Malware Detection on Highly Imbalanced Data through Sequence Modeling

Rajvardhan Oak
rvoak@berkeley.edu
University of California, Berkeley

Min Du
min.du@berkeley.edu
University of California, Berkeley

David Yan
david.yan@berkeley.edu
University of California, Berkeley

Harshvardhan Takawale
harshvardhantakawale@gmail.com
BITS Pilani

Idan Amit
iamit@paloaltonetworks.com
Palo Alto Networks

ABSTRACT

We explore the task of Android malware detection based on dynamic analysis of application activity sequences using deep learning techniques. We show that analyzing a sequence of the activities is informative for detecting malware, but that analyzing longer sequences does not necessarily lead to a more accurate model. In the real-world scenario, the number of malware is low compared to that of harmless applications. Our dataset has more than 180,000 samples, two-thirds of which are malware. This dataset is significantly larger than other datasets used in previous studies. We mimic real-world cases by randomly sampling a small portion of malware samples. Using the state-of-the-art model BERT, we show that it is possible to achieve desired malware detection performance with an extremely unbalanced dataset. We find that our BERT based model achieves an F1 score of 0.919 with just 0.5% of the examples being malware, which significantly outperforms current state-of-the-art approaches. The results validate the effectiveness of our proposed method in dealing with highly imbalanced datasets.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

malware detection; sequence modeling; imbalanced data

ACM Reference Format:
Rajvardhan Oak, Min Du, David Yan, Harshvardhan Takawale, and Idan Amit. 2019. Malware Detection on Highly Imbalanced Data through Sequence Modeling. In *12th ACM Workshop on Artificial Intelligence and Security (AISec'19), November 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 0 pages. https://doi.org/10.1145/3338501.3357374

1 INTRODUCTION

Malicious applications known as malware have plagued personal machines and large scale systems from even before the inception of the internet. Once a malware application is installed in the system, it can potentially siphon off sensitive data to third parties, switch on your camera, or consume your device resources for activities like bitcoin mining. In severe cases, malware applications, such as those that perform Advanced Persistent Threat (APT) [38] attacks, have been linked to election disruption [15] and murder [22]. As the advent of the internet has given rise to new methods of remotely installing and running software, the potential avenues for infecting systems have only grown in number. The increasing threat of malware has caused companies to spend billions of dollars on security systems and pressured consumers to spend hundreds on malware detection services. Traditional malware detection mechanisms [6, 8, 42] explore rule-based detection methods to monitor the activities of each software, and classify it as either benign or malicious. The shortcomings of this approach are that it tends to have a high false positive rate due to varying system behaviors, and may require a significant amount of domain knowledge to construct heuristics. Attackers can also disguise malware as benign applications by spreading apart harmful actions among a sequence of harmless or normal ones. In the past, such shortcomings have been a huge detriment to malware detection services.

Recent advances in machine learning have shown increasingly impressive results in a wide variety of classification tasks, such as image classification [32] and natural language processing [10]. In the area of natural language processing, two of the most well known model architectures are Long Short-Term Memory (LSTM) [17], and the recent state-of-the-art language model, Bidirectional Encoder Representations from Transformers (BERT) [10]. LSTM networks are a special class of neural networks that excel at remembering historical sequence data. They use the current as well as a number of history inputs for prediction. BERT attempts to create deep bidirectional representations by pre-training on unlabeled text with masked words and sentences. Its pre-trained model can then be fine-tuned with a single output layer for a wide variety of downstream tasks with a minimum amount of additional training needed. This model has been shown to be empirically very powerful at various text classification tasks, while also requiring significantly less time to train.

LSTM has already been shown to be effective in log file analysis and system anomaly detection [11]. Log files or activity sequences are similar to language models in the sense that both consist of ordered sequences of tokens. Our hypothesis, therefore, is that applying language models on malware detection would perform well. We validate this through some initial experiments. From these findings, we then hypothesize that BERT would also perform well for

AISec'19, November 15, 2019, London, United Kingdom
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6833-9/19/11…$15.00
https://doi.org/10.1145/3338501.3357374

the task since it performs even better on several language modeling tasks. This hypothesis is not only validated through experiments which include our entire dataset, but also in the ones where we intentionally re-balanced the dataset to contain more harmless examples, than harmful ones, simulating the unbalanced nature of malware datasets [7]. This is a significant finding since traditional machine learning models do not work well with such data, pushing all predictions to the majority class.

In our experiments, we use a dataset produced by Palo Alto Networks WildFire [3, 25, 43], from which we extract the labeled action sequences of Android applications. The actions in the dataset are not a log of the specific function calls, but rather a high level description of each application's behavior. Such actions include *"APK file used the HTTP POST method"* and *"APK file created a hidden file"*. These activities are recorded while the application is being monitored, and a chronological sequence of the activities describe the behavior of the application.

We summarize our contributions as follows.

(1) We propose and implement a supervised LSTM based approach for malware classification, validating the assumption that sequence information helps with the classification. We achieve a near-perfect F1 score of almost 0.985 on a large dataset having 183, 000 samples, 2/3 of which being malware.
(2) To deal with highly imbalanced datasets which are common in real-world cases, we borrow the knowledge learned from natural language domain. Using the state-of-the-art language model BERT, we achieve an F1 score of 0.919 on a dataset with only 0.5% malware samples.
(3) We systematically study the benefits brought by BERT pretraining, by comparing the detection results of the models with different initializations (randomly initialized or pretrained on Android data), and trained on small Android datasets having different sizes. We show that using a pretrained model is particularly useful when the labeled training dataset is small. The pre-trained models are also available for downloading to save further malware detection efforts.
(4) We extensively study the effectiveness of applying BERT model for Android malware detection with multiple experiment settings, as well as studying the activity sequences in-depth, and present various statistics to understand the validity of utilizing such data. This study may shed light on future work regarding other security sequence data analysis.

The rest of the paper is organized as follows. Section 2 introduces some basic preliminaries about LSTM networks and BERT sequence modeling approach. Section 3 illustrates our data and approach towards the malware classification task. Section 4 presents the evaluation results of the proposed methods, along with multiple baselines. Finally, we review previous work in Section 5 and conclude the paper in Section 6.

2 PRELIMINARIES

2.1 Overview of Android malware detection and Wildfire

Android is a free, open source, Linux-based operating system for mobile devices. It is structured in the form of a software stack comprising of several major components: a Linux kernel operating system, an Android run-time environment, middleware, libraries, and applications.

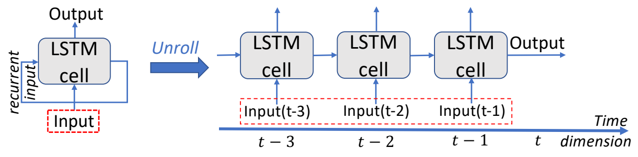
Malware detection has two main approaches: static and dynamic analysis. In static analysis [12] , the application is analyzed without being executed. The detection is usually based on signatures, program structure similarity (e.g., AST), and used utilities. In dynamic analysis [12] , the inspected program is run and observed in a sandbox. The detection is based on the application's actual activities such as API calls being made and Internet domains being contacted. A service called WildFire [25] provided by Palo Alto Networks can perform both types of analysis. For dynamic analysis, WildFire generates a sequence of activities while running an application in a sandbox. An example activity is *"APK file leaked the phone number of the device"*, which possibly indicates that a privacy leakage issue is detected. (APK stands for Android Package Kit, the Android application we examine.)

2.2 LSTM

Long short-term memory (LSTM) [17] is a type of recurrent neural network (RNN) that can process long sequences of data. As an RNN, LSTM contains feedback connections that include previous outputs in the current input, as shown in Figure 1. Because of this, LSTM models are able to analyze entire sequences instead of single examples, and have been broadly applied to time series data analysis. An LSTM cell contains three gates: an input gate i_t which controls the size of the input, a forget gate f_t that determines what of information from the feedback connections to discard, and an output gate o_t. Assuming x_t is the input at step t and h_t denotes the output of the LSTM cell, h_t could be calculated as follows:

$$\begin{aligned} i_t &= \sigma(x_t U_i + h_{t-1} W_i); & f_t &= \sigma(x_t U_f + h_{t-1} W_f); \\ o_t &= \sigma(x_t U_o + h_{t-1} W_o); & c_t &= \tanh(x_t U_c + h_{t-1} W_c); \\ m_t &= m_{t-1} \odot f_t + c_t \odot i_t; & h_t &= o_t \odot \tanh(m_t) \end{aligned} \quad (1)$$

$U_i, W_i, U_f, W_f, U_o, W_o$ are the LSTM network parameter matrices that can be trained through deep learning optimization methods, while c_t, m_{t-1} and m_t are the internal states used to carry information from the input and previous steps.



(a) LSTM network (b) unrolled version of Figure 1(a)

Figure 1: LSTM architecture.

2.3 BERT

Bidirectional Encoder Representations from Transformers [10] (BERT) is a recently proposed language model which has obtained state-of-the-art results on several tasks in the natural language processing (NLP) space. At the heart of BERT lies the Transformer [37]. In contrast to recurrent neural networks such as LSTMs, transformers rely entirely on attention mechanisms for sequence modeling.

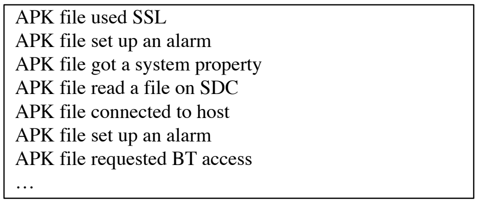
BERT utilizes a new pre-training objective called the Masked Language Model (MLM). In MLM, some words from the input text are randomly masked, and the model tries to predict the masked words based on context. While previous language models generate unidirectional representations, BERT looks at sentences in both the left-to-right and the right-to-left directions to generate bidirectional representations. The key innovation of BERT is that it can learn the context of a particular token with respect to tokens both before and after it. Pre-trained BERT representations can be fine-tuned with just one additional output layer for custom classification tasks. BERT has established state-of-the-art results on natural language classification tasks such as SST-2 [33] and CoLA [39]. The pretrained BERT model on BooksCorpus (800M words) and English Wikipedia [10] saves a lot of training time since only a few epochs of fine-tuning are needed to obtain desired results[13, 28, 36].

3 METHODOLOGY

3.1 Data sources

In this paper, we focus on Android malware detection by analyzing the activity sequence an application produces, which describes the application's chronological behavior after installation. The *activities* may refer to the Android operating system APIs called by the application, internal function calls, or various events identified by some monitoring services (e.g., WildFire). There are several properties for the activities being collected: 1) the types of activities are from a finite set; 2) each activity has the same meaning across all applications on the same Android operating system; and 3) the finer the granularity of the activities, the better the detection results can be. As stated in Section 2.1, for each Android application to be investigated, WildFire generates a sequence of high level activities. A snippet of an activity sequence is shown in Figure 2.

Note that not all aspects of the activities are analyzed in this paper. For example, studying records of contacted domains is a great method of detecting malware, but orthogonal to this paper. Investigating a different approach that considers all aspects of the activities for malware detection is considered as our future work.



APK file used SSL
APK file set up an alarm
APK file got a system property
APK file read a file on SDC
APK file connected to host
APK file set up an alarm
APK file requested BT access
…

Figure 2: A snippet of activity sequence by WildFire.

3.2 Data pre-processing

As previously mentioned, each activity is a high level description of the Android application behaviour, e.g., *"APK file read a file on the device"*. To make the activities learn-able by machine learning models, we need to parse them into digital representations that could be taken as model inputs. Recall that a property of the activities is that they exist in a finite set. Therefore, we can use an integer to represent each of the 174 distinct activities. Based on the requirements of downstream machine learning applications, these integers can either be regarded as symbols and encoded accordingly, or simply used as the numerical inputs of corresponding models.

3.3 Analyzing Android activities

In this section, we first review the possible ways to identify Android malware by analyzing activity sequences, and then present our proposed methodology.

3.3.1 *Representation alternatives.*

The naive approach for detecting malware is to analyze activities made by an application individually. This approach uses only a small part of the available information. In most cases, a single activity does not provide enough information to determine if an entire sequence is malicious. More often than not, a sequence of normal activities combined together could achieve a malicious effect, and that sequence can be interleaved among other activities to mask the intended purpose.

A potential solution to this problem is to use a method that analyzes multiple activities together, such as N-gram. N-gram analysis processes a *continuous* sequence of activities together. The length of the continuous sequence can be adjusted according to the expected length of malicious sequences. However, this technique has many shortcomings. To begin with, the number of N-grams is $|L|^N$, exponential with respect to the N-gram length given an alphabet L, forcing the use of short N-grams. Also, N-grams tend to be redundant. If the real indication is a sequence longer than the length of the N-gram, any of its sub-sequences will be common. Of course, the predictive power of each sub-sequences will be significantly lower than that of the full one. Lastly, locality is not guaranteed. In trying to evade detection, a malicious actor can deliberately delay the activities, and inject dummy activities to make the representation and detection of such sequences harder. To validate these shortcomings, we provide 1-gram and 2-gram analysis of the data using mutual information in Section 4.4.

Thankfully, recent developments in deep neural networks have made malware detection more automatic and intelligent. A model can be fed with the entire activity sequence, extract the relevant features indicating malicious activity, and ignore the dummy activities that were meant to throw it off. From this, the model should be able to construct an accurate classification of the application. Both LSTM and BERT employ this approach in the experiments.

3.3.2 *Utilizing advanced language models.*

Malware samples are extremely hard to find. The ratio of malicious Android applications in real-world ranges somewhere between 0.01% to 2%. Because of this, having a model that can be trained on an imbalanced dataset is highly desirable. This, unfortunately, is a major challenge for many existing machine learning methods, including LSTM. As presented in Section 3.3.1 and evaluated in Section 4.2, sequence information greatly helps with malware detection and eliminates the efforts of manual feature engineering. This observation inspires us to experiment with more advanced language models that surpass LSTM, especially for highly imbalanced datasets.

BERT is the state-of-the-art language model for several natural language tasks [33, 39]. The predecessor of BERT is called Transformer [37]. In contrast to recurrent neural networks that use history sequences for prediction, Transformer utilizes a self-attention mechanism that is able to model relationships between all words in a sentence. BERT takes this a further step and uses bidirectional training of Transformers. BERT has a pre-training mechanism that is able to effectively learn the parameters through self-supervised learning on unlabeled datasets. The core innovation of BERT lies in the use of the masked language model. Before feeding word sequences into BERT, 15% of the words in each sequence are replaced with a masking token. The model then attempts to predict the original value of the masked word(s) based on the context provided by the other words in the sequence. An important point to note here is that it uses the whole context, i.e., words both before and after the masked word to predict what it should be. A model pre-trained on this task is able to effectively model relationships between tokens (words) in different contexts.

BERT can be used in two ways. First, the bidirectional architecture of a self-attention Transformer model is a powerful approach to extract features for input sequences. As a result, the BERT model can be trained from scratch for any machine learning task with sequence data as inputs, and corresponding labels as outputs. Second, the pre-training mechanism presented in [10], as well as summarized above, is able to capture contextual encodings and essential sequence relations in the NLP domain through training on large datasets such as Wikipedia data. These pre-trained models can be fine-tuned with labeled, domain-specific sequences for downstream machine learning tasks.

Our first finding is that a pre-trained BERT model on NLP data performs surprisingly well for Android malware detection, after fine-tuning on Android activity sequence data. Our intuition is that the bi-directional training of a self-attention mechanism using BERT effectively learns the (non-continuous) sequence patterns that are relevant to Android maliciousness. Moreover, the pre-training on natural language data, although having different contextual encodings with Android activity data, helps to identify the parameters that are important for sequence modeling, while deactivating others. We extensively study both effects in the experiments section, which validates our assumption.

The fine-tuning process on Android sequence data works as in Figure 3. BERT is essentially the bi-directional training process of a multi-layer Transformer model. The BERT encoder produces a sequence of hidden states. For classification tasks, we need to reduce the sequence of tokens to a single value. In order to do this, we consider the hidden state corresponding to the token representing the first activity. As the model is trained with bi-directional self-attention mechanism, it is expected that the token encodes all of the sequence information. As shown in Figure 3, we utilize this architecture and add a final softmax layer for classification. In order to fine-tune our model for the specific malware detection task, we train it with labeled Android activity sequence data. During fine-tuning, all pre-trained model parameters are adjusted with the added classification layer to optimize for the task.

Note, that if a pre-trained model is not available or suitable, BERT architecture can also be trained from scratch for domain specific tasks with labeled datasets. For Android malware classification from activity sequences, training the model is the same procedure presented in Figure 3 without the pre-training.

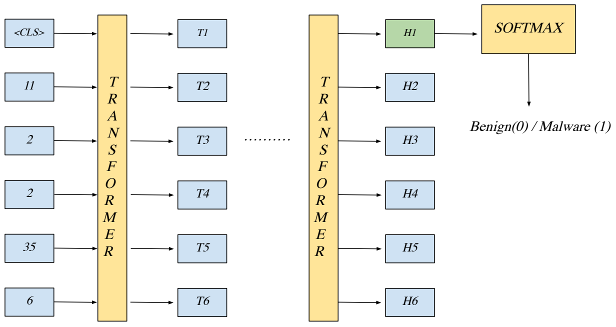


Figure 3: Fine-Tuning BERT for malware classification. T_i are the intermediate states and H_i are the final output.

In our evaluation, we explore the effectiveness of Android malware detection of both BERT models pre-trained on NLP datasets, and BERT models trained from scratch. The results suggest that the BERT model architecture, and the pre-training process on NLP data are both important in achieving strong F1 scores.

4 EXPERIMENTS

In this section, we systematically evaluate the effectiveness of using BERT for Android malware detection with unbalanced labels, through comparing with multiple baseline models and exploring different settings. The source code for our BERT approach, and the pretrained models are available at https://github.com/sunblaze-ucb/Android-malware-detection-imbalanced.

4.1 Set up

Previous work have contributed several datasets such as DREBIN[4] and Kharon[20], to foster research related to Android malware. However, these datasets do not contain the sequence information regarding activities. The dataset used for this work is the Android malware family dataset released by Palo Alto Networks [3], which contains WildFire [25] reports consisting of activity sequence information obtained from over 180, 000 APK files. WildFire is a cloud service for malware analysis provided by Palo Alto Networks. WildFire has more than 26, 000 customers, who may feed the service with various suspicious files logged by firewalls, endpoints, and cloud services for security analysis. After receiving a suspicious file, WildFire applies machine learning combined with static analysis, dynamic analysis and threat intelligence to identify known and novel threats. In the process of dynamic analysis, WildFire produces a dynamic activity sequence made by the file being investigated. The dataset contains 60, 390 (33%) benign, and 120, 780 (66%) malware files.

For each Android application, the collected data includes various information, such as contacted domains, requested permissions, a sequence of dynamic activities made by the application, and the certificate used to sign the APK. As explained in Section 3.1, this paper focuses on the analysis of dynamic activity sequences. Therefore, for each application, we only extract its application sequence from the WildFire report, while ignoring all others.

4.1.1 Data pre-processing. This section illustrates how we process the WildFire reports to produce sequences of activities that can be fed into machine learning models for malware classification. The analysis reports generated by WildFire[25] are in XML format. To obtain the action sequence for each application, we extract the action description in chronological order by following each $< ACTION - MONITORED >$ tag in the XML files. A unique integer token is assigned to each distinct action. The generated lookup table (i.e., the vocabulary set for machine learning models), contains a total of 174 distinct actions/activities.

Finally, with the lookup table, we obtain a sequence of integers representing the dynamic action sequence made by each application. Figure 4 presents the sequence length distribution. Note that there is a negligible number of activity sequences that contain more than 1, 000 actions. These are ignored in this figure for better illustration.

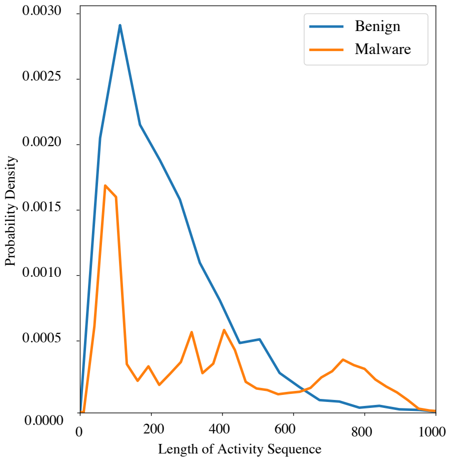


Figure 4: Distribution of the length of activity sequences for benign and malware samples.

After pre-processing, we get a total of 183000 sequences, out of which 120, 780 are malware and 60, 390 are benign. Each token in the sequences is an integer from 1 to 174, indicating 174 possible activities. Note that the characteristics of the sequences are very different from those of any natural language. Therefore, the excellent performance shown by BERT demonstrates remarkable versatility of the architecture and the pre-trained model.

4.1.2 Evaluation metrics. To evaluate our model, we will use four key metrics: Accuracy, Precision, Recall, and F-1 score, which could be calculated from the number of true positives (TP), true negatives (TN), positives (P) and negatives (N) [40]. *Accuracy* is the fraction of the number of correct predictions over the total number of samples, calculated as $(TP + TN)/(P + N)$. *Precision*=TP/P' indicates the correctness of the predictions indicating malware. *Recall*=TP/P measures the percentage of malware being detected by the model. Finally, *F1 score*=2· Precision · Recall / (Precision+Recall) together is an informative, overall evaluation of the model.

4.2 Sequence analysis for malware detection

In this section, we explore how much better sequence analysis performs than bag-of-words analysis. We use methods that analyze activities as bag-of-words as baselines, and compare the evaluation results with those that analyze the sequences.

4.2.1 Dataset. For this set of experiments, we utilize all available data samples, and randomly split them into a training dataset and a test dataset. Our training data consists of 75% (137, 250 samples) of the total dataset, while the remaining 25% (45, 750 samples) is used as test data. The train-test split is *stratified*, which means that both the training and test data have the same ratio of malware and benign applications.

4.2.2 Malware detection without sequence analysis. In this section, we consider the activities made by the application but not the order in which they are made. This is analogous to the *Bag of Words* [44] approach in language modeling where a piece of text is represented as the bag (multi-set) of the words it contains. Specifically, each activity sequence, e.g., <5,3,2,2,1>, is represented as an *activity vector*, for example, [1,2,1,0,1,...], where the *i*-th position represents the token *i* (indicating an activity as in Section 4.1), and the value at each position means the index of the activity in that sequence. To eliminate the influence of the background activities (the equivalents of stop words such as "the", "and" in NLP domain) that are ubiquitous among all vectors, each value in the activity vector is further replaced by its Term Frequency-Inverse Document Frequency (TF-IDF) [27] value. For classification of the activity vectors, we use a multi-layer perceptron approach [16]. Our neural network has 3 hidden layers, each with 100 neurons, which give the optimal results among all hyperparameter combinations we've experimented with. We observe the accuracy, precision, recall and F1 score values to be 0.96 each.

A disadvantage of this approach is that it is only suitable for offline detection where the test dataset is available while the model is being trained. This is because the TF-IDF vector construction needs to be consistent across all activity vectors in training and test datasets. In our experiment, we first process the whole dataset with the TF-IDF approach, and then split the resulting TF-IDF activity vectors into training and test datasets.

4.2.3 Malware detection with sequence analysis. Compared with the TF-IDF approach, analyzing each activity sequence directly for classification makes malware detection easier. Moreover, it has the potential to further boost malware detection performance through sequence analysis. For this experiment, we simply use all activity sequence data after pre-processed in Section 4.1, without any further processing.

We use LSTM as described in Section 2.2 for sequence classification. Since the length of each sequence varies as in Figure 4, we truncate each sequence with a fixed length, H. Specifically, for each sequence, we only use the first H tokens for classification. A sequence shorter than H is padded with the value 0, which represents no activity. We vary H from 5 to 75 and test the malware detection performance. The results are shown in Table 1. Initially, all measurements continue to improve with the increase of length H until $H = 50$. We observe that in Figure 4, most activity sequences, especially malware, have sequence lengths smaller than

50, which means that $H = 50$ could capture most information for most sequences. However, F1 score drops when H increases to 75. We believe this is caused by the memorability of LSTM. Specifically, the longer the input sequence, the harder it is for LSTM network to remember the full history, especially the first several tokens in the sequence. Nevertheless, we show that the best performance for LSTM (F1 score = 0.985) is almost perfect, and outperforms the TF-IDF approach. These related proposals are discussed in greater detail in Section 5. We also tested LSTM with sequences that have varying lengths, and found that such a network trained very slowly, and performed worse than the results in Table 1.

Sequence Length	Accuracy	Precision	Recall	F1 score
5	0.837	0.826	0.909	0.871
10	0.871	0.892	0.895	0.894
20	0.901	0.909	0.929	0.919
50	0.975	0.989	0.979	0.985
75	0.941	0.970	0.931	0.942

Table 1: LSTM performance with varying sequence length.

4.3 Dealing with highly imbalanced dataset

As explained above, malware samples are rare. To estimate the positive rate, i.e., the ratio of malware among all Android applications, we utilize the AutoFocus system [24] provided by Palo Alto Networks. AutoFocus is a service which combines machine learning analysis with human intelligence on the samples collected by the WildFire system. It utilizes up to trillions of data points to deliver improved contextual threat intelligence. This analysis allows us to drill down into the dataset with respect to data type and time. According to the AutoFocus analysis, the positive rate varies based on different years, definitions or scopes of the collected samples. When considering instances of programs, popular applications like "Google Play" will be dominant, and the ratio of malicious instances are negligible. If we consider application level, e.g. counting all instances of "Google Play" as a single application, the positive rate of malicious applications rises to around 0.01%. Maliciousness can be more popular in “new” files, defined by their creation time or first exposure to security vendors. For new files, positive rate increases to 0.5% for Windows executable files, and to 2% for Android executable files.

4.3.1 Dataset sub-sampling. A special focus of this paper is dealing with highly imbalanced datasets. For this purpose, based on the original dataset described in Section 4.1, we deliberately construct datasets that have different low positive ratios. Specifically, we utilize all benign application sequences (60, 390 in total), and randomly select some malware sequences according to the positive ratio. For each experiment, we vary the positive ratio among the values of 0.2%, 0.5%, 1%, 2%, and 5%. We don’t analyze cases of ratios of the order 0.01% and below, because they would result in extremely low positive examples, making the learning and estimation prone to errors.

4.3.2 Attempts of outlier detection. In the past, when the percentage of anomalies is extremely low in the dataset, unsupervised outlier detection has been a popular direction to pursue. In particular, unsupervised detection aims to model a distribution that fits the majority of the samples in the dataset, and label the ones that do not fit as outliers.

In particular, for unsupervised outlier detection, we present two pervasively used methods, clustering and Autoencoder [16], and a recently proposed state-of-the-art method, DAGMM [45]. We also compare with a semi-supervised anomaly detection method, DeepLog[11], which is the state-of-the art approach for system log sequence anomaly detection. These baselines help to show the difficulty in dealing with highly imbalanced datasets. For this set of experiments, we use two datasets from Section 4.3.1, with positive ratios of 1% and 2% respectively. The best results for each method is shown in Table 2, and we detail them below.

Clustering. We implement K-Means clustering [16] with cosine distance as the distance metric and attempt to separate malware and benign applications using $K = 2$. The first step is to encode each activity sequence with an embedding vector, such that cosine distance could be calculated between every two sequence representations. For the encoding approach, we tried various ways including the aforementioned TF-IDF vectors in Section 4.2.2, as well as an average-Word2Vec embedding approach, which works as follows. By treating each sequence as a “sentence”, we first encode each integer token using the word2Vec approach [23], and then calculate the average embedding for all tokens in each sequence, as the embedding of the data sample. However, the best results we can get, as shown in Table 2, has an F1 score of only 0.029. To understand whether malware samples are easily separable from benign samples, we visualize the embedding vector for each sample in a 2-dimensional space. In particular, we use Principal Component Analysis (PCA) to reduce the dimension of each sample embedding to 2. The resulted representations are plotted in Figure 5, where each green dot represents a benign sample, while each red cross-mark represents a malware sample. This figure shows that malware sample embeddings are mixed with benign ones and not easily separable, which further indicates the difficulty in dealing with highly imbalanced data.

Autoencoder. Autoencoder is a commonly used neural network for anomaly detection. It contains an encoder which reduces the dimension of each input example, and a decoder which aims to reconstruct the input data. Because dimensionalality reduction inevitably causes information loss, Autoenocoder learns to keep the information that is common among most of the input samples. An outlier sample is detected by measuring the error between the input and corresponding output, referred to as reconstruction error. This error score is supposed to be bigger for outliers because it has patterns that are not learned by the network. To use Autoencoder for outlier detection on activity sequence data, we use LSTM as the encoder and decoder. More specifically, we train a sequence to sequence model, where the training objective is to minimize the error between input sequence and output sequence. For the dataset containing 1% malware data, we obtain an F1 score of 0.61, as shown in Table 2, and 0.66 for 2% positive ratio. We find that there is no clear decision boundary between the reconstruction errors of the malware and benign activity sequences. Moreover, the

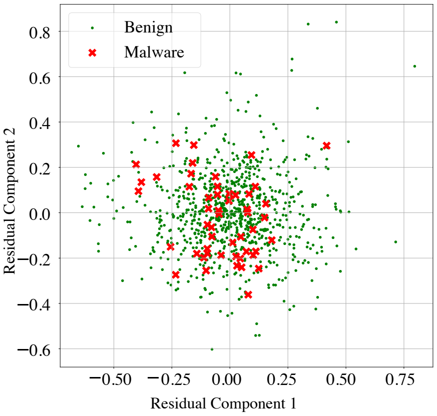
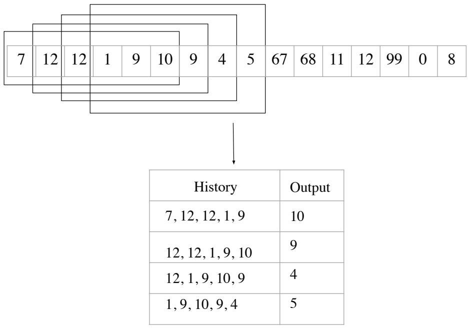


Figure 5: Visualization of 2-dimensional embedding vectors for benign/malware sequences (not easily separable for K-means clustering).

average reconstruction error for malware samples is only 1% higher than that of the benign sequences.

DAGMM. DAGMM is short for Deep Autoencoder Gaussian Mixture Model [45], which is a recently proposed model that claims to achieve state-of-the-art results on anomaly detection for high dimensional data. DAGMM contains two parts, an Autoencoder, followed by a Gaussian mixture model. Specifically, each data sample is first fed into an Autoencoder for dimensionality reduction. Then, a vector V is constructed by concatenating the output embedding of the encoder (i.e., the input of the decoder), and the distance (error) between the input and output. Vector V is further used to fit the following Gaussian mixture model. The Autoencoder and Gaussian mixture model parameters are optimized together, details of which can be found in [45]. DAGMM has been found to boost F1 scores by up to 14% on several public benchmark datasets such as KDDCup [21] and Thyroid [21] datasets. In our experiment, we use LSTM-Autoencoder, i.e., the sequence to sequence model as described in Section 4.3.2 for the Autoencoder network, and fit a Gaussian mixture model with 4 mixture components. Using this method, we obtain an F1 score of 0.53-0.55, which is even worse than the Autoencoder method. We think that the superiority of DAGMM is not revealed here because the input dimension is not very high.

DeepLog. DeepLog [11] is a recently proposed work for system log anomaly detection and remains the current state-of-the-art approach. Similar to our goal in Android malware detection, DeepLog also analyzes system log sequence directly. Note that each system log message, e.g, *"Transaction A finished."*, can be viewed as a type of activity, as described in Section 3.1. Inspired by this, we utilize a similar method to analyze Android activity sequences for malware detection. Specifically, the training data should only contain normal/benign sequences. Given a sequence, an LSTM sequence prediction model is trained to predict the next activity based on a fixed number of history activities. For example, as shown in Figure 6, if the activity sequence is *<7,12,12,1,9,10,9,4,5....>*, we could generate the following input-label pairs for LSTM model training: *[7,12,12,1,9]: 10*; *[12,12,1,9,10]:9*; *[12,1,9,10,9]:4*; *[1,9,10,9,4]:5*; .... When the trained model is used for detection, given a new activity sequence (e.g., *<12,1,9,10,9,4,5....>*), we could apply a sliding window with the same fixed length. Given a sliding window history (e.g., *[12,1,9,10,9]*) as the input for DeepLog model, it outputs a probability distribution on all activities that may appear as the one following the inputs: {4: 80%, 5: 19%, ...}. Then we can compare the actual activity at the position with the predicted one, and classify it as anomalous if the difference is too large. For Android malware detection, we consider an activity sequence is malware if at least one activity is detected as anomalous, and benign otherwise. The F1 score for DeepLog method on a 2% positive ratio dataset is 0.65, although among the best of all baselines, is still not good enough for real-world Android malware detection.



History	Output
7, 12, 12, 1, 9	10
12, 12, 1, 9, 10	9
12, 1, 9, 10, 9	4
1, 9, 10, 9, 4	5

Figure 6: Training sample construction from sequence data (DeepLog).

Model	Positive Ratio	Accuracy	Precision	Recall	F1 score
Clustering	1%	0.749	0.005	0.153	0.010
	2%	0.696	0.016	0.233	0.029
Autoencoder	1%	0.993	0.790	0.501	0.613
	2%	0.985	0.611	0.709	0.656
DAGMM	1%	0.992	0.729	0.423	0.535
	2%	0.984	0.658	0.472	0.550
DeepLog	1%	0.990	0.502	0.846	0.630
	2%	0.985	0.615	0.684	0.648

Table 2: Performance of baseline detection methods.

4.3.3 Applying BERT for Android malware detection. We present our major finding that BERT, the current state-of-the-art language model, is surprisingly effective in dealing with highly imbalanced datasets.

Effectiveness of BERT for highly imbalanced datasets. For the first experiment, we adopt a pre-trained BERT model, i.e., the BERT Base-Uncased model [10]. The model, with around 110 million parameters, was pre-trained with the masked language modeling task on the Google News and Wikipedia corpora. We find that using the hyper-parameters suggested in [10] guarantees decent results. Specifically, we use Adam optimizer with a learning rate of 5×10^{-5}, a batch size of 32, and a dropout probability of 0.1. The pre-trained model is fine-tuned by 3 epochs on the Android activity sequence datasets with different positive ratios.] We split the test and training sets evenly so that both are approximately the same size. The results are shown in Table 3. Surprisingly, the BERT model pre-trained on natural language is able to get exceptional results on highly imbalanced Android sequence datasets with only 3 epochs of fine-tuning. As an example, the F1 score for the dataset containing only 2% positive ratios is 0.91.

One may point out that even though the training datasets contain extremely low positive ratios, BERT model is still trained as a supervised classification task, compared with the unsupervised and semi-supervised baselines presented in Section 4.3.2. To further show the superiority of BERT model, we apply the supervised LSTM model as described in Section 4.2.3 on the imbalanced training dataset with different positive ratios. Note that the LSTM model is able to achieve an F1 score of up to 99% on a somewhat balanced dataset (1/3 benign and 2/3 malware sequences). However, a direct application of the same model on imbalanced dataset only gives an F1 score of 0.49 for the dataset with a 5% positive ratio, and an F1 score of 0 for the datasets with positive ratios of 2% under, as shown in the "LSTM" column of Table 4. To deal with imbalanced datasets for supervised classification, a commonly adopted approach is to use *weighted loss function*. Specifically, if there are a total of 100 training samples, among which only 1 has a positive label, then in the training objective function, the weight for label 0 is 1/100=0.01, and the weight for label 1 is (1-0.01)=0.99. The idea is to emphasize the minority class by assigning higher weights. We use this approach to improve the supervised LSTM method, and present its results in the "LSTM-weighted" column shown in Table 4. The performance improves significantly compared to the LSTM approach without weighed loss, e.g., from 0 to 0.49 for positive ratio 2%. However, BERT is able to achieve an F1 score of 0.91 for the same positive ratio, and much better results on all other cases. We also tried weighted loss function for BERT model while fine-tuning, but did not get better results. This may be due to the fact that the existing BERT model is already powerful enough at extracting the hidden information for highly imbalanced datasets.

Positive Rate	Accuracy	Precision	Recall	F1 score
0.2%	0.999	1.000	0.647	0.786
0.5%	0.999	0.957	0.776	0.857
1%	0.998	0.975	0.805	0.882
2%	0.997	0.956	0.876	0.914
5%	0.993	0.982	0.892	0.934

Table 3: BERT performance, model pre-trained on NLP data and fine-tuned for 3 epochs.

To validate if the effectiveness is brought by the pre-training itself, or by the BERT model architecture, we further conduct two other experiments, to train a clean BERT model from scratch, for 3 epochs and 30 epochs respectively.

Positive Rate	LSTM	LSTM-weighted	BERT
0.2%	0.001	0.001	0.786
0.5%	0.001	0.279	0.857
1%	0.002	0.406	0.882
2%	0.003	0.491	0.914
5%	0.487	0.615	0.934

Table 4: Comparison of F1 scores of BERT and LSTM.

Effectiveness of pre-training. To understand whether the pre-training process on NLP data is helpful for Android sequence modeling, we train a clean BERT model from scratch for 3 epochs, on the same dataset used to fine-tune the pre-trained BERT model in previous section. The results on datasets with different positive ratios are shown in Table 5. We can observe that, the results for the model trained from scratch are not as good as the ones presented in Table 3, which contains the results trained from a pre-trained BERT model. This indicates that the pre-training process on NLP data indeed helps with Android malware detection. One possible explanation is that the pre-trained model is easier to optimize compared to a random initialization, since the exact way of how a model is randomly initialized also matters a lot. Moreover, we find a closely related reference in [46], where a neural machine translation model is pre-trained on a high-resource language (e.g., French), and fine-tuned on a low-resource language (e.g., Uzbek). For fine-tuning, the Uzbek words are initially mapped to random French embeddings. The results show that the pre-training process on a high-resource language domain is able to boost the performance on the low-resource language domain. In our case, we can understand Android activity sequences as a language different than English, and that with a more advanced language model BERT, pre-training on English texts helps with the sequence modeling in Android activity sequences. This finding could potentially be applied to other sequence data.

Positive Rate	Accuracy	Precision	Recall	F1 score
0.2%	0.999	0.789	0.441	0.566
0.5%	0.998	0.896	0.706	0.789
1%	0.997	0.961	0.726	0.827
2%	0.996	0.972	0.804	0.880
5%	0.992	0.976	0.851	0.909

Table 5: BERT performance, model trained from scratch for 3 epochs.

Effectiveness of the BERT architecture. In this experiment, we train a clean BERT model from scratch for 30 epochs to evaluate the effectiveness of the BERT model architecture. The results with different positive ratios in the training datasets are shown in Table 6. We can observe that this set of results is in general better than the results obtained from training a clean BERT model for 3 epochs (Table 5), and comparable to fine-tuning a pre-trained BERT model for 3 epochs (Table 3). This shows the effectiveness of BERT architecture in modeling security sequence data.

Positive Rate	Accuracy	Precision	Recall	F1 score
0.2%	1.000	0.960	0.706	0.814
0.5%	0.999	0.974	0.871	**0.919**
1%	0.998	0.922	0.822	0.869
2%	0.996	0.947	0.849	0.895
5%	0.992	0.950	0.896	0.922

Table 6: BERT performance, model trained from scratch for 30 epochs.

4.3.4 Pre-training BERT on Android dataset. With the abundant Android data we have, it is intriguing to explore the benefits brought by pre-training, where the pre-training data are from the same domain as the training data used for the detection task. The problem settings is: suppose we need to train an Android malware detection model and we only have a small dataset with unbalanced labels, but we have a large unlabeled Android sequence dataset, we are interested to explore whether pre-training on the unlabeled dataset could improve the performance of the model trained on the small unbalanced dataset. We want to answer the following three questions: 1) whether pre-training on Android data helps to improve the detection performance; 2) what kind of pre-training data produces better results, a dataset containing only benign execution sequences, or a dataset that contains all available data which include malware execution sequences; 3) whether pre-training is more beneficial if the training dataset is smaller.

Datasets and set up. For each experiment, we'll need three non-duplicated datasets: 1) a large dataset that could be used for pre-training; 2) a small labeled dataset that is used as the training dataset for Android malware detection task; and 3) a test dataset that could be used to evaluate the model performance.

As described in Section 4.1, we have 60,390 benign, and 120,780 malware execution sequences that could be exploited, which we call as *base dataset*. To construct the three datasets explained above, we first split the base dataset into three portions with 60%, 30%, 10% each, referred as datasets D1, D2, D3 respectively. From D1, we construct two types of datasets for pre-training. The first one *D1-all* contains all data in D1. The second one *D1-benign* contains only benign sequences in D1. For D2 and D3, we further down-sample them to contain a positive ratio of 2% in order to simulate the highly imbalanced case. The new *D3* is then used as the test dataset. To evaluate the benefits of using a pre-trained model when the available training data might be insufficient, we extract different portions of data from D2 (10%, 20%, 50%), and call them as *D2-0.1, D2-0.2, D2-0.5* respectively. This dataset construction procedure along with all constructed datasets are summarized in Table 7.

With these datasets, we conduct the following experiments which help to answer the aforementioned three questions. For each experiment, we will train a malware detection model using a variant of D2, and test the model performance on D3. The difference is the initial model to be used. We exploit three choices: 1) a clean BERT model that has randomly initialized parameters; 2) a BERT model pre-trained on D1-all; 3) a BERT model pre-trained on D1-benign. Following [10], each pre-training process is done for 40 epochs. It takes roughly 24 hours to pre-train on D1-benign and 80 hours on D1-all, using one NVIDIA GeForce GTX 1080 GPU card.

base dataset: 60,390 benign, and 120,780 malware sequences		
60%	30%	10%
D1-all	down-sample each to have 2% positive ratio	
	D2	**D3**
D1-benign: all benign sequences from D1-all	**D2-10%, D2-20%, D2-50%**: sample 10%, 20%, 50% data from D2	
pre-train	train	test

Table 7: Dataset construction.

Table 8 lists the results. In correspondence to the questions at the beginning of this section, we can observe the following. First, compared with a randomly initialized BERT model, pre-training on Android sequence data in general achieves better results, both for D1-all and D1-benign. Second, for each variant of training dataset D2, pre-training on benign sequences only is slightly better than pre-training on all Android data including malware sequences. One possible explanation is, a *malicious* token in training data could be treated as *out of vocabulary* if pre-training on D1-benign, which is another clue to be classified as malicious. Nevertheless, considering the facts that the malware ratio in D1-all is particularly high (66.7%) but in real world the positive ratio is extremely low, and that pre-training on D1-all also has significant improvements, it should be safe to pre-train the model on all available unlabeled sequences. Finally, the smaller the training dataset, the more evident of the benefits brought by pre-training. For example, for training dataset D2-10% which only contains 10% data of D2, pre-training improves the F1 score from 0.677 to 0.787, a 16.2% improvement; while the improvement for dataset D2 is only 2.5% (from 0.878 to 0.900). This indicates that a pre-trained model is especially useful when the training dataset is small, which aligns with the observation in previous work [10, 46].

4.4 Activity analysis

The analysis we've done so far can be applied to any sequence data. In this section, we would like to examine the specific dataset behavior and explain what leads to the usefulness of the sequence data.

We begin with the simplest process to examine the predictive power of each activity alone. Taking a look at the top 10 activities with the highest correlation to malware, according to mutual information metric, leads to some interesting insights. Some activities that strongly infer malicious behaviors, such as removing the launcher icon, an evasion method that benign application do not perform. Other activities, such as connecting to a socket, might be due to both harmless and harmful causes. By using threat intelligence and prior analysis one can determine if the socket is

Initial BERT model	Training dataset	Accuracy	Precision	Recall	F1 score
Randomly initialized	D2-10%	0.982	0.933	0.546	0.677
	D2-20%	0.988	0.981	0.649	0.777
	D2-50%	0.989	0.967	0.707	0.816
	D2	0.989	0.957	0.813	0.878
Pre-trained on D1-all	D2-10%	0.982	0.957	0.647	0.772
	D2-20%	0.983	0.977	0.651	0.781
	D2-50%	0.983	0.901	0.751	0.819
	D2	0.989	0.966	0.804	0.878
Pre-trained on D1-benign	D2-10%	0.982	0.889	0.705	**0.787**
	D2-20%	0.984	0.939	0.697	**0.799**
	D2-50%	0.987	0.993	0.757	**0.859**
	D2	0.991	0.962	0.846	**0.900**

Table 8: BERT performance comparison with different pre-trained models and training datasets.

Activity	Mutual information	Hit Rate	Precision
APK file set up an alarm	0.22	0.65	0.76
APK file removed the launcher icon	0.21	0.33	0.99
APK file fetched the information of apps installed on the device	0.19	0.47	0.37
APK file fetched device specific information	0.17	0.45	0.41
APK file deleted a file	0.17	0.51	0.37
APK file tried to connect to a socket	0.16	0.66	0.68
APK file fetched the current running tasks on the device	0.15	0.31	0.35
APK file tried to connect to a malicious socket	0.14	0.23	1.00
APK file tried to connect to the URL	0.11	0.48	0.44
APK file ran a command	0.10	0.18	0.56

Table 9: Activities and their mutual information.

malicious. This would identify "malicious socket connection", an activity of a lower hit rate but higher precision.

However, this information of singular activities is not nearly as informative as that from sequences. Certain activities may be completely benign on their own, but can be indicated to be more suspicious when linked with other activities. The mutual information from a bag of words representation help with this type of analysis but it does not take into account the ordering of the activities. This is critical to capture in malware detection as the ordering of activities is oftentimes very important. An example would be to consider the two activities, *"APK file deleted SMS records in the database" "* and *"APK file sent out an SMS message to a premium number"*. If the deletion happened after sending a message, it is more suspicious since the malware could be trying to evade detection of the SMS by deleting its record. The opposite ordering would imply something completely different. Hence, there are informative indications in the ordering of the activities, not just the existence of both.

4.5 Discussion

4.5.1 Applying BERT to other security sequence data. A valuable result from our experiments is the effectiveness of BERT language model for highly imbalanced Android activity sequence datasets. This finding is particularly interesting for security domain. For one thing, a lot of security data is sequential, for example binary code sequence, and any kind of system event sequences, e.g. system log sequence, syscall sequence, system command sequence, etc. For another, malicious data are especially hard to collect in security domain, which could be at the cost of attackers successfully breaking the system. Therefore, we believe our findings could shed light on other security data analysis, like binary analysis in DEEPVSA [14], and system log analysis in DeepLog [11]. We also demonstrate that the pre-training process in NLP domain is helpful in saving the training efforts for security data, validated by having decent results with only 3 epochs of fine-tuning, compared with using a BERT model trained from scratch (Section 4.3.3).

4.5.2 Threat to validity. For imbalanced dataset construction in our experiments, our goal is to conduct experiments that would represent the use in real world cases on a sufficiently large dataset. Although our dataset is significantly larger than the ones in previous studies, the positive rate in nature is so low that we cannot be arbitrarily close to this goal. We could not fully simulate a real world dataset since a positive rate of 0.01% on a dataset having 180, 000 samples, would have only 18 positive samples. As a compromise to make sure that the model performed decently well, we ran the experiments on slightly larger positive ratios.

Another threat to validity is the likely violation of the Independent Identical Distribution (i.i.d.) assumption. This is caused by malicious actors that attempt to avoid detection by uploading the same malware software colored in different ways [5]. This leads to multiple variants of the same malware existing in both the training and testing datasets, boosting the results. In the most severe case, there could be as many as 1, 000 samples that are taken from a single malware software. Due to the large scope of variant identification, we don't address this issue here, and leave it to future work.

In addition, one of the limitations of this work is the lack of time-aware experiments. The dataset we used was collected from about a month without a specific timestamp for each sample, making such an analysis in-feasible. It is possible that some of the data points in the training dataset are posterior to items in the test set [1], which possibly biases our observed results. We leave the analysis of time-aware splits, which may be available by exploring the AndroZoo [2] dataset, to our future work.

5 RELATED WORK

Early methods for malware detection can be broadly classified into two categories: signature-based[8] and behavior based[6, 42]. Signature-based techniques cross check application signatures with a known list of malware signatures. On the other hand, behavioral methods evaluate an application based on it's program structure, and monitor deviations from normal program states.

One of the earliest works in machine learning based malware detection by Sahs and Khan [30] uses permissions and control flow graphs (an abstract representation of a program in which vertices represent atomic blocks of non-jump instructions, and edges represent the possible paths of program flow) as features, and a One-Class Support Vector Machine (SVM) [31] to classify. The work also provides guidelines on the choice of kernel for the SVM and compares the performance of different kernel functions. SVM has also been used for malware classification along with decision trees by Peiravian and Zhu [26].

Sequential pattern mining on application program opcodes by Darabian et al [9] has shown to be effective in detecting malware in the Internet of Things(IoT) space. Kakisim et al [34] have performed extensive experiments on dynamic feature-based malware detection. Their experiments reveal that API calls, system library usage and operation sequences are important features for malware detection. The API calls made by applications are used as features for malware classification in a recent work by Jung et al [18]. The approach presented in the paper does not consider sequence of API calls as a whole. Instead, it uses a list of common API calls obtained from the Android website. Using ranked lists of API calls in malicious and benign applications as features, malware classification is performed using a Random Forest classifier. Although the authors present some guidelines on how to effectively select the subset of API calls, the use of only certain calls and excluding others completely might lead to the loss of valuable information from the feature space.

The work most closely related to our work is MalDozer [19], a deep learning based framework for malware classification. Similar to our work, MalDozer it uses API call sequences as input. An important contribution of this work is the attribution of malware with their specific families, and binary malware classification. Using word embeddings, each call is mapped to a fixed length vector. The application is thus represented by a series of embeddings. Although the results obtained by MalDozer are significant (F1 score of more than 0.96), the problem setting does not accurately mimic the real world. In MalDozer , the malware forms around 50% of the dataset. The techniques and the results have not been evaluated under low positive rate settings.

A framework called DroidDeep [35] also effectively leverages deep learning for Android malware classification. In DroidDeep , a high dimensional, multi-level feature space is constructed from a variety of information such as the static information including permissions, API calls, and deployment of components for characterizing the behavioral pattern of Android apps. DroidDeep uses a neural network for feature extraction, and an SVM to perform the actual classification based on those features. A notable contribution of DroidDeep is the remarkable run-time efficiency and scalability it achieves; the run-time varies almost linearly with number of applications being processed.

6 CONCLUSION

In this paper, we have explored the problem of solving the malware classification task using a few machine learning techniques. Our focus has been on using just the activity sequences as the features for classification. Our experiments with both sequence and non-sequence data show that considering the order and locality of activities adds some value to the classification. We show that LSTM networks have good performance, but with a high positive rate for training. We also find that a larger history window does not necessarily lead to better performance in classification. Finally, we experiment with BERT, a state-of-the-art language model. Our simulations show that BERT model achieves state-of-the-art results for malware classification in a low positive rate setting. An interesting research problem that deserves attention is evaluating whether this technique can also attribute the malware to its class or family. In addition, it would be a worthwhile research direction to evaluate how well other recent language models, such as XLNet [41] perform for the malware detection task. Due to the versatility of the language models it will be interesting to apply them to more cyber sequence datasets. Finally, exploiting the robustness of the proposed technique to adversarial manipulation of activity sequences [29] is also an intriguing direction to explore.

ACKNOWLEDGMENTS

We would like to acknowledge Yinnon Meshi, Erez Levy, Tomer Schwartz and Xiao Zhang from Palo Alto Networks, and Richard Shin and Dawn Song from UC Berkeley, for the valuable discussions and helpful feedback. We also thank the anonymous reviewers for their insightful comments which helps to improve the quality of the paper. This work was supported by the CLTC (Center for Long-Term Cybersecurity) at UC Berkeley.

REFERENCES

[1] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Are your training datasets yet relevant?. In *International Symposium on Engineering Secure Software and Systems*. Springer, 51–67.

[2] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 468–471.

[3] Idan Amit, John Matherly, William Hewlett, Zhi Xu, Yinnon Meshi, and Yigal Weinberger. 2019. Machine Learning in Cyber-Security - Problems, Challenges and DataSets. *The AAAI-19 Workshop on Engineering Dependable and Secure Machine Learning Systems* (2019). Available at https://arxiv.org/abs/1812.07858.

[4] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket.. In *Ndss*, Vol. 14. 23–26.

[5] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. 2012. Camouflage In Malware: From Encryption To Metamorphism. *International Journal of Computer Science And Network Security (IJCSNS)* 12 (01 2012), 74–83.

[6] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 15–26.

[7] Nitesh V. Chawla and Nathalie Japkowicz. 2004. Editorial: special issue on learning from imbalanced data sets. *SIGKDD Explor. Newsl* (2004), 1–6.

[8] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. 2005. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*. IEEE, 32–46.

[9] Hamid Darabian, Ali Dehghantanha, Sattar Hashemi, Sajad Homayoun, and Kim-Kwang Raymond Choo. 2019. An opcode-based technique for polymorphic Internet of Things malware detection. *Concurrency and Computation: Practice and Experience* (2019), e5173.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[11] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1285–1298.

[12] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. 2014. Malware analysis and classification: A survey. *Journal of Information Security* 5, 02 (2014), 56.

[13] Jianfeng Gao, Michel Galley, Lihong Li, et al. 2019. Neural approaches to conversational AI. *Foundations and Trends® in Information Retrieval* 13, 2-3 (2019), 127–298.

[14] Wenbo Guo, Dongliang Mu, Xing Xinyu, Min Du, and Dawn Song. 2019. DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In *Proceedings of The 28th USENIX Security Symposium*. USENIX.

[15] David M. Halbfinger and Ronen Bergman. 2019. *Gantz, Netanyahu's Challenger, Faces Lurid Questions After Iran Hacked His Phone.* https://www.nytimes.com/2019/03/15/world/middleeast/gantz-netanyahus-challenger-faces-lurid-questions-after-iran-hacked-his-phone.html

[16] Jiawei Han, Jian Pei, and Micheline Kamber. 2011. *Data mining: concepts and techniques*. Elsevier.

[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[18] Jaemin Jung, Hyunjin Kim, Dongjin Shin, Myeonggeon Lee, Hyunjae Lee, Seong-je Cho, and Kyoungwon Suh. 2018. Android Malware Detection Based on Useful API Calls and Machine Learning. In *2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*. IEEE, 175–178.

[19] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. 2018. MalDozer: Automatic framework for android malware detection using deep learning. *Digital Investigation* 24 (2018), S48–S59.

[20] Nicolas Kiss, Jean-François Lalande, Mourad Leslous, and Valérie Viet Triem Tong. 2016. Kharon dataset: Android malware under a microscope. In *The {LASER} Workshop: Learning from Authoritative Security Experiment Results ({LASER} 2016)*. 1–12.

[21] Lichman. 2013. *UCI Machine Learning Data Repository*. http://archive.ics.uci.edu/ml

[22] Oren Liebermann. 2019. *How a hacked phone may have led killers to Khashoggi*. https://edition.cnn.com/2019/01/12/middleeast/khashoggi-phone-malware-intl/index.html

[23] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3111–3119.

[24] Palo Alto Networks. 2019. *AutoFocus Threat intelligence for security analysts.* https://www.paloaltonetworks.com/products/secure-the-network/autofocus

[25] Palo Alto Networks. 2019. *WILDFIRE MALWARE ANALYSIS Find and stop unknown attacks automatically*. https://www.paloaltonetworks.com/products/secure-the-network/wildfire

[26] Naser Peiravian and Xingquan Zhu. 2013. Machine learning for android malware detection using permission and api calls. In *2013 IEEE 25th international conference on tools with artificial intelligence*. IEEE, 300–305.

[27] Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, Vol. 242. Piscataway, NJ, 133–142.

[28] Siva Reddy, Danqi Chen, and Christopher D Manning. 2019. Coqa: A conversational question answering challenge. *Transactions of the Association for Computational Linguistics* 7 (2019), 249–266.

[29] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. 2018. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 490–510.

[30] Justin Sahs and Latifur Khan. 2012. A machine learning approach to android malware detection. In *2012 European Intelligence and Security Informatics Conference*. IEEE, 141–147.

[31] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. 2001. Estimating the support of a high-dimensional distribution. *Neural computation* 13, 7 (2001), 1443–1471.

[32] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[33] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*. 1631–1642.

[34] Ibrahim Sogukpinar. 2019. Analysis and Evaluation of Dynamic Feature-Based Malware Detection Methods. In *Innovative Security Solutions for Information Technology and Communications: 11th International Conference, SecITC 2018, Bucharest, Romania, November 8 {u2013} 9, 2018, Revised Selected Papers*, Vol. 11359. Springer, 247.

[35] Xin Su, Dafang Zhang, Wenjia Li, and Kai Zhao. 2016. A deep learning approach to android malware feature learning and detection. In *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 244–251.

[36] Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. 2019. BERT4Rec: Sequential Recommendation with Bidirectional Encoder Representations from Transformer. *arXiv preprint arXiv:1904.06690* (2019).

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[38] N. Virvilis and D. Gritzalis. 2013. The Big Four - What We Did Wrong in Advanced Persistent Threat Detection?. In *2013 International Conference on Availability, Reliability and Security*. 248–254. https://doi.org/10.1109/ARES.2013.32

[39] Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. 2018. Neural network acceptability judgments. *arXiv preprint arXiv:1805.12471* (2018).

[40] Wikipedia contributors. 2019. Confusion matrix — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Confusion_matrix&oldid=906886050. [Online; accessed 28-August-2019].

[41] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. arXiv:cs.CL/1906.08237

[42] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 116–127.

[43] Xiao Zhang and Zhi Xu. 2018. On the Feasibility of Automatic Malware Family Signature Generation. In *Proceedings of the First Workshop on Radical and Experiential Security (RESEC '18)*. ACM, New York, NY, USA, 69–72. https://doi.org/10.1145/3203422.3203430

[44] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. 2010. Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics* 1, 1-4 (2010), 43–52.

[45] Bo Zong, Qi Song, Martin Renqiang Min, Wei Cheng, Cristian Lumezanu, Daeki Cho, and Haifeng Chen. 2018. Deep autoencoding gaussian mixture model for unsupervised anomaly detection. (2018).

[46] Barret Zoph, Deniz Yuret, Jonathan May, and Kevin Knight. 2016. Transfer learning for low-resource neural machine translation. *arXiv preprint arXiv:1604.02201* (2016).