Malware Need "Attention" Too!

Malware continues to pose a deeply evolving challenge to the world of security. There is an ongoing fight between attackers and malware analysts. Traditional malware detection methods require a lot of time and human resources. We turn to machine learning-based solutions for the problem in the hand of identifying malicious programs. In this paper, we analyze the malicious properties present within the API call sequence patterns of the programs. We use API fragments and LSTM based model with attention layers for classification. We present our experimental results on two publicly available datasets. Our method based on API fragments and techniques like attention gives better performance than other works that adopt similar techniques after comparing the experiments.

CCS Concepts: • **Security and privacy** → **Malware and its mitigation**; • **Computing methodologies** → Machine learning algorithms.

Additional Key Words and Phrases: malware analysis, machine learning, Natural Language Processing (NLP)

ACM Reference Format:
. 2021. Malware Need "Attention" Too!. *ACM Trans. Graph.* 1, 1, Article 1 (August 2021), 7 pages. https://doi.org/xyz

1 INTRODUCTION

There has been a lot of research on malware analysis and classification of malware. This topic is of interest among many researchers, and various tools and techniques are developed for it. There already exists a lot of literature on tackling the problem of malware detection and malware classification. However, the malware remains a severe problem to individuals, companies and organisations as attackers continuously use it as a tool to get confidential information or perform attackers on the other machine. Malware analysis can be performed using static or dynamic analysis techniques. Though static analysis techniques are powerful and accurate but the attackers hide the program's main intent through techniques like obfuscation, which leads to failing most of the static analysis techniques. The attackers are becoming clever daily and have even deployed techniques like polymorphism to reorder the codes and create multiple virus variants. This demands for developments of techniques that are less cumbersome and more adaptable to changes in the programs.

Deep Learning has seen a significant rise in almost every field like image processing, audio recognition, language translation and whatnot. Even seemingly unrelated fields like Software Engineering have started deploying these techniques nowadays. Since machine learning has the remarkable ability to facilitate the task of feature extraction from low-level data, many scholars have naturally resorted to machine learning techniques for detecting malware. In studies like [Kosmidis and Kalloniatis 2017] people have used image processing techniques to classify malware. The authors in [Vu 2020] used API call sequences to detect malware. In [Nataraj 2015] uses signal processing techniques and NLP methods to handle the assembly code and building a model using LSTMs. These methods have definitely shown the effectiveness of deep learning in this field, but they still suffered from being unstable and getting easily disturbed.

We chose to study the programs dynamically and hence extract out the behaviour of the program during execution. This paper explores this problem using a machine learning perspective and tries different techniques for the same. Here, we have explored the idea to combine techniques of NLP with Malware analysis.

Our method is used on programs made for the Windows platform by extracting the API execution sequences. To exploit the local malicious properties present in a program, we cut the API sequence into smaller API fragments and worked on them. We generate word embeddings for many API calls in our approach and then use these embeddings to generate sequences (sentences) of these calls. We use the analogy from English vocabulary and practice frameworks like Word2Vec to prepare the required embeddings. Using such a technique allows us to generate semantically valid embeddings for each API call, which naturally helped us in the process employed ahead. We use two layers of LSTMs and two layers of attention in our model. Since an API call can be highly correlated to a previous API call in a different fragment, so we use the attention layers to help us model the relation between API calls present in different fragments. The final embedding represents the program and operates as an input for the machine learning classifiers. The paper also discusses the further classification of the program into the different types of malware classes.

Experimenting on multiple datasets indicated that our technique is stable and produces good results even with only a few thousand samples in hand. The approach used in this paper of combining LSTMs with attention network outperforms the other works that use similar strategies. Our approach is stable towards techniques like obfuscation since we are using API calls at the very heart of our approach 3.2. Furthermore, using the model on a binary for prediction is also relatively easy since it just involves extraction of API sequences which can be automated using analysis systems like Cuckoo Sandbox [Cuc 2020].

Our work contributes as follows:

- Use the analogy of language vocabularies and using Word2Vec like models to generate embeddings that made sense semantically and naturally helped achieve good results.
- Analyse the local malicious properties by converting them into fixed-length API fragments.
- Continuing with the analogy with language, we used NLP models like LSTMs to locally utilise the features/knowledge present in the API sequences.
- Combining the normal LSTMs with attention layers to get the correlations present between calls globally.

Author's address:

- Our paper demonstrates the effectiveness of LSTM models and techniques like attention in malware analysis which can be taken up and explored further for research.

The paper is structured into the following sections further. Section 2 discusses the current literature present for this field, Section 3 explains the proposed model in detail and provides theoretical significance of the method, Section 4 provides experimental evidence to validate our idea and Section 5 concludes the paper.

2 RELATED WORK

It has been quite a few ages since the start of the era of the development of malware. Nowadays, attackers have become more clever in preparing these malicious programs, and there is endless competition between malware developers and malware analysts. The speed of malware development is relatively high, and everyday malware developers come up with new and more sophisticated ways. The malware being developed in recent times is highly complex and uses obfuscation techniques, which makes it extremely difficult to analyse such programs. Malware can be analysed using either behaviour-based or signature-based techniques. The signature-based techniques though being fast, lose their effectiveness against obfuscation techniques and the behaviour-based techniques since they require observing the behaviour take a lot of time and make the task much more cumbersome for the analyst. Thus, the detection of malware using traditional approaches like heuristic-based, graph-based, entropy-based etc., is not possible. To tackle this ever-growing field of malware development, analysts need to learn from the malware program's behaviour. Thus, machine learning provides a solution to develop classification models to get ahead of the new variants of malware. Malware analysis techniques fall mainly under these two broad categories.

- *Static Malware Analysis*
  In this analysis, various static features of a program like hash values, opcodes, strings and PE header information are extracted without executing the program. The malicious programs are disassembled using tools like IDA Pro, Capstone, Radare etc. and then assembly code is examined to get the execution flow of the file and patterns present in the file for detecting some signs of malicious activity. This type of analysis suffers from the drawback of being highly time-consuming and much more complicated. Techniques deployed by developers like obfuscation such as code encryption, reordering instructions, dead code insertion further make the analyst's task much harder.
- *Dynamic Malware Analysis*
  In this technique, the malware is executed on the host system by making virtual environments and then logging the program's activities. Various activities of the program like file system operation, process generation and execution, API calls, and network activities are observed. Based on them, the file is classified as benign or malicious. The files which can not be analyzed using static analysis techniques can be analyzed using this technique.

Since the traditional methods suffer from the requirement of a significant workforce and time, we turn to machine learning-based approaches. Machine learning-based methods are highly generalizable and do not require much manual work. Machine learning can even learn some features that are too difficult or can not be extracted manually because of its ability to learn. In [Vu 2020], the authors use the assembly file of the program (produced by the disassembler) and convert the assembly bytecodes into pixel features and then use CNNs to learn. Although this deploys the program information, an attacker can still confuse the classifier by inserting external assembly functions. The authors in [Zhang et al. 2017] use SVM to build a malware detection framework based on the concept of supervised learning and achieve good results. In [Alazab et al. 2010], the authors use the API calls appearing made by the program. Their method relies on a single malicious API that could appear on a series of call sequences, and only the exact API sequence is harmful.

In another work, the authors [Kumar et al. 2019] classified malware using early-stage behavioral analysis. The goal is to classify malware into six malware classes (*backdoor, trojan, trojandownloader, trojandropper, virus, and worm*). *pefile* python library is used to extract the static information, and Cuckoo sandbox is used to extract the dynamic features. They use 15,000 malware samples for the analysis and extract 52 static and 72 dynamic features. The primary category for dynamic features are network-based features, API bin-based features, Process-based features, and Signature-based features. Authors perform the time analysis for three time frames of 4s, 8s, and 12s each. The best accuracy for 4s is 98.55%, for 8s is 98.61%, and for 12s is 98.65%. The accuracy for static analysis is 97.952%, and dynamic analysis is 99.135%. They also perform a hybrid analysis which achieves an accuracy of 99.736%.

Many researchers use the graph-based analysis techniques [Dam and Touili 2017][Jiang et al. 2018][Østbye 2017]. The authors in [Jiang et al. 2018] use graph-based techniques as well as Deep Learning techniques for malware analysis. The authors in [Cheng et al. 2019] use the concept of clustering based on a given binary's family dependency graph. The authors in [Pektaş and Acarman 2020] use Deep Learning to create embeddings for malware based on their API Call Graphs. In [Xiao et al. 2019] the authors use high-level features of extracted behaviour graph using Stacked-AutoEncoders. This method is precise has the disadvantage of working on the whole sample while malicious fragments are only partial, which affects the prediction of malicious behaviour.

Multiple types of research on malware detection demonstrates that feature-based deep learning malware classifiers have a better generalization to currently unseen threats and unsigned software. These methods, however, assume that API calls are independent, resulting in lower accuracy. In a work [Fadadu et al. 2019], the authors use API call sequences to reduce the false positive. These approaches use the API method call sequence as malicious patterns to detect malware. In this work, the API call sequence is extracted for each malware and the benign program. Then they extract effective sequences for malware detection. Finally, the dependency between API calls is considered based on their call sequence. The main difficulty in these approaches is the method adopted for API call sequence extraction.

The authors in [Tang and Qian 2018] analyze the API attributes and propose a map color method based on categories and occurrence time for a unit time the API executed and then use CNNs for classification. In [Xiaofeng et al. 2018], the authors proposed a new statistical method that is based on extra information addition and removal and hence led to reduction of the length of API call sequences. These sequences are then fed to LSTMs for training. Researchers also explore ways to extract features using machine learning techniques. In [Alazab and Venkatraman 2013] the authors explore ways of extracting features from the frequency of API and compare them with other neural networks. The methods based on API call sequences are accurate. Still, they suffer from long execution sequences appearing in the program whilst the actual malicious part is a tiny portion of the total code. The method of extracting efficient sequences is explored in [Liangboonprakong and Sornil 2013], but they only retain the sequential nature of the code execution. In [Massarelli et al. 2018], authors tackle the problem of finding similarity between two binary functions by producing the function embeddings through a self-attentive neural network. They also provided ways to detect malware by comparing the given program with a program known to be malicious.

3 PROPOSED METHOD

3.1 Datasets

We use namely two datasets in our experiments. The first dataset is collected from *Oliveira et. al. [Catak et al. 2020]*. It consists of 42,797 malware API call sequences and 1,079 benign API call sequences. We use this to train our network for binary classification by splitting the dataset into 1079 examples of malware and benign classes, which we split in a 7:3 train to test ratio. The dataset consists of the first 100 API call sequences of various program files created for running in the Windows Operating System.

The second dataset we use is collected from *Catak et al. [Oliveira 2019]* which features 7107 malware API call sequences across eight categories, namely *Trojan, Backdoor, Downloader, Worms, Spyware Adware, Dropper, Virus*. We use this dataset for our 8-way classification experiments. The dataset consists of unfiltered API call sequences of various malicious programs created for attacking the Windows Operating System. The API call sequences are of diverse lengths, varying from a length of just 10 to a maximum of 400K API calls. The sequences also consist of repeating calls.

Due to the almost similar natures of all the categories from the execution standpoint and the wide variety in API calls, we consider Dataset-2 a more challenging dataset than Dataset-1.

3.2 Methodology

The methodology pursued by us aims to employ the ideas of Natural Language Processing to the problem of Malware Detection, as highlighted by us earlier, and we make use of API calls used by Malware to perform this task.

Our main motivation for using this approach is the similarities we find in how the API calls are arranged in any binary file and how the words are arranged in any document in a natural language.

For example, consider any sentence from a document in the English language and any function from a Windows Executable. Just like the sentence is composed of words, the function is composed of API calls. Furthermore, consider the natural language dependency

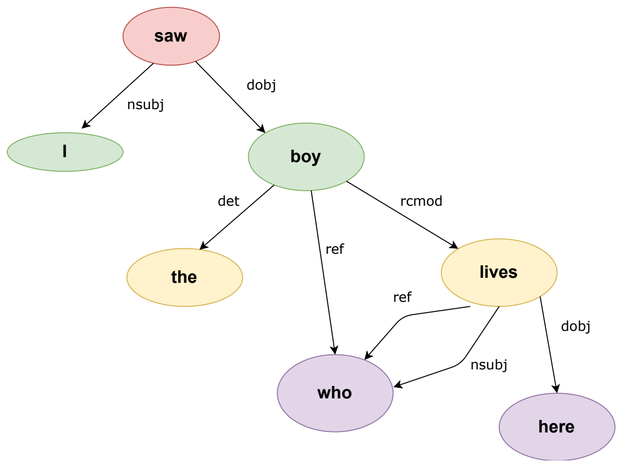

Fig. 1. Sentence Dependency Graph in English

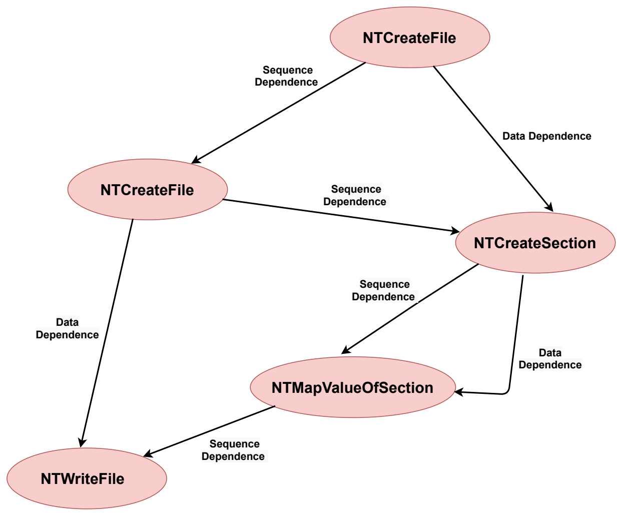

Fig. 2. An API call graph. As can be seen, its structure is very similar to that in Figure 1

graph [Chen and Manning 2014] of a sentence in the English language such as *"I saw the boy who lives here"* which is displayed in the Figure 1. The structure and connections of the generated graph are very similar to the structure of the API call graph generated from API call sequences, which is shown in Figure 2. It is intuitive as different API calls in the call graph should be related in purpose and operate upon the results of the previous API call, similarly as words in a sentence are related and operate upon the context till the previous word.

Thus we decided to capture this intuition of modelling API call sequences as a human language with the help of Natural Language Processing techniques. Our approach first tries to judge the functionality of different API calls by finding related API calls in the entire corpus, like building up a vocabulary of words.

Subsequently, we establish relations at a function and Binary File-level (sentence and document level for a language). That is, we use methods to train our classifier such that it understands what different API calls (words) represent when presented in a specific order to form a function (sentence). Further, it needs to understand how the various functions (sentences) combine to form the Binary File (document). We employ Word2Vec and LSTMs with an Attention mechanism.

3.2.1 ***API call level understanding – Word2Vec.*** To get a semantic understanding of the API call sequences, we use Word2Vec [Mikolov et al. 2013]. Word2Vec is a popular method to generate meaningful representations and understand the semantics of words and sentences and used mainly for natural languages by working on their inherent properties and structures. It figures out the relation between different words and observing the similarities in the structure of natural language sentences and API call sequences. We apply the same to generate embeddings for various API calls and thus use it to perform malware analysis.

We showcase our intention through Table 1. The first column shows a sentence in the English Language, and the words highlighted in [Galassi et al. 2020] according to their Attention score related to the task given. We give 3 more examples from our dataset [Catak et al. 2020] and the tasks being the functionality of the API calls. We take 3 tasks - *Cryptography*, *System Metrics* and *Resource Handling*. The sentences are the 10-length sequences picked up directly from the dataset. For *Cryptography*, *CryptAcquireContextW*, *CryptCreateHash* and *CryptHashData* are all examples of functions which are related to a cryptograhic module which performs operations for authentication, encoding or encryption. In the *System Metrics* task, *GetSystemMetrics* is the API function from Windows User Controls header file *Winuser.h* which provides system metrics, for example, the width of a cursor in pixels, or the number of display monitors on a Desktop. The *Resource Handling* task also accurately highlights the *LoadResource* and *FindResourceExW* functions which are used to handle resources in the memory.

Firstly, to get a representation of the relationship between the API calls at a function level, we model the functions as separate sentences and feed them into an LSTM with an Attention layer on top. The output is a vector representation of a function which is formed by applying self-attention on the constituent API calls of the function.

It is to be noted that the API call sequences obtained are usually wildly varying in length and have a lot of repetitions, which makes it difficult for them to be modelled as ordinary length sentences as found in human language. Thus we preprocess the API calls by allowing only a set number of consecutive repetitions of the same API call.

Further, we used the N-gram model and considered fixed-length sequences of these API calls equivalent to one sentence of a human language. N-grams constitute words, which makes N-grams sentences.

Subsequently, to establish a relationship between the different functions and thus in the entire Binary file, we use a second LSTM layer with Attention, which works similarly to the previous LSTM layer, however working with vectors representing functions and not individual API calls. The output of these two layers is a vector representation of the entire Binary file, which considers the order and contexts of its constituent API calls and functions. Thus, it is an accurate representation of the properties of the file, much like the embeddings produced by NLP models when a human language document is passed through them.

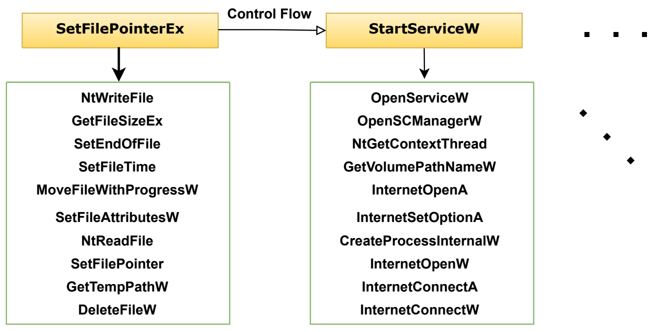

Fig. 3. API call sequences, and their respective 10 closest API functions.

3.3 Network Architecture

We train a custom Word2Vec model, as mentioned earlier. Word2Vec helps us in getting embeddings for a completely new vocabulary of API call sequences. We train the Word2Vec model on Dataset-1[Catak et al. 2020]. Figure 3 shows a small snippet of an API call sequence. We present 2 API calls, namely *SetFilePointerEx* and *StartServiceV*, and the top 10 closest API calls calculated by measuring the cosine similarity between the embeddings generated by our trained Word2Vec model. As observed in both the examples indicated in Figure 3, the functions found closest is quite related to the ones being compared. For instance, *StartServiceW* is an API function to start a service, and the two most similar API functions found are *OpenServiceW* & *OpenSCManagerW*. These functions open an existing service, establish a connection to the service control manager, and open the specified service control database. Similarly, *SetFilePointerEx* also gives us closely related functions such as *NTWriteFile* which writes data to an open file, and *SetEndOfFile* which sets the physical file size for the specified file to the current position of the file pointer. It verifies the fact that Word2Vec successfully train a model which identifies the semantics between API calls.

In total, we experiment with two varieties of models. One of them is the vanilla flavour, which contains two stacked LSTM layers with 2 Linear layers. In the second one, we attach an Attention layer after each LSTM layer, which we hypothesize will enable the representations to incorporate more relevant and dominant API calls/sequences in the vector-space representations. This will be passed to further layers to give us more meaningful and accurate representations, hence delivering better results in identifying newer and fresh malware.

With the above setup, we conduct experiments for both binary and 8-way classification. Furthermore, our model consists of an embedding layer that shares the weights of the custom trained Word2Vec model, from which we use the embeddings $v \in \mathbb{R}^k$. As

Table 1. Attention visualization for API call sequence. The first column gives is an example of a sentence in English.

Task: Cleanliness	*Task: Cryptography*	*Task: System Metrics*	*Task: Resource Handling*
Not	CryptAcquireContextW	GetSystemMetrics	LoadResource
the	NtOpenKey	NtClose	DrawTextExW
cleanest	NtQueryValueKey	GetSystemMetrics	GetSystemMetrics
rooms	NtClose	NtAllocateVirtualMemory	FindResourceExW
but	NtOpenKey	LdrLoadDll	LoadResource
bed	NtClose	LdrGetProcedureAddress	GetSystemMetrics
and	LdrGetProcedureAddress	LdrGetDllHandle	DrawTextExW
bathroom	CryptCreateHash	FindResourceExW	LdrGetDllHandle
was	LdrGetProcedureAddress	LoadResource	FindResourceExW
clean	CryptHashData	FindResourceExW	LoadResource

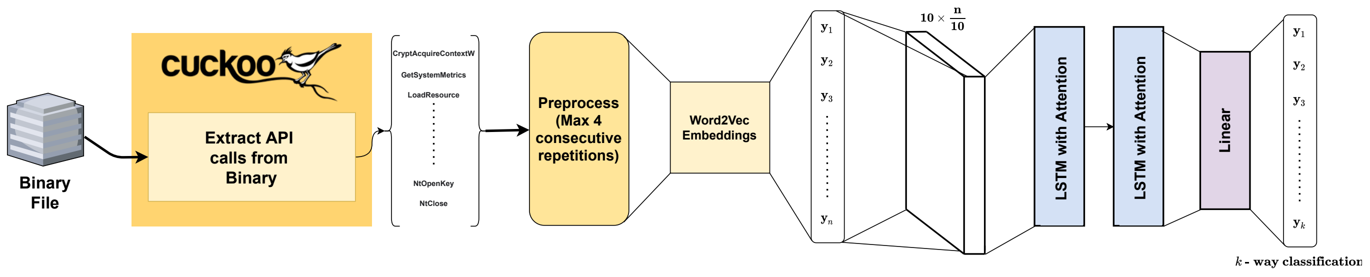

Fig. 4. Pipeline for our model

mentioned above, we experiment with models consisting of two stacked LSTM layers. The model with Attention has single Attention layer succeeding each of the LSTM layers, after which follows a Batch Normalization layer and two Dense layers, which are common in both of the models above.

Figure 4 shows the pipeline of our approach in which one may automate the process using cuckoo sandbox before Preprocess (Max 4 consecutive repetitions) block. After the Preprocess (Max 4 consecutive repetitions) block, one may automate all the steps using our approach, discussed in subsection 3.2.

4 EXPERIMENTS AND RESULTS

4.1 Experimental Setup

For the conduction of experiments, we use two variants of our model architecture as described in subsection 3.3. We use a dropout rate of 0.2 in both our LSTM layers and an N-gram size of 10 for our experiments. The length for N-gram is chosen for reasons as referred to in [Kim 2018]. We use the Adam optimizer with default configuration for training. The word embedding size is set to 20, as this value is giving us the maximum cosine similarity between related API calls.

We conduct experiments by first freezing the Word2Vec embeddings, pre-training the deep layers, and then unfreezing the Word2Vec layers for fine-tuning to learn more robust features. In case any unknown API calls are encountered in the input, they are given a $< UNK >$ tag and are assigned embeddings equal to the average of all embeddings corresponding to the known API calls.

In order to conduct Ablation studies and showcase the advantages of using Attention for malware classification, we also present results without using the Attention mechanism after the LSTM layers. We perform our experiments, modelling them as classification problems. For Dataset-1 [Catak et al. 2020], we classify the samples as either Malware or Benign.

Similarly, we report the results using Dataset-2 [Oliveira 2019]. Even this dataset is a multi-class dataset, and we report the results as a binary classification problem – performing classification as class 1 to be Trojan and 0 for other class once, then in the next iteration, classifying the samples as class 1 to be Backdoor and 0 for rest, and so on). We report the results in this fashion to be consistent with the existing literature using this dataset and the relatively small quantity of samples of each class available.

Evaluation Metrics: The datasets we use are mostly unbalanced and generally lean more towards one category than the other. For example, for Dataset-2, as we are modelling it as a binary classification problem, the number of samples with label 0 is approximately seven times that of those with label 1. Thus along with accuracy, we need to report class-wise metrics such as Recall to display our model's robustness and classification abilities.

We also report the same metrics for Dataset-1 along with the Precision and F1-score. However, we use the same number of test samples for both classes in this dataset to maintain consistency with results in the literature.

4.2 Results

4.2.1 ***Dataset-1.***

It is necessary to make the number of training and testing samples equal for Dataset-1, which is highly imbalanced, as in subsection 3.1 to perform experiments. Thus, we randomly sample 1079 data points from Malware, take all the 1079 data points from the benign category, and then randomly divide these in the ratio of 7:3 in the train to test data. The remaining samples from Malware Class are discarded as they result in an imbalance in training. This exact procedure is followed by the original paper presenting the dataset, [Catak et al. 2020], which ensures experimental consistency.

We compare the results of both methods with and without attention with the results presented by [Catak et al. 2020], that is, using 1 and 2 layer Deep Graph Convolutional Neural Networks and a two-layer LSTM. The results using various evaluation metrics are shown in Table 2.

Table 2. Results of experiments on Dataset-1 [Catak et al. 2020] on various models. The best results for each metric are in **bold**

Method	F1-Score	Precision	Recall	Accuracy
1-Layer DGCNN [Catak et al. 2020]	0.9076	0.8879	0.9283	0.9105
2-Layer DGCNN [Catak et al. 2020]	0.9201	0.9216	0.9186	0.9244
LSTM [Hochreiter and Schmidhuber 1997]	0.8738	0.8542	0.8932	0.8727
Our approach (no Attention)	0.9586	0.9508	0.9667	0.9583
Our approach (Attention)	**0.9697**	**0.9586**	**0.9810**	**0.9693**

As see in the results present in Table 2, our methods far outperform the results of the methods reported in [Catak et al. 2020]. We also report the accuracy of a two-layer LSTM method. We feed-forward the outputs from the first LSTM to the second without using any N-grams and concatenating them before being passed as input in the second LSTM layer. The results on a single layer LSTM come out to be even lower than the DGCNN results. It shows that using just the recurrent properties of LSTM is not enough to ensure enough attention paid to the inputs and the context in the API calls is taken care of by the network.

Our results are also visibly better when using Attention with our method, as compared to without it. It shows that using just Word2Vec embeddings is not enough to ensure that the network utilizes the context-dependence of the API calls. We need a dedicated attention mechanism to ensure that the context is utilised.

4.2.2 ***Dataset-2.***

Dataset-2 is relatively complicated, as mentioned in subsection 3.1. The API calls in this dataset are repeating in nature and also are vary significantly in length. Thus, it is important to preprocess this dataset.

We observe that the significant variation in lengths from 10 to 400K is mostly due to repetitions. If we removed any two consecutive calls to the same API function, it resulted in a maximum length of 345 API calls. Thus, to test the effectiveness of our formulation, we allowed a small amount of repetition in which we let a maximum of 4 consecutive calls to the same API function to get our final data points. In contrast, excessive repetitions are removed, and the first distinct function call took its place after all the repeats.

Subsequently, we obtain a dataset where the maximum length of the input is 485, while the shortest length stays at 10. In order to ensure good results and make sure that this high variation in lengths does not cause a problem in classification, we trim the API call sequences to a maximum length of 200.

Following these pre-processing methods allow us to get our final dataset to conduct experiments. We again use this dataset in a binary classification setting for comparing with other results available in the literature. Thus to perform experiments on one class, we labelled all the samples belonging to that class as 1 and the remaining samples as 0. We use an 8:2 ratio train-test split, ensuring that the number of samples with label 1 and 0 are equal in the training and testing data to demonstrate our performance better.

We compare the results of both of our methods with the results present in [Oliveira 2019], which uses a simple two-layer feedforward LSTM, learning the embeddings while training. We also compare our results with popular machine learning algorithms which do not utilize deep feature extraction layers, using TF-IDF vectors as the embeddings. The results are presented in Table 3. We report the Class wise accuracy as it is a multi-class dataset.

As we see in Table 3, both our methods perform much better than the other methods in all the tasks and have a significant boost in the mean performance per class. Our methods also perform very well on the harder classification tasks in this dataset: Spyware and Trojan (which have significantly less recall values when the other methods are used).

Similar to Table 2, our attention based performs better than the method without attention, even in Table 3.

5 CONCLUSION

In this work, we explore a way for analyzing the maliciousness in the program using the API call sequences present in that. We utilise the inherent structure of these API call graphs by looking at their similarity with the dependency graphs of the English language. It naturally hints us to go for the domain of Natural Language Processing.

Hence, we design an NLP-based detection framework for the detection of malicious programs. Modelling the API segments in the English language sentences and then learning the features using techniques like "Attention" and NLP models like LSTMs help us produce better results than previous work adopting similar methods. The experimental results also show that our approach is stable and efficient across different datasets. Our work effectively explores the application of NLP-based techniques in the malware analysis field, which can have important significance on future researches in this field. In future work, one may explore more complex NLP-based primitives like Transformers instead of LSTMs. Deploying the method as a practical application is also a potential future work after optimising the technique and pipeline further.

Table 3. Results of experiments on Dataset-2 [Oliveira 2019] on various models. The best results on each malware type are in **bold**. The precision of the results is upto 2 decimal places for consistency with the reported results in the literature.

Model	Adware	Backdoor	Downloader	Dropper	Spyware	Trojan	Virus	Worm	Average
Adaboost [Freund and Schapire 1995]	0.76	0.52	0.69	0.57	0.41	0.51	0.74	0.57	0.60
Decision Tree [Breiman et al. 1984]	0.45	0.40	0.51	0.37	0.11	0.16	0.41	0.78	0.40
kNN [Wang and Su 2011]	0.70	0.57	0.67	0.45	0.32	0.32	0.62	0.49	0.52
RF [Breiman 2001]	0.48	0.62	0.52	0.35	0.17	0.16	0.80	0.58	0.42
2-layer LSTM [Hochreiter and Schmidhuber 1997]	0.77	0.56	0.59	0.44	0.42	0.28	0.68	0.45	0.52
Ours (no Attention)	0.84	0.77	0.83	0.84	0.80	0.71	0.92	0.82	0.82
Ours (Attention)	**0.94**	**0.85**	**0.96**	**0.87**	**0.84**	**0.77**	**0.96**	**0.88**	**0.88**

REFERENCES

2020. Cuckoo Sandbox - Automated Malware Analysis. https://cuckoosandbox.org/.

Mamoun Alazab and Sitalakshmi Venkatraman. 2013. Detecting malicious behaviour using supervised learning algorithms of the function calls. *International Journal of Electronic Security and Digital Forensics* 5, 2 (2013), 90–109.

Mamoun Alazab, Sitalakshmi Venkatraman, Paul Watters, Moutaz Alazab, et al. 2010. Zero-day malware detection based on supervised learning algorithms of API call signatures. (2010).

Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (Oct. 2001), 5–32. https://doi.org/10.1023/A:1010933404324

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees.* Wadsworth and Brooks, Monterey, CA.

Ferhat Ozgur Catak, Ahmet Faruk Yazı, Ogerta Elezaj, and Javed Ahmed. 2020. Deep learning based Sequential model for malware analysis using Windows exe API Calls. *PeerJ Computer Science* 6 (July 2020), e285. https://doi.org/10.7717/peerj-cs.285

Danqi Chen and Christopher Manning. 2014. A Fast and Accurate Dependency Parser using Neural Networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).* Association for Computational Linguistics, Doha, Qatar, 740–750. https://doi.org/10.3115/v1/D14-1082

B. Cheng, Q. Tong, J. Wang, and W. Tian. 2019. Malware Clustering Using Family Dependency Graph. *IEEE Access* 7 (2019), 72267–72272. https://doi.org/10.1109/ACCESS.2019.2914031

Khanh-Huu-The Dam and Tayssir Touili. 2017. Malware Detection based on Graph Classification. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy, ICISSP 2017, Porto, Portugal, February 19-21, 2017*, Paolo Mori, Steven Furnell, and Olivier Camp (Eds.). SciTePress, 455–463. https://doi.org/10.5220/0006209504550463

Fenil Fadadu, Anand Handa, Nitesh Kumar, and Sandeep Kumar Shukla. 2019. Evading API Call Sequence Based Malware Classifiers. In *International Conference on Information and Communications Security.* Springer, 18–33.

Yoav Freund and Robert E. Schapire. 1995. A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting. In *Proceedings of the Second European Conference on Computational Learning Theory (EuroCOLT '95).* Springer-Verlag, Berlin, Heidelberg, 23–37.

Andrea Galassi, Marco Lippi, and Paolo Torroni. 2020. Attention in Natural Language Processing. *IEEE Transactions on Neural Networks and Learning Systems* (2020), 1–18. https://doi.org/10.1109/tnnls.2020.3019893

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

H. Jiang, T. Turki, and J. T. L. Wang. 2018. DLGraph: Malware Detection Using Deep Learning and Graph Embedding. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA).* 1029–1033. https://doi.org/10.1109/ICMLA.2018.00168

Chan Woo Kim. 2018. NtMalDetect: A Machine Learning Approach to Malware Detection Using Native API System Calls. (02 2018).

Konstantinos Kosmidis and Christos Kalloniatis. 2017. Machine Learning and Images for Malware Detection and Classification. https://doi.org/10.1145/3139367.3139400

N. Kumar, S. Mukhopadhyay, M. Gupta, A. Handa, and S. K. Shukla. 2019. Malware Classification using Early Stage Behavioral Analysis. In *2019 14th Asia Joint Conference on Information Security (AsiaJCIS).* 16–23. https://doi.org/10.1109/AsiaJCIS.2019.00-10

Chatchai Liangboonprakong and Ohm Sornil. 2013. Classification of malware families based on n-grams sequential pattern features. In *2013 IEEE 8th Conference on Industrial Electronics and Applications (ICIEA).* IEEE, 777–782.

Luca Massarelli, Giuseppe Luna, Fabio Petroni, and Leonardo Querzoni. 2018. SAFE: Self-Attentive Function Embeddings for Binary Similarity.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs.CL]

L. Nataraj. 2015. A Signal Processing Approach To Malware Analysis.

Angelo Oliveira. 2019. Malware Analysis Datasets: API Call Sequences. https://doi.org/10.21227/tqqm-aq14

Morten Oscar Østbye. 2017. Multinomial malware classification based on call graphs.

Abdurrahman Pektaş and Tankut Acarman. 2020. Deep learning for effective Android malware detection using API call graph embeddings. In *Soft Computing.* 1027–1043. https://doi.org/10.1007/s00500-019-03940-5

Mingdong Tang and Quan Qian. 2018. Dynamic API call sequence visualisation for malware classification. *IET Information Security* 13, 4 (2018), 367–377.

Duc-Ly Vu. 2020. DeepMal: Deep Convolutional and Recurrent Neural Networks for Malware Classification. https://doi.org/10.13140/RG.2.2.19224.42246

Juntao Wang and Xiaolong Su. 2011. An improved K-Means clustering algorithm. In *2011 IEEE 3rd International Conference on Communication Software and Networks.* 44–46. https://doi.org/10.1109/ICCSN.2011.6014384

Fei Xiao, Zhaowen Lin, Yi Sun, and Yan Ma. 2019. Malware detection based on deep learning of behavior graphs. *Mathematical Problems in Engineering* 2019 (2019).

Lu Xiaofeng, Zhou Xiao, Jiang Fangshuo, Yi Shengwei, and Sha Jing. 2018. ASSCA: API based sequence and statistics features combined malware detection architecture. *Procedia Computer Science* 129 (2018), 248–256.

Kai Zhang, Chao Li, Y. Wang, Xiaobin Zhu, and H. Wang. 2017. Collaborative Support Vector Machine for Malware Detection. In *ICCS.*