ANDROID MALWARE DETECTION USING STATIC MALWARE ANALYSIS AND NATURAL LANGUAGE PROCESSING

A DISSERTATION

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE AWARD OF THE DEGREE

OF

MASTER OF TECHNOLOGY

IN

INFORMATION SYSTEMS

Submitted by:

HIMANSHU VERMA

2K21/ISY/09

Under the supervision of

Rahul Gupta

Assistant Professor

DEPARTMENT OF INFORMATION TECHNOLOGY

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-110042

MAY 2023

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Bawana Road, Delhi-110042

CANDIDATE'S DECLARATION

I, Himanshu Verma, 2k21/ISY/09 of M.Tech (Information System), hereby declare that the Dissertation titled "Android Malware Detection using Static Malware Analysis and Natural Language Processing" which is submitted by me to the department of Information System, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology, is the original and not copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma Associateship, Fellowship or other similar title or recognition.

Place: Delhi Himanshu Verma

Date: 31-05-2023

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Bawana Road, Delhi-110042

CERTIFICATE

I, Himanshu Verma, 2k21/ISY/09 of M.Tech (Information System), hereby declare that the dissertion titled “Android Malware Detection using Static Malware Analysis and Natural Language Processing” which is submitted by me to the department of Information System, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology, is the original and not copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma Associateship, Fellowship or other similar title or recognition.

Place: Delhi

Date: 31-05-2023

RAHUL GUPTA

SUPERVISOR

ASSISTANT PROFESSOR

ACKNOWLEDGEMENT

I want to thank my Mentor, Mr Rahul Gupta, for his constant support, patience, and trust in me, as well as for providing a good environment and giving helpful comments to me to research in this area. It has been a totally new area to work upon, but sir has generously supported us to research this area.

HIMANSHU VERMA

ABSTRACT

Android malware detection represents a current and complex problem, where black hats use different methods to infect users' devices. One of these methods consists in directly uploading malicious applications to app stores, whose filters are not always successful at detecting malware, entrusting the final user the decision of whether installing or not an application. Although there exist different solutions for analysing and detecting Android malware, these systems are far from being sufficiently precise, requiring the use of third-party antivirus software which is not always simple to use and practical. We propose a novel method for analysing and detecting malicious Android applications by employing meta-information available on the app store website and also in the Android Manifest. Its main objective is to provide a fast but also accurate tool able to assist users to avoid their devices becoming infected. The method is mainly based on a text mining process that is used to extract significant information from meta-data, that later is used to build efficient and accurate classifiers.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 PREFACE

In modern time the number of people owning and using mobile device has increased exponentially and it is set to increase furthermore because of upcoming technological advancement in this field. Mobile devices enable us to conveniently access many online services with ease like online banking, Commuinication via voice or video call, accessing video content, etc. But this penetration of technology comes with onset of getting attacked by a malware. Programs and file stored within the device itself are attacked by the malware aka the malicious software. As per the technique used by this malicious software, malware is of different type like spyware, and adware, trojans, worms, rootkit, spyware and backdoor. Spyware as the name suggest are used by attacker for surveillance and snooping purposes. Adware on other hand provides with inquisitive and unwanted unnecessary ads to the user.

Since android operating system has highest penetration thus they are hotspot of attack by the malware. Malware frequently starts off by tricking users into opening bogus messaging. Users who show interest in these communications are then charged for fictitious services. Less risks often affect other systems. A virus is a piece of software that can replicate by itself and can infect programs by altering the host or its abode, and it can spread quickly through network propagation. A worm is a piece of self-replicating software which have ability expand over a network.; generally, completely self-contained are created which may propagate on their own. Thus, Android Malware analysis and detection is area of research due to alarming rise of amount malicious apps of android. In the literature, several ways for detecting and categorising Android malware have been created and evaluated.

We can use static or dynamic analysis to collect the characteristics utilised while evaluating malware. We can also use a combination of both. The phrase static analysis refers to information acquired without the code being run. On the other hand, dynamic analysis entails executing (or simulating) the code. Though dynamic analysis has the potential to be more

comprehensive and revealing and is regarded to be less vulnerable to code obfuscation but other is faster.

While it has analysis has several drawbacks especially in contrast to dynamic malware analysis, such as the difficulty in combating code obfuscation, it also offers unique benefits:

- Complete Code Coverage: By analysing code or metaphorical pseudo execution can encompass all code and all resource. Dynamic analysis is unlikely to encompass the whole code. Many apps ask users to give login credentials in order to utilise many of the features. This makes it hard to identify in dynamic analysis, functions fail to extract enough features, resulting in inadequate feature extraction.

- More Efficient: It can accomplish discovery process within predicted since wouldn't require the app to be running state. Dynamic analysis, on the other hand, necessitates invoking multiple functions during code execution, which takes a long time. It takes some time for the software to replicate the real thing. While executing software may do extremely complicated operations or enter an endless loop. Due to these circumstances, completing the detection process within the stipulated time range is challenging.

- Abstracted from malware: Because this analysis does not need the execution of the programmes, rogue apps are unaware of fact of being monitored. Even though certain malicious app attempts to make it more difficult via adding interfering programs, these additional programs might be utilised as a marker to aid in the detection of malicious software.

- Facile Identification: The extraction of characteristics with non-linearities and ubiquity is more likely with malicious sample analysis. Dynamic analysis, on the other hand, is extremely expected to be influenced due to operational conditions. It is possible to employ statically derived features for identification and to detect a wide range of hostile code fast.

Analysing different characteristics of Android applications to find probable malware indicators is necessary when trying to detect Android malware using meta-information. Data like the app's 1 1 1 1 1 7 permissions, certificate details, package name, desired features, and more are all included in meta-information. Android applications are signed with digital certificates to prove their legitimacy and integrity, according to a certificate analysis. Apps issued with shady or selfsigned certificates, which may signal malicious intent, can be found by analysing certificate information. An app's package name contains details about the creator and intended use of the programme. Applications with false or suspicious names that can be connected to malware can be found by looking at the package name. Android apps may ask for particular hardware or software characteristics from the device, according to a feature analysis. When requested features are analysed, it becomes easier to spot programmes that ask for unusual or superfluous characteristics, which might be a sign of malicious behaviour. While not precisely classified as meta-information, examining the app's actual code and API calls can provide us important information on how the app behaves. In order to find any harmful or suspicious activity, this stage uses techniques like dynamic analysis or reverse engineering. By using machine learning algorithms on the gathered meta-data, it is possible to create models that automatically categorise programmes as benign or malicious. Using a labelled dataset of well-known malware and innocuous programmes, a model is trained to discover patterns and attributes typical of malware. The detection model may be combined with security tools like antivirus software or mobile device management (MDM) programmes to offer real-time malware detection and prevention after it has been created.To enable real-time malware detection and prevention, the created detection models may be included into security solutions like antivirus software or MDM systems.The detection model analyses the meta-data associated with an app's installation so that appropriate actions, such as banning installation or marking the programme as possibly hazardous, may be implemented. It's crucial to keep up with the most recent malware trends and to constantly modify the detection strategies to tackle new and developing threats. The efficiency of Android malware detection utilising meta-information may also be increased by working with security researchers and utilising community-driven threat intelligence. Malware creators are constantly improving their methods of avoiding detection. They could employ polymorphism, encryption, or obfuscation techniques to hide their virus from detection by meta-data alone. Malware detection frequently results in both false positives and false negatives. To maintain user confidence and usefulness, it's essential to strike a balance between detecting accuracy and reducing false alarms.The whole range of virus behaviour might not be captured by metainformation analysis alone. A more thorough defence against

Android malware may be achieved by combining it with additional detection techniques including static and dynamic 8 analysis, network traffic monitoring, and anomaly detection. Android malware is a neverending task and new threats appear often. Using threat information feeds, different detection approaches, and keeping up with the most recent malware trends may all assist to increase efficiency of utilising meta-data to find Android malware.

Instead of undertaking software analysis (which complicates and slows the analysis), the project seeks to contemporary and enhanced offering by studying meta-information. Because of the vast and representative set of features, it is possible to create an easy-to-use technology while maintaining high accuracy. This methodology can also be used to feed recommender systems, which can assist users in making decisions. In order to increase the categorization quality, we've additionally used supervised learning for this strategy. Most used approach for detecting malware are Anomaly and Signature-based. The signature-based approach relies on recognising the signature of the malware behaviour. In contrast, the anomaly-based approach uses its knowledge to compare the normal and abnormal behaviours of a system. Programmes that deviate from the specifications are assessed as anomalous and, usually, as malware. We will be using anomaly-based detection approach. A text mining evaluation where various meta information to build classifier is present which help user to take final decision is also present.

1.2 LITERATURE REVIEW

Machine Learning based approach automates android malware's analysis and detection via spotting out malware patterns. High success rate of picking out malware can be achieved from it. Use of dataset is fundamental in this specific application of artificial intelligence for making future decision and predcit output. It refers to the process of characterising malware behaviour and applying classifiers to evaluate the dataset [1]. In previous studies, the classifiers most frequently used to assess the necessary features and malware detection include Naive Bayes, support vector machines, decision trees, Random Forest, K-means, K-nearest Neighbours AdaBoosting, logistic regression, and J48 [2]. Random Forest aka hybrid method incorporates trump card of incorporating different tree classifiers. Mechanism invoved here uses multiple decision tree which are trained. Each decision tree is assigned different weightage. Now the decision is taken combing results from all these decision tree. Sanz [3] used random forest and developed a method that extracts several features from the manifest file to build machine

learning classifiers. Permissions used by the application are sued as feature set. Input vector was made using application permission. After that experiment was enacted using various classifiers like J48, Naive Bias, Random Forest, and other classifiers to perform experiments. Out of all the classifiers best result was obtained via Random Forest. It had a Accuracy and AUC of 94.83% and 98% respectively. Tiwari and Shukla [4] proposed a method to detect android malware using permissions and API. A four-step approach was used by the author. First step was reverse engineering done for second step of feature extraction. Based on extracted feature, feature vector was generation which was their third step and finally classification was carried out. Reverse engineering tools were used on apk to get Smali and AndroidManifest. Smali Files have APIs AndroidManifest.xml helped to obtain permissions used. XGBoost [5] is a decision tree ensemble based on gradient boosting designed to be highly scalable. XGBoost reduces loss function for additive expansion of objective function. For accelerating training speed and reducing overfitting randomizations were added. XGBoost implements several methods to increment the training speed of decision trees not directly related to ensemble accuracy.[6]Method used by author for the similarity function just returns a value between 0–1 measuring the strength of similarity between two documents making it more or less like a black box uninterpretable similarity. Though it might limit the performance of applications like recommender systems that mostly rely on document similarity as their base. [7]Process of picking out critical words pertinent to given document is termed as keyword extraction. It enables quicker search over document as in above process they are indexed as document alias. Nature of cateogarization of text can be extractive or abstractice i.e corresponding keywords are from outside of relevant text or are present there itself. Authors work on Swisscom AG on extractive keyphrase extraction in an example of unsupervised way. Author's specifically chose unsupervised way because of the advantages and flexibility it delivers over supervised methods such as out-of-domain generalization, no demand for large hand-annotated corpus with keywords, etc. Many present-day extraction systems have shortcoming either because of their slower speed or with respect of creating keyword thath ar relevant and are not redundant. Here, in this paper author's address both these problems with their proposed unsupervised algorithm. [8]A deliberate attempt was made to use sensitive API calls and permissions to discover and detect malware. The test produced interesting findings, showing that these particular traits demonstrated a high level of success in identifying Android malware. This finding shows that using sensitive API calls and permissions can be a reliable way to spot and remove malicious software, which is very encouraging for the field of malware detection on the Android platform. [9]The author presented two unique methods for extracting

keyphrases from a single document. These techniques make use of sentence embeddings and provide easy scaling of the extraction process.EmbedRank and EmbedRank++, in contrast to conventional methods, are unsupervised and do not rely on a particular corpus or outside training data. They are only reliant on the document's content, making them helpful even in the absence of a large corpus. These approaches deviate from traditional methods by using text embeddings rather than graph representations. They successfully capture the document's informativeness and diversity by utilising the power of sentence embeddings. These innovative techniques enable effective extraction of keyphrases from individual documents while considering the informational value and diversity of sentences. [10]The researchers unveiled DroidFusion, a state-of-the-art fusion method that integrates various machine learning (ML) approaches to improve malware detection accuracy. To capture various features of malware detection, the approach entails applying ML algorithms to train a variety of classifiers. The predicted accuracy is then evaluated, and the most useful features are chosen for the final classifier using a feature ranking algorithm. DroidFusion is more accurate than the commonly utilised stacking ensemble approach, according to experimental data, proving its efficacy in identifying malware. This development in malware detection demonstrates DroidFusion's potential to increase the precision and dependability of malware detection systems and demonstrates its superiority to current approaches. [11] The usefulness of intentions in identifying dangerous apps was studied by the researchers. In contrast to permissions, they discovered that intentions are more important for the categorization of malware. According to the findings of their investigation, 91% of intentions were detected, compared to 83% of permits. The scientists also noted that combining the purpose and permission features produced a detection accuracy of 95.5%, which was greater than the accuracy obtained by utilising each feature alone. [12] For detecting malware, the authors suggested a mechanism called Significant Permission Identification (SigPID). The three-level pruning method used by SigPID includes gathering permission data in order to find pertinent permissions that can reliably distinguish between malicious and good programmes. The authors used methods for machine learning (ML) to categorise Android applications.. [13] To evaluate the risk connected with permissions in Android apps, the authors conducted a research. To rank certain licences according to their degree of danger, they used statistical approaches including the T-test, correlation coefficient, and mutual information. To find the permission subsets that were considered dangerous, sequential forward selection and principal component analysis were used. The authors used the Decision Tree (DT), Support Vector Machine (SVM), and Random Forest (RF) algorithms to assess how well these hazardous permissions worked in identifying

dangerous programmes. With a False Positive Rate (FPR) of 0.6, the findings showed that the Malapp detector had a detection accuracy of 94.62%. This demonstrates the detector's effectiveness and accuracy in detecting malicious applications based on the chosen dangerous permissions.

CHAPTER 2

METHODOLOGY

2.1 ANDROID MALWARE DETECTION USING META-INFORMATION

We have built a novel android malware detection system which rely on meta information extracted from the manifest file and application description. The objective of this project is to provide user with a dependable and quick android malware classifier.

Our method offers android malware analysis using anomaly detection in which android applications are analysed with executing them rather by extracting meta information, in our case we have extracted meta information from the android manifest file. Meta information extracted is quite essential and is key to identify the malicious app. There are 182 features of android manifest file present in dataset like rating_number, description, devoloper name, country, etc and permissions like bluetooth, read_logs, Min_SDK, Read_External_Storage, etc .

In the first step we will perform data preprocessing. Dataset has metadata of 11476 applications and have 182 features i.e meta information from android manifest file. Out of all the available applications 8058 applications are benigware applications and 3418 applications are malicious applications. First of all in preprocessing we will look for features(columns) with missing values. We will look for relevance of that features in analysing malware and will drop it if that feature is not relevant or else if it is essential then we will work for correcting redundant entries. For example, we found columns MD5, Min_SDK, Locality, etc columns with redundant values. Now MD 5 algorithm is a crpytography algorithm used to protect password. This column contains hash values for the same. So we can drop this column. Min_SDK column tells the minimum android version required too run the app. Generally higher the Min_SDK version the better it is. Here we will place 0 in place of unknown values and will dop NaNvalues. SO in this way we will deal with missing values in dataset.

Now after checking and acting upon missing values we will analyse data we have got. Upon analysing we find that we have 166 numerical features. Out of these 160 are binary features and 6 are continuous features. We will look for outliers in continuous variables. We will have to remove outliers from the continuous variables before inputting to training model. We will use turkey's rule to eliminate outliers. It says that outliers are those values which are out of bound of 1.5 times the inter quartile range. And for boolean columns, we will look for variance. We will remove boolean columns with low variance.

2.2 NATURAL LANGUAGE PROCESSING

Now to work on NLP Data we will have to preprocess our Text. We will remove non Latin characters as there are application description available in other language too. After data cleanup for classifier, we will initiate vectorizer aka design pipeline. Will perform TF IDF Vectorization and will tune parameters using Grid Search Cross Validation. After that we will analyse performance of different classifiers.
The method used is based on the traditional term-document matrix, in which each description is treated as a document, and a set of words is retrieved from the whole corpus of descriptions. These words are picked via a cleaning procedure that gets rid of irrelevant terms and harmonises those with shared etymologies.

The description or metadata of Android applications may be examined using NLP to spot possibly shady or harmful behaviour. NLP models can assist in identifying suspicious applications for additional examination by analysing the text and extracting pertinent characteristics, such as the usage of certain keywords, peculiar language patterns, or deceptive information. Approaches should be used in combination with other malware investigation techniques even if they might offer useful insights into the programme description. From the application description, essential words or phrases may be extracted using NLP algorithms. These keywords can be used to spot potentially hazardous programmes by comparing them to a list of known malicious or suspicious phrases. Words such as "free," "unlimited," for instance, may denote a programme engaged in illicit activity. Network traffic monitoring, static or dynamic code analysis, and meta-information analysis may all be used with NLP to improve the precision and efficiency of Android malware investigation.

2.3 CLASSIFICATION

Differentiating between malware (malicious programmes) and benign-ware (legal applications) utilising patterns found in their authorization policies and instructions is the classification procedure for Android malware analysis using NLP. The objective is to teach a classifier to recognise the essential distinctions between these two categories of applications. A labelled dataset is necessary to begin the classification process, in which each programme is assigned to one of two categories: malware or benign-ware. The authorization guidelines and instructions connected to each application are included in the dataset. The language of the authorization regulations and instructions is then processed and analysed using NLP methods. Tokenization, stopping words removal, stemming or lemmatization, and extracting pertinent characteristics are a few of the processes involved in this. Machine learning techniques are used to train the classifier once the dataset has been preprocessed and the features have been retrieved. Classifiers like Random Forest, XGBoost, and Logistic Regression are frequently utilised in this situation. These classifiers are trained on a labelled dataset, with the goal variable being the associated labels (malware or benign-ware), and the input variables being the characteristics gleaned from the authorization regulations and instructions. The classifier gains the ability to identify patterns and connections in the data that differentiate between malicious and benign software during the training phase. It creates a model that can extrapolate these trends to forecast outcomes for novel, unforeseen uses.

2.4 DATASET

The dataset includes various information such as the application's description, developer, rating score, and other meta-information. However, it was observed that a significant number of samples in the dataset did not have a description. As a result, those applications were removed from the dataset, leading to a decrease in its size. After the initial data collection, data pre-processing techniques were applied to prepare the dataset for further analysis. Data preprocessing involves transforming and cleaning the data to make it suitable for applying classification techniques. In this case, the dataset consists of 11478 rows (representing individual applications) and 183 columns (features). Among the features, 166 are numerical, out of which 160 are binary (taking values of 0 or 1) and 6 are continuous (taking continuous numeric values). To improve the performance of the classification techniques, it is important 1

11 to consider the variance of each feature. Features with low variance may not provide much discriminatory information and can potentially hinder the performance of the classifiers. Therefore, it is common practice to remove features with low variance to reduce the dimensionality of the dataset. By examining the variance of each feature, it is possible to identify those features with low variance and remove them from the dataset. This helps to streamline the dataset and focus on the most informative features that contribute significantly to the classification task.

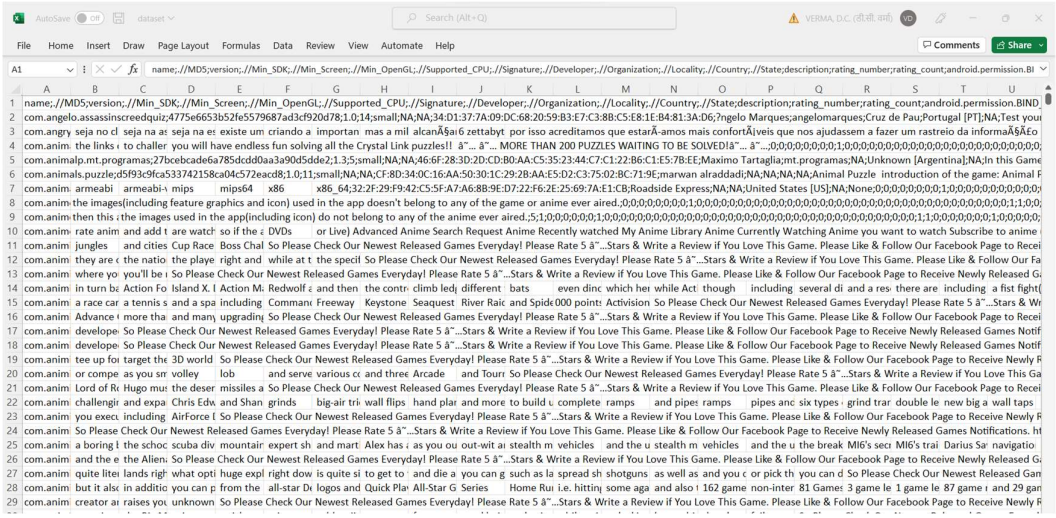

Figure 1: Dataset Snapshot

2.5 TEXT MINING

Text mining is the practise of analysing textual data related with Android applications, such as their descriptions, user reviews, permissions, and other metadata, using natural language processing (NLP) techniques. Text preprocessing refers to the steps are taken to clean and normalise the data before the text is analysed. In order to get rid of stopwords (common words like "the," "is," etc.) and get terms down to their simplest forms, this may entail eliminating punctuation, changing the case to lowercase, managing stopwords, and stemming or lemmatization. In order to extract useful information from the textual material, pertinent characteristics must first be retrieved. N-grams (word sequences), bag-of-words encoding, and

term frequency-inverse document frequency (TF-IDF) are a few examples of these properties. Analysis of User Reviews: User reviews offer insightful information on the usability and enjoyment of an application. Text mining may aid extract sentiment analysis, which identifies whether users expressed positive or negative feelings. Specific complaints or unfavourable sentiment patterns in the reviews may be signs of malware or other questionable activity. Classification and Detection: Machine learning methods may be trained on labelled datasets to categorise programmes as benign or malicious based on their textual properties. Examples of these algorithms are decision trees, random forests, and support vector machines. By combining textual data with additional variables, text mining can improve the precision of virus detection algorithms. When analysing Android malware, text mining offers a complimentary method to static analysis and dynamic analysis. It makes use of the rich textual data connected with Android applications to derive insights, spot trends, and raise the precision of models for malware detection and categorization.

2.6 ANDROID PERMISSIONS

Android permissions are essential for understanding Android malware since they reveal what functions and activities an application may carry out on a user's device. Android applications ask for permission to use particular resources or carry out particular tasks on a user's device. One can learn more about the application's intended functionality by looking at the permissions that are being asked for. The application's AndroidManifest.xml file contains declarations of permissions.To Spot Malicious Behaviour, a malicious programmes frequently ask for too many rights, beyond what is necessary for their intended use. A torch app that requests access to the user's contacts or SMS messages, for instance, may be a warning sign. An application's requests for permissions might reveal questionable or potentially harmful behaviour. Misuse of Permissions as Authors of malware may take advantage of valid permissions to engage in bad behaviour. They could take advantage of certain rights to gather private user data, monitor user behaviour, place unauthorised calls or send unauthorised messages. Detecting such misuse involves looking at the permissions that are sought. Vulnerabilities Caused by Permissions occurs if Some permissions, if misused, might leave an application open to security flaws. The requested permissions can be examined to find any security flaws or hazards that malware or attackers could try to exploit.

Android permissions may be divided into many areas according on how they work, such as access to the camera, location, contacts, storage, and network. the authorization groups are

examined. Analyzing the permission groups can provide insights into the scope of access an application requires. Mapping permissions to their corresponding functionalities helps in understanding the potential impact on user privacy and security. Permission-Based Access Control of android's permission system acts as a security mechanism to control access to sensitive resources and user data. By analyzing an application's requested permissions, one can assess whether the requested permissions are necessary for its intended functionality. Applications that request excessive or unnecessary permissions may be flagged for further investigation. Detection of Permission-Evasive Techniques is required as some malware may employ techniques to avoid requesting permissions explicitly. For example, they may dynamically download additional code or modules after installation to bypass the scrutiny of the initial permission requests. Such evasive tactics can be found by analysing permission-related patterns and behaviours. Permissions may be utilised as characteristics in machine learning-based classification models for detecting malware on Android devices. Classifiers can learn to distinguish between benign and dangerous programmes based on their permission requests by training models on labelled datasets with permission characteristics. In conclusion, Android permissions offer useful data for analysing Android malware. Understanding access control, spotting potentially harmful behaviour, detecting permission abuse or vulnerabilities, and improving the precision of malware detection models all benefit from analysing the requested permissions.

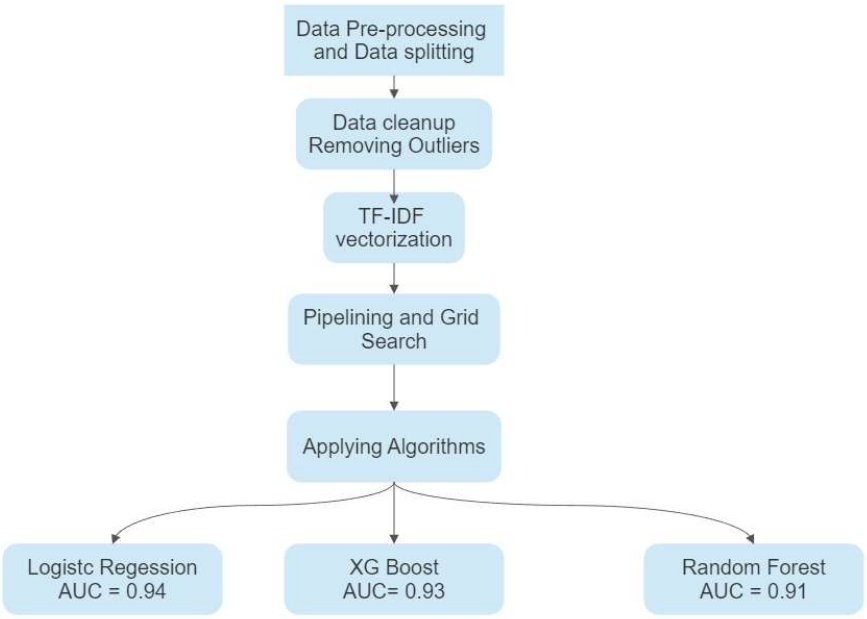

Figure 2: Workflow Diagram

CHAPTER 3

IMPLEMENTATION

Firstly, we are required to pre-process data so that we can feed them to our machine learning algorithms and for natural language processing. We will split up our data into training and testing data. We will look up at training data so as to get insight into meaning of different features. Then we will try to uncover trends with the plots of data. We will inspect for missing and erroneous entries in our dataset. We will drop NaN values, Replace unknown values with 0 and will drop attributes with too many unique values since then they will not help much in classification. Min_SDK' - Verdict: Drop NaN Values, repalce "Unknow" with 0 and convert to int. Min_Screen- Verdict: Drop the NaN and replace values with 0,1,2...,
Min_OpenGL Verdict: Convert NaN to 0, keep values as is, Supported_CPU- Drop the feature, no idea how to make sense of it, Signature - Drop it Too many unique values to turn in into useful numerical feature, Developer- Drop it Too many unique values to turn in into useful numerical feature, Organization Verdict: Drop it Too many unique values to turn in into useful numerical feature Country Verdict: Drop it State, Verdict: Drop it, Entirely empty, LocalityVerdict: Drop it.

```
X_train['.//Min_Screen'].isna().mean()
```

```
0.0003872966692486445
```

```
X_train['.//Min_Screen'].isna().sum()
```

```
4
```

```
X_train['.//Min_Screen'].value_counts()
```

```
small     9921
normal     381
large       19
xlarge       3
Name: .//Min_Screen, dtype: int64
```

```
X_train['.//Min_OpenGL'].isna().mean()
```

```
0.7328621223857474
```

```
X_train['.//Min_OpenGL'].value_counts()
```

```
2.0    2670
```

Figure 3: Attributes with NaN

```
def minSDK(df_nan):
    df = df_nan.copy(deep=True)
    df['.//Min_SDK'] = df['.//Min_SDK'].str.replace('Unknown', '0', regex=False)
    df = df[df['.//Min_SDK'].notna()]
    df['.//Min_SDK'] = df['.//Min_SDK'].astype(int)

    return df
```

```
def minScreen(df_nan):
    df = df_nan.copy(deep=True)
    df = df[df['.//Min_Screen'].notna()]

    #create mapping
    mapping = {'small': 0, 'normal': 1, 'large': 2, 'xlarge': 3}

    df['.//Min_Screen'] = df['.//Min_Screen'].replace(mapping)

    return df
```

```
def minOpenGL(df_nan):
    df = df_nan.copy(deep=True)
    #fill nan with zeroes
    df['.//Min_OpenGL'] = df['.//Min_OpenGL'].fillna(0)
    return df
```

```
def NaNCleanup(df_nan):
    df = df_nan.copy(deep=True)
    df = minSDK(df)
    df = minScreen(df)
```

Figure 4: Applying Clean up

Now we will look out for outliers, data distribution, columns with useless information and their datatypes. We will look for column with constant value, binary and continuous values. For continuous variables, we note outliers. We need to exclude those for our training model. Most apps have no rating and rating counts. Will have to eliminate outliers to make values sensible. Outlier detection applies only on TRAINING data. Tukey's rule says that the outliers are values more than 1.5 times the interquartile range from the quartiles — either below Q1 − 1.5IQR, or above Q3 + 1.5IQR.

```
def getOutlierIndeces(df, col, multiplier = 1.5):
    #get Q1 and Q3
    q1, q3 = df[col].quantile([0.25, 0.75],
                              interpolation='nearest')
    # calculate IQR
    iqr = q3 - q1
    #calulate lower and upper boundaries
    bound_lower = q1 - multiplier*iqr
    bound_upper = q3 + multiplier*iqr

    #return indeces of outliers outside the boundaries
    df_outlier = df.copy(deep=True)
    df_outlier = df_outlier[(df_outlier[col]>bound_upper) | (df_outlier[col]>bound_upper)]

    return df_outlier.index
```

```
#create a set since we might have repeating rows
outlier_set = set()
for col in df_continuous.columns:
    outlier_rows = getOutlierIndeces(df_continuous, col)
    outlier_set.update(outlier_rows)


print(f"Flagged {len(outlier_set)} rows in training data as outliers")
```

```
Flagged 3431 rows in training data as outliers
```

Figure 5: Finding Outliers

Now we will work on NLP columns. Will check if we can I understand what a feature is about by skimming the entries, does it use the Latin alphabet and is it all in the same language, what's the pre-processing functions needed, What's the most compact and representative way to transform the text. We will perform TDIF vectorization.

```
def keepLatinChars(text):
    #encode to ASCII
    encoded_text = text.encode(encoding="ascii",
                               errors="ignore")
    #decode from bytes to utf
    return encoded_text.decode()
```

```
df['description'] = df['description'].apply(keepLatinChars)
```

```
#instantiate vectorizer
vectorizer = TfidfVectorizer(encoding='utf-8',
                             decode_error='strict',
                             strip_accents='ascii',
                             lowercase=True,
                             analyzer='word',
                             stop_words='english',
                             token_pattern=r'(?u)\b\w\w+\b',
                             ngram_range=(1,3),
                             max_df=0.5,
                             min_df=0.05,
                             max_features=50,
                             use_idf=True,
                             smooth_idf=True )
```

```
vectorizer.fit_transform(df['description'])
```

```
<10324x50 sparse matrix of type '<class 'numpy.float64'>'
        with 78414 stored elements in Compressed Sparse Row format>
```

Figure 6: Vectorization

After applying clean up to training and testing data, we will first apply XG Boost. We will initiate pipeline design and subsequently initialise grid search.

```
#instantiate vectorizer
vectorizer = TfidfVectorizer(encoding='utf-8',
                             decode_error='strict',
                             strip_accents='ascii',
                             lowercase=True,
                             analyzer='word',
                             stop_words='english',
                             token_pattern=r'(?u)\b\w\w+\b',
                             ngram_range=(1,3),
                             #max_df=0.5,
                             #min_df=0.05,
                             #max_features=50,
                             use_idf=True,
                             smooth_idf=True )
```

```
scale_pos_weight = (np.sum(y_train_clean == 0)) / np.sum(y_train_clean == 1)

scale_pos_weight = scale_pos_weight[0]
scale_pos_weight
```

```
3.1585639491398654
```

```
#define preprocessor
preprocessor = ColumnTransformer([('tfidfvect',
                          vectorizer,
                          'description')
                        ],
                        remainder=MinMaxScaler(),
                        n_jobs=-1
                       )

#define pipeline
pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                     ('clf', xgb.XGBClassifier(n_estimators=100,
                                               scale_pos_weight = scale_pos_weight,
                                               eta=0.9,
                                               num_boost_round=15,
                                              )
                     )
                         ])
pipeline
```

```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(n_jobs=-1, remainder=MinMaxScaler(),
                                   transformers=[('tfidfvect',
                                                  TfidfVectorizer(ngram_range=(1,
                                                                               3),
                                                                  stop_words='english',
```

Figure 7: Pipeline Designing

We will compare and try to find best classifier among XGBoost, Logistic Regression and Random Forest (Trees). Text data tends to be sparse and can result in a high number of features. Features are related, therefore a classification method like Naive Bayes does not seem like a good fit here.

```
fig = plt.figure(figsize=(8,8))
ax = fig.gca()


roc_xgb_train = plot_roc_curve(grid_search_xgb,
                        X_train,
                        y_train,
                        name='Train', ax=ax)
roc_xgb_test = plot_roc_curve(grid_search_xgb,
                        X_test,
                        y_test,
                        name='Test', ax=ax)
plt.title("ROC Curve For XGB: Train Vs Test")
plt.show()
```

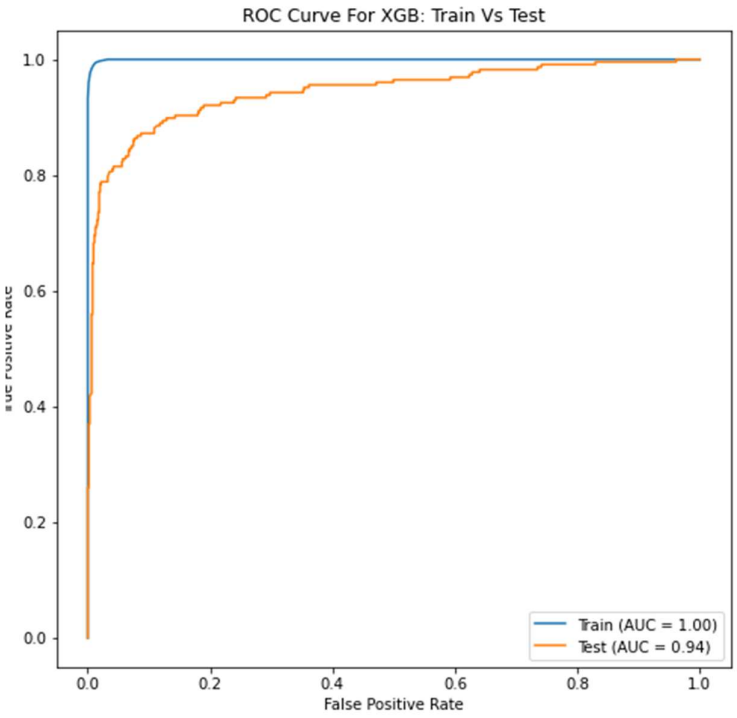

Figure 8: XG BOOST

```
fig = plt.figure(figsize=(8,8))
ax = fig.gca()


roc_lr_train = plot_roc_curve(grid_search_lr,
                              X_train,
                              y_train,
                              name='Train', ax=ax)
roc_lr_test = plot_roc_curve(grid_search_lr,
                              X_test,
                              y_test,
                              name='Test', ax=ax)
plt.title("ROC Curve For LR: Train Vs Test")
plt.show()
```

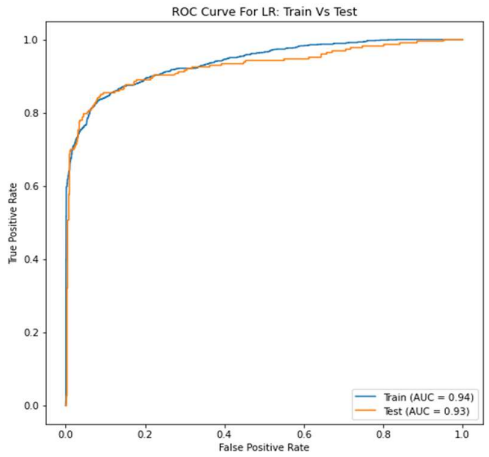

Figure 9: Logistic Regression

```
fig = plt.figure(figsize=(8,8))
ax = fig.gca()


roc_rf_train = plot_roc_curve(grid_search_rf,
                              X_train,
                              y_train,
                              name='Train', ax=ax)
roc_rf_test = plot_roc_curve(grid_search_rf,
                              X_test,
                              y_test,
                              name='Test', ax=ax)
plt.title("ROC Curve For Random Forest: Train Vs Test")
plt.show()
```

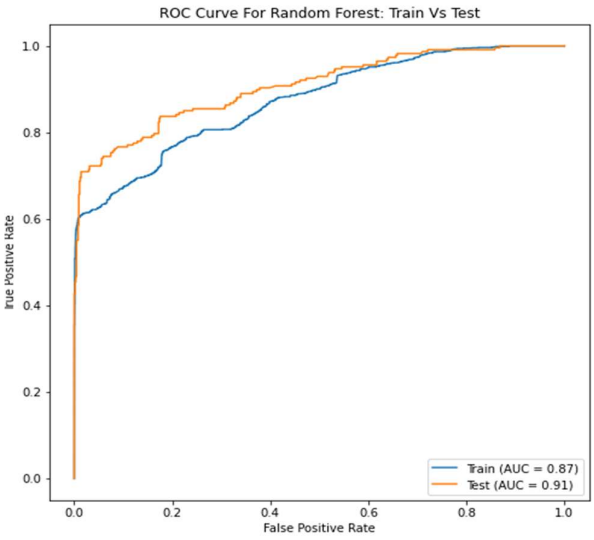

Figure 10: Random Forest

CHAPTER 4

RESULTS

In our analysis, we achieved a noteworthy Area Under the Curve (AUC) value of 0.94 using logistic regression. What's particularly noteworthy is that the weights for the model were primarily derived from the numerical features extracted from the Android manifest. This indicates that the information contained in the text data had limited influence on the analysis, as can be observed from the graph presented below. The emphasis is on the fact that the numerical features played a more significant role in the model's predictive capabilities, leading to the high AUC score.

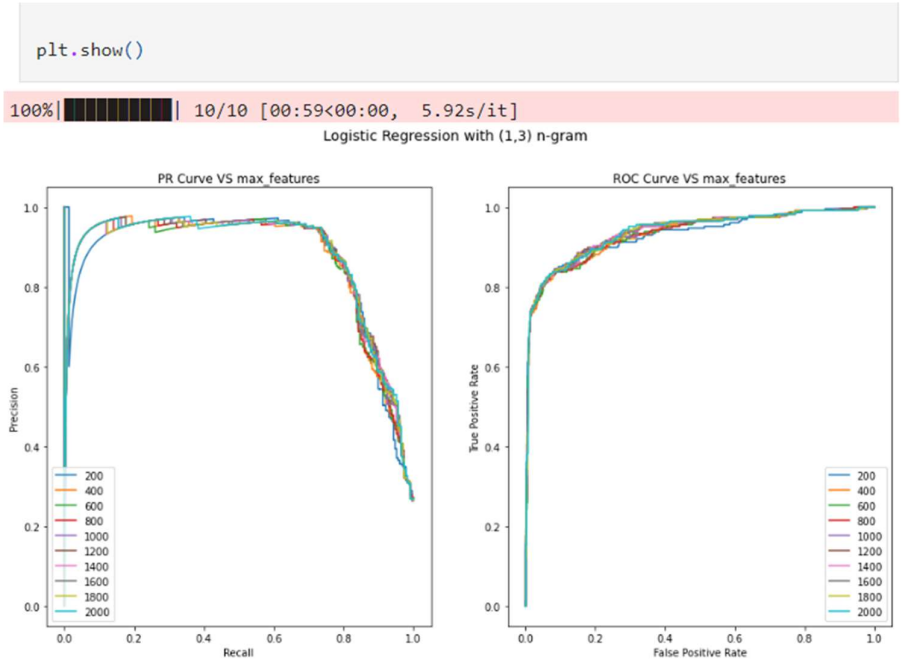

Figure 12: Effect Of Vocab Size On Prediction

CHAPTER 5

5.1 CONCLUSION

First, we pre pre-processed data to handle missing entries. Then we removed outliers from continuous numerical attributes and removed low variance Boolean attributes. After that we processed our data for NLP. After that we created common vector for our classifier and analysed the AUC and F1 scores of different models. At last, we checked the influence of corpus size on our prediction which we found to be negligible and hence we can conclude that numerical attributes had more effect on prediction.

5.2 FUTURE WORK

Some apps have no description. In addition, there's little to no useful information in knowing the app name or what version it's on. In the future we can divide the prediction model in two kinds i.e Numerical feature only (so no NLP work) and Hybrid: Numerical features + NLP. Also, we can target applications from other application store or can amalgate and compare information of same application present on different application stores and can incorporate further meta data.

LIST OF REFERENCES

[1] Yan P. and Yan Z., "A survey on dynamic mobile malware detection," Softw. Qual J, no. May 2017, pp. 891–919, 2018

[2] Razak M. F. A., Anuar N. B., Othman F., Firdaus A., Afifi F., and Salleh R., "Bio-inspired for Features Optimization and Malware Detection," Arab. J. Sci. Eng., vol. 43, no. 12, pp. 6963–6979, 2018

[3] T. K. Ho, "The random subspace method for constructing decision forests," Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 20, no. 8, pp. 832–844, 2018.

[4] B. Sanz, "MAMA: manifest analysis for malware detection in android," Cybernetics and Systems, vol. 44, pp. 6-7, 2013.

[5] S. R. Tiwari and R. U. Shukla, "An android malware detection technique based on optimized permissions and API," in Proceedings of the 2018 International Conference on Inventive Research in Computing Applications (ICIRCA), IEEE, Coimbatore, India, July 2018.

[6] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, pages 785 794, New York, NY, USA, 2016. ACM.

[7] Malte Ostendorff, Terry Ruas, Till Blume4 Aspect-based Document Similarity for Research Papers Proceedings of the 28th International Conference on Computational Linguistics, pages 6194–6206 Barcelona, Spain (Online), December 8-13, 2020

[8] S. Y. Yerima, I. Muttik, and S. Sezer, ''High accuracy Android malware detection using ensemble learning,'' IET Inf. Secur., vol. 9, no. 6, pp. 313–320, Nov. 2015

[9] Kamil Bennani-Smires, Claudiu Musat1, Andreaa Hossmann Simple Unsupervised Keyphrase Extraction using Sentence Embeddings Data, Analytics & AI, Swisscom AG 5 Sep 2018

[10] S. Y. Yerima, and S. Sezer, "Droidfusion: A novel multilevel classifier fusion a pproach for android malware detection," IEEE transactions on cybernetic, vol. 49, no. 2, pp. 453-466, 2018

[11] A. Feizollah, N. B. Anuar, R. Salleh, G. S. Tangil, and S. Furnell, "Androdialysis: Analysis of android intent effectiveness in malware detection," Computers & Security, vol. 65, pp. 121-134, 2017.

[12] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisaan, and Y. Heng, “Significant permission identification for machine-learning-based android malware detection,” IEEE Transactions on Industrial Informatics, vol. 14, no. 7, pp. 3216-3225, 2018.

[13] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, “Exploring permission-induced risk in android applications for malicious application detection,” IEEE Transactions on Information Forensics and Security, vol. 9, no. 11, pp. 1869-1882, 2014.

[14] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, “Exploring permission-induced risk in android applications for malicious application detection,” IEEE Transactions on Information Forensics and Security, vol. 9, no. 11, pp. 1869-1882, 2014.

PAPER NAME

thesis part for plag check.docx

WORD COUNT	CHARACTER COUNT
4497 Words	**25528 Characters**
PAGE COUNT	FILE SIZE
26 Pages	**1.3MB**
SUBMISSION DATE	REPORT DATE
May 29, 2023 5:17 PM GMT+5:30	**May 29, 2023 5:17 PM GMT+5:30**

4% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

- 3% Internet database
- 2% Publications database
- Crossref database
- Crossref Posted Content database
- 3% Submitted Works database

Excluded from Similarity Report

- Bibliographic material
- Quoted material
- Cited material
- Small Matches (Less then 8 words)