International Journal of Computing and Digital Systems
2025, VOL. 17, NO. 1, 1–11

http://dx.doi.org/10.12785/ijcds/1571000835

Strengthening Android Malware Detection: from Machine Learning to Deep Learning

Diptimayee Sahu[1*], Satya Narayan Tripathy[2] and Sisira Kumar Kapat[3]

[1,2]*Department of Computer Science, Berhampur University, Berhampur, India*
[3]*Utkal Gourav Madhusudan Institute of Technology, Rayagada, Odisha, India*

Received 23 February 2024, Revised 4 August 2024, Accepted 29 June 2024

Abstract: In the recent era of the modern world, Android malware continues to escalate, and the challenges associated with its usage are growing at an unprecedented rate. This causes rapid growth in Android malware infections, which points to an alarming and swift rise in their prevalence, signaling a cause for concern. Traditional anti-malware systems, reliant on signature-based detection, prove inadequate in addressing the expanding scope of newly developed malware. Various strategies have been introduced to counter the escalating threat in the Android mobile field, with many leaning towards machine learning (ML) models limited by a constrained set of features. This paper introduces a novel approach employing a deep learning (DL) framework, incorporating a significant number of diverse features. The proposed framework uses Deep Neural Network (DNN) techniques on a static OmniDroid dataset, comprising 25,999 features extracted from 22,000 Android Package Kits (APKs). Of these, 16,380 features are meticulously selected for analysis, encompassing Permission, Opcodes, Application Programming Interface(API) calls, System Commands, Activities, and Services. Additionally, the data is partitioned feature-wise and subjected to feature selection on each feature set to ensure equitable consideration of all features. A comparative analysis is presented by comparing the framework accuracy with the accuracies produced by the existing ML models. The presented framework demonstrates notable enhancements in detection accuracy, achieving 89.04% accuracy, attributed to the incorporation of a substantial number of features.

Keywords: Android malware, malware detection, deep learning, artificial neural network, feature selection.

1. Introduction

Technology integration makes smartphones an essential part of everyday life and a significant factor in the mobile security landscape. The development of Android malware is a global issue that needs to be resisted. This paper focuses on developing a detection framework based on DNN. A study reports that 450,000 new malware programs and potentially unwanted apps targeting individual handheld devices emerge daily [1]. Security professionals are encountering many malware attacks targeting smartphones, including banking trojans, spyware, adware droppers, etc. [2]. Notably, the Google Play Store alone hosts approximately 3.55 million Android applications of various types [3]. Despite that users can also install applications from third-party sources on the Android platform, increasing the risk of downloading malicious apps from hostile servers.

To develop an Android app, developers must include specific files in the package format with a .apk extension. One crucial file, AndroidManifest.xml, contains information about the app, such as the package version, required permissions, intents, actions, and services. The classes.dex file encapsulates the complete bytecode that defines the application's functionality. Android employs a permission-based security model, granting applications only those permissions explicitly allowed by the user, which enhances the overall security framework. However, these security measures have not eliminated the diverse range of attacks targeting Android, highlighting ongoing challenges in protecting the platform against evolving cybersecurity threats.

Most antivirus detection systems rely on signature-based detection, which malware writers can easily obfuscate by altering the malicious application's signature. Minimal obfuscation of malware code can also evade detection. Zhan et al. [4] demonstrated a patching technique in which a piece of code is appended at a nonfunctional location, enabling the malware to evade detection. Detecting Android malware is crucial for users, and numerous researchers have proposed various solutions to mitigate these attacks, reflecting continuous efforts to enhance the platform's resilience against

E-mail address: ds.rs.cs@buodisha.edu.in, snt.cs@buodisha.edu.in, skk.rs.cs@buodisha.edu.in

malicious threats.

The effectiveness of detection models may be compromised if they do not incorporate a diverse set of features during the training process, emphasizing the importance of exploring a wide range of characteristics for robust model development. The literature survey in this study reveals that although many detection frameworks are available, they often train on a limited number of features to reduce model complexity. This limitation can affect the detection of malicious applications whose features are discarded during the feature selection process in a given dataset. Overlooking such features can lead to errors in the detection model. Instead, considering a substantial number of features as input for training a model may increase the likelihood of detecting malicious applications. Addressing this problem, the proposed framework presents a solution in the following sections. This research aims to develop a supervised deep learning framework using DNN techniques, leveraging an array of static features identified through an extensive review of diverse research works. Comparative analysis is conducted through several experiments using a substantial number of input features.

The subsequent sections cover: the related literature survey in Section 2, the methodology of the proposed system in Section 3, the proposed framework in Section 4, the experimental setup in Section 5, the results and discussion in Section 6, a comprehensive comparative analysis in Section 7, and a summary of the work in Section 8.

2. Literature Survey

Gopinath et al. [5] surveyed the efficiency of deep learning in the field of malware detection and stated that models based on deep learning are robust and offer solutions to the shortcomings of traditional detection models.

Wang et al. [6] proposed a Multi-Network (MN) based classification model with multilevel permission extraction. They extracted 135 permissions and applied Principal component analysis (PCA) on permissions to eliminate redundant features and finally used 25 features for classification. They compared MN with Support Vector Machines (SVM), Decision Tree (DT), and Random Forest (RF) algorithms and found that MN performed better with over 95.8% accuracy.

ALTAİY et al. [7] In their study, utilized three deep learning methods, namely Convolutional Neural Network (CNN), DNN, and Long-Short-Term Memory Network (LSTM), focusing on a dataset comprising network traffic of botnet samples. Their experimental results indicated that, among the three methods, LSTM demonstrated superior performance.

Bai et al. [8] in their study introduced a framework aimed at Android malware family classification through the analysis of permissions, API calls, and Intents. Their research included a comparative evaluation of multiple machine learning models, including SVM, DT, RF, K-Nearest Neighbor (KNN), and Multi-Layer Perceptron (MLP), with findings indicating that MLP outperformed the other models in terms of classification accuracy.

Le et al. [9] proposed a machine learning-based approach on three different feature sets such as APIs, permissions, and other characteristics of APK files such as the size of the application, the number of classes included in the application, the number of User Interfaces (UIs) created by the application that refers to the total count of distinct screens, pages, or interactive components designed for user interaction within the application. They used RF, Stochastic Gradient Boosting (SGB), and AdaBoost classification methods. They extracted 65 features; 62 features about behavior and permissions and three features about the size of the application, the number of classes in the application, and the number of User Interface of the application. They achieved 98.66% accuracy using the RF among all other models.

Rodrigo et al. [10] introduced a hybrid detection model utilizing the Omnidroid dataset. Employing Pearson Correlation for feature selection, they narrowed it down to 840 features. The model underwent training and validation using a neural network, yielding an initial accuracy of 85.8%. Subsequent refinement involving threshold relabeling led to a notable accuracy boost, reaching 92.9%.

Oliveira et al. [11] introduced a hybrid detection approach combining the results of different models employing DNN techniques. One model utilizes 200 static features, the second model uses images, and the third model uses system call sequences. The final classifier produced 90.9% accuracy on a combination of all features.

Gao et al. [12] proposed a framework using DNN and generative adversarial network(GAN) for malware family classification against packed malware. They considered two datasets of packers from 10 different malware families. One dataset contains malware packed by a single packer, and the other dataset contains malware packed by multiple packers. They used DNN for malware detection and family classification. The detection accuracy of the packed malware achieved 98.20% and for family classification of the malware they achieved 91.66% accuracy which is elevated to 97.8% after using GAN. Hence they concluded that their framework is a solution to the packed malware.

Li et al. [13] in their proposed framework addressed the issue that, models trained on outdated datasets often result in suboptimal decision-making, particularly when confronted with contemporary malware types. Instead of using DNN for prediction and classification, they used the misclassifications or uncertainties done by the model. They trained another model called the correction model that evaluates whether a sample has been accurately or inaccurately predicted by the DNN model, providing crucial insights into the model's performance and identifying areas

TABLE I. Dataset Feature Distribution Overview

Name of the Features	Considered number of Features
Permissions	5501
Opcodes	224
API Calls	2129
System Commands	103
Activities	6089
Services	2334

for improvement. Then they used the outcomes of the Correction Model to refine and optimize the performance of the DNN model, thereby improving its accuracy and effectiveness in predicting malware instances. Their proposed model achieved 94.38% accuracy.

Aamir et al. [14] introduced a framework using CNN for Android malware detection. They used the Drebin dataset for their experiment. As CNN works on image data they converted the APK files into images using techniques like Spectrogram or Scalogram and trained the model using CNN and achieved an accuracy of 99.92%.

Nasser et al. [15] proposed a deep learning-based detection model called DL-AMDet. Their model performs the detection in stages. In the first stage, the model performs static analysis using permission and API calls trained on the CNN-BiLSTM model. If the application is detected as malware in the static analysis stage the model does not perform the dynamic analysis otherwise if the application is detected as non-malicious then the application again has to go for dynamic analysis. The used system calls as input feature for an anomaly detection model using deep autoencoders. They evaluated the model on different datasets and achieved 99.93% higher accuracy in the anomaly detection model.

3. Proposed Methodology

For detection, the proposed framework uses six important features such as permissions, opcodes, API calls, system commands, activities, and services. The static OmniDroid dataset [16] offers an extensive array of features, including 5501 permissions, 224 opcodes, 2129 API calls, 103 system commands, 6089 activities, 4365 services, 6415 receivers, 212 API packages, and 961 FlowDroid outputs, all derived from 11,000 benign and 11,000 malicious samples. In this paper, the experiments utilize 16,380 distinct features from the 25,999 available, comprising 5501 permissions, 224 opcodes, 2129 API calls, 103 system commands, 6089 activities, and 2334 services. These features are identified based on a comprehensive literature survey conducted for this study, with detailed explanations provided in Table I.

Permissions: Permissions are nothing but the safeguards that control the access rights of the apps to the features and data of the device. The permissions are declared under the <uses-permission> tag of the AndroidManifest.xml file in the APK. Permissions play a crucial role in malware

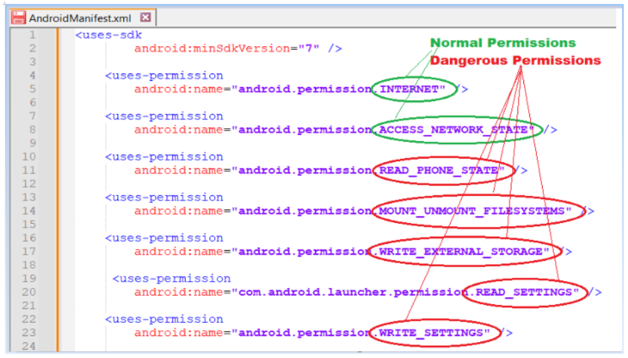

```
<uses-sdk
        android:minSdkVersion="7" />

    <uses-permission
        android:name="android.permission.INTERNET" />

    <uses-permission
        android:name="android.permission.ACCESS_NETWORK_STATE" />

    <uses-permission
        android:name="android.permission.READ_PHONE_STATE" />

    <uses-permission
        android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS" />

    <uses-permission
        android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

     <uses-permission
        android:name="com.android.launcher.permission.READ_SETTINGS" />

    <uses-permission
        android:name="android.permission.WRITE_SETTINGS" />
```

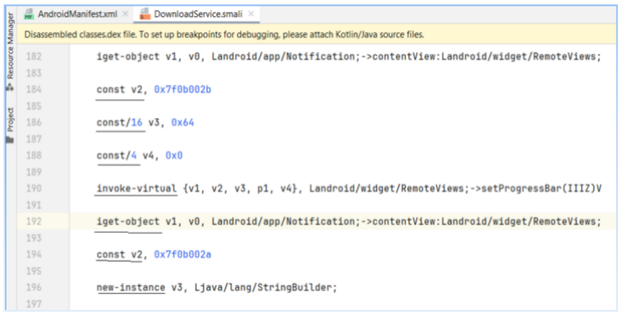
Figure 1. Example of Normal and Dangerous Permissions

detection. It brings the user's consent towards the app's access to sensitive data by seeking an explicit grant for the requested access. Permissions are categorized into normal, dangerous, and signature. Figure 1 lists some permission extracted from a malicious app of Trojan type. It contains both normal and dangerous permission. As shown in Figure 1 the app asks for INTERNET, and ACCESS_NETWORK_STATE permissions, which are commonly asked by most applications and considered normal permissions whereas the other permissions such as; READ_PHONE_STATE, WRITE_EXTERNAL_STORSGE, MOUNT_UNMOUNT_FILESYSTEM, READ_SETTINGS and WRITE_SETTINGS are the dangerous permissions that may authorize access to external data and resources beyond the application's controlled environment, in most cases putting user data and system integrity at risk. Various studies have been put forth to date aimed at detecting Android malware through the analysis of permission features [17] [18].

Opcodes: these are the human-readable instructions in a program that the Dalvik Virtual Machine (DVM) executes. With Android's shift to the Android Runtime (ART), these programs are now executed by the ART instead. These opcodes are generated during the compilation process that represents the operations to be performed within the application. Figure 2 is an example that shows the opcodes that is extracted from Android applications. Many studies with good performance have been conducted so far utilizing opcodes [19][20].

Services: services are one of the primary components that every Android app has to have. Services do not need a visible interface instead they seamlessly operate in background for the tasks like downloading a large file, uploading a large data, playing music, etc. Inter-process Communication (IPC) also happens through services between the apps. Figure 3 shows an example of the services listed by a malicious app in its AndroidManifest.xml file.

Activities: activities in Android applications are another component that operates in individual User Interfaces (UI)

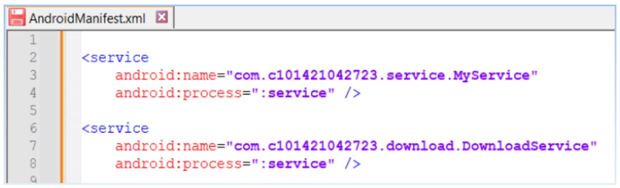


AndroidManifest.xml | DownloadService.smali

Disassembled classes.dex file. To set up breakpoints for debugging, please attach Kotlin/Java source files.

```
iget-object v1, v0, Landroid/app/Notification;->contentView:Landroid/widget/RemoteViews;

const v2, 0x7f0b002b

const/16 v3, 0x64

const/4 v4, 0x0

invoke-virtual {v1, v2, v3, p1, v4}, Landroid/widget/RemoteViews;->setProgressBar(IIIZ)V

iget-object v1, v0, Landroid/app/Notification;->contentView:Landroid/widget/RemoteViews;

const v2, 0x7f0b002a

new-instance v3, Ljava/lang/StringBuilder;
```

Figure 2. Example Of Opcodes



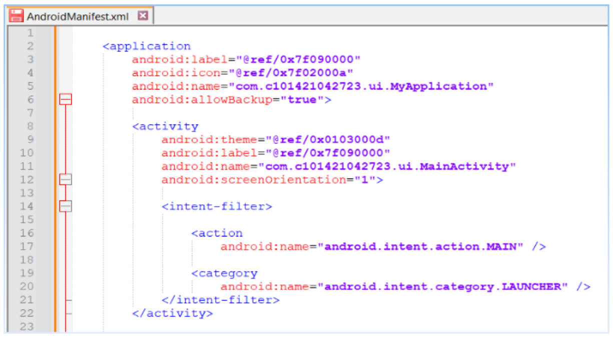
AndroidManifest.xml

```

<service
    android:name="com.c101421042723.service.MyService"
    android:process=":service" />

<service
    android:name="com.c101421042723.download.DownloadService"
    android:process=":service" />
```

Figure 3. Example Of Services

within an app. They are the entry points for user interaction like the main() method in other programs. The code initiation by OS happens by a callback method in an activity instance and it corresponds to stages of lifecycle to accomplish a single task. Activities manage the user interface, handle user input events, and facilitate interaction between the app and the user. They play a crucial role in the Android app lifecycle, transitioning between different states such as creation, pausing, resuming, and destruction based on user interaction and system events. Figure 4 is an example to show the activities in an application.

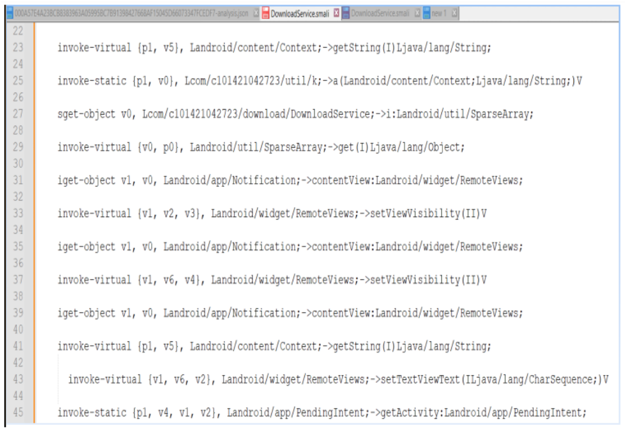
API Calls: API calls in Android refer to the services, resources, or functionalities the applications access from remote servers or web services. Any action that needs interaction with the remote server such as data fetching, user authentication, sending notifications, etc. is done by making API calls. These calls are made using HTTP requests by the API of the service provider. Figure 5 shows the API



AndroidManifest.xml

```

    <application
        android:label="@ref/0x7f090000"
        android:icon="@ref/0x7f02000a"
        android:name="com.c101421042723.ui.MyApplication"
        android:allowBackup="true">

        <activity
            android:theme="@ref/0x0103000d"
            android:label="@ref/0x7f090000"
            android:name="com.c101421042723.ui.MainActivity"
            android:screenOrientation="1">

            <intent-filter>

                <action
                    android:name="android.intent.action.MAIN" />

                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

```

Figure 4. Example Of Activities



000A57E4A23BCB83B3963A05995BC7B91398427668AF15045D66073347FCEDF7-analysis.json | DownloadService.smali | DownloadService.smali | new 1

```

    invoke-virtual {p1, v5}, Landroid/content/Context;->getString(I)Ljava/lang/String;

    invoke-static {p1, v0}, Lcom/c101421042723/util/k;->a(Landroid/content/Context;Ljava/lang/String;)V

    sget-object v0, Lcom/c101421042723/download/DownloadService;->i:Landroid/util/SparseArray;

    invoke-virtual {v0, p0}, Landroid/util/SparseArray;->get(I)Ljava/lang/Object;

    iget-object v1, v0, Landroid/app/Notification;->contentView:Landroid/widget/RemoteViews;

    invoke-virtual {v1, v2, v3}, Landroid/widget/RemoteViews;->setViewVisibility(II)V

    iget-object v1, v0, Landroid/app/Notification;->contentView:Landroid/widget/RemoteViews;

    invoke-virtual {v1, v6, v4}, Landroid/widget/RemoteViews;->setViewVisibility(II)V

    iget-object v1, v0, Landroid/app/Notification;->contentView:Landroid/widget/RemoteViews;

    invoke-virtual {p1, v5}, Landroid/content/Context;->getString(I)Ljava/lang/String;

      invoke-virtual {v1, v6, v2}, Landroid/widget/RemoteViews;->setTextViewText(ILjava/lang/CharSequence;)V

    invoke-static {p1, v4, v1, v2}, Landroid/app/PendingIntent;->getActivity:Landroid/app/PendingIntent;
```

Figure 5. Example Of API Calls

calls of an application mentioned in its classes.dex file. The API calls involve; constructing the request, sending the request, handling the response, parsing the response, and finally performing the requested action or updating the user interface (UI). The HTTP request contains the method, URL, header, and body. Then the constructed request is sent using any of the libraries like Retrofit, Volley, etc. Once the request is processed by the server, the application receives a response. This response may include data or the status of the request such as success or failure. Then the received response is parsed to extract the relevant information and based on it the application performs the specific required action. The API call is another extensively utilized feature for Android malware detection, with numerous researchers incorporating it into their studies [21][22].

System Commands: system commands sometimes modify the installed applications, which can potentially cause unintended damage to the device. Some common system commands are; logcat, reboot, install, and uninstall, etc.

A. *Data Pre-processing and Feature Selection*

In this study data cleaning process involves a row reduction technique to eliminate duplicate entries. Rather than conducting feature selection on the entire dataset as a whole, the data is partitioned feature-wise, and feature selection is subsequently carried out using the Information Gain (IG) algorithm on each feature set separately. The IG algorithm computes a value, referred to as the IG score, for each feature in the feature set. This IG score indicates the information content of the respective feature within the dataset. Each feature's IG score is determined by subtracting the individual feature's entropy from the entropy of the output column, where entropy signifies uncertainty in this context. The calculation of IG scores for each feature follows Equation (1).

$$IG(f_i) = H(y) - H(f_i) \tag{1}$$

In Equation (1), IG(f_i) is the score calculated using the IG technique for each feature that reflects the amount of

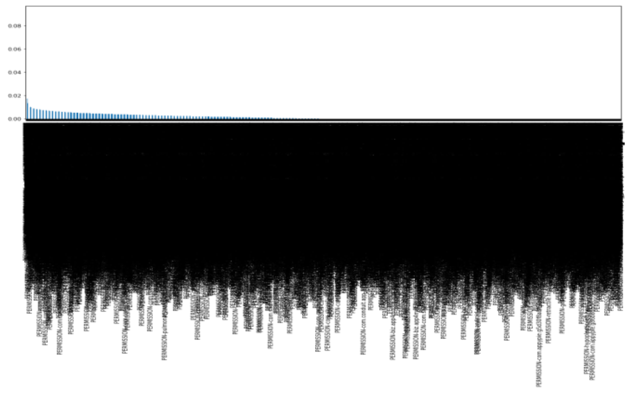


Figure 6. Ranking Of The Permissions In Descending Order.

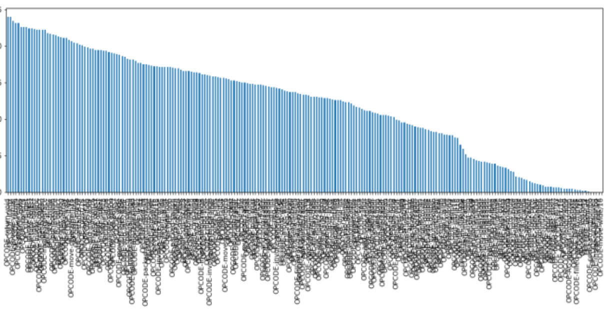


Figure 8. Ranking Of The Opcodes In Descending Order.

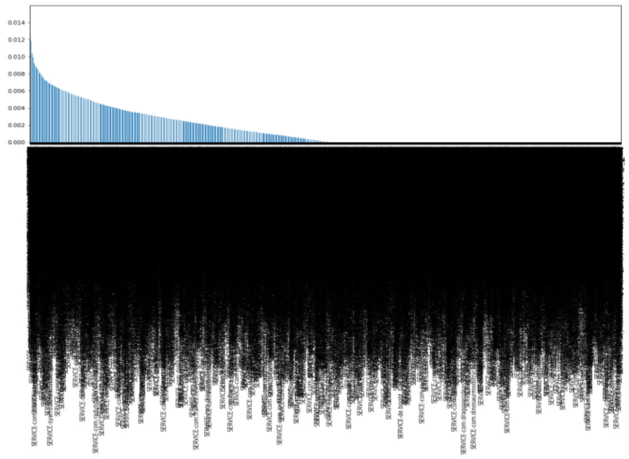


Figure 7. Ranking Of The Services In Descending Order.

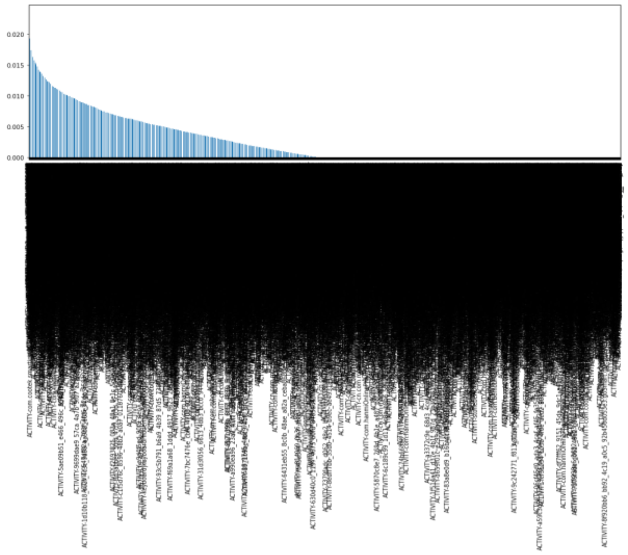


Figure 9. Ranking Of The Activities In Descending Order.

information each feature contributes to the dataset; here, 'i' denotes the feature's column index. H (y) represents the entropy of the output column, and H(f_i) is the entropy of each feature. In this proposed work individual features (Permissions, Opcodes, API calls, System Commands, Activities, and Services) are separated from the whole dataset. Then using the IG technique the features are evaluated and scored to understand the information content of individual feature types such as; the impact of permission in malware detection, the impact of API calls in malware detection, and likewise for other features. Figures 6 through 11 provide a comprehensive insight into the information content of various feature types. Figure 6 represents the information content of permissions concerning the target variable. Figure 7 represents the information content of services concerning the target variable. Figure 8 represents the information content of opcodes concerning the target variable. Figure 9 represents the information content of activities concerning the target variable. Figure 10 represents the information content of API calls concerning the target variable and Figure 11 represents the information content of system commands concerning the target variable.

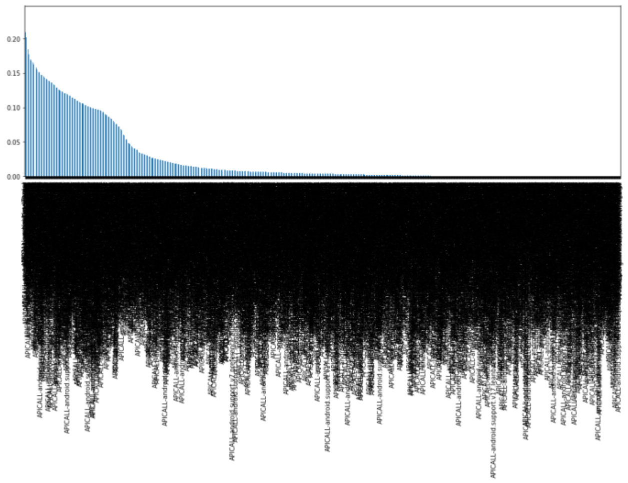


Figure 10. Ranking Of The API Calls In Descending Order.

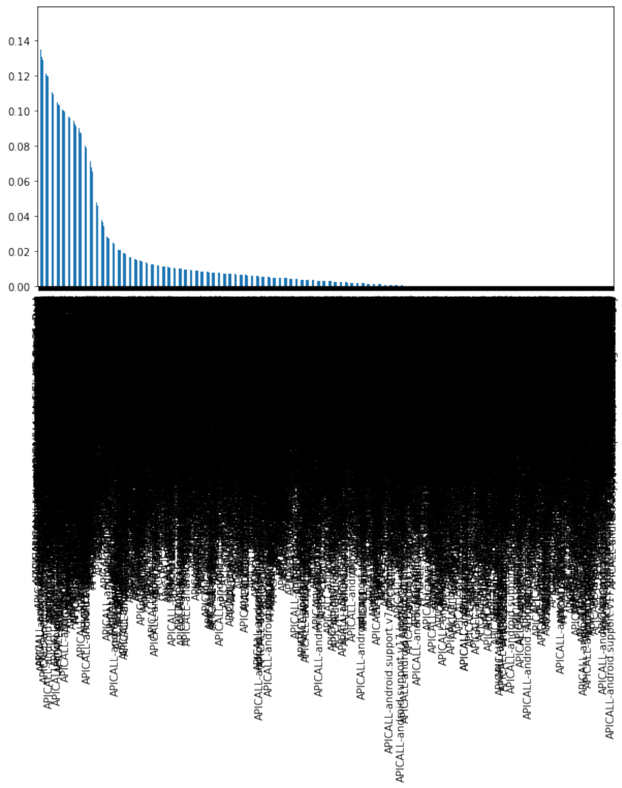
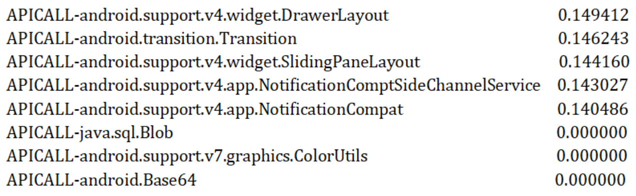


Figure 11. Ranking Of The System Commands In Descending Order.



APICALL-android.support.v4.widget.DrawerLayout	0.149412
APICALL-android.transition.Transition	0.146243
APICALL-android.support.v4.widget.SlidingPaneLayout	0.144160
APICALL-android.support.v4.app.NotificationComptSideChannelService	0.143027
APICALL-android.support.v4.app.NotificationCompat	0.140486
APICALL-java.sql.Blob	0.000000
APICALL-android.support.v7.graphics.ColorUtils	0.000000
APICALL-android.Base64	0.000000

Figure 12. Exemplary Representation of Features and their IG Score

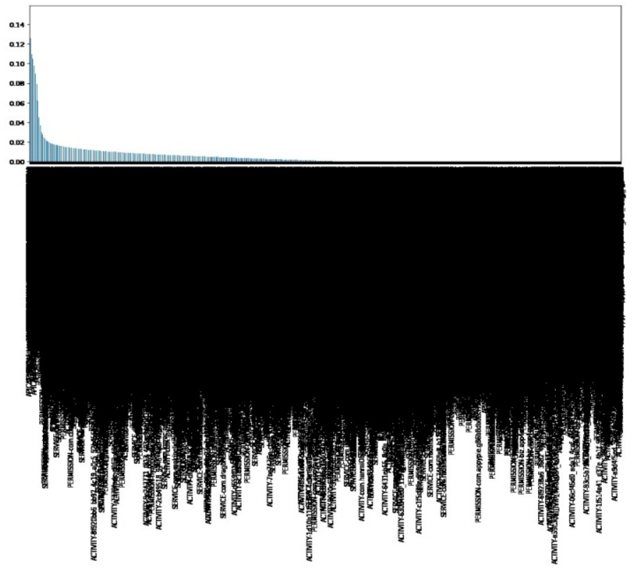


Figure 13. Features Plotted In Descending IG Score

Upon individual feature evaluation, it becomes evident that each feature makes a distinct contribution to the prediction of the output. When multiple feature types are considered within a single dataset, there exists a possibility that during feature selection, the algorithm might disregard one feature type entirely due to its lower information gain score compared to another feature type. For instance, in scenarios where features of opcode type demonstrate higher information gain values than those of permission types, permission type features may probably be entirely overlooked during the feature selection stage. To address this scenario, feature selection is applied individually to each feature type ensuring equal importance is given to all considered feature types. Based on the observation, 40% of features from each feature type are selected and subsequently combined into a single dataset. Following feature selection, the dataset now comprises 6552 features, including 2200 permissions, 90 opcodes, 852 API calls, 41 system commands, 2436 activities, and 933 services. Subsequently, IG is reapplied to the selected feature data to evaluate the IG scores. Figure 12 represents the IG score of the features and those are plotted based on the score in descending order as shown in Figure 13.

4. Proposed Framework

This segment presents a framework using the DNN approach for Android malware detection. The proposed system architecture, illustrated in Figure 14, involves collecting raw data, selecting the desired subset from the entire dataset, performing feature selection, and conducting various experiments using the DNN model. The details about the dataset and the feature selection process have already been explained in Section III. The DNN architecture encompasses the arrangement of an input layer, hidden layers, and an output layer. The data is computed through forward propagation, and the back-propagation algorithm is employed to fine-tune the efficient parameters at each layer. The input is given to the network in batches for processing. The hidden layers within the network process the input and the neurons in the output layer generate the final predictions. The output computed by the model is expressed in a simplified form by equation (2). The model incorporates the rectified linear activation unit (ReLU) for non-linear transformations on each hidden layer. It addresses the vanishing gradient problem as denoted in equation (3). The computation at the output layer by the sigmoid activation function is shown in equation (4).

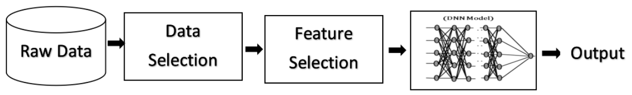


Figure 14. Proposed System Architecture

TABLE II. The Final Set Of Model Parameters For The Proposed System

Hyper-Parameters in the Network	Values
Number of epochs	400
Number of hidden layers	13
Dropout	0.2
Batch size	128
Loss optimization function	Binary_crossentropy

$$Z_n = f_{sigmoid}\left(\left(f_{relu}\left(l_{hidden}\right)\right)_{i=1}^{p}\right)_{j=1}^{q} \quad (2)$$

$$where, f_{relu} = \max\left(0, Z_n\right), \forall n \in N \quad (3)$$

$$and, f_{sigmoid} = \frac{1}{1+e^{(-z)}} \quad (4)$$

In equation (2), 'i' varies from 1 to p, where 'p' represents the number of neurons within a given hidden layer, 'j' spans from 1 to q, where 'q' denotes the total number of hidden layers in the network, and 'Z_n' signifies the output score produced by the model.

Efficiently training a DNN demands significant effort in exploring and identifying optimal parameters that enhance the model's performance. Instead of employing a trial-and-error approach to determine efficient parameter values, the experiments utilized the RandomSearch optimization algorithm. This algorithm identifies a suitable combination of hyperparameters, encompassing variables such as the number of hidden layers, the number of neurons in each hidden layer, and the learning rate, among others. In this study, several parameters are configured for the Random Search algorithm to perform hyperparameter optimization. For instance, the number of layers is set within a range of 2 to 20, meaning the network will have at least 2 and at most 20 hidden layers. Similarly, the number of neurons per layer is specified to be between 32 and 512, ensuring that each layer will contain a minimum of 32 neurons and a maximum of 512 neurons. Details about the hidden layers and neurons are provided in Table II. Several experiments were performed with epoch (number of times the network performs learning) numbers (i.e., 50, 100, 200, 400, 500), with 5 values for the dropout rate (0.0, 0.1, 0.2, 0.3, 0.5). Based on the results obtained in different experiments 400 as the epoch no and the dropout rate of 0.2 as best values are considered for all the experiments. In this study, the considered hyper-parameters are explained in Table II.

In this proposed framework regularizers are used at different levels of the model such as kernel level, bias level, and output layer with L2 regularization penalty to deal with the overfitting problem during model training. The loss calculation by the model using the L2 regularization technique is represented in Equation (5) which says the sum of the squares of the entire feature weights are added to the original entropy and the penalty is calculated based on it.

$$Loss\_with\_regularizaion = E + \lambda \sum_{k=1}^{n} w_k^2 \quad (5)$$

In equation (5); E is entropy (the loss generated by the model),λ is a regularization constant ($\lambda > 0$),W_k represents the weight of the k^{th} parameter of the model.

5. Experimental Setup

All experiments conducted in this study utilized a "Tesla T4" GPU, with TensorFlow 2.8.0 [23] as the backend and Keras [24], provided by Google Colaboratory. The experimental setup employed a system running Microsoft Windows 10 Professional (64-bit) with a 1.80 GHz Intel Core i5 processor and 8.00 GB of memory. Dataset preprocessing was facilitated using the Scikit-learn [25] Python library. Model performance was evaluated using key metrics, including True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN), which are essential components in computing the mean accuracy derived using Equation (6).

$$Accuracy = \frac{TP + TN}{TN + TP + FP + FN} \quad (6)$$

6. Result and Discussion

Multiple experiments are conducted on the proposed system to generate a comprehensive comparative analysis, shedding light on various aspects of the study. It includes evaluating the impact of feature selection versus without considering feature selection, as well as contrasting the performance of machine learning algorithms with the proposed DNN-based model using a substantial number of features. The accuracy and loss are monitored in all the experiments on both the train and test dataset. The framework's effectiveness is determined through the analysis presented in the subsequent sections.

A. Analysis Based on the Data Without Feature Selection

This is the first set of experiments performed on the framework without performing feature selection. A total of 16,380 distinct static features, including permissions, opcodes, API calls, system commands, Activities, and Services, are taken into account for analysis. The dataset is partitioned, with 75% allocated for training, 15% for testing and 10% for validation. The model is trained and tested using DNN techniques. Keras TensorFlow is used to deal with the overfitting of the model. To ensure a valid comparative analysis, the model parameters, as outlined in Table II, remain consistent across all experiments conducted in this study.

The experiments are done on the dataset without performing any feature selection mechanism on it. Accuracy and loss are monitored as they are crucial metrics for evaluating the performance of a model, particularly in the

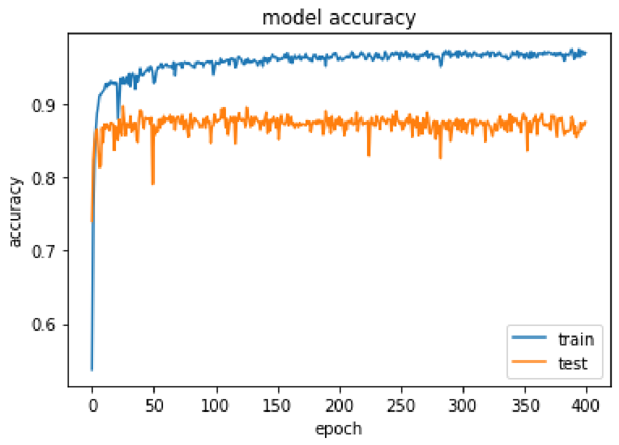


Figure 15. Model Accuracy Score Without Feature Selection

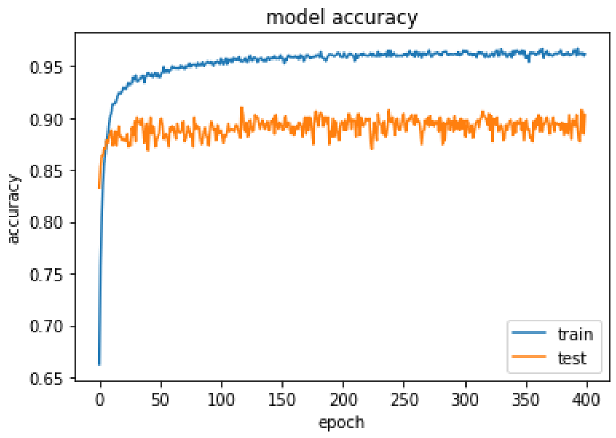


Figure 17. Model Accuracy Score With Feature Selection

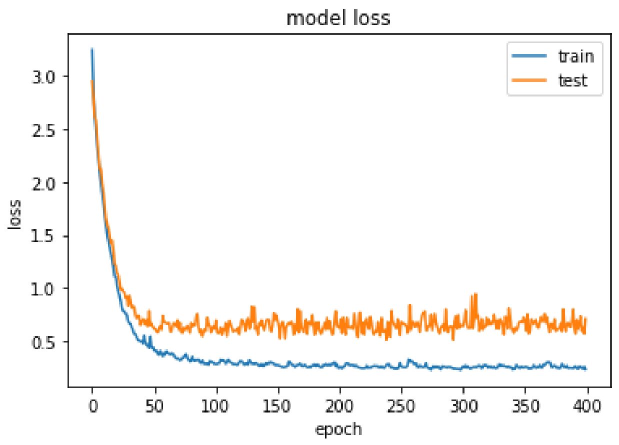


Figure 16. Model Loss Score Without Feature Selection

context of supervised learning. The model undergoes 400 iterations through the entire train dataset in batches of 128, followed by evaluating its performance on a test dataset after each epoch. The training and testing accuracy at each epoch is presented in Figure 15 and the loss is presented in Figure 16. The x-axis of all the model accuracy and model loss plots represents the number of epochs the model executed, the y-axis of the model accuracy represents the accuracies generated at each epoch, and the y-axis of the model loss plots represents the loss generated at each epoch. The model's performance in initial epochs fluctuates significantly as it starts learning and adjusting the weights which can distort the overall performance. To mitigate the impact of this noise and based on the observations from the accuracy and loss plots the mean accuracy and mean loss are calculated from 50^{th} epoch onwards. The model achieves a mean accuracy of 87.22% while demonstrating a mean loss of 0.71 on the testing set.

B. Analysis Based on the Data With Feature Selection

This is another set of experiments performed on the framework with the same model parameters, but here feature selection is performed on the dataset using the IG method. Based on the IG ranking of the features it is observed that near about 40% of the total features have more or less contribution to the dataset and the remaining features have very less or zero contribution to the dataset. Therefore, out of 16380 features 6552 features are considered for classification purposes. The accuracy and loss for training and testing at each epoch are presented in Figure 17 and Figure 18 respectively and based on the observations the mean accuracy and mean loss are calculated from 50^{th} epoch onwards. The findings reveal a mean accuracy of 89.04% and a decreasing mean loss of 0.61 on the feature-selected test dataset containing 6552 features.

In addition to the mean, the standard deviation (SD) for accuracy and loss is also calculated for both the training and test sets of the feature-selected data, starting from the 50^{th} epoch onwards. The SD of model accuracy is 0.015 on the training set and 0.010 on the test set of the feature-selected data. Similarly, the SD of model loss is 0.061 on the training set and 0.047 on the test set of the feature-selected data. A confusion matrix is generated on the validation set of the feature-selected dataset, providing scores for precision, recall, and F1 scores. The model gives 88% precision, 88% recall, and 89% F1 score which signifies the model is consistent.

7. Comparison Analysis

This section performs the comparison between the results obtained by performing feature selection and without considering feature selection. A detailed breakdown of the comparative analysis, focusing on evaluation metrics such as accuracy and loss, is presented in Figure. 19 and Figure. 20. Accuracy and loss are monitored at each epoch in both the train and test dataset.

In Figure 19, train accuracy represents the accuracy monitored on the feature-selected train dataset containing 6552 features, and the comparison is made between the accuracies obtained from the test dataset with 16,380

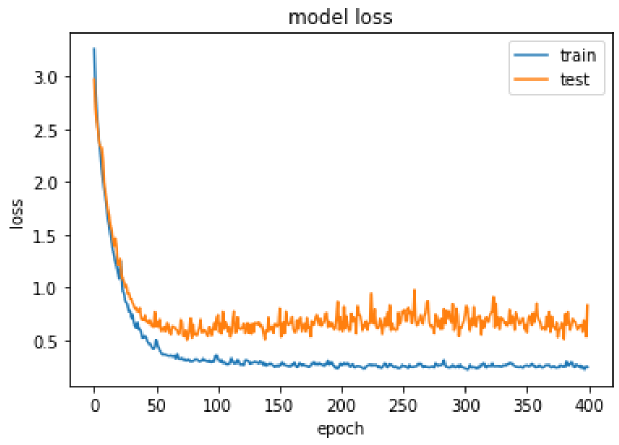


Figure 18. Model Loss Score With Feature Selection

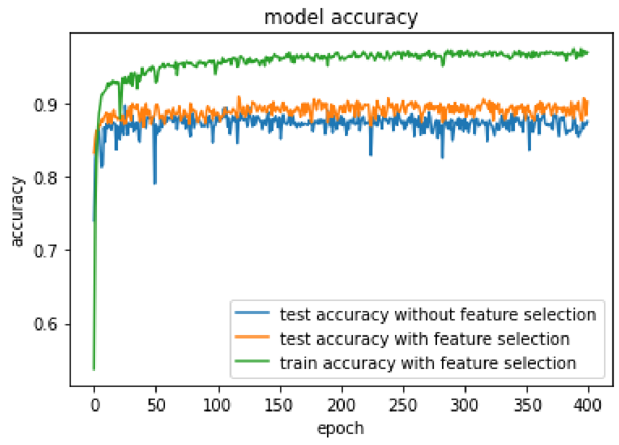


Figure 19. Comparison Of Model Accuracy Score

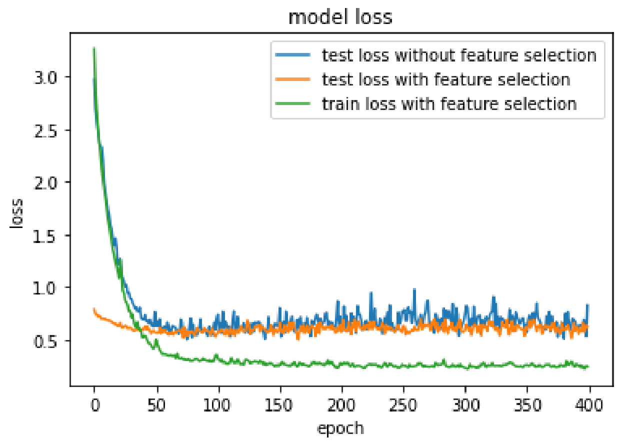


Figure 20. Comparison Of Model Loss Score

features and the feature-selected test dataset with 6,552 features. Similar to the previous experiments, the mean accuracy in this experiment is also calculated from the 50th epoch onwards. Figure 19 clearly illustrates that the model's accuracy increases from 87.22% to 89.04% on the feature-selected test dataset. 87.22% is the mean accuracy obtained by the model on the test dataset without performing any feature selection on it and 89.04% is the mean accuracy obtained by the model on the feature selected test dataset. Similarly, in Figure 20, train loss is the loss generated by the model on the training set of the feature-selected dataset. The comparison is drawn between the loss generated by the model on the test set of feature-selected data and non-feature-selected data at each epoch, and it is observed that the mean loss generated by the model on the non-feature-selected test set is 0.71 which is reduced to 0.61 on the test set of feature-selected data. The mean loss is also calculated from 50th epoch onwards.

A. *Time Comparison Between Both the Experiments*

Based on the experiments, it is observed that model training time is significantly reduced when feature selection is applied compared to when it is not. Although the feature selection process itself takes a bit of additional time, it is a one-time process and can be disregarded in the overall model evaluation. In our experiment, training the model with all 16,380 features took 250 seconds. In contrast, the feature selection process took 10 minutes, but subsequently, training the model with 6,552 selected features only took 174 seconds. Our experiment incorporates a sufficient number of features compared to other existing models in the literature, as shown in Table IV, ensuring comprehensive coverage of potential vulnerabilities for effective malware detection.

B. *Comparison With Other Machine Learning Models*

In the assessment of the proposed system, the model's performance is compared with the performance of other machine learning methods on the feature selected dataset that contains 6552 features. Among the existing ML models RF, KNN, SVC, and DT are used for performance comparison. For the above ML models, the RandomizedSearchCV hyperparameter tuning algorithm is employed to perform a random search over a specified parameter grid and cross-validate the results. For RF, RandomizedSearchCV evaluated 100 different sets of hyperparameters using 5-fold cross-validation, achieving an accuracy of 86.58% on the test set of non-feature-selected data and 87.37% on the test set of feature-selected data. In SVM, RandomizedSearchCV uses 5-fold cross-validation and achieves 85.30% on the test set of non-feature-selected data and 85.44% on the test set of feature-selected data with regularization parameter c=100. In DT the parameters provided to RandomizedSearchCV allowed for the selection of the criterion between 'gini' and 'entropy', offered options for 'max-depth' including (None, 5, 10, 15), and provided choices for 'min-samples-split' among (2, 5, 10). It produced 83.94% accuracy on the test set of the non-feature-selected dataset

TABLE III. Comparison With Other Machine Learning Models Based On Accuracy

Classifier	Accuracy (without feature selection)	Accuracy (with feature selection)
RF	86.58%	87.37%
KNN	82.32%	83.57%
SVC	85.30%	85.44%
DT	83.94%	84.44%
DNN	87.22%	89.04%

TABLE IV. Comparison With Similar Works In Terms Of Number Of Features

Other Similar Works	Number of Features used
Wang et.al. [6]	25
Le et.al [9]	65
Rodrigo et.al [10]	3359
Oliveira et.al [11]	200
Proposed framework	**6552**

and 84.44% accuracy on the test set of the feature-selected dataset using the parameters criterion as 'entropy', max-depth as None, and min-samples-split as 2. Similarly, KNN also used RandomizedSearchCV and produced 82.32% accuracy on the test set of the non-feature-selected dataset and 83.32% on the test set of the feature-selected dataset with the number of nearest neighbors as 10, Weight function used in prediction as 'distance' and the algorithm used in computing the nearest neighbors is 'brute'. The performance comparison, outlined in Table III, which provides insights into the efficacy of the approach relative to other machine learning classifiers.

The results presented in Table III indicate that all classification models employed in our research effectively detect malware apps, with only marginal differences in performance. The RF classifier achieves higher accuracy compared to the other ML classifiers. Nevertheless, the proposed DNN model outperforms the RF classifier, as neural networks tend to deliver superior detection accuracy when trained on large datasets.

C. Comparison With Similar Work

To show the effectiveness of the framework the accuracy obtained by our proposed framework is compared with other related static approaches based on the same OmniDroid dataset [16] listed in the literature survey section. Rodrigo et.al [10] and Oliveira et.al [11] also used the OmniDroid dataset. Rodrigo et.al [10] produced a static OmniDroid simplified dataset using various selection methods and obtained 85.8% accuracy using 3359 features. Oliveira et al. [11] similarly obtained impressive results with their static detection models. In comparison, our proposed model demonstrates superior performance, achieving an accuracy of 89.04%, given it is trained on an adequate number of features as illustrated in Table IV. It is believed that when the dataset is bigger enough, it would be more representative and consequently the resulting classifier is more effective in detecting malicious content. Therefore, the experimental results affirm the assertion that the proposed model yields substantial improvement in the realm of malware detection.

8. Conclusion

In response to the escalating infection rate of Android malware, a critical need for gateway-level malware detection has emerged. This study introduces a framework that employs DNN techniques, on static features such as permissions, Opcodes, API calls, Activities, Services, etc. extracted from Android applications. Feature selection is independently carried out on each feature set to prevent overlooking any specific type of feature. The proposed framework demonstrates a remarkable 89.04% accuracy, leveraging an extensive feature set for model training. This not only signifies a more precise malware detection capability but also outperforms frameworks trained on limited feature sets. Comparative analysis with existing literature and studies utilizing the static OmniDroid dataset reveals that the proposed system is validated on a substantial number of features, surpassing the accuracy achieved by models with fewer features. As part of future work, we aim to enhance malware detection accuracy further by exploring additional DL methods.

References

[1] "Avtest, malware, development of android malware and pua," last accessed 16 Feb 2024. [Online]. Available: https://www.av-test.org/en/statistics/malware/

[2] "Types of mobile malware," last accessed 24 Jan 2024. [Online]. Available: https://www.kaspersky.co.uk/resource-center/threats/mobile

[3] "Number of apps available in leading app stores as of 3rd quarter 2022," last accessed 8 Dec 2023. [Online]. Available: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[4] D. Zhan, Y. Duan, Y. Hu, W. Li, S. Guo, and Z. Pan, "Malpatch: Evading dnn-based malware detection with adversarial patches," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 1183–1198, 2024.

[5] M. Gopinath and S. C. Sethuraman, "A comprehensive survey on deep learning based malware detection techniques," *Elsevier*, vol. 47, 100529, 2023.

[6] Z. Wang, K. Li, Y. Hu, A. Fukuda, and W. Kong, "Multilevel permission extraction in android applications for malware detection," pp. 1–5, 2019.

[7] M. Altaiy, İ. Yıldız, and B. Uçan, "Malware detection using deep learning algorithms," *AURUM Journal of Engineering Systems and Architecture*, vol. 7, no. 1, pp. 11–26, 2023.

[8] Y. Bai, Z. Xing, D. Ma, X. Li, and Z. Feng, "Comparative analysis of feature representations and machine learning methods in android family classification," *Computer Networks*, vol. 184, 107639, 2021.

[9] N. C. Lê, T.-M. Nguyen, T. Truong, N.-D. Nguyen, and T. Ngô, "A

machine learning approach for real time android malware detection," pp. 1–6, 2020.

[10] C. Rodrigo, S. Pierre, R. Beaubrun, and F. El Khoury, "Brainshield: a hybrid machine learning-based malware detection model for android devices," *Electronics*, vol. 10(23), 2948, pp. 1–19, 2021.

[11] A. S. de Oliveira and R. J. Sassi, "Chimera: an android malware detection method based on multimodal deep learning and hybrid analysis," *Authorea Preprints*, 2023.

[12] X. Gao, C. Hu, C. Shan, and W. Han, "Malicage: A packed malware family classification framework based on dnn and gan," *Journal of Information Security and Applications*, vol. 68, 103267, 2022.

[13] H. Li, G. Xu, L. Wang, X. Xiao, X. Luo, G. Xu, and H. Wang, "Malcertain: Enhancing deep neural network based android malware detection by tackling prediction uncertainty," pp. 1–13, 2024.

[14] M. Aamir, M. W. Iqbal, M. Nosheen, M. U. Ashraf, A. Shaf, K. A. Almarhabi, A. M. Alghamdi, and A. A. Bahaddad, "Amd-dlmodel: Android smartphones malware detection using deep learning model," *Plos one*, vol. 19(1), e0296722, pp. 1–16, 2024.

[15] A. R. Nasser, A. M. Hasan, and A. J. Humaidi, "Dl-amdet: Deep learning-based malware detector for android," *Intelligent Systems with Applications*, vol. 21, 200318, pp. 1–10, 2024.

[16] "Omnidroid dataset csv static features experiments v2," last accessed 14 Jan 2024. [Online]. Available: http://aida.etsisi.upm.es/download/omnidroid-dataset-csv-features-v1/

[17] D. Ö. Şahin, O. E. Kural, S. Akleylek, and E. Kılıç, "A novel permission-based android malware detection system using feature selection based on linear regression," *Neural Computing and Applications*, pp. 1–16, 2023.

[18] F. Akbar, M. Hussain, R. Mumtaz, Q. Riaz, A. W. A. Wahab, and K.-H. Jung, "Permissions-based detection of android malware using machine learning," *Symmetry*, vol. 14(4), 718, pp. 1–19, 2022.

[19] H. Liu, L. Gong, X. Mo, G. Dong, and J. Yu, "Ltachecker: Lightweight android malware detection based on dalvik opcode sequences using attention temporal networks," *IEEE Internet of Things Journal*, 2024.

[20] A. Lakshmanarao and M. Shashi, "Android malware detection with deep learning using rnn from opcode sequences." *International Journal of Interactive Mobile Technologies*, vol. 16, no. 1, 2022.

[21] P. Maniriho, A. N. Mahmood, and M. J. M. Chowdhury, "Api-maldetect: Automated malware detection framework for windows based on api calls and deep learning techniques," *Journal of Network and Computer Applications*, vol. 218, 103704, pp. 1–18, 2023.

[22] M. AL-Akhras, S. Alghamdi, H. Omar, and H. Alshareef, "A machine learning technique for android malicious attacks detection based on api calls," *Decision Science Letters*, vol. 13, no. 1, pp. 29–44, 2024.

[23] "Tensorflow," last accessed 13 Jan 2024. [Online]. Available: https://www.tensorflow.org/

[24] "Keras," last accessed 13 Jan 2024. [Online]. Available: https://keras.io/getting_started/

[25] "Scikitlearn, machine learning in python," last accessed 13 Jan 2024. [Online]. Available: https://scikit-learn.org/stable/