharkonSK/Malware-Data-Base [Accessed: 6/30/2024]. (Repository no longer available).

[21] Malwarebazar database. [Online]. Available: https://bazaar.abuse.ch/

[22] Openai. chatgpt 4, version 4.0. [Online]. Available: https://openai.com/chatgpt/