EAI Endorsed Transactions
on Security and Safety
Research Article EAI.EU

Leveraging attention-based deep neural networks for security vetting of Android applications

Prabesh Pathak[1,*], Prabesh Poudel[1], Sankardas Roy[1,†], Doina Caragea[2]

[1]Bowling Green State University, Bowling Green, Ohio, USA
[2]Kansas State University, Manhattan, Kansas, USA

Abstract

Many traditional machine learning and deep learning algorithms work as a black box and lack interpretability. Attention-based mechanisms can be used to address the interpretability of such models by providing insights into the features that a model uses to make its decisions. Recent success of attention-based mechanisms in natural language processing motivates us to apply the idea for security vetting of Android apps. An Android app's code contains API-calls that can provide clues regarding the malicious or benign nature of an app. By observing the pattern of the API-calls being invoked, we can interpret the predictions of a model trained to separate benign apps from malicious apps. In this paper, using the attention mechanism, we aim to find the API-calls that are predictive with respect to the maliciousness of Android apps. More specifically, we target to identify a set of API-calls that malicious apps exploit, which might help the community discover new signatures of malware. In our experiment, we work with two attention-based models: Bi-LSTM Attention and Self-Attention. Our classification models achieve high accuracy in malware detection. Using the attention weights, we also extract the top 200 API-calls (that reflect the malicious behavior of the apps) from each of these two models, and we observe that there is significant overlap between the top 200 API-calls identified by the two models. This result increases our confidence that the top 200 API-calls can be used to improve the interpretability of the models.

Received on 14 July 2021; accepted on 03 August 2021; published on 27 September 2021

Keywords: Android Apps, Android Security, Malware Detection, Deep Neural Networks, Attention

doi:10.4108/eai.27-9-2021.171168

1. Introduction

Android is the most popular OS with 71.9% of global market share as of February 2021 [1]. Google Play is one of the largest Android app stores, hosting more than 3 million apps that are collectively used by billions of users. Unfortunately, this success story of the Android ecosystem also motivates adversaries to inject malicious apps into the app store for monetary gains and more. Once in a while, we see headline news of a malicious app landing up in Google Play breaching their security vetting system. This implies that all the users who installed that malicious app from Google Play on their Android phones could be vulnerable to the relevant attack. The potential attacks include forceful advertisements, sensitive credential theft, hidden surveillance, and many more. Even worse, a user can install apps from a third-party market which may not even enforce a security vetting process for the apps. Hence, there is a pressing need to build an efficient security vetting system for Android apps.

Approaches to malware detection include dynamic analysis and static analysis. Dynamic analysis involves running the code in a controlled environment and vetting based on the run-time behavior of the app. Alternatively, static analysis involves studying the app code without executing it – this can be the analysis of the source code (which can be in Java) itself or the executable (i.e., Dalvik bytecode), manifest files, other resource files, decompiled code (which can be in the form of smali code), etc.

*Corresponding author. Email: ppathak@bgsu.edu
†Corresponding author. Email: sanroy@bgsu.edu

In the current work, as the app artifacts we use the API-calls[1] that are obtained from the decompiled code, i.e., the smali files. By scanning the smali files, we extract the API-calls that are in the form of strings. A sample API-call is: *Ljava/io/BufferedWriter;.close:()V*.

We use the sequence of API-calls as artifacts to train the deep learning models. In particular, we get the embeddings of the API-calls via the *word2vec* [3] mechanism and feed them as the input to deep learning algorithms. We experiment with two deep learning algorithms as mentioned below.

Recurrent Neural Networks (RNN) represent a type of deep learning approach suited for learning from sequence-type data [4, 5]. Long Short Term Memory (LSTM) cells are RNNs that can potentially handle longer inputs. A bidirectional LSTM (Bi-LSTM) can find predictive dependencies in both forward and backward directions, typically giving better performance than a regular LSTM. Prior research [6, 7] in the domain of Neural Machine Translation (NMT) showed that a Bi-LSTM combined with the attention mechanism holds even better promise, as the use of attention mechanism allows the model to focus on the most important parts of the input. Motivated by this success, we choose an attention-based Bi-LSTM as the first deep learning algorithm to experiment with. Note that we build a classification model (instead of performing NMT) and analyze the attention weights to identify the most predictive features that reflect maliciousness of Android apps. Given that we rank the features based on their attention weights, we also refer to the most predictive features as *top features* in this paper.

In our second deep learning algorithm, we leverage the self-attention mechanism as used in the encoder block of *Transformer* [8]. In contrast to RNN-based attention where sequential processing is necessary (i.e., there is no parallelization), *Transformer* facilitates some parallelization during training. This parallelization allows training on larger datasets with potentially longer sequences. In particular, with multi-headed self-attention, we can process all input tokens at the same time and subsequently retrieve attention weights. These weights can be further used to rank the features and ultimately identify the top features.

A simplified flowchart of our work is shown in Fig. 1. The major contributions of our work are: **(a)** We built two attention-based classification models with Bi-LSTM Attention and Self-Attention, respectively. **(b)** Using the attention weights from these models, we identified top 200 API-calls for each model, which reflect maliciousness of Android apps. Interestingly, we found that 21 API-calls were common between the top 200 API-calls from these two models.**(c)** Finally, we collected an aggregate set of top 200 API-calls by combining the two individual models' output. These can be further studied by the research community to potentially discover new malware signature.

[1]Formally, an API-call is a method-invoking statement in the app's code whose implementation is not present within the app's package (i.e., the apk file) [2].

2. Related Works

Here we review prior works on Android malware detection, which are related to our work. We also briefly discuss the relevant deep learning approaches.

2.1. Static Analysis

FlowDroid [9] is a static analysis tool that can be used to detect certain malicious behaviors of an Android app. It, however, does not track Inter-component communications in an app. Amandroid [10] supports constructing inter-component CFG (control flow graph) and inter-component DFG (data flow graph) that can be used to detect multiple malicious signatures. However, building such CFG and DFG is computationally expensive.

2.2. Machine Learning (ML)

Drebin [11] used over 500k static features (extracted from the decompiled code and the manifest file) to train ML algorithms (e.g., SVM) to classify apps as malicious or benign. Out of those (500k) features, Roy et al. [12] handpicked (applying domain knowledge) only 471 features to train the ML models, and claimed to attain similar detection accuracy. However, with the evolving Android ecosystem, an automated feature selection approach is required (instead of manual handpicking) for building a scalable vetting system.

DroidAPIMiner [13] performed ML-based frequency analysis on 169 APIs to identify the most relevant APIs. The authors of another tool, MaMaDroid [14], claimed that their proposed tool outperforms DroidAPIMiner. MaMaDroid abstracts the API calls to their class, package, or family, and builds a model (in the form of a Markov chain) from the API call sequences obtained from the call graph of an app.

ApiChecker [15] is an ML-powered malware detection system that utilizes ground-truth data from a large Android app market called Market-X. The authors identified 426 key APIs in three categories: Restricted API, Highly-Correlation API, and Sensitive Operation API. Their API selection strategy is based on the SRC (Spearman's rank correlation coefficient) value of each API with the malice of apps.

2.3. Deep Learning (DL)

Xu et al. [16] performed multi-phase security vetting of Android apps. In the first phase, they feed the XML

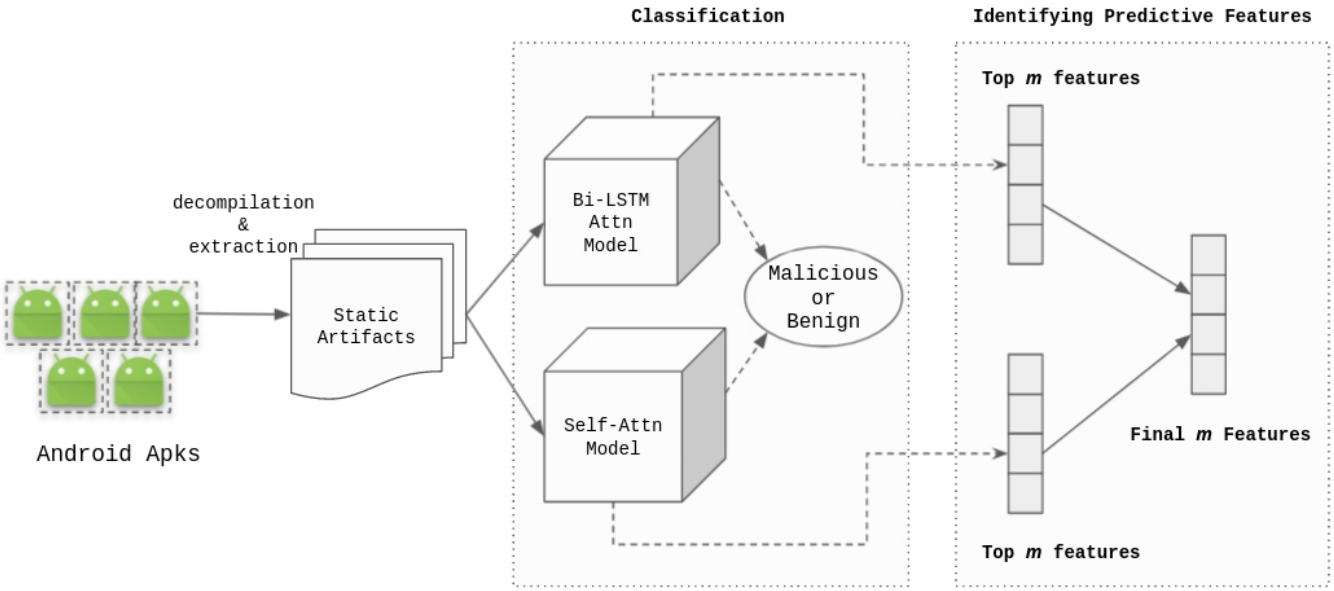

Figure 1. System design of attention-based security vetting of Android apps

artifacts (e.g., the manifest file in the app apk) to a multi-layer perceptron (MLP). Only the *uncertain apps* not meeting a specific threshold are passed on to the second phase. In the second phase, the authors use bytecode semantics to train an LSTM model.

Deep4MalDroid [17] extracts the (Linux kernel) system calls while executing Android apps under test. It performs classification using Stacked AutoEncoders (SAEs) where the feature set consists of the system calls.

The most closely related work to ours is MalDozer [18] that uses deep learning for security vetting of Android apps. MalDozer extracts raw sequences of API calls from the app executable (i.e., DEX file in the apk). The API calls are then tokenized and are populated in fixed-length vectors which are then used to train multi-layer Convolutional Neural Networks (CNNs). The main differences between our work and MalDozer can be summarized as follows: (a) we extract API calls from the decompiled app code (i.e., smali). (b) Our deep learning approach uses sequence models together with the attention mechanism, as opposed to CNNs.

Furthermore, Chaulagain et al. [2] built an Android app vetting system using variants of LSTM models. Amandroid [10] was used to collect API calls as static artifacts. System calls were collected as dynamic artifacts using the Genymotion [19] emulator. The hybrid approach makes the prediction based on the average of the probabilistic predictions of the static and dynamic models.

Attention. The attention mechanism was first proposed in the context of neural machine translation (NMT). Sutskever et al. [20] proposed an NMT architecture of Encoder-Decoder using LSTMs. The architecture uses a fixed-length context vector where all the necessary information from the input sequence is compressed. However, as the length of the input sentence increases, the performance deteriorates as demonstrated by Cho et al. [21].

Bahdanau et al. [6] proposed to address this issue using the attention mechanism which acts as an intermediate layer between encoder and decoder, and allows the decoder to selectively retrieve the information relevant to the input sequence. This type of attention is known as additive attention as it performs a linear combination of encoder and decoder states.

Luong et al. [7] improved Bahdanau's attention mechanism with a different method of alignment score calculation. They introduced the concepts of local and global attention, and proposed three different methods to calculate the alignment scores: *dot*, *concat* and *general*. Global attention takes all source hidden states into account, whereas local attention focuses only on a subset of the source hidden states.

Vaswani et al. [8] proposed the Transformer, a novel approach of processing sequences without the use of RNNs. Their model consists of a multi-head self-attention mechanism together a simple fully connected layer in the encoder, and a similar architecture for the decoder. Transformer attends to information from different angles, which is mathematically seen as different linear subspaces. For instance, consider two sentences: "The animal didn't cross the road because it was too tired." and "The animal didn't cross the road because it was too wide." In these sentences, the word *it* refers to animal and street, respectively. Such relations can be captured by Transformer. Transformer

has been the building block for state-of-the-art language representation models.

Recently, Wu et al. [22] proposed an attention-based mechanism to interpret Android malware classification results. In particular, they combined an attention-module with a Multi-Layer Perceptron (MLP) networks to find the attention weights of the input features. The difference between Wu et al.'s work and our work is as follows: (a) We use Bi-LSTM [23] or Transformer [8] models instead of an MLP network. Note that Bi-LSTM and Transformer are deep learning approaches specifically designed for sequence data, and Transformer architectures are state-of-the-art in NLP; (b) Our input feature set does not include app *permissions* or *intents* as in Wu et al.'s work. Thus, our work is complementary to Wu et al.'s work, and a direct comparison is beyond the scope of this paper.

3. Background

In this section, we present some background on the concepts of attention-based deep learning algorithms.

3.1. Deep Learning

Here, we discuss briefly on the deep learning algorithms used in our work.

LSTM. LSTM cells are special RNNs designed to address the vanishing gradient problem of traditional RNNs [24]. Each LSTM block consists of an input gate, a forget gate and an output gate. These gates determine which information from the previous step flows through to the next step in time. Their internal cell states can extract and hold temporal information hidden in input sequences. This allows the network to learn when to truncate the gradient and thus avoid vanishing gradients. LSTMs are only capable of leaning information from the past by parsing the sequence from left-to-right.

Bi-LSTM. To make use of both past and future information, Bi-LSTMs (which parse the sequence both from left-to-right and also from right-to-left) were introduced. Bi-LSTMs, also referred to as Bidirectional LSTMs, use two LSTMs in forward and backward manner to capture more information from the input sequence [23, 25]. In traditional LSTM, information flows in a forward direction with respect to time. In the case of API calls, looking at both directions when processing sequences could help the model make better inference using inherent dependencies of the function calls. Hence, the whole context of the input sequence can be taken into account.

Attention. The attention mechanism was originally proposed to address the RNN's imitations when processing longer inputs. Originated in the context of the encoder-decoder architecture in NMT [20, 26], the attention mechanism involves an additional layer after the encoding process where all the outputs produced by the encoder time steps are preserved and passed to the decoder at each time step. Bahdanau et al. [6] and Luong et al. [7] proposed encoder-decoder architectures with the attention mechanism, but the two groups used different score calculation methods.

Bi-LSTM Attention. Zhou et al. [23] proposed a Bi-LSTM with attention approach for relation classification tasks, inspired from Luong's attention. Let H be a matrix consisting of output vectors $[h_1, h_2 \ldots, h_n]$ that the LSTM layer produced, where n is the sequence length. The representation r of the sequence is formed by a weighted sum of these output vectors. Let α represent the attention weight of each sequence. We have:

$$\begin{aligned} M &= \tanh(H) \\ \alpha &= \text{softmax}\left(w^T M\right) \\ r &= H\alpha^T \\ h^* &= \tanh(r) \end{aligned} \tag{1}$$

where $H \in \mathbb{R}^{d^w * n}$, d^w is the dimension of the word vectors, w is a trained parameter vector and w^T is a transpose. The dimension of w, α, r is d^w, n, d^w respectively; h^* is the final sentence-pair representation which can be further used for classification. Chaulagain et al. [2] used this attention mechanism for malware classification.

Transformer and Self-Attention. Vaswani et al. [8] proposed the Transfomer, a novel approach of processing sequences without the use of recurrent cells. RNNs exhibit sequential processing which is incompatible with parallel computation. This can affect the robustness and efficiency when used on longer input sequence. The self-attention mechanism is fully parallelizable. The transformer model is an extension of the Encoder-Decoder structure consisting of self-attention blocks with linear layers and residual connections. It is a combination of multi-head self-attention layers and regular feed-forward neural networks. The use of self-attention ensures that long-distance dependencies can be captured effectively and efficiently.

Firstly, the embeddings from the inputs are combined with some positional information. The positional embeddings take into consideration the sequential nature of input data. This is typically carried out in the form of sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{2}$$

$$PE_{(\text{pos},2i+1)} = \cos\left(\frac{\text{pos}}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{3}$$

where pos denotes token's position, i denotes a specific dimension/index in the embedding, d_{model} is the embedding dimension. Hence, i belongs to the interval $[0, d_{model}]$. Each dimension of the positional embedding corresponds to a sinusoid.

The encoder in Transformer consists of a multi-head attention block followed by linear transformation with a non-linear activation function. Here, the motivation behind using multi-head self-attention is to give multiple perspective of the same input sequence.

Self-Attention in Transformer is an extension of the generic attention mechanism which makes use of queries Q, keys K and values V to calculate an alighment score. These three vectors provide alternative representations of the same positionally encoded input sequence. As in other forms of attention, the alignment here is computed on Q and K, and subsequently applied to V. Intuitively, the process is analogous to search engines, where a user's $query$ is matched against engine's $keys$ and the $values$ are the results.

Transformer uses a scaled dot-product attention as in Equation (4) to calculate the alignment score. The input consists of queries and keys of dimension d_k, and values of dimension d_v. The formula introduces a scaling factor to prevent the softmax function from giving values close to 1 for highly correlated vectors and values close to 0 for non-correlated vectors. This makes gradients easier to work with during back-propagation.

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

The multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. The vectors from each head are concatenated into one vector followed by a linear projection to the subspace of the initial dimension. The output from the attention block is then fed to a feed-forward network. There are residual connections around each block. These residual connections help keep track of data, and are followed by layer normalization which helps in reducing features variance. The decoder block is similar to the encoder block, except that it inserts a third attention sub-layer between multi-head self-attention and a piece-wise fully connected layer.

The transformer suffers from the issue of memory requirements. Self-attention layers are faster than recurrent layers when the sequence length n is smaller than the embedding dimension d, which is most often the case with sentence representations. With the increase in sequence length, the model has to include more and more parameters for both intermediate feed-forward layers and attention. Computational complexity per layer for RNNs is $O(n.d^2)$ whereas for self-attention is $O(n^2.d)$. However, massive computational power is required for training big transformer models. This indicates that such level of experiment is suitable only for large industrial research labs as it quickly becomes impossible to fit the model in a single GPU.

3.2. Preparatory Work

As a preparatory work, we initially tested our models on two freely available datasets: Amazon Reviews and MNIST.

Amazon Review Dataset. The Amazon Review Dataset is a freely available collection of user reviews, which has been widely used in the field of natural language processing for sentiment analysis. Amazon Review Dataset is based on English vocabulary. The goal of the classification task is to separate positive reviews from the negative reviews. On training our models with 200,000 reviews, we got impressive auPRC scores for classifying the test data, which is 0.9762 with Bi-LSTM-Attn and 0.9691 with Self-Attn, respectively.

An example review (after data cleaning) is as follows: *"fantastic i love this book absolutely a keeper the length of time it for the book to get to me was very short i am very satisfied."* Our models identified keywords such as $fantastic$, $love$ and $satisfied$ as predictive with respect to the classification decision. This set of keywords matches with the expectation of a human with knowledge of the English language.

MNIST Dataset. MNIST is a widely used image dataset of handwritten digits. It has thousands of grey-scale images of handwritten digits from '0' to '9'. The goal of the classification task is to identify individual digits, e.g., '0', '1', etc. The dimension of the image is 28 × 28 pixels. We test how our attention-based models can identify the important features of the images. In other words, we want to identify which pixels of the image is more important (in identifying the digit) compared to other pixels of the image.

We trained our models with 48,000 images from MNIST. On evaluating our models on MNIST images, we found that the top features (aka. pixels) have darker shades and are at the central part of the image. This gives us some visual explanation on how the model was able to classify the test data correctly. Both of our algorithms are able to identify the top features (aka. pixels) of the images.

4. Approach

4.1. APK Collection and Artifacts Extraction

In this work, we use a public repository named AndroZoo [27] as our primary source of Android apks. Both benign and malicious apps along with the VT (Virus Total) scores are available in AndroZoo. the

VT score represents the number of anti-virus engines that have marked that app as malware. We divide the set of malware apps into two sections based on their VT scores: Apps with VT score between 2 and 10 are considered as low quality malware, whereas apps with VT score greater than 10 are considered as high quality malware. Apps with VT score zero are considered as benign apps.

Android apps frequently use *android* library and *java* library. That means most of the APIs that an Android app invokes are implemented in these two libraries. With the popularity of the Android ecosystem and it being an open community, several third-party libraries are also available to be used by Android apps. We identified such 132 popular third party libraries. That means if an Android app invokes a method implemented in one of these libraries, we consider it as an API call. So, there are three main categories of APIs under our consideration: Android library, Java library, and third party libraries.

We used a Python script that uses apktool to decompile an app to smali code, and obtain the sequence of API-calls from the smali file. Note that this sequence of API-calls represents a true execution sequence only within the boundary of a *code block*[2]. In other words, the actual execution sequence of the API-calls may be different based on the actual execution order (that is decided only in run time) of multiple code blocks [2, 18] with respect to each other. Note that to establish the true sequence, we need to build inter-procedural control flow graph (CFG) of the app, which is computationally expensive. So, we make a trade-off: our process of collecting artifacts is lightweight but we get only a quasi-sequential order of API-calls. Finally, we store the API-calls (as artifacts) in a text file where each API-call obtained is delimited from the next one by a space.

4.2. Deep Learning

We experiment with two deep learning algorithms: Bi-LSTM attention and Self-attention. In particular, using each such algorithm, we build one deep learning model as sketched in Fig. 1 via training with the aforementioned artifacts.

Fig. 2 shows an overview of the Bi-LSTM attention architecture. This involves two LSTM cells unfolding in forward and backward directions, respectively, and capturing information from the input sequences. Afterwards, the outputs are sent to an attention layer from where we extract the attention weights. This output is then fed to a feed-forward neural network for the classification task.

[2]A code block is a linear chunk of code, i.e., it has no branching out.

Fig. 3 shows us the architecture of the self-attention model. In this scheme, we first combine the *word2vec* embedding with positional embedding module. We use positional embedding to preserve the order of the "words" (i.e., APIs) in the sequence. The combined embeddings are then fed to the encoder of a multi-head self-attention block with layer normalization. The output is then passed to a feed-forward classification network.

For either of the two models, we perform hyper-parameter tuning by observing the training and validation losses.

4.3. Selection of Top Features

We identify a set of top features from each of the two deep learning models. To do so, we analyze the attention weights of the model after the training phase is completed. In particular, for each model we identify the top m API-calls using the 'true positive' malware apps in the test dataset. Let the set of true positive malicious apps be P. Note that an API-call can appear multiple times in a single app, and it can have different attention weights for the same app based on its position in the sequence. To address this issue, we take a cumulative weight of an API-call; actually, we take the cumulative weight of an API-call across all the true positive malware apps. This also helps us deal with any possible outlier. We store the unique API-calls along with the corresponding weights in a hash map, and then sort them to get the top m API-calls. After obtaining top m features from each of the two models, we also check for any overlapping features. In summary, we take the following steps to get the aggregate top features, representing both of our models.

1 For each app we take the first n API calls; if an app has more, we discard the later API calls.

2 For each model j:

- 2a Softmax scores (i.e., attention weights) are assigned to each API call in an app.
- 2b API calls can be repeated within an app; we cumulatively add the weights.
- 2c Furthermore, we cumulatively add the weights of a repeated API call across P, the set of true positive malicious apps.
- 2d Get top m ranked API calls that is denoted by set A_j.

3 Compute union U of the two sets, A_1 and A_2. Sort the union set U according to the corresponding weights. Identify the top m API-calls from U, which makes the aggregate top features.

4 Compute intersection I of the two sets, A_1 and A_2.

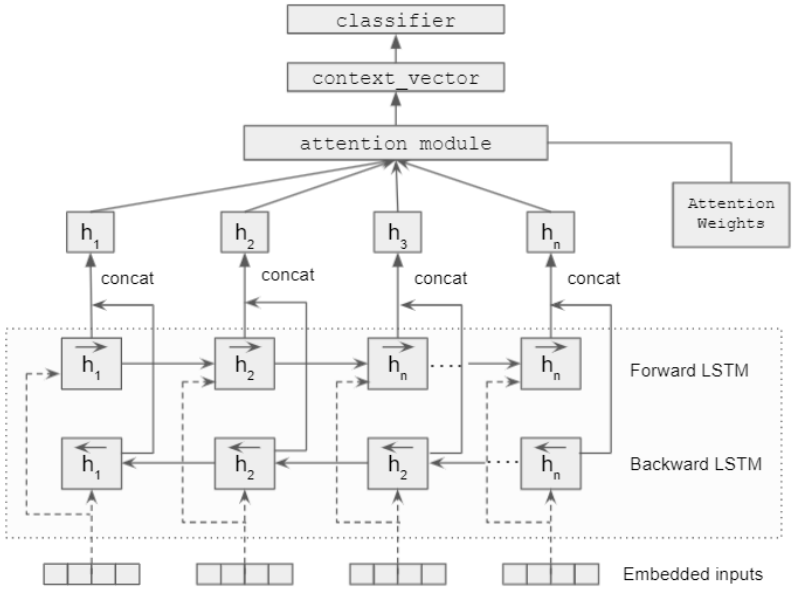

Figure 2. Bi-LSTM attention architecture

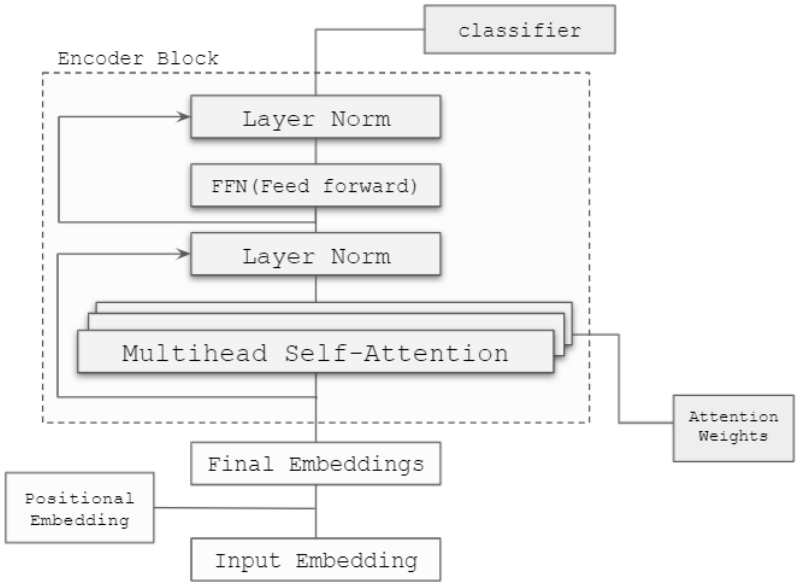

Figure 3. Self-Attention architecture

In step 1, the value of n is chosen such that for most apps no API is discarded. For our app dataset, we found that a good value of n is 4000. Furthermore, in our work, we chose 200 as value of m. Also, for step 4, note that the API-calls in set I appear in both the sets, A_1 and A_2. The bigger the size of I, the more confident we are in identifying the top predictive features.

Note that alternative schemes to identify top API-calls are possible, and will be explored in the future. For instance, as an alternative scheme to the current version of step 2, we could specifically aim to capture an API with one occurrence but a very high weight in an app versus an API with many occurrences but smaller weights in an app (such APIs can be ranked as top features in the current version of step 2). Differentiating between the two types of APIs identified as top features can help security analysis to identify interesting patterns explored by the coders of malicious apps.

5. Experimental Setup and Implementation

Here we discuss the experimental setup along with the experiment specifics.

Table 1. The App Dataset

Year	Benign apks	Malicious apks
2016	30,000	10,000
2017	30,000	10,000
2018	30,000	10,000
2019	30,000	10,000
2020	9000	3000
Total	**129,000**	**43,000**

5.1. Dataset

Our dataset comprises of Android apks that were published over the last 5 years (2016-2020). We use 40, 000 apks each from years 2016 to 2019. We collected only 12, 000 apks from year 2020. To maintain the data balance factor over the years, we kept the malicious app to benign app ratio at 1 : 3 for each year. Note that in real life also, malicious apps are far less in number than benign apps in an app store. The dataset composition is shown in Table 1. Note that we have a total of 43,000 malware apps in the dataset.

To emulate the real life situation, we include both high quality malware apps and low quality malware apps in the dataset as discussed in Section 4.1. The detailed division of these two types of malware over the years is shown in Table 2.

5.2. AWS Setup

Setup for Artifacts Extraction. For artifacts extraction, we use an Amazon's EC2 instance with Ubuntu Version 39.0 as our AMI (Amazon Machine Image). Artifacts extraction process requires a machine with high memory power and large number of cores. So, we use memory optimized instance type $r5.8xlarge$. It comes with 32 cores and 256 GB of memory. We also require external EBS to copy the apks from S3 bucket.

Setup for Deep Learning. For the deep learning tasks, we use Amazon's EC2 instance with Deep Learning Ubuntu Version 40.0 as the AMI. This comes with pre-installed data science libraries and frameworks. To facilitate deep learning, we need to use an instance with high computational power. So we choose, $p3.2xlarge$ as our instance which comes with 8 cores and 61GiB of RAM. It delivers high performance with 8 NVIDIA V100 Tensor Core GPUs and up to 100 Gbps of networking throughput. We use *ssh* to connect to the instance. We also define security group to forward the port 8888 which can be used to run Jupyter notebook on the local machine.

5.3. Artifacts Extraction

On successful decompilation of the Android apk using apktool, we get smali files and more. We scan smali files to find keywords *.method*, *invoke–* and *.endmethod* keywords (that do not represent a user-defined function) to spot API-calls. We list API-calls of an app in a text file, separating each API-call by a space and delimiting each function boundary with a # symbol. As noted in Section 4.1, the API-calls of an app are listed in quasi-sequential order only.

Over our total dataset of 172, 000 apps, we compute the distribution of the source of APIs, which is illustrated in Figure 4. We observe that *android* and *java* APIs comprise of the majority of the APIs in the vocabulary. Note that the total number of unique APIs (i.e., the vocabulary length of the corpus) is 858, 193.

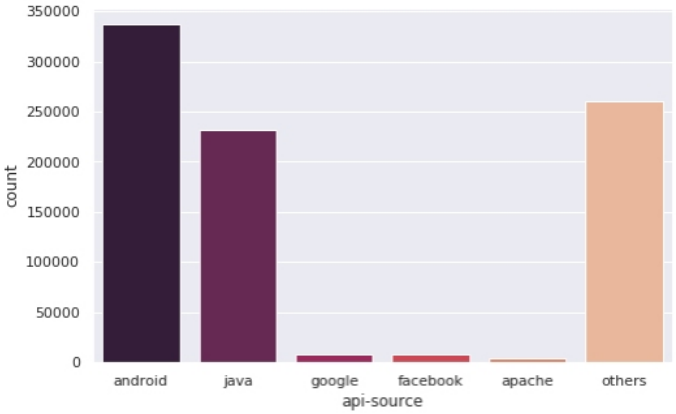

Figure 4. The distribution of the API Source

5.4. Implementing the Deep Learning Approaches

The first step is to obtain the embeddings of the app artifacts via *word2vec* algorithm. We get embeddings of three different dimensions: 128, 256 and 512. The length of our input sequence of API-calls is 4000. This is a large number which, along with the embedding dimension, significantly increases the computational complexity per layer, especially when self-attention is used.

We build our deep learning models using two approaches: Bi-LSTM Attention and Self-Attention.

Fig. 5 gives a detailed overview of the Bi-LSTM Attention approach with matrix dimensions. The heart of the computation lies in Equation 1. Bi-LSTM gives an output in the form of [*batch_size* x *seq_len* x *hidden_units*], which is fed to the attention module. The output from the attention module gives our feature representation in terms of attention weights. This is amplified again with multiplication with input as Batch-wise Matrix Multiplication (BMM) to get a context vector. This vector is then is then fed to the classifier network.

The hyper-parameters in our experiments were finalized based on the training and validation loss observed through the epochs. Other parameters not

Table 2. The Malicious dataset

Year	# of Malicious Apps	Low Quality Malware	High Quality Malware
2016	10,000	8999	1001
2017	10,000	8599	1401
2018	10,000	9005	995
2019	10,000	8507	1493
2020	3000	2920	80
Total	**43,000**	**38,030**	**4970**

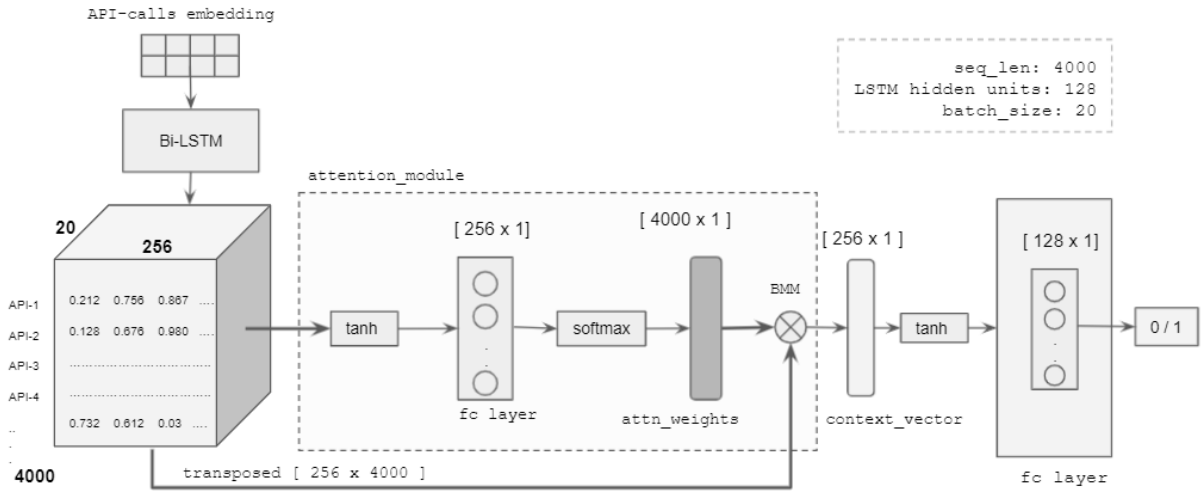

Figure 5. Detailed Bi-LSTM Attention

mentioned are set as default parameters provided by the PyTorch library.

For Bi-LSTM Attention, we get our best performing model with the following hyper-parameters: 128 LSTM hidden units, Adam Optimizer with learning rate of 0.0005, weight decay of $5e-4$ and dropout of 0.3. We train this model for 20 epochs with a batch size of 20. The training app set is as in Table 3. Total training time for 20 epochs was about 7 GPU hours.

As Self-Attention is computationally expensive, we set the batch size to 2 and train the model for 20 epochs. The training app set is as in Table 3. The best performing hyper-parameters are: Adam Optimizer with learning rate of 0.0005 and a decay factor $5e-4$, embedding dimension of 256, 2 attention heads and 16 hidden units in the dense layer of encoder block. It took around 17 GPU hours to train this model.

6. Evaluation

In this section, we present the results of our experiments with Bi-LSTM Attention and Self-Attention. We consider auPRC (Area Under Precision Recall Curve) [12] as the core evaluation metric. Then, we present results from a few additional experiments, e.g., comparison of the two deep learning algorithms' malware detection accuracy with that of some traditional ML algorithms [12].

Training and Test Data Split. Table 3 gives an overview of the train-test split of the (app) dataset that we used in the experiments. Note that the same dataset is used for both Bi-LSTM Attention and Self-Attention. Furthermore, we used 20% of the training data as the validation dataset.

6.1. Malware Detection Accuracy

We now evaluate our two deep learning models on the API-calls dataset. As shown in Table 3 , the test dataset contains 17200 apps: 4300 malware apps and 12900 benign apps.

Bi-LSTM-Attention Model. Fig. 6 (Left) shows the confusion matrix of the Bi-LSTM-Attention model on the test dataset. We see that 634 malicious apps were falsely classified as benign apps. Furthermore, the auPRC is 0.9391 whereas the Precision-Recall curve (PRC) is illustrated in Figure 7 (Left).

Self-Attention Model. Fig. 6 (Right) shows the confusion matrix of the Self-Attention model on the test dataset. We see that 955 malicious apps were falsely classified

Table 3. The train-test data split

Year	Ben-train	Mal-train	Ben-test	Mal-test	Total
2016	27,000	9000	3000	1000	40,000
2017	27,000	9000	3000	1000	40,000
2018	27,000	9000	3000	1000	40,000
2019	27,000	9000	3000	1000	40,000
2020	8100	2700	900	300	12,000
Total	**116,100**	**38,700**	**12,900**	**4300**	**172,000**

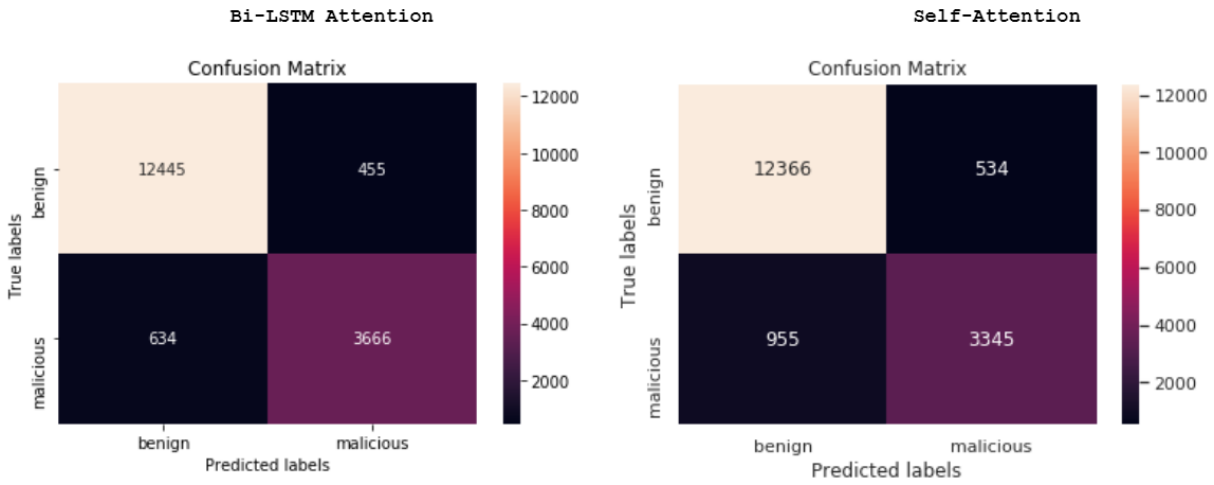

Figure 6. Confusion matrix for Bi-LSTM Attention (Left) and Self-Attention (Right)

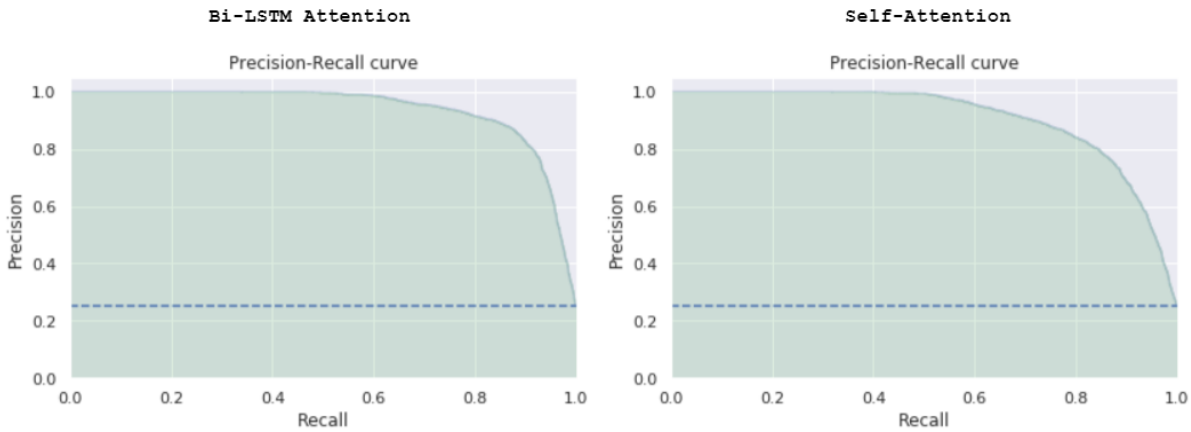

Figure 7. Area under PR-Curve with Bi-LSTM Attention (Left) and Self-Attention (Right)

as benign apps. We found that 850 out of 955 were low-quality malware. We also note that a large proportion of the training dataset was comprised of low-quality malware which possibly explains the large-number of false positives. Furthermore, the auPRC is 0.9074 whereas the PRC is illustrated in Figure 7 (Right).

Comparison of Bi-LSTM Attention and Self-Attention Models. When we compare and observe the results from the two deep learning approaches, we see that both models give competitive results. Bi-LSTM Attention has a slight edge over the Self-Attention model. Training Self-Attention is computationally expensive and hyper-parameter tuning was difficult as compared to Bi-LSTM attention. Thus, we employed a very simple Self-Attention model which can be potentially improved with further tuning, a task that we leave as part of future work.

Table 4. Comparison of the two deep learning models

Model	App Type	Precision	Recall	F1-Score	Area Under PR Curve
Bi-LSTM Attention	Benign	0.9515	0.9647	0.9580	0.9391
	Malicious	0.8895	0.8525	0.8706	
Self-Attention	Benign	0.9283	0.9586	0.9432	0.9074
	Malicious	0.8623	0.7779	0.8179	

6.2. Top Feature Selection

To identify the top features, we only make use of the true positive apps, i.e., the malware that were correctly predicted.

Fig. 8 lists the top 10 API calls of a true-positive malware as computed by the Self-Attention model. The top API call on the list is: *ljavax/crypto/spec/deskeyspec*. In this API-call, DES (Data Encryption Standard) is a symmetric-key block cipher. This indicates that our model is capturing relevant top features.

Fig. 9 shows an overview of how the selection process occurs. For the sake of readability, this figure illustrates the selection process with a tiny dataset of two apps where the goal is to identify only 4 top features. Here, even though y ranks second in both the apps, it ends up being the top feature. We believe that this deals with the bias of single occurring highly scored features like m.

To aggregate the API-calls from Bi-LSTM Attention and Self-Attention models, we do a simple concatenation and get the combined top 200 API-calls. We found 21 overlapping API-calls. Our total vocabulary was 850,193 and we were able to find 21 common features between our two models. That is more than 10% of common features which we believe is a good number considering such big vocabulary. Table 5 shows the common API-calls between the two models.

6.3. Results from additional experiments

Comparison with Traditional ML-Algorithms. Roy et al. [12] used traditional machine learning models with 471 handpicked features to detect malware apps. We use Roy et al.'s feature extraction scheme to extract those 471 features for our 172k apps. We use the same train-test split as in Table 3. The results obtained are presented in Table 6. The result might look competitive here, but we note that ML requires handpicked features whereas the deep learning models learn the features themselves. Chaulagain et al. [2] also used 471 features on their dataset and got auPRC of 0.9819 with SVM. This high auPRC could be a result of using only high quality malware (both for training and testing) and an older dataset dated before 2016.

We should note that traditional ML techniques such as Random Forest can also rank the features according to their importance. However, we did not consider ranking based on Random Forest in this work, as Random Forest cannot leverage the sequential nature of our artifacts (i.e., API-calls) and can suffer from scalability issues and overfitting if the whole set of APIs present in the apps (i.e., approximately 858K APIs) is used in training. As opposed to that, deep learning techniques (such as LSTM and Transformer) can use sequential data that may include a large number of APIs.

Imbalanced Ratio Experiment. In this experiment, we vary our malicious to benign apps ratio in our test dataset and discuss the results. We test with 5 different ratios: 1:3, 1:5, 1:10, 1:20 and 1:50, varying number of malicious apps and keeping the number of benign apps constant. Fig. 10 shows that auPRC of our model decreases greatly as we decrease the ratio of malicious to benign apps. We also notify the readers that in real world this ratio is very low. Thus, when building an Android vetting system with a classification model, researchers should be wary about the ratio of malware to benign apps. The malware percentage in the test set should approximate the real world malware percentage [28].

High Quality Malware Experiment. To observe the effect of high and low quality malware, we conduct another experiment by varying the malware quality in our test dataset. Out of 4300 malicious apps in test dataset, we had only 512 high-quality malware. Thus, we brought down the number of benign apps to 1536 to match the original 1 : 3 ratio of malicious to benign apps.

Table 7 shows that the performance of the model increases when we deal only with high quality malware. Our model was trained with few high quality malware. We believe that on increasing the number of high quality malware during training, we can expect better results. Due to time and resource constraints, we leave such experiments as future work.

7. Limitations and Future Works

We identified a few possible directions for the future work, which are as follows.

In the current work, the sequence of API-calls is not obtained via dynamic analysis whereas for a malicious app the true sequence of API-calls can be decided only at the runtime. We plan to analyze the sequence of API-calls collected via dynamic analysis in a future work.

ljavax/crypto/spec/deskeyspec;.<init>:([b)v	0.057388
landroid/app/application;.<init>:()v	0.056203
ljava/lang/reflect/field;.gettype:()ljava/lang/class;	0.037152
ljava/lang/class;.forname:(ljava/lang/string;)ljava/lang/class;	0.027028
lorg/json/jsonarray;.tostring:()ljava/lang/string;	0.014487
ljavax/crypto/cipher;.getinstance:(ljava/lang/string;)ljavax/crypto/cipher;	0.011853
ljava/io/bytearrayoutputstream;.flush:()v	0.008844
ljava/io/randomaccessfile;.writelong:(j)v	0.007683
landroid/widget/toast;.maketext:(landroid/content/context;ljava/lang/charsequence;i)landroid/widget/toast;	0.005666
ljava/lang/reflect/array;.getint:(ljava/lang/object;i)i	0.003103

Figure 8. API-calls and attention weights for an Android malware app

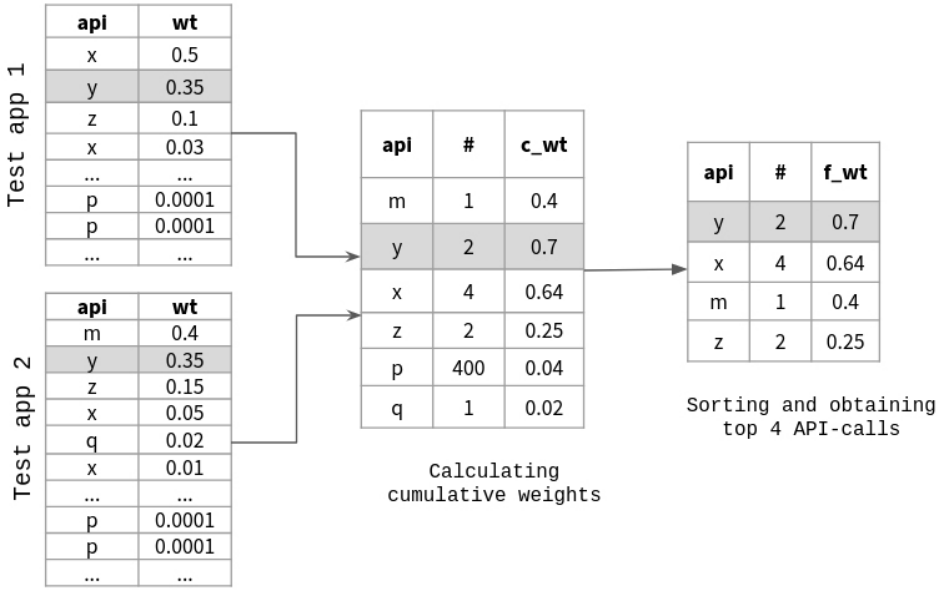

Figure 9. Top feature selection strategy: For the sake of readability, here we illustrate the selection process with a tiny set of two apps whereas the goal is to identify only top 4 features.

In the current work, we finally extracted 200 top features of malicious apps. In order to prove that these top features are valid, we plan to compare them with the manually extracted features in a future work.

Someone may wonder that instead of considering so many individual API calls as in the current work, whether it is better to merge API calls into groups such as permission related, string related, File/Network IO related, and so on. Although the features will then become more coarse-grained, it is important to explore this route in the future.

8. Conclusions

We leveraged attention-based deep learning approaches for security vetting of Android apps. API-calls extracted from the Android apps were used as the artifacts for deep-learning models. Such API-calls are quasi-sequential in nature. We experimented with two attention-based models: Bi-LSTM Attention and Self-Attention. Both models gave competitive results. Additionally, after analyzing the attention weights from these two models, we identified top 200 API-calls that reflect the maliciousness of an Android app.

Our experiments show that deep learning models can be implemented for large scale Android app security vetting. This can save human time and effort from manually handpicking the malicious features. The top features identified from the models can be further studied by the research community to potentially discover new malware signature.

Table 5. Common API-calls from the two models

Common API-calls
ljava/io/file;.mkdir:()z
ljava/io/file;.exists:()z
ljava/io/fileinputstream;.close:()v
ljava/lang/reflect/method;.setaccessible:(z)v
landroid/content/res/resources;.getassets:()landroid/content/res/assetmanager;
ljava/io/file;.delete:()z
ljava/io/filenotfoundexception;.printstacktrace:()v
landroid/telephony/telephonymanager;.getdeviceid:()ljava/lang/string;
landroid/os/asynctask;.onpostexecute:(ljava/lang/object;)v
ljava/io/file;.listfiles:()[ljava/io/file;
landroid/content/res/resources;.getboolean:(i)z
landroid/util/base64;.decode:([bi)[b
landroid/app/application;.onconfigurationchanged:(landroid/content/res/configuration;)v
ljava/lang/stringbuilder;.insert:(iljava/lang/string;)ljava/lang/stringbuilder;
lorg/apache/http/statusline;.getstatuscode:()i
landroid/content/sharedpreferences;.edit:()landroid/content/sharedpreferences$editor;
ljava/io/objectoutputstream;.writeobject:(ljava/lang/object;)v
landroid/app/application;.onterminate:()v
ljava/io/inputstream;.mark:(i)v
ljava/lang/process;.destroy:()v
ljava/util/zip/zipinputstream;.closeentry:()v

Table 6. Accuracy of standard ML algorithms

ML Model	App Class	Precision	Recall	F1-Score	Area Under PR Curve
Bernoulli Naive Bayes	Malicious	0.6209	0.6105	0.6156	0.6845
	Benign	0.8709	0.8757	0.8733	
K-Nearest Neighbor (K=5)	Malicious	0.9071	0.8881	0.8975	0.9471
	Benign	0.9630	0.9697	0.9663	
Support Vector Machine	Malicious	0.9133	0.8940	0.9035	0.9547
	Benign	0.9649	0.9717	0.9683	

Table 7. Deep learning models' accuracy with high quality malware

Model	**Review Type**	**Precision**	**Recall**	**F1-Score**	**Area Under PR Curve**
Bi-LSTM Attn.	Benign	0.9554	0.9772	0.9662	0.9548
	Malicious	0.9266	0.8632	0.8938	
Self Attention	Benign	0.9283	0.9586	0.9432	0.9291
	Malicious	0.9084	0.7949	0.8479	

Acknowledgement. This work has been partially supported by the U.S. National Science Foundation (NSF) under grant no. 1718214 and 1717871. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Statcounter (2021), Android OS Market Share. URL https://gs.statcounter.com/os-market-share/mobile/worldwide.

[2] Chaulagain, D., Poudel, P., Pathak, P., Roy, S., Caragea, D., Ou, X. and Liu, G. (2020) Hybrid analysis of android apps for security vetting using deep learning. In *IEEE conference on communications and network security (CNS)*.

[3] Pascanu, R., Mikolov, T. and Bengio, Y. (2012) Understanding the Exploding Gradient Problem. *arXiv e-print* : arXiv:1211.5063.

[4] Bengio, Y., Simard, P. and Frasconi, P. (1994) Learning Long-term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks* : 157–166.

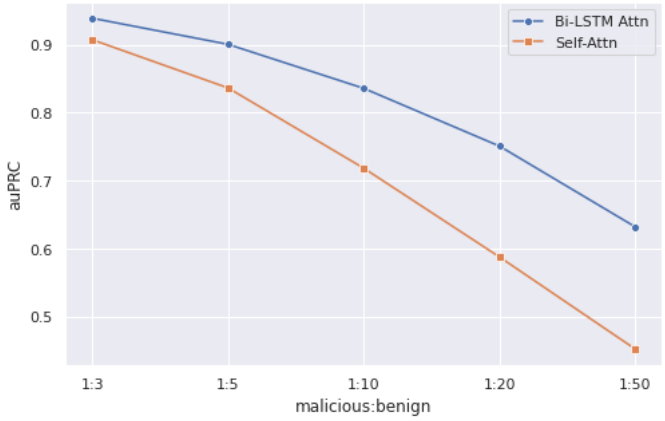

Figure 10. Experimenting with varying malware apps to benign apps ratio in the test set

[5] Gers, F.A., Schmidhuber, J. and Cummins, F. (1999) Learning to forget: continual prediction with lstm. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*: 850–855.

[6] Bahdanau, D., Cho, K. and Bengio, Y. (2016), Neural machine translation by jointly learning to align and translate. 1409.0473.

[7] Luong, M.T., Pham, H. and Manning, C.D. (2015), Effective approaches to attention-based neural machine translation. 1508.04025.

[8] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. *et al.* (2017), Attention is all you need. 1706.03762.

[9] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y. *et al.* (2014) FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *SIGPLAN* : 259–269.

[10] Wei, F., Roy, S., Ou, X. and , R. (2018) Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. *ACM Transactions on Privacy and Security* : 1–32.

[11] Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H. and Rieck, K. (2014) DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Symposium on Network and Distributed System Security (NDSS)*: 23–26.

[12] Roy, S., DeLoach, J., Li, Y., Herndon, N., Caragea, D., Ou, X., Ranganath, V.P. *et al.* (2015) Experimental study with real-world data for android app security analysis using machine learning. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*: 81–90.

[13] Aafer, Y., Du, W. and Yin, H. (2013) DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *International conference on security and privacy in communication systems*: 86–103.

[14] Onwuzurike, L., Mariconti, E., Andriotis, P., De Cristofaro, E., Ross, G. and Stringhini, G. (2017) MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. *arXiv e-prints* : arXiv:1612.04433.

[15] Gong, L., Li, Z., Qian, F., Zhang, Z., Chen, Q., Qian, Z., Lin, H. *et al.* (2020) Experiences of landing machine learning onto market-scale mobile malware detection. *Proceedings of the Fifteenth European Conference on Computer Systems* .

[16] Ke, X., Li, Y., Deng, R.H. and Chen, K. (2018) DeepRefiner: Multi-layer Android Malware Detection System Applying Deep Neural Networks. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*: 473–487.

[17] Hou, S., Saas, A., Chen, L. and Ye, Y. (2016) Deep4MalDroid: A Deep Learning Framework for Android Malware Detection Based on Linux Kernel System Call Graphs. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*: 104–111.

[18] Karbab, E.B., Debbabi, M., Derhab, A. and Mouheb, D. (2018) MalDozer: Automatic Framework for Android Malware Detection using Deep Learning. *Digital Investigation* : S48–S59.

[19] (2018), Genymotion Android Emulator. URL https://www.genymotion.com/.

[20] Sutskever, I., Vinyals, O. and Le, Q.V. (2014), Sequence to sequence learning with neural networks. 1409.3215.

[21] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H. and Bengio, Y. (2014), Learning phrase representations using rnn encoder-decoder for statistical machine translation. 1406.1078.

[22] Wu, B., Chen, S., Gao, C., Fan, L., Liu, Y., Wen, W. and Lyu, M.R. (2021) Why an android app is classified as malware: Toward malware classification interpretation. *ACM Trans. Softw. Eng. Methodol.* **30**(2).

[23] Zhou, P., Shi, W., Tian, J., Qi, Z., Li, B., Hao, H. and 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*: 207–212.

[24] Hochreiter, S. and Schmidhuber, J. (1997) Long Short-Term Memory. *Neural Computation* : 1735–1780.

[25] Schuster, M. and Paliwal, K. (1997) Bidirectional Recurrent Neural Networks. *IEEE Transactions on Signal Processing* : 2673–2681.

[26] Cho, K., van Merrienboer, B., Bahdanau, D. and Bengio, Y. (2014), On the properties of neural machine translation: Encoder-decoder approaches. 1409.1259.

[27] Allix, K., Bissyandé, T.F., Klein, J. and Le Traon, Y. (2016) AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories*: 468–471.

[28] Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J. and Cavallaro, L. (2019) TESSERACT: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium*: 729–746.