National College of Ireland

Novel Text and Image Based Approach to Android Malware Detection

MSc Research Project
Data Analytics

Agnideep Pal
Student ID: x21141631

School of Computing
National College of Ireland

Supervisor: Prof. Anderson Simiscuka

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Agnideep Pal
Student ID:	x21141631
Programme:	Data Analytics
Year:	2022
Module:	MSc Research Project
Supervisor:	Prof. Anderson Simiscuka
Submission Due Date:	15/12/2022
Project Title:	Novel Text and Image Based Approach to Android Malware Detection
Word Count:	5362
Page Count:	20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Agnideep Pal
Date:	1st February 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Novel Text and Image Based Approach to Android Malware Detection

Agnideep Pal
x21141631

Abstract

This research examined two unique, image and text-based malware detection strategies before proposing novel classification techniques. It is clear from previous research in the field that feature engineering is crucial for producing high-quality training datasets. This study mainly focused on feature engineering techniques for proper classification of android malwares with the data obtained from either Dex or manifest files, or both. While reshaping the malware detection challenge to a text processing problem, a novel approach was identified, proposed and tested. Along with it, another innovative technique was developed in the image-based approach. Both the approaches demonstrated encouraging results in terms of classifier accuracy, precision, recall, and F1 score. Although, the dataset was highly unbalanced, the text-based models attained approx. 95% accuracy. On the other hand, using the same dataset, the image-based models had an accuracy rate of about 84%.

1 Introduction

Around 6.5 billion handset subscribers were reported[1] by Statista 2022. They also predicted that number to rise to 7.5 billion by 2027. The majority of these portable devices run on either "Android" or "IOS". Another Statista research[2] about the global market share of mobile operating systems showed that Android was leading with about 71% of the share. This huge customer base and the opensource nature of "Android" makes it an alluring target and incredibly susceptible to hacking and cyber-attacks. Although this platform's user apps are distributed through safe app stores like the Google Play Store, Huawei App Store, etc., they are not completely secure against cyberattacks. This year, the Bitdefender cyber security team found 35 malicious apps being distributed through the Google Play Store. Till date these applications were downloaded by over 2 million users. Attackers mostly gain access to the user devices through well-known repackaged apps offered in app stores like apkpure.com and similar websites. These applications mimic the originals in terms of appearance and functionality, but they conceal a virus inside. Hence, this research is very important for understanding and mitigating the problem. To date, numerous research has been conducted to develop strategies for identifying malicious software, or "Malwares". Static and dynamic analysis are the two primary approaches, the industry often uses for malware detection. While dynamic analysis entails

[1] 1: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

[2] 2:https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

executing the application in a secure sandboxed environment, static analysis relies on detailed inspection of the code segments of the apps to spot malicious behavior. This research examines previous works related on malware detection using static analysis and tries to provide suggestions for improving them. Although there has been much research on enhancing the functionality of neural network models, the focus of this study is on enhancing app characterisation techniques for improved feature extraction. It attempts to provide an answer to the question of whether and by how much, a more cautious approach to feature selection and app characterization affects the accuracy of malware classification. The next paragraph outlines the structure of the report.

Section 2 conducts detail discussion on the past research works undertaken in the domain by reputed researchers. Section 3 gives step-by-step overview of the research work conducted using KDD methodology. Section 4 talks about core technical design and the proposed solutions which brings novelty to the project. Section 5 describes in detail the implementation of the proposed solution. Section 6 evaluates and compares the results of all the experiments carried out in the project.

2 Related Work

This section reviews earlier research in the field. Two main strategies are being investigated in this project, hence distinct subsections have been included for distinct article reviews.

2.1 Natural Language Processing (NLP) based Android Malware Detection

In their paper (Zhang, Tan, Yang and Li (2021)), the researchers suggested TC-Droid, a brand-new endto-end malware detection method based on NLP. This technique uses a CNN model that receives a sequence of strings as input, which were retrieved from detailed analysis reports on each sample application. These reports were generated using hybrid malware analysis tool named "AndroPyTool". The drawbacks of traditional feature engineering are eliminated by this strategy's completely automated feature extraction and selection process. However, a significant amount of information that could have been mined from the source code was left behind because the experiments in the research were primarily focused on semantic features derived exclusively from the manifest file.

Another study (Lakshmanarao and Shashi (2022)) exploited textual semantic features in the Dalvik bytecode obtained from Dex files for feature engineering and malware classification. They started with the extraction of actual opcode sequences and before creating a list out of it to feed the proposed RNN model, they tokenized them and sent them to the embedding layer to construct the corresponding embedding space. Although the proposed model showed performance well above 90on the size of the application, each sample application has anywhere from a few thousand to hundreds of thousands of opcodes. However, in this strategy the sequences length for each sample were limited to 200 tokens, which meant much of contextual information in the opcodes were lost in the process.

In the research(Garg and Baliyan (2022)), the researchers proposed a two-step strategy for malware detection and family classification. They also generated a mapping between

malwares and OS vulnerabilities. In the first step, they used encoder-based transformers like BERT and XLNET to generate dynamic word embeddings from a large text corpus. This corpus was generated from malware description reports by companies like Kaspersky, VirusTotal etc. In the last step they used DL classifiers like MLP, TextCNN and RNN for multilabel classification. The reports used as input corpus were very comprehensive and were bound to provide promising results. Although the results were impressive, the strategy solely depends on the description reports generated by the security companies rather than the readily available APK file.

Yet again in another research (Oak et al. (2019)) the researchers have proposed a BERT based sequence modelling strategy for android malware detection. However, this research made use of only dynamic analysis to collect required input data. The main goal of the research was to capture and utilize the activity sequences of each application for malware classification. For this they generated n-grams sequence of features for each application before passing onto BERT for classification. Apart from this the study also devised a strategy to handle the problem of highly imbalanced dataset. Although the results of the research were satisfactory, there is a scope for investigation into malware family classification using the same approach.

In the research(Raghav et al. (2021)) the researchers have proposed an unique strategy for malware classification using document vectors. They used PV-BOW (Paragraph Vector - Bag of Words) model to generate necessary document embeddings. Each embedding vectors were then labelled before passing them into models like Logistic Regression, SVM and CNN for training. These embedding vectors were generated from both the manifest and dex files using the afore-mentioned algorithm. Multiple experiments were conducted using different dimensional vectors (200 and 400) which yielded overall good results

In the research (Yeboah and Baz Musah (2022)) the researchers have proposed an unique 1D CNN model based NLP technique for malware detection. In this research malware detection was treated as a time series problem. Like other research the technique involves decompilation of the APK files and generation of corresponding bigram and unigram sequences. The sequences were then unified across application using the method of truncation. These sequences were passed through CBOW Word2Vec embeddings to generate separate features for both bigram and unigram sequences before feeding them onto the CNN model. The outputs of the CNN model were then fused together using weighted average boosting ensemble to derive the final classification results. Although, the strategy is unique there is still a problem with the truncation of the input sequences. Such action is bound to cause huge loss of information and thus underfitting of the model

In the paper (Fan et al. (2020)), the researchers proposed a novel language based strategy to automatically identify useful information about android malwares from various internet blogs and map them to application code. CTDroid was the moniker given to the approach. This approach has produced commendable outcomes thus far.

In another research (Islam et al. (2020)), the researchers experimented with n-gram techniques for android malware detection. They proposed a unique stacking mechanism using unigram, bigram and trigram sequences. The results of the experiments on the machine learning algorithms like Random Forest (RF), XGBoost were very impressive.

In the paper (Wang et al. (2022)) the researchers proposes a framework for malware detection based on multiple features. They used analysis tools to generate detailed reports (both static and dynamic analysis) on the application and then enhanced the reports using statistical tools. Finally, they transformed the reports into 2D matrix before passing them into CNN model for representation learning. Although, the strategy outperformed most

other state-of-art models, it is still a unclear, whether it can be run on premise in hand held devices.

2.2 Image Processing based Android Malware Detection

The research (Li et al. (2022)) proposes an unique malware detection mechanism using multiple features and a decision layer to unite them. Unlike other hybrid (merging static & dynamic analysis output into one input source) methodologies, this research used three parallel input sources (dex files, manifest and certificates) forwarding them to three different VGC16 classifiers and fusing their results at the end using a decision algorithm. Before passing the data to the classifiers they converted the data sources to their respective binary strings and finally to RGB images. Although, the researchers conclude the paper by citing better performance of the model compared to the existing single and hybrid malware detection mechanisms, the paper lacks the comparative study of efficiency of the model in terms of utilizing computational resources.

In the papers(Darwaish and Naït-Abdesselam (2020); Darwaish et al. (2021)) researchers propose a new technique to convert apk files into RGB images for training a CNN detector. They first built a dictionary of benign and malign features from the manifest and dex files extracted from all the compressed apk files. Using the dictionary, they encoded all the benign features from dex file of an application in red channel of the RGB images. The Green channel hold benign information from the manifest. whereas the blue channel comprised of the malicious features from both the dex and manifest files. To unify the dimensions of all the channels they were subjected to Nearest Neighbor Interpolation (NNI) which generated uniform 64 x 64 pixels RGB images. This methodology showed promising results by detecting malwares in both legacy and new applications from the year 2009 to 2020. However, this paper didn't study the effects of obfuscation techniques used in the present-day applications.

The research (Almomani et al. (2022)) proposed 16 state-of-art pre-trained vision based CNN models (Xception, VGC, MobileNetV2, ResNet50, InceptionV3 etc) for training and detecting malwares. As part of input pre-processing, all the apk files were converted to their respective grayscale and colour images and then resized according to the input requirements of the respective pre-trained models. These transfer learned models were then fine-tuned further to yield even better results. Among them the Xception model gave the best results. It also compared the performance of the models for the grayscale and RGB images. However, apart from a comparative study of the models they didn't provide any insight to combine the bests of each model to build a single concrete and comprehensive AMD classifier.

This paper (Huang and Kao (2018)) was among the first to propose an image based mechanism to detect android malwares. The researchers used tools like APKTool, Baksmali to convert bytecodes of each apk file to their respective RGB images. They then used these images as input to the state-of-art models like AlexNet, VGC, GoogleNet and InceptionV3. They also performed image distance test using algorithms like Levenstein, RMS deviation on the generated images and determined that the image of samples of same android family share similarity in the visual pattern. This research demonstrated automatic feature extraction by the CNN models from the images and reducing resources required for manual pre-processing of input data and feature selection and extraction. Although, it had fast detection time when compared to existing mechanisms it fails in certain parameters like DR /FPR and Accuracy due to large sample size.

In the paper (Zhang, Luktarhan, Ding and Lu (2021)) the researchers proposed an unique methodology for generating images from apks and forward them to well known time series model TCN (Temporal Convolutional Network). They generated four set 24 x 24 images by mixing and matching binary data from xml files and the dex files. With this data as an input they performed several experiments with different optimizers and learning rates on three different models viz CNN (Base model), MobileNetV2 and TCN. TCN stood out to be the best in terms of both computational efficiency and classification accuracy. However, the measured accuracy of the TCN can be improved by expanding the receptive field of the model.

2.3 Discussion

Numerous studies have already been conducted on the semantic analysis of opcode sequences for malware identification. One common thread that ran across all of the NLP techniques was the restriction on the length of input sequences provided to the language models. As required by the most recent models, nearly all studies had to set the length between 200 and 500 tokens. This meant that much of the information was lost during the training process. Some strategies even managed to get around the issue by using unsupervised algorithms like transformers (BERT), auto-encoders, etc. where new representations of reduced dimensionality were learned from high dimensional sequences. Additionally, LSTM experiences performance overhead as sequence length increases. In the paper (Vaswani et al. (2017)) the researches introduced transformers to solve this problem.

This research proposed a novel solution to the above problem using unique strategy discussed in the Section 4. Apart from that, it also provided a scope for improvement in the feature selection and app characterization process in the Image based models. This new methodology for app characterization is discussed in detail in Section 4.

3 Methodology

The methods employed in the project to study end-to-end techniques for Android Malware Detection are explained in this section. The key topics covered are, general overview of the chosen dataset, data pre-processing steps, model architectures, and the malware detection procedure. These details are explained in the sub-sections 3.1 and 3.2 below.

3.1 Dataset

The dataset (Taheri et al. (2019)) selected for this research was collected from the official website of Canadian Institute of Cybersecurity at UNB (University of New Brunswick). It includes 1596 Apk files with the most current samples belonging from the year 2020. The 6 samples are divided into six malware categories, with each category are then further divided into their respective malware families. The primary categories, which comprises up to 496 Apk files, are Riskware, Adware, Banking Malware, SMS Malware, and Banking Malware. The remaining 1100 Apk files are the benign samples that were gathered from the years 2015, 2016, and 2017.

3.2 KDD Steps

The project used the KDD (Knowledge Discovery in Databases) technique, which is illustrated in Figure 1 along with the details in the subsections below

3.2.1 Data Collection

In this step the dataset was downloaded from official website of Canadian Institute of Cybersecurity. The APK files in the downloaded dataset were already organized into zip files according to their categories. These zip files were extracted to a local filesystem and made accessible to project code for further analysis

3.2.2 Data Pre-processing

In this pre-processing step the downloaded zip files were dynamically unpacked into suitable location. Next the APK files were renamed using a hashing algorithm in order to uniquely identify them in future steps of the project. The available file information for each APK were then captured and organized into a logical tabular format for further analysis and experiments.

3.2.3 Data Transformation

During this step the unpacked APK files were further decompiled into its constituent components using specialized software. These components were then filtered and chosen based on the proposed direction of the research. Finally, the chosen components were then transformed, using specialized strategies, into formats suitable for deep neural networks employed in the project.

3.2.4 Data Mining

In this step, advanced Image and text based neural network models were employed to automatically learn the latent semantic features from within the huge amount of generated input data. Apart from the feature extraction, the models were also useful in classifying the input data into its appropriate classes.

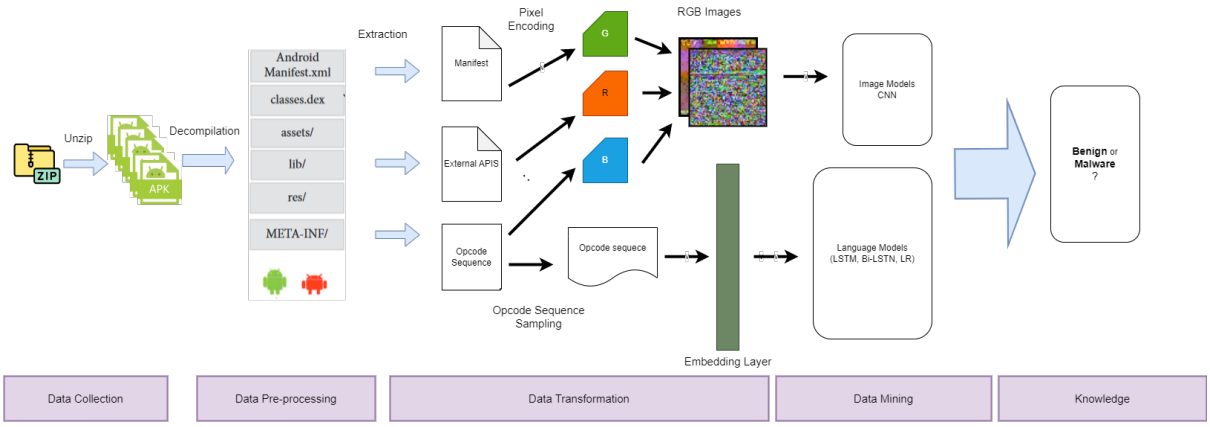

Figure 1: Research Methodology

3.2.5 Evaluation

The proposed methodologies and models were evaluated in this step. Several experiments were carried out and their results compared based on accuracy, precision, recall, F1 scores and their Inference times. Apart from that, the models from both text and image-based approaches were compared as well. Line graphs were generated using the validation accuracy and losses against the epochs for the each one of the implemented neural network models to track the training progress. AUC/ROC curves for each of the models were also generated to better understand their performances. Details on these are provided in the Section 6

3.2.6 Interpretation

Several scenarios were simulated in the experiments and their performances were recorded during the evaluation phase. In this step, these results were compared to extract critical knowledge.

4 Design Specification

This project mainly focused on Static analysis of the APK files. This technique of analysis is governed by the thorough study of the application behaviour through its source code. The objective is twofold viz

- Misuse Detection (resource misuse)
- Anomaly Identification (behavioural anomaly) in the target application.

Unlike traditional machine learning approaches, the end-to-end deep learning approach, adopted in this project automatically learns latent features within the source code of the application to achieve the above objectives. The quality of this learning mostly depends upon two core things namely **App characterization** and **Neural Network Model implementation**. A brief description of the Android App Characterization and Model Implementation is provided in the sub-sections 4.1 and 4.2 respectively for clear understanding

4.1 Android App Characterization

In the paper(QiuJunyang et al. (2020)) the researchers have defined **"App Characterization"** as the systematic process of properly representing an application to the appropriate neural network models. They have also listed two broad ways of characterizing an application, widely adopted by most researchers. These are

- Dalvik bytecode sequences from **"classes.dex"** and meta information from the **"AndroidManifest.xml"** file for sequential analysis.
- Dalvik bytecode translated to RGB values to generate images for advanced image processing.

Both the ways have been used for the experiments in this project. They have been described in detail in the following subsections.

4.1.1 App Characterization for Text based Analysis:

Dalvik Bytecode/Opcode sequences from **"classes.dex"** files were chosen to represent an application for this task. Ideally, the cumulative number of all opcodes extracted from the Dex files of an application may range somewhere between 1k to 200k. If each opcode is 8 considered as a single token, the number is too large to be max sequence length for sentiment analysis models like, LSTM. Moreover, plain splitting of the large sequences into multiple sequence may introduce abnormal behaviour into the language models. Under these circumstances, an intelligent technique was adopted to generate sequences of length between 100 – 500 tokens without losing much information. This novel technique was named as **Opcode Sequence Sampling**. The step-by-step description of the strategy is provided below along with its illustration in the Figure 2:

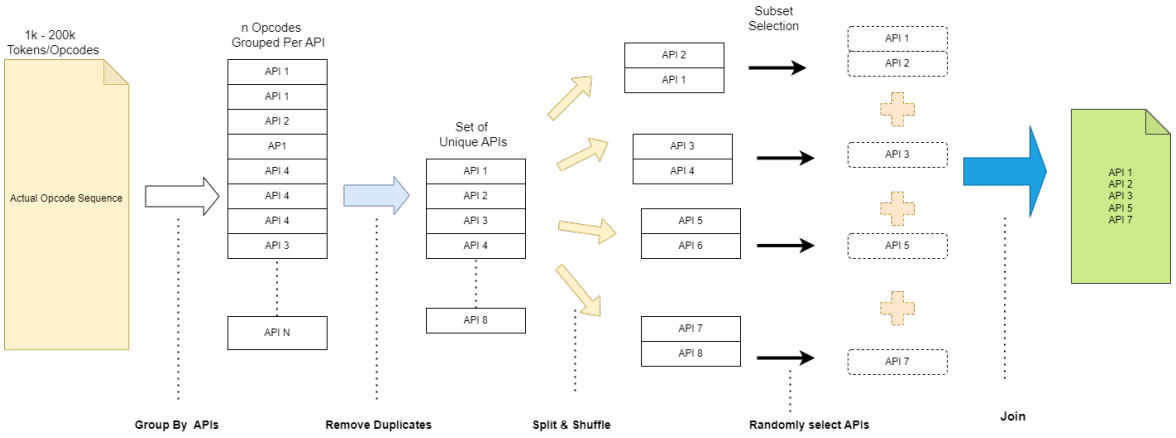

Figure 2: Opcode Sequence Sampling

- **Step 1 : Grouping** - In the Dex files the opcodes are organized as classes and methods. A class contains a State and a Behaviour. A State is represented by variables, fields and constants, whereas the behaviour by its methods/api. These apis are basically, blocks of basic instructions or opcodes as defined in the android official3 website. In this step these opcodes were grouped together based on their parent api and a list were generated. Essentially, the process converted each api into a token.

- **Step 2 : Filtering** - In this step duplicate methods were removed from the list to reduce redundancy.

- **Step 3 : Splitting and Shuffling** The filtered method list was then split into four equal parts and a new list was formed with each splitted list as its entries. After each split operation the list of lists is shuffled.

- **Step 4 : Sampling** - In this step random samples are retrieved from the split lists and concatenated to form a single sequence. This is illustrated in the figure above

4.1.2 App Characterization for Image based Analysis:

The basic idea is to generate RGB images using characteristic features of the application. The process adopted to generate these images is depicted in Figure 3. There are two parts to this process as described below.

- The first one revolves around the strategy behind transforming string values to the corresponding colour pixels. This was done by encoding them into ASCII values and taking their sum. Then modulus operation determined the corresponding pixel for the string.

- The second part employed the colour encoding strategy for generating pixel values for red, green and blue channels using the application information. These generated colour channels were then merged using an image processing library to form the final image. Also, during merging, an interpolation algorithm called **"Nearest Neighbour Interpolation"** are applied on the channels to unify the dimensions of all the channels. Some examples of the generated image files are provided below in the Figure 4. In a novel technique adopted in this research each of the colour channels encoded one distinct feature of the application. The Red channel was used to encode all the external API calls that doesn't appear in the opcodes. The Blue channel comprised only opcodes. The green channel captured all the important information available and accessible from the manifest file. This information constitutes all the permissions, intents, libraries, features, and meta data of the android components. This channel also encoded all the string values from the application. The final image was then stored in an array prior to feeding them to the image classification models.

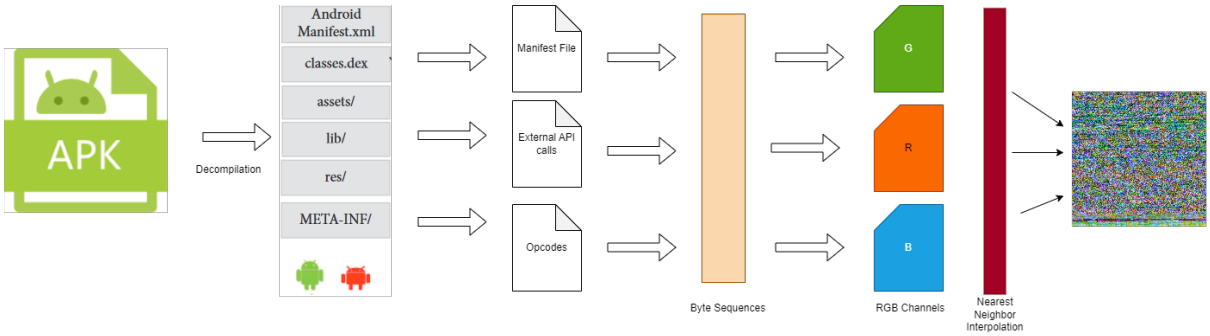

Figure 3: APK to Pixel Transformation

4.2 Model Implementation and Hyper-parameter Tuning

Three language-based models were used for malware classifications, and those are **Logistic Regression**, **LSTM** and **Bidirectional-LSTM**. The tokenization and text embeddings for these models were obtained using **CountVectorizer**, **Tokenizer** and **Word2Vec**. For Image based classification one state-of-the-art pre-trained model was chosen among several based on its parameter size, accuracy and its suitability for running on handheld devices and that was **EfficientNetB4**. A detailed comparison of the most popular Pre-Trained Image models, suitable for running on handheld devices are provided in

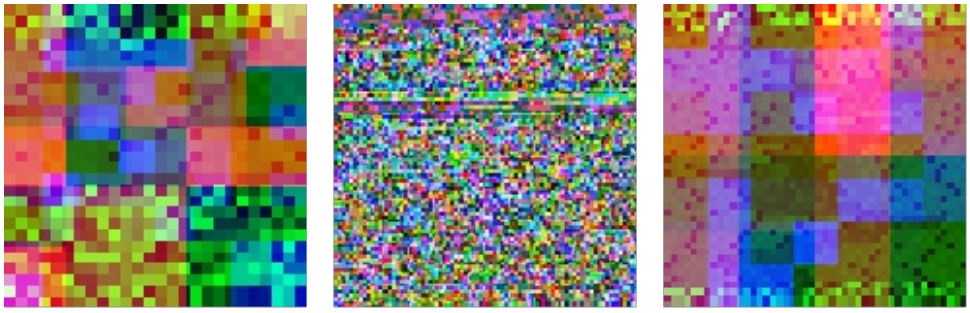

Figure 4: Generated RGB Images

the Table 1 below. Also, a **CNN model** was designed for making a comparison to the chosen pre-trained model. Finally, a novel bandit based hyper-parameter tuning algorithm namely **"HyperBand"** proposed in the paper Li et al. (2017) was used for tuning all the models.

Table 1: Pre-Trained Image Models Comparison

Models	**Size(mb)**	**Acc(Top 1)**	**Acc(Top 5)**	**Parameters**	**Depth**
ResNet50V2	98	76.0%	93.0%	25.6M	103
MobileNetV2	14	71.3%	90.1%	3.5M	105
NasNetMobile	23	74.4%	91.9%	5.3M	389
EfficientNetB4	**75**	**82.9%**	**96.4%**	**19.5M**	**308.3**

5 Implementation

This section discusses the implementation details of experiments conducted in the research. It begins with a brief anatomy of a typical APK file. Consecutively it lists the tools used for decompilation of these files and for transforming the data extracted from it. Before that it also describes the development environment used in the research. Finally, it discusses the implementation details of the app characterization, neural network models and hyperparameter tuning.

5.1 Anatomy of an APK:

APK is a compressed android executable file for installing applications in an android device. This file is generated and signed using **Android App Packaging Tool (AAPT)** and **jar signer** respectively. AAPT is responsible for packaging all the compiled source code, UI resources, manifest file and checksum certificates into a single .apk file format as shown in Figure 5. Also shown in the same figure contents of a typical decompressed apk. A decompressed apk usually consists of 3 important files viz **AndroidManifest.xml, classes.dex, resources.arsc** and 4 subfolders namely **assets, lib, META-INF** and **res**. This project makes use of mainly two files, namely **"classes.dex"** and **"AndroidManifest.xml"**. The details of these files are given below:

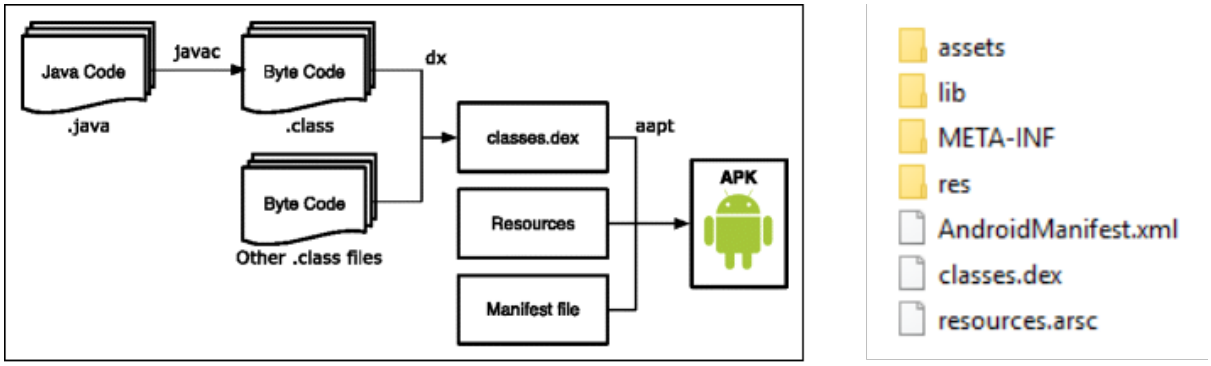

Figure 5: APK Packaging and File Structure

- **Classes.dex:** Dex stands for Dalvik Executable. This file consists of all the java source code compiled into Dalvik bytecode by DX compiler as shown in the figure above.
- **AndroidManifest.xml:** Also known as the manifest file is a xml file, which holds all the meta information like application name, package name, list of required permissions, names of the important components of the applications like activities, services, receivers etc. This file also defines the entry points of the application and details of various action based intent navigation between different components of the app.

5.2 Development Environment:

The following software platforms and tools were employed to carry out the research's experiments:

- OS: Windows 11 Home Single Language
- Language: Python
- Development Tool: Jupyter Notebook
- Package Manager: Conda Environment
- Versioning: Git
- Repository: GitHub
- Cloud Resource: Google Collaboratory & Google Drive

5.3 Important Libraries used in the project

- **AndroGuard:** Interactive and API base python tool for APK analysis and decompilation
- **OpenCV:** Versatile image processing library used in Computer Vision applications.
- **TensorFlow:** Data Pre-processing, Machine Learning and Deep Learning Library
- **Gensim:** Natural Language Processing Library

5.4 Data Pre-processing:

Before the APK files were decompiled, they were dynamically extracted from the compressed zip files and renamed with their md5 dig

5.5 Decompilation:

One of the most important tasks in the project was to extract "classes.dex" and "AndroidManifest.xml" files from the target APK. This was achieved by a powerful tool named **"AndroGuard"**. It was used to dynamically decompile the apk files into its constituent component files described in the Section 5.1. It was also used to extract the opcodes and all relevant information from the Dex and manifest file respectively. This extracted information was then used to generate text sequences for NLP and RGB images for Image processing-based analysis respectively. These two strategies are discussed in the following subsections 5.6 and 5.7.

5.6 NLP Based Strategy

5.6.1 Preparing Input:

Using **"AndroGuard"** opcode names from the application dex files were extracted dynamically and stored in their respective text file. Next using the novel **Opcode Sequence Sampling** technique described in the Section 4.1.1, the original sequences were reduced to smaller ones. A corpus was then generated with all the transformed opcode sequences and vocabulary extracted for training the word embedding models. **CountVectorizer** from **Scikit learn** and **Word2Vec** from **Gensim** were then used to train the respective embedding spaces. Simultaneously all the sequences were tokenized either manually or automatically using **Keras Tokenizer**. To unify the length of the sequences across all the applications necessary padding and masking were added. Finally, these uniform sequences were passed through the embedding layer to generate respective word embeddings suitable to feed onto the next layer of DL models. In case of the Word2Vec embeddings the weights were used to initialize the Keras Embedding Layer

5.6.2 Model Implementation:

The architecture for the basic unit of LSTM and Bi-LSTM models are depicted in the Figure 6 below. Both the models were implemented using Keras APIs

5.7 Image Based Strategy

5.7.1 Preparing Input:

After the decompilation, the opcodes, external api calls and manifest information were dynamically retrieved using **"AndroGuard"** and stored in their respective python lists. During the next step. these lists were converted into a list of respective pixels values using the technique mentioned in Section 4.1.2. These lists were then transformed into their respective 2D matrixes and then subjected to interpolation before getting merged into one single image file. Both the tasks of interpolation and merging was handled by a specialized library called **OpenCV**. The image files were then converted to **3D Numpy array** and stored as **.npy** file prior to feeding them to the image classification models.

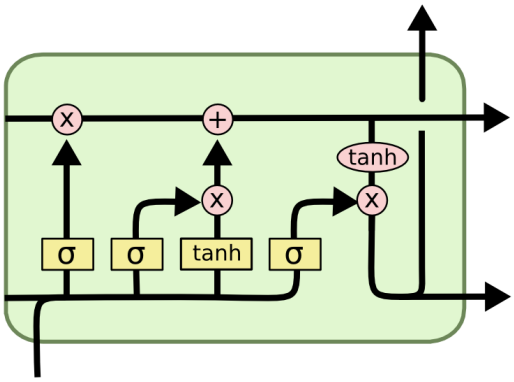

Figure 6: LSTM Unit architecture

5.7.2 Model implementation:

Two models out of which one base **CNN model** and one state-of-art models were implemented during the experiments. The pre-trained model that was employed in the experiments was **EfficientNetB4**. The pre-trained model was selected based on multiple criteria namely, **model size**, **number of parameters** and **accuracy** as can be seen in Table 1 under Section 4. The architecture overview for a typical EfficientNet is provided in the Figure 7.

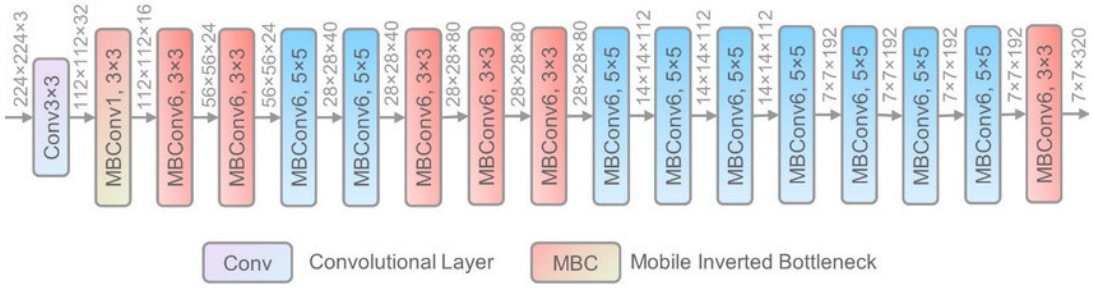

Figure 7: EfficientNet Architecture

6 Evaluation

In this section, the neural network and machine learning models employed during the experiments are compared and evaluated using various metrics. The dataset was split into training, validation, and a test set for the evaluation. The different metrics calculated are Accuracy, Precision, Recall, F1 score, AUC/ROC and Inference Time as displayed in Table 2 and 3

6.1 Model Outcomes

The models implemented in the project classifies whether an android application sample is either **malicious** or **not**, which makes it a two class problem. Malicious applications are labelled as **"1"** and belongs to the **"Positive"** class whereas the benign applications are labelled as **"0"** and belongs to the **"Negative"** class. As a result, any benign application predicted as **"1"** can be referred as **"False Positives"** and any malicious applications predicted as **"0"** as **"False Negatives"**. Similarly, the opposite outcomes are respectively **"True Negatives"** and **"True Positives"**

6.2 Performance Comparison of ML & DL models used in both NLP & Image based experiments

All the models are compared on the basis of performance scores and inference time as explained in the Sub-Sections 6.2.1 and 6.2.2 below

6.2.1 Scores Comparison

Among all the metrics the most important for performance measurement in this project is **"Sensitivity"** or **"Recall"** of the models along with the **"AUC/ROC"** score. This is due to the fact that it is very important that all the models correctly predict the maximum **True Positives** out of all the actual positives while displaying improved capability for correctly separating the classes. The models need not be very accurate when predicting actual negatives.

Table 2: Model Evaluation

Models	Strategy	Acc	Precision	Recall	F1 Score	AUC/ ROC
Count Vectorizer + LR	NLP	0.970	0.962	**0.979**	0.970	0.973
LSTM	NLP	0.934	0.904	**0.964**	0.933	0.936
Bi-LSTM	NLP	0.936	0.967	**0.898**	0.931	0.935
Bi-LSTM + Word2Vec	NLP	0.756	0.670	**0.936**	0.781	0.768
Bi-LSTM + Word2Vec(Tuned)	NLP	0.873	0.854	**0.878**	0.866	0.874
LSTM + Word2Vec	NLP	0.775	0.745	**0.782**	0.763	0.775
LSTM + Word2Vec(Tuned)	NLP	0.838	0.805	**0.859**	0.831	0.840
Base CNN	Image	0.878	0.881	**0.988**	0.931	0.641
Base CNN(Tuned)	Image	0.887	0.897	**0.977**	0.936	0.694
EfficientNetB4	Image	0.775	0.912	**0.811**	0.858	0.700

Following observations are made by comparing the **"Sensitivity"**, **"AUC/ROC"** and the generated plots for the models:

- Out of all the models, the Logistic Regression model's performance paired with the CountVectorizer was the best as can be seen in the Table 2. The AUC/ROC curve for the model is provided in the Figure 8

- It is also clear from the table that the **LSTM** models **without word2vec embeddings** perform significantly better than the rest. The plots for the two best NLP based neural network models are provided below in the Figure 9 and 10 respectively. The **left plot** in both the figures displays the **AUC/ROC** curve. The **top** and the **bottom plots** on the **right side** of the figures show the **learning accuracy** and the **loss** of the model during the training phase.

- When compared to text-based models, the evaluation metrics show the image-based models to perform poorly. **EfficientNetB4** and the **base CNN model** both fell short of outperforming the most elementary text models. Figure 11 and 12 show the plots for the best **EfficentNetB4** and **Base CNN** models obtained, respectively.

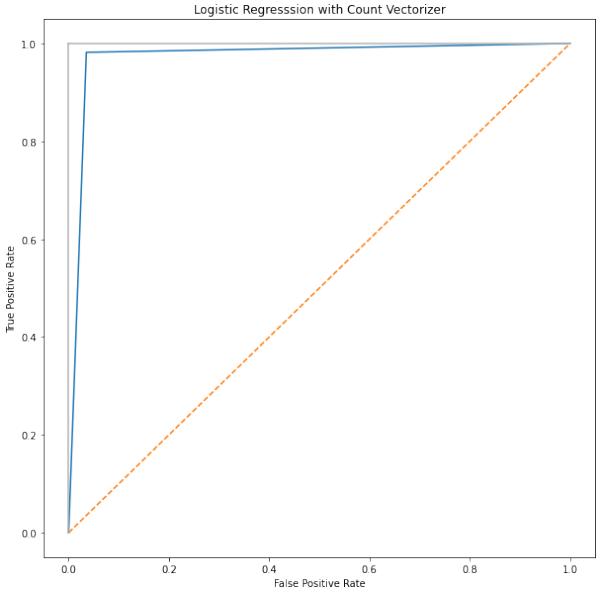

Figure 8: AUC/ROC curve for Logistic Regression + Count Vectorizer

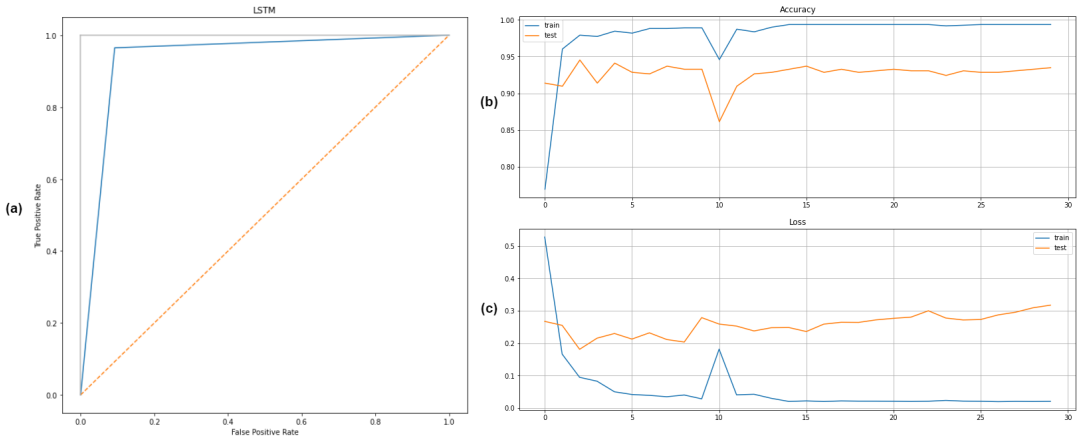

Figure 9: LSTM : a) AUC/ROC Curve, b) Epoch vs Accuracy, c) Epoch vs Loss

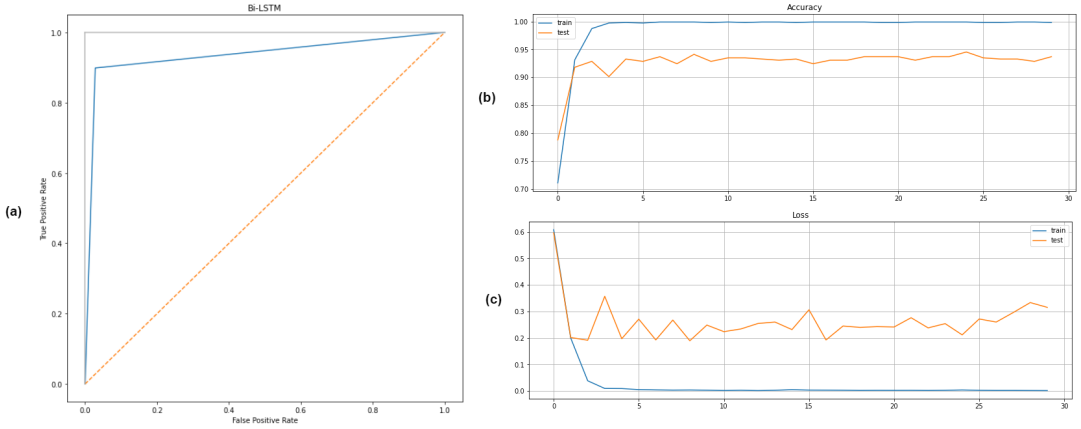

Figure 10: Bi-LSTM : a) AUC/ROC Curve, b) Epoch vs Accuracy, c) Epoch vs Loss

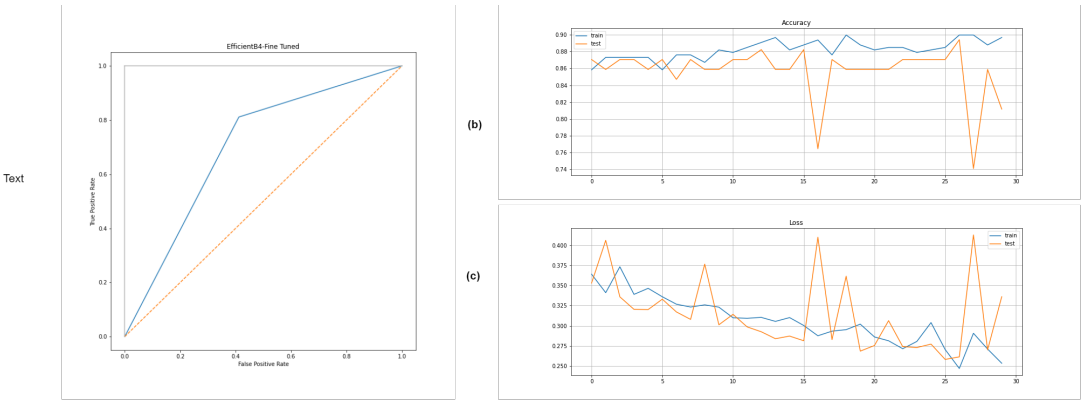

Figure 11: EfficientNetB4: a) AUC/ROC Curve b) Epoch vs Accuracy c) Epoch vs Loss

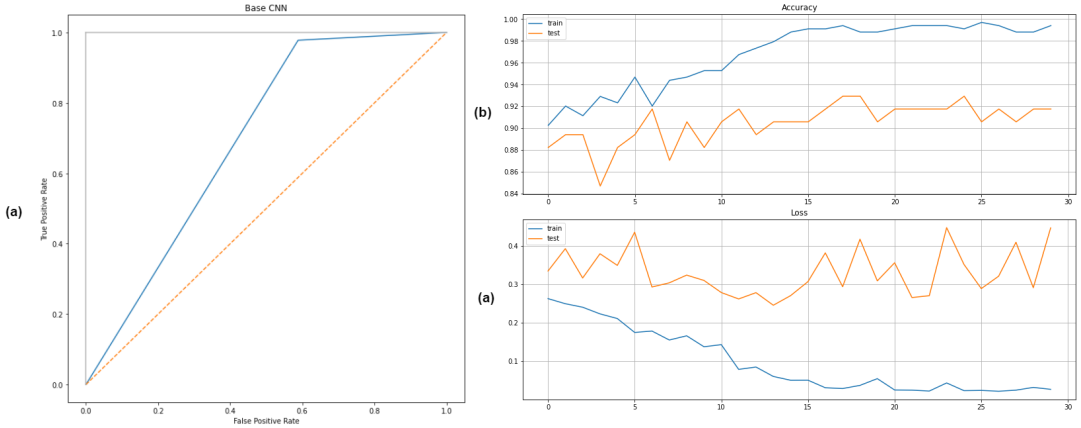

Figure 12: Base CNN: a) AUC/ROC Curve, b) Epoch vs Accuracy, c) Epoch vs Loss

6.2.2 Inference Time Comparison

Inference time in deep learning refers to the total time required for a model to perform one full forward propagation, which results in the production of an output given there is an input. It is a crucial metric for deep learning model optimization.

This inference time can be deduced using two important metrics viz

- **FLOPs :** It refers to the total number of **Floating-Point Operations** performed by a model during one forward pass.This can be estimated by adding up the occurence of all the floating-point arithmetic operations like addition, subtraction, multiplication, and division in the target deep learning model.
- **FLOPS :** It refers to the total number of **Floating-Point Operations** a piece of **hardware** can perform in a **second**.This can be estimated using data such as CPU speed, core count, CPU instruction rate, and number of CPUs per node as explained below:

$$FLOPS = R * S * T * U \tag{1}$$

Where,

- **R =** CPU speed in GHz,
- **S =** Number of CPU cores,
- **T =** CPU Instructions Per Cycle,
- **U =** Number of CPUs per node

Finally **Inference Time** can be calculated as:

$$InferenceTime = FLOPs/FLOPS \tag{2}$$

Using the PIP package namely **"keras-flops"**, the **FLOPs** for all the deep learning models were calculated, which is displayed in the Table 3 below.

Table 3: Inference Time

Models	**FLOPs**
Base CNN (not tuned)	0.251 G
Base CNN (Tuned)	0.504 G
Efficient Net (Not Tuned)	1.56 G
LSTM + Tokenizer	2.06e-05 G
Bi-LSTM + Tokenizer	2.06e-05 G
BI-LSTM + Word2Vec (Tuned)	1.08e-05 G
LSTM + Word2Vec (Tuned)	1.933e-05 G

From the above table following observations can be made:

- The NLP-based methods are the quickest of all the stated solutions. Based on the Inference Time the fastest language model is the fine-tuned **Bi-LSTM** Model using **Word2Vec Embeddings** with **1.08e-05 G** FLOPs.

- Second Fastest solution would be fined tuned **LSTM model using Word2Vec embeddings** with **1.933e-05 G** FLOPs.
- The fastest solution in the image-based approach is the **Base CNN model** with **0.251 G** FLOPs whereas the second fastest is the **fine-tuned Base CNN model** with **0.504 G** FLOPs

It can be clearly observed from the Table 2 and Table 3, that **Inference Time** is directly proportional to the **accuracy** of the deep learning models.

7 Conclusion and Future Work

The metrics above show that the language-based models outperformed the image models in terms of performance. It should be emphasized that the CNN models were trained with additional data from manifest files and external APIs, but the language-based models were trained exclusively on opcode data. Therefore, it is safe to say that the innovative opcode sequence sampling technique used in this research to characterize the applications performed very well. Additionally, it demonstrates how crucial, feature selection from the decompiled APK files is to malware detection. By, generating more samples of the opcode sequences, there is still room to enhance the performance of the language models.In the future, further experiments can be conducted by concatenating together the best models from both the approaches.

References

Almomani, I., Alkhayer, A. and El-Shafai, W. (2022). An Automated Vision-Based Deep Learning Model for Efficient Detection of Android Malware Attacks, *IEEE Access* **10**: 2700–2720. Conference Name: IEEE Access.

Darwaish, A. and Naït-Abdesselam, F. (2020). RGB-based Android Malware Detection and Classification Using Convolutional Neural Network, *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, pp. 1–6. ISSN: 2576-6813.

Darwaish, A., Naït-Abdesselam, F., Titouna, C. and Sattar, S. (2021). Robustness of Image-based Android Malware Detection Under Adversarial Attacks, *ICC 2021 - IEEE International Conference on Communications*, pp. 1–6. ISSN: 1938-1883.

Fan, M., Luo, X., Liu, J., Nong, C., Zheng, Q. and Liu, T. (2020). CTDroid: Leveraging a Corpus of Technical Blogs for Android Malware Analysis, *IEEE Transactions on Reliability* **69**(1): 124–138. Conference Name: IEEE Transactions on Reliability.

Garg, S. and Baliyan, N. (2022). M2VMapper: Malware-to-Vulnerability mapping for Android using text processing, *Expert Systems with Applications* **191**: 116360. **URL:** *https://www.sciencedirect.com/science/article/pii/S0957417421016572*

Huang, T. H.-D. and Kao, H.-Y. (2018). R2-D2: ColoR-inspired Convolutional NeuRal Network (CNN)-based AndroiD Malware Detections, *2018 IEEE International Conference on Big Data (Big Data)*, pp. 2633–2642.

Islam, T., Motiur Rahman, S. S. M., Hasan, M. A., Md. Mostafizur Rahaman, A. S. and Jabiullah, M. I. (2020). Evaluation of N-Gram Based Multi-Layer Approach to Detect Malware in Android, *Procedia Computer Science* **171**: 1074–1082.
URL: *https://www.sciencedirect.com/science/article/pii/S1877050920310930*

Lakshmanarao, A. and Shashi, M. (2022). Android Malware Detection with Deep Learning using RNN from Opcode Sequences, *International Journal of Interactive Mobile Technologies (iJIM)* **16**(01): 145–157. Number: 01.
URL: *https://online-journals.org/index.php/i-jim/article/view/26433*

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A. and Talwalkar, A. (2017). Hyperband: a novel bandit-based approach to hyperparameter optimization, *The Journal of Machine Learning Research* **18**(1): 6765–6816.

Li, X., Zhao, Z., Tang, Y., Zhang, J., Wu, C. and Li, Y. (2022). An Android Malicious Application Detection Method with Decision Mechanism in the Operating Environment of Blockchain, *Security and Communication Networks* **2022**: e3111540. Publisher: Hindawi.
URL: *https://www.hindawi.com/journals/scn/2022/3111540/*

Oak, R., Du, M., Yan, D., Takawale, H. and Amit, I. (2019). Malware Detection on Highly Imbalanced Data through Sequence Modeling, *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security - AISec'19*, ACM Press, London, United Kingdom, pp. 37–48.
URL: *http://dl.acm.org/citation.cfm?doid=3338501.3357374*

QiuJunyang, ZhangJun, LuoWei, PanLei, NepalSurya and XiangYang (2020). A Survey of Android Malware Detection with Deep Neural Models, *ACM Computing Surveys (CSUR)* . Publisher: ACM PUB27 New York, NY, USA.
URL: *https://dl.acm.org/doi/abs/10.1145/3417978*

Raghav, U., Martinez-Marroquin, E. and Ma, W. (2021). Static Analysis for Android Malware detection with Document Vectors, *2021 International Conference on Data Mining Workshops (ICDMW)*, pp. 805–812. ISSN: 2375-9259.

Taheri, L., Kadir, A. F. A. and Lashkari, A. H. (2019). Extensible Android Malware Detection and Family Classification Using Network-Flows and API-Calls, *2019 International Carnahan Conference on Security Technology (ICCST)*, pp. 1–8. ISSN: 2153-0742.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, and Polosukhin, I. (2017). Attention is All you Need, *Advances in Neural Information Processing Systems*, Vol. 30, Curran Associates, Inc.
URL: *https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html*

Wang, Z., Li, G., Zhuo, Z., Ren, X., Lin, Y. and Gu, J. (2022). A Deep Learning Method for Android Application Classification Using Semantic Features, *Security and Communication Networks* **2022**: e1289175. Publisher: Hindawi.
URL: *https://www.hindawi.com/journals/scn/2022/1289175/*

Yeboah, P. N. and Baz Musah, H. B. (2022). NLP Technique for Malware Detection Using 1D CNN Fusion Model, *Security and Communication Networks* **2022**: e2957203. Publisher: Hindawi.
URL: *https://www.hindawi.com/journals/scn/2022/2957203/*

Zhang, N., Tan, Y.-a., Yang, C. and Li, Y. (2021). Deep learning feature exploration for Android malware detection, *Applied Soft Computing* **102**: 107069.
URL: *https://www.sciencedirect.com/science/article/pii/S1568494620310073*

Zhang, W., Luktarhan, N., Ding, C. and Lu, B. (2021). Android Malware Detection Using TCN with Bytecode Image, *Symmetry* **13**(7): 1107. Number: 7 Publisher: Multidisciplinary Digital Publishing Institute.
URL: *https://www.mdpi.com/2073-8994/13/7/1107*