Neural Computing and Applications (2021) 33:11963–11983
https://doi.org/10.1007/s00521-021-05861-7

ORIGINAL ARTICLE

Deep learning-aided runtime opcode-based Windows malware detection

Enes Sinan Parildi[1] · Dimitrios Hatzinakos[1] · Yuri Lawryshyn[2]

Received: 13 September 2020 / Accepted: 19 February 2021 / Published online: 21 March 2021

Abstract

Thousands of new malware codes are developed every day. Signature-based methods, which are employed by common malware detectors, are susceptible to code obfuscation and novel malware. In this paper, we present an alternative method for malware detection, which makes use of assembly opcode sequences obtained during runtime. First, for sequential opcode data, we utilize natural language processing and deep learning techniques to facilitate the extraction of deeper behavioral features. Due to these features, this method can be impervious to code obfuscation and effective against novel malware. Finally, these features are fed to various machine learning algorithms for classification. The experiments on a more class balanced dataset of 26869 samples demonstrated that MCC (Matthew's correlation coefficient) score as high as 0.95 is achievable with this approach. The MCC score results for the experiments conducted on imbalanced and artificially balanced datasets are 0.81 and 0.83, respectively.

Keywords Malware detection · Deep learning · Opcodes · Natural language processing

1 Introduction

Malware is any type of software created intentionally to cause malicious effects on software-based systems, such as computers, servers, and industrial control systems. [23] summarizes some of the malicious effects such as disrupting the ordinary course of computer operation, gathering or eliminating sensitive information through encrypting or gaining access to a computer system. A malware detector (more commonly known as an anti-virus program) is a specialized computer program that attempts to locate and eliminate malware. [32] states that detecting all kinds of malware with a high success rate is still an unsolved problem as attackers develop new kinds of malware and evasion techniques every day. Kaspersky [17], which is one of the most prominent anti-virus vendors, states that in 2017 the number of malware they encountered attained 360000 mark in a day. [38] states that the vast majority of most common commercial malware detectors utilize a detection approach that relies on signatures. This signature approach involves a lookup of a pre-existing database for query signature. The success of this strategy naturally depends on how fast the database is updated with new malware samples. Therefore, the fact that an enormous amount of new malware is being produced every day makes it gradually harder for the signature-based detection approach to be effective. Another approach, which is called heuristic-based malware detection in the malware research community, as defined by [2] aims to extract behavioral information of malware by primarily utilizing predictive modeling techniques such as traditional machine learning method or less commonly deep learning methods. This approach, unlike the signature-based approaches, can be successful against unseen samples from the real world. Based on these arguments, in this work, we primarily aim to study deep learning techniques on dynamic opcode input to assess its potential of becoming a generalizable malware detector.

✉ Enes Sinan Parildi
enessinan.parildi@mail.utoronto.ca

[1] Department of Electrical and Computer Engineering, University of Toronto, 27 King's College Cir, Toronto, ON, Canada

[2] Department of Chemical Engineering & Applied Chemistry, University of Toronto, 27 King's College Cir, Toronto, ON, Canada

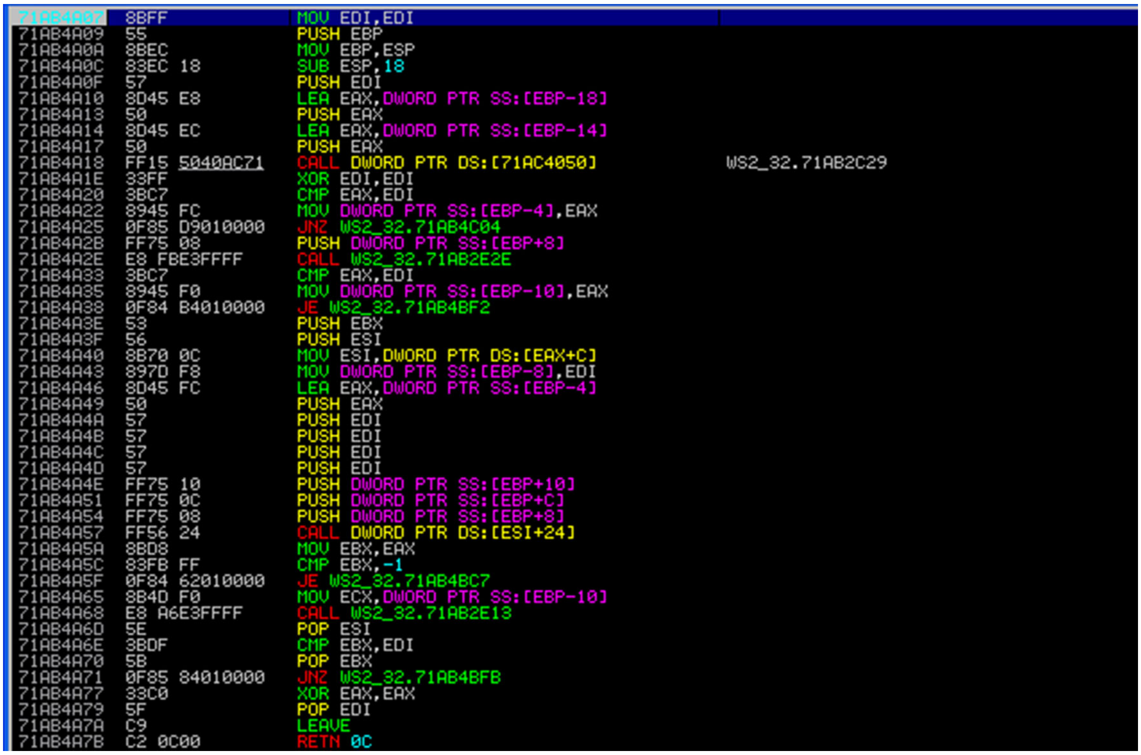
Static analysis and dynamic analysis are two well-known general forms of malware analysis. Static analysis, which is more ubiquitous than dynamic analysis, tries to identify malware under inspection before its execution using structural information (sequence of codes). For instance, the signature-based method is a common form of static analysis. Static analysis can also be applied in a heuristic-based setting. Conversely, dynamic analysis attempts to identify the malware during its execution or after execution using only the runtime information such as behavior or actions of the malware. In dynamic analysis, suspected malware is executed on a virtual operating system. After that, memory access, order of memory access, assembly instructions (opcodes), and system call statistics are analyzed. [29] claims that static techniques, while in general work quite well against most threats, code obfuscation methods, which aim to make machine code more difficult to analyze, can make the malware most of the times undetectable. As exemplified by [3], junk code insertion, code transposition, code substitution are some of the common examples of code obfuscation methods. [28] states that dynamic techniques, on the other hand, are inherently more resistant to code obfuscation methods but are more resource-intensive. This resistance stems from the fact that dynamic analysis only considers the instructions executed during runtime.

In an effort to assist the development of much more effective heuristic-based malware detection methods, [6] introduces a dataset that consists of dynamically yielded opcodes belonging to various malware and benign executables. These opcodes were obtained from runtime trace using virtual machine-aided dynamic analysis. Consequently, this runtime opcode dataset is valuable to develop an advanced malware detection system, provided that the dataset is combined with effective predictive modeling techniques. This system has the potential to be effective against novel malware as well as various concealment and code obfuscation strategies employed by the malware developers according to [6]. The skeleton of our work originates from this dataset. Figure 1 illustrates a small snapshot of this dataset in its unprocessed form.

1.1 Motivations and contributions

The main objective of this research is to assess the effectiveness of various machine learning-based predictive model pipelines for malware detection when these pipelines admit runtime, in other words, dynamic opcode type input. In general, to the best of our knowledge, this work constitutes one of the largest-scale work on x86 architecture runtime opcode-based Windows malware detection that involves a novel application of cutting edge natural language processing (NLP) and deep learning methods. We hypothesize that the most widespread techniques used in NLP could facilitate effective and robust feature extraction by providing heightened comprehension from raw opcode sequence data. The predictive modeling pipeline applied to the dataset consists of preprocessing, embedding, deep learning feature extraction, and a classification stage. The same pipeline is assessed for a variety of deep learning and machine learning method in order to compare their performances when it comes to relatively unstudied opcode sequence data from a predictive modeling perspective. The comparison could be important since different deep learning algorithms may have distinct inductive biases. Secondly, it is crucial to state that the original dataset from [6] has disproportionately more malware than benign instances. This inadequacy can be critical because class imbalance may severely disturb classification performance by exacerbating false positive error rate. As stated in [18], the false-positive error rate is a lingering issue within commercial signature-based malware detectors. Therefore, a special focus is given to investigate and alleviate the effect of data imbalance. Accordingly, the experiments are further organized to assess the effect of the data augmentation methods, sample weighting, and natural data collection. Thirdly, motivated by the fact that deep learning operations may require substantial computing power and resources, a version of our pipeline that omits deep learning is employed and assessed in a number of experiments. This pipeline mainly relies on self-supervised opcode embeddings, which is much more lightweight in terms of memory usage and computation time. Fourthly, considering that runtime opcode-based detection is relatively less covered in the literature, we hope that our work gives a comprehensive overview to the wider research community about the capabilities of runtime opcodes for the purpose of malware detection. In this sense, our work can be regarded as complementary to the [6] and [7]. Finally, in order to rectify the aforementioned data imbalance, we implemented a data collection pipeline that begins with the benign executable collection and ends with 1000 runtime opcodes. The pipeline has generated 6405 samples so far. The same pipeline can be used to generate more. We believe that the benign runtime opcode dataset we have collected so far can be beneficial to the wider research community and industrial applications. Moreover, the unprocessed version of the benign executable corpus can be used to build a malware detector classifier based on techniques other than runtime opcodes, and therefore, it can be useful for the research community in other ways. Our contributions are summarized as follows:

- We proposed an effective prediction pipeline that consists of a data collection, data preprocessing, feature



```
71AB4A07  8BFF            MOV EDI,EDI
71AB4A09  55              PUSH EBP
71AB4A0A  8BEC            MOV EBP,ESP
71AB4A0C  83EC 18         SUB ESP,18
71AB4A0F  57              PUSH EDI
71AB4A10  8D45 E8         LEA EAX,DWORD PTR SS:[EBP-18]
71AB4A13  50              PUSH EAX
71AB4A14  8D45 EC         LEA EAX,DWORD PTR SS:[EBP-14]
71AB4A17  50              PUSH EAX
71AB4A18  FF15 5040AC71   CALL DWORD PTR DS:[71AC4050]            WS2_32.71AB2C29
71AB4A1E  33FF            XOR EDI,EDI
71AB4A20  3BC7            CMP EAX,EDI
71AB4A22  8945 FC         MOV DWORD PTR SS:[EBP-4],EAX
71AB4A25  0F85 D9010000   JNZ WS2_32.71AB4C04
71AB4A2B  FF75 08         PUSH DWORD PTR SS:[EBP+8]
71AB4A2E  E8 FBE3FFFF     CALL WS2_32.71AB2E2E
71AB4A33  3BC7            CMP EAX,EDI
71AB4A35  8945 F0         MOV DWORD PTR SS:[EBP-10],EAX
71AB4A38  0F84 B4010000   JE WS2_32.71AB4BF2
71AB4A3E  53              PUSH EBX
71AB4A3F  56              PUSH ESI
71AB4A40  8B70 0C         MOV ESI,DWORD PTR DS:[EAX+C]
71AB4A43  897D F8         MOV DWORD PTR SS:[EBP-8],EDI
71AB4A46  8D45 FC         LEA EAX,DWORD PTR SS:[EBP-4]
71AB4A49  50              PUSH EAX
71AB4A4A  57              PUSH EDI
71AB4A4B  57              PUSH EDI
71AB4A4C  57              PUSH EDI
71AB4A4D  57              PUSH EDI
71AB4A4E  FF75 10         PUSH DWORD PTR SS:[EBP+10]
71AB4A51  FF75 0C         PUSH DWORD PTR SS:[EBP+C]
71AB4A54  FF75 08         PUSH DWORD PTR SS:[EBP+8]
71AB4A57  FF56 24         CALL DWORD PTR DS:[ESI+24]
71AB4A5A  8BD8            MOV EBX,EAX
71AB4A5C  83FB FF         CMP EBX,-1
71AB4A5F  0F84 62010000   JE WS2_32.71AB4BC7
71AB4A65  8B4D F0         MOV ECX,DWORD PTR SS:[EBP-10]
71AB4A68  E8 A6E3FFFF     CALL WS2_32.71AB2E13
71AB4A6D  5E              POP ESI
71AB4A6E  3BDF            CMP EBX,EDI
71AB4A70  5B              POP EBX
71AB4A71  0F85 84010000   JNZ WS2_32.71AB4BFB
71AB4A77  33C0            XOR EAX,EAX
71AB4A79  5F              POP EDI
71AB4A7A  C9              LEAVE
71AB4A7B  C2 0C00         RETN 0C
```

Fig. 1 Illustration of an assembly code sequence. First part of each line denotes an opcode. Second part is called operand

extraction, and classification process, tailored for detecting malware using runtime opcode sequence.

- We carried out a large number of experiments that assess our pipeline in the imbalanced dataset setting, artificially balanced dataset setting, and naturally balanced dataset setting. The experiments demonstrated that our method could distinguish malware from benign with close to 98% accuracy.
- We demonstrated that even simple classification algorithms that primarily utilize the mean of self-supervised Word2Vec-based opcode embeddings without expensive deep learning operations can achieve respectable classification performance from only 1000 opcodes.
- Building upon [6] and [7], we further validate the effectiveness of runtime (dynamically yielded) opcodes for malware detection with our method.
- We developed a data collection pipeline for runtime opcodes similar to the one described in [6]. In contrast, our pipeline focuses on collecting and processing benign samples.

Section 2 presents some existing literature. The emphasis is given to the works that include deep learning techniques. Section 3 explains the methodology by first describing the original dataset and then the data collection pipeline. The next subsection presents the preprocessing of the dataset and the generation of the embedding vectors. Then, the three deep learning algorithms that we used for the feature extraction are introduced. Section 4 presents all the results of the three categories of dataset settings. Section 5 first covers the general discussions of our approach, the results, and some possible next steps to further this work.

2 Related research

Malware detection is a very broad problem, and it has been investigated by many researchers from a large number of different perspectives. Here, we only review the studies that make use of machine learning and especially deep learning. Other approaches that are less relevant to our work and are mostly based on signature matching are omitted for simplicity. Data mining or machine learning-based malware detection methods can also be categorized in itself with respect to the raw dataset type they use (opcode sequence, opcode n-gram, system call, hexadecimal bytes), how the dataset is yielded (dynamic, static), and the platform they target (Windows, Android). In general, methods that rely on dynamically yielded are rare compared to others. Our method stands out among these studies by relying primarily on a dynamically yielded opcode dataset. Moreover, our method involves extensive usage of advanced deep learning methods (CNN, LSTM, Transformer), which is also a very recent development in the malware detection problem. It is also worth noting that, to our knowledge, Transformer has not been utilized in the existing literature. Several properties arguably make Transformer superior to CNN and LSTM in natural language processing. Therefore, it is crucial to investigate Transformer for malware detection as well.

[20] investigated the performance of a convolutional neural network by treating dissembled opcode sequence as a text. Thus, they modeled malware detection problem as a text classification problem. The authors emphasized that a deep learning approach is particularly useful in the sense that it removes the reliance on feature engineering, since convolutional neural network can discover its features. These features can be complementary to hand-crafted features produced by human experts. CNN (convolutional neural network) is originally designed for processing images; in this problem, when the opcode sequence passes through the embedding layer, it obtains the same form as image type, thus becoming suitable for convolutional neural network. The authors achieved 97% f1 score on their small dataset and 86% on their large dataset. Both datasets are Android malware datasets. Reference [20] shares the idea of modeling malware detection problem as a text classification problem and extensive usage of deep learning methods with our work. On the other hand, our method also emphasizes Word2Vec, which facilitates deep learning-based feature extraction greatly.

[7] tried to investigate if it is possible to improve malware detection using dynamically yielded opcodes. To achieve this objective and assist future research, they assembled a huge, dynamically yielded opcodes dataset consisting of 13 different malware types and benign software with close to 48,000 samples. They merged 13 malware types into a single malware class and compared them to benign set to perform binary classification. Since the benign class only possesses 1045 samples in total, synthetic minority oversampling technique was utilized to boost the size of the benign class artificially. The size was increased to five times the original size. Secondly, they used n-gram representation for feature extraction. In order to combat huge feature dimensionality, a feature selection methods based on information gain ratio was applied. Thirdly, they utilized the random forest algorithm for the final classification. Some hyperparameters, such as the number of trees and features, are optimized via tenfold cross-validation and grid search. As a result, they achieved an impressive 99% F1 score with 32K opcodes. However, the f1 score drops to 93.8% when only 1000 opcodes are used. Finally, note that most of the malware samples used in this work are the extended version of the malware section of our dataset. As future work, we plan to use this extended version of the malware portion of the dataset.

[40] developed LSTM (long short-term memory)-based hierarchical denoise network using hierarchical structure to solve very long statically yielded opcode sequence learning and gradient vanishing problems. The authors aimed to process long opcode sequences without resorting to n-grams. As a result, they showed that their method outperforms the n-gram-based method and vanilla LSTM without a hierarchical denoise network. Moreover, they also compared methods for embedding opcodes. Specifically, they compared simple one-hot encoding with an embedding learned from data and showed that opcode embedding that is learned from data in a supervised way improves the performance compared to one-hot encoding. As a result, their method achieved accuracy close to 99% percent on an open-source pre-crafted android malware dataset. [40] relies on supervised learning of embedding as opposed to Word2Vec. Supervised embedding learning may require significantly more labeled data instances.

[15] tried to classify ten different malware clusters using a combined deep learning method. More specifically, they first obtained a tri-gram of dynamically yielded kernel API call sequences (system calls) of malware, and then they processed one hot encoded version of these sequences by using a convolutional layer. Finally, they created a sequential model by using LSTM cells. The dataset that they utilized contains 4753 malware samples. As a result, they achieved 89.4% accuracy. [15] primarily uses a one-hot encoding and n-gram approach to encode tokens as opposed to vector embeddings. [14] combined information obtained from portable executable metadata such as names of imported DLLs, timestamp of the compilation, and statically yielded one-hot encoded opcode sequences. More specifically, the authors used feed-forward neural network layers to extract features from PE metadata and combined convolutional layers to extract features from static opcode sequences. In the end, the authors used a final dense layer to get classification scores. They utilized the dataset of 22,694 malicious executables and 63 benign executables, which is imbalanced. However, they noted that multi-class classification, where they identified 13 different malware types, alleviated the balance problem. As a result, the authors achieved 93% on precision and recall with a 92% on F1-measure [14] also relies on one-hot encoding instead of Word2Vec. However, their approach of incorporating executable metadata inside the deep learning context could be implemented as future work.

[5] used Word2Vec algorithm to obtain a fixed vector representation of statically yielded opcodes belonging to executables. Word2Vec representation of opcodes (opcode embeddings) was fed into gradient boosted trees for final classification. The dataset they utilized was derived from Microsoft Malware Classification Challenge Dataset. In the end, the authors achieved 96% accuracy on 8 class

malware classification. [5] utilizes Word2Vec to obtain semantically meaningful representations of opcodes but does not attempt to perform sequence learning from Word2Vec representations. Similarly, opcode embedding approach in conjunction with CNN is also applied by [27] to statically yielded opcodes.

[39] used a shallow ensemble of CNN (convolutional neural network) and LSTM (long short-term memory) to build a malware detection system that relies on static analysis. More specifically, the CNN part of the system uses greyscale images generated from a raw binary file. LSTM part utilizes statically yielded opcodes. A simple logistic regression classifier generates classification labels using a stacked ensemble feature vector of LSTM and CNN. For the LSTM classification of opcode sequences, the authors employed various strategies to reduce the noise present in the statically yielded opcode sequence as well as to facilitate long sequence learning. As the dataset, they used a dataset derived from Microsoft Malware Classification Challenge Dataset. In the end, the authors achieved accuracy close to 99% on binary malware-benign classification. [39] also makes use of executable metadata, which is not included in our method.

A very recent study by [25] highlights the fact that the traditional signature-based malware detection approach may underperform to detect malware in drones and their ground control stations. They propose another detection approach based on the fastText model and bidirectional LSTM to classify different malware types. More specifically, they utilize the fastText model to create input embeddings that are lower-dimensional than vectors created by the one-hot encoding approach. After that, Bi-LSTM analyzes statically yielded opcode sequences to obtain classification results. Moreover, static opcode sequences are augmented by extra features based on API function names. They utilized the Microsoft Malware Classification Challenge Dataset in their experiments. Finally, their method demonstrates a performance improvement of 1.87% compared to performance obtained by a one-hot encoding-based approach. Furthermore, their deep learning-based method achieves performance improvement of 0.76% over a similar decision tree-based approach.

Another recent study by [26] explores a hybrid deep learning approach that combines convolutional autoencoder and dynamic recurrent neural network. They utilize statically yielded opcode sequences with a random branch selection strategy to approximate the execution path to detect malware. More specifically, CNN at the front end compresses long opcode sequences to shorter sequences, and after that, a dynamic recurrent neural network performs the detection from the compressed sequence. The dataset that they constructed for the experiments consists of 1000 malware 1000 benign. As a result, they achieved 96% accuracy score.

[9] implemented a lightweight and simple android malware detection method that relies on statically yielded opcode sequences. First, they utilized a symbol-based simplification method to abstract the Dalvik opcode sequence. Second, they employed an n-gram that converts sequential data to a fixed-size feature vector (vectorization). Most well-known machine learning algorithms accept fixed-size features, unlike sequential data. Third, they experimented with random forest, support vector machine (SVM), the k-nearest neighbors (k-NN), and naive Bayes method to build the final classifier. They achieve the maximum of 0.988 recall and 0.921 precision score on 2400 balanced samples with their 3-gram random forest model. They also applied tenfold cross-validation.

A large majority of the aforementioned papers have several deficiencies. First, [9], and many other works utilize the n-gram approach that enables conversion of sequence to a fixed size feature vector, but it ignores most of the temporal information in the sequence. Increasing n would mitigate this issue, but in turn, can create a huge number of features. [7] achieves high performance with dynamic opcode n-gram feature set, which is obtained from 32K opcodes, by using feature selection and random forest. However, for the same settings, the performance is much poorer when 1000 opcodes are used instead. Note that in all settings, we use 1000 opcodes. Using longer opcode sequences can be advantageous in terms of predictive power. On the other hand, longer opcode sequences incur much higher computation time during the initial raw executable processing stage. The computation time spent on extracting 32K runtime opcodes from each sample would be significantly higher than 1000 runtime opcodes. Furthermore, the natural benign dataset in their experiments is quite limited in number. [40] demonstrates that full sequential learning can be superior to n-gram when it comes to Android malware detection. Moreover, [21] asserts that for natural language, neural language models such as Word2Vec, which is used extensively in our work, performs better than n-gram. Moreover, in terms of speed and scalability, Word2Vec offers several advantages. It also does not require labeled data. On the other hand, only three papers [40, 20] and [14] attempt to detect malware from full sequential modeling using advanced deep learning methods similar to the general approach embraced in our work. However, each of these three papers has its own distinctions and weaknesses. The most glaring weakness common in [14, 20] and [40] is that they all rely on statically yielded opcodes that could make their method more vulnerable to code obfuscation. Additionally, in [40] statically yielded long opcode sequences undergo extensive preprocessing to be suitable for LSTM. Dynamic opcode sequence requires significantly less preprocessing in our case. Secondly, [40] and [20] primarily target the Android operating system, unlike our work, which exclusively targets Windows operating system. These two operating systems are based on distinct instruction set architecture, and thus opcodes. Thirdly, in terms of predictive modeling methods, [40] and [20] are similar to our work when it comes to usage of LSTM-RNN (LSTM recurrent neural network) and convolutional network (CNN). One difference is, in their method, opcode embeddings are trained in a supervised manner along with the rest of the network. In our work, opcode embeddings are generated in a self-supervised manner using Word2Vec, which overall is similar in terms of performance but gives our method more flexibility. This flexibility can be crucial in the case of a semi-supervised setting since Word2Vec does not require any labels. Once the Word2Vec embeddings are obtained from a larger and cheaper unlabeled corpus, they can be used in any new labeled samples very efficiently. This property of Word2Vec also addresses the scalability and speed issue emphasized in [9]. Moreover, the results of an experiment done on the Transformer architecture, which will be introduced in Sect. 3.3, showed that attempting to learn embeddings jointly with the rest of the network yielded a poorer classification performance compared to using Word2Vec embeddings directly. On the other hand, [14] utilize one-hot encoding instead of explicit opcode embedding. One-hot encoding is simple, but it creates huge vector dimensionality and sparse input that does not include semantic information of words. Huge vector dimensionality at the input requires a larger deep learning architecture, which can be undesirable in many ways. According to [40], one-hot encoding may yield worse training stability with LSTM. [25] also emphasizes and demonstrates that high dimensionality incurred by one-hot encoding may decrease learning efficiency and the final classification performance. It is also important to state that vanilla LSTM, which is utilized by many works in the literature, could be inefficient at learning patterns in very long sequences. We propose an extension to vanilla LSTM in order to mitigate this issue. Finally, we addressed the dataset imbalance issue, which is another largely unstudied subject in malware detection literature. We think that addressing data imbalance better reflects real-life conditions.

3 Methodology

3.1 Dataset

As it has been emphasized before, the original skeleton of the sequential opcode dataset is adopted from [6]. This first version of the sequential dataset contains 731 benign, 18827 malware samples in total. According to reference, [6], benign samples were harvested from native Windows 7 applications and a compendium of games. Likewise, malware samples were collected from VirusShare.com. Evidently, the imbalance class issue is present in this original dataset. This issue will be addressed in the next section. Each sample consists of the first 1000 opcodes of their respective runtime trace, meaning each data point is a fixed 1000 length sequence. All possible x86 opcodes here can be used to build a dictionary of tokens similar to the text preprocessing done in natural language processing. By considering these fundamental properties of our dataset, we can draw an analogy between our opcode sequence classification and the text classification problem. More specifically, each opcode can be projected as words, and likewise, a whole opcode sequence can be processed as a long sentence or text. This treatment of the opcode dataset enables us to take advantage of some of NLP (natural language processing) methods by adapting them to our needs. Note that, despite some similarities, opcode sequences are very different from natural languages. However, since the NLP methods that we utilize in this work and most of the recent well-known language models are statistical in nature, differences are less impactful than similarities. A statistical language model could learn and encode the necessary statistical properties of opcode sequences from scratch.

3.2 Dataset class imbalance

In the original dataset, which is adopted from [6], the ratio of the negative benign class size to the size of the positive malware class is 0.038. Machine learning algorithms and deep learning algorithms, in general, perform poorly when the class imbalance is severe. The ratio value of 0.038, in our case, definitely indicates that class imbalance can be a severe issue. We explored four different strategies to combat class imbalance and prevent it from severely undermining an underlying classifier's performance. The first strategy involves using resampling methods such as Synthetic Minority Over-sampling Technique (SMOTE) or perform over-sampling using Adaptive Synthetic (ADASYN). According to [39], due to their reliance on Euclidean measures, these methods are not directly applicable to sequential data. One way to resolve this issue is to utilize LSTM Autoencoder to get compressed fixed vector representation and apply resampling methods on this compressed fixed vector representation. We plan to try out this method as future work. The second set of strategies is related to modifications or extensions to the loss function, such as weighted cross-entropy loss. Several experiments done with the weighted cross-entropy loss demonstrated no noticeable improvement as well. In the end, a data augmentation method can be directly applied to raw sequences before any preprocessing is applied. The technique was used solely to boost the number of benign examples in the training set. This method starts by splitting each input sequence that belongs to the particular class into two chunks; 200 length at the beginning and 800 length at the end. After that, new instances are created by appending the first chunk to a second chunk from another sample. This method generates variability in the newly created instances while retaining most of the local structure in the original sequences. The malicious opcode sequences are mostly preserved inside the local structures. This data augmentation method can be considered similar to the sentence or text piece shuffling method used in some NLP tasks as a method for data augmentation [19]. The results of the experiments that include this data augmentation strategy will be analyzed under the title artificially balanced dataset classification.

3.3 Natural dataset collection

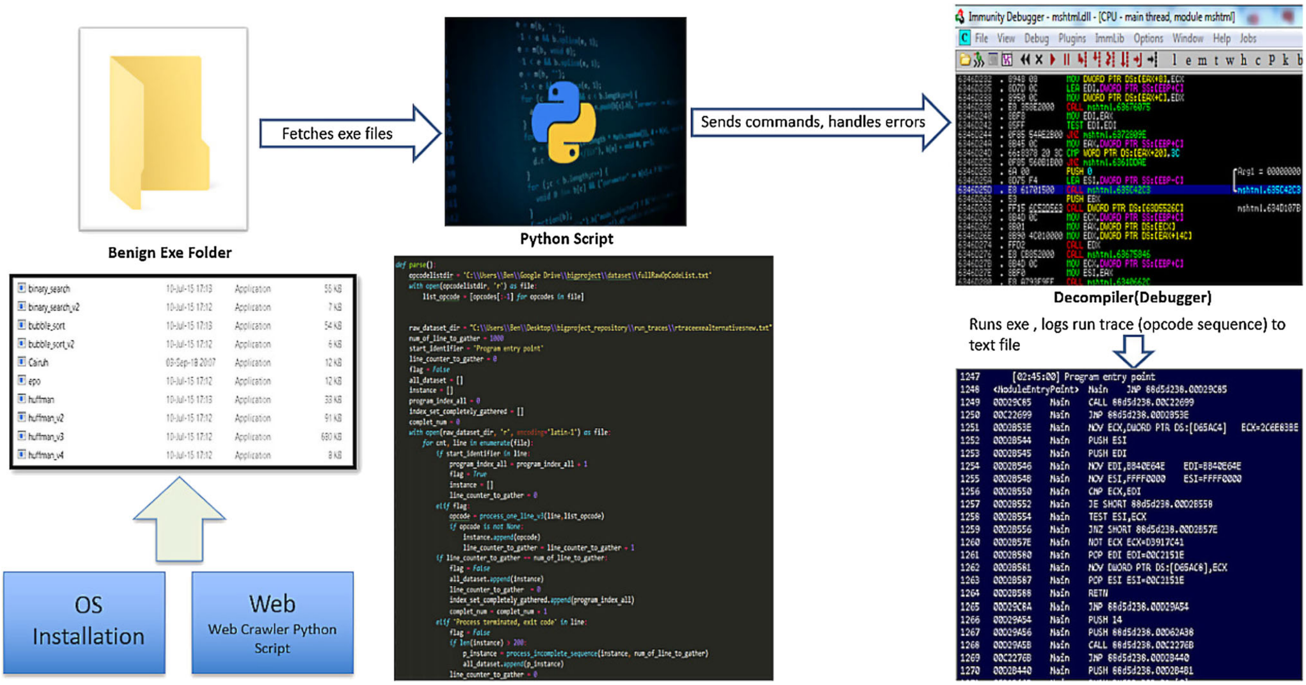
In the end, we decided to gather additional benign instances to assess the capabilities of our approach more precisely, especially when it comes to recognizing the benign behavior of executables by exposing our approach to a larger number of benign samples. This section describes the data collection process, and the related experiment results will be covered under the title naturally balanced dataset classification.

The data collection pipeline starts with gathering a large number of benign legitimate software executables. Executable in this context refers to 32-bit Windows PE format. Unlike in the case of malware for which we have virusTotal.com or virusShare.com as repositories, there is no centralized benign software repositories or benign software corpus. We designated several sources to provide us with reliable, clean benign software samples. These sources are listed as follows: Fresh 32-bit Windows 10 installation in a virtual machine, filehippo.com, softonic.com, and chocolatey package manager for Windows. In order to extract benign executables from the aforementioned website sources, we first developed a web crawler. Secondly, for the chocolatey package manager, several PowerShell scripts were developed. Finally, for Windows 10 installation, a C++ program that searches for executables in all directories recursively was deployed. These three methods, in the end, after some additional filtering, yielded about 10431 executables.

The second stage of the data collection pipeline involves feeding the resultant exes to specialized reverse engineering debugger software, in our case called Immunity Debugger. These debugger programs are used to analyze target software when the source code is not available. In particular, Immunity Debugger also comes with several plugins that help to fend off anti-debugging and anti-virtualization efforts. This analysis, besides many other features, includes disassembling runtime trace. Finally, feeding all 10431 exes manually using GUI would take an immense amount of time and effort. Thus, to reduce the time and effort, an automation script was designed to feed exes one by one, send necessary commands to the debugger, handle errors from the debugger, and obtain the output. Since API functionality for Immunity Debugger was regarded as inadequate for this task, the GUI automation strategy was embraced. A GUI automation script that primarily interacts with the command line interface of Immunity Debugger was implemented. In the end, the automation script generated an opcode sequence for 6405 new benign samples. Note that noticeable attrition occurs between the number of input exes to the debugger and the number of final output samples. According to [6], the attrition can be attributed to missing DLL, lock-outs, and forced reboots. Moreover, a subset of all benign proprietary software may have applied strategies to resist debugging in order to prevent reverse engineering efforts. More sophisticated reverse engineering tools may resolve these issues. Finally, a bespoke parser processed each runtime trace and saved clean number-coded opcode sequences in a CSV file. The data collection process is illustrated in Fig. 2.

3.4 Dataset preprocessing and embeddings

The preprocessing consists of two stages: Tokenization and opcode embedding generation. Before the opcode embedding generation stage by the technique called Word2Vec, for the tokenization stage, we applied some modifications to each raw opcode sequence to facilitate prediction. The first modification was made to eliminate synonym opcodes present in x86 assembly language, such as JNE, JNZ pair. The second modification involved fixing the extremely rare and out of vocabulary opcodes issue. Some of the opcodes are rarely encountered in the training set. Moreover, some of the unseen opcodes absent in the dictionary might be encountered in new test samples. Both issues are detrimental to the Word2Vec technique used to generate opcode embedding vectors; since the Word2Vec technique assumes an established universal dictionary for all training or test samples possible. The presence of unseen opcodes breaks this assumption. Secondly, Word2Vec is likely to produce embedding vectors of lesser quality for rarer opcodes. To mitigate this issue, we took two precautions: First, all opcodes that appear less than three times in all training set was assigned to the single special token and second, unseen opcodes in external test samples, inferred to be rare overall, were assigned to this same token before


Benign Exe Folder
Fetches exe files
Python Script
Sends commands, handles errors
Immunity Debugger - mshtml.dll - [CPU - main thread, module mshtml]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b
Decompiler(Debugger)
Runs exe , logs run trace (opcode sequence) to text file
OS Installation
Web
Web Crawler Python Script

binary_search	10-Jul-15 17:13	Application	55 KB
binary_search_v2	10-Jul-15 17:12	Application	7 KB
bubble_sort	10-Jul-15 17:13	Application	54 KB
bubble_sort_v2	10-Jul-15 17:12	Application	6 KB
Cairuh	03-Sep-18 20:07	Application	12 KB
epo	10-Jul-15 17:12	Application	12 KB
huffman	10-Jul-15 17:13	Application	33 KB
huffman_v2	10-Jul-15 17:12	Application	91 KB
huffman_v3	10-Jul-15 17:12	Application	680 KB
huffman_v4	10-Jul-15 17:12	Application	8 KB

```python
def parse():
    opcodelistdir = 'C:\\Users\\Ben\\Google Drive\\bigproject\\dataset\\fullRawOpCodeList.txt'
    with open(opcodelistdir, 'r') as file:
        list_opcode = [opcodes[:-1] for opcodes in file]


    raw_dataset_dir = "C:\\Users\\Ben\\Desktop\\bigproject_repository\\run_traces\\rtraceexealternativesnew.txt"
    num_of_line_to_gather = 1000
    start_identifier = "Program entry point"
    line_counter_to_gather = 0
    flag = False
    all_dataset = []
    instance = []
    program_index_all = 0
    index_set_completely_gathered = []
    complet_num = 0
    with open(raw_dataset_dir, 'r', encoding='latin-1') as file:
        for cnt, line in enumerate(file):
            if start_identifier in line:
                program_index_all = program_index_all + 1
                flag = True
                instance = []
                line_counter_to_gather = 0
            elif flag:
                opcode = process_one_line_v3(line,list_opcode)
                if opcode is not None:
                    instance.append(opcode)
                    line_counter_to_gather = line_counter_to_gather + 1
            if line_counter_to_gather == num_of_line_to_gather:
                flag = False
                all_dataset.append(instance)
                line_counter_to_gather  = 0
                index_set_completely_gathered.append(program_index_all)
                complet_num = complet_num + 1
            elif 'Process terminated, exit code' in line:
                flag = False
                if len(instance) > 200:
                    p_instance = process_incomplete_sequence(instance, num_of_line_to_gather)
                    all_dataset.append(p_instance)
                line_counter_to_gather = 0
```

```
63460232  . 8948 08        MOV DWORD PTR DS:[EAX+8],ECX
63460235  . 8D7D 0C        LEA EDI,DWORD PTR SS:[EBP+C]
63460238  . 8950 0C        MOV DWORD PTR DS:[EAX+C],EDX
6346023B  . E8 35BE2000    CALL mshtml.63676075
63460240  . 8BF8           MOV EDI,EAX
63460242  . 85FF           TEST EDI,EDI
63460244  . 0F85 54AE2B00  JNZ mshtml.6372809E
6346024A  . 8B45 0C        MOV EAX,DWORD PTR SS:[EBP+C]
6346024D  . 66:8378 20 3C  CMP WORD PTR DS:[EAX+20],3C
63460252  . 0F85 560B1B00  JNZ mshtml.6361DDAE
63460258  . 6A 00          PUSH 0                              ┌Arg1 = 00000000
6346025A  . 8D75 F4        LEA ESI,DWORD PTR SS:[EBP-C]        │
6346025D  . E8 61701500    CALL mshtml.635C42C3                └mshtml.635C42C3
63460262  . 53             PUSH EBX
63460263  . FF15 6C520563  CALL DWORD PTR DS:[6305526C]         mshtml.634D107B
63460269  . 8B4D 0C        MOV ECX,DWORD PTR SS:[EBP+C]
6346026C  . 8B01           MOV EAX,DWORD PTR DS:[ECX]
6346026E  . 8B90 4C010000  MOV EDX,DWORD PTR DS:[EAX+14C]
63460274  . FFD2           CALL EDX
63460276  . E8 CB852000    CALL mshtml.63675846
6346027B  . 8B4D 0C        MOV ECX,DWORD PTR SS:[EBP+C]
6346027E  . 8BF0           MOV ESI,EAX
63460280  . E8 A793F9FF    CALL mshtml.6340662C
```

```
1247       [02:45:00] Program entry point
1248    <ModuleEntryPoint>  Main    JMP 68d5d238.00D29C85
1249    00D29C85    Main    CALL 68d5d238.00C22699
1250    00C22699    Main    JMP 68d5d238.00D2853E
1251    00D2853E    Main    MOV ECX,DWORD PTR DS:[D65AC4]   ECX=2C6E63BE
1252    00D28544    Main    PUSH ESI
1253    00D28545    Main    PUSH EDI
1254    00D28546    Main    MOV EDI,BB40E64E    EDI=BB40E64E
1255    00D2854B    Main    MOV ESI,FFFF0000    ESI=FFFF0000
1256    00D28550    Main    CMP ECX,EDI
1257    00D28552    Main    JE SHORT 68d5d238.00D28558
1258    00D28554    Main    TEST ESI,ECX
1259    00D28556    Main    JNZ SHORT 68d5d238.00D2857E
1260    00D2857E    Main    NOT ECX ECX=D3917C41
1261    00D28580    Main    POP EDI EDI=00C2151E
1262    00D28581    Main    MOV DWORD PTR DS:[D65AC8],ECX
1263    00D28587    Main    POP ESI ESI=00C2151E
1264    00D28588    Main    RETN
1265    00D29C8A    Main    JMP 68d5d238.00D29A54
1266    00D29A54    Main    PUSH 14
1267    00D29A56    Main    PUSH 68d5d238.00D62A38
1268    00D29A5B    Main    CALL 68d5d238.00C2276B
1269    00C2276B    Main    JMP 68d5d238.00D2B440
1270    00D2B440    Main    PUSH 68d5d238.00D2B481
```

Fig. 2 Illustration of the data collection pipeline. The first column represents the raw executable collection section. The second column represents a set of python scripts that are used for the automation of debugger and parsing of debugger output. The third column shows snapshots from the debugger software and its output

they were fed to our predictive model for inference. Input opcode embeddings were generated according to the dictionary, which includes the unique rare opcode token. This simple practice is common in natural language processing [11]. Note that this practice could be satisfactory and straightforward, but it may not be ideal for opcodes. Rare opcodes are likely to have a high predictive value. By tuning the negative sampling parameters, Word2Vec learning algorithm can be adjusted to favor rare opcodes more. A negative sampling strategy can be preferred instead of dropping extremely rare opcodes in the training set. Unknown token, in this case, should only be used for unseen opcodes in the test set. To finalize the tokenization stage, a dictionary of tokens was created. The large majority of tokens correspond to a unique opcode. However, tokens are not exactly opcodes. They are abstractions of opcodes in order to facilitate self-supervised learning of statistical patterns in dynamic opcode sequences through the Word2Vec algorithm.

Each token in the dictionary must be encoded or embedded appropriately before it can be fed to any predictive model. One-hot encoding is a common way to implement this encoding. It is simple but poses two problems: First, it is very inefficient space-wise. This inefficiency could mean a considerable increase in computation time and memory usage, especially when it comes to deep learning methods. Secondly, by assigning all opcodes practically the same vector, one hot encoding completely ignores any contextual information that opcodes may carry concerning each other. Better encoding schemes may induce better performance in terms of both computation time and accuracy. As a better encoding scheme, we opted for Word2Vec, noting that it has been beneficial in various NLP tasks [31]. Moreover, the practice that is first utilizing a self-supervised method such as Word2Vec to obtain dense representations of tokens and then fine-tuning those representations with a supervised learning method has recently gained serious traction in the NLP community. Our overall pipeline approximately follows this practice.

Word2Vec model consists of a shallow feed-forward neural network with one hidden layer. In our case, this neural network is trained to optimize the objective likelihood of the surrounding opcodes within a context window conditioned on the center word of that window. This objective likelihood function learns statistical patterns among opcodes within a particular context inside each input sequence. The parameters of the final fully trained network yield an N-dimensional vector for each token. Two particular vectors created by this method tend to be close in Euclidean space, if the corresponding opcodes appear in a similar context. The length of the context window is chosen to be 10 tokens. This is larger than the typical length of context window chosen for natural language. Since the implementation of even simple high-level programming operations such as printing, looping, or conditional blocks may require long opcode sequences, capturing the context of the higher-level operations is only possible with larger windows. In our experiments, the second important parameter N parameter is set to be 64. Note that, in our case, the choice of 64 was made heuristically. An ideal way to determine the optimal value would be doing extensive hyperparameter optimization on this parameter. However, this particular hyperparameter optimization would be costly in terms of computation. For the English language, which has thousands of words, NLP practitioners typically choose values between 300 and 500. For our dataset, which has approximately 300 token dictionary size, intuitively, this dimensionality value should optimally be lower such as between 50 and 100. [40] uses 30 dimensionality value in their supervised embeddings for their 218 size Android opcode dictionary. Based on these arguments, we decided that 64 should correspond to a sweet spot; moreover, several experiments with 100 dimensionality values show no drastic change in performance.

3.5 Feature extraction and classification

Feature extraction was done through deep learning-based approaches that have been demonstrated to be groundbreakingly successful at many tasks involving large amounts of high-dimensional and unstructured data. Deep learning algorithms, which provided an adequate amount of data, can automatically engineer useful features from the data. These features can be determinant alone, or they can be complementary to the handcrafted features produced by experts. More specifically, this property of deep learning algorithms also makes them very handy at malware detection based on predictive modeling, since feature engineering from malware, just like in any other domain, is instrumental to the predictive model's success. In our problem, we have a reasonably large dataset that resembles a large text corpus; therefore, it is sensible that deep learning approaches may perform well in our dataset. This dataset can be expanded considerably in the future by harvesting additional executables. Furthermore, although not exactly the same as natural language, opcode sequences can be considered high-dimensional and unstructured data. One disadvantage of deep learning-based feature extraction is that these features are not directly interpretable. On the other hand, input attribution methods are available for deep learning algorithms. They can provide some level of interpretation, such as which opcodes in the sequence influence the classification result most. A user-friendly library introduced by [37] implements many input attribution methods. As future work, the library can be used to investigate the influence of each opcode on the final model result.

We decided to introduce three distinct deep learning model types to our opcode sequence detection problem. All three model types commonly make use of backpropagation algorithm to update their learnable parameters; however, the types of mathematical operations they utilize to process input are distinct. The three different deep learning algorithms also impose distinct inductive biases, which are the set of assumptions taken a priori. For LSTM-RNN, the specific inductive bias takes the form of sequentiality; the same function is applied to all-time steps recurrently. For CNN, it is spatial invariance, meaning that the same kernel is applied to the different spatial locations. On the other hand, the self-attention operation used in the Transformer architecture maintains a structure closer to fully connected computation and therefore assumes weaker inductive bias. Weaker inductive bias could lead to more expressive models overall. On the other hand, models that assume weaker inductive bias may require more training data. Therefore, in situations where a structural prior can be deduced either from the target task or the data type, using an appropriate version of stronger inductive bias may lead to more accurate and data-efficient models. In this context, data types can be images, short sentences, long text, or opcode sequences. In our case, the a priori universal properties of runtime opcode sequences, unlike more ubiquitous images and text, are not well known and well studied. It is also important to emphasize that these properties might be wholly different for static opcode sequences. For instance, some of these properties might indicate whether recognizing long-range interactions in opcode sequences is useful or not for malware detection. Since all these specific properties are not clear at this point, we decided to assess all three algorithms to gain a better understanding possibly. That being said, discovering specific properties of the opcode sequence is not among the purposes of our experiments. Further research is required to discover and understand the specific, intrinsic properties of runtime opcode sequences.

The next three subsections provide a brief introduction to each deep learning architecture used for feature extraction. As a separate experiment, we also utilized simple mean of opcode embeddings as the main way to extract features for the downstream classification algorithms. Most specifically, each instance that has 1000×64 dimensionality is mean reduced to $M \in \mathcal{R}^{64}$. The vector M is used as the input for the downstream classification algorithms. Mean of embedding classification is also included in Imbalanced Dataset Classification Setting to make comparison with results reported in [6]. Mean of opcode embeddings provides a computationally lightweight alternative.

For the final classification stage, five different machine learning algorithms were utilized. These algorithms are logistic regression, support vector machine, the k-nearest neighbors classifier, the random forest classifier, and gradient boosted trees. The classifiers in this stage operate on the fixed vector representation of the input, which is generated by the upstream deep learning algorithm, as opposed to sequential data. The five algorithms that we assess are not directly applicable to sequential data. The deep learning stage enables the conversion of raw sequences to fixed-size vector features. Usage of these machine learning algorithms provides flexibility when it comes to tuning the whole predictive model for some specific needs. It also brings performance improvement. Moreover, these machine learning algorithms are much less cumbersome than deep learning methods when it comes to hyperparameter optimization, tuning, and retraining, if necessary. In our experiments, parameters of machine learning are optimized with fivefold cross-validation. The cross-validation is performed on output embedding data, not on raw sequences. On the other hand, performing direct hyperparameter optimization on deep learning algorithms can be incredibly expensive in terms of computation. Using more nimble machine learning methods at the end of the pipeline makes the whole pipeline more robust. High performance is not strictly tied to a specific hyperparameter set of deep learning algorithms, which could be hard to find. The performance gap caused by non-optimal hyperparameters could be filled with machine learning algorithms in the end. Usage of machine learning algorithms decreases the reliance on rigorous hyperparameter search. Our experimental results also confirm this argument. Machine learning algorithms, especially nonlinear ones, provide an accuracy boost. Similar to the three deep learning algorithms, the five machine learning algorithms have a set of distinct properties, such as the type of inductive bias, that make them worthwhile to investigate separately.

3.5.1 Convolutional neural network

Convolutional neural network (CNN), in its current form, which is introduced by [16] has been shown to be very successful at tasks that involve understanding and inference from visual imagery. [12] demonstrated that CNN could be applied successfully to the sentence classification task, despite the irrelevance of the sentence classification task to the image-related tasks. Inspired from [12], we decided to use a CNN model in our opcode sequence classification task. A convolutional neural network typically consists of several building blocks and smaller-scale operations between them. First, some of the common building blocks include 2D convolution block, which performs 2D convolution operation on its 2D input, max-pooling block, which performs downsampling operation to reduce input size and summarize the information. Finally, the fully connected layer gathers features from upstream convolution layers and performs an affine transformation on its input. Second, intermediate smaller operations include nonlinearity function, dropout, and batch normalization. Given our data's unique nature, instead of using well-known architectures, we decided to use our two custom small and larger CNN architectures. These architectures are illustrated in Figs. 3 and 4.

3.5.2 LSTM recurrent neural network

Recurrent neural network (RNN) is a special type of deep neural network that contains internal memory or state. Having a memory makes RNN well-suited to process and learn from sequences. Their internal memory can extract and hold temporal information hidden in sequences. LSTM (long short-term memory) refers to the particular architecture of a RNN's main cell. LSTM architecture was developed by [10] to tackle the vanishing gradient problem that prevents vanilla RNN from being able to handle long-term dependencies in a sequence. Since then, by the agency of more powerful computation, LSTM has become the most successful variant of RNN, which achieved tremendous successes in speech, handwriting recognition tasks.

An LSTM cell, at each time steps, performs several operations on input value at time or position t which is x_t, previous cell state memory C_{t-1}, previous output h_{t-1} to update the cell state and compute an output at time t, θ denotes model parameters; g denotes generic function to highlight arguments. More specifically, for cell state:

$$C_t = g(C_{t-1}, x_t, h_{t-1}; \theta) \tag{1}$$

Cell state acts like long-term memory which typically does not see a drastic change in a single time step. On the other hand, hidden state h_t which is also the output, change more drastically at every step, acts as a short-term memory:

$$h_t = f(C_t, x_t, h_{t-1}; \theta) \tag{2}$$

For vanilla LSTM if we unroll these equations:

$$f_t = \sigma(W_f[C_{t-1}, h_{t-1}, x_t] + b_f) \tag{3a}$$

$$i_t = \sigma(W_i[C_{t-1}, h_{t-1}, x_t] + b_i) \tag{3b}$$

$$o_t = \sigma(W_o[C_{t-1}, h_{t-1}, x_t] + b_o) \tag{3c}$$

$$u_t = \tanh(W_u[h_{t-1}, x_t] + b_u) \tag{3d}$$

$$C_t = f_t * C_{t-1} + i_t * u_t \tag{3e}$$

$$h_t = o_t * \tanh(C_t) \tag{3f}$$

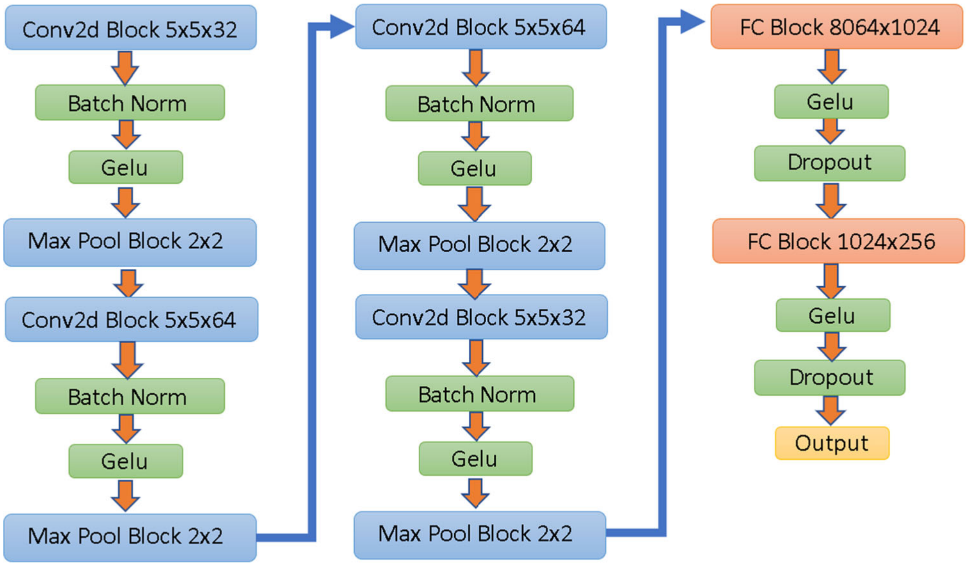


Fig. 3 Illustration of the larger CNN architecture. In conv2d block, the first two dimensions are the size of a convolution filter, which is a rank two matrix. The third is the number of filters in a block. In max pool blocks, the numbers define the downsampling rate of spatial dimensions. In FC (fully connected) blocks, it is the size of the linear transformation matrix

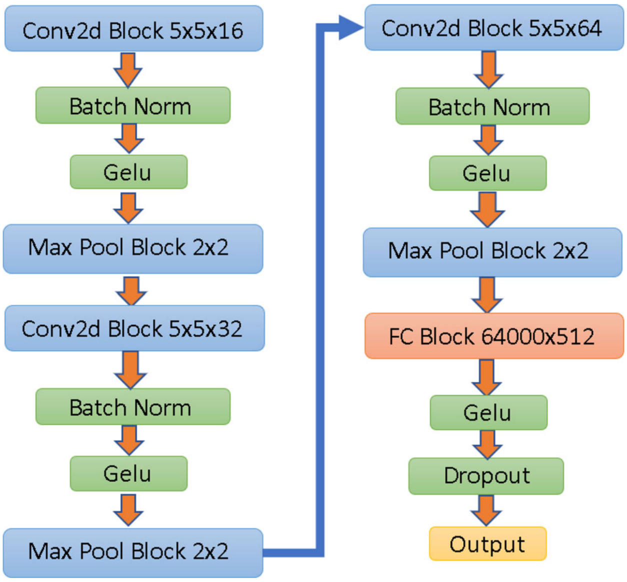


Fig. 4 Illustration of the smaller CNN architecture

To further increase robustness for long sequence learning, we employed a special variety of LSTM cell called phased LSTM cell which was introduced by [22]. Phased LSTM cell contains some extensions which makes it more efficient at handling very long sequences compared to vanilla LSTM. More specifically, phased LSTM introduces a new time gate k_t. This time, the gate is governed by three learnable parameters and it ensures that C_t and h_t get an update once in while, not at each time step. This mimics the effect of time truncated backpropagation and increases robustness against very long sequences by blocking gradient from flowing too long. Imposing a shorter time range on backpropagation mitigates vanishing and exploding gradient issue. Truncated backpropagation through time achieves the robustness by splitting the whole sequence into shorter subsequences.

The LSTM architecture used in the experiments is bidirectional; the input is processed in both a forward and backward manner. The bidirectional architecture produces a pair of output vector for each time step. The bidirectionality could increase robustness for very long sequences by providing additional context from the backward direction. Moreover, [25] reports slightly higher accuracy with Bi-LSTM compared to regular LSTM. Finally, the self-attention mechanism processes these output vectors to yield the final single context vector. The exact architecture is described in Fig. 5.

3.5.3 Transformer architecture

The Transformer architecture, which was introduced by [33], was designed to overcome the shortcomings of recurrent and convolutional models. More specifically, it is emphasized in the paper that despite all the proposed modifications, the sequential nature of recurrent models prevents full parallel computation within the training dataset. Incompatibility with parallel computation leads to diminished robustness and efficiency, especially in a long sequence regime. Additionally, in convolutional models, the number of operations required to connect two positions in sequence grows linearly with respect to the distance between these positions. It is argued that linear growth here is suboptimal. On the other hand, Transformer architecture relies primarily on the self-attention mechanism, and it does not utilize convolutional and recurrent computation whatsoever. In self-attention, the aforementioned number of operations this time is reduced to a constant rate. Furthermore, self-attention, unlike sequential computation, is

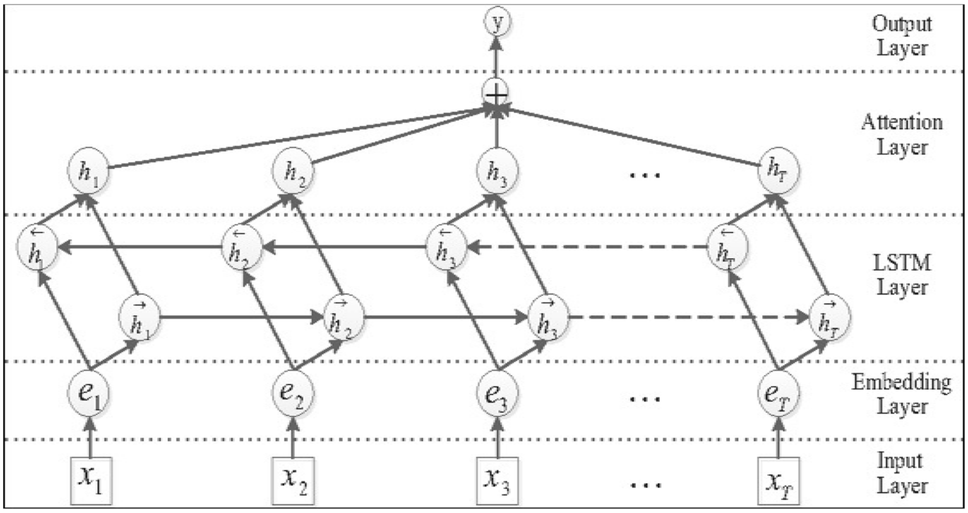


Fig. 5 Illustration of LSTM-RNN architecture used in experiments taken from [41]. In this image, the layer outputs are annotated. All layers shown here have their own parameter set. In our case, the parameters of the embedding layer are trained first and separately from the rest of the architecture in an unsupervised manner by utilizing Word2Vec technique

fully parallelizable. The downside of the Transformer architecture is that memory requirement is quadratic with respect to the input sequence length. The quadratic memory complexity might cause limitations when it comes to very long opcode sequences. An extension to the Transformer architecture called the Reformer has recently been proposed to address the huge memory requirement caused by very long sequences [13]. An alternative to the Reformer, the Longformer architecture, which was introduced by [4] also aims to tackle memory-efficient processing of extremely long sequences. With these two extensions, much longer opcode sequences can be processed. Therefore, it could be possible to process an entire code sequence of an executable at once with this extension. The Transformer-based architectures may lead to fast and powerful automatic malware detection systems.

Secondly, closely related to the earlier discussion about inductive bias, the Transformer architecture follows a somewhat different paradigm than LSTM-RNN and CNN. Specifically, the architecture assumes, by virtue of self-attention, a more relaxed inductive bias than LSTM-RNN and CNN. As has been emphasized before, a more relaxed form of inductive bias can be advantageous in data-rich situations. It can also be a better choice when prior knowledge about the data is limited. As accentuated earlier in the first subsection of this section, the task of processing opcode sequence satisfies both of these conditions. Therefore, the Transformer architecture is a viable option.

The Transformer architecture in total consists of encoder and decoder parts. Encoder–decoder paradigm is common to almost all competitive models that aim to solve the sequence to sequence tasks. Since we are only concerned about the classification task, we omit the decoder section entirely. An encoded context embedding vector is used in the classification layer.

The encoder part as a whole is composed of several computation blocks stacked on top of each other. Each computation block consists of two sub-parts which output $\mathrm{LayerNorm}(x + \mathrm{subLayer}(x))$. Here, sublayer can be either a dense feed-forward or attention layer. Layer normalization is applied to the output.

The Transformer architecture mainly utilizes an offshoot of self-attention, which is called scaled dot-product attention. Mathematically, for the encoder, it is defined as follows:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4}$$

In Eq. 4, Q, K, and V refer to query, key, and value, respectively. These three have distinct meanings in the context of the decoder. However, in the context of the encoder, they are the same. The scaled dot-product attention aims to learn a similarity function between input tokens. The similarity function is used to construct a context vector that indicates which part of the input the network should attend to. Furthermore, the Transformer architecture, in order to boost its expressive power, employs the scaled dot-product attention (Eq. 4) a number of times in parallel with distinct learnable linear projection matrices. This operation is called multi-head attention. Each head learns distinct weight matrices and representation. In the end, each head's output is concatenated and then again projected linearly to obtain the final output. Multi-head attention is described as follows:

$$Q_i = XW_i^Q, K_i = XW_i^K, V_i = XW_i^V, i = 1, \ldots, N \tag{5}$$

$$\text{where head}_i = \text{Attention}(Q_i, K_i, V_i) \tag{6}$$

$$\text{MultiHeadAttention}(X) = [\text{Head}_1, \ldots, \text{Head}_N]W^0 \tag{7}$$

In Eqs. 5, 6 and 7, X refers to input sequence after embedding mapping and W^Q, W^K, W^V and W^0 are parameter matrices to be learned. The formulation given above is only valid for the encoder part.

Considering that our task is more uncomplicated than large-scale NLP applications and the dataset is much smaller than a typical NLP corpus, we used the reduced version of the Transformer architecture. In this version, only two layers are used, and the number of heads is 4. The original architecture is significantly larger. The architecture can be expanded with respect to the size opcode sequence dataset. As an addition to the regular Transformer encoder architecture above, we included a final dense layer to the end in order to reduce the dimensionality of output encoding. Epoch number and learning rate were again adjusted manually during the whole training process.

3.6 A practical illustration of the methodology

In a more practical setting, the methodology should be considered under two different stages; training and testing or deployment stage. First, in the training stage, a training raw executable dataset needs to be gathered. Several sources and strategies to gather raw executable are covered in Sect. 3.3. Malware examples can be gathered from various repositories to ensure variety. It is highly advisable to ensure that the training dataset stays balanced in terms of class distribution. Second, a debugger tool, such as Immunity Debugger, processes raw executable dataset to generate dynamic opcode sequence dataset. In this step, alternative tools can be considered. Conversion to dynamic opcodes largely removes the effects of the code obfuscation. Dynamic opcode sequences undergo several data preprocessing as outlined in Sect. 3.4. The data preprocessing steps can be implemented as a Python program. After that, Word2Vec embeddings can be trained with all of the data, including unlabeled data using the recommended hyperparameters. For the feature extraction step, one of the three deep learning algorithms or alternatively mean of embeddings-based classification needs to be chosen. The choice depends on the factors such as available computation resources, target performance, and training dataset size. For the classification step, this time one of the five machine learning algorithms needs to be chosen. The selection criteria are similar to deep learning algorithms. The results section provides an example comparison between all the algorithms with respect to the dataset used in this work. A hyperparameter optimization on a separate validation set may also be utilized to select the best algorithm and its respective parameters.

The second stage consists of the deployment and maintenance of the trained model. The fully trained model can preferably be used in conjunction with a signature-based malware detector within a malware detection system to accelerate the overall rate of inference. The incoming sample should be first quarantined inside a secure sandbox. Several different sandbox implementations exist for malware analysis. Then, inside the sandbox, the dynamic opcode sequence belonging to the sample can be collected. After that, the output sequence should go through the data preprocessing pipeline. Finally, the model inference can be obtained. Periodically, opcode embeddings can be fine-tuned with the number of newly obtained opcode sequences without ground truth labels. When the ground truth labels are obtained through expert analysis, the machine learning algorithm at the top can be retrained easily with measly computation power and requirements. Less frequently, the deep learning model can also be retrained. The deep learning model retraining can be less desirable due to the much higher computation time and cost associated with it.

4 Results

All experiment setups follow the process illustrated in Fig. 6. Before the main classification experiments, a detailed analysis of opcode Word2Vec embeddings is presented in the first subsection. After that, all the experiments in this section are presented in three groups. The groups are formed with respect to the properties of the source dataset type these experiments utilize. In all of these sections, the experiments were conducted by utilizing Python library TensorFlow-GPU 1.12.0 for the implementation of deep learning models. Scikit-Learn library is used for the implementation of the machine learning models in the classification layer. Furthermore, Pandas, Numpy, and Scipy libraries were utilized for auxiliary data loading, manipulation, and computation operations. In all experiments, the experiments were done on the single fold with a 7/3 train-test split. The results are assessed with respect to four metrics: accuracy, Matthew's correlation coefficient (MCC), precision, and recall.

4.1 Opcode Word2Vec embeddings

Before the actual deep learning training and classification part of experiments, our sequence-based classification process involves unsupervised learning of dense vector

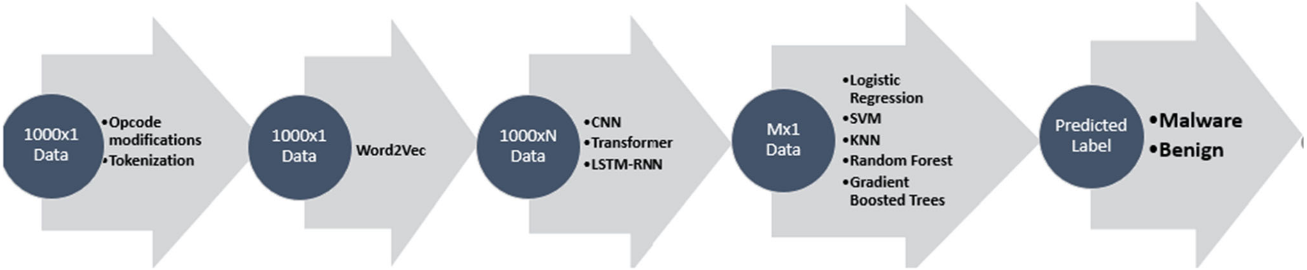


Fig. 6 Illustration of the predictive modeling pipeline. N is the input embedding dimensionality, which is 64. M is the output embedding dimensionality, which can be 128, 256 or 512 depending on the deep learning model type. This illustration does not include mean of embeddings classification, for which both M and N are 64

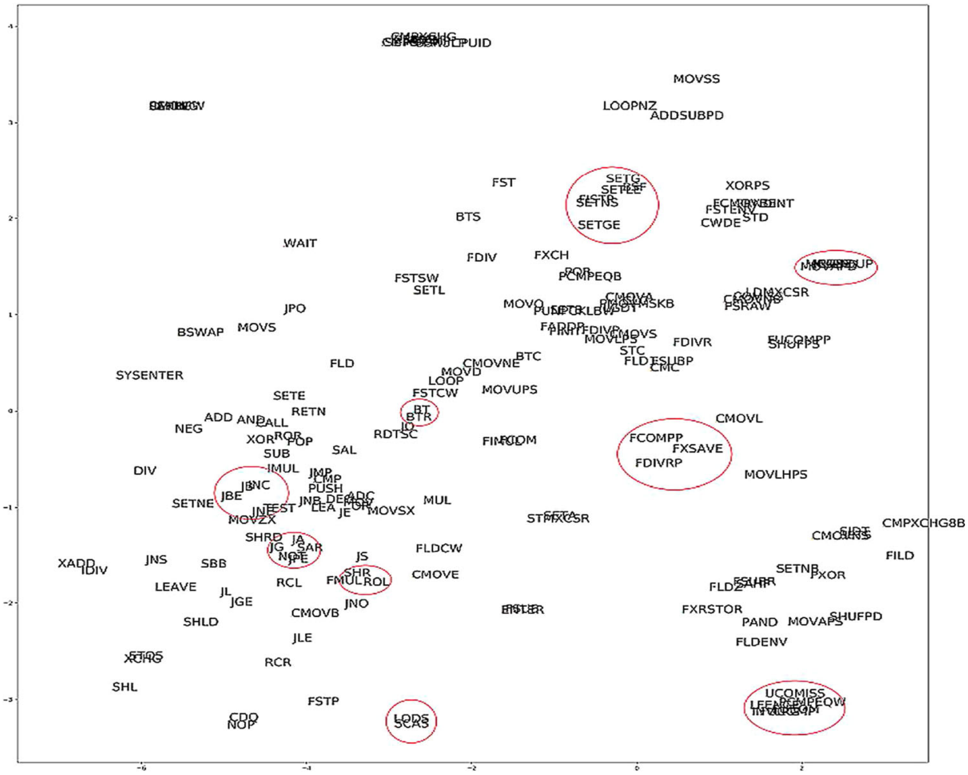


Fig. 7 Illustration of the opcode embedding vectors when they are reduced to two dimensions using the TSNE algorithm. Their original dimensionality is 64. Significant opcode clusters are annotated with red circles

representation of each opcodes using the Word2Vec algorithm. The Word2Vec embeddings matrix was generated before the actual training of deep learning models by using Python NLP library Gensim [30]. During the deep learning layer training, they were loaded as a fixed pre-trained constant lookup matrix that maps opcode tokens to their dense vector representation.

Figure 7 demonstrates the visualization of Word2Vec vector embeddings with their associated opcodes. Some of the opcodes that have similar purposes and functionalities cluster close to each other. For example, LODS and SCAS load and scan string instructions. SETLE, SETG, SETNS, and SETGE are instructions that set byte according to various different conditions. BT and BTR are bit test and reset instructions. JNB, JB, JNC, and JNE are all conditional jumps. SHR and ROL, which are shifting and rotating instructions, are also close to each other. In addition, in general, frequent and common instructions are clustered the left-down section of figure 4.1. Less frequent, more specialized instructions, such as floating-point instructions, are dispersed around the outside of the area that hosts common instructions. Common instructions are approximately located in the third quadrant. As you may also see, some of the floating-point instructions are also

Table 1 Results of the imbalanced classification setting

	Accuracy	MCC	Precision	Recall
CNN				
Logistic regression	0.9873	0.7981	0.8531	0.7587
Support vector machine	0.9867	0.7753	0.9416	0.6482
Random forest	0.9722	0.6521	0.5652	0.7839
Gradient boosted trees	0.9882	0.8079	0.8963	0.7386
K-nearest neighbors	0.9885	0.8131	0.9074	0.7386
LSTM				
Logistic regression	0.9867	0.7805	0.8810	0.7035
Support vector machine	0.9717	0.6409	0.5608	0.7638
Random forest	0.9846	0.7626	0.7823	0.7587
Gradient boosted trees	0.9764	0.6793	0.6255	0.7638
K-nearest neighbors	0.9853	0.7544	0.8741	0.6633
Mean of embeddings				
Logistic regression	0.864	0.3807	0.2	0.8812
Support vector machine	0.8595	0.3686	0.1928	0.8675
Random forest	0.9841	0.7481	0.8867	0.6438
Gradient boosted trees	0.9847	0.7607	0.8848	0.6666
K-nearest neighbors	0.9848	0.7706	0.8779	0.6894

clustered into several subgroups. For instance, it may be hard to see from the image, but MOVAPD and MOVDDUP, which are two different instructions for moving floating-point values, are also very close. Another hard to discern cluster is located at the fourth quadrant, which contains FUCOMP, FUCOM, UCOMISS instructions, which are all related to comparing floating-point data.

Note that it is implausible to infer the same significance between opcodes clustered together since, for some of the opcodes, their vectors might not be too accurate due to the rarity of their corresponding opcodes in the dataset. This can be mitigated by employing the subsampling of frequent opcodes for future works. Moreover, the well-known word vector representations in NLP typically are trained from billions of tokens. In our case, the opcode embeddings were trained from approximately 20M tokens. A larger opcode sequence corpus will lead to more accurate vectors. The critical advantage is that for Word2Vec the opcode sequences do not need to be labeled at all. Therefore, each opcode vector can become gradually more accurate by merely processing more unlabeled data, which is cheaper to obtain.

4.2 Imbalanced dataset classification setting

The imbalanced dataset refers to the original state of the dataset taken from [6] before the application of dataset augmentation methods and inclusion of the new benign dataset introduced. The Transformer model also was unintroduced at this point. Moreover, in this setting, the smaller CNN architecture and Bidirectional RNN with vanilla LSTM cells were utilized.

You may refer to Table 1 for the results of this section. The CNN model overall gives slightly better results than the LSTM model, and for both models, accuracy is very high. CNN may have gained a slight edge over LSTM in this setting due to its better robustness. LSTM model, in general, exhibited less adequate robustness than CNN when it comes to training stability. The lack of robustness may impact LSTM more severely than CNN when it comes to the imbalanced setting. CNN is faster in terms of training and inference time. On the other hand, CNN incurs a larger model size than LSTM. In memory constraint devices, LSTM may be a better choice. The k-nearest neighbors, which is in general quite simple, fast, and robust algorithm, performs quite well on the top of both LSTM and CNN. Gradient boosted trees and random forest, which may require more extensive tuning, especially if dataset class imbalance is present, perform somewhat worse than the k-nearest neighbors. Gradient boosted trees also takes significantly longer to fit. Finally, despite being linear classifiers, logistic regression and support vector machine perform quite satisfactorily.

Mean of embeddings largely performs worse than CNN and LSTM. On the other hand, its performance is still respectable despite its simplicity. Different from CNN and LSTM, nonlinear classifiers, such as gradient boosted trees, random forest, and the k-nearest neighbors algorithm, seem to make a huge difference for mean of embeddings. Evidently, nonlinear interactions need to be modeled for high classification performance. CNN and LSTM can handle nonlinear interactions. For mean of embeddings, nonlinear classifiers can model nonlinear interactions and thus achieve better performance. Additionally, due to underlying Word2Vec, mean of embeddings can utilize unlabeled data making it potentially more powerful and practical. Note that, [6] using HMM reports %11 benign class accuracy, %99 malware class accuracy which amounts to approximately 0.11 MCC Score. Results from the mean of embeddings classification are clearly an improvement compared to the sequence-based classification method outlined in [6]. On the other hand, [6] also reports %85 benign class accuracy, %99 malware class accuracy, and combined %98.4 using random forest on 1-gram representations. We replicated this experiment and obtained approximately the same performance. This performance is almost equal to the results achieved by the k-nearest neighbors algorithm and gradient boosted trees classification on the mean of embeddings features in terms of MCC score. Sequential learning with CNN and the k-nearest neighbors algorithm and gradient boosted trees achieves

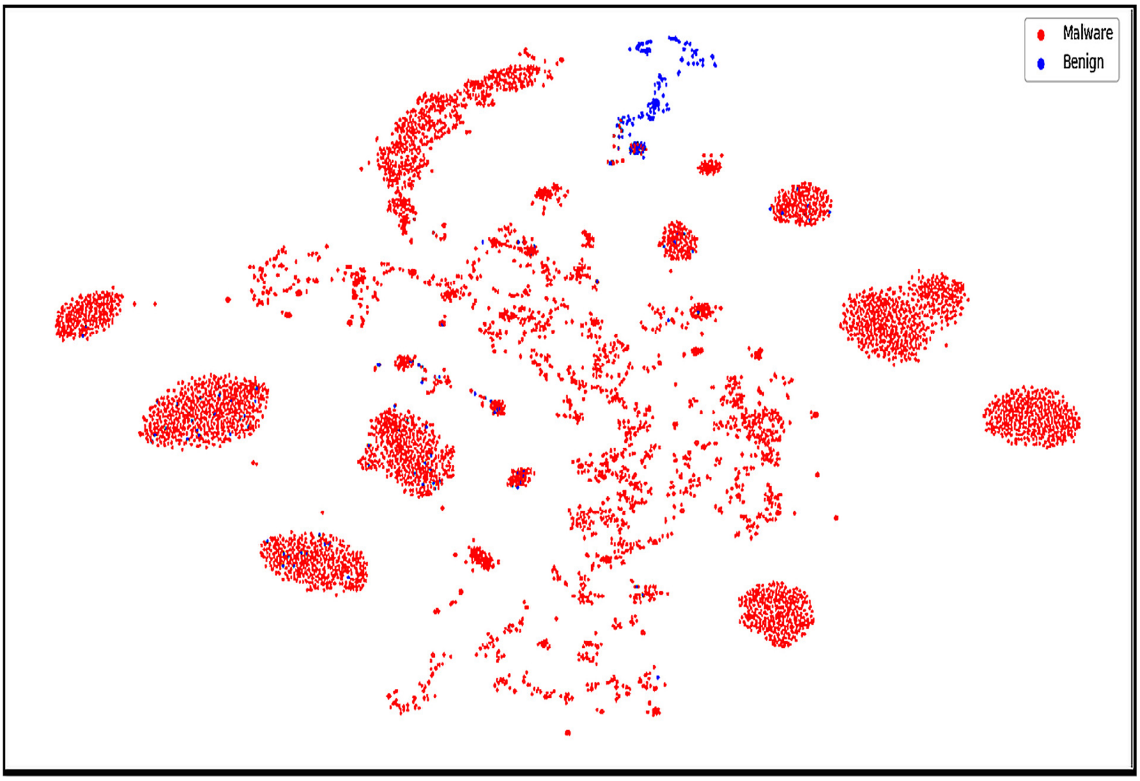


Fig. 8 TSNE visualization of the final context feature vector produced by the CNN model on imbalanced training setting. TSNE reduces the dimensionality from 512 to 2. This illustration only depicts the training set

more superior performance than both in terms of MCC score. Overall good performance of simple 1-gram representations, in this case, can be attributed to the quality of dynamic opcode features, since [40] reports much worse detection performance when they use N-gram static android opcodes features compared to full sequential learning. Likewise, for dynamic opcode features the performance can be boosted by utilizing sequential learning.

Finally, Fig. 8 shows the TSNE (t-distributed stochastic neighbor embedding) visualization of the final output context vector produced by the CNN model from the training data. TSNE, which was introduced by [8], is an algorithm that probabilistically projects each high-dimensional vector to a lower-dimensional vector such that two instances close to each other in the high dimension with respect to Euclidean distance are likely to be close in the low dimension as well. In the figure, the position of clusters and their separation can be easily modeled by the classification layer. Therefore, this visualization gives us valuable insight into how the preceding deep learning model models the data. For instance, first, some of the malware instances produced their own cluster. We have not verified this, but these clusters may correspond to different malware types. However, TSNE fails to generate a clear cluster for a substantial number of malware instances. These instances instead form a sparse giant cloud in the middle. The formation of this cloud could be explained by the fact that not all malware instances can be categorized unambiguously. As stated by [34], malware instances are characterized on the basis of their primary function, and many malware instances are multi-functional primary or secondary. Therefore, for some malware instances, clear categorization may not exist. Second, except for some contested instances from both sides, the benign and

Table 2 Results of the artificially balanced classification setting

	Accuracy	MCC	Precision	Recall
CNN				
Logistic regression	0.9899	0.8238	0.8722	0.7899
Support vector machine	0.9885	0.8158	0.8837	0.7638
Random forest	0.9892	0.8281	0.9096	0.7587
Gradient boosted trees	0.9894	0.8285	0.9107	0.7638
K-nearest neighbors	0.9884	0.8153	0.8540	0.7939
LSTM				
Logistic regression	0.9890	0.8250	0.8857	0.7758
Support vector machine	0.9884	0.8071	0.9571	0.6934
Random forest	0.9894	0.8285	0.9101	0.7638
Gradient boosted trees	0.9890	0.8239	0.8947	0.7688
K-nearest neighbors	0.9892	0.8259	0.9047	0.7638

malware clusters are clearly separated. Interestingly, the model, in general, is quite good at recognizing the large majority of the correct malware instances. However, for some of the benign instances, it makes mistakes. The disparity here can be explained by the fact the amount of benign instances is simply not adequate for proper learning. Therefore, the model is not able to learn the general data distribution for the benign class. Furthermore, it could be difficult to discern it from the figure due to the low number of benign instances. However, the correct benign instances do not establish many apparent tight clusters like some malware instances establish. Lack of clear clusters may indicate that benign data distribution shows substantial intraclass variance, probably more than the intraclass variance of the malware class. That being said, a richer dataset of benign instances would give us a clearer idea about this matter.

4.3 Artificially balanced classification setting

The set of experiments in this section utilizes the same methods as an imbalanced dataset classification setting. The primary distinction is that in this set of experiments, the training dataset was bolstered by the data augmentation method explained earlier. Note that the test dataset was not bolstered similarly. The test dataset itself only consists of legitimate samples, not artificially crafted samples in any way.

You may refer to Table 2 to see the result of this section. Recall scores in this section are better than the previous imbalanced training setting, and accordingly, MCC values are also better in general. This improvement means in this experiment, better performance at benign class does not completely devastate classification performance at malware class across all classification algorithms. Better performance in this section can be attributed to more stable training of the deep learning models. Artificial balancing probably mimicked the effect of balanced mini-batch sampling, which is a common method to tackle the class imbalance issue in deep learning. The balanced mini-batches provide more stable training of the deep learning models. Our data augmentation method also provides some level of variability, which is something balanced mini-batch sampling cannot provide. Moreover, data augmentation has been repeatedly proven to be effective in supervised deep learning.[24] Finally, the CNN and LSTM models, as well as all their respective machine learning models, perform almost equally. Random forest and gradient boosted trees, which are in general more powerful than the other algorithms, gain a slight edge in this setting due to the more balanced dataset. The discussion made in the previous section regarding the model size and training time efficiency of CNN and LSTM applies here as well. Memory constraint environment would better suit LSTM due to the smaller model size. On the other hand, assuming GPU resources are abundant, CNN provides faster training and inference. In terms of machine learning-based classification algorithms, gradient boosted trees, random forest, and the k-nearest neighbors algorithm bring slight improvement over linear classifiers. However, in this case, especially the k-nearest neighbors algorithm and logistic regression can be preferred over all others due to their computational simplicity.

Table 3 Results of the naturally balanced classification setting

	Accuracy	MCC	Precision	Recall
CNN				
Logistic regression	0.9787	0.9479	0.9800	0.9459
Support vector machine	0.9775	0.9452	0.9679	0.9539
Random forest	0.9785	0.9476	0.9782	0.9469
Gradient boosted trees	0.9755	0.9401	0.9802	0.9342
K-nearest neighbors		0.9482	0.9791	0.9469
0.9788				
LSTM				
Logistic regression	0.9791	0.9492	0.9748	0.9526
Support vector machine	0.9789	0.9485	0.9744	0.9522
Random forest	0.9789	0.9507	0.9783	0.9513
Gradient boosted trees	0.9797	0.9504	0.9770	0.9522
K-nearest neighbors	0.9795	0.9501	0.9762	0.9526
Transformer				
Logistic regression	0.9787	0.9479	0.9748	0.9526
Support vector machine	0.9793	0.9495	0.9804	0.9474
Random forest	0.9788	0.9482	0.9791	0.9469
Gradient boosted trees	0.9786	0.9479	0.9787	0.9469
K-nearest neighbors	0.9779	0.9461	0.9861	0.9369
Mean of embeddings				
Logistic regression	0.8769	0.7171	0.7501	0.8619
Support vector machine	0.8788	0.7234	0.7499	0.8724
Random forest	0.9436	0.8615	0.9246	0.8768
Gradient boosted trees	0.9504	0.8788	0.9222	0.9049
K-nearest neighbors	0.9466	0.8695	0.8992	0.9369

4.4 Naturally balanced classification setting

The set of experiments explained in this section involves all legitimate, benign samples collected by the pipeline introduced in the section, as well as the original imbalanced dataset. These two datasets make in total 7450 benign samples, 18827 malware samples. This class ratio is still somewhat imbalanced but not to the extent that it can cause serious problems. In general, in this set of experiments, some of the deep learning methods have some differences compared to the previous two sections. First,

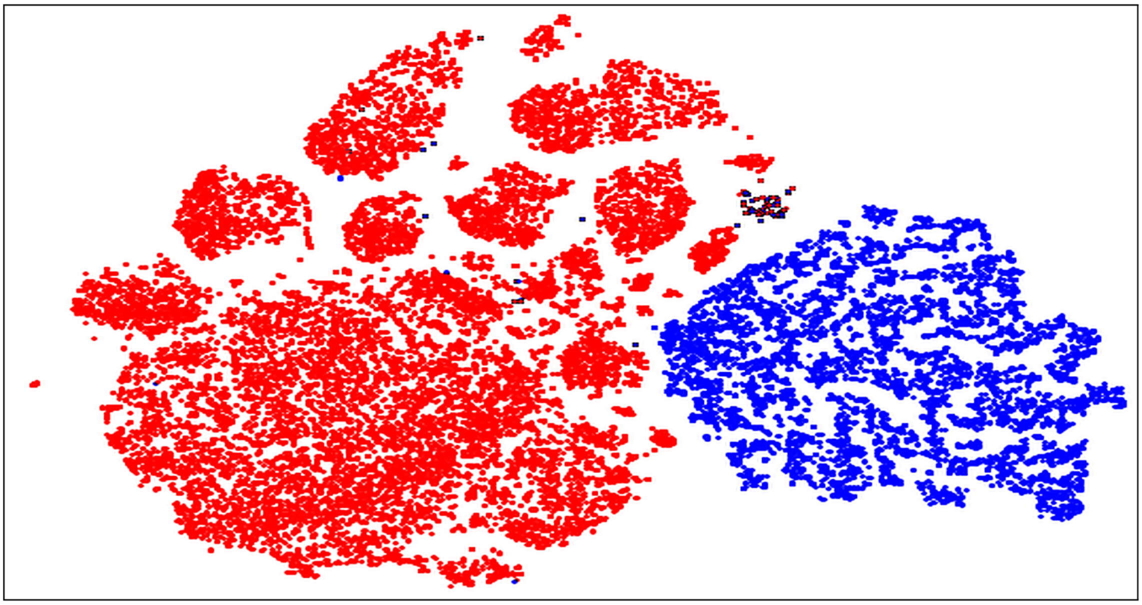


Fig. 9 TSNE visualization of the final context feature vector produced by the CNN model on naturally balanced training setting. TSNE reduces the dimensionality from 512 to 2. This illustration only depicts the training set. Just like Fig. 8, red represents malware, and blue represents benign

the larger CNN architecture is used, and RNN with phased LSTM cell is utilized. Second, the Transformer model is added to deep learning methods in this section.

Table 3 shows the results from the phased LSTM, CNN, and the Transformer model, respectively. The phased LSTM model, this time, yields the best overall performance. The Transformer comes second, and CNN comes the last. There are no drastic performance differences between machine learning algorithms across the three deep learning algorithms. Phased LSTM combined with random forest or gradient boosted trees achieves the best performance. In general, for all the models, both MCC score and recall are much higher compared to before due to the effect balanced dataset. Higher MCC scores clearly indicate that our predictive modeling pipeline performs well for both a wide variety of benign instances collected from wild and existing malware instances from the original dataset. Furthermore, we believe that the high accuracy obtained from the Transformer is significant considering it is a very recent model and unlike CNN and LSTM, the training recipes, the good hyperparameter set has not been established, and the Transformer model type has not been tried extensively except for NLP tasks. Note that, in this section, phased LSTM is utilized instead of Vanilla LSTM. Phased LSTM may provide an extra performance boost in this setting. On the other hand, as is emphasized before, LSTM and phased LSTM require significantly more time to train. The huge training time requirement is especially noticeable for large datasets. Transformer, assuming adequate GPU memory is provided, is much faster to train. Especially with its several extensions, the Transformer can be very efficient for large datasets while retaining high prediction performance. CNN, compared to LSTM, is also quite fast to train. However, it has a larger model size, and in this setting, it achieves poorer performance than Transformer.

Finally, Fig. 9 shows TSNE visualization of final context vectors produced by the CNN from the training data. This figure shows a somewhat different structure than Fig. 8 because of the probabilistic nature of the TSNE algorithm and different underlying deep learning-based feature extraction model. On the other hand, reminiscent of Fig. 8, in Fig. 9, several dense malware clusters are visible again. The possible cause that leads to the formation of these clusters is explained before for Fig. 8. The same cause is valid for Fig. 9. Secondly, in contrast to Fig. 8, there are a lot fewer misplaced benign instances in Fig. 9. In the naturally balanced setting, the underlying model can learn the characteristics of the benign data distribution adequately. Thirdly, similar to Fig. 8, no markedly observable dense benign data cluster is present in Fig. 9. Reinforced by the fact that the larger number of benign instances utilized the naturally balanced setting, it is evident that benign executable set demonstrates significantly more intra-class variance than malware. This argument also makes sense considering that the space of functions benign executable may perform much larger than the space of functions primarily defined by maliciousness. Fourthly, the inter-cluster distances do not seem well adjusted. The main malware cloud is equally far away from malware clusters and the main benign cloud. Inter-cluster distances or global structure may not always be adequately preserved. According to [36], an alternative UMAP algorithm, which was introduced by [35], could arguably preserve global structure better than TSNE. As future work, UMAP algorithm can be utilized for visualization. Finally, the visualization clearly demonstrates that malware and benign clusters are clearly separated for the vast majority of instances.

5 Conclusion

In this project, we proposed an effective malware detection method that uses runtime opcodes of the executable at its input. Using runtime opcode sequences for prediction filters unnecessary noise in executable code, and directly focuses on useful behavioral features of executable. Our method then utilizes Word2Vec algorithm to generate vector representation of opcodes. Finally, the hybrid prediction algorithm, which combines deep learning methods and machine learning methods, is performed. The deep learning methods enable feature engineering without any expert knowledge. Word2Vec algorithm brings two benefits to our predictive modeling pipeline first by drastically compressing input representation and second by encoding contextual information. Word2Vec also has the advantage of not requiring labeled data, making it more scalable. Moreover, we demonstrated that classification based on simple averaging of opcode vectors achieves a respectable performance. We can expect that weighted averaging of Word2Vec vector could achieve even more superior performance. TF-IDF method or smooth inverse frequency introduced by [1] on opcodes can be utilized to obtain weights. Furthermore, we also developed a data collection pipeline geared toward extracting opcode sequences from benign executables during their runtime. Our pipeline collected and processed a total of 6405 benign executables as well as 592 malware executables, which could be very useful for future research.

It is also worth noting that a combination of 1-gram opcode features and TF-IDF weighted average of opcode embeddings could also potentially achieve a respectable performance without rather expensive deep learning operations. This method could especially be useful in environments where operations required for deep learning inference are regarded as too computationally and memory intensive.

Consequently, the results demonstrate that our method yields an overall high classification performance on the runtime opcode sequence dataset. Additionally, it should be emphasized that apart from CNN and LSTM, the performance of the Transformer architecture is remarkable. Due to its efficiency and performance, the Transformer has become a staple building block of many successful large-scale NLP models. Therefore, we think that it is significant to show that the Transformer-based models can potentially be utilized to build a large-scale commercial malware detector. The best choice for a deep learning algorithm depends on the situation. Observably, CNN generally provides better efficiency in terms of memory usage, convergence (training) speed, and inference speed than LSTM. On the other hand, CNN has the largest overall model size. LSTM has slower convergence (training) speed and inference speed but also has the smallest model size and sometimes better accuracy. On the other hand, the Transformer provides a balanced trade-off between inference speed and model size. Experiments in the naturally balanced classification setting clearly demonstrate that the Transformer provides better accuracy than CNN while being slightly worse than phased LSTM. As a result, considering its computational efficiency and performance, the Transformer can be a viable option to build a malware detection system. On the other hand, our method has some possible limitations. First, the initial dynamic analysis of executables is very costly in terms of time. Better dynamic analysis tools might mitigate the inefficiency of the dynamic analysis process. Second, currently, our method only operates on the first 1000 opcode of the executable. The performance can be boosted by using sequences longer than 1000 opcodes. That being said, the amount of information can be easily increased by concatenating statistical features from header files, system calls, or static opcodes to the output embedding generated by the deep learning methods.

Author Contributions All authors contributed equally.

Funding No funding was received to assist with the preparation of this manuscript.

Availability of data and material All the data used in this study can be made available upon request after contacting with any of the authors.

Code availability The relevant code base is stored in the private repository. It can be made available upon request after contacting with any of the authors.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Arora S, Liang Y, Ma T (2016) A simple but tough-to-beat baseline for sentence embeddings (2016)
2. Bazrafshan Z, Hashemi H, Fard SMH Hamzeh A (2013) A survey on heuristic malware detection techniques. In: The 5th conference on information and knowledge technology, IEEE, pp 113–120
3. Christodorescu M, Jha S (2006) Static analysis of executables to detect malicious patterns. WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES, Tech. Rep
4. Beltagy I, Peters ME, Cohan, A (2020) Longformer: The long-document transformer. arXiv preprint arXiv:2004.05150
5. Cakir B, Dogdu E (2018) Malware classification using deep learning methods. In: Proceedings of the ACMSE 2018 Conference, ACM, p 10
6. Carlin D, Cowan A, O'Kane P, Sezer S (2017) The effects of traditional anti-virus labels on malware detection using dynamic runtime opcodes. IEEE Access 5:17742–17752
7. Carlin D, O'Kane P, and Sezer S (2017) Dynamic analysis of malware using run-time opcodes. In: Data analytics and decision support for cybersecurity. Springer, pp 99–125
8. Maaten LVD, Hinton G (2008) Visualizing data using t-sne. J Mach Learn Res 9:2579–2605
9. Chen T, Mao Q, Yang Y, Lv M, Zhu J (2018) Tinydroid: a lightweight and efficient model for android malware detection and classification. Mobile information systems 2018
10. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780
11. fnl (https://stats.stackexchange.com/users/44585/fnl): How to set the dictionary for text analysis using neural networks. Cross Validated. https://stats.stackexchange.com/q/163032. URL: https://stats.stackexchange.com/q/163032 (version: 2017-06-26)
12. Kim Y (2014) Convolutional neural networks for sentence classification. arXiv preprint arXiv:1408.5882
13. Kitaev N, Kaiser Ł, Levskaya A (2020) Reformer: The efficient transformer. arXiv preprint arXiv:2001.04451 (2020)
14. Kolosnjaji B, Eraisha G, Webster GD, Zarras A, Eckert C (2017) Empowering convolutional networks for malware classification and analysis. In: 2017 International Joint Conference on Neural Networks (IJCNN), pp 3838–3845
15. Kolosnjaji B, Zarras A, Webster GD, Eckert C (2016) Deep learning for classification of malware system call sequences. In: Australasian joint conference on artificial intelligence, pp 137–149
16. Krizhevsky A, Sutskever I, Hinton GE (2017) Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp 1097–1105
17. Lab K (2017) Kaspersky Security Bulletin. https://www.kaspersky.com/about/press-releases/2017_kaspersky-lab-detects-360000-new-malicious-files-daily
18. Martinez E (2015) A first shot at false positives. [Online]. Available: https://blog.virustotal.com/2015/02/a-first-shot-at-false-positives.html
19. Lapiello E (2018) Shuffling paragraphs: Using data augmentation in nlp to increase accuracy.https://medium.com/bcggamma/shuffling-paragraphs-using-data-augmentation-in-nlp-to-increase-accuracy-477388746bd9
20. McLaughlin N, del Rincón JM, Kang B, Yerima SY, Miller PC, Sezer S, Safaei Y, Trickel E, Zhao Z, Doupé A, Ahn GJ (2017) Deep android malware detection. In: Proceedings of the seventh ACM on conference on data and application security and privacy, pp 301–308
21. Mikolov T, Chen K, Corrado G, Dean J (2013) Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781
22. Neil D, Pfeiffer M, Liu SC (2016) Phased lstm: Accelerating recurrent network training for long or event-based sequences. In: Advances in neural information processing systems, pp 3882–3890
23. Osborn, M.: Malware detection techniques. Int J Comput (IJC) **18**(1) (2015)
24. Perez L, Wang J (2017) The effectiveness of data augmentation in image classification using deep learning. arXiv preprint arXiv:1712.04621
25. Sung Y, Jang S, Jeong Y-S, Hyuk J et al (2020) Malware classification algorithm using advanced word2vec-based bi-lstm for ground control stations. Comput Commun 153:342–348
26. Jeon S, Moon J (2020) Malware-detection method with a convolutional recurrent neural network using opcode sequences. Inform Sci 535:1–15
27. Popov I (2017) Malware detection using machine learning based on word2vec embeddings of machine code instructions. In: 2017 Siberian Symposium on Data Science and Engineering (SSDSE), IEEE, pp 1–4
28. Sihwail R, Omar K, Ariffin KAZ (2018) A survey on malware analysis techniques: static, dynamic, hybrid and memory analysis. Int J Adv Sci Eng Inf Technol 8(4–2):1662
29. Idika N, Mathur AP (2007) A survey of malware detection techniques. Purdue University
30. Řehůřek R, Sojka P (2010) Software Framework for Topic Modelling with Large Corpora. In: Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, ELRA, Valletta, Malta, pp 45–50. http://is.muni.cz/publication/884893/en
31. Rong X (2014) word2vec parameter learning explained. arXiv preprint arXiv:1411.2738
32. Shijo P, Salim A (2015) Integrated static and dynamic analysis for malware detection. Procedia Comput Sci 46:804–811
33. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems, pp 5998–6008
34. Kaspersky (2018) Types of malware. [Online]. Available: https://www.kaspersky.com/resource-center/threats/malware-classifications
35. McInnes L, Healy J, Melville J (2018) Umap: Uniform manifold approximation and projection for dimension reduction. arXiv preprint arXiv:1802.03426
36. Becht E, McInnes L, Healy J, Dutertre C-A, Kwok IW, Ng LG, Ginhoux F, Newell EW (2019) Dimensionality reduction for visualizing single-cell data using umap. Nat Biotechnol 37(1):38–44
37. Kokhlikyan N, Miglani V, Martin M, Wang E, Alsallakh B, Reynolds J, Melnikov A, Kliushkina N, Araya C, Yan S et al. (2020) Captum: A unified and generic model interpretability library for pytorch," arXiv preprint arXiv:2009.07896
38. Vemparala S, Di Troia F, Corrado VA, Austin TH, Stamo M (2016) Malware detection using dynamic birthmarks. In: Proceedings of the 2016 ACM on international workshop on security and privacy analytics, ACM, pp 41–46
39. Yan J, Qi Y, Rao Q (2018) Detecting malware with an ensemble method based on deep neural network. Secur Commun Networks 2018:1–16
40. Yan J, Qi Y, Rao Q (2018) Lstm-based hierarchical denoising network for android malware detection. Secur Commun Netw 2018:1–18

41. Zhou P, Shi W, Tian J, Qi Z, Li B, Hao H, Xu B (2016) Attention-based bidirectional long short-term memory networks for relation classification. In: Proceedings of the 54th annual meeting of the association for computational linguistics (Volume 2: Short Papers), vol 2. pp 207–212