AIMS Mathematics, 9(6): 15223–15245.
DOI: 10.3934/math.2024739
Received: 15 March 2024
Revised: 07 April 2024
Accepted: 17 April 2024
Published: 26 April 2024

https://www.aimspress.com/journal/Math

Research article

Leveraging deep learning and image conversion of executable files for effective malware detection: A static malware analysis approach

Mesut GUVEN*

TOBB University of Economics and Technology, TR-06560 Ankara, Turkey

* **Correspondence:** Email: mesuttguven@gmail.com, mesutguven@etu.edu.tr;
Tel: +905305224189.

Abstract: The escalating sophistication of malware poses a formidable security challenge, as it evades traditional protective measures. Static analysis, an initial step in malware investigation, involves code scrutiny without actual execution. One static analysis approach employs the conversion of executable files into image representations, harnessing the potency of deep learning models. Convolutional neural networks (CNNs), particularly adept at image classification, have potential for malware detection. However, their inclination towards structured data requires a preprocessing phase to convert software into image-like formats. This paper outlines a methodology for malware detection that involves applying deep learning models to image-converted executable files. Experimental evaluations have been performed by using CNN models, autoencoder-based models, and pre-trained counterparts, all of which have exhibited commendable performance. Consequently, employing deep learning for image-converted executable analysis emerges as a fitting strategy for the static analysis of software. This research is significant because it utilized the largest dataset to date and encompassed a wide range of deep learning models, many of which have not previously been tested together.

Keywords: artificial intelligence; deep learning; convolutional neural networks; autoencoders; transfer learning; malware detection; executable files
Mathematics Subject Classification: 68T45, 68U20

1. Introduction

Malware or malicious software, is a persistent threat to computer systems and networks. Malware is designed to perform harmful actions, such as stealing sensitive data, disrupting system operations, or gaining unauthorized access to resources. As malware continues to evolve and become more sophisticated, it is essential to have effective methods to detect and analyze it. Malware analysis methodology relies on two main approaches, i.e., static analysis and dynamic analysis [1].

Static malware analysis is useful for detecting known types of malware and identifying code patterns that may be indicative of malicious behavior. This approach is based on the idea that certain characteristics of malware are consistent and can be identified without running the code. It uses various tools and techniques to analyze the code, such as disassemblers, decompilers, and signature-based detection. Dynamic malware analysis, on the other hand, is useful for detecting previously unknown malware and understanding how it operates [2, 3]. This approach involves executing the malware sample in a controlled environment, such as a virtual machine or sandbox, to observe its behavior and identify its effects on the system. Dynamic analysis involves using various tools and techniques to monitor the behavior of the malware, such as system call monitoring, memory analysis, and network traffic analysis. Both static and dynamic malware analysis have their advantages and disadvantages. One advantage of static analysis is that it is faster and less resource-intensive than dynamic analysis, making it suitable for large-scale analysis of malware samples. While dynamic analysis can provide more detailed and comprehensive information about the malware's behavior, it is more time-consuming and requires more resources. So, this is an accuracy versus agility problem and effective malware analysis requires a combination of these approaches, as well as a deep understanding of the malware landscape and the techniques used by attackers. By using these methods, security professionals can identify and mitigate malware threats to protect computer systems and networks. In this work, we focus on the static analysis of the software at the code level and make use of the great potential of deep learning in image classification tasks.

In this paper, we focus on static malware analysis. To detect malware, we conducted experiments by applying convolutional neural networks (CNN) and transfer learning models to image-converted forms of the portable executable data. Image-based malware analysis is structurally different from traditional malware analysis since CNNs networks accept only two-dimensional data. So, in the image-based malware analysis, first, executable files are transformed into a two-dimensional image-like format. Then, the model is fed and trained on these converted inputs. In the previous paragraphs, introductory information has been provided about malware analysis strategies and deep learning concepts.

Deep learning is a sub-field of machine learning that utilizes artificial neural networks to learn from and make predictions based on complex data. Deep neural networks are composed of many layers of interconnected neurons, and, unlike classical feature extraction-based machine learning models they can automatically extract and learn useful features from the data through a process called backpropagation. Deep learning has achieved state-of-the-art results in many domains, including computer vision, natural language processing (NLP), and speech recognition. However, deep learning also has some limitations. It requires large amounts of data and computational resources to train the models effectively. So, it is not effective to implement deep neural networks on small data. But, with the help of knowledge transfer, deep neural networks can be trained on a large dataset of labeled examples and adjust its parameters to minimize the difference between its predictions and the true labels. Once trained, the network can be used to make predictions on new, unseen data from a very different domain. This property is known as transfer learning.

Transfer learning is a machine learning technique that involves leveraging knowledge from one domain to improve performance in a different but related domain. In transfer learning, a model is first trained on a source task and then fine-tuned on a target task. The idea behind transfer learning is that many tasks share common features, and these features can be learned from one task and reused for another task. This can be especially useful when the target task has limited data available or when the

model needs to be trained quickly. Transfer learning has been successfully applied to various domains. It is a powerful technique that can help to improve model performance, reduce training time, and overcome the data scarcity problem. There are different types of transfer learning, i.e., pre-training, domain adaptation, multi-task learning, etc. In pre-training, a model is trained on a large, diverse dataset, typically in an unsupervised manner, to learn general features that can be reused for other tasks. In domain adaptation, the model is trained on a source domain and then adapted to a target domain that is different but related. For example, a model trained on images of cats and dogs could be adapted to recognize images of lions and tigers. And, in multi-task learning, a model is trained on multiple tasks simultaneously, allowing it to learn features that are useful for all tasks. Several prominent deep learning models widely employed for transfer learning include VGG16, ResNet, InceptionV3, and MobileNet. A shared characteristic among these models is their training on extensive image datasets, notably ImageNet which is recognized as the largest repository of images.

In the pre-processing part, the main task is to convert portable executable data into a proper input type for CNN and transfer learning models. This process consists of three sub-steps: pixel conversion, reshaping, and resizing. Pixel conversion is a simple process of assigning a value between 0 and 255 to every byte, and this value corresponds to the pixel intensity. This step converts the binary into a one-dimensional stream of pixels. To enable transfer learning and computer vision tasks, the pixel streams must be reshaped into a two-dimensional format. The width and height of the image are determined by the file size after conversion. The height of the image is calculated by dividing the total number of pixels by the width. After reshaping the image, data are resized into a fixed size since transfer learning models only accept fixed-sized inputs such as 224 by 224 by 3.

1.1. Goals of this work

The primary objectives of this research were to explore, evaluate, and advance the application of deep learning techniques in the realm of malware detection through the use of static analysis. The specific goals can be stated as follows:

(1) **Investigate image-converted executable analysis:** Evaluate the feasibility of converting executable files into image representations for static analysis.

(2) **Assess deep learning models:** Evaluate the performance of deep learning models, both tailor-made and pre-trained, on the task of detecting malware from image-converted executables.

(3) **Demonstrate effectiveness and advocate integration:** Showcase the effectiveness of deep learning models in static analysis workflows for detecting sophisticated malware threats. Advocate for the integration of image-based preprocessing techniques to enhance security measures against evolving threats.

1.2. Contribution

By addressing these goals, this study was designed to provide a comprehensive understanding of the potential and limitations of leveraging deep learning for effective malware detection through static analysis.

Another important contribution of this work is the application of pre-trained models on our Trapmine dataset, which is the largest set tested in the literature for static malware analysis tasks.

Using pre-trained models is crucial for a couple of reasons. First, malware training datasets are scarce and not large enough to train deep learning models. Even though tech industry giants such as Microsoft have relatively large malware datasets for their own commercial usage, they are still not large enough to compare with the ImageNet dataset. Second, the main problem with malware datasets is that they quickly become outdated as attackers frequently modify their tactics. So, transferring knowledge from a large image dataset to an image-converted malware dataset solves the data scarcity problem and the zero-day malware representation problem.

2. Related work

In the literature, there are works based on the features extracted from both static and dynamic approaches. However, we only mention the literature for static features since this work focuses on static malware analysis. Static features are those that are derived from an examination of the source code or related information associated with an application. The process of analyzing these features is known as static analysis, as stated in [4, 5]. For example, when examining Android applications, the primary focus of static analysis is the APK file, which serves as the installation package for the application. By decompiling APK files, files such as AndroidManifest.xml and small files can be obtained. These files can then be further analyzed to extract a collection of static features, such as permissions, API calls, opcodes, and various other components. From this point of view, the literature review is filtered and divided into three categories of deep learning methods, i.e., NLP, CNNs, and transfer learning.

NLP is one of the methods used to analyze static features. In a previous study [6], an NLP model based on the bag-of-words approach is used to detect whether the opcodes belong to malicious software. In another study [7], ASCII strings were extracted from executable files and used to analyze malware via the NLP approach. In another study [8], the behavior of executables and processes, such as file read/write activities, process creation, network connections, or registry modifications were converted to graph streams, and abnormal patterns were searched. To achieve this detection, behavioral graph streams were converted into documents, and embedded by using NLP models. Then, an outlier detection technique was applied to the high-dimensional vector representation of the documents. the authors claim that it can be used in a real-world scenario alongside a large-scale international enterprise's security information and event management systems known as SIEM systems for short, which generates over 3 TB of logs. In another study [9], to create feature vectors that can capture the information from both the Android manifests and Dalvik executables files, the NLP method was applied to document embeddings. These feature vectors were then used to train binary classifiers that can accurately distinguish between benign and malicious Android applications.

Another approach for malware analysis through the use of static features is the use of CNN models. For example, in a study, an autoencoder model which is a special form of CNN was used to detect malware [10]. Autoencoder networks are generally designed in a symmetrical form to code and decode the input image. The network is fed and targeted with the same image. This design property allows autoencoders to be a good fit for denoising and dimensionality reduction tasks. In this work, greyscale image representations of malware were used with an autoencoder network in a deep learning model. The proposed model achieved a 96% classification accuracy for malware from benign software. In 2018, Le and colleagues [11] introduced a method for identifying malware by transforming the entire

binary file into a grayscale image, followed by training a CNN algorithm to classify the malware. The effectiveness of this approach was assessed by using 10,568 binary files, which resulted in a 98.8% accuracy rate. Kim et al. [12] proposed a method for detecting zero-day attacks in malware by using autoencoders and generative adversarial networks (GANs). GANs have two neural networks that act as a generator and the discriminator [13]. The discriminator network tries to distinguish real samples from ones created by the generator. The authors pre-processed the data by representing the code used for the attack as an image matrix and trained a denoising autoencoder (DAE) to extract relevant features from the bottleneck layer. The decoder of the DAE was used as the data generator of the GAN, and the discriminator of the GAN was trained to detect the adversarial examples generated by the decoder. The proposed model achieved an accuracy of 98% on the task of detecting zero-day attacks on the dataset from the Microsoft Malware Classification Challenge, outperforming conventional models such as support vector machines, decision trees, the random forest, K-nearest neighbors (KNN), naive Bayes, and quadratic discriminant analysis that were tested.

Leveraging transfer learning models is another technique that is used to detect malware. Most of the works claim that the pre-trained models that have been trained on a vast image dataset, i.e., the ImageNet, can distinguish the malware from benign software. There are some works on this issue. For example, in a previous study [14], researchers suggested using a deep transfer learning method to classify malware images. To extract features, they utilized various CNN models including VVG16, VVG19, ResNet50, and Google's InceptionV3. By implementing this approach, they were able to achieve a classification accuracy of 98.92% for the Malimg dataset and 93.19% for the Microsoft dataset. In another study [15], a technique for detecting malware was introduced, and it involves employing a hybrid model utilizing multiple transfer learning models. The feature vector is extracted by using two deep learning models, namely AlexNet and ResNet-152, which are then employed to classify the malware variants by using a fully connected neural network. To evaluate the efficacy of the approach, two publicly available datasets, namely the Malimg and Microsoft BIG 2015 datasets, were utilized. The proposed technique achieved an accuracy of 97.78% for the Malimg dataset. In a recent study, the same dataset was used with pre-trained models, particularly, VVG-16 and ResNet-50 models [16]. In this study, two models were used in a sequentially stacked form. The second model uses the first model's output as the input target label. The authors claimed that this methodology results in 100% accuracy.

Malware detection through the use of image-converted representations of software is a hot research area. For example, in a recent paper, the authors proposed a novel method for Android malware detection by using unbalanced heterogeneous graph embedding, demonstrating its effectiveness on the DREBIN dataset with an accuracy of 0.9360 and an F1 score of 0.9360, and thus surpassing existing state-of-the-art approaches [17].

3. Dataset

To train the model, we used a relatively larger dataset than the publicly available malware datasets. Our dataset is a subset version of the data used for the design of a commercial XDR product called Trapmine, which has been used in governmental and non-governmental institutions for years. The dataset used for to train the models in this work has 10,217 benign and 7,000 malign samples. Some of the samples were acquired from publicly available sources such as the Google app store [18] and

VirusShare [19]. The rest of the samples were taken from firms' confidential data which consist of samples that have been analyzed and classified by the Trapmines' machine learning module. In addition, to verify the labels, we picked a random selection of software and crosschecked their hash signatures from VirusTotal [20]. The raw form of the dataset is the assembly code version. In the pre-processing phase, the binaries are extracted from the raw data and converted into RGB format. Since this dataset is still being used, we do not provide all of them. But, to give a sense of the data, the test set will be made publicly available.

The data used for testing purposes constitute a mixture of several datasets, particularly, the Malimg and MaleVis datasets. The Malimg dataset comprises images from different types of malware [21]. The images were created by transforming the 1D malware binaries into image features. This process is described in [21]. The second dataset used for the tests is the Malevis dataset [22]. The MaleVis dataset is publicly available as images of 224×224 pixels and 300×300 pixels [23]. These two popular datasets have been widely used for the multiclass classification of malware via a machine learning method. The Malimg dataset is highly imbalanced in terms of class distribution and the MaleVis dataset is a well-balanced dataset, but is relatively smaller. Generally, both of the datasets were mixed to form a single dataset which has all of the classes from the MaleVis dataset and 5 classes from the Malimg dataset. We also used these two datasets to test the models' performance.

4. Methodology

In this paper, a framework for detecting malware software is presented. The proposed framework involves first, converting portable executables (PEs) into RGB image formats, then leveraging deep learning algorithms to classify the image converted PE samples. So, the methodology in this work can be divided into two main steps: Image conversion of the PEs and model training.

First, we applied the training data which is relatively large malware data. For the training data, we used the Trapmine dataset, which has a total of 17,217 PEs. In this step, all samples are converted into RGB images, and converting 17,217 PEs to RGB images required several hours with Python 3 and the Google Compute Engine (TPU). Out of a total of 17,217 samples, 13,972 PEs were used for training and validation. In the data partitioning process, we adhered to the 80% to 20% rule, allocating 11,172 images for training and reserving the remaining 2,800 images for validation. Additionally, 3,245 samples were set aside exclusively for testing purposes. Within the training and validation sets, a balanced distribution was maintained for both classes; for instance, the validation set comprised an equal number of 1.400 benign and 1.400 malignant samples.

Second, the RGB images were used to train deep learning models. Various deep learning models were created such as basic CNN models and autoencoder models. Along with these models, some popular and well-known pre-trained models are also tested. All models were trained and validated on the Trapmine dataset. Then, the models' overall performance on new unseen data was evaluated on the Malevis and Malimg datasets. During the training, for pre-trained models, it required approximately 25 minutes for one epoch with the Python 3, Google Compute Engine (TPU).

4.1. Image conversion of the PEs

In general, there exist multiple methods for transforming binary code into images [24]. In this study, we utilized a simple and direct approach to visualize the pixel errors (PEs). First, the binary values are

organized into groups of 8 bits and then converted into their decimal equivalent as shown in Eq (4.1).

$$B0 \cdot 1 + B1 \cdot 2 + B2 \cdot 4 + B3 \cdot 8 + B4 \cdot 16 + B5 \cdot 32 + B6 \cdot 64 + B7 \cdot 128. \tag{4.1}$$

Subsequently, each set of 8 bits is assigned sequentially as the red, green, and blue components of a pixel as shown in Eq (4.2).

$$\begin{aligned} \text{Red} &= B0 \cdot 1 + B1 \cdot 2 + B2 \cdot 4 + B3 \cdot 8 + B4 \cdot 16 + B5 \cdot 32 + B6 \cdot 64 + B7 \cdot 128 \\ \text{Green} &= B8 \cdot 1 + B9 \cdot 2 + B10 \cdot 4 + B11 \cdot 8 + B12 \cdot 16 + B13 \cdot 32 + B14 \cdot 64 + B15 \cdot 128 \\ \text{Blue} &= B16 \cdot 1 + B17 \cdot 2 + B18 \cdot 4 + B19 \cdot 8 + B20 \cdot 16 + B21 \cdot 32 + B22 \cdot 64 + B23 \cdot 128. \end{aligned} \tag{4.2}$$

The process of visualizing PEs as RGB images is depicted in Figure 1.

This process is iterated until all available binary values are taken into account. The dimensions of the resulting RGB image were determined to be equal in width and height. So, the dimensions of the RGB image depend on the PEs file size. To create a squared RGB image that has equal width and height dimensions, all binaries are used except the last vestigial binaries. The vestigial part is skipped and not taken into account. Figure 2 showcases the converted RGB image representations of files of varying sizes, encompassing both benign and malicious files.

The dataset had a wide range of file sizes, spanning from 1.5 bytes to 8,150,341.5 bytes, demonstrating its diversity. As an illustration, an image with a resolution of 224 x 224 pixels corresponds to a file size of approximately 18,816.0 bytes.

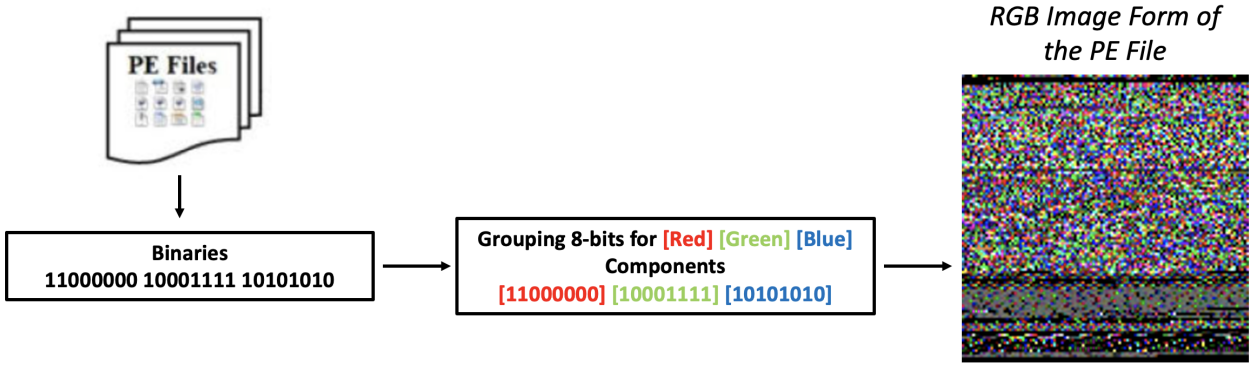

Figure 1. Visualization process for the PEs.

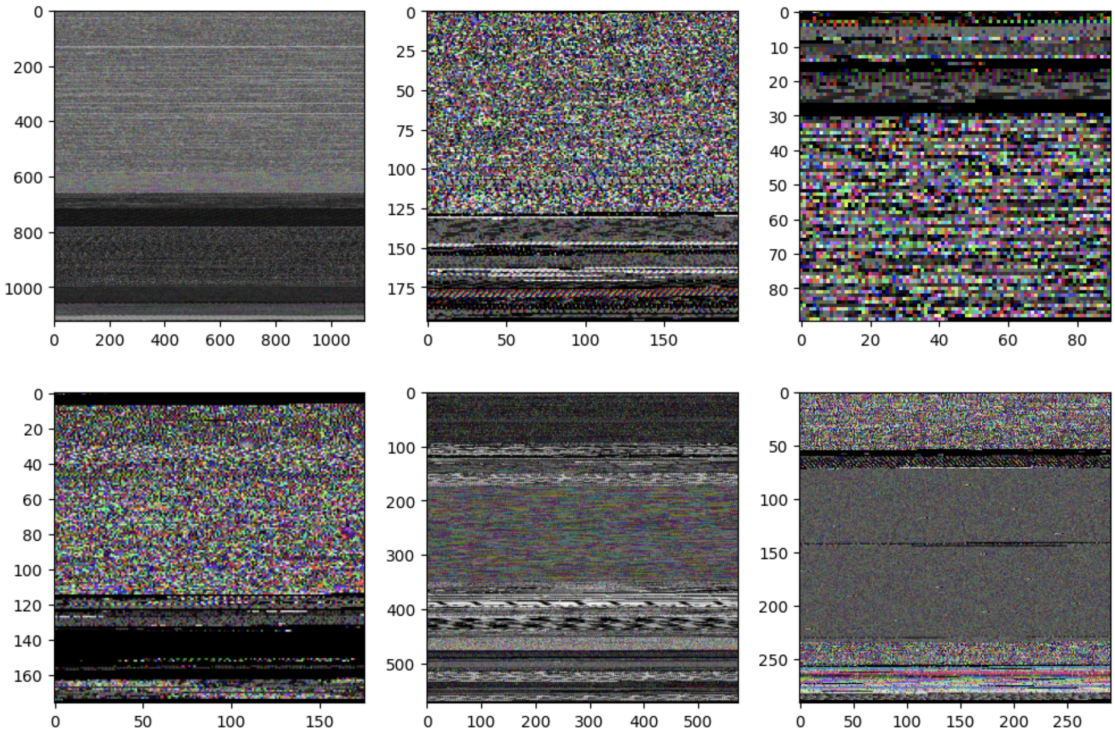

Figure 2. RGB image representations of PEs files at varying sizes. The upper row exhibits RGB images corresponding to benign files, while the lower row displays RGB images associated with malicious files.

4.2. Models used for malware classification

In this paper, we present a novel framework that has been designed for the analysis of converted representations of PEs as RGB images, leveraging various deep learning models. Specifically, our focus was exclusively on CNN-based models and CNN-based pre-trained models. This choice stems from the widely acknowledged suitability of CNN architecture relative for image classification tasks.

The initial model implemented in our study is a fundamental CNN model, comprising several sequential layers. Subsequently, we introduced a second model, also based on CNN architecture, distinguished by a greater depth in layering relative to the preceding model. Our third model adopts the autoencoder architecture, serving as a feature extractor. Lastly, we employed a collection of renowned pre-trained models that are widely recognized in the literature, further emphasizing their prevalence.

4.2.1. Basic CNNs

In this subsection, we provide a detailed analysis of the architecture of our first model. The model follows a sequential design, which allows for the orderly stacking of various layers to extract meaningful representations from the input images. The architecture of the model is presented in Figure 3.

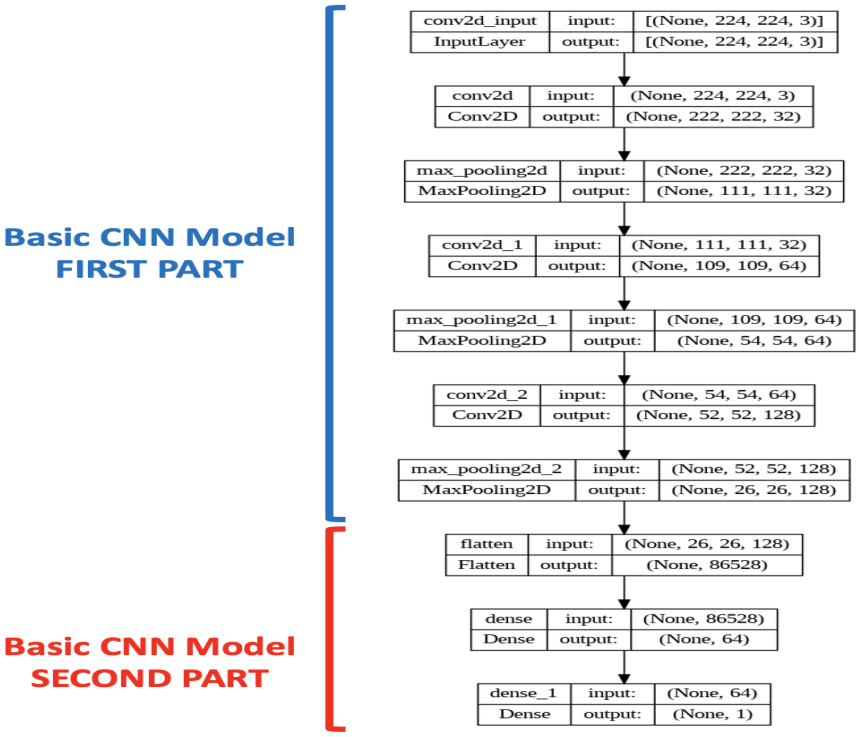

Figure 3. Architecture of the basic CNN.

The initial layer of our model is a convolutional layer (Conv2D) with 32 filters, each of size 3 x 3. This layer utilizes the rectified linear unit (ReLU) activation function to introduce non-linearity into the network and enhance its capacity to capture complex patterns in the data. The input shape of the images is defined by a parameter in the code, and it affords flexibility in its accommodation of different image dimensions. In this model, we use 224 x 224 pixels as the input image size. All images were converted to this fixed resolution by using the interpolation property of the TensorFlow.

Following the Conv2D layer, a MaxPooling2D layer with a pool size of 2 x 2 is employed to downsample the spatial dimensions of the feature maps. This process aids in reducing the model's computational complexity while retaining the most salient features. Next, another Conv2D layer with 64 filters and a 3 x 3 filter size is included to capture higher-level features. Subsequently, a MaxPooling2D layer is applied once again, reducing the feature map dimensionality. To further enhance the model's capacity for abstraction, a third Conv2D layer with 128 filters and a 3 x 3 filter size is included. This layer captures increasingly complex and abstract representations of the input images. Another MaxPooling2D layer is then applied, further downsampling the feature maps.

Following the convolutional layers, a flattened layer is introduced to transform the multi-dimensional feature maps into a one-dimensional vector, which is compatible with the subsequent fully connected layers. The model continues with two fully connected (Dense) layers. The first Dense layer consists of 64 units, utilizing the ReLU activation function to introduce non-linearity and enable complex feature combinations. The final Dense layer, with a single unit, employs the sigmoid activation function to produce a binary classification output, indicating the likelihood of the input image belonging to a specific class.

4.2.2. CNNs

We also employed another simple CNN model which is presented in Figure 4. This second CNN model was constructed by adding some extra layers between the first part of the basic CNN model and the second part of the basic CNN model.

Both models utilize sequential architectures with Conv2D and MaxPooling2D layers to facilitate feature extraction through three Conv2D layers. However, in the second model, we chose to introduce additional Conv2D layers with the expectation that this augmentation would enhance accuracy. The rationale behind incorporating more convolutional layers was to allow the model to capture more intricate and abstract features from the input images, potentially leading to improved performance.

In addition to the architectural design, the choice of optimizer plays a crucial role in training the deep learning model effectively. For our CNN models, we employ the Adam optimizer with a learning rate of 0.0003. The Adam optimizer combines the benefits of both the AdaGrad and RMSProp algorithms, adapting the learning rate dynamically for each parameter during training. This adaptive learning rate scheme enhances the model's convergence speed and stability. The selection of an appropriate optimizer is critical as it directly impacts the model's ability to find optimal parameter values and improve its classification performance.

However, augmenting model complexity introduces certain drawbacks, including increased computational cost and the risk of overfitting. The additional layers can lead to a significant rise in training time and memory usage. Contrary to our expectations, the inclusion of these extra layers did not result in a meaningful accuracy improvement. Instead, it prolonged the training process without yielding the desired enhancements in model performance.

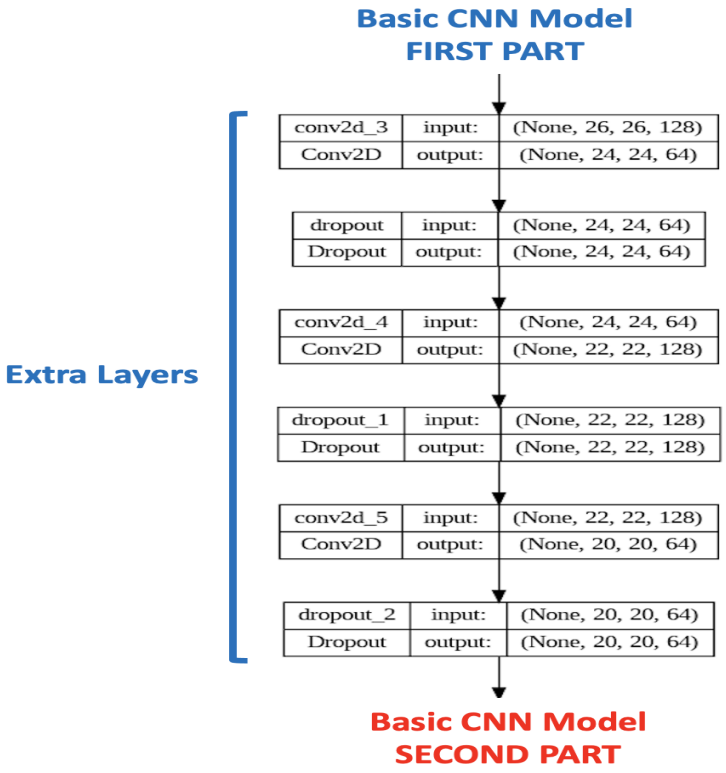

Figure 4. Architecture of the second CNN model.

In summary, experimenting with a second model allows for the exploration of whether the increased complexity and additional layers lead to improved performance on the specific image classification task at hand. The ultimate decision on which model to choose would depend on a comprehensive evaluation of their respective performances on a validation dataset, as well as consideration of computational resources and potential overfitting issues which will be addressed in the model performance part.

4.2.3. Autoencoder network model

Autoencoders are a type of neural network that is used in unsupervised learning schemes for image reconstruction tasks. The main objective of an autoencoder is to learn a compact representation of the input data, thus capturing its most salient features in a lower-dimensional encoding. This encoding is then used to reconstruct the original input data. In the context of malware detection, autoencoders can be utilized to reconstruct image representations of malware samples, allowing for the identification of anomalies or deviations from normal behavior. Other possible applications of autoencoders are data compression and denoising tasks. To employ the autoencoders for a two-class classification task, the general outline presented in Figure 5 is followed.

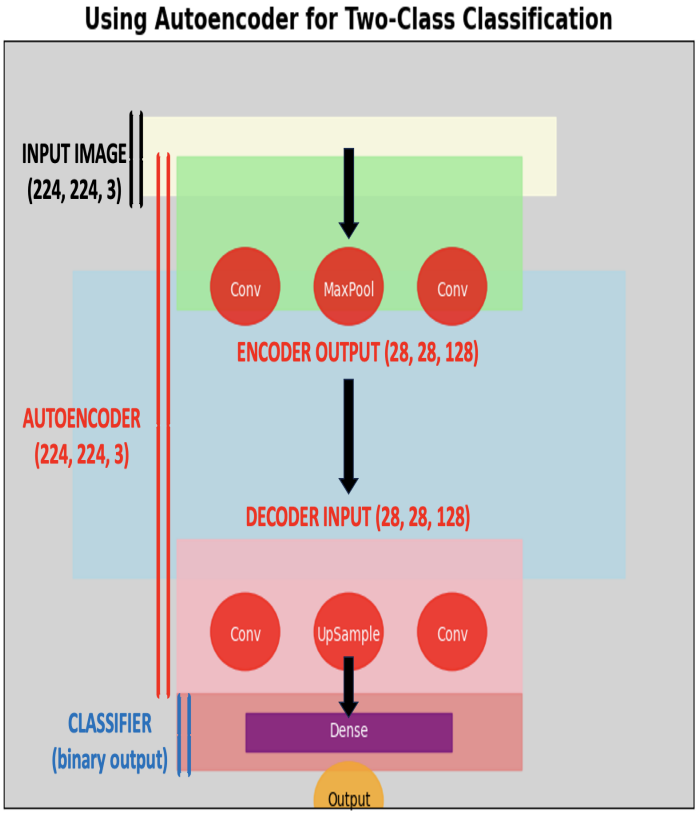

Figure 5. Architecture of the autoencoder-based model.

To effectively utilize an autoencoder model for a two-class classification task, we employed a two-step approach:

Step 1: Pretraining the autoencoder The first step involves training the autoencoder on the input data, which in this case are images. The primary objective during pretraining is to learn a compressed

representation of the input data. the autoencoder consists of two main components: The encoder and the decoder.

The encoder is responsible for encoding the input data into a lower-dimensional representation, typically referred to as the latent space. The encoder achieves this by employing a series of convolutional and pooling layers, which effectively reduce the spatial dimensions of the data while simultaneously increasing the number of channels, also known as features.

The latent space serves as a bottleneck layer that represents the compressed feature representation of the input data. It captures the most essential features of the original data, allowing the model to capture significant patterns and structures.

On the other hand, the decoder is designed to reconstruct the input data from the learned compressed representation in the latent space. It utilizes a series of convolutional and upsampling layers, effectively reversing the process performed by the encoder. The goal of the decoder is to produce a reconstruction that is as close as possible to the original input data.

Step 2: Building a classifier on top of the encoder Once the autoencoder has been trained, the next step involves using the encoder part as a feature extractor. By discarding the decoder part, the encoder output now serves as a compact and informative feature representation of the input data.

With the encoder output, a classification layer is included on top to perform the two-class classification task. This classification layer typically consists of one or more fully connected layers, often incorporating appropriate activation functions, such as a ReLU, to introduce non-linearity in the learning process.

The output layer of the classification module consists of a single neuron and employs a sigmoid activation function. This setup is ideal for binary classification tasks, as the output neuron's activation value falls within the range [0, 1], allowing the model to predict the likelihood of each sample belonging to one of the two classes.

General architecture of the implemented autoencoder model: The model based on autoencoder architecture, as depicted in Figure 5, comprises two distinct components: An autoencoder and a classifier. Initially, the autoencoder model undergoes training on the dataset. Subsequently, the output generated by the autoencoder serves as the input for the classifier. This methodology can be succinctly encapsulated as the integration of two individual models, namely the autoencoder and the classifier. The holistic architecture is summarized as follows.

As stated above, all autoencoder models are comprised of distinct components, i.e., the input layer, encoder layers, llatent space, and decoder layers. To extract the latent features via an autoencoder model, we built a symmetrical autoencoder model where the encoding and decoding networks mirror each other. Details of the first part of the implemented autoencoder model are presented in Figure 6.

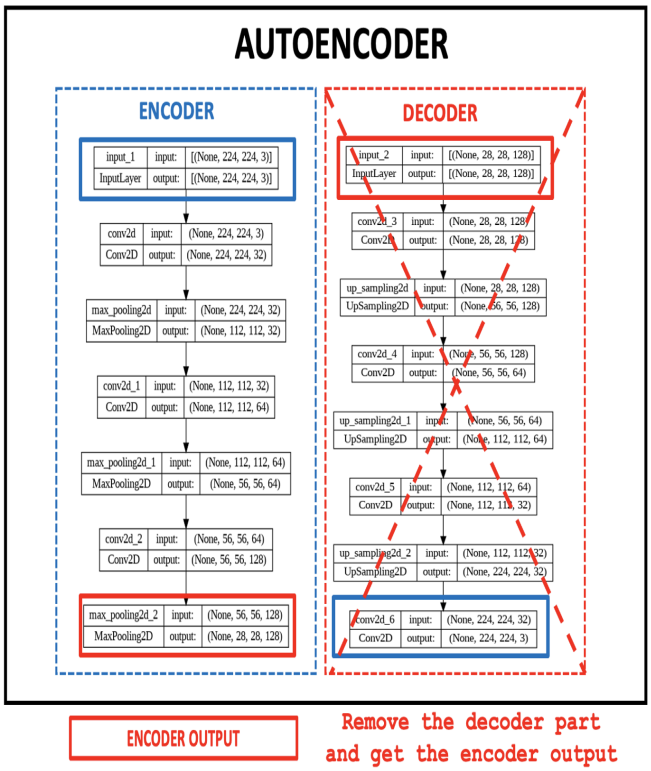

Figure 6. Architecture of the autoencoder part of the autoencoder-based model. The blue rectangles are used to represent input and output. The red rectangles are used to display the latent space.

The encoder takes as input an image of size 224 x 224 pixels with three color channels, and it employs three convolutional layers, each followed by a max-pooling layer. The activation function used throughout the encoder is the ReLU, which introduces non-linearity and helps , and it capture complex patterns. The last pooling layer's output serves as the encoded representation, which is the bottleneck of the autoencoder.

The decoder architecture mirrors the encoder's structure, using upsampling instead of max-pooling to enlarge the spatial dimensions back to the original image size. The decoder starts with an upsampling layer, followed by a convolutional layer with ReLU activation. Subsequently, two more upsampling layers are used, along with corresponding convolutional layers. The final layer of the decoder uses a convolutional filter with a sigmoid activation function to produce the reconstructed image.

The autoencoder is trained by using the mean squared error loss function, which measures the difference between the input image and its reconstructed output. The Adam optimizer is employed to update the model's weights during training. The training process is monitored by using early stopping to prevent overfitting, as well as by interrupting the training if there is no improvement in validation loss for a certain number of epochs. To achieve optimal performance, hyperparameter tuning is also performed by using a grid search approach. The hyperparameters considered for tuning are the encoding dimensions and the optimizer used during training. The encoding dimension, which were set as 28, 28, and 128 in this case determines the number of nodes in the bottleneck layer, ultimately affecting the level of compression and information retention in the encoded representation. After

training the autoencoder, the decoder segment is omitted and the encoder output is used as a latent representation of the original input image as represented in Figure 7.

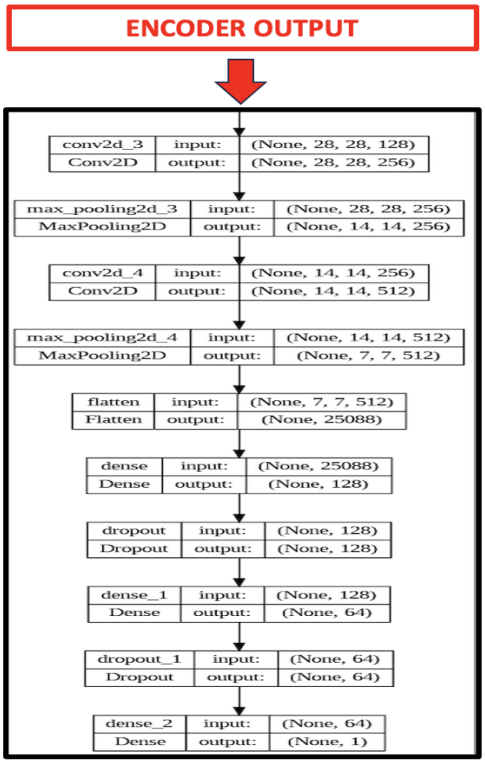

Figure 7. Architecture of the classifier part of the autoencoder-based model.

In the classifier module, encoder output is augmented by introducing additional convolutional layers, comprising 256 and 512 filters that are intended to mitigate overfitting concerns. This extended encoder output becomes the foundation for the ensuing classifier architecture, and it is aimed at enhancing classification performance. The classifier's architecture commences with a flattened representation of the augmented encoder output, followed by two densely connected layers. The initial layer, housing 128 units and employing the ReLU activation function is accompanied by a dropout layer (0.5) to enhance regularization. Subsequently, a second densely connected layer of 64 units, with ReLU activation and a dropout layer (0.5), has been introduced to enhance model robustness.

The pinnacle of the classifier's design lies in its singular-neuron output layer, which adopts the sigmoid activation function to achieve binary output. This architecture aptly suits the binary classification task's nature, as it consists of two classes. By synergizing the encoder's informative representations with the classifier's capabilities, the integrated model significantly enhances binary classification accuracy and efficiency.

The classifier is compiled by using the Adam optimizer and binary cross-entropy loss function. To facilitate training, model checkpointing and early stopping mechanisms were incorporated. The training regimen spans 500 epochs, incorporating validation on the dataset provided. This comprehensive strategy integrates the encoder's acquired feature representations with the novel classifier, leading to improved binary classification outcomes while fostering interpretability and generalization.

4.2.4. Pre-trained models

Transfer learning is a powerful technique in machine learning and deep learning, where knowledge gained from solving one problem is applied to a different but related problem. Instead of training a model from scratch, transfer learning allows us to leverage pre-trained models that have been trained on large-scale datasets for general tasks, such as image recognition [25].

In the context of our malware detection task, transfer learning becomes especially relevant due to several reasons. First, malware images, particularly those converted from PEs, can be complex and highly intricate, making it challenging to build effective models from scratch with limited data. Using pre-trained models, we can benefit from the knowledge and feature representations learned on massive image datasets, which helps to capture essential patterns and features in the malware images.

Second, pre-trained models, such as VGG16 [26], ResNet50 [27], InceptionV3 [28], and MobileNet [29], have been trained on diverse and extensive image datasets like ImageNet, allowing them to learn general features that are useful for various image-related tasks. This general knowledge can be transferred to our specific malware detection problem, reducing the need for extensive training data and accelerating the model's convergence. Moreover, the performance of transfer learning models can often surpass that of models trained from scratch, especially when dealing with limited data. The pre-trained models have already learned to recognize common features in images, making them more robust and less prone to overfitting [30].

By employing transfer learning with state-of-the-art pre-trained models, we can enhance the efficiency and accuracy of our malware detection system. The pre-trained models serve as strong feature extractors, allowing us to focus on building custom classifiers that can effectively distinguish malware from benign software.

In summary, transfer learning is a valuable tool for our malware detection task, as it allows us to capitalize on the knowledge gained from general image recognition tasks and efficiently build effective models even with limited malware image data [31].

5. Results

The proposed model was is deployed on an Intel Core i7-8700K CPU, 3.70 GHz machine and Windows 10 Pro operating system with 17.1 GB RAM and 2 GB NVIDIA GPU. We employed the proposed methodology on the Jupyter Notebook in Python by using several libraries such as Tensorflow, Keras, and Sci-kit learn. The performance of the models was evaluated in terms of accuracy, precision, and sensitivity (recall).

5.1. Results on the Trapmine test dataset

In the initial phase of the experimental procedures, the models underwent training and testing exclusively on the Trapmine dataset. This dataset comprises a total of 17,217 images, with 10,217 belonging to the benign class and the remaining allocated to the malicious class. To establish a well-balanced training set, a random selection process was employed, resulting in the acquisition of 11,172 images. Among these, 5,586 images were assigned to the benign class, and an equal number were assigned to the malicious class. Subsequently, 2,800 images were chosen from the remaining pool for validation, and they were evenly distributed with 1,400 images in each class.

Following the training phase, wherein malicious images were depleted, additional instances were drawn from the Malevis and Malimg datasets to supplement the deficiency. Subsequently, for the creation of an independent test set, 3,231 residual benign images from the Trapmine dataset, which were not employed during the training phase, were applied with 3,231 randomly selected images from the blended dataset. The blended dataset is a composite amalgamation of the Malevis and Malimg datasets. The resulting accuracy and other pertinent metrics for the models evaluated on the test dataset are detailed in Table 1.

Table 1. Performance metrics of different models on Trapmine dataset.

	Accuracy (%)	**Specificity (%)**	**Sensitivity (%)**
CNN model 1	79.94	78.07	80.46
CNN model 2	80.05	77.45	81.34
Autoencoder	90.56	85.87	**90.77**
VGG16	**91.25**	**88.67**	90.68
ResNet50	91.11	87.50	90.61
InceptionV3	90.41	85.79	90.68
MobileNet	86.94	85.45	89.11

These observations underscore the distinctive strengths inherent to each model, elucidating their unique roles in evaluating efficacy for malware detection. For instance, the fundamental CNN model excelled in the recognition of general malware patterns, presenting a robust defense mechanism, particularly against commonplace threats, and it achieved an 80% accuracy rate. Additionally, pre-trained models incorporating a diverse array of CNN layers demonstrated commendable performance. This discovery accentuates the inherent robustness of the CNN infrastructure, which is predominantly employed for image classification tasks.

The success of the CNN infrastructure, as evidenced by both of the CNN models and the pre-trained models, is noteworthy. Beyond the achievements of the CNN infrastructure, the autoencoder model has emerged as a pivotal asset, contributing significantly to feature extraction and the learning process. This contribution resulted in an accuracy rate of 90%. These nuanced findings collectively underscore the multifaceted strengths of each model, thereby enhancing our understanding of their respective contributions to malware detection.

Among the models evaluated on the Trapmine dataset, VGG16 demonstrated the highest accuracy (91.25%) and specificity (88.67%), showcasing its overall effectiveness in classifying both benign and malicious software. However, the autoencoder model exhibited the best sensitivity (90.77%), excelling in accurately detecting malicious instances. This indicates that while VGG16 is strong in overall performance, autoencoder is particularly adept at identifying true positive cases among malware samples. Hence, the choice between these models may depend on the specific emphasis on overall accuracy versus sensitivity in a malware detection system.

5.2. Results on the MaleVis and Malimg test dataset

The results presented in Table 1 show how the trained models performed on a completely unknown test dataset. But, we also want to present how these models will perform on public datasets, particularly, the Malimg and MaleVis datasets. These two datasets consist of malicious files from various variants.

The MaleVis dataset consists of 14.226 RGB images in total from 25 different classes: 9100 samples for training and 5126 samples for validation. Comprehensive results obtained for a combination of datasets including Trapmine, MaleVis, and Malimg are outlined in Table 2.

Table 2. Performance metrics of different models on MaleVis and Malimg test datasets.

	Accuracy (%)	**Specificity (%)**	**Sensitivity (%)**
CNN model 1	88,15	87,61	88,70
CNN model 2	87,76	90,25	85,27
Autoencoder	89,10	87,49	**90,71**
VGG16	**90,65**	96,24	85,06
ResNet50	81,84	89,21	74,48
InceptionV3	90,36	**97,74**	83,49
MobileNet	90,65	97,05	84,26

Regarding the training data, all classes had equal numbers of samples but the class distributions in the validation data were not uniform. On the other hand, the Malimg dataset is highly imbalanced in terms of class distribution, and it only consists of malicious samples from 25 different families. Both of these two datasets were used to test the multiclass classification success of the models.

In our scenario, since we wanted to develop a two-class classification model, we needed to create an unseen malicious test set. In the training phase, we used 70% percent of the benign images, and there were no malicious images. But, we had plenty of malicious images from the MaleVis and Malimg data. So, to construct a second test, and, particularly, to try to measure the true positive results of the models, we randomly selected 10,000 malicious images from both MaleVis and Malimg. And, in addition to the 3,231 unseen benign images, we added 6,769 benign images which were used in the training phase. As a result, for the second test results, we employed 20,000 images: 10,000 unseen malicious and 10,000 benign images 30% of which corresponded to unseen data.

To provide a clearer understanding of the complexity and computational burden of the models, we offer relevant information regarding the total parameter numbers and the corresponding training times for each model in Table 3.

Table 3. Complexity of the models and time consumed during training.

	Total Parameters	**Trainable Parameters**	**Average Time (seconds/epoch)**
CNN model 1	5,631,169	5,631,169	390
CNN model 2	1,953,217	1,953,217	430
Autoencoder (autoencoder)	333,955	333,955	1140
Autoencoder (classifier)	12,853,505	12,853,505	735
VGG16	14,780,481	65,793	1250
ResNet50	23,850,113	262,401	735
InceptionV3	22,065,185	26,2401	380
MobileNet	3,360,193	131,329	265

To emphasize the main contribution of this work, we present a comparison in Table 4, which

summarizes the performance of previously studied image-based models on the MaleVis and Malimg datasets, alongside our models.

Table 4. Comparison of the performance of previously studied models and our models.

	Accuracy (%)	**Specificity (%)**	**Dataset**
Vinayakumar et al. [32]	86.29	-	MaleVis
Luo and Lo [33]	92.24	-	MaleVis
Cui et al. [34]	92.13	-	MaleVis
Gilbert [35]	90.59	-	MaleVis
Singh et al. [36]	93.00	-	MaleVis
Vinayakumar et al. [32]	96.30	-	Malimg
Luo and Lo [33]	93.72	-	Malimg
Cui et al. [34]	94.50	-	Malimg
Gilbert [35]	95.33	-	Malimg
Singh et al. [36]	96.08	-	Malimg
CNN model 1	88.15	87.61	Trapmine, MaleVis, and Malimg
CNN model 2	87.76	90.25	Trapmine, MaleVis, and Malimg
Autoencoder	89.10	87.49	Trapmine, MaleVis, and Malimg
VGG16	90.65	96.24	Trapmine, MaleVis, and Malimg
ResNet50	81.84	89.21	Trapmine, MaleVis, and Malimg
InceptionV3	90.36	97.74	Trapmine, MaleVis, and Malimg
MobileNet	90.65	97.05	Trapmine, MaleVis, and Malimg

It is important to note that this is not a direct comparison since these models were designed to be multi-class malware detectors, whereas our models were designed to be two-class detectors. Furthermore, these models were trained and tested on their respective datasets, while our model was trained on a distinct dataset called Trapmine and tested by applying Trapmine for the normal class and a subset of the MaleVis and Malimg datasets for the abnormal class. However, by comparing the accuracies and true positive scores of the models, we were able to gain insights into their performance as a tool to detect abnormal samples.

VGG16 demonstrated the highest accuracy (90.65%) and specificity (96.24%), while the autoencoder model achieved the best sensitivity (90.71%). This observation aligns with their results from in the previous test conducted exclusively on the Trapmine dataset, further establishing VGG16 and the autoencoder model as the top-performing models.

5.3. Results for convergence metrics

Regarding the convergence metrics for our applications, we present details in Table 5 which provide insights into the training process. This table includes information on the number of iterations, convergence criteria, and training stability during training.

Table 5. Convergence metrics for the models during the training.

	Number of Iterations (Epoch Numbers)	**Loss Function** (Function & Convergence Point)	**Training Stability** (Early Stopping)
CNN model 1	60	binary cross-entropy & 0.13	50
CNN model 2	68	binary cross-entropy & 0.19	50
Autoencoder	135	binary cross-entropy & 0.19	50
VGG16	225	binary cross-entropy & 0.27	100
ResNet50	345	binary cross-entropy & 0.45	100
InceptionV3	174	binary cross-entropy & 0.29	100
MobileNet	112	binary cross-entropy & 0.19	100

By incorporating these metrics into Table 5, we aim to offer a more nuanced understanding of how our models converged during training, as well as how effectively they reached their optimal performance levels. Additionally, we implemented early stopping across all models to ensure training stability and prevent overfitting. This means that if the validation loss does not show improvement over consecutive epochs, training will stop early to mitigate the risk of overfitting. This proactive approach helped our models to generalize well to unseen data and maintain optimal performance levels.

6. Discussion

Future research directions may involve further optimizing model architectures, exploring additional datasets, and investigating real-time deployment possibilities to enhance the practicality and efficiency of malware detection systems. To address complex malware in future works, advanced techniques such as ensemble learning, where multiple models work collaboratively to improve detection accuracy and resilience against evasion strategies can be used. Since ensemble methods combine the strengths of different models, thereby mitigating individual weaknesses and enhancing overall performance, combining multiple models can be an efficient methodology. However, it is important to note that this approach comes with a significant computational burden, representing the primary drawback of the ensemble methodology. Moreover, continuous monitoring and updating of malware detection models by using real-time threat intelligence feeds and dynamic learning mechanisms can enhance their ability to adapt to evolving malware landscapes and detect new variants effectively.

7. Conclusions

In this study, we deployed a comprehensive approach to malware detection, utilizing a variety of models such as CNN and autoencoder models, VGG16, ResNet50, InceptionV3, and MobileNet. The models were extensively evaluated on multiple datasets, including the Trapmine, MaleVis, and Malimg datasets.

The results obtained for the Trapmine test dataset revealed the distinctive strengths of each model. Notably, the CNN models exhibited robust capability to recognize general malware patterns, achieving an accuracy rate of around 80%. Furthermore, the autoencoder model emerged as a pivotal asset, contributing significantly to feature extraction and learning processes and achieving an impressive accuracy rate of 90%.

Expanding our evaluation to the MaleVis and Malimg test datasets provided insights into the models' performance on a completely unknown dataset. CNN models, the autoencoder model, and other pre-trained models demonstrated competitive accuracy rates, showcasing their adaptability to diverse malware variants.

Considering the computational aspects, we have presented the complexity of each model in terms of the total number of parameters, trainable parameters, total number of and average training time per epoch. These metrics provide valuable insights into the computational burden associated with each model.

In conclusion, our study highlights the multifaceted strengths of each model and their unique contributions to the field of malware detection. The robustness of the CNN infrastructure, coupled with the significant role played by the autoencoder model, collectively enhances our understanding of effective methodologies in the realm of malware detection.

In this paper, the highlight of CNNs lies in its robust ability to recognize general malware patterns. Even a simple and modest model constructed using several CNN layers exhibited strong detection performance. Furthermore, we selected pre-trained models with a CNN infrastructure with the expectation that the CNN architecture would be robust when applied for image classification tasks. The results are very promising. To compare the strengths of the models with previously studied models, we constructed a test set that combined both our training set Trapmine and public datasets MaleVis and Malimg. Since the abnormal samples were taken from public datasets, the specificity score, or the strength of distinguishing abnormal samples, was around 97% for the models. This result reflects the ability of CNNs to effectively distinguish between benign and malicious software based on visual signatures, contributing significantly to the overall effectiveness of the malware detection system discussed in the paper.

Use of AI tools declaration

The authors declare they have not used Artificial Intelligence (AI) tools in the creation of this article.

Acknowledgments

We would like to thank Mr. Celil UNUVER, the CEO and founder of TRAPMINE XDR (the company has recently merged with SONICWALL).

Conflict of interest

The authors declare no conflict of interest.

References

1. K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, H. Liu, A review of android malware detection approaches based on machine learning, *IEEE Access,* **8** (2020). https://doi.org/10.1109/ACCESS.2020.3006143

2. B. Amos, H. Turner, J. White, *Applying machine learning classifiers to dynamic Android malware detection at scale*, In: 2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC), IEEE, Italy, 2013, 1666–1671. https://doi.org/10.1109/IWCMC.2013.6583806
3. M. Egele, T. Scholte, E. Kirda, C. Kruegel, A survey on automated dynamic malware-analysis techniques and tools, *ACM Comput. Surv.*, **44** (2012), 1–42.
4. B. Amro, Malware detection techniques for mobile devices, *Int. J. Mobile Netw. Commun. Telemat.*, **7** (2017). https://doi.org/10.1145/2089125.2089126
5. K. Kavitha, P. Salini, V. Ilamathy, *Exploring the malicious Android applications and reducing risk using static analysis*, In: 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), IEEE, India, 2016, 1316–1319. https://doi.org/10.1109/ICEEOT.2016.7754896
6. E. M. B. Karbab, M. Debbabi, MalDy: Portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports, *Digit. Invest.,* **28** (2019), 77–87. https://doi.org/10.1016/j.diin.2019.01.017
7. R. Ito, M. Mimura, *Detecting unknown malware from ASCII strings with natural language processing techniques*, In: 2019 14th Asia Joint Conference on Information Security (AsiaJCIS), IEEE, Japan, 2019. https://doi.org/10.1109/AsiaJCIS.2019.00-12
8. P. Najafi, D. Koehler, F. Cheng, C. Meinel, *NLP-based entity behavior analytics for malware detection*, In: 2021 IEEE International Performance, Computing, and Communications Conference (IPCCC), IEEE, USA, 2021. https://doi.org/10.1109/IPCCC51483.2021.9679411
9. U. Raghav, E. Martinez-Marroquin, W. Ma, *Static analysis for Android Malware detection with document vectors*, In: 2021 International Conference on Data Mining Workshops (ICDMW), IEEE, New Zealand, 2021. https://doi.org/10.1109/ICDMW53433.2021.00104
10. X. Xing, X. Jin, H. Elahi, H. Jiang, G. Wang, A malware detection approach using autoencoder in deep learning, *IEEE Access*, **10** (2022), 25696–25706. https://doi.org/10.1109/ACCESS.2022.3155695
11. Q. Le, O. Boydell, B. Mac, M. Scanlon, Deep learning at the shallow end: Malware classification for non-domain experts, *Digit. Invest.*, **26** (2018), S118–S126. http://dx.doi.org/10.1016/j.diin.2018.04.024
12. J. Y. Kim, S. J. Bu, S. B. Cho, Zeroday malware detection using transferred generative adversarial networks based on deep autoencoders, *Inform. Sci.*, **460–461** (2018), 83–102. https://doi.org/10.1016/j.ins.2018.04.092
13. I. Goodfellow, NIPS 2016 Tutorial: Generative adversarial networks, *arXiv preprint*, 2014. https://doi.org/10.48550/arXiv.1701.00160
14. S. Kumar, B. Janet, DTMIC: Deep transfer learning for malware image classification, *J. Inf. Secur. Appl.*, **64** (2022). https://doi.org/10.1016/j.jisa.2021.103063
15. Ö. Aslan, A. A. Yilmaz, A new malware classification framework based on deep learning algorithms, *IEEE Access*, **9** (2021), 87936–87951. https://doi.org/10.1109/ACCESS.2021.3089586

16. F. Rustam, I. Ashraf, A. D. Jurcut, A. K. Bashir, Y. B. Zikria, Malware detection using image representation of malware data and transfer learning, *J. Parallel Distr. Com.*, **172** (2023), 32–50. https://doi.org/10.1016/j.jpdc.2022.10.001
17. T. Li, Y. Luo, X. Wan, Q. Li, Q. Liu, R. Wang, et al., A malware detection model based on imbalanced heterogeneous graph embeddings, *Expert Syst. Appl.*, **246** (2014), 123109.
18. *Google play store*. Available from: `https://https://play.google.com/store/apps.`
19. *Virusshare*. Available from: `http://virusshare.com/.`
20. *Virustotal*. Available from: `https://www.virustotal.com/gui/home/upload.`
21. L. Nataraj, S. Karthikeyan, G. Jacob, B. S. Manjunath, *Malware images: Visualization and automatic classification*, In: Proceedings of the 8th International Symposium on Visualization for Cyber Security, 2011, 1–7. https://doi.org/10.1145/2016904.2016908
22. A. S. Bozkir, A. O. Cankaya, M. Aydos, *Utilization and comparison of convolutional neural networks in malware recognition*, In: 2019 27th Signal Processing and Communications Applications Conference (SIU), IEEE, Turkey, 2019, 1–4. https://doi.org/10.1109/SIU.2019.8806511
23. *MaleVis*. Available from: `https://web.cs.hacettepe.edu.tr/~selman/malevis/.`
24. S. Venkatraman, M. Alazab, R. Vinayakumar, A hybrid deep learning image-based analysis for effective malware detection, *J. Inf. Secur. Appl.*, **47** (2019), 377–389.
25. A. Krizhevsky, I. Sutskever, G. E. Hinton, ImageNet classification with deep convolutional neural networks, *Adv. Neural Inform. Proc. Syst.*, 2012.
26. K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, *arXiv preprint*, 2014. https://doi.org/10.48550/arXiv.1409.1556
27. K. He, X. Zhang, S. Ren, J. Sun, *Deep residual learning for image recognition*, In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, 770–778.
28. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna, *Rethinking the inception architecture for computer vision*, In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, 2818–2826.
29. G. Huang, Z. Liu, L. Van Der Maaten, K. Q. Weinberger, *Densely connected convolutional networks*, In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, 4700–4708.
30. J. Yosinski, J. Clune, Y. Bengio, H. Lipson, *How transferable are features in deep neural networks?* In: Advances in Neural Information Processing Systems (NIPS), 2014.
31. S. J. Pan, Q. Yang, A survey on transfer learning, *IEEE T. Knowl. Data Eng.*, **22** (2010), 1345–1359. https://doi.org/10.1109/TKDE.2009.191
32. R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, S. Venkatraman, Robust intelligent malware detection using deep learning, *IEEE Access*, **7** (2019), 46717–46738. https://doi.org/10.1109/ACCESS.2019.2906934
33. J. S. Luo, D. C. T. Lo, *Binary malware image classification using machine learning with local binary pattern*, In: 2017 IEEE International Conference on Big Data (Big Data), IEEE, USA, 2017, 4664–4667. https://doi.org/10.1109/BigData.2017.8258512

34. Z. Cui, F. Xue, X. Cai, Y. Cao, G. G. Wang, J. Chen, Detection of malicious code variants based on deep learning, *IEEE T. Ind. Inform.*, **14** (2018), 3187–3196. https://doi.org/10.1109/tii.2018.2822680

35. D. Gibert, *Convolutional neural networks for malware classification*, M.S. thesis, Univ. Rovira Virgili, Tarragona, Spain, 2016.

36. A. Singh, A. Handa, N. Kumar, S. K. Shukla, *Malware classification using image representation*, In: Proc. Int. Symp. Cyber Secur. Cryptogr. Mach. Learn. Cham, Switzerland: Springer, 2019, 75–92. https://doi.org/10.1007/978-3-030-20951-3_6

AIMS Press