ITC 4/54
Information Technology and Control
Vol. 54 / No. 4/ 2025
pp. 1358-1382
DOI 10.5755/j01.itc.54.4.35472

Android Malware Detection Using Static Feature Analysis and Deep Learning Techniques

Received 2023/10/31 | Accepted after revision 2025/01/22

HOW TO CITE: Akbar, S., Khan, A., T. (2025). Android Malware Detection Using Static Feature Analysis and Deep Learning Techniques. *Information Technology and Control*, 54(4), 1358-1382. https://doi.org/10.5755/j01.itc.54.4.35472

Android Malware Detection Using Static Feature Analysis and Deep Learning Techniques

Saima Akbar
Department of Software Engineering, Bahria University, Islamabad, Pakistan, e-mail: 01-241212-008@student.bahria.edu.pk

Tamim Ahmed Khan*
Department of Software Engineering, Bahria University, Islamabad, Pakistan, e-mail: tamim@bahriah.edu.pk

Corresponding author: tamim@bahriah.edu.pk

The increasing use of Android mobile devices and applications leads to an increase in malware threats. There is a requirement to investigate if a more detailed feature extraction from APK files with deep learning can produce more accurate results. We investigate using deep learning techniques to detect Android Malware considering the latest datasets. We aim to improve the system's ability to accurately classify and detect a wider range of Android malware variants. We propose a mechanism to carry out APK analysis for feature extraction capable of extracting 46,648 features. We retain 10,523 features after applying feature selection and subsequently use these selected features to train the neural networks. We make use of APK retrieved from Androzoo for dataset generation. We contribute a dataset with code and scripts to arrive at our proposed dataset using a public repository. We compare deep learning models based on deep neural networks (DNN), convolutional neural networks (CNN), and transfer learning-based models using static features. We consider our contributed datasets and conclude that the DNN-based models outperform the CNN models with a wider range and number of features.

KEYWORDS: Malware detection, Machine learning, Deep learning, Transfer learning, Deep Neural Networks, and Convolutional Neural Networks.

1. Introduction

Everyday use of mobile devices, particularly Android operating system devices, has increased with time. Malware attacks on these devices are becoming increasingly common. In malware, the attacker develops a program to damage a computer system without the user's consent. For Android mobiles, third-party app stores such as Google Play Store have recently been penetrated by malware, which poses serious privacy and security risks for users [9], [16]. Over 2.56 million mobile applications (Apps) were available for download in the Google Play Store alone as of the first quarter of 2022 [27].

We can make use of datasets containing static or dynamic features of an application to determine whether an Android App is possibly benign or malicious. The datasets that are based on static features consider the Android Package Kit file (APK) of the application for extracting feature-related data without executing the application. Examples of static features include permissions, API calls, strings, etc. Static feature analysis techniques have emerged as an approach to identifying and classifying malware without executing the applications. By extracting features from the binary code or manifest files, these techniques offer valuable insights into the potential malicious behavior of Android Apps. Dynamic analysis techniques, on the other hand, involve executing Apps in controlled environments to observe their behavior in real-time.

It is pertinent to note that prior research focused either on binary classification [11] or multi-classification considering a limited number of classes or sub-classes of Android malware [28], [12]. Since we discover new classes of Android malware with time, it becomes imperative to consider additional classes for the development of prediction models. A zero-day attack, also known as a zero-day exploit, is a type of cyberattack that takes advantage of a previously unknown vulnerability. These vulnerabilities are called "zero-day" because they are exploited by attackers before the software developer becomes aware of the issue, leaving zero days for the developer to prepare and release a patch or fix and provide safeguards against the newly emerged vulnerability [28]. There is a need to incorporate techniques such as transfer learning to effectively address zero-day attacks [12]. We also note that a limited number of features are considered for the development of models [15], [29]. Static feature extraction techniques extracting up to a maximum of 19000 features and applying pre-processing returning 350 features is reported in [11]. APK development, code complexity, and feature extraction techniques for static feature extraction as well as simulator-based execution for dynamic feature extraction became more advanced over time and hence we can extract a larger number of features. This helps in creating datasets with more classes and features which can assist in developing models with more diverse and relevant examples enhancing their generalization capabilities [14]. We contribute the following:

We consider a new or unknown class, in addition to a list of malware classes, and demonstrate the use of transfer learning techniques to make the models capable of handling zero-day attacks.

We provide a feature extraction mechanism capable of extracting 43,376 features from APK analysis.

We extract features including Application Tags, Feature Tags, Activity Count, Library Tags, Metadata Tags, Permission Tags, Provider Tags, Receiver Tags, and Service Tags to contribute a dataset with 10,523 features.

We propose models based on Convolutional Neural Network (CNN) and Deep Neural Network (DNN) based models that consider more malware classes and more features.

This paper is organized as follows. We present a literature review in Section 2 and provide a comprehensive overview of our proposed methodology, dataset, and pre- processing in Section 3. We present our experimental design and performance evaluation metrics in Section 4 and, finally, present a conclusion in Section 5.

2. Related Work

We report studies implementing CNN, DNN, or Transfer learning-based models. We consider machine learning and deep learning-based Android malware detection techniques with special reference to

models, datasets, and claimed model accuracy. The authors in [18] proposed a machine learning-based approach for Android malware detection, utilizing a static permission-based methodology. Their approach shares similarities with Drebin in terms of being lightweight and computationally efficient. The paper included four main experiments: Permission-based clustering, Permission-based classification, source code-based clustering, and source code-based classification. For conducting their research, the authors used a dataset comprising 400 applications, equally split into 200 benign and 200 malicious samples. To improve the accuracy and reliability of the results, an Ensemble learning technique was employed, consisting of an odd number of classifiers. This allowed for a more robust determination of outcomes based on the probabilities generated by each model.

Authors in [15] enhance the efficiency and reliability of Android malware detection by focusing solely on the permission feature and employing binary classification with an imbalanced dataset sourced from the M0Droid dataset. The study evaluates the performance of commercial anti-virus tools in classifying samples as malicious or benign. The findings indicated low detection rates, with only 14.37% of the 5902 malicious samples correctly identified, and a false positive rate of 18.4% observed on the 4297 benign samples. Furthermore, the research explored the effectiveness of various machine learning (ML) algorithms using only the APK manifest file for analysis. Notably, all ML algorithms outperformed commercial anti-virus engines. A literature review is presented in [4] that explored the application of deep learning techniques for Android malware detection, specifically focusing on static features. In the study, a dataset of 426 malware and 5,065 benign samples was utilized, and these samples were categorized into Ransomware, Adware, SMS Malware, and Scareware. The research employed the BiLSTM model, which demonstrated an exceptional accuracy of 98.85% on the CICInvesAndMal2019 dataset containing 8,115 static features. Notably, the selected features, including Permissions, Activities and Services, Broadcast Receivers, Meta-data, API calls, System calls, and Opcode, played a vital role in contributing to the improved detection of Android malware.

Automatic Modulation Classification (AMC) is pivotal for cognitive radios, enabling adaptive modulation and demodulation. Our proposed Deep Learning Prior Regularization (DL-PR) method optimizes loss during model training by augmenting inter-class distances and minimizing intra-class ones based on SNR distribution [31]. Validated on RadioML 2016.10a dataset, DL-PR outperforms state-of-the-art methods, offering robustness to hyper-parameters and SNR estimation for practical applications.

In the backdrop of challenges posed by unauthorized broadcasting in complex electromagnetic environments, our study delves into introducing the manifold regularization-based deep convolutional autoencoder (MR-DCAE) model for effective identification. The MR-DCAE utilizes a specially designed autoencoder optimized by entropy-stochastic gradient descent, with a focus on reconstruction errors during testing to discern authorized signals. We propose adding this paper [32] to elaborate on advanced techniques like MR-DCAE, offering insights into cutting-edge methods for addressing unauthorized broadcasting issues. Automatic modulation classification (AMC) is crucial in both civilian and military contexts. The proposed multi-scale radio transformer (Ms-RaT) with dual-channel representation offers a refined approach to fine- grained modulation classification (FMC). By integrating dual-channel representation (DcR) and multi-scale analysis, Ms-RaT demonstrates superior classification accuracy com- pared to existing methods, as validated through extensive simulation results. The author in [5] focused on multi- classification, considering adware, ransomware, Scareware, and SMS malware, while utilizing 19 selected features. The study employed the Long Short-Term Memory (LSTM) algorithm for improved ransomware detection in the Android environment. The proposed deep learning-based malware detection model was evaluated using the CI-CAndMal2017 android malware dataset and standard performance parameters. Remarkably, the proposed algorithm achieved an outstanding detection accuracy of 97.08%. Based on these impressive results, the proposed algorithm was endorsed as an efficient approach for malware.

Malware detection using feature extraction is proposed in [21] and authors claim promising results where the authors in [20] propose a dynamic machine-learning algorithm for content-aware healthcare data. The authors in [3] stress the need to classify

and detect adversarial agents that may compromise the security of cyber systems and agents whereas the authors in [26] provide a comprehensive account of such adversarial agents and proposals that consider obfuscated samples to train the models. of people, organizations, and countless other forms of digital assets, we find proposals where the authors consider network

Such as 5G networks [6] and obfuscated samples using Siamese networks [1] for Malware classification. The authors worked on binary Android malware detection in [30]. The proposed Deep Classify Droid detection system was a deep learning-based approach focused on distinguishing malicious and benign Android applications. The system utilized CNN-based malware detection and achieved an impressive accuracy rate of 97.4%. Comparative evaluations demonstrated that DeepClassifyDroid outperformed most existing machine learning-based methods, accurately detecting 97.4% of malware with minimal false alarms. Additionally, the approach showcased exceptional efficiency, being 10 times faster than linear SVM and 80 times faster than kNN. The evaluation dataset consisted of 5546 malware and 5224 benign software samples from the Drebin dataset, underscoring the effectiveness and efficiency of the Deep Classify Droid detection system for binary Android malware detection. We find a static analysis-based Android malware detection model proposed using features from benign and malicious apps collected from Google Play and Virus Share in [33]. The model utilized a fully connected deep learning approach (DNN) and achieved an outstanding accuracy of approximately 94.65%. The dataset included 331 features with classifier labels, focusing on binary and multi-label categorical data, particularly permissions in API, which were often misused by hackers. The study also identified malware and benign applications, contributing to a safer user experience on Android devices.

The author used a novel machine learning model and considered the co-existence of static features aimed at detecting Android malware [22]. Considering the permissions and APIs compared to benign applications. To validate this hypothesis, the study constructs a new dataset, examining co-existed permissions and API calls across various levels of combinations. Employing the frequent pattern growth (FP-growth) algorithm, relevant co-existed features are extracted from a range of Android APK samples. The proposed model undergoes evaluation using conventional machine learning algorithms, showcasing its efficacy in accurately classifying Android malware. Notably, the Random Forest algorithm achieves exceptional performance, achieving a peak accuracy of 98% when considering co-existed permissions features at the second combination level.

We compare our results with the state-of-the-art models. With the rise of mobile technology, Android has become a prime target for attackers. Despite existing literature on Android malware detection, there's a gap in utilizing attribute selection methods. This study introduces a machine learning-based detection system aiming to distinguish Android malware from benign apps. Employing a linear regression-based feature selection approach, unnecessary features are eliminated to reduce dimensionality and enhance real-time detection capabilities. Results indicate a peak F-measure of 0.961 with at least 27 features.

We show the existing literature using static analysis of applications in Table 1. It is pertinent to note that there are numerous Android malware families and their sub-classes, and new malware is introduced regularly. However, existing research considers at most 10 classes and sub-classes [8]. Therefore, a deeper APK analysis capable of revealing more static features for constructing datasets and deep learning models considering a wider range of malware classes and sub-classes for the Android malware detection process is required. We aim to deploy a feature extraction mechanism for dataset development. We also propose deep-learning techniques-based models considering more classes and static features and use deep-learning for Android malware detection techniques handling zero-day attacks. We present in the following sections how can we construct datasets with larger sets of examples by considering more features and families/classes and how can we develop deep learning approach-based models that make use of a bigger range of malware classes and features in the dataset, and improve the efficiency of existing systems. We finally provide a comparison of deep learning-based classifiers that can be employed to identify malware with and without the possibility of handling zero-day attacks.

3. Our Approach

We use applied research design principles and provide research design including data collection, measurement, and analysis steps. We develop a mechanism for feature extraction and develop a dataset. We make this dataset publicly available at Kaggle[1]. We show in Figure 1 the methodology used for conducting this research.

We present an overview of existing datasets in Table that allow users to extract filtered information in the form of processed datasets containing examples and labels. However, it is important to highlight that they contain a limited number of features e.g., the Drebin dataset offers 215 features and it becomes a motivation for us to develop a feature extraction mechanism that we introduce in Subsection 3.1 extracting "10523" features. We distinguish between malware categories and malware families in our context. Malware categories categorize malicious software based on their general behavior, while malware families serve to group together related variants that share common characteristics or origins within the Android ecosystem.

This distinction forms a foundational aspect of our research as we carefully select and analyze datasets for our study.

Table 1
Selected Previous Studies for Android Malware Detection

Ref.	Dataset	Classification	Classifiers	Features	Limitation
[4]	CICInves-AndMal2019	Multi-class (Ransomware, Adware, SMS Malware, Scareware and Benign)	BiLSTM	Permissions, Activities, Services, Broadcast, Receivers, Metadata, API calls, System calls	Considers only 4 malware classes i.e Ransomware, Ad- ware, SMS Malware, Scare- ware and Benign.
[11]	Androzoo and AMD	Multi-class (Benign, Trojan, and Backdoor)	CNN-BILSTM	permissions, API calls, and intents	Considers only 2 malware classes only and considers a limited number of features e.g. permissions, API calls, intents
[11]	Drebin and CICMal-droid2020	Multi-class (Adware, Banking, Benign, Riskware, and SMS)	1D-CNN, RF models	permissions, intents, services, and API calls	Considers 4 classes only
[7]	CICAnd-Mal2017	Multi-class (Adware, Radware, rootkit, SMS malware, and ransomware)	DNN	Not mentioned	Considers only 5 classes only
[23]	Private Dataset	Binary Class (Malware, Benign)	CNN	Manifest file, OpCodes, N-flrams, Permission and API Calls	Focuses on binary classification.
[2]	flenome, Drebin and private dataset	Binary Class (Malware, Benign)	Random Forest	Permission-based features	Considers permission feature limiting the dataset to only Android permission features, overlook behavioral, struc- tural, and code-based indi- cators.

[1] https://www.kaggle.com/datasets/hiraak27/samdroid-dataset

Ref.	Dataset	Classification	Classifiers	Features	Limitation
[13]	Virus Share, Maltrieve and private Dataset	Binary Class (Malware, Benign)	SVM	API call sequences and system call	Does not provide a break-down of specific malware families or classes included in the dataset, only consider the API call and system calls.
[28]	Ember, VirusTotal, VirusShare and Private Dataset	Multi-Class (Dailer, Backdoor, worm, trojan, wormautoit, trojan, downloader, rouge and pws)	Random Forest (RF), SVM, CNN, DNN	Systemcalls	Considers only the system calls.
[17]	Drebin, AMD, VirusShare, floogle Play	Multi-class (Trojan, backdoor, worms, botnet, spyware classes and benign)	BiLSTMs	Permissions API calls	Considers a limited number of features for analysis, only the Permission and API Calls and only 5 malware classes.
[10]	Not Mentioned	Multi-class (10 classes)	SVM	Permissions and API calls and Intent	Lacks details about datasets used
[17]	Private Dataset	Binary class (malware and benign)	flAN and quantum support vector machine (QSVM)	Not Mentioned	Limited information features used and works on binary classification.
[22]	Private Dataset	Binary class (malware and benign)	Random Forest	API calls, Permissions	Limited features used.
[33]	Private Dataset	Binary class (malware and benign)	KNN, Naive Bayes (NB), Sequential Minimal Optimization (SMO), MLP, RF, C4.5 and Logistic Regression (LR)	Not mentioned	Only 27 features used.

Figure 1
Methodology.

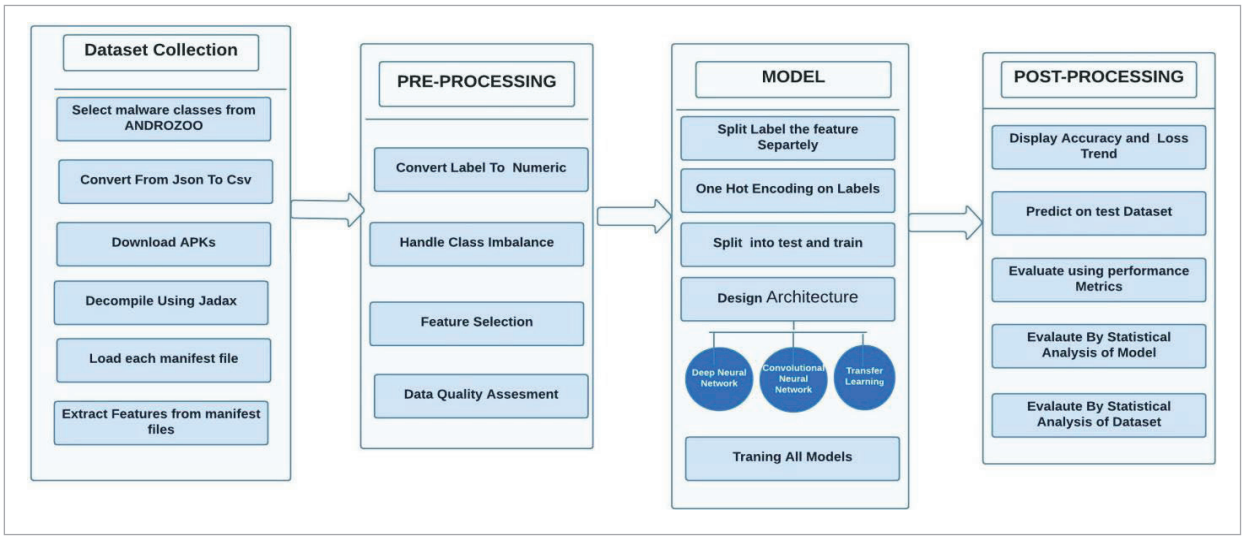


Table 2
Prominent Public Repositories for Android Malware Detection.

Repositories	Link	No. of APK (examples)
Google play	https://support.google.com/googleplay/?hl=entopic=3364260	3.553 Million
Androzoo	https://androzoo.uni.lu/	23 Million
MARVIN	https://www.insightplatforms.com/platforms/marvin/	135 Thousand
Virus share	https://virusshare.com/	69 Million
Contagio	https://zeltser.com/malware-sample-sources/	10 Thousand

It is pertinent to note that there are two types of data origins for Android malware detection considering machine-learning or deep-learning techniques. The first type of origin is where we find repositories of APK files that we can use for accessing APK files for feature extraction and dataset development. We present information about such repositories in Table 2. The second type of origin is where we find extracted features and labeled data in the form of datasets such as CICmal2020, CICmal2019, and CICmal2017, etc. We present information about such repositories in Table 2. It is further to note that only CICmal2020 provides information about malware classes and families. One of our objectives was to consider a wider number of classes with more features than what is available in currently available datasets. Therefore, we retrieved information about APK files from Androzoo[2] which is a publicly available APK repository and wider range of features and classes.

Using deep learning techniques for Android malware detection has the following motivating factors: Present malware includes obfuscation and other methods, making it difficult to identify. This necessitates complex approaches capable of handling large and diverse data sets to identify malicious behavior. Deep learning techniques are effective in dealing with big data, which is crucial due to the numerous apps and possible malware versions. These models can be trained from large data sets, improving accuracy and reducing sensitivity to noise. The ability to learn from new and unknown malware (zero-day attacks) is especially useful. Techniques like transfer learning can efficiently update models, enhancing their ability to identify new types of malware. Deep learning models can be extended to accommodate a large number of features and different classes of malware, enhancing the models' generalization capabilities. This scalability is crucial in ensuring comprehensive protection against various types of malware.

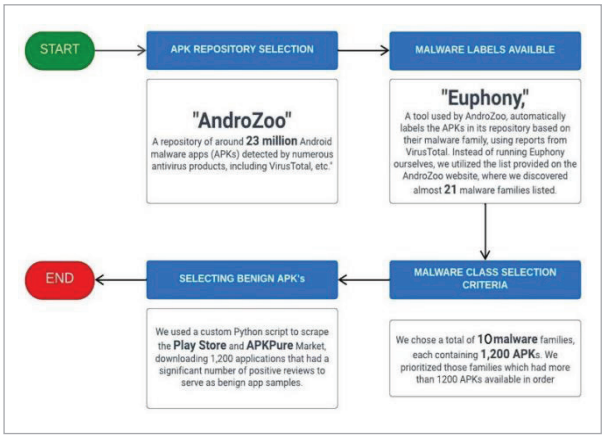
These techniques address the challenges associated with the growing number of malware threats on Android devices by offering a more precise, efficient, and flexible approach to malware identification.

3.1. Dataset Development

We located seven malware classes including Trojan, Banking, Backdoor, SMS Trojan, Ransomware, Adware, and Riskware from the Androzoo repository. We selected the remaining malware classes only if we were able to obtain and download a specific number of APK files from the Androzoo repository. We found that we would be able to download "1,200" APK files available in the Androzoo repository. We included Spyware, Monitors, and Exploits.

To identify potential zero-day threats, we utilized the K-means algorithm to detect patterns and identify such APK files from the available dataset. We chose a

Figure 2
Selection of Classes.



[2] https://androzoo.uni.lu/

total of ten malware classes from Androzoo, specifically focusing on those classes that consist of more than "1,200" APK files. Our reason for this selection is the limitation of free access to Androzoo, which provides access to only 1 million APK out of a total of 23 million APK files available with Androzoo. Our reason for choosing the figure of "1,200" is purely based on a number that can be extracted from free samples such that we do not need to carry out any class balancing step.

We also contribute our Python codes and algorithms so that users can enhance the number of examples keeping a broader list of features. We select malware classes that we want to collect data on. We followed a series of steps to determine which classes we utilize to train our model as shown in Figure 2. We download APK files from Androzoo, a repository with over a million apps including malicious ones to construct the dataset. We make use of a command-line tool Euphony [19], which is created to infer a single label per malicious application from VirusTotal reports. Every entry in the Androzoo records has a column called "VT Score," which shows how many times VirusTotal has reported the application, along with a malware class label. We focus on specific malware classes required for the research to ensure diversity and relevance. We select the top 1200 APKs with the highest VT scores within these particular malware classifications.

We make sure that our selected examples contain a wider variety of Android malware classes by selecting the most significant and identifiable samples. We do feature extraction first and then do feature selection. We write a script to download and decompile these APKs and extract the manifest files individually. Additionally, this script organizes the data for later study by compiling it into a CSV file and reading particular tags from the manifest files. This procedure made sure the dataset represented a broad range of Android malware.

3.1.1. Selecting Required Links into a CSV File

We develop a CSV file that contains the SHA256, Malware Class and APKs name.

1. **Searching for malware classes:** We search for them in the web databases of VirusTotal and Androzoo after determining the malware classifications. We found a large number of malware samples for our study from these databases.
2. **Converting data to CSV format:** We develop a Python script to convert the data from the Virus Total JSON format to CSV format, as shown in Algorithm 1. We obtain data such as the virus's name, file type, file size, and number of detections as a result of this step. We use a Python script that uses the Apktool tool to decode the apps and get the AndroidManifest.xml files out of these samples to get the APK files. We extracted the AndroidManifest.xml files from the samples of malware and stored them in a separate directory. The process is repeated for each Android malware class including benign samples.
3. **Organizing the data:** Finally, we organize the data by dividing the CSV file according to the various malware types selected. We achieve this using another Python script. We save the data in the form of CSV file locally for developing the dataset.

3.1.2. Data Extraction from APK Files

We provide the physical location of the APK file sources of a benign and malware nature which we have stored in the CSV file in the previous step. The next set of steps is:

1. **Load CSV of each malware class:** We need to load each CSV file into memory to extract the APK links After splitting the CSV file based on malware types. This process involves reading the CSV files and getting the data that is needed by processing.
2. **Download APK Files for Processing** After getting the relevant information we create a loop to download the requested amount of APKs. This loop will run over the number range supplied by the user. Each iteration of the loop will involve downloading an APK from the specified malware class using its corresponding link from the previously loaded CSV file.
3. **Decompile downloaded APK using JADX:** After downloading the APK, we need to decompile it using a tool such as JADX to obtain the source code. This is necessary in order to investigate the code for future research.
4. **Find and move the Android manifest file:** After decompiling the APK, we need to find and extract the Android manifest file. This file provides essential APK information, such as its components, permissions, and services. This file will be moved to a separate directory for future use for feature gathering.

Table 3
Prominent Datasets for Android Malware Detection.

Dataset	Link	Classes
Drebin	https://www.sec.cs.tu-bs.de/danarp/drebin/	They do not mention names of classes which they consider
AMD	https://ieee-dataport.org/documents/data-set-android-malware-detectionfiles	Not mentioned Clearly
CICAndMal2017	https://www.unb.ca/cic/datasets/and-mal2017.html	Adware, Ransomware, Scareware, SMS Malware
Genome	http://www.malgenomeproject.org/	Not clearly mention
Ember	https://paperswithcode.com/dataset/ember	Not mentioned
CICInvesAndMal2019	https://www.unb.ca/cic/datasets/invesand-mal2019.html	Adware, Ransomware, Scareware, SMS Malware
Microsoft Malware Classification Challenge	https://paperswithcode.com/dataset/micro-soft-malware-classification-challenge	They do not mention name of classes
Malgenome	http://www.malgenomeproject.org/?fb-clid=IwAR1xhnvHsP3CcrYC va8kreKvwfps-Mwji67kEpIy7x-RvLRU4LPpwaxpjlPA	They also do not mention the name of classes or families what they have consider
Omnidriod	https://aida.etsisi.upm.es/download/omni-droid-dataset-v1/	Not clearly mentioned
CCCS-CIC-And-Mal-2020	https://www.unb.ca/cic/datasets/and-mal2020.html	Adware, Backdoor, File Infector, No Category, PUA, Ransomware, Riskware, Scareware, Trojan, Trojan-Banker, Trojan- Dropper, Trojan-SMS, Trojan-Spy, Zero- day

The steps for downloading APK files considering CSV provided by repositories are presented in Figure 3.

We write a Python script to separate each malware class in a separate CSV file, as shown in Figure 4 after converting the JSON to CSV. We present the process of downloading malware APK in Figure 5.

Algorithm 1: Convert JSON to CSV

Input: json_filename, csv_filename
Convert_JSON_to_CSV(json_filename, csv_filename)
1: data ← OPEN json_filename and LOAD its content
2: OPEN csv_filename for writing AS csv_file
3: writer ← INITIALIZE CSV WRITER for csv_file
4: WRITE header to csv_file: 'SHA256', 'Malware Class', 'Malware Class Name'
5: **for** each key, value in data DO **do**
6: malware_class_name ← value +' *_application.apk*'
7: WRITE to csv_file: key, value, malware_class_name
8: **end for**
9: CLOSE csv_file

Figure 3
Collection of Malware Classes.

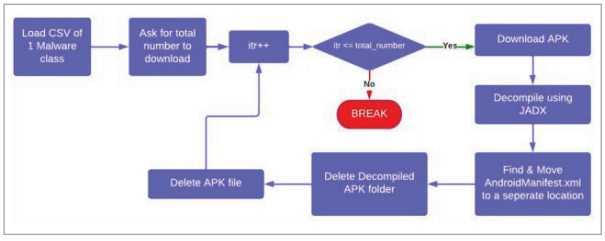


Figure 4
Process of Malware APKS Download.

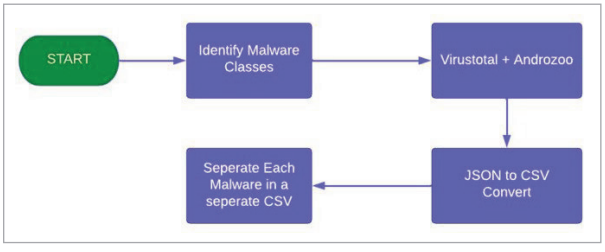


Table 4
Considered Features for Dataset Creation.

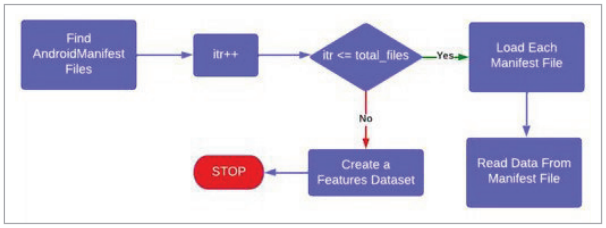
Application	The main component of an APK containing code and resources required to run the app.
Libraries	Pre-built code modules used by the app to add functionality or reduce development time.
Receivers	Components that receive and handle messages or events from other apps or system components.
Providers	Components that manage access to a structured set of data, used to share data between Apps or provide access to data stored in a database.
Meta Data	Additional information about the app, such as the version number, developer information, and licensing information.
Permissions	Security settings that control access to system resources or user data, required for certain App functionality such as accessing the camera or microphone.
Services	Feature that executes in the background and perform long-running operations, such as playing music and downloading files, etc.
Features	That provide additional functionality to the applications, such as support for specific software or hardware features.
App ID	A unique identifier for the application used to distinguish it from other applications on the devices.
Activity Count	The number of activities is in the applications.
Labels	Names are given to various components of the applications, for example, activities, providers, and services, It is utilized to identify and make them more easily understandable and readable.

We gather the information of the features that are mentioned in Table 4 from each APK.

3.1.3. Feature Extraction from Selected APK Files

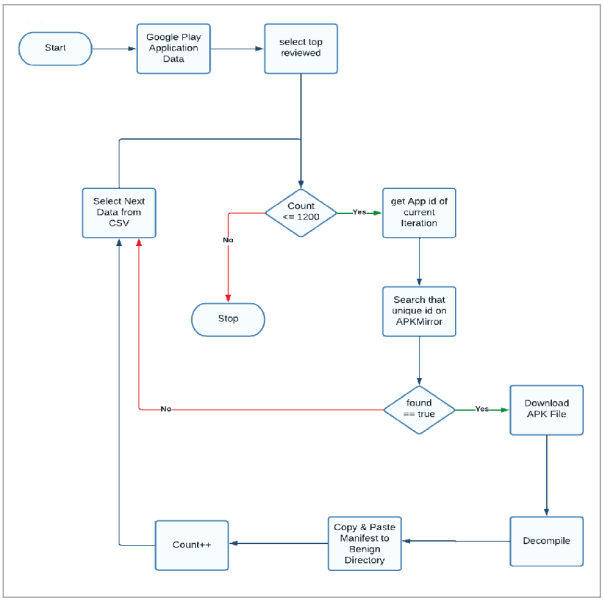
We note that the feature extraction process is complete once all the files have been processed. Figure 5 shows the process of malware feature extraction.

Figure 5
Malware Feature Extraction.



We develop a benign malware sample collector as well which is shown in Figure 6. We designed a web scrapper to scrape through the Google Play store and other third-party App stores to acquire benign samples. We present the steps of downloading a benign APK and processing it for dataset development in Figure 6.

Figure 6
Benign Downloader.



We use application Tags, Feature Tags, Library Tags, Meta Data Tags, Permission Tags, Provider Tags, Re-

ceiver Tags, and Service Tags. Feature extraction in the context of Android malware analysis involves extracting relevant information or attributes from the decompiled APK files that can be used for further analysis. We use APKtool to get the AndroidManifest.xml files from the APK files and decode them. APKtool is a useful open-source tool to reverse engineer Android apps and extract various files such as XML, Java, and resources. We use "elem.getAttribute" from the "xml" library available with Python. We then call "xmldoc.getElementsByTagName(' ')" with application Tags, Feature Tags, Library Tags, Meta Data Tags, Permission Tags, Provider Tags, Receiver Tags, and Service Tags. Feature extraction in the context of Android malware analysis involves extracting relevant information or attributes from the decompiled APK files that can be used for further analysis. We use APKtool to get the AndroidManifest.xml files from the APK files and decode them. APKtool is a useful open-source tool to reverse engineer Android apps and extract various files such as XML, Java, and resources. We use "elem.getAttribute" from the "xml" library available with Python. We then call "xmldoc.getElementsByTagName(' ')" with application Tags, Feature Tags, Library Tags, Meta Data Tags, Permission Tags, Provider Tags, Receiver Tags, and Service Tags as arguments and do feature extraction. We get "43,391" features to which we apply feature reduction and get "10523" features. We show the features we get from our script before pre-processing and after pre-processing in Table 5. Our dataset development process[3], our Python scripts (code)[4], and the resulting datasets are available for public use on Kaggle[5].

Algorithm 2: Deep Neural Network (DNN) Model

Input: Dataset-Training T, Dataset-Testing t, Epochs E, Batch-Size B, Weights W, DNN Layers DL

Normalize T (0 1)
Reshape T
Initialize DNN architecture:
Dense(unit = 2^n, activation=relu,
kernel_regularizer=l2(0.0001) + Dropout(rate = 0.2))
output layer = Dense(units = 12, activation = softmax)
Optimisation Settings:
optimizer = Adam(learning_rate=0.0001)
loss = categorical_crossentropy
metrics = accuracy
for epoch in range(E) **do**
  for i in range(0, len(T), B) **do**
    batch = $T[i : i + B]$
    Train w.r.t B from T
    Calculate loss for the B
    if Wrong Prediction **then**
      Update W
    end if
  end for
end for

Figure 7
Deep Neural Network Architecture.

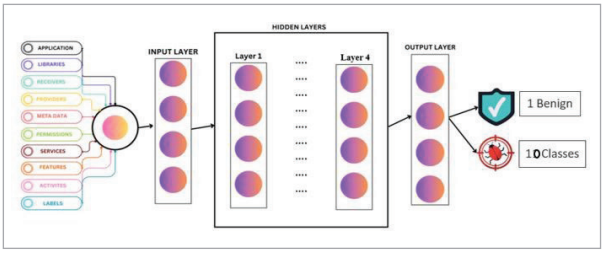


Table 5
Features Before Processing and After Processing.

Feature Set	# of Features before feature selection	# of Feature after feature selection
Application Tags	6	3
Feature Tags	89	65
Library Tags	21	9
Meta Data Tags	14051	3777
Permission Tags	4161	848
Provider Tags	5965	619
Receiver Tags	8584	2476
Activity Tags Count	1	1
Service Tags	10496	2725
Total	**43374**	**10523**

3.2. Data Pre-processing

We handle missing data, class imbalance, and dataset noise in this step. We removed any irrelevant or redundant data from the dataset. We removed the App Name column as it does not contain any useful information for our analysis. We also remove any missing or null values in the dataset and eliminate them as needed. We implement balancing techniques to ensure that each class has an equal number of representative samples for our analysis and model training even though we selected an equal number of APK files for each class in the dataset development phase. We use SMOTE to generate synthetic samples of the minority class. This ensures that the model is not biased towards the majority class. Feature selection is a method of selecting a group of important features to use in the model. In this phase, we counted the number of ones in each column to determine

[3] https://www.kaggle.com/code/hiraak27/script-for-samdroid-dataset/notebook
[4] https://www.kaggle.com/code/hiraak27/samdroid-features-analysis
[5] https://www.kaggle.com/datasets/hiraak27/samdroid-dataset

Table 6
DNN Architecture.

		HL1	HL2	HL3	HL4	Output Layer
0	Input	10523	512	256	128	64
1	Nodes	512	256	128	64	12
2	Weights (%)	96.88%	2.36%	0.59%	0.15%	0.01%
3	Biases (%)	52.67%	26.34%	13.17%	6.58%	1.23%
4	Variance	0.02%	0.26%	0.53%	1.03%	2.58%

the frequency of each feature in the collection. Then we set a threshold to eliminate features with fewer than a specified number of 1s. For instance, we set a threshold of 2, meaning that any feature with fewer than two 1's was removed from the set. Our number of features reduced from a total of 43,377 features to 10,523 during the preprocessing stage.

3.3. DNN Model Development Details

We make use of a "feedforward" DNN and we implement the "Transfer learning" (TL) approach. We provide a smaller dataset for the second DNN model to fine-tune its parameters further. This transfer of knowledge enabled us to capitalize on the pre-learned features and representations from the first model, thus enhancing the performance and efficiency of the second model while working with a limited amount of data.

Our DNN consists of an input layer, three hidden layers with ReLU activation, and an output layer with a softmax activation as presented in Table 6, where we show its input size, number of nodes, and the distribution of weights, bias, and variance. Beginning with an input size of 10,523, the first hidden layer (HL1) encompasses 512 nodes, which dominate the model by holding 96.88% of the total weights and 52.67% of the biases. As we progress deeper into the network, subsequent hidden layers sequentially decrease in node count, with each layer holding 256, 128, and 64 nodes, respectively. The final output layer is streamlined to 12 nodes. The variance values across the layers, as presented, offer a glimpse into the weight distribution within each layer, suggesting the potential intricacy of patterns that each layer might discern. This tabulated representation furnishes a snapshot of the DNN's architecture, highlighting the layered composition and the relative significance of each component, as shown in Algorithm 2.

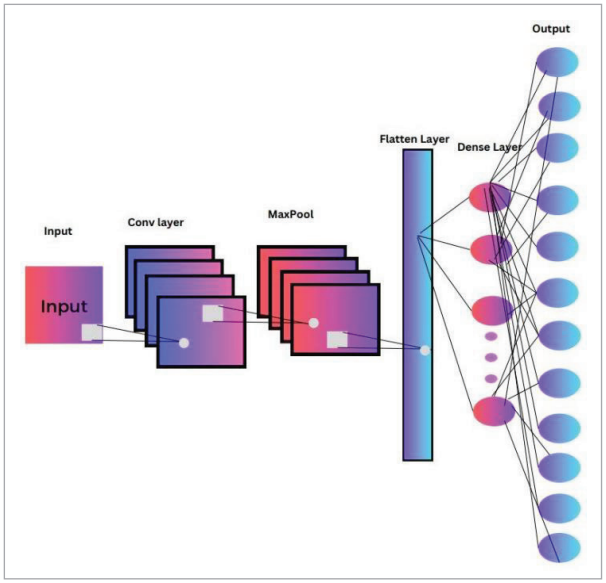
We use the categorical cross-entropy loss function and the Adam optimizer to train the DNN. The input layer receives "10523" features, which are extracted from the AndroidManifest.xml files of the APK samples. We use ReLU activation function as "f(x) = max(0, x)" which means that it outputs the maximum between 0 and the input value. We use The ReLU activation in three hidden layers to introduce non-linearity to the model and help it learn complex patterns. in the data. The output layer has 11 nodes, where 10 nodes correspond to the 10 malware sub-classes, and the eleventh node corresponds to the benign class. The softmax activation function is applied to the output layer, which outputs a probability distribution over the 11 classes. The softmax function ensures that the sum of the outputs of all nodes is equal to 1, which gives us a probability distribution. The categorical cross entropy measures the difference between the predicted probability distribution and the true probability distribution. We used the Adam optimizer with a learning rate of 0.0001 to get the settings of the model to work best. The Adam optimizer is an adaptive learning rate optimization algorithm that is efficient and robust to noisy gradient information. We also applied "L2 regularization" to the three hidden layers to prevent overfitting. L2 regularization adds a penalty term to the loss function that encourages the model weights to be small. We use the early stopping functionality to stop the training process when the validation loss stops improving. We trained the model for 200 epochs, and the early stopping functionality helped us to prevent overfitting and achieve better generalization performance. The graphical presentation of the neural network architecture is shown in

Figure 7 and the proposed algorithm is presented in Algorithm 2.

3.3.1. Feedforward Deep Neural Network (DNN)

In this section, we explain the architecture training methodology of the feedforward Deep Neural Network (DNN) employed in our approach. The choice of a feedforward DNN ability to balance simplicity with powerful feature learning capabilities contributes to its effectiveness in analyzing the AndroidManifest.xml files.

Figure 8
CNN Architecture.



3.3.2. Architecture Overview

The feedforward DNN architecture is characterized by its sequential arrangement of layers, where data flows linearly from the input layer through multiple hidden layers to the output layer. Each layer comprises interconnected neurons, with connections directed forward, hence the term "feed-forward". We include an input layer, three hidden layers utilizing Rectified Linear Unit (ReLU) activation functions, and an output layer with "softmax" activation function to generate a probability distribution across output classes, thereby streamlining multi-class classification tasks. We also use ReLU activation function which is used to introduce non-linearity into the model which helps in improving the capacity to capture complex data patterns effectively.

3.3.3. Training Methodology

We employ L2 regularization in the hidden layers to mitigate overfitting by penalizing large weights with dropout regularization to enhance generalization by introducing redundancy and reducing reliance on specific neurons. We benefit from the hierarchical structure of the DNN that enables automatic feature learning, wherein raw input features are progressively transformed into higher-level representations. This hierarchical feature learning capability is crucial for analyzing complex patterns within the AndroidManifest.xml files, contributing to the model's high accuracy.

3.3.4. Comparison with Other Neural Networks

Compared to other neural network architectures such as r convolutional neural networks (CNNs)or recurrent neural networks (RNNs), the feedforward DNN provides a favorable balance of simplicity and performance. While RNNs and CNNs excel in tasks involving sequential or spatial data, the feedforward DNN's sequential nature and hierarchical feature learning make it particularly well-suited for the analysis of structured data like AndroidManifest.xml files.

3.4. Architecture of CNN

The graphical presentation of the neural network architecture is shown in Figure 8. We present the architecture of our proposed CNN-based model in Algorithm 3 and transfer learning-based model architecture using Algorithm 4 also shows the graphical presentation of the transfer learning is shown in Figure 9.

Our CNN model consists of four Conv1D layers, each with a different filter size determined by 2n, followed by a ReLU activation function. After each convolutional layer, there is a MaxPooling1D operation to down-sample the data, and a Flatten layer to convert it into a suitable format for subsequent processing. BatchNormalization is applied to improve training stability, and Dropout is incorporated to mitigate overfitting. It is presented in Algorithm 3.

In the Convolutional Neural Network (CNN) architecture detailed in the Table 7, the model is structured with a series of convolutional layers followed by dense layers. Starting with an expansive input size of 10,523, the first convolutional layer (Conv1D-1) introduces 8 filters. Thus, it retains only 2.06% of the biases, it holds a relatively small portion of the total

Table 7
CNN Architecture.

	Conv1D-1	Conv1D-2	Conv1D-3	Conv1D-4	Dense-1	Output Layer
Input	10523	8	16	32	*	128
Filters/Nodes	8	16	32	64	128	12
Weights (%)	0.00%	0.01%	0.03%	0.12%	99.81%	0.03%
Biases (%)	2.06%	4.12%	8.25%	16.49%	32.99%	3.09%
Variance	5.83%	2.73%	1.40%	0.70%	0.00%	0.00%

weights. The number of filters increases gradually to 16, 32, and 64 as the data moves through the network's successive convolutional layers (Conv1D-2, Conv1D-3, and Conv1D-4).

The variance values across these layers decrease, suggesting a stabilization in the weight distribution as the data is processed. Transitioning from convolutional to dense layers, the Dense-1 layer stands out prominently, holding a staggering 99.81% of the total weights and 32.99% of the biases, with 128 nodes. The final output layer is streamlined to 12 nodes. This tabulated representation provides a comprehensive overview of CNN's architecture, emphasizing the layered hierarchy and the relative significance of each segment. The analysis of the CNN architecture in this section comprehensive understanding of its design and functionality.

3.4.1. Input Layer

CNN receives input data comprising a sequence of 10,523 features. This initial layer serves as the entry point for data into the neural network.

3.4.2. Convolutional Layers (Conv1D)

The layers utilize convolutional filters to extract relevant features from the input data, capturing increasingly complex patterns. The CNN architecture has four layers include Conv1D-1, Conv1D-2, Conv1D-3, Conv1D-4. To enhance the model's ability to discern intricate relationships within the data by employing the ReLU activation function, non- linearity is introduced

3.4.3. MaxPooling1D Layer

Following each convolutional layer, a MaxPooling1D operation is applied to down-sample the data. This operation AIM reduces the dimensionality of feature maps while pre-serving critical information, thereby expediting computation and mitigating the risk of overfitting by promoting spatial in-variance.

3.4.4. Flatten Layer

The flatten layer transforms the output from the convolutional and pooling layers into a one-dimensional array, preparing it for input into subsequent dense layers.

3.4.5. Batch Normalization and Dropout

Batch normalization enhances speed by normalizing the activations of each layer and training stability. Dropout is employed to counter overfitting by randomly dropping a fraction of connections between neurons during training.

3.4.6. Dense Layer (Dense-1)

Featuring 128 nodes, the Dense-1 layer plays a pivotal role in learning complex patterns within the data. Applying a softmax activation function, it generates probability distributions over the output classes, The final output layer encompasses 12 nodes corresponding to the classification task's classes. Leveraging the capturing high-level features acquired from lower layers and facilitating final classification.

3.4.7. Output Layer

softmax activation function, it yields probabilities for each class, indicating the input's likelihood belonging to each class.

3.5. Architecture of Transfer Learning

We construct a new deep neural network model by lever- aging the architecture of a pertained model. This pertained model serves as a starting point for our work. To adapt this model for the specific task of ze-

ro-day attack detection, we make a crucial modification to the output layer. It is pertinent to note that we have 10 malware classes and one benign class that we considered earlier and the twelfth one is for zero-day attacks raising the total number of classes from a total of 11 to 12 classes. For training, we configure the model by specifying the optimization method (Adam with a learning rate of 0.0001) and the loss function (categorical cross-entropy). The training process involves iterating over the data for a defined number of epochs, and we process the data in batches, enhancing efficiency. During training, we monitor the loss, and if it surpasses a predefined threshold (typically set at 0.1 in this case), it signals a potential zero-day attack.

In response to this detection, we can take appropriate actions, such as updating the model's weights or implementing further security measures. After training, we save the trained model for future use. To assess its performance, we evaluate it on a separate set of testing data. We primarily measure the accuracy as an indicator of how well the model can classify data into its respective classes in the evaluation phase, which includes identifying zero-day attacks when they occur. Transfer learning involves the knowledge of how the algorithm's architecture interacts with the pre-existing model to handle zero-day attacks. We present the architecture of our transfer learning-based model in Algorithm 4.

Finally, the output layer is a Dense layer with 12 units and a softmax activation function, which is suitable for tasks involving multi-class classification. For optimization, the model uses the Adam optimizer with a learning rate of 0.0001, employs categorical cross-entropy as the loss function, and tracks accuracy as a metric for model performance. The training process unfolds within a loop that runs for the specified number of training epochs. Inside each epoch, the training data is divided into batches of a specified size, and the model is trained on each batch while computing the loss. If the model makes incorrect predictions on a batch, it updates its weights to enhance its performance. After completing the training, the function saves the trained model to a file for later use. We use the following evaluation metrics to assess the functioning of the deep learning models:

Accuracy which refers to the proportion of the total number of instances in the data set with the number of correctly classified instances (both true positives and true negatives), measures how often the model correctly predicts the labels.

Precision: Precision refers to a measure of the accuracy of positive predictions and the percentage of true positive instances (among those predicted as positive).

Algorithm 3: Convolutional Neural Network (CNN) Model

Input: Dataset-Training X_{train}, Dataset-Training Labels y_{train}, Epochs E, Batch-Size B

Normalize X_{train} (0 1)
Reshape X_{train}
Initialize CNN model:
    Conv1D(2^n, 3, activation='relu', input_shape=(size, 1)) + MaxPooling1D(2) + Flatten()
    BatchNormalization() + Dropout(0.3)
    Output Layer = Dense(12, activation='softmax')
Optimisation Settings:
    optimizer = Adam(learning_rate=0.0001)
    loss = categorical_crossentropy
    metrics = accuracy
for epoch in range(E) **do**
    for i in range(0, len(X_{train}), B) **do**
        batch = $X_{train}[i : i + B]$
        Train the model on the batch
        Calculate loss for the batch
        if Wrong Prediction **then**
            Update the model weights
        end if
    end for
end for

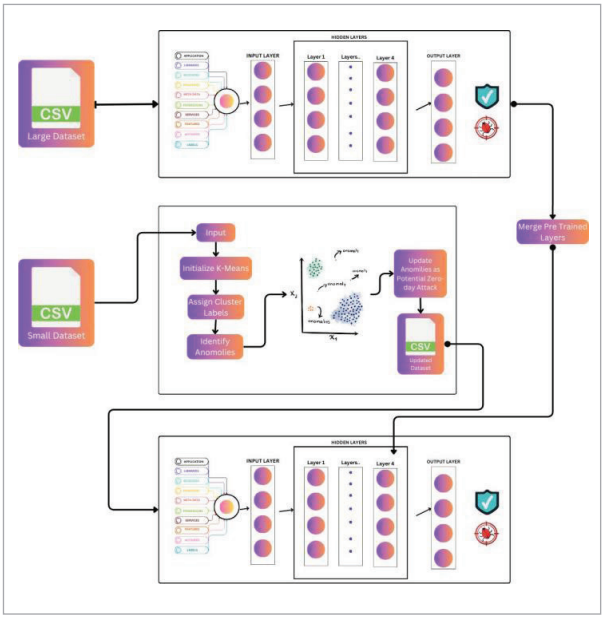
F1-Score The F1-score is the harmonic average of the precision and recall, achieving the balance. These scenarios help us find whether the model is performing accurately or not and it is generally preferable to use F1-score when we have imbalanced datasets since it takes both precision and recall into account.

Confusion Matrix Performance metrics of a classification algorithm through which counts of False Positives and False Negatives True Positives, True Negatives and helps us to get a better understanding of how well the model is doing, especially when it comes to distinguishing between different classes.

Recall (Sensitivity or True Positive Rate): The proportion of true positive examples, which has been correctly identified by the model. This gives a good idea regarding how well the model tries to capture all the instances of a class. In other words, how well it tries to capture whatever is useful to us.

ROC-AUC Receiver Operating Characteristic – Area Under the Curve. It is a measure of how well a model separates complex classes. The ROC curve plots the class-specific True Positive rate against the FP rate at different thresholds, the AUC is a mathematical measure of a curve of how much the model is capable of distinguishing between the classes.

Figure 9
Transfer Learning Architecture



The accuracy rates of 97% for DNN model and 96% for the CNN model were also determined by calculating the proportion of instances that have been correctly classified (both true positives and true negatives) out of the total instances in the test set. DNN model shows better results than the CNN model by having a slightly higher accuracy than the CNN model, which means DNN proves to be more effective at successfully classifying the malicious and benign samples of the dataset.

3.6. Statistical Analysis

We perform statistical analysis using p-value to assess the significance of observed differences between models. This analysis helped us understand whether the observed variations in model performance were statistically significant. We conducted the binomial test and chi-square test using a significance level (alpha) of 0.05. The results yielded significant evidence to reject the null hypothesis based on both tests. This indicates that the DNN model exhibits a statistically significant ability to detect malware. Furthermore, the validation accuracy of 83.56%

Algorithm 4: Zero-Day Detection with Transfer Learning

Input: Dataset-Training X_{train}, Dataset-Training Labels y_{train}, Epochs E, Batch-Size B, *pretrained_model*

Normalize T (0 1) and reshape
$T = T/\text{np.max}(T)$
$T = T.reshape(T.shape[0], -1)$
Initialize DNN architecture
$model$ = Sequential()
for *layer* in *pretrained_model.layers* **do**
  $model.add(layer)$
end for
Modify the output layer for zero-day detection
$model.layers[-1].output\_dim$ = 12 (12 classes - 10 malware, one benign, and 1 zero-day)
$model.layers[-1].activation = 'softmax'$
Optimization settings
optimizer = Adam(learning_rate=0.0001)
model.compile(loss = categorical_crossentropy
optimizer = optimizer, metrics=['accuracy'])
for *epoch* in range(E) **do**
  for i in range(0, len(T), B) **do**
    $batch = T[i : i + B]$
    Training w.r.t B from T
    Calculate loss for the batch
    $loss = model.train\_on\_batch(batch, batch)$
    If "loss" is above a threshold, update weights
    $threshold = 0.1$
    if $loss > threshold$ **then**
      $model.layers[-1].set\_weights(new\_weights)$
    end if
  end for
end for

We split our test such that we test Top 50 Correlated Columns exhibiting the highest correlation with the malware label column shown in Table 8. These columns were chosen based on their potential significance in understanding the presence of malware in the dataset. Following this selection, we subjected these columns to the Chi-Square (χ^2) test, a statistical method known for its ability to assess independence between categorical variables. Surprisingly, the results of this first test revealed zero instances of test failures, signifying the robustness of the correlations identified.

Table 8
P-Value Analysis of Top 50 Columns.

Column	P-Value	Test Passed	
0	Malware Class	0.000000e+00	True
1	PERMISSION:android.permission.READ_PHONE_STATE	0.000000e+00	True
2	PERMISSION:android.permission.GET_TASKS	0.000000e+00	True
3	meta-data_value:@integer/google_play_services_...	0.000000e+00	True
4	meta-data_value:none	0.000000e+00	True
5	provider:com.android.vending.expansion.zipfile...	0.000000e+00	True
6	meta-data_value:@string/google_maps_v2_api_key	0.000000e+00	True
7	provider:com.qbiki.util.InternalFileContentPro...	0.000000e+00	True
8	receiver:com.qbiki.seattleclouds.ExpansionFile...	0.000000e+00	True
9	SERVICE:com.qbiki.modules.videolist.DownloadSe...	0.000000e+00	True
10	SERVICE:com.qbiki.seattleclouds.ExpansionFiles...	0.000000e+00	True
11	receiver:com.qbiki.gcm.GCMBroadcastReceiver	0.000000e+00	True
12	SERVICE:com.qbiki.geofencing.ReceiveTransition...	0.000000e+00	True
13	meta-data:com.google.android.maps.v2.API_KEY	0.000000e+00	True
14	provider_readPermission:	0.000000e+00	True
15	PERMISSION:android.permission.WRITE_EXTERNAL_S...	0.000000e+00	True
16	provider_writePermission:	0.000000e+00	True
17	PERMISSION:android.permission.SYSTEM_ALERT_WINDOW	0.000000e+00	True
18	provider_grantUriPermissions:	0.000000e+00	True
19	allowBackup_false	9.768805e-242	True
20	meta-data:com.google.android.gms.version	0.000000e+00	True
21	provider:com.google.firebase.provider.Firebase...	0.000000e+00	True
22	FEATURE:android.hardware.screen.landscape	0.000000e+00	True
23	SERVICE:com.wyh.framework.KenelService	0.000000e+00	True
24	receiver:com.wyh.framework.MonitorReceiver	0.000000e+00	True
25	receiver:com.google.firebase.iid.FirebaseInsta...	0.000000e+00	True
26	PERMISSION:com.google.android.c2dm.permission....	0.000000e+00	True
27	meta-data:com.android.vending.derived.apk.id	0.000000e+00	True
28	meta-data:android.app.default_searchable	0.000000e+00	True
29	meta-data_value:gfan	0.000000e+00	True
30	FEATURE:android.hardware.screen.portrait	0.000000e+00	True
31	meta-data:android.app.searchable	4.760756e-279	True
32	receiver:com.qbiki.modules.goaltracker.GoalTra...	0.000000e+00	True
33	PERMISSION:android.permission.ACCESS_WIFI_STATE	0.000000e+00	True

Column	P-Value	Test Passed	
34	meta-data:UMENG_CHANNEL	0.000000e+00	True
35	PERMISSION:com.android.launcher.permission.INS...	0.000000e+00	True
36	SERVICE:com.google.firebase.messaging.Firebase...	0.000000e+00	True
37	meta-data_value:com.google.firebase.components...	0.000000e+00	True
38	SERVICE:com.google.firebase.components.Compone...	0.000000e+00	True
39	meta-data:UMENG_APPKEY	0.000000e+00	True
40	SERVICE:com.google.android.gms.measurement.App...	0.000000e+00	True
41	PERMISSION:android.permission.MOUNT_UNMOUNT_FI...	0.000000e+00	True
42	allowBackup_	0.000000e+00	True
43	PERMISSION:android.permission.VIBRATE	0.000000e+00	True
44	PERMISSION:com.google.android.finsky.permissio...	0.000000e+00	True
45	PERMISSION:android.permission.SEND_SMS	0.000000e+00	True
46	SERVICE:com.google.android.gms.measurement.App...	0.000000e+00	True
47	receiver:com.google.android.gms.measurement.Ap...	0.000000e+00	True
48	provider_grantUriPermissions:true	0.000000e+00	True
49	meta-data:android.support.FILE_PROVIDER_PATHS	0.000000e+00	True

Figure 10
P-Value Analysis.

Out[15]:

	Column	P-Value	Test Passed
0	FEATURE:	0.000000	True
3	FEATURE:android.hardware.LOCATION	0.000525	True
4	FEATURE:android.hardware.audio.low_latency	0.000000	True
5	FEATURE:android.hardware.audio.pro	0.000525	True
6	FEATURE:android.hardware.autofocus	0.000002	True
...	...	...	...
10518	receiver_enabled:true	0.000000	True
10519	usesCleartextTraffic_	0.000000	True
10520	usesCleartextTraffic_false	0.000000	True
10521	usesCleartextTraffic_true	0.000000	True
10522	Activity Count	0.000000	True

8819 rows × 3 columns

Figure 11
Failed P-Value Analysis.

Out[16]:

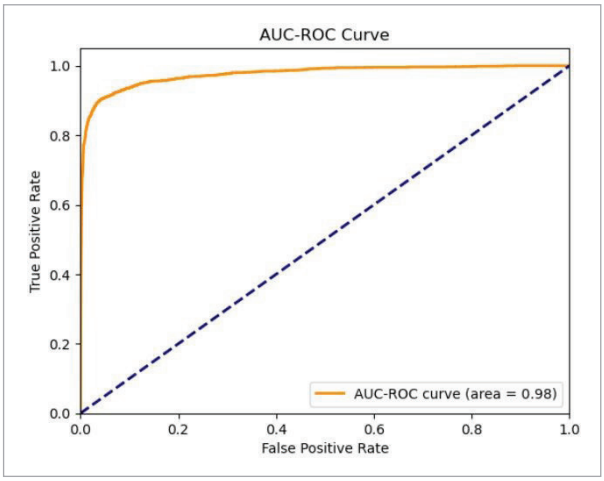
	Column	P-Value	Test Passed
1	FEATURE: android.permission.ACCESS_WIFI_STATE	0.530262	False
2	FEATURE:android.hardware.Camera	0.530262	False
32	FEATURE:android.hardware.sensor.barometer	0.621719	False
35	FEATURE:android.hardware.sensor.light	0.621719	False
36	FEATURE:android.hardware.sensor.proximity	0.621719	False
...	...	...	...
10508	receiver:xcxin.filexpert.receiver.ApkReceiver	0.530262	False
10509	receiver:xcxin.filexpert.receiver.MediaReceiver	0.530262	False
10510	receiver:xcxin.filexpert.wxapi.AppRegister	0.530262	False
10515	receiver_enabled:@bool/hasKitKat	0.530262	False
10516	receiver_enabled:@bool/preKitKat	0.530262	False

1704 rows × 3 columns

In the second phase, we extend the analysis to complete 10,523 columns within the dataset, individually paired with the malware label column. This comprehensive approach involved the execution of a total of 10,523 Chi-Square tests, each assessing the independence between a single column and the malware label column. Impressively, out of these tests, 8819 successfully passed as shown in Figure 10, revealing significant relationships between these columns and the presence of malware. However, the remaining 1,704 tests resulted in failures which is shown in Figure 11, highlighting the complexity and diversity of factors present in the dataset. This meticulous and thorough statistical examination provides valuable insights into the dataset's composition, offering a deeper understanding of the correlations and potential indicators associated with the presence of malware. The utilization of P-value analysis in

this research underscores its efficacy in identifying significant relationships and patterns, serving as a valuable resource for further investigation into malware detection and mitigation strategies.

Figure 12
DNN AUC-ROC Curve.



4. Results and Discussion

We evaluate and compare all three models by using the evaluation metrics which included accuracy, confusion matrix, F1 score, recall, and ROC-AUC. We present the results of ROC-AUC in Figure 13, 12, and 14. We perform experiments using the Keras python library with Python version 3.11.3, by using Scikitlearn, Numpy, Tensorflow, and Pandas libraries to achieve the desired results. The experiments are done on a single PC server with a Core-i7 Processor Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz, 2.71 GHz, and a GPU NVIDIA Quadro M2000M. The operating system used for the experiments is Windows 10-64 Bit, with 16 GB-3600 MHz DDR4 RAM.

We calculate Accuracy, Recall, Precision, ROC, AUC, and F1-Score. The Confusion Matrix is used to present the actual values of True Negative (TN), True Positive (TP), False Positive (FP), and False Negative (FN). When dealing with the classes, the Confusion Matrix without normalization accurately represents the results for each predicted label. In the case of balanced datasets, the normalized Confusion Matrix displays the results as a percentage, allowing for a comprehensive assessment of each class. By experiments, we get these results which are mention above.

We present the results of our experiments comparing the performance of the Deep Neural Network DNN) approach with the alternative Convolutional Neural Network (CNN) and Transfer Learning (TL) method for Android malware detection which is shown in Table 9.

Table 9
DNN VS CNN VS Transfer Learning (TL).

S#	Item	DNN	CNN	TL
1	Training time (approx)	53.33 Minutes	173 Minutes	05 Minutes
2	Epochs	194	200	50
3	Accuracy	97.62%	96.44%	94.45
4	Loss	0.1424	0.1278	0.3550
5	COnfusion Matrix	Figure 15	Figure 16	Figure 17
6	AUC-ROC	0.98 Fig12	0.97 Fig13	1.0 Fig14
7	Recall	0.84	0.82	0.96
8	F1-Score	0.84	0.82	0.97
9	Precision	0.84	0.82	0.98

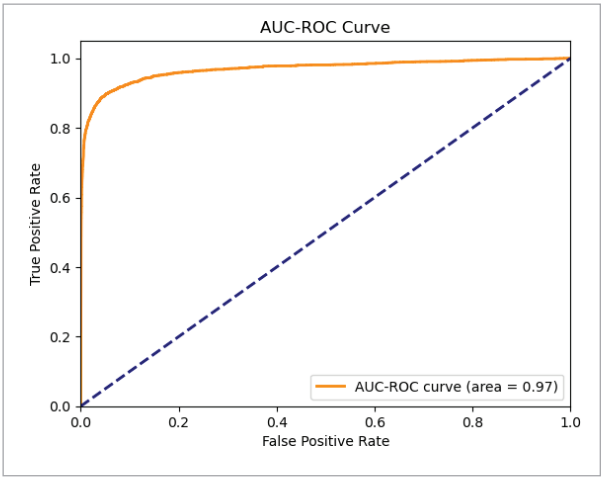
1. **Training Time:** The DNN model required approximately 53.33 minutes to complete the training process, while the CNN model took significantly longer, with a training time of approximately 2 hours and 53 minutes.
2. **Epoch:** The DNN model was trained for a total of 194 epochs, whereas CNN model was trained for 200 epochs.
3. **Accuracy:** The DNN model's accuracy was 97.62%, which surpassed the CNN model's accuracy of 96.44%.
4. **Loss:** The DNN model had a loss value of 0.1424, whereas the CNN model had a loss value of 0.1278.
5. **Confusion Matrix:** The confusion matrices for CNN and DNN models are depicted in Figures 15-16, respectively.
6. **AUC-ROC:** The DNN model exhibited an AUC-ROC score of 0.98, indicating excellent discriminative power. The CNN model also performed well, achieving an AUC-ROC score of 0.97.

7 **Recall:** The DNN model had a recall rate of 0.84, while the CNN model had a slightly lower recall rate of 0.82.

8 **F1 Score:** The DNN and CNN models obtained F1 scores of 0.84 and 0.82, respectively, indicating their usefulness in balancing accuracy and recall.

9 **Precision:** For both models, the precision values were 0.84 for the DNN model and 0.82 for the CNN model, indicating their ability to correctly classify threatening and non-malicious applications for Android.

Figure 13
CNN AUC-ROC Curve.

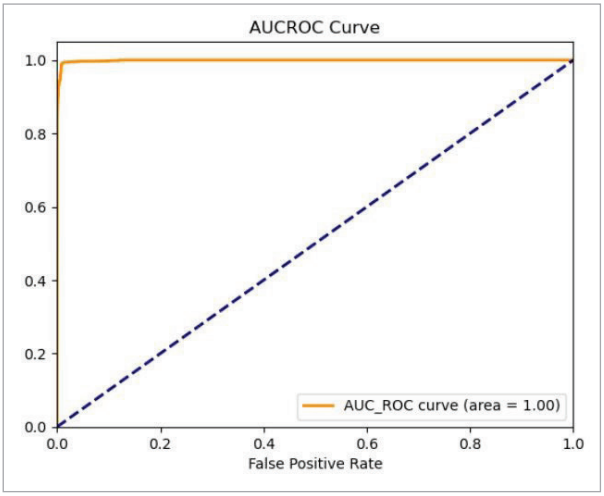


4.1. Selection of Hyper-Parameters

We employ various hyper parameters to enhance the performance of our deep learning algorithm, as highlighted in the study [24]. Our approach involves the following components:

1 **Epochs:** Epochs denote the training iterations per- formed on our deep neural networks. Initially, we configured the model to undergo 1000 training iterations. Subsequently, we implemented an early stopping technique, which continually assessed the model's performance by monitoring the training loss. If, for five consecutive epochs, the loss failed to exhibit any further reduction, we made the informed decision to conclude the training process at that specific epoch. This choice was grounded in the understanding that an increase in loss indicates that the model has likely reached its optimal training state. Further iterations would carry the potential risk of overfitting. Employing this approach, our DNN Model successfully completed its training after the 200th epoch in our experiments.

Figure 14
Transfer Learning AUC-ROC Curve.



2 **Batch-size:** The batch size denotes the number of data samples processed together during each training iteration. In our series of experiments, we varied batch sizes, ranging from 8 to 128, and found that a batch size of 64 emerged as the most optimal choice for both our DNN, CNN, and TL-DNN (transfer Learning with DNN). Importantly, the preference for a batch size of 64 aligns with our dataset. In this context, larger batch sizes expedite training while mitigating the risk of slower convergence, thereby facilitating the accomplishment of desirable model performance levels.

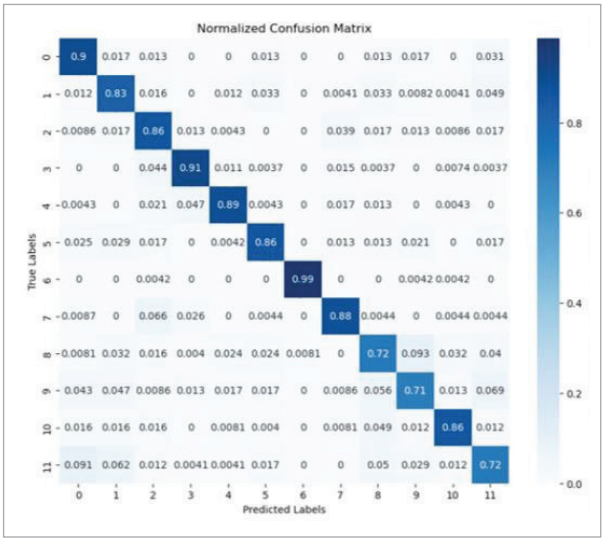
3 **Dropout** is a regularization technique where for each layer, a fraction of neurons (represented as a percentage) is randomly deactivated during training to prevent overfitting. We implemented a 20% dropout rate for each layer in the DNN and a 30% dropout rate in the CNN. These dropout rates emerged as the most suitable choices through systematic experimentation, where we tested a range of values from no dropout to 90%.

4 **Optimizer function:** We use Adam optimizer with a minimum learning rate of 0.001 to Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs) and pre-trained DNN used in trans-

fer learning ensures stable and efficient fine-tuning for new tasks.

5 **Activation Function:** In our neural network, we chose to employ the Rectified Linear Unit (ReLU) activation function for all hidden layers due to its computational efficiency and its ability to mitigate the vanishing gradient problem, thus allowing the model to effectively capture complicated patterns and advance the training process. We use "softmax" activation function for the output layer. This selection was made to transform raw scores into a probability distribution, which is particularly beneficial for tasks involving multi- class classification. It enables us to obtain interpretable and probabilistic class predictions.

Figure 15
Confusion Matrix DNN.

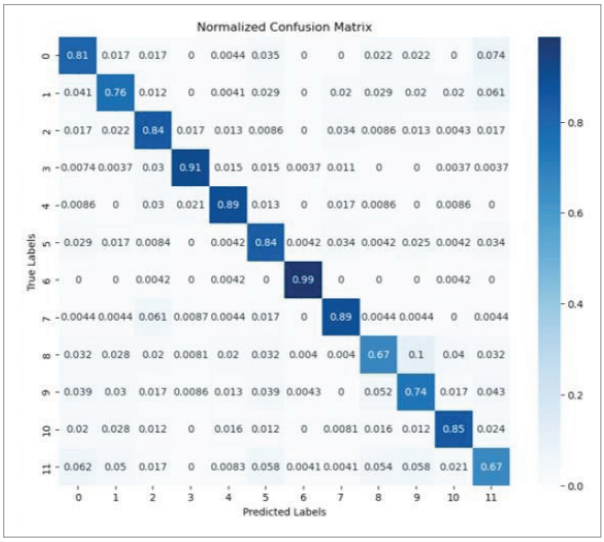


Overall, the DNN-based model outperformed the CNN- based model in key areas, including training time, accuracy, and several assessment criteria. These results suggest that the DNN model holds promise as an effective solution for Android malware detection. Both models' confusion metrics are shown in Figures 15-17.

4.2. Robustness and Sensitivity Analysis

We evaluate the robustness of our Android malware detection model by sensitivity analysis. We obtain significant insights about our model's robustness and adaptability by applying it to diverse situations and analyzing its performance under different settings. We explore the impact of changes in training data size and class distribution, shedding light on the model's stability and effectiveness across different settings. Through these analyses, we gain a deeper under- standing of the model's capabilities and uncover important considerations for its deployment in real-world scenarios. We conduct a sensitivity analysis by changing the dataset from balanced to unbalanced. This analysis aimed to assess the performance of our DNN model under varying class distribution scenarios. The results obtained from this sensitivity analysis are shown in Figure 18.

Figure 16
Confusion Matrix CNN.



In addition, we also conducted a sensitivity analysis using holdout method. The analysis focuses on key metrics such as epoch, training time, loss, accuracy, precision, recall, ROC and AUC, F1 score, and confusion matrix. In Table 10, The sensitivity analysis of the DNN model clearly demonstrates its resilience and efficacy in malware detection, even with a reduced training data size. Despite the challenges posed by a smaller dataset, our model exhibits consistently high accuracy, precision, recall, and F1 score, showcasing its capability to effectively identify and classify Android malware instances.

This illustrates the robustness and strength of our approach, highlighting its potential for real-world applications.

Figure 17
Confusion Matrix Transfer Learning.

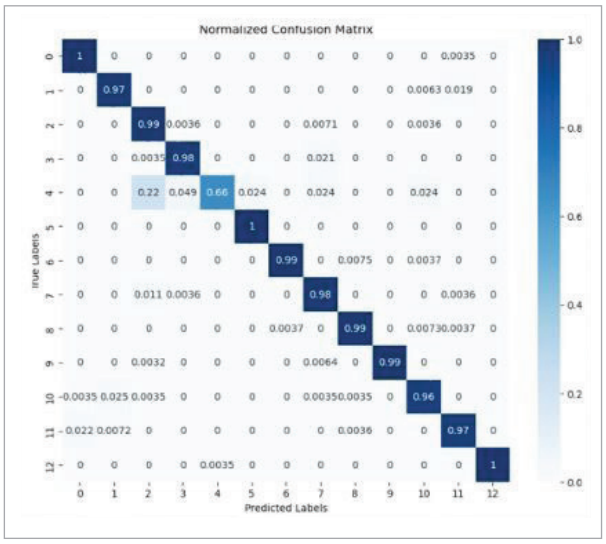


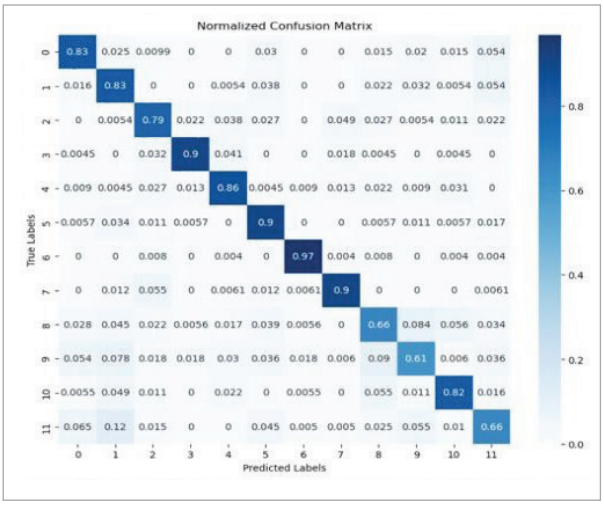
Table 10
Sensitivity Analysis by Performance Metrics.

Epoch	200
Training Time	40 Minutes
Loss	0.1376
Accuracy	97.76%
Precision	0.81
Recall	0.81
ROC and AUC	0.97
F1 Score	0.81

We report that the training time of the DNN model outperformed the CNN model indicating that the DNN model offers a more time-efficient solution for Android malware detection. Moreover, the DNN model achieved higher accuracy, with a rate of 97.62%, surpassing the CNN model's accuracy of 96.44%. The AUC-ROC scores for both models are quite high, with the DNN model achieving a score of 0.98 and the CNN model achieving 0.97, as shown in Figures 12-14.

This demonstrates the models' ability to effectively differentiate between positive and negative instances, further validating their effectiveness in detecting Android malware. Our findings highlight the effectiveness of utilizing deep neural networks in the context of Android malware detection using static feature analysis. The sensitivity analysis further substantiated the robustness of our DNN model. Even with reduced training data, the model maintained high accuracy, precision, recall, and F1 score.

Figure 18
Sensitivity Analysis by unbalancing the dataset.



5. Conclusion

We proposed a deep learning-based approach for addressing the security concerns associated with Android-based applications, specifically focusing on malware detection. Our model exhibits the capability to process a substantial 10,524 features as input data, a considerable leap beyond the limitations of prior DNN models, which typically accommodated a maximum of 350 features. We evaluate the performance of our proposed mechanism using our proposed datasets that contain more features and more classes and benchmark deep-learning algorithms. We do statistical evaluation using P-value analysis and we employ various performance metrics such as Recall, F1 score, Confusion Matrix, Accuracy, AUC (Area under the Curve), ROC (Receiver Operating Characteristic to assess the effectiveness of our multi-threat malware detection techniques. Our experimental results demonstrate that our approach achieves high detection accuracy with DNN while maintaining efficient processing times. The deep neural networks (DNNs)

train model gives an accuracy of 97.62%, we also apply Convolutional neural networks (CNNs) yields an accuracy rate of 96.44%, underlining the robustness of our approach in tackling complex malware threats. Even in the case of Transfer Learning, where the accuracy reaches 94.45% with the addition of zero-day attacks alongside the other 10 malware families and 1 benign class are considered. This highlights the proficiency of our system in addressing the complex challenges posed by multi-threat malware and detecting zero-day attacks in Android environments. Our proposal is useful, particularly, in handling zero- day attacks, and considers a dataset with a larger number of features. However, the number of examples and malware families considered in our proposal is limited. This can be handled by running our publicly contributed scripts for a longer time and downloading more APK files. We consider this as future work for us.

We plan to consider feature extraction from dynamic analysis and develop a hybrid approach for Android malware analysis as future work. We also plan to contribute a dataset covering more malware families and more examples.

Funding

No funding was received for this work.

Conflict of interest

The authors declare that there is no conflict of interest.

Data Availability

All datasets used in this paper are publicly available.

Author Contributions

Both authors have contributed equally to the preparation of this manuscript.

References

1. Almarshad, F.A., Zakariah, M., Gashgari, G., Aldakheel, E., Alzahrani, A. Detection of Android Malware Using Machine Learning and Siamese Shot Learning Technique for Security. IEEE Access, 2023, 1-1. doi:10.1109/ACCESS.2023.3331739. https://doi.org/10.1109/ACCESS.2023.3331739
2. Arslan, R. S., Doğru, I. A., Barişçi, N. Permission-Based Malware Detection System for Android Using Machine Learning Techniques. International Journal of Software Engineering and Knowledge Engineering, 2019, 29, 43-61. doi:10.1142/S0218194019500037. https://doi.org/10.1142/S0218194019500037
3. Awan, M.J., Masood, O.A., Mohammed, M.A., Yasin, A., Zain, A. M., Damaševičius, R., Abdulkareem, K. H. Image-Based Malware Classification Using VGG19 Network and Spatial Convolutional Attention. Electronics, 2021, 10. doi:10.3390/electronics10192444. https://doi.org/10.3390/electronics10192444
4. Bayazit, E. C., Sahingoz, O. K., Dogan, B. A Deep Learning-Based Android Malware Detection System with Static Analysis. Proceedings of the 2022 4th International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA), 2022. doi:10.1109/HORA55278.2022.9800057. https://doi.org/10.1109/HORA55278.2022.9800057
5. Bibi, I., Akhunzada, A., Malik, J., Ahmed, G., Raza, M. An Effective Android Ransomware Detection Through Multi-Factor Feature Filtration and Recurrent Neural Network. Proceedings of the 2019 UK/China Emerging Technologies Conference (UCET), 2019, pp. 1-4. doi:10.1109/UCET.2019.8881884. https://doi.org/10.1109/UCET.2019.8881884
6. Dawar, A. Enhancing Wireless Security and Privacy: A 2-Way Identity Authentication Method for 5G Networks. International Journal of Mathematics, Statistics, and Computer Science, 2024, 2, 183-198. doi:10.59543/ijmscs.v2i.9073. https://doi.org/10.59543/ijmscs.v2i.9073
7. Djenna, A., Bouridane, A., Rubab, S., Marou, I. M. Artificial Intelligence-Based Malware Detection, Analysis, and Mitigation. Symmetry, 2023, 15, 677. doi:10.3390/sym15030677. https://doi.org/10.3390/sym15030677
8. Farhat, H., Rammouz, V. Malware Classification Using Transfer Learning. arXiv Preprint, 2021, arXiv:2107.13743.
9. Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M., Rajarajan, M. Android Security: A Survey of Issues, Malware Penetration, and Defenses. IEEE Communications Surveys and Tutorials, 2015, 17, 998-1022. doi:10.1109/comst.2014.2386139. https://doi.org/10.1109/COMST.2014.2386139

10. Fereidooni, H., Conti, M., Yao, D., Sperduti, A. ANASTASIA: Android Malware Detection Using Static Analysis of Applications. Proceedings of the 2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS), 2016, 1-5. doi:10.1109/NTMS.2016.7792435. https://doi.org/10.1109/NTMS.2016.7792435

11. Haq, I. U., Khan, T.A., Akhunzada, A. A Dynamic Robust DL-Based Model for Android Malware Detection. IEEE Access, 2021, 9, 74510-74521. doi:10.1109/ACCESS.2021.3079370. https://doi.org/10.1109/ACCESS.2021.3079370

12. Huang, Y., Li, X., Qiao, M., Tang, K., Zhang, C., Gui, H., Wang, P., Liu, F. Android-SEM: Generative Adversarial Network for Android Malware Semantic Enhancement Model Based on Transfer Learning. Electronics, 2022, 11, 672. doi:10.3390/electronics11050672. https://doi.org/10.3390/electronics11050672

13. Kang, B.H., Bai, Q. (Eds.). AI 2016: Advances in Artificial Intelligence. Lecture Notes in Computer Science, 2016, Vol. 9992. Springer International Publishing. doi:10.1007/978-3-319-50127-7. https://doi.org/10.1007/978-3-319-50127-7

14. Karbab, E. B., Debbabi, M., Derhab, A., Mouheb, D. Maldozer: Automatic Framework for Android Malware Detection Using Deep Learning. Digital Investigation, 2018, 24, S48-S59. doi:10.1016/j.diin.2018.01.007. https://doi.org/10.1016/j.diin.2018.01.007

15. Kiefer, D., Bauer, M., Grimm, F., van Dinther, C. 53rd Hawaii International Conference on System Sciences (HICCS), Online, January 5-8, 2021. Proceedings of the 54th Hawaii International Conference on System Sciences, 2021, University of Hawai'i at Mānoa.

16. Labs, M. McAfee Labs 2020 Threats Predictions Report. McAfee Blog, 2019. URL: https://www.mcafee.com/blogs/other-blogs/mcafee-labs/mcafee-labs-2020-threats-predictions-report/

17. Mahindru, A., Sangal, A.L. MLDroid-Framework for Android Malware Detection Using Machine Learning Techniques. Neural Computing and Applications, 2021, 33, 5183-5240. doi:10.1007/s00521-020-05309-4. https://doi.org/10.1007/s00521-020-05309-4

18. Milosevic, N., Dehghantanha, A., Choo, K. K. R. Machine Learning-Aided Android Malware Classification. Computers and Electrical Engineering, 2017, 61, 266-274. doi:10.1016/j.compeleceng.2017.02.013. https://doi.org/10.1016/j.compeleceng.2017.02.013

19. Mindermann, F. Euphony. Software Repository, 2024. https://github.com/fmind/euphony. Accessed: 25-Jul-2024.

20. Mohammed, M., Lakhan, A., Zebari, D., Abdulkareem, K., Nedoma, J., Martinek, R., Tariq, U., Alhaisoni, M., Tiwari, P. Adaptive Secure Malware Efficient Machine Learning Algorithm for Healthcare Data. CAAI Transactions on Intelligence Technology, 2023, 1-12. doi:10.1049/cit2.12200. https://doi.org/10.1049/cit2.12200

21. Nuiaa, R.R., Alomari, E.S., Mortatha, M., Alyasseri, Z., Mohammed, M., D, R.K., Manickam, S., Kadry, S., Anbar, M., Karuppayah, S. Malware Cyberattacks Detection Using a Novel Feature Selection Method Based on a Modified Whale Optimization Algorithm. Wireless Networks, 2023, 1-17. doi:10.1007/s11276-023-03606-z. https://doi.org/10.1007/s11276-023-03606-z

22. Odat, E., Yaseen, Q. M. A Novel Machine Learning Approach for Android Malware Detection Based on the Co-Existence of Features. IEEE Access, 2023, 11, 15471-15484. doi:10.1109/ACCESS.2023.3244656. https://doi.org/10.1109/ACCESS.2023.3244656

23. Poorvadevi, R., Keerthi, N. B., Lakshmi, N. V. Android Malware Identification and Detection Using Deep Learning. Proceedings of the 2022 6th International Conference on Trends in Electronics and Informatics (ICOEI), 2022, 1266-1270. doi:10.1109/ICOEI53556.2022.9776920. https://doi.org/10.1109/ICOEI53556.2022.9776920

24. Raphael, R., Mathiyalagan, P. Intelligent Hyperparameter-Tuned Deep Learning-Based Android Malware Detection and Classification Model. Journal of Circuits, Systems and Computers, 2023, 32. No Access. https://doi.org/10.1142/S0218126623501918

25. Şahin, D., Kural, O., Akleylek, S. A Novel Permission-Based Android Malware Detection System Using Feature Selection Based on Linear Regression. Neural Computing and Applications, 2023, 35, 4903-4918. doi:10.1007/s00521-021-05875-1. https://doi.org/10.1007/s00521-021-05875-1

26. Siddiqui, S., Khan, T. An Overview of Techniques for Obfuscated Android Malware Detection. SN Computer Science, 2024, 5. doi:10.1007/s42979-024-02637-3. https://doi.org/10.1007/s42979-024-02637-3

27. Statista. Number of Active Android Users Worldwide From 2016 to 2022 (in Millions). Statista Research Portal, 2022. URL: https://www.statista.com/statistics/687566/number-of-android-users/. Retrieved January 22, 2022.

28. Vinayakumar, R., Alazab, M., Soman, K.P., Poornachandran, P., Venkatraman, S. Robust Intelligent Malware Detection Using Deep Learning. IEEE Access, 2019, 7, 46717-46738. doi:10.1109/ACCESS.2019.2906934.https://doi.org/10.1109/ACCESS.2019.2906934

29. Wu, Q.C., Zhu, X., Liu, B. A Survey of Android Malware Static Detection Technology Based on Machine Learning. Mobile Information Systems, 2021. (No volume/pages available.) https://doi.org/10.1155/2021/8896013

30. Zhang, Y., Yang, Y., Wang, X. A Novel Android Malware Detection Approach Based on Convolutional Neural Network. ACM International Conference Proceedings Series, 2018, 144-149. doi:10.1145/3199478.3199492. https://doi.org/10.1145/3199478.3199492

31. Zheng, Q., Tian, X., Yu, Z., Wang, H., Elhanashi, A., Saponara, S. DL-PR: Generalized Automatic Modulation Classification Method Based on Deep Learning with Priori Regularization. Engineering Applications of Artificial Intelligence, 2023, 122, 106082. https://doi.org/10.1016/j.engappai.2023.106082

32. Zheng, Q., Zhao, P., Zhang, D., Wang, H. MR-DCAE: Manifold Regularization-Based Deep Convolutional Autoencoder for Unauthorized Broadcasting Identification. International Journal of Intelligent Systems, 2021, 36, 7204-7238. https://doi.org/10.1002/int.22586

33. Zhu, H., Li, Y., Li, R., Li, J., You, Z., Song, H. SEdMDroid: An Enhanced Stacking Ensemble Framework for Android Malware Detection. IEEE Transactions on Network Science and Engineering, 2021, 8, 984-994. doi:10.1109/TNSE.2020.2996379 https://doi.org/10.1109/TNSE.2020.2996379