LEVERAGING ATTENTION-BASED DEEP NEURAL NETWORKS FOR SECURITY VETTING OF ANDROID APPLICATIONS

Prabesh Pathak

A Thesis

Submitted to the Graduate College of Bowling Green
State University in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

May 2021

Committee:

Sankardas Roy, Advisor

Robert C. Green II

Qing Tian

ABSTRACT

Sankardas Roy, Advisor

Traditional machine learning and deep learning algorithms work as a black box and lack explainability. Attention-based mechanisms come to the rescue to address interpretability of such models by providing insights and explaining why a model makes its decisions. In recent years, attention-based mechanisms have been able to explain the role of the inputs in natural language processing. This success motivates us to extend these attention-based approaches for security vetting of Android apps. An Android app's code contains API calls that can give us some clue of if the app is either malicious or benign. By observing the pattern of such API calls being invoked, we can build a model to separate benign apps from malicious apps. In this thesis work, using attention-based mechanisms, we aim to find the important API calls that are responsible for reflecting the maliciousness of android apps. As an example, we target to identify a subset of APIs that malicious apps exploit, which might help the community discover a new signature of malware. In our experiment, we work with two attention models: Bi-LSTM Attention and Self-Attention. Our classification models achieve an area under the precision-recall curve (AUC) score of 0.9391 and 0.9074 with Bi-LSTM Attention and Self-Attention, respectively. Using the attention weights from our models, we are also able to extract top 200 API-calls from each of our models that reflect the malicious behavior of the apps.

ACKNOWLEDGMENTS

I would like to take this opportunity to address the people without whom I wouldn't have completed this thesis. Firstly, I would like to thank my advisor Dr. Sankardas Roy for his continuous guidance, support and encouragement. Next, I am grateful to my committee members: Dr. Robert Green and Dr. Qing Tian for their valuable suggestions. Additionally, I wish to express my gratitude to Dr. Doina Caragea from Kansas State University for helping me understand various deep learning concepts and providing AWS resources. I would also like to thank Guojun Liu and Dr. Xinming Ou from University of South Florida for their valuable comments. Finally, I am thankful to my family members and friends who supported me throughout this journey.

This work was partially supported by the U.S. National Science Foundation under grant numbers: 1718214, 1717871 and 1717862.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

Mobile phones are one of the most important entities of modern life. They have simplified our means of communication. They are also used for entertainment and even for reaching our destination through inbuilt GPS. Such handheld devices play a crucial role in our life on a daily basis. All such smartphones operate under different operating systems (OS). Android is the most popular OS with 71.9% of global market share as of February 2021 [1]. Google Play being one of the largest android app stores itself hosts more than 3 million apps. One of the reasons for the popularity of Android is that the users can download and install apps from any third-party markets. In the absence of a proper security vetting process, Android users can be vulnerable to security threats. Users have been victim of several malicious activities such as forceful advertisements, sensitive credential theft, hidden surveillance and so on. Some forms of Android malware have been developed with advanced evasion techniques in order to remain undetected on infected devices.

Considered one of the most advanced mobile malware in 2019, Triada was found pre-installed on a number of Android smartphones, infecting hundreds of thousands of victims. Google itself published a report stressing that during the production process, Triada was injected into the system image of mobile devices through a third party [2]. Google tracked a malware family named Bread for 3 years and recently removed 1.7k unique Bread apps from the Play Store [3]. This malware takes advantage of mobile billing techniques involving the user's carrier. Currently Google employs Play Protect in the play store with machine learning backed engines to deal with such malware. With the evolving Android ecosystem, the attackers have similarly evolved in novel and evasive ways of injecting malware into an app. Android apps can be uploaded to or downloaded from various third-party marketplace, which may or may not perform accurate malware checks. The growing number of android apps makes it impossible to manually analyze each app.

Two of the most prominent research oriented techniques for malware detection are dynamic and static analysis. Dynamic analysis involves running the code in a controlled environment and analyzing based on the action and behavior of the app. Alternatively, static analysis involves studying the code of the app without executing the app itself. This can be the analysis of the source code itself or bytecode, manifest files, resources, smali files, etc. Our work revolves around static analysis. We construct our artifacts using the API-calls obtained from the smali files after decompiling an app. Smali files are generated during DEX file decompilation. On scanning such files, we can extract the API-calls in the form of strings. Formally, an API-call is a method-invoking statement in the app's source code whose implementation is not present within the app's package [4]. A sample API-call looks like: *Ljava/io/BufferedWriter;.close:()V*.

We use the sequence of API-calls as input for our deep learning models. Recurrent Neural Networks (RNN) is the deep learning model suited for learning from sequences. But the efficiency of RNN decreases as the input sequence length increases. Learning such long term dependency on RNNs with gradient descent based learning is not efficient. It can lead to vanishing or exploding gradients [5][6]. Long Short Term Memory (LSTM) cells can deal with the long term dependency learning of RNNs. The use of stateful gates in LSTM can retain information within the cells. LSTM cells process the input tokens in a sequential manner. This is done so that the hidden state vector retains the information of the data seen in a sequential order. The modification of LSTM as Bidirectional LSTM (Bi-LSTM) to retain information in both forward and backward direction gave better results. Bi-LSTM was further improved with an attention mechanism which yielded better results in Neural Machine Translation (NMT) [7] [8]. The use of attention mechanism allowed the model to focus on the important part of inputs. Our first model is based on this approach. Instead of NMT, we build a classification model and analyze the attention weights to select the top features.

Our second model is based on the encoder block of *Transformer* [9] which uses multi-head self-attention. In RNN based attention, sequential processing is necessary which prevents parallelization. Transformer model facilitates more parallelization during training, which allows training on larger datasets with longer sequences. Transformer model introduces multi-headed self-attention which involves processing all input tokens at the same time and calculating attention weights. We analyze the attention weights from multi-head self-attention block to select our top features.

We build two separate classifiers based on two approaches: Bi-LSTM Attention and Self-Attention. We then analyze the attention weights from these models. This can help us find the important features from our input. The goal of our work is to build two separate classification models and to collect top features from both these models. With attention, we can look more on how the model is capable of attending to relevant features. Such explainability of a deep learning model is very rare. Thus we believe that attention mechanisms can give potential insights on the decision making capability of a deep learning model.

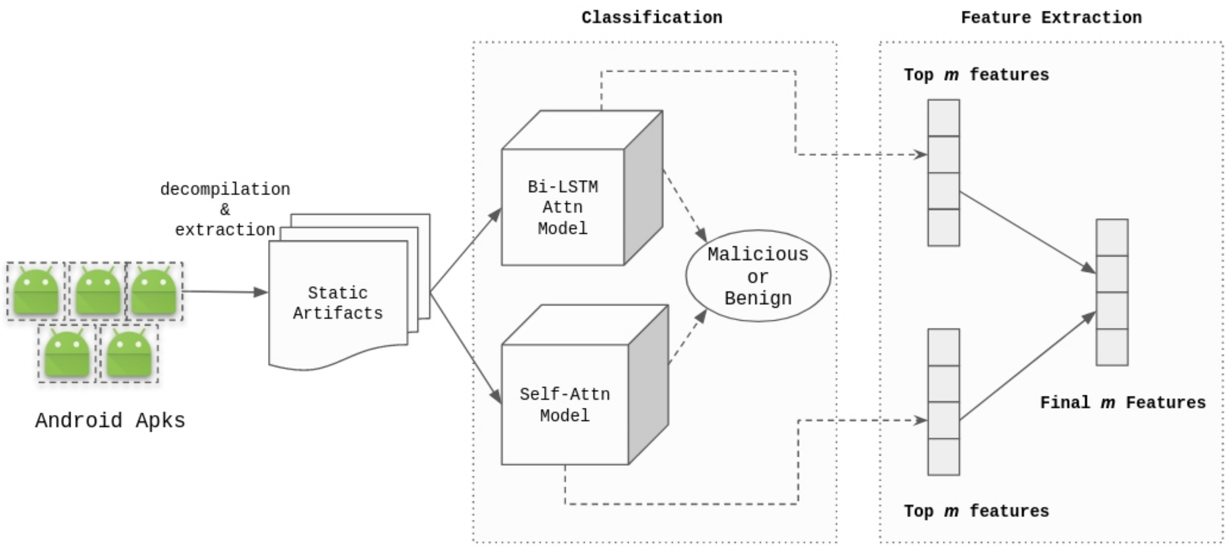


Fig. 1.1. System design of attention-based Android security vetting system

To summarize, we obtain API-call as our artifacts by decompiling the android apks and extracting the API-calls in the form of strings from the smali files. The number of API-calls in a single apk can be long, so we fix our input length to 4000 and pad when necessary. Then we get embeddings of our input through $word2vec$ [10]. The embedding matrix obtained is used as our final input to our two models: Bi-LSTM Attention and Self-Attention. Both these models are built as a classifier. After training, we then extract the attention weights for the malicious apps which were correctly predicted by our model. The ranking of attention weights gives us the dominant malicious features that our models relied upon when making the decision of classifying an app as malicious. A simplified system design of our work is shown in Fig. 1.1.

The major findings of our work are: **(a)** We built two attention-based classification models with an auPRC (area under precision-recall curve) score of 0.9391 with Bi-LSTM Attention and 0.9074 with Self-Attention model. **(b)** Using the attention weights from these models, we were able to select top 200 API-calls for each model. On further analyzing the API-calls, we found that 21 API-calls were common between the top 200 API-calls from two models. Thus, we were able to collect a total of top 200 API-calls that can potentially reflect maliciousness of android apps which can be further studied by the research community to potentially discover new malware signature.

CHAPTER 2 BACKGROUND

In this chapter we provide the reader with necessary background information on the key topics of our thesis. First we discuss on the Android ecosystem and static analysis followed by the concepts of deep learning with attention based models.

2.1 The Android Ecosystem

In modern era, mobile applications have a big impact on our personal and professional life. We can do online banking, grocery shopping, attend a conference call, buy or trade a stock and many other things with the app installed on our phones. The android ecosystem is evolving in such platforms in a great pace. After their first stable release in 2008, and as of February 2021, Google has released Android 12. In each version, the inner libraries and methods are updated. This evolving process has only made Android more popular. Currently Google Play Store hosts more than 3 million apps. With a developer fee of $25 we can simply upload an android app ourselves. Even with certain security measures employed by freely available stores, some malicious apps find a way to sneak into the market.

An android app is simply a compiled files of java source code. With freely available decompilation tools, we can dig further into its core parts. The modules in an APK are compiled into Dalvik Executable (DEX) form when they are built. The DEX bytecode is then used by the Android Runtime(ART) which maintains an application runtime environment used by the Android operating system. This runs on Dalvik Virtual Machine (DVM). During compilation, the core application and all the library modules are built and combined to a single DEX file.

To get some meaning from the DEX file, we have to further decompile them. Tools like Androguard [11], Amandroid [12], Soot [13] and apktool [14] can convert the Dalvik bytecode into an intermediary language. In our work, we work with *apktool* which converts the underlying dex file obtained from an apk to smali files. We also get other files like Manifest file which gives information about the component and the permission used by the app. Dex files are not quite human redable, so apktool can convert them to smali files which are somewhat readable to humans. Fig. 2.1 shows a process of compilation of app(left) and a decompilation(right) process through apktool. This process aligns to the android app written on JAVA source code only. In our work we follow the similar process and analyze the API-calls available from the smali files.

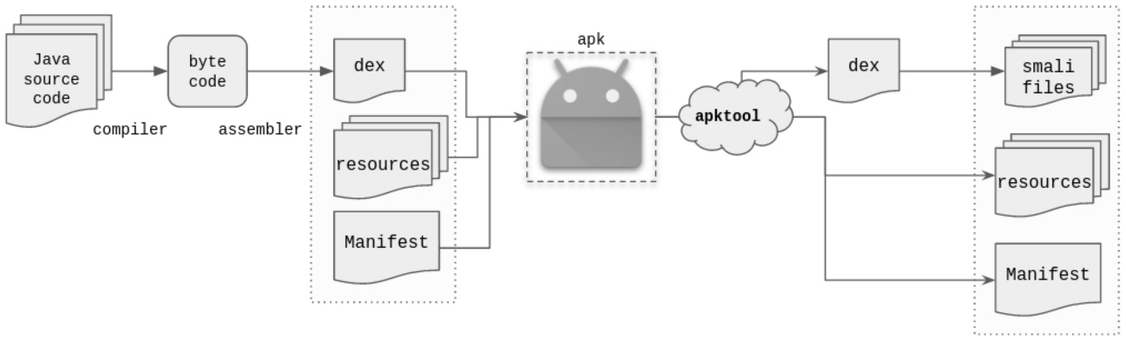


Fig. 2.1. Building and decompiling an Android apk

2.1.1 Analysis Techniques

Security vetting of android applications can be classified into two groups: static analysis and dynamic analysis. Static analysis analyzes an app without executing it and by analyzing the code. Alternatively, dynamic analysis analyzes the app during its execution or after execution using runtime information such as behavior or actions.

2.1.1 Static Analysis

In static analysis, we analyze source code information from an android app such as bytecode, manifest file, resource files, etc. This is done without executing the app whether in the phone or in a virtual environment. We have to solely rely on the static information from the decompiled files. With static analysis, we can uncover some information such as control flow and data flow within an app. This can be also visualized as a path from source to sink and understanding traversal of control flow and data flow, etc. However, acquiring such information can be computationally expensive in both memory and time constraints.

Avoiding such computationally expensive operations, we keep our focus on API-calls. They can be obtained by decompiling apks and scanning the smali files. In our experiment, we use the sequence of such API-calls as our artifact. Formally, we define API-call as *any call statement that is invoked in the app source code but is not a user-defined function*. Here, the function does not belong to a java class of the app.

One major disadvantage of static analysis is that we are unable to track the dynamic behavior or action of the android app. It also suffers from false alarms. Attackers can utilize evading features like code obfuscation, dynamic code loading, reflection, etc. Flowdroid, Amandroid, Droidsafe, etc., are some popular static analysis tools which check for malicious activities such as data leakage, data injection, API misconfiguration, etc. To detect such features, static analysis frameworks have to manually work on them. Also, such frameworks have a difficult time in dealing with zero-day malware. Some signature based static analysis approaches [15] [12] were popular in the past. Currently machine learning and deep learning approaches are being popular among the research community.

Using the knowledge of the android ecosystem, Roy et al. extracted 471 manually selected features from categories such as native code signature, obfuscation signatures, permissions, intent actions, etc. This was extracted from $500,000$ static features that were collected from source code and manifest files from Drebin [16]. Building such features manually takes a lot of human effort. With the advent of deep learning approaches, automatic representation learning is possible. The deep learning models are scalable and are more resilient to zero-day malware. In our work, we build attention models to extract the important malicious API-calls that can benefit the research community.

2.1.1 Dynamic Analysis

In dynamic analysis we execute the app in an emulator or a similar environment and track the behavior of the app. There are several ways to track the app behavior, it could be analyzing the kernel functions invoked during app execution, or manually observing the app behavior. Manually observing each execution path of every app is very time consuming. One disadvantage of dynamic analysis is that some malwares realize the use of emulators under scrutiny and thus hide their malicious action. Without a proper event execution strategy we may miss some code execution paths and miss malicious patterns. Popular event execution tool such as Monkey[17] is based on random event triggers. Apichecker [18] is an API-centric dynamic analysis approach which extracted 426 key APIs using ML-algorithms. We leave the use of dynamic analysis using attention model as future work.

2.2 Word Embedding

Machine learning requires data to be in numbers. Different forms of inputs such as images, text, audio, etc needs to be converted to some suitable numeric form. These need to be converted to feature vectors. Mikolov *et al.* [19] proposed word2vec by embedding words to vector space using Continuous Bag of Words (CBAG) and Skip-Gram (SG) algorithms. In SG, model scans over the words of each sentence and tries to use the current word in order to predict its neighbors whereas in CBAG the model employs each of the neighbour's contexts to predict the current word. Word embedding from word2vec also, based on the usage of words, tries to capture the semantic, contextual and syntactic meaning of each word in the corpus vocabulary. It also showed that words with similar semantic and contextual meaning have similar vector representations and semantically different words have a unique set of vector representation. Simply put, it accepts text corpus as an input and outputs a vector representation for each word. Fig. 2.2 shows a simple process of word embedding.

In our implementation, we need to convert the API-calls to embedded vectors before feeding to our models. So we use gensim [20] library's implementation of word2vec to convert our vocabulary from API-calls to an embedding matrix that we feed as input to our models.

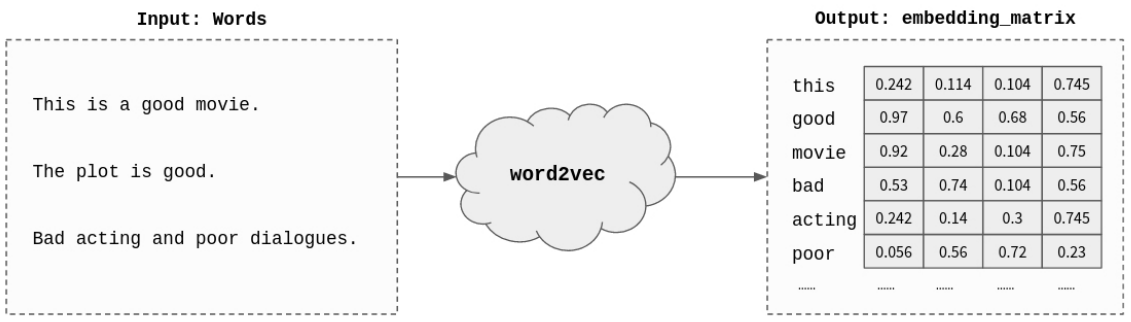

Input: Words
This is a good movie.
The plot is good.
Bad acting and poor dialogues.
word2vec
Output: embedding_matrix
this 0.242 0.114 0.104 0.745
good 0.97 0.6 0.68 0.56
movie 0.92 0.28 0.104 0.75
bad 0.53 0.74 0.104 0.56
acting 0.242 0.14 0.3 0.745
poor 0.056 0.56 0.72 0.23

Fig. 2.2. Converting word to embedded vectors: word2vec

2.3 Deep Learning

Inspired from the working of a biological neuron, Artificial Neural Networks (ANN) were originally developed as mathematical models. Every neuron in an ANN consists of a variable value and a bias parameter. The connections between the layers of neurons are denoted with weight parameters. An activation function determines the output of the neurons given the weights and the biases. This activation function is a nonlinear function, such as Sigmoid, Softmax, Tanh, Rectified Linear Unit (ReLu) etc. The goal of training a neural network is to optimize and update the weights by using a backpropagation algorithm.

The neurons in ANN are ordered in layers. When multiple such layers are stacked, it is called a deep neural network. The application of such networks in different forms can be referred to as Deep Learning. The availability of a myriad of data in the present world has attracted the research community in the usage of deep learning in different forms. Data involving high-dimensional and unstructured features are of great interest in deep learning.

One of the most prominent properties of deep learning is feature engineering. It involves transforming the raw data into features that helps to increase the predictive power of the model. Feature engineering the android apks can help build a robust predictive model for malware detection and thereby extract the most important features.

2.3.1 RNN

RNNs are a special instance of deep neural networks that have internal memory or state. In an RNN, as shown in the Fig. 2.3, each unfolded RNN represents a layer of the neural network. Because there are multiple layers stacked upon each other, a RNN is a deep neural network. RNNs have the ability to process variable-length inputs and generate a feature vector, also called a context vector. This vector has the compressed information about the whole input. RNNs learn by adjusting the inner weights of each RNN cell through backpropagation with time. RNN uses gradient descent based learning and in each backpropagation the chain rule affects the effective gradient. This leads to exploding or vanishing gradients [5][6].

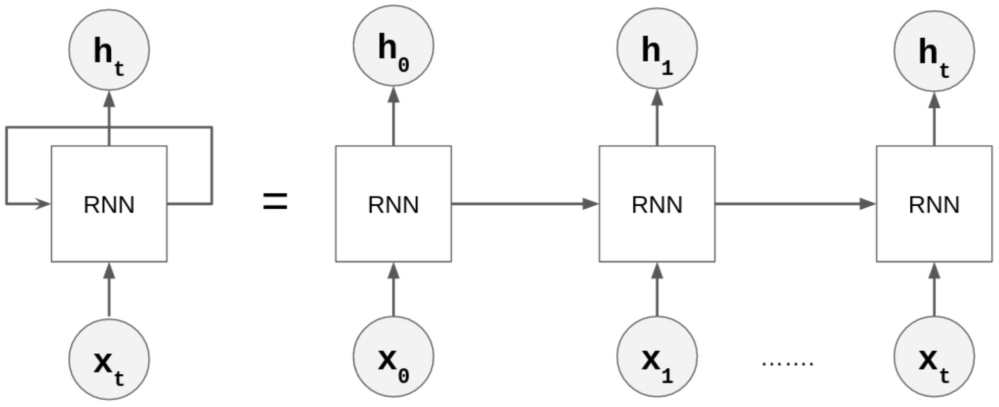


Fig. 2.3. Example of an RNN cell

2.3.2 LSTM

LSTM cells [21] are special RNNs designed to solve the vanishing gradient problem of traditional RNNs. 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 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. Fig. 2.4 gives an overview of working of a simple LSTM cell.

To describe the working mechanism of an LSTM cell, we denote C_t as the cell state at time t.

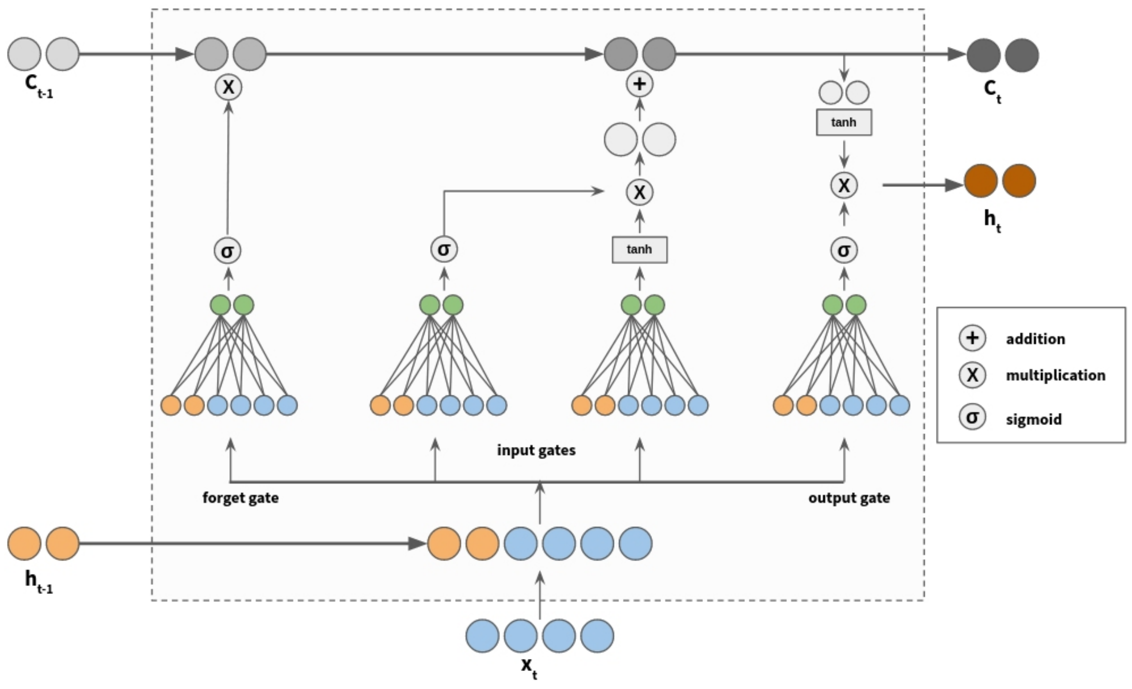


Fig. 2.4. LSTM Cell

h_t denotes the hidden state at time t. x_t is the input sequence at time step t which is concatenated with the hidden state vector h_{t-1}. W_i and W_f indicates the weights at input and forget gates respectively. We start computing the forget and input gate vectors as follows:

$$\begin{aligned} f_t &= \sigma(W_f * [h_{t-1}, x_t] + b_f) \\ i_{\mathrm{t}} &= \sigma(W_{\mathrm{i}} * [h_{\mathrm{t\text{-}1}}, x_{\mathrm{t}}] + b_{\mathrm{i}}) \end{aligned} \tag{2.3.1}$$

The forget gate determines how much information to forget from previous cell's state based on sigmoid activation on previous cell's hidden state h_{t-1} and input x_t to the current cell at time t. W_f represents the weights at the forget gate and b_f represents the forget bias. Cell state is then updated with pointwise multiplication of of f_t and C_{t-1} as in equation 2.3.2.

$$C_t = C_{t-1} * f_t \tag{2.3.2}$$

Now, we compute a vector of new candidate values that will be updated to the cell state. Then, the multiplication of i_t and $\tilde{C_t}$ will determine what values are to be added to the cell state. Pointwise addition is done to update the cell state C_t as in Equation 2.3.3 where W_c represents the weights at the updating input gate and b_c represents the bias.

$$\begin{aligned} \tilde{C_t} &= tanh(W_c * [h_{t-1}, x_t] + b_c) \\ C_t &= i_t * \tilde{C_t} + C_t \end{aligned} \tag{2.3.3}$$

As in Equation 2.3.4 the output gate decides what parts of the cell state will be at the output. The sigmoid layer activation determines the relevant parts to output. This undergoes pointwise multiplication with the updated cell state, to give the hidden state output represented by dark orange circles.

$$\begin{aligned} o_t &= \sigma(W_o * [h_{t-1}, x_t] + b_o) \\ h_t &= tanh(C_t) * o_t \end{aligned} \tag{2.3.4}$$

One of the disadvantages of LSTM is it's capability of dealing with variable sized sequences. The maximum length of the input has to be known in advance and we have to truncate or pad the input sequence to a finite number. Additionally, LSTMs is only capable of leaning information from the past. To make use of the future information Bi-LSTMs were introduced which is described in the next section.

2.3.3 Bi-LSTM

Bi-LSTM, also referred to as Bidirectional LSTM, uses two LSTMs in forward and backward manner to capture more information from the input sequence [22] [23]. In traditional LSTM, information flows in a forward direction with respect to time. Similarly for API calls, looking at both directions when processing sequences could help the model make an inference using inherent property of function calls. Hence, the whole context of the input sequence can be taken into account. As shown in Fig. 2.5, two LSTM cells process the sequence in forward and backward direction and the output are concatenated at the end.

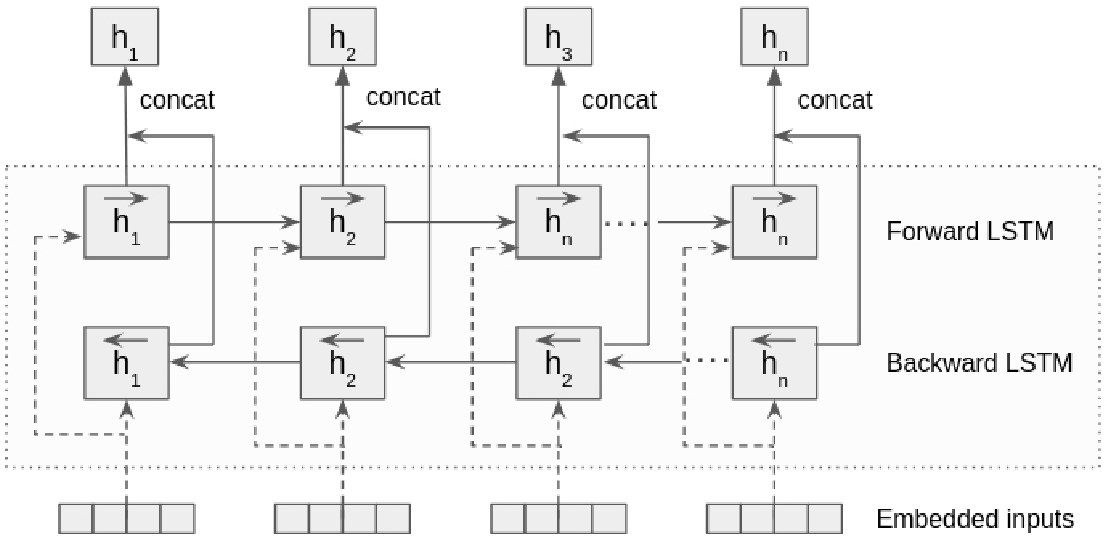


Fig. 2.5. Bi-LSTM

2.3.4 Attention

The use of RNN was popular in Neural Machine Translation (NMT) [24] [25]. It consists of an encoder-decoder architecture. The encoder processes the input sequence and encodes the information to it's hidden representation called *context vector*. Decoder gets this context vector and starts generating the learned output. A simple architecture is show in Fig. 2.6.

The intuition behind the encoder-decoder model with RNNs can be analogous to this following example: In a translation task from French to English, translator A starts reading the French text. Once finished reading, he then relies on his memory power to translate each word. There is a chance he may forget the early words in the text due to the length of the sequence.

One major drawback of this architecture is the ability to cope with long sentences. On processing the later parts of a long sequence, the early parts of the input was not being learned efficiently. Attention mechanism was then originated to address this issue with RRN based architectures. Attention mechanism involves an additional layer after the encoding process where all the outputs from each encoder output is preserved and is passed as an information to decoder at each time

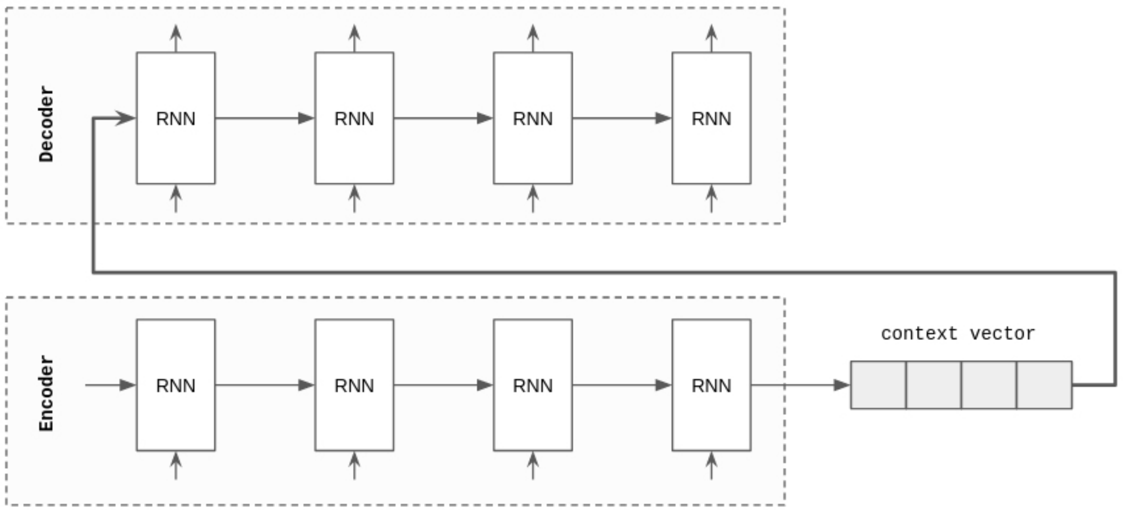


Fig. 2.6. Encoder-Decoder architecture

step.

The intuition behind the encoder-decoder model with attention can be analogous to this following example: In a translation task from French to English, translator A starts reading the French text. He then scans and makes a note of the keywords from the start to the end of the sequence. Now referring back to these keywords, he starts translating them to English.

2.3.4 Bahdanau's Attention

In this attention, the input sentences are mapped to annotations h_1, h_2, ... h_{Tx}, where T_x denotes the last word in the input sequence. Annotation h_i has a strong focus on the parts surrounding the i-th word of the input sequence.

The context vector c_i is then calculated as:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{2.3.5}$$

Here, the weight $\alpha_{ij} h_j$ for each h_{ij} is calculated as softmax function:

$$\alpha_{ij} = \frac{\exp\left(e_{ij}\right)}{\sum_{k=1}^{T_x} \exp\left(e_{ik}\right)} \tag{2.3.6}$$

where,

$$e_{ij} = a\left(s_{i-1}, h_j\right) \tag{2.3.7}$$

is a soft alignment model which scores how well the inputs around position j and the output at position i match. The score is based on the RNN hidden state s_{i-1} and the j-th annotation of the of the input sequence.The alignment model a is parametrized as a feed-forward neural network which is jointly trained with all the other variables in the network.

2.3.4 Luong's Attention

It is a simplified variant of Bahdanau's attention. The idea is to put additional information to the decoder's output for better output. The decoder outputs hidden states h_t for all input tokens to translate at each timestamp t. The score of this decoder output with all encoder outputs h_s can be calculated from one of the three forms:

$$\operatorname{score}\left(\boldsymbol{h}_t, \overline{\boldsymbol{h}}_s\right) = \begin{cases} \boldsymbol{h}_t^\top \overline{\boldsymbol{h}}_s & \text{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W}_{\boldsymbol{a}} \overline{\boldsymbol{h}}_s & \text{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W}_{\boldsymbol{a}}\left[\boldsymbol{h}_t; \overline{\boldsymbol{h}}_s\right]\right) & \text{concat} \end{cases} \tag{2.3.8}$$

We take the general approach in our experiment where the matrix $\boldsymbol{W}_{\boldsymbol{a}}$ represents a linear layer with the weights that are also learned during back-propagation.

Similar to 2.3.6 we then use Softmax to convert scores computed previously to the probability distribution.

$$align\ (\ score\ ) = \frac{\exp\left(score\left(h_t, \overline{h_s}\right)\right)}{\sum_{s'} \exp\left(score\left(h_t, \overline{h'_s}\right)\right)} \tag{2.3.9}$$

Finally with a dot product, we get the context vector.

$$c = align(score)^T \overline{h_s} \tag{2.3.10}$$

2.3.4 Bi-Attn-LSTM

Zhou *et al.* [23] proposed this attention mechanism using Bi-LSTM for relation classification tasks which was inspired from Luong's attention. Let H be a matrix consisting of output vectors $[h_1, h_2 \ldots, h_T]$ that the LSTM layer produced, where T is the sentence length. The representation r of the sentence is formed by a weighted sum of these output vectors. Here, α gives the attention weight of each sequence.

$$\begin{aligned} M &= \tanh(H) \\ \alpha &= \operatorname{softmax}\left(w^T M\right) \\ r &= H\alpha^T \\ h^* &= tanh(r) \end{aligned} \tag{2.3.11}$$

$H \in \mathbb{R}^{d^w * T}$, 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, T, d^w respectively. h^* is the final sentence-pair representation which can be further used for classification. Chaulagain *et al.* [4] used this attention mechanism for malware classification.

2.3.4 Transformer and Self-Attention

Vaswani *et al.* [9] proposed Transfomer, a novel approach of processing sequences without the use of recurrent cells. RNNs exhibit sequential nature which are incompatible with parallel computation. This can affect the robustness and efficiency with longer input sequence. The self-attention mechanism introduced is fully parallelizable. Transformer model is an extension of 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 efficiently.

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

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{2.3.12}$$

$$PE_{(\text{pos},2i+1)} = \cos\left(\frac{\text{pos}}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{2.3.13}$$

where pos denotes token's position, i denotes the embedding dimension
to be encoded, d_{model} is the embedding dimension. Hence, i belongs to the interval [0, d_{model}].

The encoder in transformer consists of multi-head attention block followed by linear transformation with 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 the improved form of generic attention mechanism which consists of queries Q, keys K and values V. These three vectors represent the same sequence of positionally encoded input sequence. As in other form of attention, the alignment here is computed on Q and K, which is then applied to V. This is analogous to search engines, where a user's $query$ is matched against engine's $keys$ and the $values$ are the results.

Transformer uses scaled dot-product attention as in Equation 2.3.14 to calculate the alignment score. It 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 more reasonable during back-propagation.

$$\text{Attention}\left(Q, K, V\right) = \operatorname{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{2.3.14}$$

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 the subspace of the initial dimension.

The output from the attention block is then fed to feed-forward network. There is a residual connection around each block. This residual connection helps keep track of data, 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.

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 $O(n^2.d)$ 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 is quite impossible to fit the model in a single GPU.

CHAPTER 3 RELATED WORKS

In this chapter, we review prior malware detection approaches. We also briefly discuss the relevant deep learning approaches that suits our experiments.

3.1 Static Analysis

FlowDroid [26] is a static taint analysis tool. It can efficiently handle the callback methods invoked by android framework revealing an overview of an app's life-cycle. It analyzed about 1000 malware samples. It, however, couldn't support Inter-component communications or handle reflection and exceptions.

Amandroid [12] constructs intra procedural CFG and intra procedural DFG as part of static analysis. Building CFG and DFG are computationally expensive operations. Additionally, it also enables plugins to detect specific app behavior like locking screens , deleting call logs, and so on.

The usage of API calls for static analysis became more popular as Android framework provides such rich APIs which can be useful to determine the malice of apps. Drebin [16] used over 500k static features from source code and Manifest file. ML algorithms were then used on these features. Out of those 500k features, Roy *et al.* [27] manually filtered out the top 471 features which gave similar classification results using standard ML algorithms. The selection were done among the categories such as native code signature, obfuscation signatures, permissions, intent actions, etc. With the evolving android ecosystem, a better feature selection approach is required instead of manual inspection. This can help to address issues when identifying the signatures from zero-day malware [28].

DROIDAPIMINER [29] performed ML based frequency analysis on 169 API calls to capture the most relevant API calls. Out of four ML classifiers, kNN achieved the best performance with 99% accuracy and 2.2% false positive rate. The authors of a similar tool, MaMaDroid [30], claim that it performs better than DroidApiMiner. MaMaDroid, abstracts the API calls performed by an app to their class, package, or family and builds a model from their sequences obtained from the call graph of an app as Markov chains.

ApiChecker [18] is an ML-powered malware detection system that utilizes ground-truth data from one of the world's largest Android app market (called Market-X). It is an API-centric dynamic analysis approach which extracted 426 key APIs in three categories: Restricted API, Highly-Correlation 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. It achieved an overall precision of 98% and recall of 96%.

3.2 Deep Learning

Xu *et al.* [31] performed multi-layer classification on $110,440$ apps using deep neural networks. First layer consisted of Multilayer Perceptron on XML files and uncertain apps not meeting the threshold were passed on to the second layer where features were constructed based on the simplified bytecode of the app. The authors reported 97.74% accuracy and FPR(False Positive Rate) of 2.54%.

MalDozer [32] utilizes raw sequences of API methods followed by DL-based analysis. The API calls from the apps are extracted by analyzing the assembly codes from the Android apps in the form of DEX files. The API methods are then populated into tokens and are vectorized to fixed length high dimensional vectors which are then fed into multi-layer neural networks involving CNNs. It introduced the static analysis concept of automatic representation learning. MalDozer achieved an F1-Score of 0.96 with false positive of 0.06 to 0.02 on the task of classifying android apps as malicious or benign.

Deep4MalDroid [33] extracts the system calls as dynamic artifacts while executing android apps under test. It involves classification using Stacked AutoEncoders (SAEs) where the feature set is a graph constructed using Linux kernel system calls. It used 1500 malicious and 1500 benign apps from Comodo Cloud Security Center.

Chaulagain *et al.* [4] performed hybrid analysis using variants of LSTM models. Amandroid [12] was used to collect API calls as static artifacts. System calls were collected as dynamic artifacts using Genymotion [34]. The hybrid approach involves taking probabilistic average of static and dynamic models. The best deep learning model with hybrid analysis achieves an area under the precision-recall curve (AUC) of 0.9998. The authors also claim that imbalanced ratio of malicious and benign apps and the quality of the malware can affect the classification. They compared the static analysis to that of Maldozer [32] and obtained a better F1 score of up-to 0.993 under the same set of apps provided by the authors of Maldozer.

3.3 Attention

Attention mechanism was inspired from neural machine translation (NMT). Sutskever *et al.* [24] proposed an NMT architecture of Encoder-Decoder using LSTMs. It 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.* [35].

Bahdanau *et al.* [7] addressed this issue using the attention mechanism which acts as an intermediate layer between encoder and decoder. This allows the decoder to selectively retrieve the information that is relevant to the contents of the input sequence. Here, alignment score is based on concatenation. It is also known as additive attention as it performs a linear combination of encoder states and the decoder states. Rush *et al.* [36] proposed local attention based model for sentence summarization. Local attention here refers to attending only a subset of words. It learns a latent soft alignment over the input text to summarize the input.

Luong *et al.* [8] improved Bahdanau's attention mechanism with a different method of alignment score calculation. It brought the concepts of local and global attention and introduced 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 a part of source hidden states.

Vaswani *et al.* [9] proposed, Transformer, a novel approach of processing sequences without the use of RNNs. Their model consisted of a multi-head self-attention mechanism and 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 coined 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 relation can be captured by Transformer. Transformer has been the building block for state-of-the-art language representation models.

A lot of research has been done on classifying the malware based on API calls, but the roles of those API calls are still unknown. This is where we can use the attention mechanism to find the most relevant API calls that are responsible for the malice of apps. The nature of API calls are quasi-sequential [4]. Using the API calls as our static artifacts, we start with the combination of Bi-LSTM and Luong's [8] attention, followed by self-attention models.

CHAPTER 4 APPROACH

4.1 APK Collection and Ground Truth

In this work, we use Androzoo [37] as our primary source of android apks. We have a local repository setup in BGSU lab server, where we have downloaded the android apks made available by Androzoo community. These apks are distributed across multiple storage devices. Androzoo provides us with a huge csv file that has the metadata of the android apks that it makes available. The metadata includes information such as $sha256$, $sha1$, $md5$, dex_date, apk_size, pkg_name, $vercode$, $vt\_detection$, vt_scan_date, dex_size and $markets$. We use python script to query the path of the stored apks. We then use a simple script as in Fig. 5.3 to do a bulk push to S3 bucket from a txt file with list of paths to the apk.

We obtain both benign and malicious apps from Androzoo based on the VT scores. $vt\_detection$ gives us the relevant scores to determine the quality of malware. We have a total of 43,000 malware which are divided into two sections based on their VT scores. Apps having $vt\_detection$ between 2 and 10 are considered as low quality malware. Similarly, the apps with $vt\_detection$ greater than 10 are considered high quality malware. Benign apps have zero $vt\_detection$ score.

4.2 Collection of Artifacts

Android apps are mostly based on the support from from $Android$ and $java$ classes. With the popularity and an open community, several third-party libraries have extended their support to Android ecosystem. We collected a total of 132 popular third party libraries that could be utilized by attackers when loading apps with malicious behaviours. We then used a python script using apktool to decompile an app and obtain the API-calls from the decompiled smali files. User defined functions are then collected from the code by scanning the smali files. This gives a quasi-sequential order of API-calls. Each API-call obtained is then delimited by a space and stored in a txt file for every apks.

```
i=1
set -e
while read line
do
  aws s3 cp $line s3://prabesh-pathak/2016/mal/
  echo "Apks copied $i"
  i=$((i + 1))
done < /media/elements/attn/2016/2016mal.txt
```

Fig. 4.1. Copy apks to AWS S3

4.3 Deep Learning

Our experiments involves the use of deep learning in two phases. First, we will test our two attention models on two freely available datasets. Once we verify that our models are performing well on these datasets, we will extend our models to android security vetting.

4.3.1 External Datasets

To further describe the intrepretability of attention models, we will train our models on two external datasets: Amazon Reviews Dataset and MNIST. The Amazon Review Dataset is freely available dataset that has hundreds of thousands of user reviews. This has been widely used in the field of natural language processing for sentiment analysis. The motivation behind using the Amazon Review Dataset is that it is based on English vocabulary. After testing our models on this dataset we aim to find the words that reflect the appropriate sentiment of the sentence in the evaluation. Here, we load the reviews directly into memory, tokenize and pre-process them so they are suitable for deep learning.

Similarly, MNIST is another widely used image dataset of handwritten digits. It has thousands of grey-scale images of handwritten images from 0 to 9. The dimension of the image that we use is 28 x 28 pixels. While using this data, we make use of each pixel and it's corresponding value. The motivation behind using this dataset is to observe how our models attend to important features of images , as opposed to texts.

To run our experiment on this dataset, we will use the computation resource from Google Colaboratory. We will upload the datasets directly to memory. It is to be noted that, if we load the files in streaming manner as opposed to loading the dataset directly to memory, the computational power decreases in Google Colaboratory.

4.3.2 Android Apps

As a part of data engineering, the API-calls for each app needs to be pre-processed. This is done to remove any extra characters or spaces. The artifacts then will be fed to an $word2vec$ model to get the embeddings for each API-call. After obtaining the embedding vector, we will use this as a non-trainable embedding layer in our models.

We build our two models separately. We study the training and validation losses to make sure that our models are learning properly. This will also involve a rigorous hyperparameter tuning will will be crucial in our experiment. We also aim to do some side experiments by varying the malicious to benign ratio and the quality of malware to get more insights of our models.

PyTorch [38] is used as the primary deep learning framework in this experiment. It allows a lot of flexibility to change or update our models and parameters. We have individual text files as artifacts for each app. We will be feeding the inputs in streams as opposed to loading everything on memory. This will address the scalability of our experiment if we want to test or train the model on bigger set of apps in the future.

For android apps, we train and evaluate models in AWS(Amazon Web Services). We will be using an external volume where we copy all our artifacts and code.

4.4 Discriminative Feature Selection

We aim to extract a combined set of top features from our two proposed models. In doing so, we analyze the attention weights of our model during inference. For each model, we collect the top m features from the True Positive malware in our test set. An API-call, can appear multiple times in a single application. Likewise, the same API-call can have different attention weights for the same app based on its position in the sequence. To accommodate this, we take cumulative weight of the API-calls. This can also help us deal with any possible outliers. This gives us an overall weight across all the true positive malware in our test set. We will store each unique API-call in a hash map and then sort them to get the top m features.

Top m features obtained from each model can later be analyzed individually. After obtaining top m features from each of our models, we also check for any overlapping features. We sort the top features by their weights to get our final top features representing both of our models.

CHAPTER 5 IMPLEMENTATION

In this chapter, we explain all the methods involved in this work. We start with our AWS workflow, artifacts collection and deep learning models.

5.1 AWS Setup

The overview of our AWS setup is as shown in Fig. 5.1.

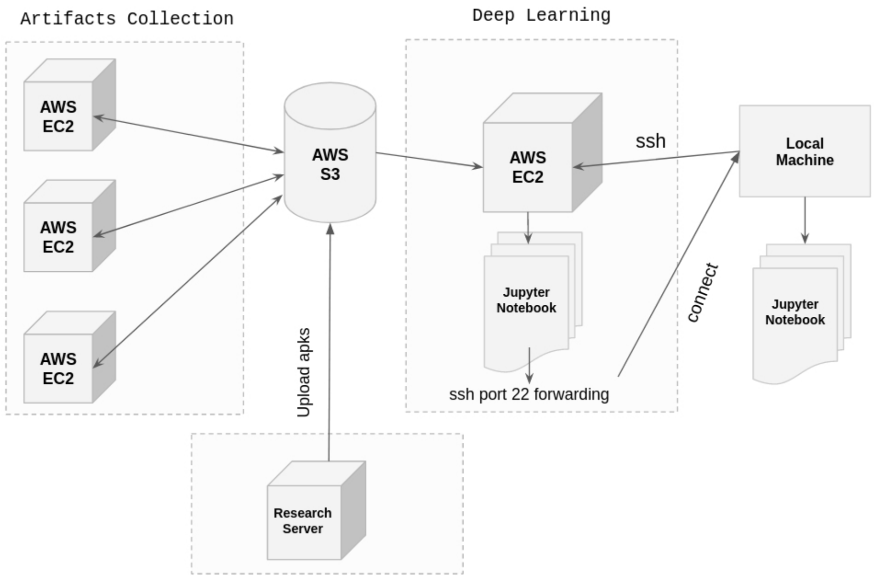


Fig. 5.1. AWS setup

5.1.1 Artifacts Collection Phase

In artifacts collection phase, we use 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 with large number of cores. So, we used memory optimized instance type $r5.8xlarge$. It comes with 32 cores and 256GiB of memory. As of $02/25/2021$, it's on-demand pricing is $\$2.016$ per hour. We also require external EBS to copy the apks from S3. We use ssh to connect to the EC2 instance from our local machine which needs to use the private key file associated with this key pair. This is also called $.pem$ file.

5.1.2 Deep Learning Phase

For our deep learning phase, we use Amazon's EC2 instance with Deep Learning Ubuntu Version 40.0 as our AMI(Amazon Machine Image). This come with pre-installed data science libraries and frameworks which saves our time. To facilitate deep learning, we need to use instance with high computational power. So we choose, $p3.2xlarge$ as our instance which come 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. On-demand pricing of $p3.2xlarge$ is $\$3.06$ per hour. So we opt to spot-instance which is relatively cheaper and costs $\$0.918$ per hour. Spot-instances are less reliable and volatile as compared to on-demand instances. As we save model checkpoints this would prevent us from losing our data and model state. 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 our local machine.

5.2 Dataset

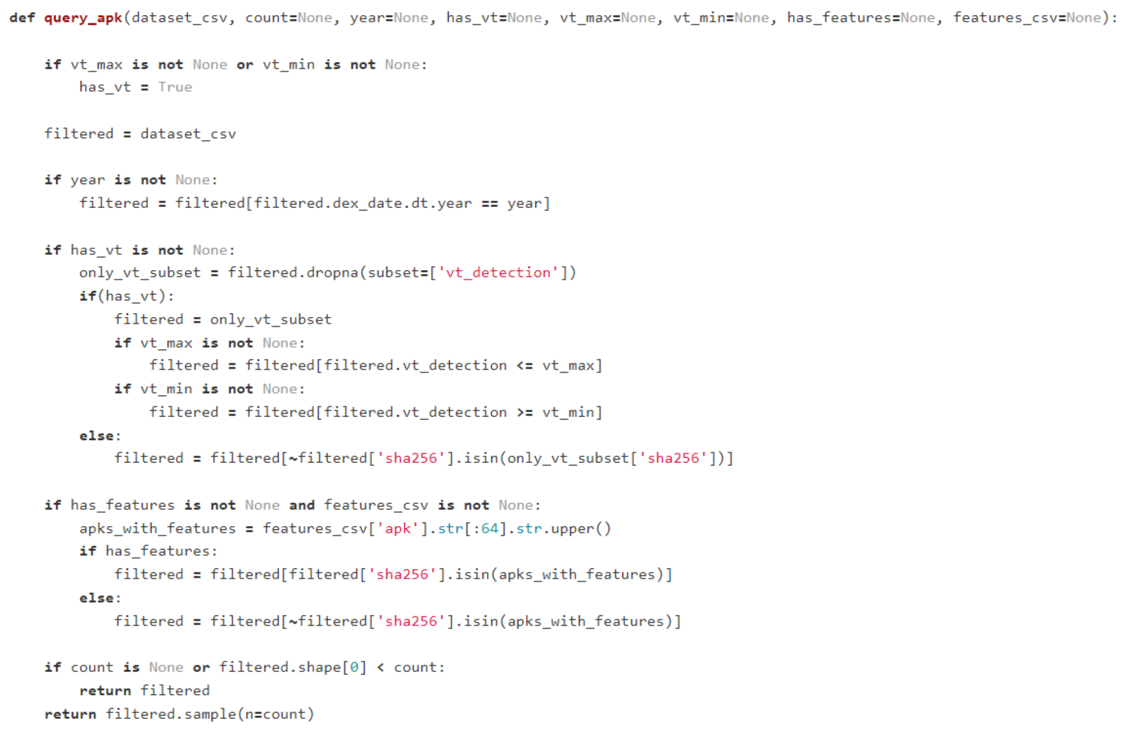
Our primary source of data are android apps. We extract potential artifacts from the apk file that are suitable for deep learning. We explain our data gathering and artifacts extraction methods in the following subsections.

5.2.1 Apk Collection

The number of Android applications are growing on a daily basis. The most popular app store, Google Play itself hosts about 3 million apps. But as we are also concerned with malicious apps , our primary source of android apps is Androzoo [37]. As of 02/25/2021 they have more than 14 million apks collected from various app markets including Google Play.

To facilitate future research opportunities in android security vetting, our research team here at BGSU has also started setting up an android apks repository. We have been downloading the apks made available from Androzoo on a regular basis. We have collected huge number of apks distributed across high end storage devices. Androzoo provides us the metadata of every apks in a huge csv file which includes information such as: $sha256$, $sha1$, $md5$, dex_date, apk_size, pkg_name, $vercode$, $vt\_detection$, vt_scan_date, dex_size and $markets$. We can use either of sha256, sha1 or md5 as an unique ID for the app. We proceed with sha256 as our unique ID. Here, the $vt\_detection$ field gives the number of anti-virus engines that have marked that app as malware. We make use of this metadata to retrieve the apks for our experiment. In this setup, we execute a python script to query the apk path.

Fig. 5.2 is a code fragment that gives us the count of the apks matching the query. This is used to get the path for apks. Fig. 5.3 shows an example of script execution. Here $LOCAL_AZ_DATASET is the metadata csv and $APK_PATH is the list of different storage devices where apks are distributed. The script returns $30,000$ apks of year 2016 having maximum $vt\_detection$ score of zero.



```python
def query_apk(dataset_csv, count=None, year=None, has_vt=None, vt_max=None, vt_min=None, has_features=None, features_csv=None):

    if vt_max is not None or vt_min is not None:
        has_vt = True

    filtered = dataset_csv

    if year is not None:
        filtered = filtered[filtered.dex_date.dt.year == year]

    if has_vt is not None:
        only_vt_subset = filtered.dropna(subset=['vt_detection'])
        if(has_vt):
            filtered = only_vt_subset
            if vt_max is not None:
                filtered = filtered[filtered.vt_detection <= vt_max]
            if vt_min is not None:
                filtered = filtered[filtered.vt_detection >= vt_min]
        else:
            filtered = filtered[~filtered['sha256'].isin(only_vt_subset['sha256'])]

    if has_features is not None and features_csv is not None:
        apks_with_features = features_csv['apk'].str[:64].str.upper()
        if has_features:
            filtered = filtered[filtered['sha256'].isin(apks_with_features)]
        else:
            filtered = filtered[~filtered['sha256'].isin(apks_with_features)]

    if count is None or filtered.shape[0] < count:
        return filtered
    return filtered.sample(n=count)
```

Fig. 5.2. Python function to return apk count

We collect android apks from the last 5 years (2016-2020). Based on our script, we select benign apps with max $vt\_detection$ of 0. Our query to retrieve malicious apps have a minimum

```
python query_apks.py $LOCAL_AZ_DATASET $APK_PATH --count 30000 \
--year 2016 --vt_max 0 > 2016ben.txt
```

Fig. 5.3. Python script execution query

Table 5.1: 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**

$vt\_detection$ score of 2. This is to include both high and low-quality malware to resemble a real-word data. We collect $40,000$ apks each from years 2016 to 2019. We only collect $12,000$ apks from 2020 due to less availability of malicious apks. On collecting apks, we kept the malicious to benign ratio of $1:3$. In real world, this ratio is much lower. The dataset is shown in Table 5.1

To reflect the ground truth of the malware from our dataset, we divide the malware into two sets: high quality and low quality malware. High quality malware are those apps having minimum $vt\_detection$ score of 10. Low quality malwares have $vt\_detection$ score between 2 and 10. The malware numbers are shown in Table 5.2. It is to be noted that if we include large number of high quality malware, our model might get biased towards it and could fail to generalize the low quality malwares during inference. We leave the study of such bias as future work.

Table 5.2: 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**

5.2.2 Artifacts Extraction

After obtaining the apks, the next process is to extract the artifacts that are suitable for deep learning. A high level extraction process is shown in Fig. 5.4. We decompile our apks with the popular tool $apktool$ [14]. On successful decompilation, we get files such as AndroidManifest.xml, resources, smali files, etc. We put our focus on the smali files which is our source for API-calls.

Here, we analyze the human readable decompiled code in the form of smali files to obtain the sequence of API-calls. We scan smali files with $.method$, $invoke-$ and $.endmethod$ keywords to get API-calls within a user-defined function. We separate each API-call by space and each function with # keyword. Our API-calls are not in a strict sequence. We hereby define that the API-calls are in $quasi-sequential$ order. In simple terms, our artifacts are collected by scanning and extracting the API-calls from the smali files obtained after decompiling the apk.

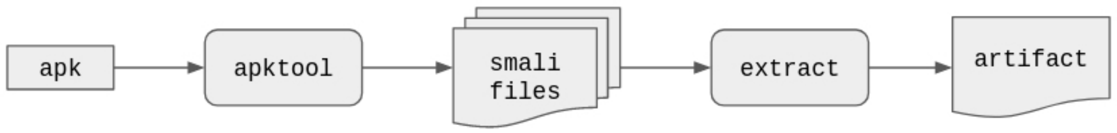


Fig. 5.4. Artifact extraction process

From our overall dataset of $172,000$ artifacts, we present the reader with the distribution of the source of API-calls. The vocabulary length of the corpus is $858,193$. Fig. 5.5 shows that $Android$ and $Java$ classes comprise of the majority of our API calls in our vocabulary.

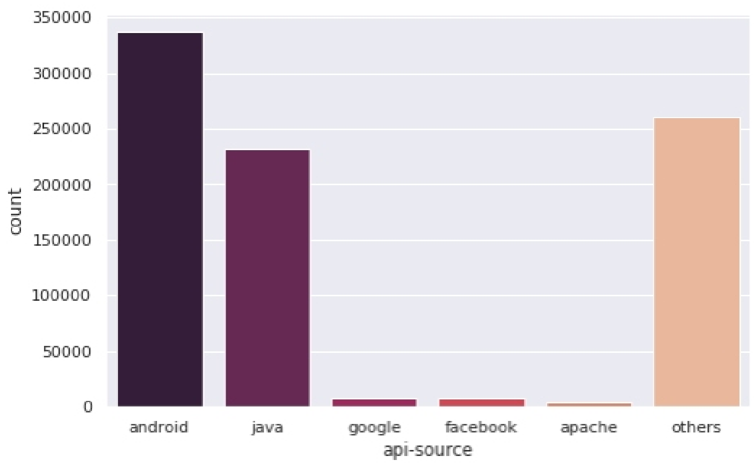


Fig. 5.5. API-Source count

5.3 Deep Learning

We build our deep learning models using two approaches: Bi-LSTM Attention and Self-Attention. These models are hereinafter referred to as $Algorithm1$ and $Algorithm2$ respectively.

$Algorithm1$ involves two LSTM cells moving in forward and backward direction and capturing information from the input sequences. Afterwards, the output are then sent to an attention layer from where we extract the attention weights. This output is then fed to a feed-forward neural network for classification through a Softmax function. Fig. 5.6 shows an overview of our Bi-LSTM attention model.

Fig. 5.7 gives a detailed overview of $Algorithm1$ with matrix dimensions. The calculation follows Equation 2.3.11. Bi-LSTM gives an output in form of [$batch\_size$ x seq_len x $hidden\_layer$]. This is fed to attention module. The output from attention modules gives our feature representation in terms of attention weights. This is amplified again with multiplication with input which is then fed to classifier network.

In $Algorithm2$, we first concatenate the $word2vec$ embedding with positional embedding module. We use positional embedding to preserve the order of the words in a possible sequence. The concatenated embedding is then fed to an encoder block of multi-head self-attention block and layer normalization. The output is then passed to a classifier network. Fig. 5.8 shows us an

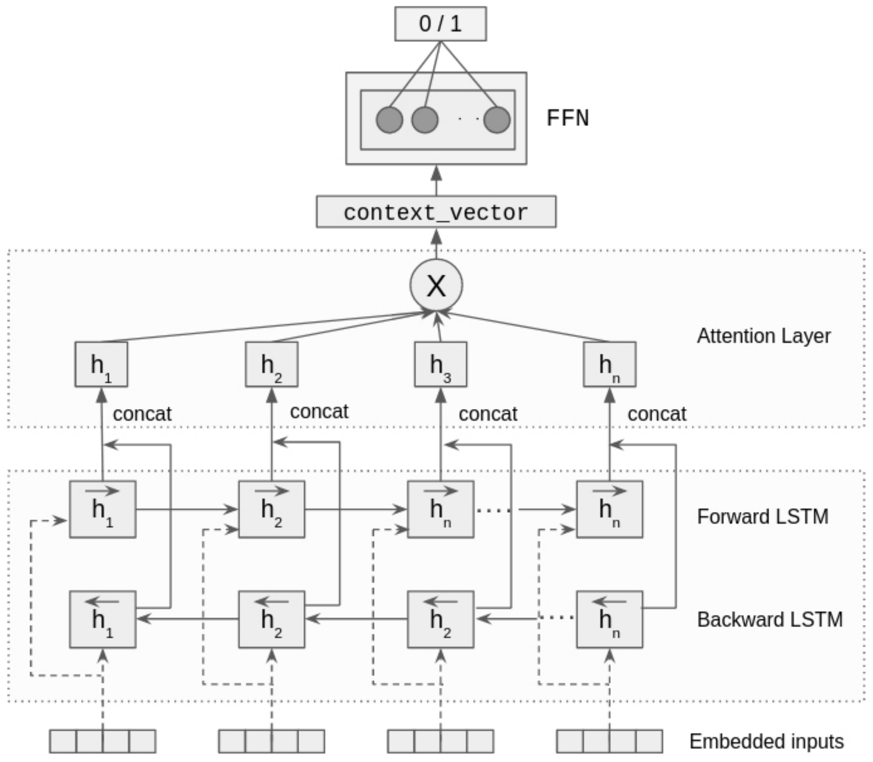


Fig. 5.6. Bi-LSTM attention architecture

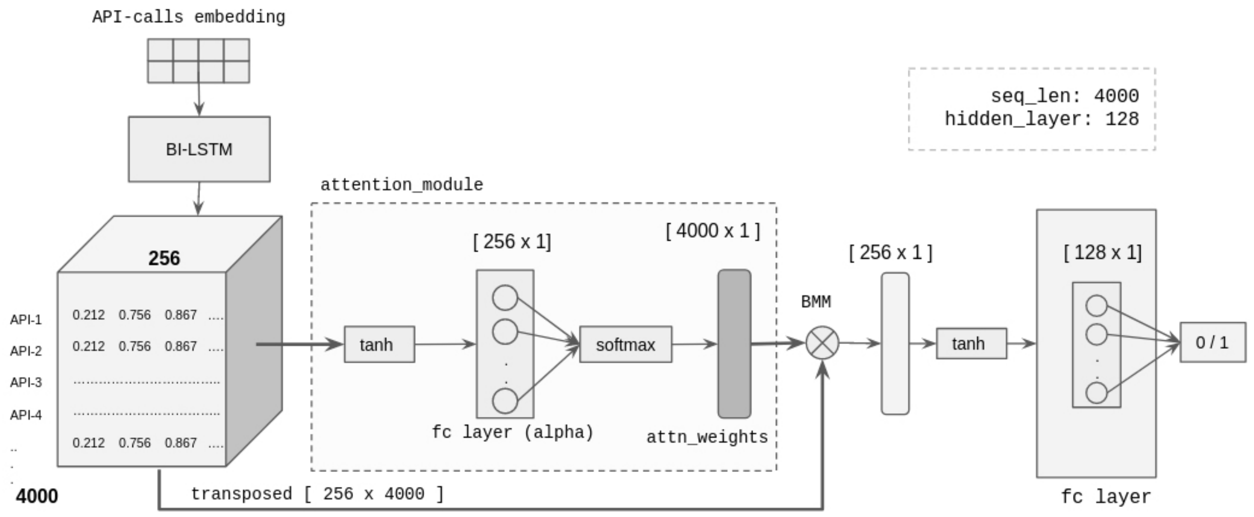


Fig. 5.7. Detailed Bi-LSTM attention

architecture of self-attention model.

The hyperparameters in our experiments were finalized based on the training and validation loss observed through the epochs. Other parameters not mentioned are set as default parameters provided by the PyTorch library.

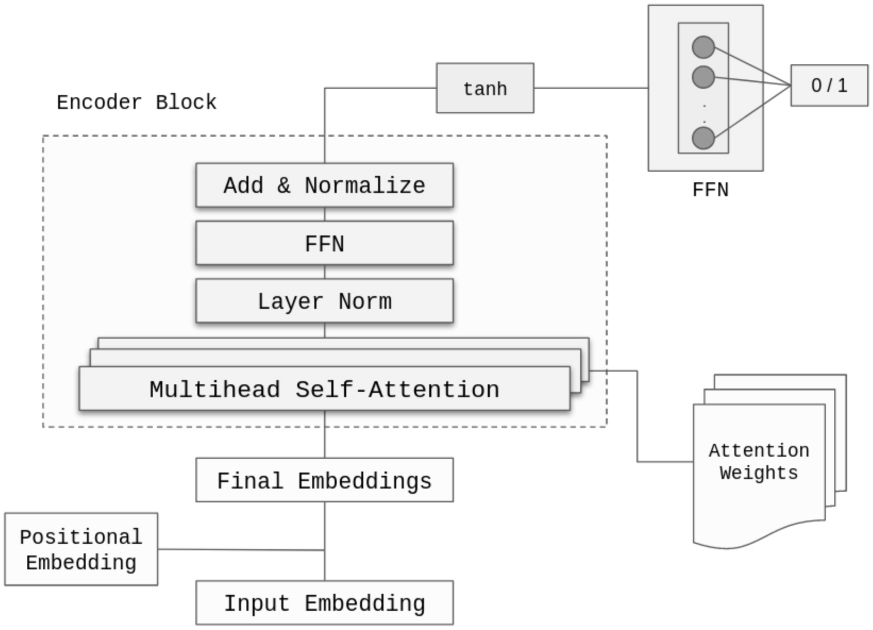


Fig. 5.8. Self-Attention architecture

5.3.1 Preparational Work on External Datasets

To make sure that our models are working properly, we test out modes on two freely available datasets: Amazon Reviews and MNIST. We stress the importance of the preparational work because it helps verify that our models can attend to the features in different domains of data.

5.3.1 Amazon Review Dataset

In this experiment, we first load all the reviews into memory. They are then labelled as 0 and 1 for positive and negative reviews respectively. Now, we do simple data cleaning which involves the removal of urls, numbers and punctuations. After data cleaning, an example review looks as: *"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 "* .

We then tokenize the sentences, where we split a sentence into individual tokens. The tokenized words are then stored in a dictionary mapping of unique words, building our vocabulary. We also use keywords like $\_PAD$ for padding and $\_UNK$ for unknown words. Here, we define our sequence length as 200. After the processing part, we separate training, test and validation sets. We use PyTorch's DataLoader class to load them into a data loader.

For $Algorithm1$, we set the batch size to 200. We performed multiple experiments and came with the following hyperparameters that worked the best: learning_rate of 0.0005 with a decay factor $5e-4$, embedding dimension of 100 and 128 LSTM hidden units . We use Adam Optimizer and train the model for 20 epochs.

Likewise, for $Algorithm2$ we use the same batch size of 200. The ideal hyperparameters were: learning_rate of 0.0005 with a decay factor $5e-4$, embedding dimension of 100, 4 attention heads and 128 units in the FFN(Feed Forward Network) of encoder block. We also used Adam Optimizer and trained the model for 20 epochs.

5.3.1 MNIST Dataset

As MNIST is a popular dataset, it's data loaders are freely available. We use PyTorch's $torchvision$ to get the MNIST data which is ready to fit in our models. We also do not require an extra embedding layer in this experiment. Here, our input length is 784 which is a flattened 28 x 28 pixels value. The inputs are loaded in a batch of 100. One interesting part of this experiment was that we were training our models on a completely new domain of non-sequential data.

For $Algorithm1$, our hyperparameters include: 128 LSTM hidden units, Adam Optimizer with learning rate of 0.01 with weight decay of $1e-5$ and dropout of 0.3. We train this model for 10 epochs. Hyperparameters for $Algorithm2$ include, single headed self-attention, Adam Optimizer with learning rate of 0.00001 with $1e-5$ decay factor, 64 hidden units in FFN and 0.3 dropout. This model is trained for 20 epochs.

5.3.2 Android Apps

For Android apps, our first process involves obtaining embedding matrix from $word2vec$. The word embedding and positional embedding combined give us the final embedding vectors. The embedding layers are not taken into account during backpropagation. So we keep the parameter $requires\_grad$ as $False$ in PyTorch. We train our models with three different dimension of embedding matrices from 128, 256 to 512. Our input sequence length of API-calls is 4000. This is a big number which, along with embedding dimension, significantly increases the computational complexity per layer, especially when self-attention is involved.

In $Algorithm1$, we get our best performing model with following hyperparameters: 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. Total training time for 20 epochs was about 7 GPU hours.

As $Algorithm2$ is computationally expensive, we set the batch size to 2 and train the model for 20 epochs. The best performing hyperparameters 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.

CHAPTER 6 EVALUATION

In this chapter, we discuss the results of our experiments implementing $Algorithm1$ and $Algorithm2$ on external datasets (Amazon Review, MNIST) and API-calls from Android apps. We also compare the ML-algorithm classificaiton results of using 471 features from Roy *et al.* [27] on our dataset of Android Apps. We consider auPRC (Area Under Precision Recall Curve) as our core evaluation metric [27] as it allows us to capture the effect of the large number of benign sample on the model's performance.

6.1 Dataset Review

Here, we briefly remind the readers about the dataset that we considered in this experiment before discussing about the results obtained.

6.1.1 External Datasets

Table 6.1 and Table 6.2 shows train-test split for our external datasets.

Table 6.1: MNIST dataset train-test split

	Train	**Validation**	**Test**
# of images	48,000	12,000	10,000

Table 6.2: Amazon Reviews dataset train-test split

	Train	**Validation**	**Test**
# of reviews	200,000	10,000	10,000

6.1.2 Android API-Calls

A more detailed information of the API-call dataset is available in Section 5.2. Table 6.3 gives an overview of the test-train split used for our models. The same dataset is used for both $Algorithm1$ and $Algorithm2$.

Table 6.3: API-calls test-train 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**

6.2 Evaluation on External Datasets

Here, we discuss our findings on the implementation of our models on our preparational datasets.

6.2.1 MNIST Dataset

Fig. 6.1 shows the distribution map of the top selected features on MNIST dataset. The red color indicates the top 100 features whereas other shades are features of lower ranks. Here, the model was able to correctly predict the digit $zero$ (on the first column) by focusing on a the circular part in the middle. This gives us some visual explanation on how the model was able to classify the test data correctly.

In the test image, only about 100-200 pixels are relevant. Here, we observe that $Algorithm1$ gives minor weights to all surrounding pixels. Alternatively, $Algorithm2$ focuses highly on top features and has less attention to all other secondary features. Both algorithms are able to focus on the important part of the input.

6.2.2 Amazon Reviews Dataset

This this section we disscuss on the results of our models on the review dataset. We stress that the reviews are based in English vocabulary and it is trivial for a person with the knowledge of English language to classify a review by reading the sentence and identifying the keywords that are responsible in the decision making. We also notify the reader that we did not implement a

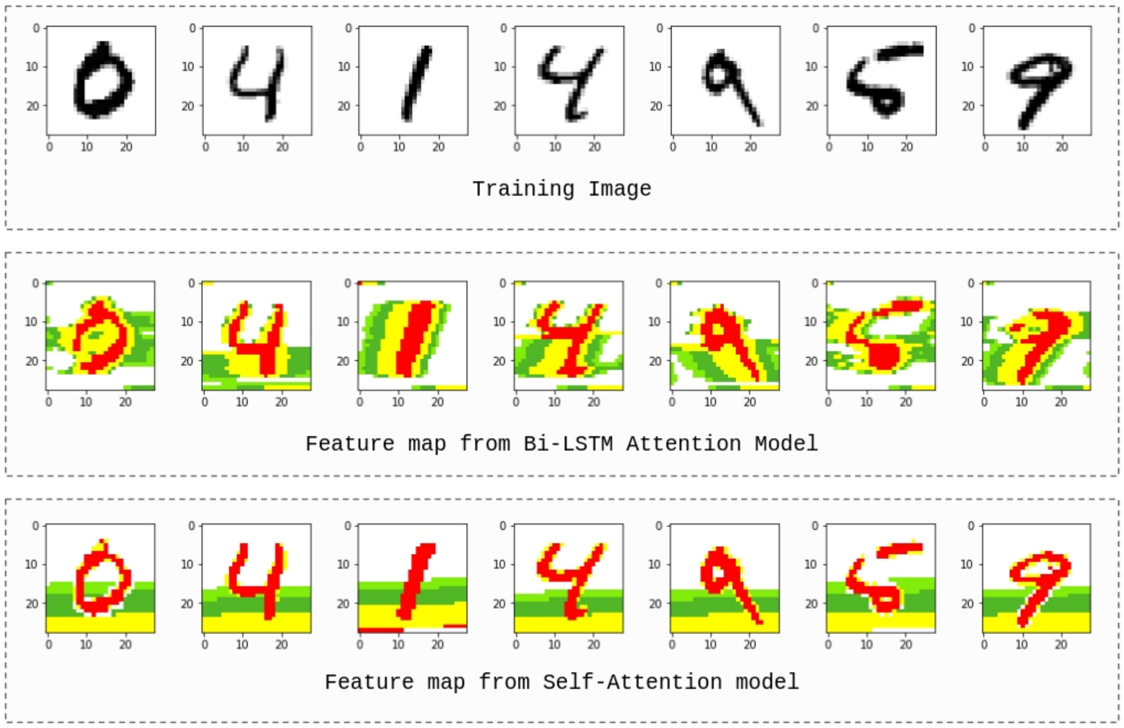


Fig. 6.1. Feature distribution map of test data in MNIST

rigorous optimization process in this experiment as this done just to test our models and verify that it is working properly.

6.2.2 Bi-LSTM Attention Model

Fig. 6.2 shows how our model has slowly moved towards convergence. Confusion matrix shown in the Fig. 6.3 reflect that our classification model is working well. It shows that 596 good reviews were classified as bad. Similarly, the false positive for bad review was 220. The recall for bad reviews was 0.9552 with F1-score of 0.9199. It performs better on bad reviews than on the good ones.

6.2.2 Self-Attention Model

Fig. 6.4 shows how our model based on self-attention slowly converges. Confusion matrix as in the Fig. 6.5 show that our classification model is good enough. It shows that 391 good reviews were classified as bad. Similarly, the false positive for bad review was 512. The recall for bad

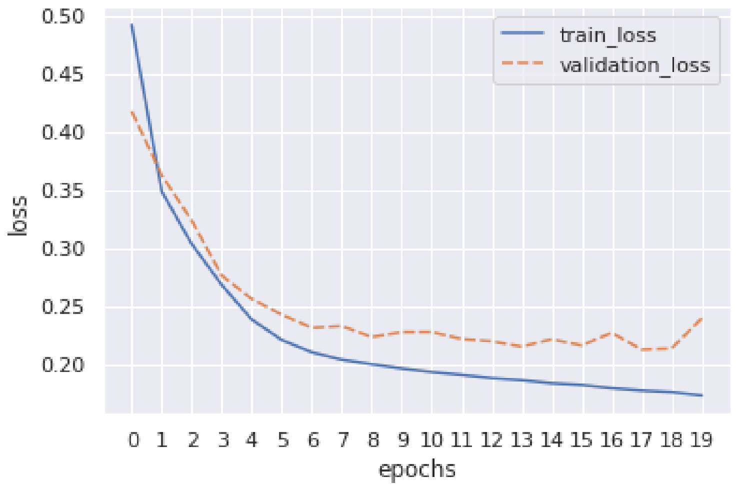


Fig. 6.2. Training vs Validation loss using Bi-LSTM attention in Amazon Reviews dataset

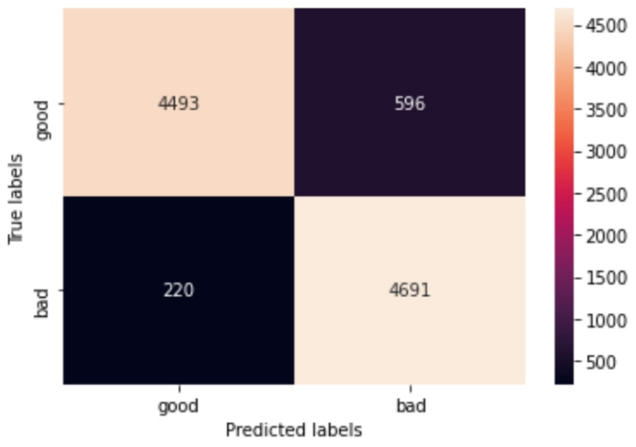


Fig. 6.3. Confusion matrix for Bi-LSTM attention model for reviews dataset

reviews was 0.8957 with F1-score of 0.9069. The model performs better on the good reviews than the bad ones. The reason for the misclassification could be due to multiple reasons such as: new word in the vocabulary, grey area between good and bad, spelling errors, etc.

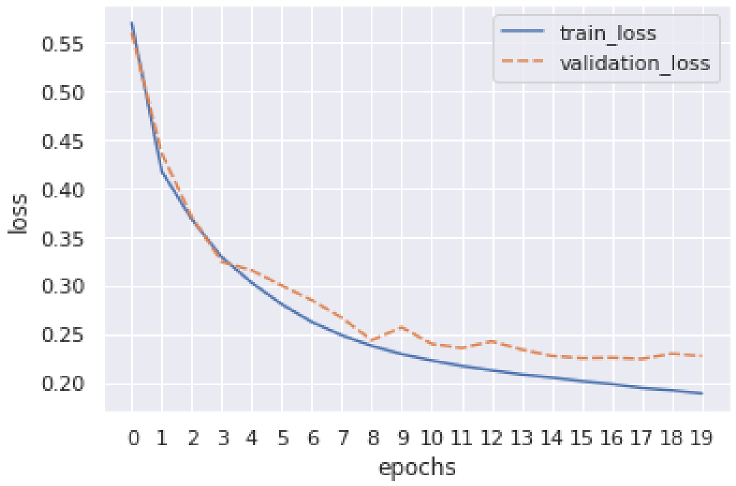


Fig. 6.4. Training vs Validation loss using Self-Attention in Amazon Reviews dataset

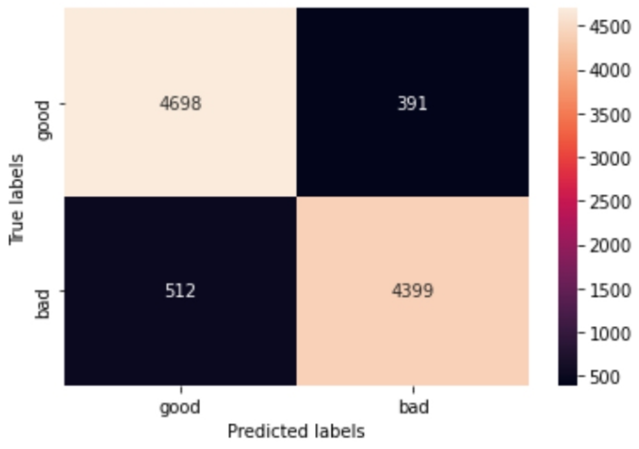


Fig. 6.5. Confusion matrix for Self-Attention model for reviews dataset

6.2.2 Comparison of Bi-LSTM-Attn and Self-Attn Models

Table 6.4 shows the evaluation results of our models. It is observed that both the model give competitive results.

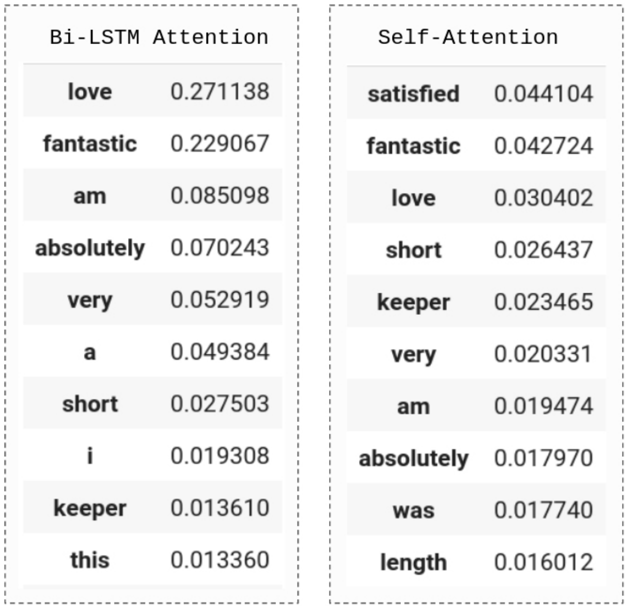
To further observe the results, we dive more into the attention weights from our two models. We take a look at few test instance and see how our model attends to the important features in the

Table 6.4: Comparison of two models for Amazon Reviews dataset

Model	Review Type	Precision	Recall	F1-Score	Area Under PR Curve
Bi-LSTM Attn	Positive	0.9533	0.8828	0.9167	0.9762
	Negative	0.8872	0.9552	0.9199	
Self-Attn	Positive	0.9017	0.9231	0.9123	0.9691
	Negative	0.9183	0.8957	0.9069	

input sentence. Here we only show the top 10 features ranked based on the $Softmax$ scores. We also notify the reader that in the Self-Attention, we take an average of 4 heads.

Example 1: *"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 "*



Bi-LSTM Attention

love	0.271138
fantastic	0.229067
am	0.085098
absolutely	0.070243
very	0.052919
a	0.049384
short	0.027503
i	0.019308
keeper	0.013610
this	0.013360

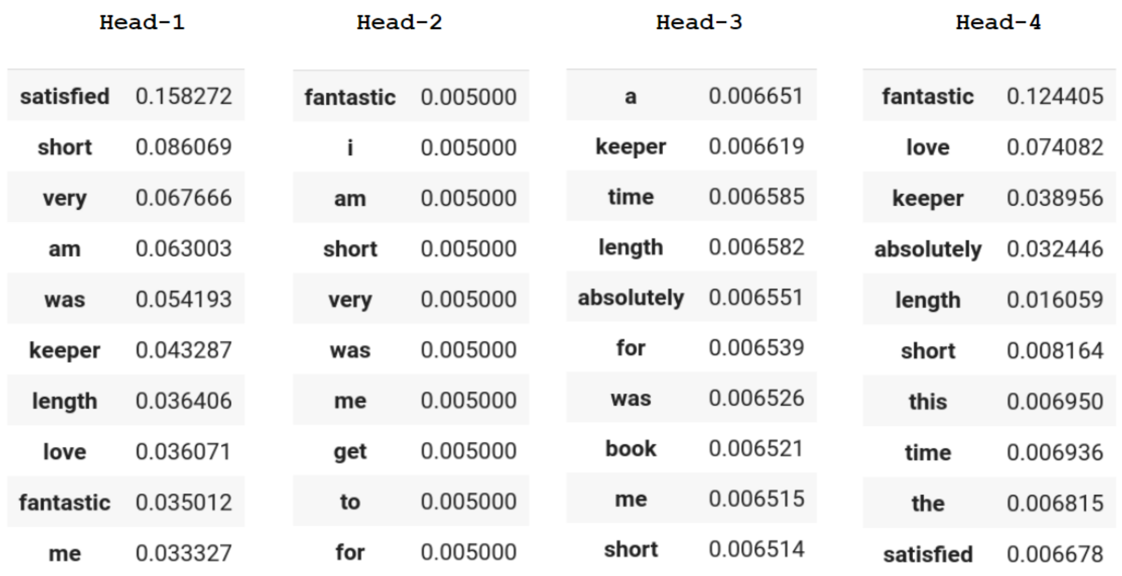
Self-Attention

satisfied	0.044104
fantastic	0.042724
love	0.030402
short	0.026437
keeper	0.023465
very	0.020331
am	0.019474
absolutely	0.017970
was	0.017740
length	0.016012

Fig. 6.6. Example review 1

Here, Bi-LSTM Attention model attends to the words $love$ and $fantastic$ to denote positive sentiment. Self-Attention model attends to word $satisfied$ and $fantastic$ as the top two words in making decision. One other thing that we observe here is that the difference in weights after the top 2 words is very large.

Fig. 6.7 shows how our model makes the decision with multiple heads. Here, Head-1 and Head-4 give us the most relevant words.



Head-1		Head-2		Head-3		Head-4	
satisfied	0.158272	fantastic	0.005000	a	0.006651	fantastic	0.124405
short	0.086069	i	0.005000	keeper	0.006619	love	0.074082
very	0.067666	am	0.005000	time	0.006585	keeper	0.038956
am	0.063003	short	0.005000	length	0.006582	absolutely	0.032446
was	0.054193	very	0.005000	absolutely	0.006551	length	0.016059
keeper	0.043287	was	0.005000	for	0.006539	short	0.008164
length	0.036406	me	0.005000	was	0.006526	this	0.006950
love	0.036071	get	0.005000	book	0.006521	time	0.006936
fantastic	0.035012	to	0.005000	me	0.006515	the	0.006815
me	0.033327	for	0.005000	short	0.006514	satisfied	0.006678

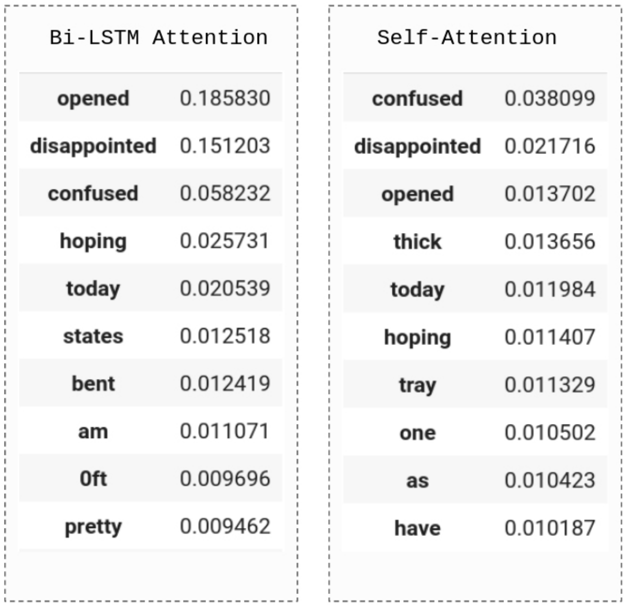
Fig. 6.7. Four heads from Example review 1

Example 2: *"confused i just opened it today and i am disappointed the outside of the box states it is a floor cover and not the trunk tray as written on the shipping label i really dont know if they shipped the wrong item it is supposed to be 00 by 0ft it is not so i am not sure if i got the correct item and i dont want to have to pay to ship it back it is pretty bent i am hoping it straightens out a friend of mine just bought a new fe and admired my cargo tray so i bought her one it is not as thick as the one i have "*

Here, our Bi-LSTM Attention model attends to the word *opened* which we believe could be an outlier. But the subsequent words, *disappointed* and *confused* relates to negative sentiment. Self-Attention model attends to word *confused* and *disappointed* which classifies the review to the negative side.

Example 3: *"darn compelling argument some other guy named shakespeare wrote all that stuff but not the shakespeare we know or actually dont "*

This example doesn't clearly indicate a positive or negative sentiment and lies in a grey area between them. Here, both the models find it hard to get the top words but *complelling* and *darn* makes into top 8 features. We also see *dont* and *not* in our top features which gives it an edge

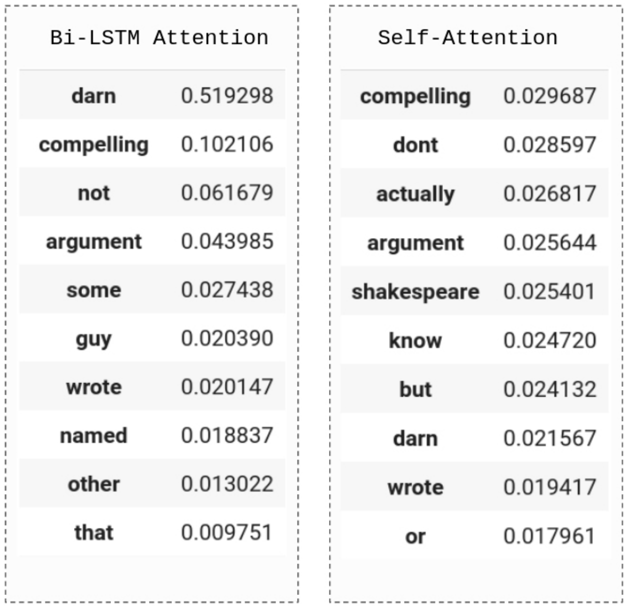


Bi-LSTM Attention	
opened	0.185830
disappointed	0.151203
confused	0.058232
hoping	0.025731
today	0.020539
states	0.012518
bent	0.012419
am	0.011071
0ft	0.009696
pretty	0.009462

Self-Attention	
confused	0.038099
disappointed	0.021716
opened	0.013702
thick	0.013656
today	0.011984
hoping	0.011407
tray	0.011329
one	0.010502
as	0.010423
have	0.010187

Fig. 6.8. Example review 2

towards negative connotation.



Bi-LSTM Attention	
darn	0.519298
compelling	0.102106
not	0.061679
argument	0.043985
some	0.027438
guy	0.020390
wrote	0.020147
named	0.018837
other	0.013022
that	0.009751

Self-Attention	
compelling	0.029687
dont	0.028597
actually	0.026817
argument	0.025644
shakespeare	0.025401
know	0.024720
but	0.024132
darn	0.021567
wrote	0.019417
or	0.017961

Fig. 6.9. Example review 3

6.3 Evaluation on API-Calls

In this section, we now evaluate two models for API-calls dataset. We remind the reader that the previous experiment on Amazon Reviews was based on English vocabulary which was intuitively understandable to people speaking English. In this dataset, we have a new vocabulary that is specific to Android API-calls. So, we trust in the predictive power of our models in identifying the top API-calls.

6.3.1 Bi-LSTM Model

Fig. 6.10 shows the training and validation loss for our Bi-LSTM Attention model. Fig. 6.11 shows the confusion matrix of our Self-Attention model on the test dataset of 17200 apps with 4300 malware and 12900 benign apps. We obtain auPRC of 0.9391. We see that 635 malicious apps were falsely classified as benign apps.

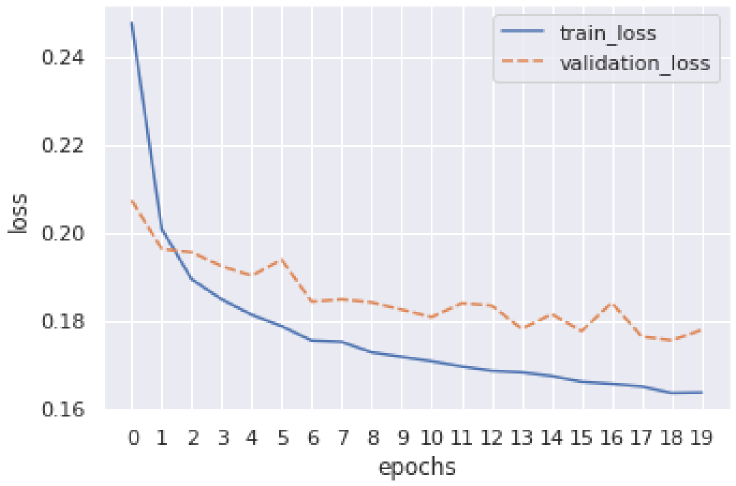


Fig. 6.10. Training vs Validation loss using Bi-LSTM attention in API-calls

6.3.2 Self-Attention Model

Fig. 6.13 shows how our model converges as the training and validation loss decrease. Fig. 6.14 shows the confusion matrix of our Self-Attention model on the test dataset of 4300 malware and 12900 benign apps. We obtain auPRC of 0.9074. We see that 955 malicious apps were falsely classified as benign apps. We found that 850 out of 955 were low-quality malware. We also stress that large proportion of our training dataset was comprised of low-quality malware which is our explanation for the large-number of false positives.

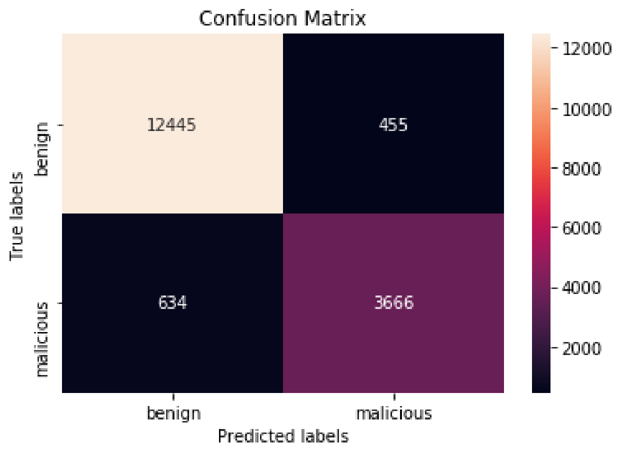


Fig. 6.11. Confusion matrix for Bi-LSTM attention model for API-calls

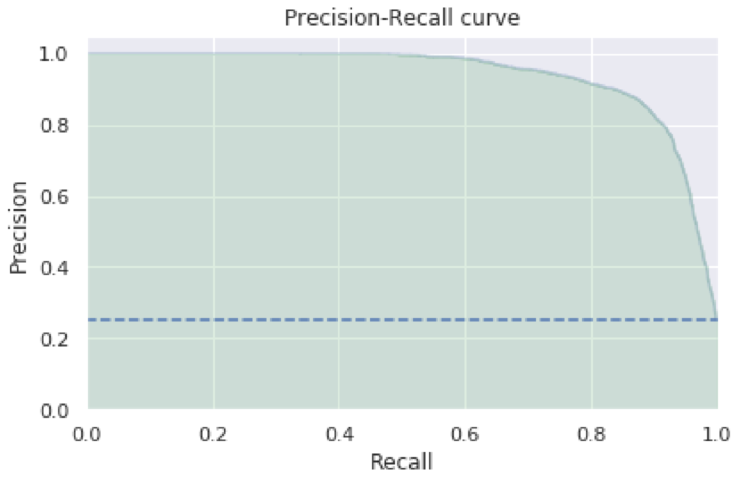


Fig. 6.12. Area under PR-Curve with Bi-LSTM attention model

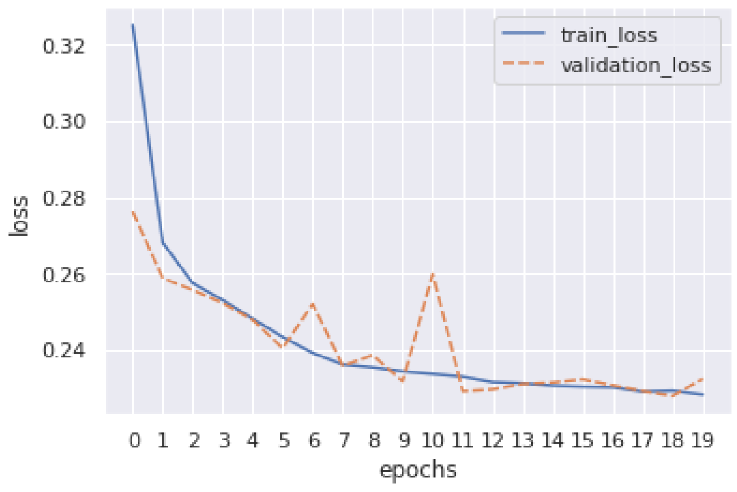


Fig. 6.13. Training vs Validation loss using Self-Attention in API-calls

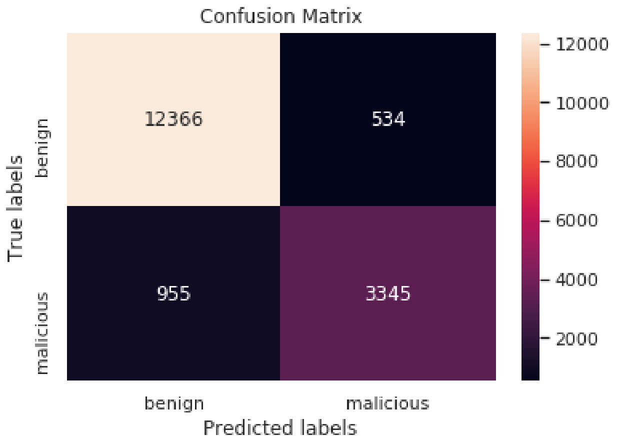


Fig. 6.14. Confusion matrix for Self-Attention model for API-calls

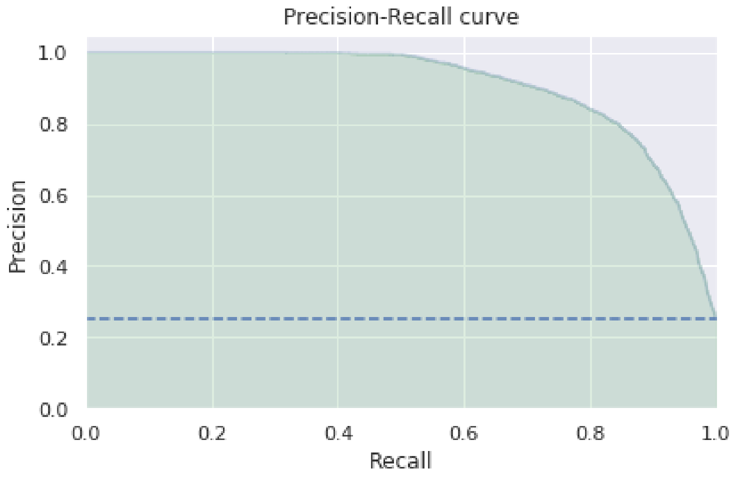


Fig. 6.15. Area under PR-Curve with Self-Attention model

6.3.3 Comparison of Bi-LSTM-Attn and Self-Attn Models

When we compare and observe the results from our two algorithms, we see that both models give competitive results. Bi-LSTM Attn has a slight edge over Self-Attention model. Training Self-Attention is computationally expensive and hyperparameter tuning was difficult as compared to Bi-LSTM attention. Also, we employed a very simple Self-attention model which can be improved with further tuning which we dedicate to future work.

Table 6.5: Comparison of two models for API-calls

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

6.3.4 Comparison with ML-Algorithms

Roy *et al.* [27] used traditional machine learning models with 471 hand-picked features to classify android apps as malicious or benign. We use Roy *et al.*'s feature extraction engine to extract those 471 features. We use the same train-test split with our 172k apps. The results obtained are presented in Table 6.6. We believe that with a larger dataset we can expect better performing deep learning models. The result might look competitive here because of the fact that ML requires hand-picked features whereas our deep learning models learn such features themselves. Chaulagain *et al.* [4] also used 471 features on their dataset to get 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 old dataset before 2016.

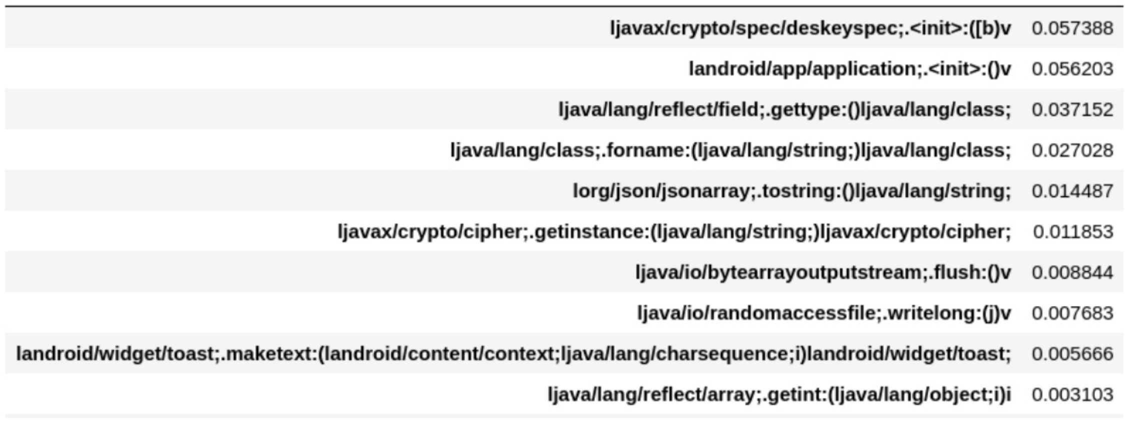
Table 6.6: Standard ML algorithms with 471 features used by Roy et al.

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	

6.4 Discriminative Feature Selection

In this section, we define our selection strategy of our most important features. We can see from the confusion matrix that there were some false positives from both models. To get high quality features, we only make use of the true positive values i.e the malware that were correctly predicted.

Fig. 6.16 shows top 10 features of a true-positive malware from Self-Attention Model. The top 2 api calls from the list: $ljavax/crypto/spec/deskeyspec$ and $landroid/app/application$ are used for malicious activities. Here, in the first API-call, DES (Data Encryption Standard) is a symmetric-key block cipher. The second API call is a constructor initialized when the application is starting, before any activity, service, or receiver objects (excluding content providers) have been created. These calls exhibits that our model is capturing relevant features from the API-calls.



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

Fig. 6.16. API-calls and attention weights for an Android app

Now, using the global attention weights, we match the index of the true positive malware from the test set. We then combine all the weights of each API-call and take a cumulative weights of all the API-calls. This is done to accommodate all the reoccurring API-calls within and across different apps. Fig. 6.17 shows an overview of how the selection process occurs. Here, even though y ranks second in both the apps, it ends up being our top feature. We believe that this deals with the bias of single occurring highly scored feature like m and features like x which may appear repeatedly and have high score as well.

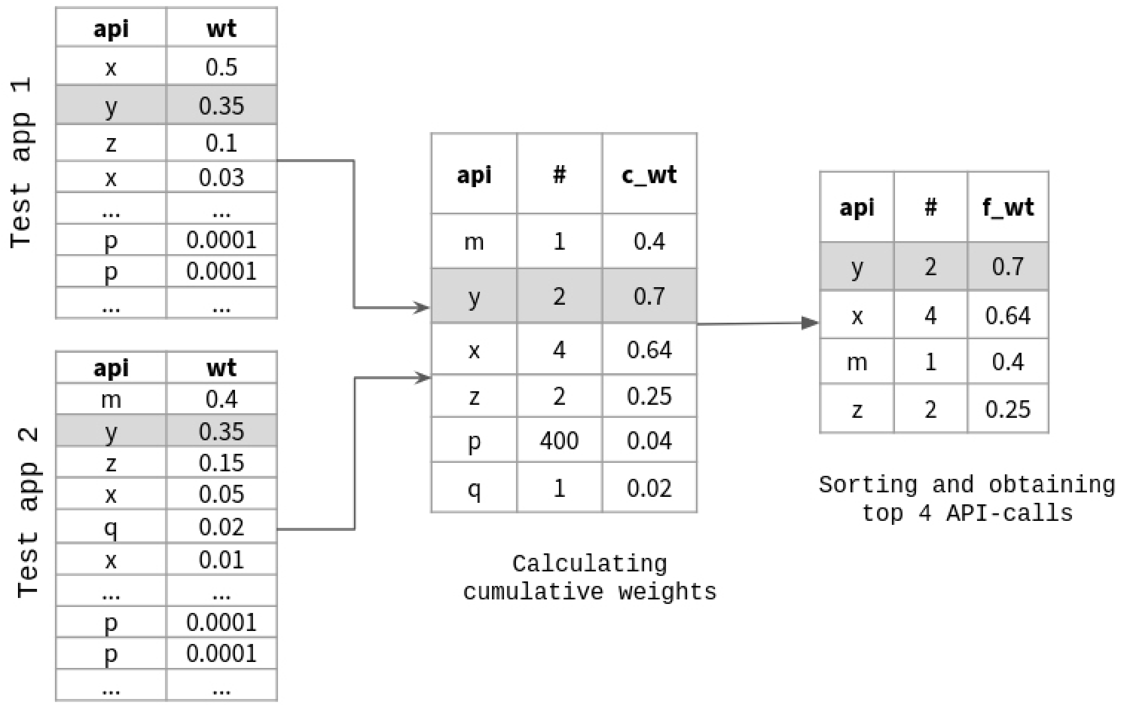


Fig. 6.17. Feature selection strategy

To combine the API-calls from Bi-LSTM Attention and Self-Attention models, we do a simple concatenation and get the combined top 200 API-calls.

On doing such concatenation, 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 6.7 shows the common API-calls between two models.

Table 6.7: 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

6.5 Imbalanced Ratio Experiment

In this side experiment, we vary our malicious to benign apps ratio in our test dataset and discuss the results. We test with 5 different values: 1:3, 1:5, 1:10, 1:20 and 1:50. Fig. 6.18 shows that auPRC of our model decreases greatly as we increase 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. This is challenging because we may not have sufficient high quality malware in our training that matches our benign apps.

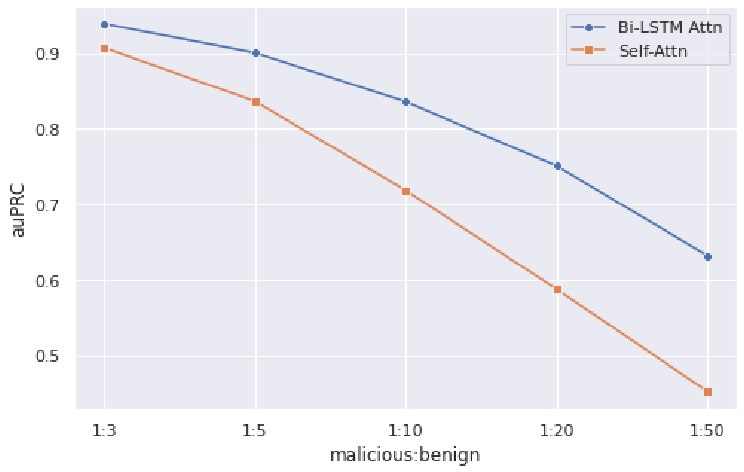


Fig. 6.18. Experiment with varying ratio of malware to benign in test set

6.6 High Quality Malware Experiment

To observe the effect of high and low quality malware, we conduct a side experiment by varying our malware quality in our test dataset. Out of 4300 malicious apps in test dataset, we had only 512 high-quality malware. Matching our original ratio, we brought down our benign apps to 1536 to match the ratio of malicious to benign of $1:3$. This has also been observed from our imbalanced ratio experiment.

Table 6.8 shows that our 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 experiment as future work.

Table 6.8: Model performance on 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-Attn	Benign	0.9283	0.9586	0.9432	0.9291
	Malicious	0.9084	0.7949	0.8479	

CHAPTER 7 CONCLUSION

In this work, we leverage attention-based static analysis approach for security vetting of Android apps. API-calls from the smali files of android apks were used as our artifacts for deep-learning model. Such API-calls are quasi-sequential in nature. We experimented with two attention-based models: Bi-LSTM Attention and Self-attention, which gave competitive results. Additionally, after analyzing the attention weights from our models on the test dataset, we present top 200 API-calls that reflect maliciousness of an Android app.

Initially, we work with two external datasets: Amazon Reviews and MNIST, to verify our models. The words from Amazon reviews are in English vocabulary and the attention weights during inference reflect the positive and negative sentiment of the review. This can simply be verified by our knowledge of English words. MNIST, being an image dataset, shows how our models can also be extended to a different domain to capture relevant features. By looking at the predicted image with attention-weights, we can analyze the top features. On extension of such analogy to our dataset of API-calls, evaluation becomes challenging. API-calls belong to a different language domain. In English language, we have a vocabulary of about 200k. Our vocabulary of API-calls, using $154,800$ apps as training set, is $850,193$. This is more than four times the words that we have in English language. This shows that we were able to cope with such large corpus and hence build two robust deep-learning models to extract the important features. The analysis of the features from attention weights give us a deeper insight on the working of deep-learning methods and thereby revealing how such decision making takes place.

After several hours of training and fine-tuning multiple hyperparameters, we came up with two classification models and obtained the most discriminative API-calls. We achieved an auPRC of 0.9391 and 0.9074 with Bi-LSTM Attention and Self-Attention respectively. Our discussion on the slight edge of Bi-LSTM over Self-Attention is that even though Self-attention can be parallelized it requires a lot of time and resources to train. Hyperparameter tuning in Self-Attention was challenging as the training time was significantly large. So we had to use a simpler model that aligns with our available computational resources. Using a sequence length of 4000 heavily increases the computational complexity per layer. This is also the case with current self-attention based state-of-the-art language models such as BERT which requires very large training corpus and works with a rather smaller maximum sequence length of 512 as compared to our sequence length of 4000. We believe that our approach can be further optimized and could use larger dataset with more high quality malware. Nevertheless, we were still able to implement two attention-based approaches, both RNN and non RNN based, which can deal with sequential inputs giving us some relevant features during inference.

As a future work, instead of selecting single discriminative API-call from the input space, we also want to find subsequence of such API-calls. This can possibly give us a richer malicious signature. Our experiment is solely based on static artifacts of API-calls. We would want to work with system calls during run-time execution of apps. To observe the app's behavior and analyze the system calls being invoked as a result of that triggered behavior would be interesting. This can give us a strict sequence of call execution as opposed to quasi-sequence from static files.

When training our self-attention model, hyperparameter tuning was expensive given the availability of time and resources. We ended up building a rather simple self-attention model with fewer parameters and trained on a single GPU. With more GPUs, we could have trained the self-attention models much faster than the Bi-LSTM models. We would like to extend our model to more sophisticated attention-based language models such as BERT. Similarly, we also simply took an average of weights across multiple heads. An interesting task would be to analyze the attention weights from each head which could give us further interesting information. Our feature selection process combines the features from two models based on their attention weights from the input space. We could also explore several other strategies to combine the two set of saliency features from our models to achieve an optimal set of features.

We experimented with a large number of low-quality malware and our training set involves a ratio of 1:3 of malicious to benign apps. We also want to do a rigorous experiment with different such ratios which could also include training and testing on apps from different years. This could lead to various temporal (yearly) and spatial (malicious to benign ratio) analysis of malware. Our top 200 API-calls can be further experimented and compared with other existing literature. They could also be extensively studied by researches that can help to potentially discover a new malware signature. Similarly, a process to quickly and easily observe the malicious nature of the top API-calls obtained is also an area of interest. One such method could be using these features as one-hot vectors and training ML based classifiers.

Our experiments show that, our models can be implemented into large scale Android security vetting system where we can analyze the ever-growing Android apps quickly. This can save a lot of human time and effort from manually analyzing the apps and hand picking the malicious features. The top features from our models can be further studied by the research community to potentially discover new malware signature.

BIBLIOGRAPHY

[1] Statcounter, “Android OS Market Share,” 2021. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[2] Arstechnica, “Google confirms that advanced backdoor came preinstalled on Android devices,” 2019. [Online]. Available: https://arstechnica.com/information-technology/2019/06/google-confirms-2017-supply-chain-attack-that-sneaked-backdoor-on-android-devices

[3] Google, “PHA Family Highlights: Bread (and Friends),” 2020. [Online]. Available: https://security.googleblog.com/2020/01/pha-family-highlights-bread-and-friends.html

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

[5] Y. Bengio, P. Simard, and P. Frasconi, “Learning Long-term Dependencies with Gradient Descent is Difficult,” *IEEE Transactions on Neural Networks*, pp. 157–166, 1994.

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

[7] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” 2016.

[8] M.-T. Luong, H. Pham, and C. D. Manning, “Effective approaches to attention-based neural machine translation,” 2015.

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2017.

[10] R. Pascanu, T. Mikolov, and Y. Bengio, “Understanding the Exploding Gradient Problem,” *arXiv e-print*, p. arXiv:1211.5063, 2012.

[11] Androguard, “Androguard,” 2011. [Online]. Available: https://github.com/androguard

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

[13] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, “Soot - a java bytecode optimization framework,” in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON ’99. IBM Press, 1999, p. 13.

[14] R. Wiśniewski, “Apktool,” 2018. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[15] Y. Zhou, Z. Wang, Z. Wu, and J. Xuxian, “Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets,” in *Symposium on Network and Distributed System Security (NDSS)*, 2012, pp. 50–52.

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

[17] “UI/Application Excerciser Monkey,” 2019. [Online]. Available: https://stuff.mit.edu/afs/sipb/project/android/docs/tools/help/monkey.html

[18] L. Gong, Z. Li, F. Qian, Z. Zhang, Q. Chen, Z. Qian, H. Lin, and Y. Liu, “Experiences of landing machine learning onto market-scale mobile malware detection,” *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[19] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector Space,” *arXiv e-print*, p. arXiv:1301.3781, 2013.

[20] R. Řehůřek and P. Sojka, “Software Framework for Topic Modelling with Large Corpora,” in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, 2010, pp. 45–50.

[21] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation*, pp. 1735–1780, 1997.

[22] M. Schuster and K. Paliwal, “Bidirectional Recurrent Neural Networks,” *IEEE Transactions on Signal Processing*, pp. 2673–2681, 1997.

[23] P. Zhou, W. Shi, J. Tian, Z. Qi, B. Li, H. Hao, and B. Xu, “Attention-Based Bidirectional Long Short-Term Memory Networks for Relation Classification,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2016, pp. 207–212.

[24] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” 2014.

[25] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, “On the properties of neural machine translation: Encoder-decoder approaches,” 2014.

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps,” *SIGPLAN*, pp. 259–269, 2014.

[27] S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara, “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)*, 2015, pp. 81–90.

[28] S. Roy, D. Chaulagain, and S. Bhusal, “Book Chapter: Static Analysis for Security Vetting of Android Apps,” in *From Database to Cyber Security: Essays Dedicated to Sushil Jajodia*

on the Occasion of His 70th Birthday. Springer International Publishing, 2018, pp. 375–404.

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

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

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

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

[33] S. Hou, A. Saas, L. Chen, and Y. Ye, "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)*, 2016, pp. 104–111.

[34] "Genymotion Android Emulator," 2018. [Online]. Available: https://www.genymotion.com/

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

[36] A. M. Rush, S. Chopra, and J. Weston, "A neural attention model for abstractive sentence summarization," 2015.

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

[38] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf