Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Static Malware Detection Using Stacked BiLSTM and GPT-2

DENİZ DEMİRCİ[1], NAZENİN ŞAHİN[1], MELİH ŞIRLANCI[2],CENGIZ ACARTURK[3,1] (Member, IEEE)
[1]Informatics Institute, Middle East Technical University, 06800 Ankara, Turkey
[2]Ohio State University, 43210, Columbus, OH, USA
[3]Jagiellonian University, 30-252 Krakow, Poland

Corresponding author: Cengiz Acarturk (e-mail: acarturk@acm.org).

This work was partially supported by Middle East Technical University funds

ABSTRACT In recent years, cyber threats and malicious software attacks have been escalated on various platforms. Therefore, it has become essential to develop automated machine learning methods for defending against malware. In the present study, we propose stacked bidirectional long short-term memory (Stacked BiLSTM) and generative pre-trained transformer based (GPT-2) deep learning language models for detecting malicious code. We developed language models using assembly instructions extracted from `.text` sections of malicious and benign Portable Executable (PE) files. We treated each instruction as a sentence and each `.text` section as a document. We also labeled each sentence and document as benign or malicious, according to the file source. We created three datasets from those sentences and documents. The first dataset, composed of documents, was fed into a Document Level Analysis Model (DLAM) based on Stacked BiLSTM. The second dataset, composed of sentences, was used in Sentence Level Analysis Models (SLAMs) based on Stacked BiLSTM and DistilBERT, GPT-2 Domain Specific Language Model (GPT2-DSLM), and GPT-2 General Language Model (GPT2-GLM). Lastly, we merged all assembly instructions without labels for creating the third dataset; then we fed a custom pre-trained model with it. We then compared malware detection performances. The results showed that the pre-trained model improved the GPT2-DSLM and GPT2-GLM detection performance. The experiments showed that the DLAM, the SLAM based on DistilBERT, the GPT2-DSLM, and the GPT2-GLM achieved 98.3%, 70.4%, 86.0%, and 76.2% F1 scores, respectively.

INDEX TERMS Malware Detection, Static Analysis, Stacked BiLSTM, GPT-2

I. INTRODUCTION

The fast development of Information and Communication Technologies (ICT) has significantly influenced the variety of malicious content, besides the complexity of mitigation methods that aim to detect and prevent malicious code from functioning. Malicious codes also gained the ability to spread rapidly in computer networks due to enhanced connectivity of end-user computers and servers, cloud systems, smartphones, and IoT devices. The exponential growth in malware content also leads to substantial economic loss. According to the Deep Instinct reports, in 2020, the variety of malware increased by more than three times, and ransomware increased by more than four times compared to 2019 [1].

Malware code pieces usually aim to violate a system's or device's security policies by executing themselves on the system. Attackers may exploit vulnerabilities in computer systems to steal sensitive information, spy on the infected system, or take over the system's control. Despite the general conceptualization of malware as malicious "files", malicious code pieces are usually embedded in a file as a part of it rather than representing the whole file. In other words, malware code pieces are typically "wrapped into" executable files [2] as payloads. From a system-level perspective, malicious lines of a software code are basic units that run a series of instructions at the machine level. Accordingly, malware refers to the commands that run instructions for malicious purposes. These instructions may perform system calls for input-output functions and a set of functions that operate computer memory and file systems. Before reaching the end-user system, detecting these malicious instructions may provide a solution to prevent infection. Therefore, the foremost challenge in malware detection is to identify lines of a software code in a file so that the suspected file, which includes the malware, has malicious functionality.

Malware detection methods rely on signature databases, including malicious instruction patterns in today's practice. The signature databases are used for matching against a signature generated from a newly encountered executable. Nevertheless, more efficient mitigation methods are needed due to the fast expansion of malicious software on the Internet and their self-modifying abilities, as in polymorphic and metamorphic malware. Recently, the detection of malicious lines of code has been critical for the development of efficient malicious detection. A significant challenge is that the source codes for executable files are not usually accessible in compiled form. Therefore, the assembly instructions are the best candidate to unveil malicious functionality in a suspected file. In the present study, we focus on modeling assembly instructions by deep learning techniques.

Researchers and practitioners have proposed various techniques that may be classified into three major groups to address such threats: static, dynamic, and hybrid analysis techniques [3]. Dynamic analysis executes the file for malware detection, while static analysis detects malware by scanning the entire file without running the executable. Static analysis has certain drawbacks against dynamic analysis [4] in resisting malicious deformation techniques such as obfuscation and dynamic code loading. On the other hand, it consumes fewer resources, identifies malware efficiently, and mitigates it before reaching end-users or servers. Moreover, static analysis is scalable and usable when facing batch unknown malware detection and may traverse all possible execution paths of the executable file. For instance, the previous work [4] shows that the dynamic analysis may achieve high accuracy rates in malware detection through assembly language modeling. Nevertheless, it suffers from the runtime overhead of dynamic data collection, e.g., run trace collection, which makes the dynamic approach hard to use in practice.

In the present study, we aim to exploit various advantages of static analysis, such as low runtime overhead to achieve high accuracy rates closer to the dynamic analysis approach by combining static analysis with advanced neural network modeling. In particular, we propose malware detection approaches using natural language processing (NLP) techniques with DL algorithms. The proposed algorithms, namely the bidirectional long short-term memory (BiLSTM) model and the generative pre-trained transformer 2 (GPT-2) detect malicious code pieces by examining assembly instructions obtained from static analysis results of Portable Executable (PE) files. Our BiLSTM model processes a sequence of input elements across time to learn and analyze the patterns. In contrast, the transformers-based GPT-2 model enables modeling long dependencies between input sequence elements with parallel sequence processing, in which sequential data constituents can connect with others simultaneously. We use the perspective of NLP modeling by DL to extract similar characteristics, i.e., syntactic and semantic characteristics of assembly instructions. Our models were designed to effectively learn and extract the features and characteristics of assembly language and classify the polarity of files.

The article is organized as follows. First, we present a review of the studies about malware detection. Next, we describe the approach of the present study, datasets, parameters, and the proposed models (viz. BiLSTM and GPT-2). Then, we report the results and compare the models. Finally, we discuss the results and present the limitations of the study.

II. RELATED WORK

The classification algorithms usually require two main steps, feature-related and classification-related—the former consists of feature extraction, selection, and reduction processes. In contrast, the latter encloses deciding the best algorithm to classify or detect the family of executables [5]. As an example, in [6], Merabet et al. compare the steps required for machine learning-based malware detection, including feature extraction, selection, and reduction. First, by applying different feature extraction techniques, they observe the effects of signatures, `dll` functions, binary string, and portable executable (PE) headers. Then they evaluate specific techniques, such as Principal Component Analysis (PCA) and Random Forest, and numerous classification algorithms, such as Support Vector Machine (SVM), Random Forest, and Artificial Neural Network, for comparative analysis.

Feature selection methods depend on a researcher's approach to the problem. For example, Bilar [7] stated that the difference between malware files and benign files was statistically significant in opcode frequency distributions. Moreover, they used rare opcodes as a predictor for malware detection. Santos et al. [8] studied the incidence of opcode sequences. They investigated the relationships among the opcodes and used statistical information to detect variants of known malware families. More recently, the method in [9] was based on analyzing the frequency of opcode sequences to create a semi-supervised machine learning classifier using a set of labeled and unlabeled data to detect novel malware. A major issue with the proposed models was that they used the opcode sequences of a fixed length, 1 and 2, for each malware and some of its variants. Therefore, those models based on opcode frequency do not adapt to changes in data, thus being not scalable to detecting polymorphic, metamorphic, and novel malware codes. In another study [10], Anderson et al. used an extensive dataset containing nine hundred thousand malicious files. They divided the dataset into the training set, validation set, and testing set, each including three thousand malicious files (viz. the EMBER dataset). They used a cross-platform library LIEF (Library to Instrument Executable Formats)[1] to parse malicious and benign executables and to extract features. The features consisted of eight groups of information. Five were the output of LIEF, such as general file information, header information, section information, imported and exported functions in `json` format. The other features obtained from the raw file were byte histogram, byte-

[1] https://lief.quarkslab.com (retrieved on February 17, 2021)

entropy histogram, and string information. They employed LightGBM (Light Gradient Boosting Machine)[2] on the features, achieving 98.2% detection accuracy. A major challenge was the processing requirement for feature extraction to make the data suitable for the model, as well as the training of the model. The hardware consisted of two TitanX model GPUs to deploy the LIGHTTGBM model, requiring 250 hours for ten epochs.

Byte n-gram is another technique commonly used in malware classifiers, motivated by its requirement for minimal or no domain knowledge. For instance, [11] utilized n-gram of the opcodes as a feature vector for classification. They detect unknown malware based on text categorization. Their methodology was more successful than byte sequence n-gram representation, and their results indicated 99.0% accuracy when trained by a set with lower than 15% malicious file content. Besides the n-gram techniques, other studies, such as [12] applied TF and TF-IDF representations for each opcode n-grams, achieving 95.6% accuracy. Other studies, such as [13] extracted the features from n-gram opcode sequences. They employed five different machine learning classification algorithms to detect and classify ransomware families. Their approach achieved 91.43% accuracy when used with datasets consisting of actual ransomware data. Although modeling by byte n-gram techniques revealed high accuracy values, they have the drawback of over-fitting and overestimating the accuracy in the malware detection domain, as also stated in [14].

A known issue in the use of the traditional malware detection methods is that they require human control over feature extraction and feature engineering, resulting in time-consuming processes in the workflow and disrupting the automated detection process [15]. Recently, deep learning techniques have become widely applied in malware analysis due to their advantage in learning from data without requiring intense human intervention. Deep learning techniques for malware detection include various approaches. A prominent one is the variations over the recurrent neural networks (RNN) and convolutional neural networks (CNN). Another is the attention-mechanism approaches, such as transformers. Their effectiveness depends on the input features extracted from the dataset. CNN is a method that computes the distances in the input sequences when they are mapped to output sequences. It employs feature extraction by converting malware into images similar to the use of CNNs in recognition. For instance, Krcal et al. [16] treated twenty million unpacked half megabyte Portable Executables (PEs) as a sequence of bytes and applied CNN for malware detection. The network had four convolutional layers and four fully connected layers. Instead of a global max-pooling layer, they used a global mean pooling layer after the convolutional layers. Moreover, their best effort was 97.1% accuracy. In [17], Khan et al. investigated GoogleNet, and they assembled five different ResNet models from opcodes of binary files using images. Histogram standardization enlargement and disintegration techniques were used to upgrade images to distinguish between malicious and benign opcode images. The accuracy rate of GoogleNet was 74.5%, and the best accuracy rate among ResNet models was 88.36%. In another study [18], the authors employed CNN to classify malware opcode images. The accuracy rate of correctly classified binary files was 98%. Although they achieve high accuracy rates, a major problem of the CNN-based malware detection models is that they do not perform well in case of elaborating the data with specific methods, such as obfuscation [19] and [20]. Given that obfuscation is widely used in malware codes in the wild, CNN-based methods would find limited use in practice.

An alternative is RNN, which overcomes the limitations of CNN modeling. In an RNN model, the data collected from malware are put into a sequential format, as in text classification tasks in NLP (Natural Language Processing), to achieve high performance as in CNN. For instance, in [21], an RNN model with Word2Vec feature vector achieved the highest area under the curve AUC value of 0.92 and a good variance among several feature vectors. Although the standard RNN architecture achieved high accuracy rates in a variety of domains, including malware detection, certain problems, such as exploding and vanishing gradients, make it ineffectual in terms of robustness. Similarly, [22] used opcodes and operands as features, mapping with different word embedding techniques to word vectors. They used word embedding results to feed into models for malware detection. They employed Long Short-Term Memory (LSTM), a complex gated RNN architecture with a long-term dependency of features. The model reached an average AUC of 0.99 for classification. Their approach is partially similar to one of the models developed in the present study in terms of the basic neural network architecture. Nevertheless, a significant drawback was the unbalanced dataset, which included 969 malware and 123 benign files.

In another study [23], Instruction2Vec was used with both opcode and operand information. A nine-dimensional feature vector was used to resemble registers and addresses. Assembly instructions were split, and each token was encoded as unique index numbers. An opcode took one token in the setup, and a memory operand took up to four tokens, including base register, index register, scale, and displacement. The tests with Instruction2Vec datasets reached 91.1% accuracy. In another study [24], the researchers proposed a malware classification method employing the fastText-based Bi-LSTM algorithm. They disassembled malicious files and obtained the list of opcodes and API function names to train the model. Their 2-layer BiLSTM model achieved 96.76% accuracy. A similar study [25] utilized static analysis to disassemble malware and obtain the assembly codes. To reduce junk codes that belonged to one of the anti-static analysis techniques, they used the attention mechanism.

[2] https://github.com/microsoft/LightGBM (retrieved on February 17, 2021)

Their CBOW model was based on LSTM to classify malware files and used a vector dimension of 100 with a window size of 10. The model accuracy was 94.25%. Compared to the proposed models in [23]–[25], we improve the accuracy rate further, to 98.3% with our DLAM model, as presented in the next section.

Recently, transformer-based models, defined with attention mechanism, are frequently used for malware detection since the data collected from malware are processed in parallel rather than in a sequential format. For instance, [26] used benign and malicious assembly codes obtained from the static content of an executable. Their model, Interpretable MAlware Detector (I-MAD) based on transformers, combined a network component called the Galaxy Transformer network that understood assembly code at the basic block (a sequence of assembly instructions), assembly function (a set of basic blocks), and executable levels (a sequence of bytes). Its feed-forward neural networks provided interpretations for its detection results by quantifying the impact of each feature for the prediction. As a result, they achieved 97.7% accuracy with the assembly functions. A drawback of the study was that their transformer topology was trained with very long sequences, whereas transformers-based successful applications, such as GPT-2, are mainly on short text, i.e., sentence-level tasks or short-document texts. Therefore, long sentences may pose a significant challenge for the transformers' time and space complexity.

In another work [27], researchers classified various malware categories with the static analysis level on the source code of Android applications. They used a transformers-based BERT model for classification, the fine-tuned it with BERT's custom pre-trained model. They obtained 97.6% accuracy with BERT and 94.1% accuracy with LSTM. The main difference between the Bidirectional Masked Language Model BERT and our approach is that the former randomly masks certain elements of the input. Its objective predicts the original vocabulary identifier of the masked word based solely on its context. On the other hand, our transformer-based models have unidirectional architectures that predict the next word given the previous words in the input sentence. The following section presents the methodology of the present study.

III. METHODOLOGY

In this section, we describe the proposed methodology for malware detection. We explain how we established the datasets. Then, we introduce the architectures of the models for the analyses.

A. APPROACH

Model building for malware detection usually begins with feature extraction, as specified by either static or dynamic analyses, and sometimes hybrid analysis. The dynamic analysis examines the behavior of PE (Portable Executable) files upon execution, whereas the static analysis processes the content of the PE files without execution. The static approach usually provides a large set of data such as PE sections, imports, symbols, and compiler strings. In traditional modeling, the models are built for extracting signatures from those information sources, occasionally via human intervention. Malware mitigation is based on comparing those signatures with the signature of an executable file of newly encountered files for malicious vs. benign detection. The signature-based malware detection is straightforward and fast, yet it may be ineffective against sophisticated malware or overlook relations. Another drawback of such detection methods is that the signatures database grows too quickly to keep up with the growth rate of new malware.

The Machine Learning (ML) algorithms, in particular Deep Learning (DL) algorithms, have been deployed to eliminate the drawbacks of traditional, signature-based malware detection. The DL is the end-to-end learning approach, which refers to training a possibly complex learning system represented by a single model, a Deep Neural Network (DNN). The network represents the complete target system, automating feature extraction nearly without preprocessing. In our study, we extract assembly codes using an open-source disassembler *objdump*. This tool creates sequences as documents or sentences. Those data are then used for model development, given that the assembly code provides accurate information for obtaining critical coding patterns. For this, we employ the disassembler output as input data to build a language model assisted with word embedding in a similar way to processing natural language. Then, by utilizing this language model, we aim to identify whether an executable file is malicious or benign. Primarily, we undertake polarity detection on executable files of assembly instructions. Below, we introduce the datasets used for developing the models.

B. DATASETS

Our datasets consist of benign and malicious executables in Portable Executable (PE) format obtained from external sources. The PE is a complicated structure, based on COFF (Common Object File Format)[3] specification, with its standard headers, optional headers, sections of various types, resources, and relocation tables. Typical COFF sections encompass code or data that linkers and Microsoft Win32 loaders do not need information about the area contents. The contents apply entirely to the software linked or executed. However, certain COFF sections have special meanings when found in executable files. Due to the special flags set in the section header, these sections are efficiently recognized by tools and loaders.

In the present study, we use the `.text` section contents of executables found in our collection (henceforth, the code section). Our collection contains Win32 PE files from Win-

[3] https://docs.microsoft.com/en-us/windows/win32/debug/pe-format (retrieved on December 5, 2021)

dows operating systems[4], and Commando VM v-2.0[5], and malicious x86 executables from the sorel-20m [28] database and VirusShare website database[6]. The sorel-20m database includes the following malware families: Adware, Flooder, Installer, Ransomware, Dropper, Spyware,Packed, Worm, Crypto-miner, File-infector, Downloader. The authors used semantic attribute tagging [29] for the sorel-20m dataset to create multi-class labeling, (e.g., the malware with sha256 value of 000008c...a9b822[7] has the tags 8, 2, 10, 10 for spyware, packed, file_infector and worm respectively). We selected 822 malware samples from sorel20m dataset which has the distribution of behavioral tags as given in Table 1. The VirusShare dataset includes the following malware families: Adware, Agent, Backdoor, Downloader, Ransomware, Riskware, Trojan, Virus, Worms. Additionally, for the purpose of the present study, we did not include packed programs since the databases that we took our samples already had unpacked versions of packed executables; therefore, we did not perform additional operation on portable executables. We chose Commando VM over the other versions of Windows OS, because it contains executables compiled using various compilers, such as Cygwin[8] and MinGW[9]. The number of the samples is presented in Table 2.

TABLE 1: Distribution of behavioral tags of selected malware from sorel-20m

Malware Family	Behavioral Tags Total Count
spyware	1689
file_infector	1434
worm	1098
adware	1044
downloader	911
packed	834
ransomware	661
dropper	569
installer	367
crypto_miner	97
flooder	61

TABLE 2: The number of malicious and benign files in the datasets.

Dataset Name	Benign Samples	Malware Samples
Sorel-20m	672	822
VirusShare	-	273
Windows OS & Commando VM (v-2.0)	314	-

After obtaining the assembly instructions in the code section of the collected executables, we saved the outputs as plain text files. Moreover, we treated opcodes and operands as words, instructions as sentences, and the complete assembly instructions in the code section as a document. We also labeled sentences and documents according to their origin, i.e. malicious or benign files. We fed the Document Level Analysis Model (DLAM) with this initial dataset, which constituted the documents. Then, we obtained another dataset from sentences labeled benign/malicious. These samples were used in building the Sentence Level Analysis Model (SLAM), the GPT-2 Domain Specific Language Model (GPT2-DSLM) and GPT-2 General Language Model (GPT2-GLM). Lastly, we gathered each malicious and benign assembly instruction without labeling to process in our custom pre-trained model, namely the third dataset.

Finally, DLAM was based on the stacked bi-directional LSTM architecture. We created two different architectures for SLAM, which were based on the stacked bi-directional LSTM and DistilBERT. On the other hand, the GPT2-DSLM, the GPT2-GLM, and our custom pre-trained models were derived from the GPT-2 transformer-based model.[10]

The overall processing pipeline is presented in Figure 1.

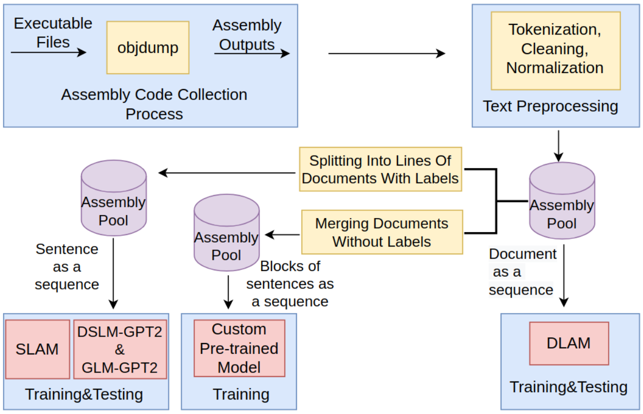


FIGURE 1: Language modeling processing pipeline

We trained and tested the DLAM and the SLAM models on a machine with a 6-Core Intel Core i9 processor with 2.9 GHz speed and 32 GB memory. In addition, we trained and tested our custom pre-trained model and the Binary Classification models on Colaboratory by Google, a Jupyter notebook-based runtime environment, to run code entirely on the cloud. There were one GPU(s) available. We used the GPU: Tesla V100-SXM2-16 GB and 32 GB memory, and the training time was limited to 24 hours.

C. MODELS

This section presents the models developed for the purpose of the present study, namely the Stacked Bidirectional Long

[4] Microsoft Windows 8.1 Pro (OS Build 9600), Microsoft Windows 10 Pro 19.09 (OS Build 18363.418)

[5] https://github.com/fireeye/commando-vm (retrieved on December 5, 2021)

[6] https://virusshare.com/ (retrieved on March 19, 2019)

[7] sha256: 000008cf1b5ecbed74f31b45e96e0fb6566b6af75a5cd87335aaa91c20a9b822

[8] https://www.cygwin.com (retrieved on December 5, 2021)

[9] http://mingw-w64.org (retrieved on December 5, 2021)

[10] The code for models are available at MetuMalwareGroup Repository, https://github.com/MetuMalwareGroup/static-analysis

Short-Term Memory Model (Stacked BiLSTM) and two Generative Pre-trained Transformer 2 (GPT-2) models.

1) Stacked Bidirectional Long Short-Term Memory Model (Stacked BiLSTM)

We built an initial classifier by stacking the layers sequentially as follows. The embedding layer is the first layer, which takes the integer-encoded opcode sequences and looks up an embedding vector for each word index. To create these vectors from the labeled dataset, first we employed a custom standardization method in the preprocessing phase with padding to maximum number of tokens.[11]

These vectors add a dimension to the output array, and during the training, the model learns the features represented by the vectors. The resulting dimensions of the first layer are batch, sequence, and embedding. Since we wanted our model to learn features related to malicious and benign executables in the sequential form, we used BiLSTM as the next layer. After the BiLSTM layer, we added a Dropout Layer with a dropout rate for starting point 0.1. Next, we added a GlobalAveragePooling layer to return a fixed-length output vector for each example by averaging the sequence dimension. Then we deployed a fully connected (Dense) layer with 128 hidden units, with this fixed-length output vector.

The last layer is a densely connected layer with a single output node. A deep learning model needs a loss function and an optimizer to calculate the weights during the training. As we focus on classifying samples in two categories and the output of our model is a probability (a single-unit layer with a sigmoid activation), we used the BinaryCrossentropy loss function. Lastly, we configured the model to use an optimizer. We preferred Adaptive Moment Estimation (Adam) [30] optimizer. The resulting model was our initial Stacked BiLSTM based model. We further improved the model by hyperparameter tuning. We developed two models, namely the Document Level Analysis Model (DLAM) and Sentence Level Analysis Model (SLAM) by using this methodology, as presented below.

a: The Document Level Analysis Model (DLAM)

We added one-layer BiLSTM at a time and observed the performance of the model until it reached the optimum point. Other than the BiLSTM layer depth, there were other hyperparameters such as batch size, number of epochs, filter size, optimization algorithm, dropout rate etc. To decide the values for those parameters, we experimented with several different values (Table 3). We employed EarlyStopping[12] from Keras with a patience value 3 to fine-tune those hyperparameters in the language modeling task.[13]

TABLE 3: Parameters for the DLAM

Parameter	Val1	Val2	Val3	Val4
Embedding Dimensions	64	128	256	1024
Max Sentence Length	50K	100K	150K	200K
Sequence Length	256	512	1024	-
Dropout Rate	0.01	0.1	0.2	0.5
Optimizer	Adam	RMSprop	Adagrad	SGD
LSTM Hidden Cells No.	64	128	256	512
BiLSTM Layers No.	1	2	3	-
Max Features	1000	-	-	-

To analyze the effect of the number of stacked BiLSTM Layers, we first fixed the hidden layer cells to 64 and the maximum sentence length to 100K, using 4-grams. We increased the number of BiLSTM layers starting with a single layer. We observed that the prediction loss decreased when the number of LSTM layers was increased. Table 4 shows the evaluation loss values and the total parameters of the DLAM. Increasing the number of stacked BiLSTM layers makes the DLAM deeper; hence the model learns more features from the dataset, thus starts to decrease loss but the training time increases. Also, a high number of BiLSTM layers may increase the likelihood of overfitting. Therefore, we used two stacked BiLSTM layers with the parameter values shown in Table 4.

TABLE 4: BiLSTM layers validation loss values

BiLSTM Layers	Loss	Trainable Parameters
1	0.04	437249
2	0.03	536065
3	0.02	634881

By conducting further experiments, we finally employed two stacked BiLSTM layers, with 64 hidden cells both, and we decreased the number of epochs to seven for each parameter. To observe the impact of the maximum sentence length, we trained DLAM with the sentence lengths 10K, 25K, 50K, and 100K, again keeping the other parameters constant. With 10K, 25K, and 50K, we present the validation losses in Table 5.

TABLE 5: The effects of sentence length on validation loss

Sentence Length	Loss	Accuracy
10K	0.05	0.97
25K	0.04	0.98
50K	0.03	0.98
100K	0.02	0.99

We found that the sentences with a length of 100K was enough to represent the full document. A longer sequence length means a longer training time and a higher likelihood of overfitting. Therefore, we did not test sentence lengths higher than 100K, and fixed this value to 100K in the DLAM.

We tested the impact of regularization and normalization; we trained the model with dropout rates 0.01, 0.1, 0.2, and

[11] The code for standardization is available at https://github.com/MetuMalwareGroup/static-analysis/blob/main/custom_standardization.py

[12] https://keras.io/api/callbacks/early_stopping/ (retrieved on December 5, 2021)

[13] https://github.com/keras-team/keras/blob/master/keras/callbacks.py#L1713 (retrieved on December 5, 2021)

0.5 to make our training noisy. We obtained the optimum dropout rate for our data by keeping other parameters fixed. Since Keras supports Variational RNNs, we used 0.5 for `recurrent_dropout` parameter for each BiLSTM layers with a dropout rate of 0.2. After we added a dropout rate of 0.2 together with Variational RNN technique in the BiLSTM layers, we removed the initially added dropout layers following each BiLSTM layer.

We also obtained training losses higher than the validation losses. Although we expected otherwise, according to the Keras documentation[14], it may be observed with the usage of Regularization Mechanisms, such as Dropout and L1/L2 weight regularization, since those mechanisms are turned off during testing period. We checked our initial loss before applying regularization methods as suggested in [31]. Since our training and validation sets are divided as 80:20, for binary classification problem we expect the value as the result of the calculation below.

$$-0.2ln(0.5) - 0.8ln(0.5) \approx 0.7$$

The initial loss of the DLAM model outputs was 0.65, being very close to the values at hand. This implies that the DLAM started the learning process randomly, as expected.

We have tested the optimizers to observe their impact on performance. In particular, we tested five commonly used optimizers, namely SGD, SGD with momentum, RMSProp, Adagrad, and Adam. SGD is an optimizer that allows updating gradient with a small batch consisting of one sample instead of the typical gradient descent using the whole dataset. It selects one sample randomly from a shuffled set. SGD with momentum improves the stochastic gradient descent by reducing the oscillations with the help of momentum, which allows performing larger updates through the desired direction. RMSProp is an adaptive optimizer that uses the magnitude of recent gradients to normalize the gradients and updates the learning rates adaptively. Adagrad manages multiple learning rates for different parameters and updates the learning rates according to the frequency of parameter updates. It performs smaller updates for the parameters frequently occurring while performing larger updates for the infrequent parameters. Adam is a commonly used optimizer that utilizes the adaptive learning rates for each parameter as RMSProp. In addition, Adam has a mechanism similar to momentum, which decays past gradients exponentially. As a result of our experiments with optimizers, we found that Adam and RMSProp achieve a similar result with our models. We preferred to use Adam since it showed slightly better performance than RMSprop.

We observed the increase in n-grams affects in a positive way, such that the loss decreases and accuracy increases as shown in Table 6. For the pooling layer, the two alternatives were the Global Average Pooling strategy and the Global Max Pooling strategy. We observed that the Global Max Pooling outperformed the Global Average Pooling, therefore we chose the latter. Moreover, we decided the number of output nodes at the LSTM layer by testing the model with different numbers of nodes and comparing the loss and accuracy rates of their results. The model loss was higher and the convergence was early in terms of epoch numbers with 256 output nodes. Since it did not result in a significant loss compared to 128 nodes, we chose 128 as the output nodes of the DLAM layer for faster training.

TABLE 6: The effects of n-grams on validation loss

n-grams	Loss	Accuracy
2	0.18	0.97
3	0.12	0.98
4	0.03	0.99

We also used different output modes provided by TextVectorization in Keras, such as TF-IDF. However, with the number of 1000 maximum features, TF-IDF performed poorly, and we increased the maximum features up to 10K. Increasing the maximum features significantly increased the training time and trainable parameters close to 16M. Since 4-grams are more effective in training time, we preferred n-grams. Finally, we used the change of losses as a sign for the parameters to be the most suitable for the DLAM and prepared it accordingly. The summary of the proposed model parameters are shown in Table 7.

TABLE 7: Model summary for the proposed DLAM.

Layer (type)	Output Shape	Params
Embedding	(None, None, 256)	256256
Bidirectional	(None, None, 256)	394240
Bidirectional	(None, None, 256)	394240
Global Max Pooling	(None, 256)	0
Dropout	(None, 256)	0
Dense	(None, 128)	32896
Dropout	(None, 128)	0
Dense	(None, 1)	129
	Total params : 1,077,761	
	Trainable params : 1,077,761	
	Non-trainable params : 0	

b: The Sentence Level Analysis Model (SLAM)

This section presents the experiments conducted for the Sentence Level Analysis Model (SLAM) with the same dataset used in the DLAM. Since we treat each assembly instruction as a sentence, it is necessary to calculate the sentence length distribution in the dataset. Using sentence length distribution as in Figure 2, we selected the group that represented 94% of the dataset by excluding the samples that have a percentage below 1%. By manual observation, we decided to use 9, 16, and 25 for the length of the maximum sentence.

We experimented with the values shown in Table 8, and the maximum feature count of 1122 (unique word count). We started with two BiLSTM layers.

Table 9 shows a summary of SLAM created by Keras with a total of 834,177 trainable parameters.

[14] https://cutt.ly/hYlrHG1 (retrieved on February 19, 2022)

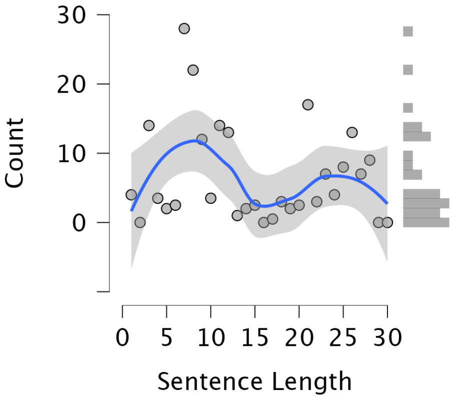


FIGURE 2: Sentence length distribution for the SLAM

TABLE 8: Parameters for the SLAM

Parameter	Val1	Val2	Val3	Val4
Embedding Dimensions	128	256	-	-
Max Sequence Length	8	16	32	-
Sequence Length	16	-	-	-
Dropout Rate	0.01	0.1	0.2	0.5
Optimizer	Adam	RMSprop	Adagrad	SGD
LSTM Output Node No.	128	256	512	1024
BiLSTM Layers No.	1	2	3	-
Max Features	1122	-	-	-

The losses are shown in Table 10 obtained after training the SLAM with the sentence lengths of 9, 16 and 25.

We trained the model on several known optimizers with their default configurations, such as Adam, RMSProp, Adagrad, Stochastic Gradient Descent (SGD), and SGD with momentum, keeping the other parameters constant. We found that Adam and RMSprop achieved similar results.

TABLE 9: Model summary for the initial SLAM

Layer (type)	Output Shape	Param
Embedding	(None, None, 128)	143744
Bidirectional	(None, None, 256)	263168
Bidirectional	(None, None, 256)	394240
Global Average Pooling	(None, 256)	0
Dropout	(None, 256)	0
Dense	(None, 128)	32896
Dense	(None, 1)	129
Total params : 834,177		
Trainable params : 834,177		
Non-trainable params : 0		

TABLE 10: The effects of maximum sentence lengths on validation loss

Sentence Lengths	Losses
9	0.67
16	0.66
25	0.65

As for the impact of the Embedding Dimension Length, the SLAM did not exhibit a significant improvement in terms of loss, with the values 128 and 256, as shown in Table 11.

TABLE 11: The effects of embedding dimensions on validation loss

Embedding Dimensions	Losses
128	0.65
256	0.66

The validation losses regarding the n-gram values for 2, 3, and 4 are shown in Table 12.

TABLE 12: The best results for TextVectorization parameters on the SLAM

n-grams	Loss	Accuracy %
2	0.65	56.1
3	0.67	54.1
4	0.69	53.7

A major finding indicated by the outputs above is that the model's hyperparameters do not significantly affect the validation loss, indicating that the sentence-level representation of assembly instructions with the TextVectorization class may not be sufficient for this task. Therefore, we employed different methods for the vectorization layer. We examined the effectiveness of methods in Word2Vec and DistilBERT considering the assembly lengths. We employed CBOW and Skip-Gram implementations with experiments related to Word2Vec, to create the vector representations of assembly instructions. We fixed the value for min_count parameter as ten to ignore the assembly instructions that do not occur more than ten times. We used different parameters for windows (looking back and forward for the number of window size words) and size. Using the values in Table 13, we trained Word2Vec for eight times for 5 epochs. We fed the embedding layer in the SLAM with the resulting vectors. Since Word2Vec creates and trains the word vectors, we set the trainable parameter of the embedding layer to false.

TABLE 13: Word2Vec parameters for the SLAM

Parameters	Value 1	Value 2
Algorithm	CBOW	Skip Gram
Window Size	10	25
Size	100	300

After we fed the embedding layer of the SLAM with Word2Vec weights, we obtained the deep neural network model summarized in Table 14 by using the values in the first column of Table 13.

Table 14 indicates that the network has more than 1M (million) parameters, although 866K parameters are trainable. That is due to the lack of training in the embedding

TABLE 14: Model summary for the initial SLAM with Word2Vec.

Layer (type)	Output Shape	Params
Embedding	(None, 10, 300)	144900
Bidirectional	(None, 10, 256)	439296
Bidirectional	(None, 256)	394240
Dropout	(None, 256)	0
Dense	(None, 128)	32896
Dropout	(None, 128)	0
Dense	(None, 2)	258
Total params : 1,011,590		
Trainable params : 866,690		
Non-trainable params : 144,900		

layer. We observed that with the CBOW implementation of Word2Vec and a window size of 25, the loss decreased to the lowest value and the accuracy increased to the highest. On the other hand, the training time doubled. We presented Table 15 below, which shows the effectiveness of the algorithms regarding validation losses and validation accuracy values.

TABLE 15: Effects of Word2Vec parameters on the SLAM

Algorithm	Window Size	size	Loss	Accuracy (%)
Cbow	25	300	0.54	63.7
Skipgram	10	300	0.55	61.8

As the third vectorization method, we employed transformers, using DistilBERT on the same dataset. We used a pre-trained base model, `distilbert-base-uncased`, to create the `embeddings` for the SLAM. Since DistilBERT outputs a tuple[15], where the first element is the `last_hidden_state` of the model's last layer, we first fed the BiLSTM layer using the hidden state from outputs. The model with DistilBERT embeddings is shown in Table 16.

We used 5,000 for the batch size value and 25 for the maximum length. It took about six hours to train for one epoch with two stacked BiLSTM layers on Google Collab Pro. With DistilBERT, the total number of parameters is 67,708,673. Since we did not train the DistilBERT layers for the first experiment, the total number of trainable parameters is 1,345,793, as shown in Table 17. After six epochs, the model achieved 70.4% accuracy.

After developing the DistilBERT model, which achieved 70.4% accuracy with short text as a sentence, we conducted further experiments on different models to enhance accuracy with short text. For this, we constructed a custom pre-trained model and two binary classification models based on GPT-2, as presented in the following section.

TABLE 16: Model summary for the SLAM with BERT.

Layer (type)	Output Shape	Params
input_ids(InputLayer)	[(None, 25)]	0
input_attention(InputLayer)	[(None, 25)]	0
TFDistilBERT	((None, 25, 768)	66362880
Bidirectional	(None, 25, 256)	918528
Bidirectional	(None, 25, 256)	394240
Dropout	(None, 25, 256)	0
Global Max Pooling	(None, 256)	0
Dropout	(None, 256)	0
Dense	(None, 128)	32896
Dropout	(None, 128)	0
Dense	(None, 1)	129
Total params : 67,708,673		
Trainable params : 1,345,793		
Non-trainable params : 66,362,880		

TABLE 17: Effects of DistilBERT parameters on the SLAM

Trainable Parameters	learning rate	Loss	Accuracy %
1,345,793	5e-5	0.49	68.4
67,708,673	2e-5	0.43	70.4

2) Generative Pre-trained Transformer 2 (GPT-2) Models

, we built custom pre-trained model based on GPT-2 architecture. Then, we improved the GPT-2 Domain Specific Language Model (GPT2-DSLM) using our custom pre-trained model's outputs. We later developed GPT-2 General Language Model (GPT2-GLM) with GPT-2's pre-trained model's outputs.

a: The Pre-trained Model

We used `run_language_modeling.py`[16] to create a custom pre-trained model. Firstly, we needed to define a merging rule of language (`merges.txt`) and a language dictionary (`vocab.json`), both obtained from assembly instructions. For creating those files, we used ByteLevelBPETokenizer[17] which is a byte-level encoding tokenizer. We customized the parameters for ByteLevelBPETokenizer, using vocabulary size (dictionary size) as 50257 , minimum frequency of the tokens as 2, sentences (assembly instructions) and special tokens[18]. Since models learn information about short text via special tokens, we used end-of-sentence (<eos>) to make the model stop generating more words. After feeding with unlabeled assembly instructions corpora to the ByteLevelBPETokenizer, the tokenizer builds the `vocab.json` file that consists of all symbols obtained from the set of unique words, and the `merges.txt` file that consists of a list of the most frequent tokens ranked by frequency. Thus, we defined a tokenizer of our custom pre-trained model with `vocab.json` and `merges.txt`.

For the tokenizer of the pre-trained model, we used

[15] https://github.com/huggingface/transformers/blob/master/src/transformers/modeling_outputs.py#L24 (retrieved on December 6, 2021)

[16] https://github.com/huggingface/transformers (retrieved on June 25, 2021)

[17] https://huggingface.co/transformers/tokenizer_summary.html (retrieved on June 25, 2021)

[18] https://huggingface.co/transformers/main_classes/tokenizer.html (retrieved on June 25, 2021)

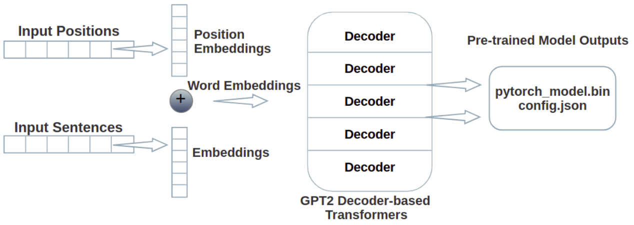


FIGURE 3: Pre-trained model based on GPT-2 architecture

GPT2TokenizerFast[19] class of the HuggingFace's Transformers module due to its performance advantage. Those tokenizers are also used for determining maximum sequence lengths. We calculated the maximum sequence length of assembly instructions as 45 tokens using GPT2TokenizerFast. Next, we defined a configuration file with default values following [32] and [33]. Those values are 50257 for vocabulary size (suggested by [32]), 12 for number of hidden layers and number of attention heads for each attention layer, 768 for dimensionality of the embeddings and hidden states, `gelu` instead of `relu` for the activation function. The dropout probability for all fully connected layers, the dropout ratio for the embeddings, and the dropout ratio for the attention were set to 0.1 following the common practice reported by [33]). In the normalization layers, the epsilon was set to 1e-05. The total number of hyperparameters becomes 117 million with these default values of GPT-2. While we fine-tuned the binary classification model, we also used same configuration file of our custom pre-trained model.

After defining the configuration file with the default configuration of GPT-2 using GPT2Config class of the HuggingFace's Transformers module, we initialized the model with GPT2LMHeadModel model found in HuggingFace's Transformers module. Finally, as shown in Figure 3, we obtained an auto-regressive model based on a left-to-right language model for the trainer. In addition, we set the mask language model (mlm) as false (unmasked model) to preserve the relation between the following and preceding words of each word in the input sentence. Hence, the `pytorch_model.bin` file, which includes model's weights, and the `config.json` file, which includes configuration hyperparameters created to fine-tune the binary classification model, presented below.

GPT-2 Domain Specific Language Model (GPT2-DSLM)

We constructed GPT-2 based model for the classification of benign and malicious assembly instructions, using HuggingFace's Transformers module, namely GPT-2 Domain Specific Language Model (GPT2-DSLM). To improve the classification phase, we used our custom pre-trained model's knowledge, which was specified by the Pre-trained Model's outputs, in other words `pytorch_model.bin` and `config.json`. We present the processing pipeline below.

Firstly, we divided our dataset as training, validation, and testing sets by a ratio of 60:20:20, for the binary classification problem. Next, we set the label number field as 2 for classifying the files as benign or malicious. Our custom pre-trained model's `config.json` file was defined as the configuration file of the GPT2-DSLM. We also defined `vocab.json` and `merges.txt` files as the GPT2-DSLM's tokenizer. After creating the tokenizer, we introduced the special tokens of GPT-2 to the tokenizer. Since the last token of the input sequence contained all the information required in the prediction in GPT-2, we set the tokenizer to pad the left side of sentences, and its pad token was |<endoftext>|. Hence, we used that information for the classification task. Then, in the tokenizer step, we first tokenized each sentence of the benign and malicious and added |<endoftext>| since training times depended on the length of sentences. While the length of the sentences was shorter than the maximum sequence length of 45, the tokenizer padded sequences with |<endoftext>| and encoded them. The processing then took shorter since the model did not have to truncate the encoded sequence during training.

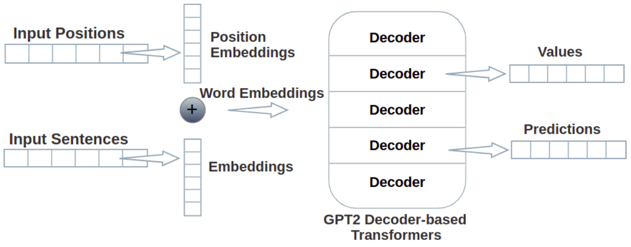
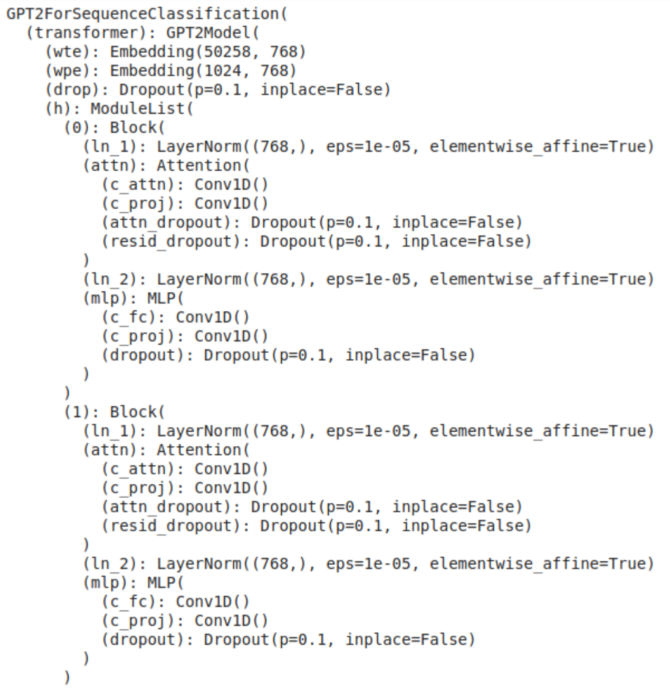


FIGURE 4: GPT2-DSLM and GPT2-GLM based on GPT-2 architecture

We further developed the constructed GPT2-DSLM, which was shown in Figure 4, by adding a classification layer to the GPT2Model, known as GPT2ForSequenceClassification model from HuggingFace's Transformers module. The new model is shown in Figure 5. The model was initialized with the `pytorch_model.bin`, which loads the weights associated with our custom pre-trained model. Next, we trained the model with word embedding and positional embedding information obtained from the tokenizer. The classification layer was a densely connected layer with a single output node. To calculate the weights during training, a deep learning model needs a loss function and an optimizer. GPT2ForSequenceClassification aims at classifying samples into two categories, and the model's output is probabilistic (a single-unit layer with a sigmoid activation). Therefore, GPT2ForSequenceClassification uses the Binary Cross Entropy loss function. Lastly, we configured the model to use an optimizer with Adamw implementing Adam [30] algorithm with weight decay [34] fix in Pytorch. The weight decay prevents overfitting, which keeps the weights as small as possible, also preventing the weights from growing out of control. Thus, the network avoids exploding gradients.

[19] https://huggingface.co/transformers/model_doc/gpt2.html (retrieved on June 25, 2021)



```
GPT2ForSequenceClassification(
  (transformer): GPT2Model(
    (wte): Embedding(50258, 768)
    (wpe): Embedding(1024, 768)
    (drop): Dropout(p=0.1, inplace=False)
    (h): ModuleList(
      (0): Block(
        (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (attn): Attention(
          (c_attn): Conv1D()
          (c_proj): Conv1D()
          (attn_dropout): Dropout(p=0.1, inplace=False)
          (resid_dropout): Dropout(p=0.1, inplace=False)
        )
        (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): MLP(
          (c_fc): Conv1D()
          (c_proj): Conv1D()
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
      (1): Block(
        (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (attn): Attention(
          (c_attn): Conv1D()
          (c_proj): Conv1D()
          (attn_dropout): Dropout(p=0.1, inplace=False)
          (resid_dropout): Dropout(p=0.1, inplace=False)
        )
        (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): MLP(
          (c_fc): Conv1D()
          (c_proj): Conv1D()
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
```

FIGURE 5: Two layers of GPT2ForSequenceClassification

In order to find the best values for the parameters' learning rate for Adamw and epoch in the language model task, we tested several different values. So, we trained 2, 3, and 4 epochs. With 3 epochs, the model found better accuracy and lower loss value compared to 2 epochs. Also, accuracy and loss results were almost the same for three and four epochs using learning rate value 2e-5, as shown in Table 18.

TABLE 18: The effects of epochs on validation losses

Epochs	Loss Values	Accuracy (%)
1.	0.40	0.84
2.	0.39	0.85
3.	0.38	0.85
4.	0.37	0.85

We selected the number of epochs as 3 due to it's shorter training time. We also set the learning rate to 2e-5, since a lower learning rate value, such as 3e-5 or 5e-5 led to a lower network accuracy and higher loss. as shown in Table 19.

TABLE 19: The effects of learning rate on validation loss

Learning Rate	Loss Values	Accuracy (%)
5e-5	0.42	0.80
3e-5	0.40	0.83
2e-5	0.37	0.85

After the experiments, the GPT2-DSLM achieved 85.4% accuracy. In addition, we also constructed another GPT-2 based model for the classification of benign and malicious assembly instructions, using HuggingFace's Transformers module, namely GPT-2 General Language Model (GPT2-GLM). The GPT2-DSLM has the same architecture as the GPT2-GLM. However, to improve the classification phase in the GPT2-GLM model, we used GPT-2's architecture custom pre-trained model, namely "gpt-2". After the experiments, the GPT2-GLM achieved 78.3% accuracy.

IV. COMPARISON OF THE MODELS

The precision, recall and F1 values obtained by our models are shown in Table 20. Overall, the findings show that the document level analysis model outperforms the sentence level analysis models. A likely reason of the difference in the performance may be due to the shorter text form (at most nine assembly instructions) in the GPT2-DSLM, which may have led to misrepresentation of the necessary features. Since we labeled each assembly instruction as malicious or benign depending on the source, we also labeled the shared instructions between the classes, which in turn may have resulted in models' lower performance. When the F1 score is considered, we may infer that it includes meaningful information and patterns to achieve a 86.0% F1 score. On the other hand, the documents of assembly instructions which consist of more instructions, are contextually related. In addition to the relation of words in an assembly instruction, there are also relations among the instructions documents-wise. Thus, the documents with longer and more complex structures include more meaningful information and more relations than shorter instructions, resulting in the DLAM achieving a 98.3% F1 score. In summary, the results of the final experiments on the two models suggest that the Document Level Assembly Analysis Model (DLAM) exhibits a better structure for Stacked BiLSTM based deep learning language modeling compared to other models.

TABLE 20: Model performance indicators

Model	Precision (%)	Recall (%)	F1 (%)	TPR/FPR/FNR
GPT2-GLM	82.7	70.8	76.2	0.71 / 0.01 / 0.29
GPT2-DSLM	82.6	89.7	86.0	0.90 / 0.19 / 0.10
DLAM	97.8	98.9	98.3	0.99 / 0.02 / 0.01

The most crucial factor that led to the differences between the two proposed models in the sentence level analysis is the pre-trained models. Compared to the GPT2-GLM, the GPT2-DSLM achieves an 86.0% F1 score. Nevertheless, the first architecture also achieves a 76.2% F1 score. The performance difference between the two architectures is likely to be the underlying language models. On first architecture the language model is based on Web Text [32], on the other hand our model is based on assembly instructions. Since we created domain specific language model with GPT-2, one may propose that this approach seems more efficient than GPT-2's general model on detecting malware.

V. THE EVALUATION OF RESULTS

Malware detection studies began with the signature extraction methods applied to executables. Nevertheless, the signature databases, distributed over the internet to the end-users online or offline, grew in time. Moreover, malware developers used innovative approaches to bypass anti-malware applications. Recently, machine learning (ML) models have become prevalent due to their capacity for adaptability, such as recognizing polymorphic malware. Early AI-based methods employed machine learning (ML) algorithms to identify malicious and benign malware and classify malware families. A set of classical classification algorithms, such as Random Forest (RF), Support Vector Machine (SVM), and Decision Tree (DT), have been employed in previous studies. They were applied to the data obtained from malicious or benign files. However, those algorithms had limitations, such as the requirements of high processing power, domain knowledge, and specific feature extraction methods in deployment. Deep learning (DL) solutions have addressed some of those problems. Therefore, the focus of malware detection solutions shifted from classical ML to DL. Recently, deep neural network architectures have been widely explored to identify malicious and benign software.

Deep learning is not a homogeneous approach, and the number of architectures and topologies is wide and varied. Therefore, researchers have proposed various architectures and methods such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and attention mechanisms. Numerous studies focus on assembly code and detect malicious code pieces by employing deep learning techniques. They mainly use RNN, LSTM, CNN, or transformers-based models for binary classification and opcode, operand sequence as binary forms or word forms. For instance, [22] employed the LSTM network to distinguish executables between malicious or benign with opcode sequences as words. Another example is [35], which used opcode and byte-code sequences with LSTM based neural networks to detect malware in IoT systems. On the other hand, [18] and [17] used binary form of the executables to create the image representations and applied CNN-based architectures to their dataset. Another study is [26], which used benign and malicious assembly code obtained from static contents of executables. Their model, Interpretable MAlware Detector (I-MAD), combined a network component called the Galaxy Transformer network that understood assembly code at the basic block (a sequence of assembly instructions), assembly function (a set of basic blocks), and executable levels (a sequence of bytes). In another work [27] employed a transformer-based model "Bidirectional Encoder Representations from Transformers (BERT)" for malware detection. The work focused on the static analysis level on the source code of benign and malicious applications on Android to detect different categories of malware.

In the present study, we aimed to conduct malware detection by classifying sentence-level and document-level assembly sentences as benign and malicious. Accordingly, the accuracy rates of our proposed methods in the SLAM address the presence or absence of assembly instruction sequences rather than a binary classification of a file or a classification of byte-code sequences. Bearing in mind that a proper comparison of the accuracy values with the previous work is challenging, below we discuss the findings in the literature in comparison to the findings obtained in the present study. Given that BERT has been commonly used in the literature, we also created a binary classification model based on BERT, BERT General Language Model (BERT-GLM). We fine-tuned it with the pre-trained model `bert-base-uncase` to improve the evaluation of the present study in comparison with the previous work on malware detection. Table 21 presents the architectures used in the previous studies and the accuracy values achieved by them. Since the common metric that judges the performance of all studies models is accuracy, we included the accuracy rate of those models in the table.

TABLE 21: Comparison of malware detection models

Study	Architecture	Accuracy (%)
[18]	CNN[20]	98.0
[17]	ResNet[20]	88.4
[17]	GoogleNet[20]	74.5
[22]	LSTM[21]	97.3
[35]	LSTM[22]	99.1
The SLAM	BiLSTM[21]	70.4
The DLAM	BiLSTM[21]	98.3
The BERT-GLM	BERT[21]	77.6
The GPT2-GLM	GPT-2[21]	78.3
The GPT2-DSLM	GPT-2[21]	85.4

The findings obtained in the present study show that the DLAM BiLSTM model has consistent performance with the state-of-the-art static malware detection models. A further evaluation of the models reveals the following findings. Kumar et al. [18] aims to detect novel types of malware using CNN's Image Similarity technique. The model was trained by the Vision Research Lab dataset, which contains 9458 gray-scale images of malware samples that come from 25 different malware families. The second dataset contained 3000 benign software of various types. Those datasets are converted into binary code and then converted into image files. The model achieved 98.0% accuracy. Another study, conducted by Khan et al. [17] aimed to identify unknown or novel malware. They utilized two CNN models, namely ResNet from Microsoft In. and GoogleNet from Google Inc. The datasets were obtained from various sources. Specifically, the malware dataset was obtained from Microsoft Inc., and 3000 benign files were obtained from open source websites. After binary unpacking the executable files, the researchers converter them into opcodes with PEID and then to images. They obtained a testing accuracy of 74.5% on

[20] Binary to Image
[21] Sequential ASM
[22] Sequential ASM, Syscall

GoogleNet and 88.36% precision on ResNet with images. Another study, which reported performance results of static malware analysis [22] proposed a model learning the opcode sequence patterns from malware, thus modeling malware as language. They used word embedding techniques cbow and skip-gram, proposing a two-stage LSTM model for malware detection. They performed experiments on the dataset provided by Microsoft Inc., MalwareDB, and Virusshare that included 969 malware and 123 benign files. Their binary classification model achieved 97.87% accuracy with cbow and its window size 10. Finally, [35] proposed a methodology that can be applied in real-time malware threat detection using a deep recurrent neural network solution as a stacked long short-term memory (LSTM) with a pre-training as a regularization method. They also evaluated the effectiveness of their proposed method on Windows, Ransomware, Internet of Things (IoT), and Android malware datasets. For the IoT malware detection with static analysis on opcode sequences. The proposed method achieved 99.1% accuracy. Those studies, presented in Table 21 show the best results obtained by recently available models, designed and implemented in different ways that select and extract features, and employed various machine learning models on various datasets. Therefore, the confound variables may differ, depending on model characteristics. In the present study, we proposed models that accomplish promising results with low processing requirements, even without extracting system calls in data preprocessing.

VI. OPEN PROBLEMS

This section presents certain aspects of our study, which may be considered open problems in the malware detection domain. First, neural network architectures allow researchers to create mainly black-box models due to the virtually incomprehensible internal logic of the hidden layers. So, as in the other studies that employ deep neural networks, being a black box at the model level is a limitation of the present study. Although searching for better hyperparameters is possible, there is no principal method to accomplish the optimal set. Therefore, future research should focus on explainable models (viz. XAI) and novel methods of reaching optimal hyperparameters. Another limitation is the variance in the datasets. As in many other previous works, our dataset consisted of a few hundred malware and benign files. Future research should consider using extensive malware databases. Recently, malware detection studies do not have common benchmark datasets. The researchers establish datasets for the purpose of the study, as done in the present study, also in the others, e.g., [10], [28]. The lack of common datasets limits the generalizability of the findings in detection performance. Consequently, specific problems, such as collecting and making them accessible through public datasets, have to be resolved [36].

Our experiments aimed to ensure that our models produce reliable and comparable results by checking the division of the data into training, test, and validation sets (without data leakage among them) and using the data sets obtained from different sources. Specifically, the common practice is to use random splitting by publicly available libraries such as scikit-learn and the Keras. In the DLAM model of the present study, we used the method provided by the Keras library (tf.keras.preprocessing.text_dataset_from_directory) that employs the method np.random.RandomState(seed). It is important to use the same seed value when separating the dataset into training, test, and validation sets since the generated random value[23] has to be the same all the time, according to the numpy documentation.[24] Otherwise, updating the random number in each run would influence model consistency. Running the model with different seed values allows processing different samples in the dataset as training and validation sets. In the present study, we experimented with five different seed values (24, 49, 63, 75, 82) and inspected its effect on the accuracy of the DLAM (Table 22). The results reveal the seed value as a factor that influences model performance, among other model parameters. Accordingly, the method of dividing the dataset into training, test, and validation may have an impact on model performance. Therefore, researchers should not only focus on the best performance obtained from a model but also on the parameters that contextualize the model performance. Table 22 also presents alternative methods for calculating the performance of the DLAM, such as the mean accuracy values. A mean accuracy value can be calculated by summing up the accuracy values and dividing them by the number of runs. Another method of evaluating model performance is to calculate False Positive Rates in different runs (FPR = FP/FP+TN, where FP is the number of false positives and TN is the number of true negatives). The FPR shows the probability of a false alarm, i.e., a benign file detected as malware. In summary, an open problem in model performance evaluation is the abundance of dependent variables that indicate model performance,[25] resulting in divergent performance reports in the literature. Although we reported F1 scores in the present study, the choice of performance evaluation parameters, such as AUC [35], F1 scores [37], or accuracy rates [18], [22] may depend on the choice of the researchers. A consistent presentation of performance evaluation is needed for efficient, comparative analysis of model findings in the literature.

VII. CONCLUSION

In the present study, we proposed generative approaches to classify malicious and benign software. We focused on assembly language since the assembly code provides accurate information about critical coding patterns. We implemented our strategy on static analysis of the data. We focused on

[23] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/keras/preprocessing/dataset_utils.py#L123 (retrieved on September 5, 2021)

[24] https://numpy.org/doc/stable/reference/random/bit_generators/pcg64.html (retrieved on September 5, 2021)

[25] Binary Classification Metrics: https://neptune.ai/blog/evaluation-metrics-binary-classification (retrieved on September 5, 2021)

TABLE 22: The effects of the seed value on performance in four runs

Seed Val.	Acc. (%)	FP/FN/TP/TN	F1 Score(%) (%)	FPR
24	98.3	3 / 0 / 87 / 92	98.3	0,03
49	99.4	0 / 1 / 89 / 92	99.5	0
63	99.4	1 / 0 / 90 / 91	99.5	0,01
75	98.9	0 / 2 / 88 / 92	98.9	0
82	98.9	0 / 2 / 88 / 92	98.9	0

opcodes and operands, instead of opcodes only, to develop stacked bidirectional long short-term memory (BiLSTM) models and the decoder-based transformers generative pre-trained transformers 2 (GPT-2) models. Overall, our models revealed efficient solutions that address a set of challenges, such as processing and memory requirements of the deep learning models that process long-term dependencies. The first architecture is based on the stacked (BiLSTM) model with a specific cell structure for solving memory size and time challenges. We found that incorporating techniques from natural language processing (NLP), specifically document-level analysis with word embedding and bidirectional LSTMs (BiLSTM), significantly improves model performance. We also found that we could obtain even better performance by including a Variational RNN technique in our model. The DLAM model detected files with an average accuracy of over 98%, showing that the context obtained from assembly instructions in DLAM is the key to achieving good performance.

The Binary Classification Model, the GPT2-DSLM, based on decoder-based transformers GPT-2, mainly processes short text on multiple head architectures, each with a single self-attention mechanism. The multi-head attentions provide global dependencies between inputs and outputs. First, we built the model to grab the full semantics behind the assembly code with a pre-trained GPT-2 model. We then modeled the GPT2-DSLM with the transformers-based model GPT-2. Furthermore, we fine-tuned the GPT2-DSLM model with the pre-trained model to improve the detection performance. Finally, we selected optimum parameter values based on the experimental results. The resulting accuracy rate 85.4% shows that it is possible to classify malicious and benign assembly codes by GPT-2 with a custom pre-trained model. The future research should address improvements in the data processing pipeline, developing an API service for extracting assembly instructions to collect them automatically for a given PE file to reduce human intervention, and in the coverage of program files, including packed programs, besides the limitations reported in the previous section, as open problems.

REFERENCES

[1] O. Shimon, “Cyber threat report on 2020 shows increases across all malware types,” Deep Instinct, 2021. [Online]. Available: https://www.deepinstinct.com/2021/02/11/cyber-threat-report-on-2020-shows-triple-digit-increases-across-all-malware-types/

[2] K. Scarfone and M. Souppaya, “Guide to malware incident prevention and handling for desktops and laptops,” vol. 7, no. 4, p. 1. [Online]. Available: http://dx.doi.org/10.6028/NIST.SP.800-83r1

[3] R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, and S. Venkatraman, “Robust intelligent malware detection using deep learning,” IEEE Access, vol. 7, p. 46717–46738, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8681127

[4] C. Acarturk, M. Sirlanci, P. G. Balikcioglu, D. Demirci, N. Sahin, and O. A. Kucuk, “Malicious code detection: Run trace output analysis by lstm,” IEEE Access, vol. 9, pp. 9625–9635, 2021.

[5] U. Pehlivan, N. Baltaci, C. Acartürk, and N. Baykal, “The analysis of feature selection methods and classification algorithms in permission based android malware detection,” p. 1–8, 12 2014. [Online]. Available: https://ieeexplore.ieee.org/document/7013371

[6] H. El Merabet and A. Hajraoui, “A survey of malware detection techniques based on machine learning,” International Journal of Advanced Computer Science and Applications, vol. 10, 01 2019.

[7] D. Bilar, “Opcodes as predictor for malware,” Int. J. Electron. Secur. Digit. Forensics, vol. 1, pp. 156–168, 2007.

[8] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, “Idea: Opcode-sequence-based malware detection,” in Engineering Secure Software and Systems, F. Massacci, D. Wallach, and N. Zannone, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 35–43.

[9] I. Santos, B. Sanz, C. Laorden, F. Brezo, and P. G. Bringas, “Opcode-sequence-based semi-supervised unknown malware detection,” pp. 50–57, 2011.

[10] H. Anderson and P. Roth, “Ember: An open dataset for training static pe malware machine learning models,” 04 2018.

[11] R. Moskovitch, C. Feher, N. Tzachar, E. Berger, M. Gitelman, S. Dolev, and Y. Elovici, “Unknown malcode detection using opcode representation,” pp. 204–215, 2008.

[12] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, “Detecting unknown malicious code by applying classification techniques on opcode patterns,” Security Informatics, vol. 1, no. 1, p. 1, Feb 2012. [Online]. Available: https://doi.org/10.1186/2190-8532-1-1

[13] H. Zhang, X. Xiao, F. Mercaldo, S. Ni, F. Martinelli, and A. K. Sangaiah, “Classification of ransomware families with machine learning based on n-gram of opcodes,” Future Generation Computer Systems, vol. 90, pp. 211–221, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18307325

[14] E. Raff, R. Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. Mclean, and C. Nicholas, “An investigation of byte n-gram features for malware classification,” Journal of Computer Virology and Hacking Techniques, vol. 14, 02 2018.

[15] D. Gibert, C. Mateu, and J. Planes, “The rise of machine learning for detection and classification of malware: Research developments, trends and challenges,” Journal of Network and Computer Applications, vol. 153, p. 102526, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804519303868

[16] M. Krcál, O. Švec, M. Bálek, and O. Jasek, “Deep convolutional malware classifiers can learn from raw executables and labels only,” in ICLR, 2018.

[17] R. U. Khan, X. Zhang, and R. Kumar, “Analysis of resnet and googlenet models for malware detection,” Journal of Computer Virology and Hacking Techniques, vol. 15, no. 1, pp. 29–37, Mar 2019. [Online]. Available: https://doi.org/10.1007/s11416-018-0324-z

[18] R. Kumar, Z. Xiaosong, R. U. Khan, I. Ahad, and J. Kumar, “Malicious code detection based on image processing using deep learning,” p. 81–85, 2018. [Online]. Available: https://doi.org/10.1145/3194452.3194459

[19] A. Nguyen, J. Yosinski, and J. Clune, “Deep neural networks are easily fooled: High confidence predictions for unrecognizable images,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2015, pp. 427–436.

[20] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, “Practical black-box attacks against machine learning,” in Proceedings of the 2017 ACM on Asia conference on computer and communications security, 2017, pp. 506–519.

[21] S. Jha, D. Prashar, H. V. Long, and D. Taniar, “Recurrent neural network for detecting malware,” Computers & Security, p. 102037, 09 2020.

[22] R. Lu, "Malware detection with LSTM using opcode language," CoRR, vol. abs/1906.04593, 2019. [Online]. Available: http://arxiv.org/abs/1906.04593

[23] Y. Lee, H. Kwon, S.-H. Choi, S.-H. Lim, S. H. Baek, and K.-W. Park, "Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with cnn," Applied Sciences, vol. 9, no. 19, 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/19/4086

[24] Y. Sung, S. Jang, Y.-S. Jeong, and J. H. J. J. Park, "Malware classification algorithm using advanced word2vec-based bi-lstm for ground control stations," Computer Communications, vol. 153, p. 342–348, Mar 2020.

[25] Q. Xie, Y. Wang, and Z. Qin, "Malware family classification using lstm with attention," pp. 966–970, 2020.

[26] M. Q. Li, B. C. Fung, P. Charland, and S. H. Ding, "I-mad: Interpretable malware detector using galaxy transformer," Computers & Security, vol. 108, p. 102371, 09 2021. [Online]. Available: https://arxiv.org/pdf/1909.06865.pdf

[27] A. Rahali and M. Akhloufi, "Malbert: Using transformers for cybersecurity and malicious software detection a preprint," 2021. [Online]. Available: https://arxiv.org/pdf/2103.03806.pdf

[28] R. Harang and E. M. Rudd, "Sorel-20m: A large scale benchmark dataset for malicious pe detection," 2020.

[29] F. Ducau, E. Rudd, T. Heppner, A. Long, and K. Berlin, "Automatic malware description via attribute tagging and similarity embedding," 01 2020. [Online]. Available: https://arxiv.org/pdf/1905.06262.pdf

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[31] A. Karpathy. (2019) A recipe for training neural networks. [Online]. Available: http://karpathy.github.io/2019/04/25/recipe/

[32] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[33] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018.

[34] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," arxiv.org, vol. 3, Jan 2019. [Online]. Available: https://arxiv.org/abs/1711.05101

[35] A. N. Jahromi, S. Hashemi, A. Dehghantanha, R. M. Parizi, and K.-K. R. Choo, "An enhanced stacked lstm method with no random initialization for malware threat hunting in safety and time-critical systems," IEEE Transactions on Emerging Topics in Computational Intelligence, vol. 4, no. 5, p. 630–640, Oct 2020.

[36] M. A. Lones, "How to avoid machine learning pitfalls: a guide for academic researchers," CoRR, vol. abs/2108.02497, 2021. [Online]. Available: https://arxiv.org/abs/2108.02497

[37] D. Vasan, M. Alazab, S. Wassan, B. Safaei, and Q. Zheng, "Image-based malware classification using ensemble of cnn architectures (imcec)," Computers & Security, vol. 92, p. 101748, 05 2020.

DENIZ DEMIRCI received the B.S. degree in system engineering from the Turkish Military Academy, Ankara, Turkey, as lieutenant, in 2007 and M.Sc. degree in Cyber Security from the Informatics Institute, Middle East Technical University (METU), Turkey, in 2021. He worked as a Platoon Leader and a Team Commander, from 2007 to 2011. From 2011 to 2016, he worked as a Software Developer. Since 2016, he has been responsible for performing and managing penetration tests, malware analysis, security incident detection and response as a Technical Lead with the Cyber Security Center of Turkish, Gendarmerie General Command. His research interests include software security, image processing, natural language processing, malware analysis, reverse engineering, and machine learning.

NAZENIN ŞAHIN received the B.S. degree from the Department of Mathematics, Hacettepe University, Ankara, Turkey, in 2004. She is currently pursuing the degree with the Cyber Security Graduate Program, Informatics Institute, Middle East Technical University (METU), Turkey. From 2007 to 2012, she worked as a Software Developer, then as a Researcher with the Department of Research and Development, Turkish General Commandership of Gendarmerie, Ankara, from 2012 to 2016. Since 2017, she has been a Cybersecurity Specialist with the Turkish General Commandership of Gendarmerie. Her research interests include variety of topics in informatics, including security software development, malware analysis, reverse engineering, and machine learning in security.

MELIH ŞIRLANCI received the B.S. degree in computer engineering from Ege University in 2017, and M.Sc. degree in Cyber Security from the Informatics Institute, Middle East Technical University (METU), Turkey, in 2021. He is currently pursuing the PhD degree in the Computer Science and Engineering department at The Ohio State University. His research interests include malware analysis, binary analysis, and machine learning.

CENGIZ ACARTURK (Member, IEEE) received the B.Sc. degree in mechanical engineering and the M.Sc. degree in cognitive sciences from the Informatics Institute, Middle East Technical University (METU), Turkey, in 1998 and 2005, respectively, and the Ph.D. degree in computer science from the Center for Intelligent Systems and Robotics (ISR), Department of Informatics, Knowledge and Language Processing Institute (WSV), University of Hamburg, Germany, in 2010. He worked at the Cognitive Science and Cybersecurity Graduate Programs, METU Informatics Institute. He is recently conducting research at Jagiellonian University, Poland. His research interests include interaction between cybersecurity and cybercognition, human-computer interaction, human factors in cybersecurity, natural language processing (NLP). He has been a member of IEEE, ACM and the Cognitive Science Society, as well as local ICT NGOs.