https://doi.org/10.5281/zenodo.8246395

International Journal of Computer Science and Information Security (IJCSIS),
Vol. 21, No. 7, July 2023

XLCNN: A TRANSFORMER MODEL FOR MALWARE DETECTION

Konstantinos Giapantzis
Centre of Research and Technology Hellas (CERTH)
Department of Applied Informatics, University of Macedonia
Thessaloniki, Greece

Spyros T. Halkidis
Computational Methodologies and Operations Research Laboratory, Department of Applied Informatics, University of Macedonia
Thessaloniki, Greece

***Abstract*—The present research describes a Transformer-based neural network model that was developed in order to detect malicious software. We believe that the scientific community should take advantage of the contribution of Transformer models in the field of cybersecurity and go beyond the limits set by the classic natural language processing. For this purpose a new and more sophisticated algorithm was created based on the methodology used by the XLNet neural network which was proposed by the Google AI Brain Team. The proposed XLCNN model detects malicious code with a higher success rate than its predecessor. The method of detecting malware is based on the extraction and analysis of metadata contained in Windows executable files. From our carried out experiments, it was found that the size and architecture of the feed-forward neural network in combination with our proposed tokenizer, is one of the most important factors of XLCNN for classification problems. To justify the concept of XLCNN as an effective approach to detecting malware, the effectiveness and efficiency of the algorithm was measured for a finite number of epochs and compared to other Transformer models such as XLNet, BERT and Transformer-XL using exactly the same inputs. Using our proposed network has proven to be not only a reliable way for security researchers to detect malware, but also an effective and highly accurate method that offers high accuracy rate of 98.88%.**

***Index Terms*—malware detection, metadata, neural network, Transformers, XLCNN**

I. INTRODUCTION

Malware is a program developed with the intent of breaking into a system to monitor, intercept personal information, encrypt data or require ransom [1]. Due to the critical threat posed by malware in cyberspace, many different methods and analysis tools have been developed to detect it.

One of these methods computes the file of the under condition signature and compares it to the ones stored in a database containing signatures of known malware. The main disadvantage of this method, which most antivirus tools use, is that, if the code is modified through processing called obfuscation, then it is impossible to detect the existence of malware. One more disadvantage of this method is that, if a new malware with unknown signature is used, then it is practically impossible to detect it.

Another detection technique is based on malware behavior. The detection process is particularly tedious as it requires isolating the file and placing it in a secure environment, such as sandbox, and then supervising its behaviour by qualified personnel. Observing behavior requires a lot of dedication and time while at the same time it may not be effective as concealment techniques are improved and malware evolves to avoid being detected.

Therefore, more complex, mathematically intelligent and automated methods are required such as machine learning. Machine learning, especially Deep Learning, is an excellent technique that deals with data variants because not only can it learn the given feature during the training process, but also automatically extracts features from data to achieve the goal of classification [2]. When an infected file is given as input to a Deep Learning algorithm, according to the characteristics it has learned from the training process, it can identify whether it is either malicious or benign.

We consider the problem of detecting malware as a classification problem that has two categories. One category is benign software and the other is malicious. The extraction of the metadata from the executable files, made possible the analysis by natural language processing (NLP) models, as they can be considered as sequence of words. Metadata are a set of ASCII characters many of which are impossible to decode by the human agent even with the help of specialized programs, let alone capable of extracting those features that contribute to its grouping.

After research, it was found that there is a set of algorithms that is more accurate and less time consuming than all the above methods. Initially, algorithms using attention mechanisms [3] were introduced to solve machine translation problems. Transformer models gradually replaced RNNs [4] in mainstream NLP.

The Transformer model architecture takes a new approach to machine learning as it completely eliminates repetition. Transformers create attributes of each word using an attention mechanism to understand how important all the other words in the sentence are. Knowing these the renewed features of the word are simply the sum of the number of linear transformations of all the words' features weighted by the average number of linear transformations [3].

In the present research a set of Transformer models was implemented for the classification problem and then a completely new Transformer model named XLCNN was proposed and developed. After comparing these algorithms, it turned out that our proposed model is more accurate in detecting new malware.

XLCNN uses a complex architecture in the feed forward neural network and this is the main feature that distinguishes it from other Transformer models. A new tokenizer was also proposed exclusively to serve the purposes of the XLCNN model we propose. This tokenizer was then applied to the rest of the models in order to assess its effectiveness. The contribution to the scientific community, both, at the theoretical level as we propose a new model architecture and at the level of implementation, is the provision of a pre-trained Transformer model which will identify new malware using their metadata.

Chapter II presents the structure followed by the XLCNN and specifically the way in which the sequence transformation is done, the Optimizer used, the Linear Schedule, how the Dropout was used to prevent overfitting, the activation function and the tokenizer we created. Chapter III presents the XLCNN model and specifically the feed forward structure that follows. Chapter IV provides information on how the dataset was created, but also how the metadata was extracted. Chapter V presents and analyzes the experimental results of the proposed XLCNN model and compares it with the rest of Transformer models. Finally, Chapter VI describes the final conclusions of this research paper.

Related Work

MALBERT [5] has used BERT [6] for Malware Detection in Android Systems. They achieved an accuracy of 97.61%. So we will elaborate on Malware Detection approaches based on Deep Learning and present only representative examples.

Hardy et al. [7] in DL4MD, based on the Windows Application Programming Interface (API) calls extracted from the Portable Executable (PE) files study how the Stacked Autoencoders (SAE) model can be designed for intelligent malware detection. The SAEs model employs greedy layerwise training operation for unsupervised learning, followed by supervised parameter fine tuning. They achieve an accuracy of 95.64% in the Testing phase.

In Rhode et al. [8] recurrent neural networks (RNNs) are used to predict whether or not an executable is malicious. They achieve 94% accuracy using 5 seconds of execution for each executable file.

Pascanu et al. [9] also use RNNs for Malware Classification. Echo State Networks (ESNs) and Recurrent Neural Networks are used for the projection stage that extracts the figures. Echo State Networks have been successfully used for predicting chaotic systems. They achieve a true positive rate of 98.3% and a false positive rate of 0.1%.

Kolosnjaji et al. [10] use Deep Learning for Classification of System Call Sequences based on data extracted from VirusTotal. They achieve an average of 85.6% on precision.

In [11] Malware Detection in Android is performed. Efficient malware detection is performed using both textual and visual features. The trained and texture features were combined and balanced using the Synthetic Minority Oversampling (SMOSE) method. Then a CNN network is used to mine the deep features. Finally, an ensemble of Gaussian Naive Bayes, Support Vector Machines, Decision Trees and Random Forests is used. The precision achieved is 98%, while the recall is 99%. The F1-score ranges from 81% to 99%. The accuracy, precision and recall range from 72% to 99% on the test data.

In [12] the SHERLOCK model, which is a self-supervision based deep-learning model to detect malware based on the Vision Transformer (ViT) architecture, is employed to identify malware in Android. A standared transformer encoder is employed. The precision achieved is 86.7% while the associated recall is 89%.

In [13] the code is disassembled and an interpretable feed-forward neural network is used. The “pefile” library to extract numerical features from PE headers is used and the IDA Pro disassembler is employed. The accuracy is up to 97.7% as well as the F1-score. The precision achieved is 97.5% and the recall 97.9%.

In [14] malware detection using the transformers based model GPT-2 is used. The custom pretrained model achieves 85.4% accuracy, while the GPT-2 pretrained model achieves 78.3% accuracy.

In [15] the BIG2015 and the SubBODMAS malware datasets where used. Malware images where used for visualization and automatic classification. CNNs for malware classification where used in grayscale, markov transition field encoded images and bigram frequency images where used (using the bigram occurrence count histogram). The accuracy achieved ranges from 91.30% to 98.61% based on the image used for the BIG2015 dataset, while the weighted-F1 score ranges from 91.38% to 98.50%. On the SubBODMAS dataset the accuracy achieved ranges from 89.22% to 95.98% while the weighted-F1 score ranges from 88.27% to 95.98%.

II. Background Architecture

A. Sequence-to-Sequence

Typically, a neural network for sequence modeling of classification problems consists of the following layers:

Embedding layer. In this layer the model receives a sequence of words and tries to predict the probability of the next word appearing [16]. Specifically the input array, which is a loosely-encoded (e.g., hot-encoded) input token, is mapped to a denser feature layer. This is necessary because a high-dimensional feature vector is more capable of encoding information about a particular token (the term for the text corpus) than a simple hot-encoded vector. Instead of a pre-trained vector of words being used in this work, a state of the art tokenizer was developed specially made for the classification of malicious applications using their metadata.

Encoder Layer. After mapping the input multidimensional token, the sequence passes through the encoder layer to compress all the information from the input embedding layer (the whole sequence) into a specific vector of a fixed length. The encoder is made up of a stack of n = 24 equal layers. Each level has two sublayers. The first is a multi-head self-focus mechanism and the second is a simple feedforward network that is fully connected according to position. A residual attachment is used around each of the two sublayers followed by normalization [17]. That is, the output of each sublayer is called layer normalization (x + sublayer(x)), whereas sublayer(x) is a function executed by the sublayer itself. To facilitate these residual connections all sublayers of the model, as well as the embedding layers, produce an output of $\text{Dimension}_{\text{Model}} = 512$.

Decoder layer. The decoder layer takes this encoded feature vector and creates the output token sequence. The decoder also has a set of equal levels n = 24. In addition to the two sublayers in each encoder layer, the decoder inserts a third sublayer that pays multiple attentions to the output of the encoder stack [18]. Like the encoder, it uses residual connections around each sublevel followed by level normalization. The self-focus sublayer in the decoder stack is also modified so that the position does not go into the subsequent position [19]. This masking, combined with fact that the outputed embeddings are offset by one position, ensures that the predictions for the next position can only depend on the known outputs in the previous positions.

Attention mechanism. The disadvantage of the encoder-decoder structure is that the performance of the model degrades as the length of the original sequence increases due to the limitation on how much information an encoded feature vector of a fixed length can contain. To address this problem, Bahdanau, D. et al. [20] proposed an attention mechanism. In the attention mechanism the decoder tries to find the point in the encoder sequence where the most important information can be found and uses that information and previously decoded words to predict the next token in the sequence. A function of attention is to map a query and arrange a set of key value pairs in the output where query, keys, values and output are all vectors. The output is calculated by the weighted portion of the values, where the weight assigned to each key is correlated with the query with the function matched with that key [21].

B. AdamW Optimizer

The main contribution of AdamW [22] is to improve regularization in Adam [23] by separating weight reduction from gradient-based updating. The weight reduction of AdamW optimizer makes the optimal settings for the learning rate and weight reduction factor much more independent, which simplifies hyperparameter optimization.

In XLCNN learning rate was set to $3 \cdot 10^{-5}$, AdamW's epsilon for numerical stability was set to $1 \cdot 10^{-8}$, decoupled weight decay was set to 0.01, while the bias should be corrected in each epoch during training.

C. Linear Schedule with Warmup

Linear schedule with warmup creates a schedule with a learning rate that decreases linearly from the initial learning rate set in the optimizer to 0. In the XLCNN model the AdamW optimization algorithm, which has been configured with a learning rate α and a warm-up factor ω has been implemented. The ω symbol is a sequence of “warm-up factors” where $\omega_t \in [0, 1]$, which are used to reduce the step size of each iteration t. In particular, XLCNN implements a warm-up program by replacing α with $\alpha_t = \alpha \cdot \omega_t$ in the algorithm’s update rule. A linear warm-up configured by a “warm-up period” T was applied by XLCNN model :

$$\omega_t(linear, T) = min\left(1, \frac{1}{T} \cdot t\right) \quad (1)$$

In order to calculate the value for the number of warmup steps, the following formula was used in XLCNN:

$$st = 0.25 \cdot \sum_{i=0}^{l_t} (it) \quad (2)$$

where s_n is the number of warmup steps, l_t is the total size of the training dataset and $t \in Z$ is the iteration on each step.

D. Dropout

Dropout is a specific neural reduction strategy implemented in the XLCNN’s feed forward neural network in order to significantly optimize its architecture while maintaining competitive performance. The result is the creation of a much smaller and faster model due to the fact that some neurons have been zeroed. It also shows that the model was trained from the beginning without the use of a pre-trained model.

The application of Dropout in XLCNN has as a result the prevention of excessive overfitting and provides a way to effectively integrate multiple architectural neural networks. The term "dropout" refers to those who enter the neural network (hidden and visible). In the simplest case, each unit is stored independently of the other units with a fixed probability p [24]. In XLCNN the probability p was set to 0.1, which seems according to the results described in Chapter V, to be close to optimal for the feed forward neural networks.

E. Activation function

In XLCNN model it was applied the Gaussian Error Linear Unit (GELU), a neural network activation function with high performance. $x\Phi(x)$, is the GELU activation function where $\Phi(x)$ is the standard Gaussian cumulative distribution function [25]. XLCNN model used an “adaptive dropout network” which was trained jointly with the GELU activation function by approximately computing local expectations of binary dropout variables, computing derivatives using back-propagation and using stochastic gradient descent.

GELU activation function is a combination from ReLUs [26], zoneout [27] and dropout. The ReLU deterministically multiplies the input by zero or one and the dropout stochastically multiplies the input by zero. In addition, a new regularizer called zoneout multiplies the inputs by one. This functionality was merged by multiplying the input by zero or one, but the values of this zero-one mask are stochastically determined depending on the input [25].

F. Tokenizer

In order to serve the purpose of classifying malware by XLCNN, we propose a custom tokenizer. The purpose of the tokenizer is to separate words into sub-words and encode them into integers and vice versa. There are various tokenization algorithms that can be used, such as Byte Pair Encoding (BPE), Unigram, character and word. Our selection focuses on Byte Pair Encoding (BPE) [28].

The Python library SentencePiece [29] was used to train it. SentencePiece is an unsupervised tokenizer system used to encode and decode words primarily for text processing systems. It is an ideal choice, as it is used for neural networks with a predefined dictionary size and provides an end-to-end system that does not depend on language-specific pre/postprocessing.

The procedure by which the XLCNN tokenizer was trained included archiving all metadata from the training set executables excluding those in the testing set. This created a 1GB file that contained only executable metadata regardless of whether it came from benign or malicious files. The advantage of this method is that it provides a tokenizer that is more compact in terms of text and vocabulary size, while providing stronger guarantees that every subword unit has appeared in the metadata training text. It caried out from our experiments that it offers faster execution time to the Transformer model compared to the default tokenizers that are commonly used. The training procedure of our proposed XLCNN tokenizer took 5 minutes while the size of the dictionary was set to 4000.

III. The Proposed XLCNN Model

The proposed XLCNN is a Transformer model having an encoder-decoder structure. The main purpose of the encoding layer is to achieve a mapping between a sequence of input symbol representations $(x_1, ..., x_n)$ and a sequence of continuous representations $z = (z_1, ..., z_n)$. The decoder having the value of the representation z, outputs a sequence $(y_1, ..., y_m)$ of symbols one element at a time [19]. At each step, the model is spontaneously accepting the input of the symbols created in the previous step to create the next symbols. XLCNN's model architecture is a multilayer bidirectional Transformer model based on the original implementation described in Yang Z. et al. [30]. XLCNN uses a composite self-attention layer and two fully connected layers inside the feed forward network, as shown on the Fig. 1.

XLCNN uses convolutional neural networks (CNNs) [31] in its core architecture. Specifically, it uses 24 layers of feed forward neural networks, with each layer consisting of 2 CNNs having as input a matrix of dimensions [512 x 1 x 512] and producing an output matrix of [512 x 1 x 2048] dimensions, 2 Linear Transformation layers and 1 Normalization layer. Our experimental results have shown that this neuron architecture is the most efficient in malware classification, as it increases network performance, reduces loss [32] and increases ultimate prediction accuracy.

Our feed forward architecture with CNNs can learn much more complex invariants (e.g to recognize malicious code only from the metadata it has), while consumes small amount of computing power. This results in faster training and easier learning compared to the other Transformer models. Fig. 1. shows the architecture of feed forward convolutional neural network.

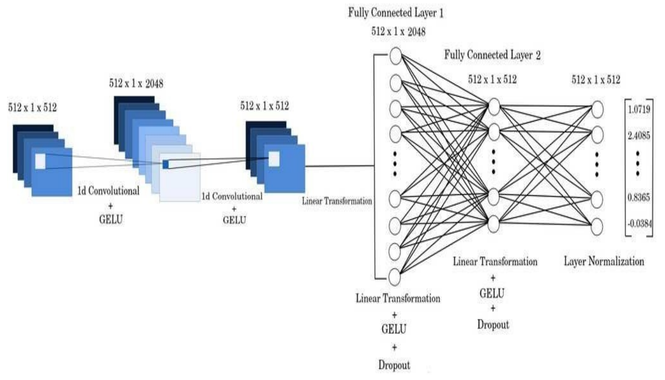

Fig. 1. Feed forward convolutional neural network architecture with input matrix [512 x 1 x 512] and output matrix [512 x 1 x 2048].

The vocabulary size of the XLCNN model, which defines the number of different tokens that can be represented by the different input words, is set to $4 \cdot 10^3$. The linear transformation is the same in different positions but uses different parameters for each layer. The additional convolutions neural networks have kernel size of 1. The inner-layer has dimensionality $n_x = 512$ and the dimensionality of output is $n_{fS} = 2048$. We proposed 24 hidden layers in the transformer encoder and 16 attention heads for each attention layer. The Fig. 2 shows the graphical representation of the fully connected feed forward neural network proposed for XLCNN.

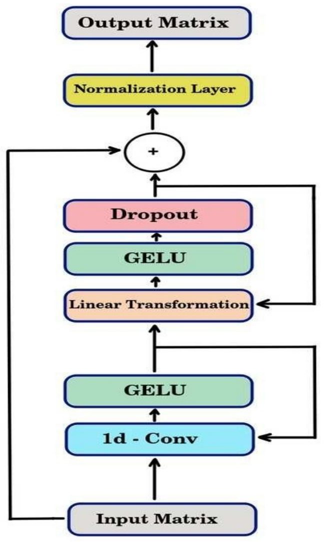

Fig. 2. XLCNN feed forward neural network structure.

The feed forward network accepts as input a vector of dimensions [512 x 1 x 512], which is multiplied by the learnable weight of dimensions [512 x 1 x 2048]. The [512 x 1 x 2048] dimension matrix is multiplied by a constant α value and the result of the multiplication between the learnable bias and the beta value is added. This convolutional equation is represented as follows:

$$C(x)=\beta\cdot b_m+a\cdot(x\cdot w_m) \tag{3}$$

where β and α were set to 1, b_m is the bias, $x\in Rd$ is the input vector and w_m is the learnable weight.

In the next step the error of the above function was calculated using the Gaussian error function [33] which is given by the completion of the Euler error [34] elevated to -x in the square with limits from -x to x. The effect of completion multiplied by 2 for the root π gives the possibility that a normally distributed random variable having an average of 0 and a variation of 0.5 falls into the region [−x, x]. The Gaussian error function can be computed:

$$erf(x)=\frac{2}{\sqrt{\pi}}\int_0^{x} e^{-x^2}dx \tag{4}$$

where x is the result of the convolutional equation. Euler's equation was used to calculate the Median Absolute Deviation (MAD) [35] through the equation:

$$MAD(x)=\sigma\cdot 2\cdot erf(x)^{-1}\cdot\frac{1}{\,} \tag{5}$$

where σ=1 is the standard deviation, erf(x) is the Gaussian error function and x is the input vector.

The Cumulative Distribution Function (CDF) [36] was then calculated, which shows the probability that a value of the function is between zero and a value x, as shown in the following mathematical form:

$$G(x)=MAD(x)\cdot\int_0^x e^{\frac{-(x-\mu)^2}{2\cdot\sigma}}dx \tag{6}$$

where the parameter μ is the mean or expectation of the distribution, σ is the standard deviation and x is the input vector. The result of function G(x) was used as input for the convolutional network, but also used as input for function G(x). The result of the second iteration marks the end of the first sub-infrastructure and the start of the second as described below.

Initially, the linear transformation function accepts as input the result of G(x) whose dimensions are [512 x 1 x 512] and produces a dimension matrix [512 x 1 x 2048]. It is essentially a mapping between the value G(x) and the weights that are towards the learning process [37]. The mathematical representation of the above sentence is described as:

$$y=G(x)\cdot w^T+b \tag{7}$$

where w is the learnable weight of the function y, b the biases and G(x) is the CDF.

After the application of the GELU function which takes as input the result of the y function, some elements of the matrix with probability p=0.1 are zeroed according to the definition of the Bernoulli distribution [38]. Those tensors that remained intact and did not zero were multiplied by a factor equal to $\frac{1}{1-p}$. The Bernoulli distribution is applied as:

$$f(x,p)=\begin{cases}1-p, & if\ x=0\\ p, & if\ x=1\end{cases} \tag{8}$$

In the second iteration of the second sub-infrastructure the result of the function f(x, p) is a matrix of size [512 x 1 x 512] and was added with the tensors that were used as input before they were introduced into the feed forward network. In order to reduce the training time of the feed forward neural network, but also to reduce the sensitivity in the final state, the normalization function [39] was introduced which is given by the following formula:

$$y=\frac{x-E[x]}{\sqrt{Var[x]+\varepsilon}}\cdot\gamma+\beta \tag{9}$$

where $\varepsilon = 10^{-5}$ is the denominator for numerical stability, γ and β are learnable affine transform parameters and x is the input matrix. After this step the output matrix size remained [512 x 1 x 512], as shown in Fig. 1.

This repetitive structure but also the architecture itself makes the XLCNN unique in its kind. The purpose of the repeating structure is to develop more complex connections between neurons without having to increase the size of the matrix used as input, thus reducing the execution time of mathematical operations.

IV. Application on malware detection

A state of the art analysis was used to identify the malware which results from the combination of XLCNN with the metadata of the Windows executable files. Different sources were used to create the data set and combined for the purpose of model training. To create the tensors we developed a tokenizer which is specially trained for the purpose of detecting malware.

Initially, executable files consisting of 2 categories were collected, the infected files and the benign ones. Malicious archives come from two different sources, the Virus Total [40] and the Das Malwerk [41]. The benign files were collected from the 3 distributions of Windows 7, 8 and 10, which did not contain infected files, as they came from clean installations. A python script file was then created and the *pefile* library [42] was used to export metadata from all executable files collected, whether they were malware or benign. They were then stored in a csv file and divided into training data and testing data. The metadata became tensors through our custom tokenizer and was used as input for the XLCNN model. After the training process the model recognized the hostile executable files with a success rate of 98.88%. The following sections present in detail the steps followed for dataset creation, metadata extraction and the comparison results between XLCNN and other Transformer models.

A. Source collection for malware detection

Two different data sets for malicious executable files were used and created from scratch. The first one was created with the help of VirusTotal [40]. VirusTotal is one of the most popular and widely used scanning services by researchers and industry professionals. VirusTotal provides file scanning services (for malware analysis) and URL scanning services (for detecting malware and phishing hosts). It works with more than 70 security vendors to aggregate the results of their analyses [43].

The second set of data was retrieved from Das Malwerk [41] which has various types of malware on its website. To retrieve the files, a python script was created from scratch using its *request* [44] and *beautifulsoup* libraries [45].

The total number of malicious files retrieved from VirusTotal and Das Malwerk was 3117 with no duplicates, while the total number of benign data retrieved from the three Windows distributions was 1794, also without duplicates. The combination of the total data constitutes the final number 4911. Of these, 10% of the malicious data and 10% of the benign data were used to verify the Transformer models. Therefore, 4452 size of data were used for training and 446 size of data for testing.

B. Metadata Extraction

With the growing availability of malware resources the quality of the learning process depends on the availability of these resources as well as the amount and expression of metadata used to explain them. The availability of malware resources is largely provided by storing these objects or their metadata in digital repositories. Reserves generally provide malware resources for exploration, demonstration and acquisition. Metadata is an important factor in the ability to find learning objects as information provides additional details of learning objects. These descriptions may relate to memory calls, address allocations, product name, product version, entropy and hash signature.

Metadata were divided into two categories [46]. The first category consists of metadata that describes the properties of the object that are not related to the domain to which the object belongs. This metadata is general and can be applied to all learning objects regardless of domain or discipline. Examples of such metadata are file format, language and so on.

The second category pertains to metadata that describes learning objects with domain-specific information. Many domains have developed classifications that classify content within a particular domain. As an example of a domain-specific metadata is the description of memory allocations in executable files in the field of cyber security.

Note that no feature export algorithm was used, but all available metadata was used as input for the Transformer models. The metadata in this research project was extracted from the executable Windows files using a python library called "*pefile*" [42]. *Pefile* is a multi-platform Python module to parse and work with Portable Executable (PE) files [47]. Most of the information contained in the PE file headers is accessible as well as all the sections details and data. The structures defined in the Windows header files are accessible as attributes in the PE instance. The naming of fields/attributes tries to adhere to the naming scheme in those headers. Some of the tasks that Pefile library makes possible are [42]:

- Inspecting headers
- Analyzing of sections' data
- Retrieving embedded data
- Reading strings from the resources
- Warnings for suspicious and malformed values

V. Experimental Implementation and Results

Fig. 3 shows the process of training the XLCNN for a period of 50 epochs while figure Fig. 4 shows the process during which the neural network is tested on data which was not in the training set to determine the progress of the model. As shown in Fig. 3 the model has a linear increase in performance until the 4th epoch while from the 5th to the 14th there is no further increase and the performance remains constant. From the 15th epoch until the last a logarithmic increase is observed. The maximum accuracy value was reached in the 43rd epoch with the percentage reaching 99.91% and the loss being 0.3%. The loss function used during training is Cross Entropy Loss [32], as it is effective in binary classification.

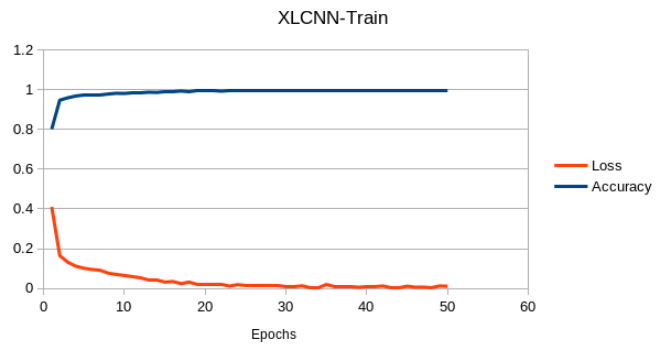

Fig. 3. XLCNN training process.

In Fig. 4 during the testing phase our model has a linear increase from the first to the fourth epoch, while the value of the accuracy varies from 96% to 98%. We notice that during the test process local maxima and local minima appear. The total maximum value is reached at the 23rd epoch and reaches 98.88% with a loss value of 0.08. Note that during the fine-tune process, our model stores those weights that have the highest accuracy during the testing process regardless of how large the percentage is during training. XLCNN took just 23 epochs to achieve the maximum accuracy. The blue line represents XLCNN's accuracy and the red line represents the loss.

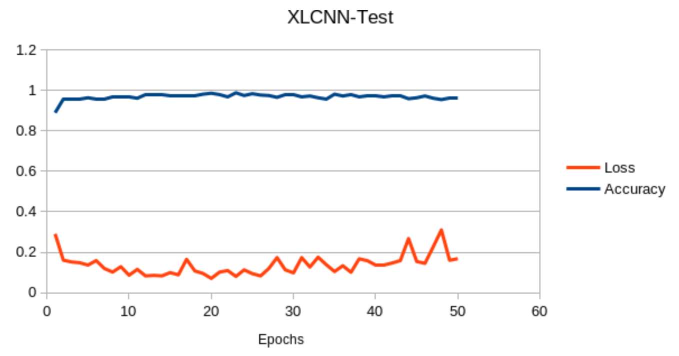

Fig. 4. XLCNN testing process.

Below are the results and comparisons between different transformer models. The models that were compared are four and they refer to our proposed XLCNN, XLNet [48], BERT [49] and Transformer-XL [50]. In order for the comparison to be objective, the same datasets were used in both training and testing with the same number of epochs for all the models.

A. Testing phase with default tokenizers

We present the results obtained during the testing process using the default tokenizers used by each model. For XLCNN we used our own BPE tokenizer that we created, for XLNet we used *xlnet-base-cased* [51] tokenizer, for BERT we used *bert-base-uncased* [52] and for Transformer-XL we used *transfo-xl-wt103* [53].

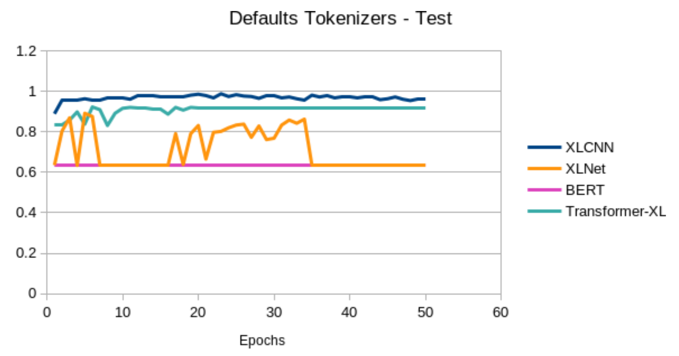

Fig. 5. Comparison of models with default tokenizers.

As can be seen in Fig. 5, our proposed XLCNN and Transformer-XL show a logarithmic increase with the exception of XLNet which has many variations in its values. XLCNN achieved the highest prediction accuracy of 98.88% in the 23rd epoch, Transformer-XL had an accuracy rate of 92.38% in the 6th epoch and XLNet 89.24% in the 5th epoch. The BERT model shows that it has a weakness in learning with the result that its prediction remains stable at 63.45%. This fact is mainly due to the type of data on which the bert-*base-uncased* tokenizer was trained. BERT's tokenizer was trained in common english vocabulary features and therefore rejected many features that existed in executables metadata.

As we can see, the XLCNN model we propose had the best prediction accuracy and the most stable trajectory without large fluctuations compared to the rest of the models.

B. Testing phase with our custom XLCNN tokenizer

The second experimental procedure involves comparing all models to which we applied our custom made tokenizer.

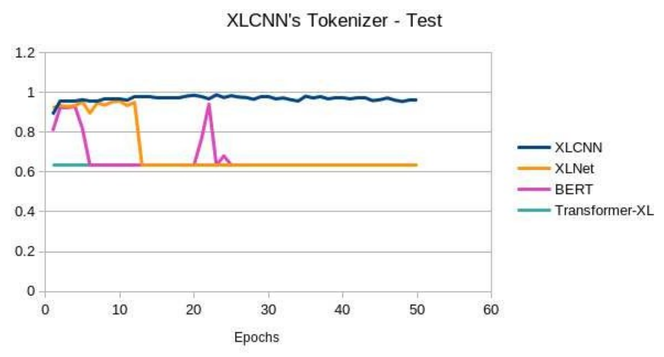

Fig. 6. Comparison of models with our proposed tokenizer.

In Fig. 6 the use of our proposed tokenizer shows that it helped to increase the accuracy of XLNet and BERT. More specifically, it increased the success rate of XLNet by 6.5% with a prediction accuracy of 95.74% in the 10th epoch, while for BERT there was an increase rate of 30.72% with a final prediction accuracy of 94.17% in the 22nd epoch. In contrast, the accuracy of the Transformer-XL measurement showed a decrease of 28.93% with the accuracy rate reaching 63.45% in the 1st epoch and remaining stable until the last. The XLCNN also surpassed the rest of the models in this case as it maintained its accuracy (98.88%) at higher percentages, not only in the epoch when it achieved the maximum percentage but also in all the other epochs. This fact indicates that the architectural structure it follows is clearly superior to the rest of the models, even if they use the same tokenizer as the one of XLCNN.

C. Comparison of efficiency between models

Table 1 shows the results of the Transformer models compared. As we can see, the proposed XLCNN had the greatest effectiveness in its predictions in relation to all the other models. XLNet with the custom tokenizer follows with high success rates but with 3.14% lower than XLCNN. In the third place comes BERT with the custom tokenizer, in the fourth place comes Transformer-XL with the default tokenizer and in the last place comes the XLNet also with the default tokenizer. In the last two positions remain BERT with the default and Transformer-XL with the custom tokenizer respectively. These two models predicted only the malicious data without being able to identify any benign ones, with the result that they could not calculate the recall, the precision and f1-score.

Table 1. Transformer efficiency comparison

Models	Accuracy (%)	Recall (%)	Precision (%)	f1-score (%)	ROC_AUC (%)	Loss
Proposed XLCNN	98.88	97.55	99.38	98.45	98.6	0.08
XLNet-base-cased-tokenizer	89.24	77.3	91.97	84.0	86.71	0.31
XLNet-proposed-tokenizer	95.74	88.96	99.32	93.85	94.3	0.17
BERT-base-uncased-tokenizer	63.45	-	-	-	50.0	0.71
BERT-proposed-tokenizer	94.17	89.57	94.19	91.82	93.2	0.24
Transformer-XL-transfo-xl-wt103-tokenizer	92.38	93.25	86.86	89.94	92.56	0.22
Transformer-XL-proposed-tokenizer	63.45	-	-	-	50.0	0.7

D. Comparison of speed during training phase

All the Transformer model experiments were carried out on the same machine so that the comparison of time and performance values is meritorious. Table 2 shows all the information about the operating system, the harware, and the framework we used.

Table 2. Experiment specifications

Distribution	Arch Linux x86_64
Kernel	5.15.79-1-lts
CPU	11th Gen Intel i7-11800H
GPU	Nvidia GeForce RTX 3080 Max-Q (16GB)
Cuda Cores	6144
Cuda version	11.8
Python version	3.10.8
Framework	pytorch 1.11.0+cu113

Table 3 shows the speed of each model during the training process for 50 epochs, the tensor length used as input, the batch size and the memory capacity consumption of GPU. As we can see, XLCNN uses 2.5% less GPU VRAM than XLNet for the same batch size, while Transformer-XL, due to its large size and memory limitation, used the maximum allowed batch size equal to 1. Transformer-XL has the longest execution time and one of the smallest prediction rates as shown in Table 1, while it consumes a lot of computing power. The execution time is 79.7% higher than the average of the other models. In the opposite case, the proposed XLCNN has one of the shortest execution times, the highest percentage of accuracy and a small consumption of computing resources. XLCNN is 49.8% faster than the average execution time of the other Transfomer models. BERT uses the least resources and has the fastest execution time of all the other models, however the metrics rate are kept at lower levels compared to XLCNN and XLNet.

Table 3. Parameters of models and execution parameter results

Model	Input Length	Batch size	VRAM (MiB)	Time (hour:min:sec)
Proposed XLCNN	512	4	8997	10:01:17
XLNet-base-cased-tokenizer	512	4	9201	10:27:59
XLNet-proposed-tokenizer	512	4	9201	09:53:23
BERT-base-uncased-tokenizer	512	4	6333	10:56:14
BERT-proposed-tokenizer	512	4	6333	03:54:37
Transformer-XL-transfo-xl-wt103-tokenizer	512	1	9251	46:43:06
Transformer-XL-proposed-tokenizer	512	1	9251	40:33:31

VI. Conclusion

In the present research, the creation of the state of the art Transformer model XLCNN proved that the addition of CNNs to the feedforward network in a combination with our proposed Tokenizer, Linear Transformation, GELU, Dropout and the Normalization layer offer a higher success rate of 98.88% compared to the other Transformer models. The effectiveness and efficiency of four different Transformer models, the proposed XLCNN, XLNet, BERT and Transformer-XL, were compared. In the experimental study, it was shown that the proposed XLCNN was clearly more effective than the rest, while XLNet and Transformer-XL achieved accuracy rates of 89.24% and 92.38%, respectively. BERT intervened at a constant 63.45% of accuracy. In the 2nd experimental procedure the same models were compared with the addition of our own tokenizer previously used for XLCNN. We observed that the execution speed of all models was increased by reducing the execution time achieved in 50 epochs, while XLNet and BERT showed significant increases in prediction by 6.5% and 30.72% respectively. In the opposite case, the application of the proposed tokenizer in Transformer- XL, although it increased the execution speed, reduced the accuracy by 28.93%. In all cases the proposed XLCNN maintained the highest accuracy rate at 98.88% while the training speed over 50 epochs was 49.8% higher than the average of the other models. This fact proves that we have created a clearly superior architecture compared to the rest of the Transformer models, that needs less computing power and shorter execution time to achieve better results. The success of the XLCNN was determined by its effective classification, while it was trained with a small amount of data, limitation that it cannot be ignored.

REFERENCES

[1] Zeidanloo, H. R., Tabatabaei, F., Amoli, P. V., & Tajpour, A. (2010, July). All About Malwares (Malicious Codes). In Security and Management (pp. 342-348).

[2] Jha, S., Prashar, D., Long, H. V., & Taniar, D. (2020). Recurrent neural network for detecting malware. computers & security, 99, 102037.

[3] Lipton, Z. C., Berkowitz, J., & Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. arXiv preprint arXiv:1506.00019.

[4] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

[5] Rahali, A., & Akhloufi, M. A. (2021). MalBERT: Using Transformers for Cybersecurity and Malicious Software Detection. arXiv preprint arXiv:2103.03806.

[6] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019, January). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In NAACL-HLT (1).

[7] Hardy, W., Chen, L., Hou, S., Ye, Y., & Li, X. (2016). DL4MD: A deep learning framework for intelligent malware detection. In Proceedings of the International Conference on Data Science (ICDATA) (p. 61). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

[8] Rhode, M., Burnap, P., & Jones, K. (2018). Early-stage malware prediction using recurrent neural networks. computers & security, 77, 578-594.

[9] Pascanu, R., Stokes, J. W., Sanossian, H., Marinescu, M., & Thomas, A. (2015, April). Malware classification with recurrent networks. In 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 1916-1920). IEEE.

[10] Bojan Kolosnjaji, Apostolis Zarras, George Webster and Claudia Eckert. Deep Learning for Classification of Malware System Call Sequences. In: Kang B., Bai Q. (eds) AI 2016: Advances in Artificial Intelligence. AI 2016. Lecture Notes in Computer Science, vol 9992. Springer Verlag

[11] Ullah, F., Alsirhani, A., Alshahrani, M. M., Alomari, A., Naeem, H., & Shah, S. A. (2022). Explainable malware detection system using transformers-based transfer learning and multi-model visual representation. Sensors, 22(18), 6766.

[12] Seneviratne, S., Shariffdeen, R., Rasnayaka, S., & Kasthuriarachchi, N. (2022). Self-Supervised Vision Transformers for Malware Detection.*IEEE Access*,*10*, 103121-103135.

[13] Li, M. Q., Fung, B. C., Charland, P., & Ding, S. H. (2021). I-MAD: Interpretable malware detector using Galaxy Transformer. Computers & Security, 108, 102371.

[14] Şahin, N. (2021). Malware Detection Using Transformers-based Model GPT-2 (Master's thesis, Middle East Technical University).

[15] Lu, Q. (2021). An Investigation on Self-Attentive Models for Malware Classification. (Master's thesis, University of Alberta)

[16] Almeida, F., & Xexéo, G. (2019). Word embeddings: A survey. arXiv preprint arXiv:1901.09069.

[17] Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., ... & Liu, T. (2020, November). On layer normalization in the transformer architecture. In International Conference on Machine Learning (pp. 10524-10533). PMLR.

[18] Jiang, Z., & Zhang, S. (2020, May). Research on Task-oriented Dialogue Based on Modified Transformer. In Journal of Physics: Conference Series (Vol. 1544, No. 1, p. 012188). IOP Publishing.

[19] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In Advances in neural information processing systems (pp. 5998-6008).

[20] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.

[21] Yan, M. (2019). Adaptive learning knowledge networks for few-shot learning. IEEE Access, 7, 119041-119051.

[22] Loshchilov, I., & Hutter, F. (2018, September). Decoupled Weight Decay Regularization. In International Conference on Learning Representations.

[23] Kingma, D. P., & Ba, J. (2015, January). Adam: A Method for Stochastic Optimization. In ICLR (Poster).

[24] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.

[25] Hendrycks, D., & Gimpel, K. (2016). Gaussian error linear units (gelus). arXiv preprint arXiv:1606.08415.

[26] Maass, W., Natschläger, T., & Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. Neural computation, 14(11), 2531-2560.

[27] Krueger, D., Maharaj, T., Kramár, J., Pezeshki, M., Ballas, N., Ke, N. R. & Pal, C. J. (2017, January). Zoneout: Regularizing RNNs by Randomly Preserving Hidden Activations. In ICLR (Poster).

[28] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

[29] Accessed on https://github.com/google/sentencepiece

[30] Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., & Le, Q. V. (2019). XLNet: Generalized Autoregressive Pretraining for Language Understanding. Advances in Neural Information Processing Systems, 32, 5753-5763.

[31] Wu, J. (2017). Introduction to convolutional neural networks. National Key Lab for Novel Software Technology. Nanjing University. China, 5(23), 495.

[32] Zhang, Z., & Sabuncu, M. R. (2018, January). Generalized cross entropy loss for training deep neural networks with noisy labels. In 32nd Conference on Neural Information Processing Systems (NeurIPS).

[33] Andrews, L. C. (1998). Special functions of mathematics for engineers (Vol. 49). Spie Press.

[34] Butcher, J. C. (2016). Numerical methods for ordinary

differential equations. John Wiley & Sons.

[35] Rousseeuw, P. J., & Croux, C. (1993). Alternatives to the median absolute deviation. Journal of the American Statistical association, 88(424), 1273-1283.

[36] Deisenroth, M. P., Faisal, A. A., & Ong, C. S. (2020). Mathematics for machine learning. Cambridge University Press.

[37] Li, C. K., Rodman, L., & Šemrl, P. (2002). Linear transformations between matrix spaces that map one rank specific set into another. Linear algebra and its applications, 357(1-3), 197-208.

[38] Grinstead, C. M., & Snell, J. L. (1997). Introduction to probability. American Mathematical Soc..

[39] Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer Normalization. stat, 1050, 21

[40] Virustotal, accessed on https://www.virustotal.com

[41] Das Malwerk, accessed on https://dasmalwerk.eu/

[42] Pefile, accessed on https://github.com/erocarrera/pefile

[43] Peng, P., Yang, L., Song, L., & Wang, G. (2019, October). Opening the blackbox of virustotal: Analyzing online phishing scan engines. In Proceedings of the Internet Measurement Conference (pp. 478-485).

[44] Requests, accessed on https://github.com/psf/requests

[45] Beautiful Soup, accessed on https://www.crummy.com/software/BeautifulSoup/

[46] Alhaag, A. A., Savic, G., Milosavljevic, G., Segedinac, M. T., & Filipovic, M. (2018). Executable platform for managing customizable metadata of educational resources. The Electronic Library.

[47] Pietrek, M. (2002). An in-depth look into the Win32 portable executable file format, part 2. MSDN Magazine, March.

[48] Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., & Le, Q. V. (2019). XLNet: Generalized Autoregressive Pretraining for Language Understanding. Advances in Neural Information Processing Systems, 32, 5753-5763.

[49] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019, January). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In NAACL-HLT (1).

[50] Dai, Z., Yang, Z., Yang, Y., Cohen, W. W., Carbonell, J., Le, Q. V., & Salakhutdinov, R. (2018). Transformer-xl: Language modeling with longer-term dependency.

[51] Xlnet-base-cased tokenizer, accessed on https://huggingface.co/xlnet-base-cased

[52] Bert-base-uncased tokenizer, accessed on https://huggingface.co/bert-base-uncased

[53] Transfo-xl-wt103 tokenizer, accessed on https://huggingface.co/transfo-xl-wt103

Acknowledgments

We would like to thank the professor of Information Systems Security at the University of Macedonia, Ioannis Mavridis, for the constructive discussion we had on the research work.

Konstantinos Giapantzis holds a Diploma in Materials Science and Engineering from the University of Ioannina, Greece, since 2018. He obtained his M.Sc. Degree in Applied Informatics from the University of Macedonia, Greece, in 2022. From March 2023 he is a PhD candidate at the Department of Digital Systems of the University of Piraeus in Athens Greece. He has been a researcher, a member of the Centre of Research and Technology Hellas (CERTH) team since 2020. He is involved in the development of artificial intelligence tools related to cyber security topics in the SHOW, SANCUS and ULTIMO projects which are funded by the European Union's Horizon 2020 Research and Innovation Programme.

Spyros T. Halkidis was born in Thessaloniki, Macedonia Greece. He received the BS and MS degrees from the Department of Computer Science, University of Crete in 1996 and 1998 respectively. He also received an MBA from the University of Macedonia in 2000. He worked as a Software Engineer for Metropolis Informatics S.A. from December 2000 to September 2003. He worked towards a Ph.D. in Applied Informatics at the Department of Applied Informatics, University of Macedonia, Greece starting December 2003, and he received it in March 2008. He also served as a school teacher from 2004 until 2017. Since October 2017 he works as Teaching and Research staff at the Computational Methods and Operations Research Laboratory, Department of Applied Informatics, University of Macedonia, Greece. His current research interests include Cyber Security, Cryptography and Digital Twins.