Strengthening Android Malware Detection: from Machine Learning to Deep Learning

***Abstract*— In the recent era of modern world, Android malware continues to escalate, the challenges associated with its usage are growing at an unprecedented rate. This cause a rapid growth in Android malware infections points to an alarming and swift rise in their prevalence, signalling 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 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, 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;*

I. INTRODUCTION

The adoption of technology is making Smartphones an essential part of everyday life and an important phenomenon in the mobile security landscape. A study says every day 11000 new malware apps come to market whose main target is individual handheld devices [1]. The widespread anxiety induced by the COVID-19 pandemic encountered various malicious entities such as banking Trojans, spyware adware droppers, etc. [2]. It is noteworthy that the Google Play app store alone hosts approximately 2.57 million Android applications of diverse types [3]. Users are also able to use applications from third parties in the Android operating system platform that misleads the users to download malicious applications from the attacker's server.

To develop an Android app developer has to add certain files into the package format with a .apk extension. AndroidManifest.xml is one among them that reveals various information about the app such as the version of the package, permissions it needs, intents, actions, services, etc.

The classes.dex file encapsulates the complete byte code that delineates the genuine functionality of the application. In upholding access control, Android utilizes a permission-based security model, meticulously granting applications only those permissions explicitly allowed by the user, thereby enhancing the overall security framework. To attain access, permissions must be explicitly listed in the AndroidManifest.xml file. Despite the incorporation of these security features, Android continues to face a diverse range of attacks, highlighting the persistent challenges in safeguarding the platform against evolving cybersecurity threats.

Most of the antivirus detection systems use signature-based detection that can be easily obfuscated by a malware writer by just changing the signature of the malicious application. A little obfuscation performed on a malware code can also evade detection. Zhan et al. [4] in their experiment used a patch technique where a piece of code is appended at a nonfunctional location of the code that resulted the malware file evade detection. Android malware detection is a vital issue for the android users. Numerous researchers have put forth various solutions aimed at mitigating Android malware attacks, reflecting ongoing efforts to enhance the platform's resilience against malicious threats. The feasibility of the detection model may be compromised if it does not incorporate a diverse set of features during the training process, underscoring the importance of exploring a wide range of characteristics for robust model development. Based on the literature survey in this work, it is observed that many detection frameworks are available but the frameworks often undergo training on a significantly limited number of features to mitigate model complexity. This in turn may affect the detection possibility of malicious application whose features are not considered at the time of feature selection in a particular dataset. Ignorance of such features may have an erroneous impact on the detection model. Instead, if a substantial no of features are considered as input features to train a model that may increase the detection possibility of a malicious application. Considering above problem in account, this proposed framework gives a solution on the subsequent following sections. This research is dedicated to developing a supervised deep learning framework using DNN techniques utilizing an array of static features identified through an extensive review of diverse research works. A comparative analysis is made by conducting several experiments using a substantial number of features as input.

The subsequent sections covers: the related literature survey are listed in Section 2, in Section 3 methodology of the proposed system is explained, proposed framework is

enlightened in Section 4, Section 5 outlines the experimental setup, result and discussion derived from the experiments are briefed in Section 6, Section 7 produce a comprehensive comparative analysis and finally Section 8 summarizes the work.

II. 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 Interface created by that 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 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 and 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 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 nonmalicious 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.

III. PROPOSED METHODOLOGY

For detection, the proposed framework uses six important features such as permissions, opcodes, API calls, system commands, activities, and services. Obtaining a representative dataset is a difficult task because of the access restriction by its researchers. In this paper OmniDroid dataset [16] is used for the experiments. Out of 25999 features 16380 distinct features

are considered for this experiment including 5500 Permissions, 224 Opcodes, 2128 API calls, 103 System Commands, 6089 Activities, and 2336 Services. The features are extracted from 11000 benign and 11000 malicious samples. The details about the features are explained below.

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 <uses-permission> of the AndroidManifest.xml file of the APK. Permissions play a crucial role in malware 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. Fig. 1 lists some permission extracted from a malicious app of Trojan type. It contains both normal and dangerous permission. As shown in the bellow figure 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].

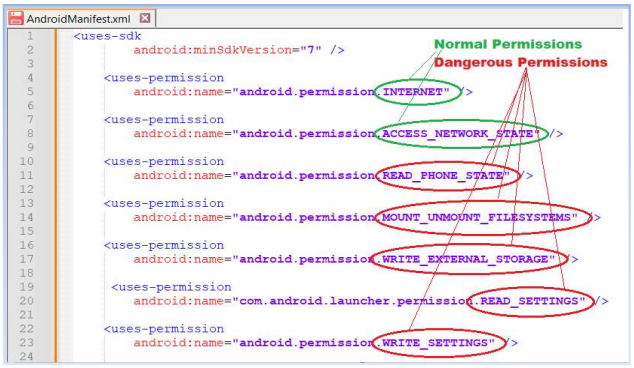

Fig. 1 Example of Normal and Dangerous Permissions

Opcodes: these are the readable instructions in a program that are executed by the Dalvik Virtual Machine (DVM). These opcodes are generated during the compilation process that represents the operations to be performed within the application. Fig. 2 is an example that shows the opcodes that can be extracted from Android applications. Many studies with good performance have been conducted so far utilizing opcodes [19] [20].

Fig. 2 Example of Opcodes

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. Fig. 3 shows an example of the services listed by a malicious app in its AndroidManifest.xml file.

Fig. 3 Example of Services

Activities: activities in Android applications are another component that operates in individual User Interfaces (UI) 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. Fig. 4 is an example to show the activities in an application.

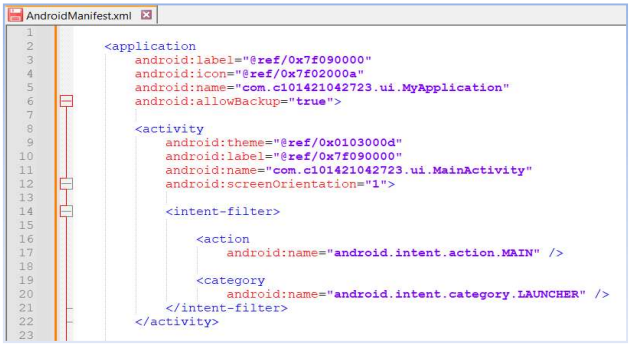

Fig. 4 Example of Activities

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 (Application Programming Interface) of the service provider. Fig. 5 shows the API 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].

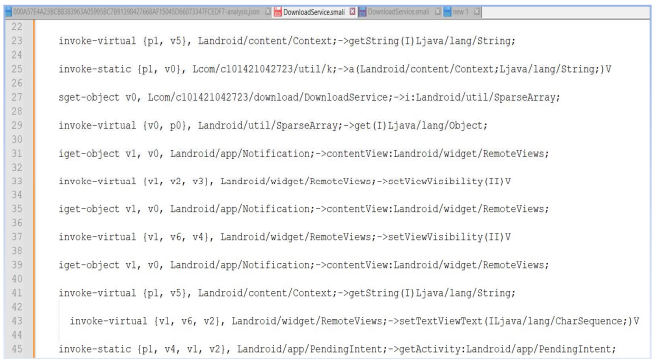

Fig. 5 Example of API Calls

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.

3.1. 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. Each feature's IG score is computed by subtracting individual feature entropies from the entropy of the output column, representing uncertainty in the given context. The calculation of IG scores for each feature follows Equation (1).

$$IG(f_i) = H(y) - H(f_i) \quad (1)$$

In Equation (1), IG(*fi*) is the score calculated using the IG technique of individual features that reflects the amount of information those features contain, is the feature column in the dataset, *i* represents the column number in the dataset, H (*y*) represents the entropy of output column, H (*)* 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 2 through 6 provide a comprehensive insight into the information content of various feature types. Fig. 2 represents the information content of permissions concerning the target variable. Fig. 3 represents the information content of services concerning the target variable. Fig. 4 represents the information content of opcodes concerning the target variable. Fig. 5 represents the information content of activities concerning the target variable. Fig. 6 represents the information content of API calls concerning the target variable and Fig. 7 represents the information content of system commands concerning the target variable.

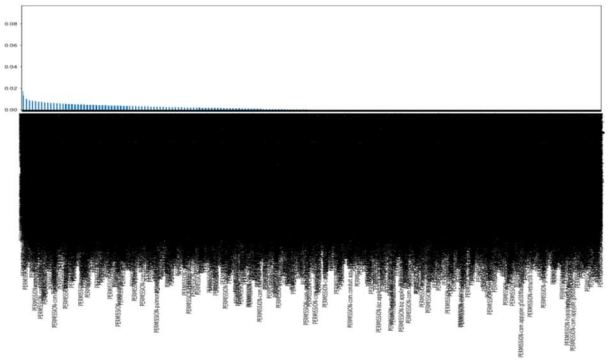

Fig. 2 ranking of the permissions in descending order.

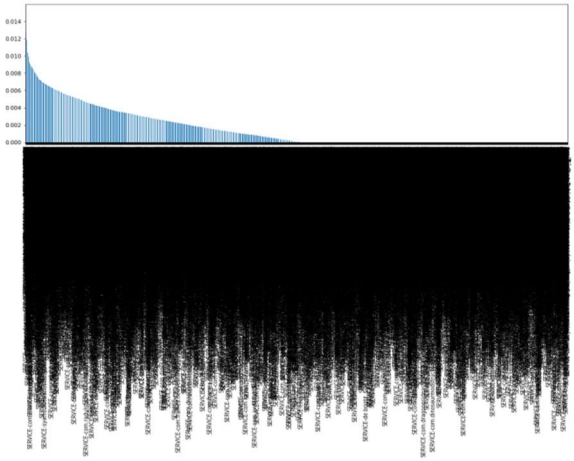

Fig. 3 ranking of the services in descending order.

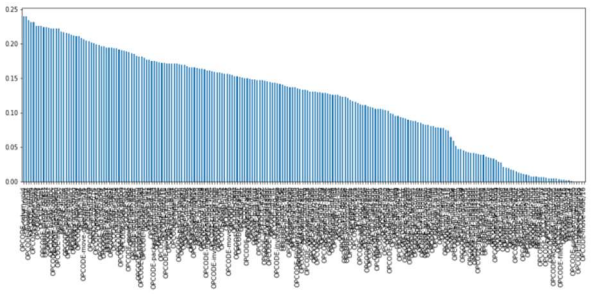

Fig. 4 ranking of the opcodes in descending order.

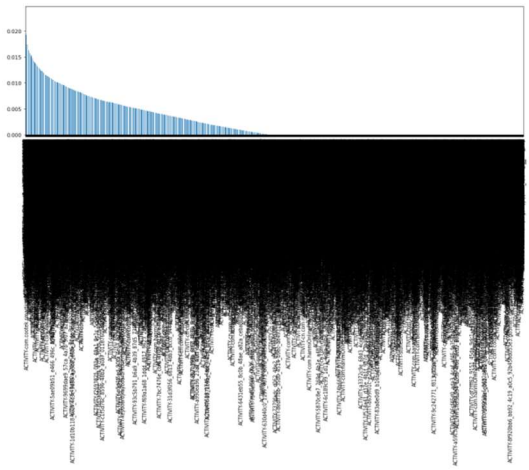

Fig. 5 ranking of the activities in descending order.

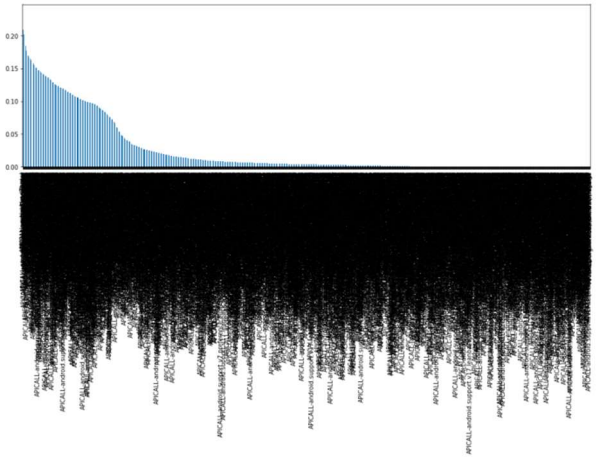

Fig. 6 ranking of the API calls in descending order.

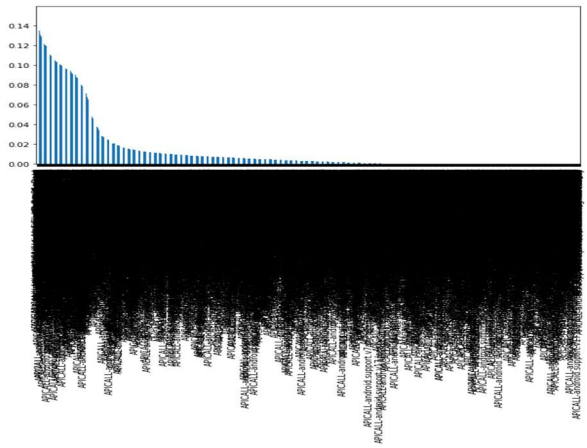

Fig. 6 ranking of the system commands in descending order.

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. Then IG is again applied to observe the considered data and calculated IG score for the considered features. Fig. 7 represents the IG score of the features and the features are plotted based on the score in descending order as shown in Fig. 8.

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

Fig. 7 IG score of the features

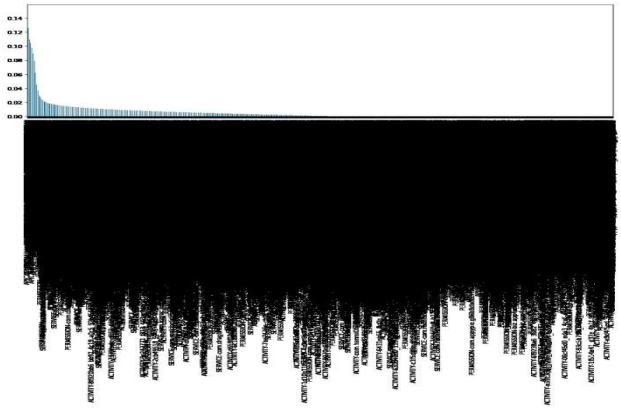

Fig. 8 Features plotted in descending IG score

IV. Proposed framework

This segment presents a framework using the DNN approach for Android malware detection. The DNN architecture encompasses the arrangement of an input layer, hidden layers, and an output layer. Through forward propagation, the data is computed, and subsequently, 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 present in the network process the input and the output is predicted by the neurons in the output layer.

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).

$$Z_n = \sigma((f_{sigmoid}\ (f_{relu}\ (l_{hidden}\ )_{i=1}^{n}))_{p=1}^{n})_b \qquad (2)$$

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

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

In equation (4); ‘i’ is the number of hidden layers the network contains, ‘p’ is the number of epochs, and Zn is the model output score.

Table 1: 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

Efficiently training an Artificial Neural Network (ANN) demands significant effort in the exploration and identification of 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. 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 1.

In this proposed model 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.

$$Loss\_with\_regularization = E + \lambda \sum_{k=0}^{n} w_i^2 \qquad (5)$$

In equation (5); E is entropy (the loss generated by the model), λ is a regularization constant ($\lambda > 0$), w_i represent the feature weights.

V. 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).

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

VI. 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 framework's effectiveness is determined through the analysis performed in the following sections.

4.1. 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 divided into three parts such as 75% of the whole data is used for training 15% for testing and 10% for validation. The model is trained using deep neural network techniques. Keras TensorFlow is used to deal with the overfitting of the model. Fig.2 describes the system architecture. To ensure a valid comparative analysis, the model parameters, as outlined in Table 1, remain consistent across all experiments conducted in this study.

Accuracy and Loss:

The experiments are done on the considered features without performing any feature selection mechanism on the dataset and accuracy and loss are measured. The model achieves a mean accuracy of 87.22%, as depicted in Fig. 2, while demonstrating a mean loss of 0.71 on the testing set, as illustrated in Fig. 3. It is observed that after a certain no of iterations, the model loss is not reducing anymore, it is floating between particular ranges from 0.60 to 0.71.

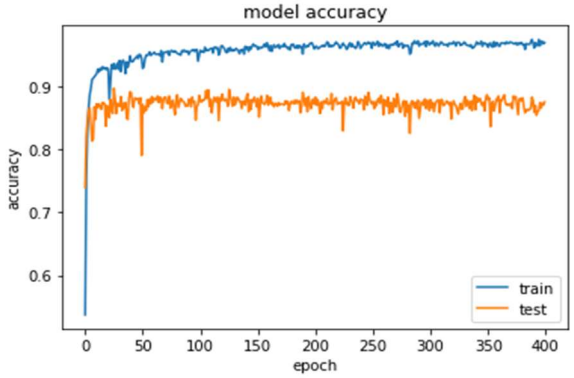

Fig. 2 : Model Accuracy Score Without Feature Selection

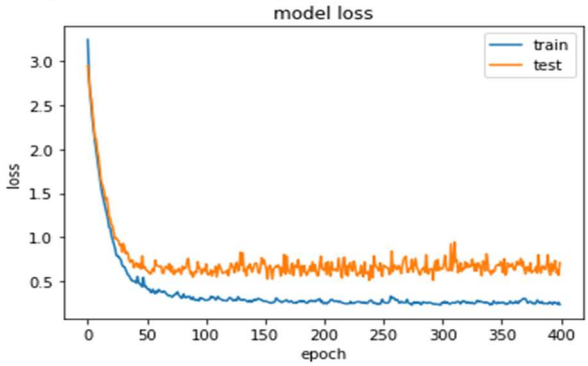

Fig. 3 : Model Loss Score Without Feature Selection

4.2. 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 50% of the total features have more or less contribution to the dataset and the rest of other features have very less or zero contribution to the dataset. Out of 16380 features 6552 features from the feature set are considered for classification purposes.

Accuracy and Loss:

Validation accuracy and validation loss are assessed on the feature-selected dataset containing 6552 features. The findings reveal a mean accuracy of 89.04% and a decreasing mean loss of 0.61 on the testing set following feature selection, as illustrated in Fig. 4 and Fig. 5.

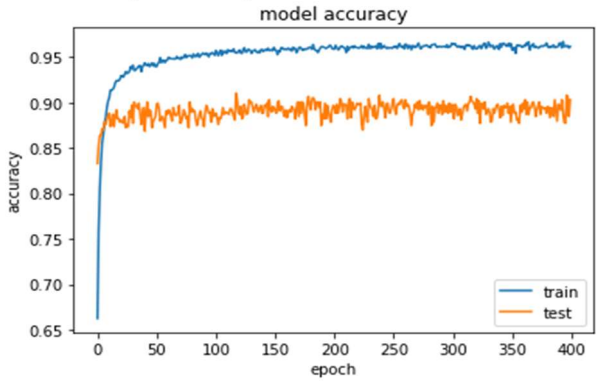

Fig. 4: Model accuracy score with feature selection

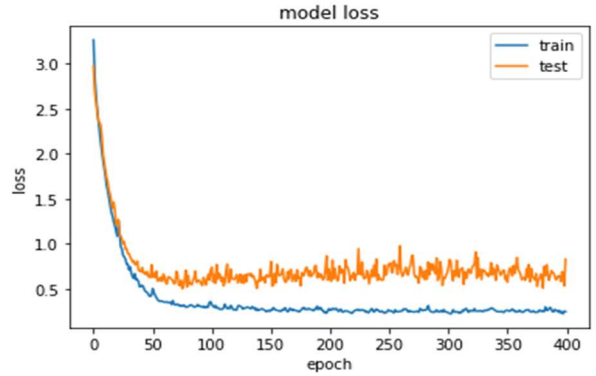

Fig. 5 : Model loss score with feature selection

VII. 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 Fig. 6 and Fig. 7. Accuracy and loss are monitored in both the training set and testing set.

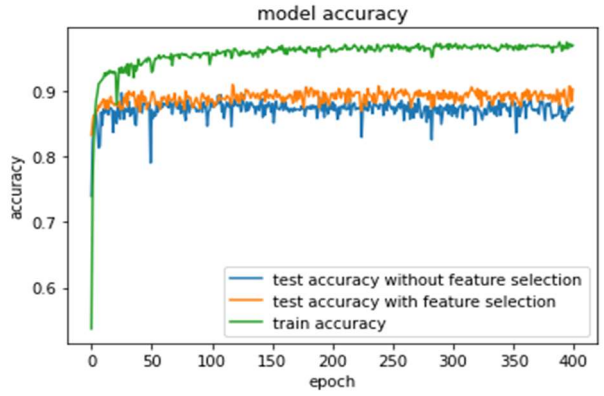

Fig. 6: Comparison of model accuracy score

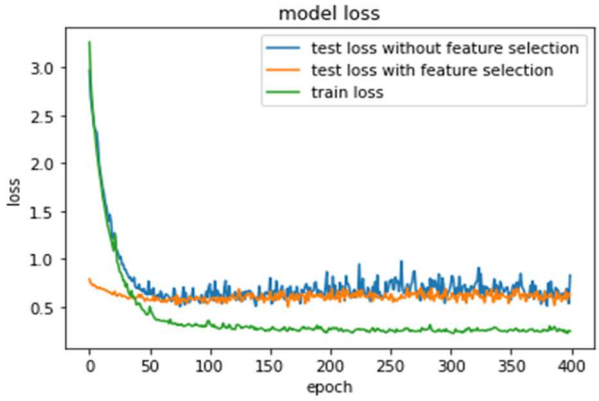

Fig. 7: Comparison of model loss score

Fig. 6 clearly illustrates that the accuracy of the model on feature selected dataset is improved from 87.22% to 89.04% with considering feature selection. 87.22% is the accuracy obtained by the model on the whole dataset without performing any feature selection. Similarly, Fig. 7 represents the model loss for both experiments. It is observed that the model loss is reduced from 0.71 to 0.61 with feature-selected data.

7.1 Time Comparison Between Both the Experiments:

Based on the statistics of Table 2; it is observed that model training time for the set of experiments with feature selection requires comparatively less time compared to experiments without considering feature selection as described in Table 3. Though the feature selection process takes a little extra time (10 minutes) as it is a one-time process this time can be ignored in the model evaluation process. In our experiment sufficient number of features is considered as compared to the other existing models in the literature survey as shown in Table 3. This implies the coverage of the most possible vulnerable holes for malware detection.

Table 2: Time Comparison between Both the Experiments.

Steps	With feature selection	Without feature selection
IG calculation	10 minute	Nil
Feature selection	10 minute	Nil
Model training	41 sec (100 epochs)	60 sec (100 epochs)
	92 sec (200 epochs)	130 sec (200 epochs)
	174 sec (400 epochs)	250 sec (400 epochs)
Model validation	1 sec	1 sec
Time taken for training	**5 minute 7 sec**	**7 minute 20 sec**
Total time taken	25 minute 8 sec	7 minute 21 sec

7.2 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 employing the same dataset. The performance comparison, outlined in Table 3, provides insights into the efficacy of the approach relative to other machine learning classifiers.

Table 3: Comparison with other Machine Learning models based on Accuracy

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

The above results show that all classification models employed in our research work effectively in detecting malware apps with a marginal difference. The RF classifier gives better accuracy compared to other considered ML classifiers. However, the proposed DNN model gives better results than the RF classifier because neural networks perform better on large amounts of training data to give better detection accuracy.

7.3 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. The result they obtained in their static detection models is quite impressive and the result obtained by our proposed model is also better reaching an accuracy of 89.04%, provided our proposed model is trained on a sufficient number of features as shown in Table 4. 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.

Table 4: 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]	840
Oliveira et.al [11]	200
Proposed framework	**6552**

VIII. 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 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 deep-learning methods.

References

[1] https://safeatlast.co/blog/mobile-malware-statistics/

[2] https://www.av-test.org/en/statistics/malware/

[3] 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, doi: 10.1109/TIFS.2023.3333567.

[5] M. Gopinath and S. C. Sethuraman, "A comprehensive survey on deep learning based malware detection techniques", Elsevier, Computer Science Review, 47 (100529), 2023, https://doi.org/10.1016/j.cosrev.2022.1005291574-0137/.

[6] Z. Wang, K. Li, Y. Hu, A. Fukuda and W. Kong, "Multilevel Permission Extraction in Android Applications for Malware Detection", International Conference on Computer, Information and Telecommunication Systems (CITS), 2019, IEEE.

[7] M. ALTAİY, İ. YILDIZ and B. UÇAN, "Malware Detection using Deep Learning Algorithms ", Aurum Journal of Engineering Systems and Architecture, Volume 7, No 1, 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", Elsevier, 2020, https://doi.org/10.1016/j.comnet.2020.107639.

[9] N. C. Le, T. M. Nguyen, T. Truong, N. D. Nguyen and T. Ngo, "A Machine Learning Approach for Real Time Android Malware Detection", IEEE Xplore, 2020 10.1109/RIVF48685.2020.9140771.

[10] C. Rodrigo, S. Pierre, R. Beaubrun and F. E. Khoury, "BrainShield: A Hybrid Machine Learning-Based Malware Detection Model for Android Devices", J. Electronics, 2021, https://doi.org/10.3390/electronics10232948.

[11] A. S. Oliveira, R. J. Sassi, "Chimera: An Android Malware Detection Method Based on Multimodal Deep Learning and Hybrid Analysis", TechRxiv, Preprint, 2020, https://doi.org/10.36227/techrxiv.13359767.v1.

[12] X. Gao, C. Hu, C. Shan, W. Han, "MaliCage: A packed malware family classification framework based on DNN and GAN", Journal of Information Security and Applications,Volume 68, 2022, 103267, ISSN 2214-2126, https://doi.org/10.1016/j.jisa.2022.103267.

[13] H. Li, et al., "MalCertain: Enhancing Deep Neural Network Based Android Malware Detection by Tackling Prediction Uncertainty," in 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), Lisbon, Portugal, 2024 pp. 934-934. doi:https://doi.ieeecomputersociety.org/

[14] M. Aamir, M. W. Iqbal, M. Nosheen, M.U. Ashraf,A. Shaf, K.A. Almarhabi, A.M. Alghamdi, A.A. Bahaddad, "AMDDLmodel: Android smartphones malware detection using deep learning model." PLoS One, 2024, doi: 10.1371/journal.pone.0296722.

[15] A.R. Nasser, A.M. Hasan, A.J. Humaidi, "DL-AMDet: Deep learning-based malware detector for android Intelligent Systems with Applications", Volume 21, 2024, 200318, ISSN 2667-3053,https://doi.org/10.1016/j.iswa.2023.200318.

[16] http://aida.etsisi.upm.es/download/omnidroid-dataset-csv-features-v1/

[17] K. Khariwal, J. Singh and A. Arora, "IPDroid: Android Malware Detection using Intents and Permissions," 2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4), London, UK, 2020, pp. 197-202, doi: 10.1109/WorldS450073.2020.9210414.

[18] F. Akbar, M. Hussain, R. Mumtaz, Q. Riaz, A.W.A. Wahab, K.H. Jung, "Permissions-Based Detection of Android Malware Using Machine Learning", Symmetry, 2022, 14(4):718. https://doi.org/10.3390/sym14040718.

[19] B. Kang, S. Y. Yerima, K. Mclaughlin and S. Sezer, "N-opcode analysis for android malware classification and categorization," 2016 International Conference on Cyber Security and Protection of Digital Services (Cyber Security), London, UK, 2016, pp. 1-7, doi: 10.1109/CyberSecPODS.2016.7502343.

[20] G. Canfora, A. De Lorenzo, E. Medvet, F. Mercaldo and C. A. Visaggio, "Effectiveness of Opcode ngrams for Detection of Multi Family Android Malware," 2015 10th International Conference on Availability, Reliability and Security, Toulouse, France, 2015, pp. 333-340, doi: 10.1109/ARES.2015.57.

[21] J. Jung; H. Kim, D. Shin, M. Lee, H. Lee, S.J. Cho, K. Suh, "Android Malware Detection Based on Useful API Calls and

Machine Learning," 2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE), Laguna Hills, CA, USA, 2018, pp. 175-178, doi: 10.1109/AIKE.2018.00041.

[22] D.J. Wu, C. -H. Mao, T.E. Wei, H.M. Lee and K.P. Wu, "DroidMat: Android Malware Detection through Manifest and API Calls Tracing," 2012 Seventh Asia Joint Conference on Information Security, Tokyo, Japan, 2012, pp. 62-69, doi: 10.1109/AsiaJCIS.2012.18

[23] https://www.tensorflow.org/

[24] https://keras.io/getting_started/

[25] https://scikit-learn.org/stable/