Android Malware Detection Using Static Features of Mobile Applications

SAIMA AKBAR

01-241212-008

A thesis submitted in fulfillment of the requirements for the award of the degree of Master of Science

(Software Engineering)

Department of Software Engineering

BAHRIA UNIVERSITY ISLAMABAD

SEPTEMBER 2023

APPROVAL FOR EXAMINATION

Scholar's Name: Saima Akbar **Registration No.: 01-241212-008**

Program of Study: MS. (Software Engineering)

Thesis Title: Android Malware Detection Using Static Features of Mobile Applications

It is to certify that the above student's thesis has been completed to my satisfaction and, to my belief, its standard is appropriate for submission for evaluation. I have also conducted a plagiarism test of this thesis using HEC-prescribed software and found a similarity index of **12%** which is within the permissible limit set by the HEC for the MS degree thesis. I have also found the thesis in a format recognized by the BU for the MS thesis.

Principal Supervisor's Signature: ____________

Date: 9/10/23

Name: Dr Tamim Khan

AUTHOR'S DECLARATION

I, Saima Akbar hereby state that my MS thesis titled "Android Malware Detection Using Static Features of Mobile Applications" is my work and has not been submitted previously by me for taking any degree from this university Bahria University Islamabad or anywhere else in the country/world.

At any time if my statement is found to be incorrect even after my graduation, the University has the right to withdraw/cancel my MS degree.

SAIMA AKBAR
01-241212-008

PLAGIARISM UNDERTAKING

I, Saima Akbar, solemnly declare that the research work presented in the thesis titled "Android Malware Detection Using Static Features of Mobile Applications" is solely my research work with no significant contribution from any other person. Small contribution/help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero-tolerance policy of the HEC and Bahria University towards plagiarism. Therefore, I as an Author of the above-titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred to/cited.

I undertake that if I am found guilty of any formal plagiarism in the above-titled thesis even after the award of my MS degree, the university reserves the right to withdraw/revoke my MS degree and that HEC and the University have the right to publish my name on the HEC / University website on which names of scholars are placed who submitted plagiarized thesis.

SAIMA AKBAR
01-241212-008

DEDICATION

Dedicated to all those who have worked like a restless ghost, and haunted by the specters of the truth of knowledge, those elusive fragment of knowledge that our society stubbornly refused to accept.

My teachers and parents who have put efforts

And

Especially to

Dr. Tamim Ahmad Khan

ACKNOWLEDGEMENTS

First and foremost, praises and thanks to Allah Almighty, for showering of His blessings throughout our research work for its successful completion.

I would like to express my deep and sincere gratitude to our research supervisor, **Professor Tamim Ahmad Khan** for giving us the opportunity to do research and providing invaluable guidance throughout this research. It was a great privilege and honor to work and conduct research under his able guidance. I am extremely grateful for what he has offered us. I would also like to thank him for his empathy, courteous nature and great sense of humor. I am extremely grateful to the management of Bahria University for their genuine support to complete this thesis successfully.

SAIMA AKBAR

01-241212-008

ABSTRACT

The increasing use of android mobile devices and the complexity of applications have led to increase in malware threats, Information, and demanding robust security measures for safeguarding user privacy. We investigate the use of deep learning techniques in detection of of 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 provide APK analysis for a feature extraction mechanism capable of extracting a total of 43,377 features from a dataset comprising 1201 each malware classes in total 13,211 malware and 1201 benign applications. After meticulous selection, we retain only 10,524 features, which are subsequently used to train the neural networks. This dataset enables thorough evaluation and validation of the proposed detection system. We make use of APK extracted from ANDROZOO for the purpose of dataset generation. Performance metrics which is used in this research are detection accuracy, recall, F1-score and precision are utilized to determine the efficacy of the enhanced detection approach. This research explores the effectiveness of convolutional neural network (CNN) and deep neural network (DNN) models for Android malware detection using static features. By utilizing our own dataset, we evaluate the performance of both models and compare their accuracy rates. Our results demonstrate that the DNN model accuracy rate of 97%, which is outperforming the CNN model, which achieves a slightly lower accuracy rate of 96%. Transfer Learning (TL) based model also achieves a slightly lower accuracy rate of.94% but has the advantage to classify unseen or zero-day attacks. These findings highlight the potential of DNN-based approaches in enhancing the detection and prevention of Android malware, showcasing their superiority over the CNN as well TL based classifiers. The evaluation also highlights the importance of considering an expanded number of malware classes, as it significantly enhances the system's capability to detect diverse malware families both known and unknown malwares.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS:

APK	Android Application Package
AMD	Andriod Malware Dataset
API	Application Programming Interface
AUC	Area Under Curve
CSV	Comma Seperated Values
CNN	Convolutional Neural Network
DL	Deep Learning
DNN	Deep Neural Network
FP	False Positive
FN	False Negative
FPR	False Positive Rate
GPU	Graphic Processing Unit
ML	Machine Learning
RELU	Rectified Linear Unit
ROC	Receiver Operator Characteristic
RNN	Recurrent Neural Network
SMOTE	Synthetic Minority Oversampling Techniques
TP	True Positive
TN	True Negative
TL	Transfer Learning
URL	Uniform Resource Locator

1. INTRODUCTION

An overview of Android malware chapter as well as the motivation behind our study, Existing gaps and research questions along with our research objectives is discussed in this chapter. Also we have provided an overview of our thesis approach and the constraints of our research.

1.1. OVERVIEW:

Everyday use of mobile devices, particularly using the Android operating system, has increased with the time. Thereof, malware attacks on these devices are becoming increasingly usual. In a malware, the attacker develops a program with the aim of damaging a computer system without the user's consent. For Android mobiles, the third-party app stores, Google Play Store is recently penetrated by malware, which poses serious privacy and security risks for users[1][2]. Over 2.56 million mobile applications (Apps) were available for download in the Google Play Store alone as of the first quarter of 2022, according to Statista[3].

We can make use of static or dynamic features of an application to determine whether an Android App is possibly benign or malicious. Android malware detection and prediction process make use of datasets that may contain static or dynamic features, extracted from either the APK of the application without executing or from the emulator based execution. The static analysis features include permissions, API calls, and strings etc. To increase the accuracy of Android malware detection, machine-learning methods have been applied to these static features [1]. With regard to identifying Android malware via static feature analysis, deep learning approaches, such as recurrent neural networks (RNNs), and convolutional neural networks (CNNs) have demonstrated promising results[4].

In short, we examine how well static features, CNNs, and DNNs, perform in identifying Android malware detection. Our research intends to contribute to the development of efficient static analysis-based methods and more reliable for Android malware detection.

1.2. MOTIVATION

The detection of Android malware using static features remains an important research area due to the increasing number of mobile malware attacks. The understanding of how to extract pertinent static features from Android applications and to develop efficient deep-learning models for malware detection remains an important initial step for Android malware detection process [5]. Increasing the precision and effectiveness of Android malware detection using static features is one of the main goals of this research.

Static analysis-based malware detection techniques may be able to do this. Consequently, the goal of the study is to create advanced static analysis-based tools that can efficiently find attacks. Our collection of malwares allows us to enhance the precision of malware detection on a different collection of applications by utilizing the information obtained from studying one group of applications [6].

By achieving these goals, the research has the potential to significantly enhance the security of Android devices and protect users from the harmful effects of malware attacks.

1.3. RESEARCH GAPS

It is pertinent to note that prior research has focused either on binary where the output class is either benign or malicious [7][8] or multi-classification considering up to 10 classes or subclasses of Android malware [9][10][11]. Since new classes of Android malware are discovered, it becomes imperative to examine the efficacy of taking into account additional classes. However, it is important to note that adding more classes can decrease the classification models' accuracy and make it easier to identify newly discovered and developing malware variants. Another effect of adding or including more classes result in a more complex model with additions layers required resulting in increased processing costs.

Secondly, previous models are developed using datasets with limited number of features [12]. Investigating the usage of more features and bigger datasets can help these models perform better. It is pertinent to note that larger datasets can provide the models with more diverse and relevant examples, which can enhance their generalization capabilities[7]. Existing research considers limited features and the latest static feature extraction techniques can extract up to 19000 features and after applying preprocessing they get 350 features [8] from APK analysis. APK development, code complexity and feature extraction techniques for static feature extraction as well as simulator based execution for dynamic feature extraction are becoming more robust and hence returning bigger features sets. Therefore, better and more robust models involving more features and malware classes are required. A zero-day attack, also known as a zero-day exploit, is a type of cyberattack that takes advantage of a previously unknown vulnerability or software flaw in a computer system, application, or piece of software. 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 [9]. An initial evaluation of zero-day attacks was conducted using a combination of Graph Convolutional Networks (GCN) and Multilayer Perceptron (MLP). However, there is a need to incorporate more sophisticated techniques, such as transfer learning with machine learning, to effectively address zero-day attacks[10].

1.4. PROBLEM STATEMENT:

Early malware threat detection can help avoiding possible malicious activities performed by Android Malware. There are numerous Android malware families and their sub-classes, and new malware are getting introduced regularly. However, existing research considers at most 10 number of Android malware classes and sub-classes [6]. 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 Android malware detection process are required. We aim to deploy a feature extraction mechanism for dataset development. We also propose deep learning techniques based models considering more number of classes and static features and use deep-learning based Android malware detection techniques handling zero-day attacks. We raise the following research questions:

1. How can we construct datasets with larger sets of examples by considering more features and families/classes?
2. How can we develop a deep learning approach based models that make use of a bigger range of malware classes and features in the dataset and improve efficiency of existing systems?
3. What is comparison of deep learning-based classifiers that can be employed to identify malware with and without the possibility of handling zero-day attacks?

1.5. RESEARCH OBJECTIVES

Objective 1:

Investigate existing methods and techniques for constructing datasets with larger sets of examples by incorporating additional features and families/classes, with a focus on enhancing dataset diversity and representativeness.

Objective 2:

Explore deep learning-based approaches for classifying malware by utilizing a broader range of malware classes and features in the dataset. Evaluate the effectiveness of these models in improving the efficiency and accuracy of existing malware detection systems.

Objective 3:

To investigate and develop the effective techniques and strategies for the proactive detection of zero-day attacks considering static features and comparing accuracy.

1.6. THESIS METHODOLOGY AND LIMITATIONS:

The methodology for collecting the dataset of Android applications for this research includes different strategies such as web crawling, downloading apps from applications stores, and using third-party sources. Once the data is gathered, useful static features are extracted using a variety of methods, such as applications code dissection, API call analysis, and manifest file inspection. Permissions, network connections, API requests, and other application features that may be indicative of malicious behavior are some examples of these static features. For malware detection to categorize applications into malicious or benign, neural network architectures are used in deep learning models, for example, Convolutional neural networks (CNNs) and deep neural networks (DNNs). The quality and variety of the training dataset and the quality amount of the extracted features have a very visible impact on how accurate these models are. The use of deep learning models for malware detection is not without limitations though. The challenge of getting a sizable and varied collection of malware and benign applications is one issue. Additionally, it might be difficult to determine between benign and malicious behavior in some circumstances, such as when an application uses APIs that could be viewed suspiciously or requests particular rights. Furthermore, given the continually evolving nature of malware, it can be difficult for deep learning models to generalize successfully to new and untested malware samples, which is a need for their efficacy. The Research Methodology which we follow is depicted in *Figure 1.1*.

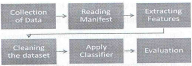


Figure 1.1: Research methodology steps

1.7. RESEARCH CONTRIBUTIONS

The collection of a balanced dataset is one of the major contributions of our study to the area of Android malware detection. The quantity and quality of data play a key role in determining how accurate a deep learning model is. Our dataset has a balanced distribution of malware and benign applications, giving each class an equal representation. A dataset like this guarantees the development of more reliable and accurate models, which can help with malware identification.

An important contribution to our research is that we have concentrated on adding more malware class variations to our dataset. Traditional malware detection methods might not be sufficient for detecting new types of harmful behavior because of how quickly malware is evolving. Therefore, incorporating a wider variety of malware classifications enables more precise and delicate malware detection. This expansion of malware types may also provide light on the basic features of harmful behavior, assisting in advancing the development of more effective detection methods.

Another contribution of our research is the extraction of a more comprehensive set of features from Android application. Identifying the features that matter for malware classification is a big challenge for deep learning-based malware detection. We face with this challenge by extracting a larger number of features from our dataset, leading to more accurate and precise deep-learning models. This feature extraction comprises the study of numerous features, including permissions, API calls, and network connections.

1.8. THESIS ROADMAP

After the abstract and introduction, the roadmap of research are as follows: Literature review on the topic of Android malware detection and static analysis is explained in Chapter 2. In Chapter 3 methodology that was used in this study is extensively explained In Chapter 4 explained results with a discussion of the findings that follows. Chapter 5 concludes by discussing the study's contributions and offering ideas for more research. With this structure in place, the reader will be able to fully understand the issue completely, from the literature review to the technique used, and finally, the findings and contributions of the research. In we depict the overview of thesis which we follow.

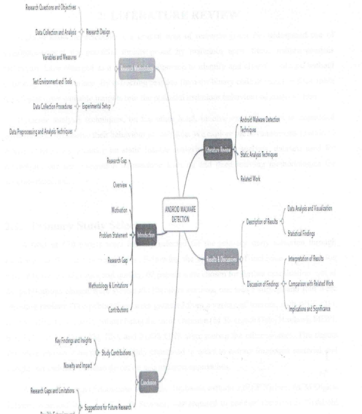


Figure 1.2: Overview of Thesis

2. LITERATURE REVIEW

Android malware detection is a crucial area of research given the widespread use of smartphones and the potential threats posed by malicious apps. Static feature analysis techniques have emerged as a prominent approach to identify and classify malware without executing the applications. By extracting features from the binary code or manifest files, these techniques offer valuable insights into the potential malicious behaviour of Android apps.

Dynamic analysis techniques, on the other hand, involve executing apps in controlled environments to observe their behaviour in real-time. We explore the advancements in android malware detection, focusing on static feature analysis, dynamic analysis, datasets used for evaluation, and the integration of machine learning and deep learning methodologies for accurate detection.

2.1. Primary Study Selection:

A total of 120 papers were initially selected for the primary study selection through database searches and other sources. Following the application of inclusion and exclusion criteria based on relevance and quality, 62 papers were chosen for further examination. Six of the publications chosen were systematic literature reviews, one was a survey, and four were literature reviews. The publications were gathered from a variety of sources, with IEEE (31), Science Direct (11), and Springer being the most common (5). Research Gate, Hindawi, MDPI, SagePub, ACM, ITCKTU, JISI, and PLOS ONE were among the other sources. The papers that were chosen were then thoroughly examined in order to extract important material and insights on android malware detection and analysis approaches.

A thorough search of numerous academic databases, including IEEE Xplore, ACM Digital Library, ScienceDirect, and Google Scholar, was required to conduct the review. "Android malware detection," "Android malware analysis," "Android malware classification," "machine learning," "deep learning," "static analysis," "dynamic analysis," "hybrid analysis," and "feature-based analysis" were among the search terms utilized. The search was restricted to studies published in English between 2010 and 2022.

At the outset, the initial search yielded 120 research papers. After conducting further screening based on the relevance of the publications, we ultimately selected 62 primary studies and 7 review papers for inclusion in this systematic literature review.

The data extraction approach included noting the year of publication, dataset(s) used, analysis technique(s) used, and study limitations. The gathered data was sorted and analyzed in order to determine the important themes and findings about android malware detection and analysis methodologies and tools.

2.2. Data Extraction:

This sub-phase involves obtaining essential data from the chosen research. The data retrieved from each research comprises the proposed static analysis approach, the evaluation methodology, the results, and the limitations of the suggested technique. A predetermined form was used to extract the data. Each study's data was retrieved by two independent reviewers, and any disagreements were handled by consensus. The gathered data was organized and synthesized to address the study objectives. The extracted information in the data extraction is listed in *Figure 2.1*.

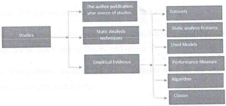


Figure 2.1: The extracted data/information

During the data extraction phase, a CSV file was created to systematically record relevant information from the selected studies. The CSV file consisted of the following header columns:

1. **Source:** This column recorded the source of the study, such as IEEE, Science Direct, Springer, Research Gate, Hindawi, MDPI, SagePub, ACM, Itcktu, JISI, and Plose-One.
2. **Cite:** This column reported the total number of citations.
3. **AI Model:** This column documented the type of artificial intelligent models such as machine learning or deep learning.
4. **Dataset:** The name of the dataset used in the study for training and testing the machine learning model was recorded in this column.
5. **Accuracy Metric:** This column documented the type of accuracy metric used in the study, such as precision, recall, F1 score, accuracy, and others.
6. **Accuracy Percentage:** This column records the percentage value of the accuracy metric achieved by the machine learning model in the study.
7. **Limitations:** This column records any limitations or downsides of the study in terms of dataset selection, model design, evaluation criteria, or other elements.
8. **Link:** For future reference, this column recorded the hyperlink to the full-text PDF of the paper.
9. **Date:** The study's publication date was noted in this column.
10. **Algorithm:** This column documented the machine learning algorithm used in the study, such as J48, SVM, KNN, Naive Bayes, and others.
11. **Features:** This column recorded the set of features used in the study for machine learning model training and testing.
12. **Classes:** The classes of malware or benign apps utilized in the study for machine learning model training and testing were recorded in this column.

The data extraction phase entailed the methodical extraction and recording of pertinent information from each of the 62 chosen research. The CSV file provided as a thorough and ordered record of the retrieved data, allowing for additional analysis and synthesis of the results.

2.3. Data Synthesis:

The data synthesis step produced various tables and graphs that gave for a better understanding of the properties of the AI models used to detect Android malware. The most widely used characteristics were Permissions, API calls, System calls and Opcode with most studies incorporating both into their AI models. However, some research used fewer common features, such as segment entropy and creator information, showing the need of studying multiple feature sets for malware identification.

The Figure 2.2 lists 22 different types of features used in the previous study, each with a corresponding count of the number of times that feature was used across all the apps analysed.

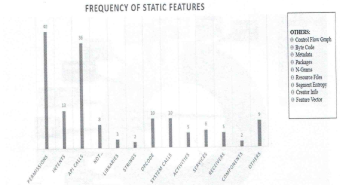


Figure 2.2: Categories of Android Features

The various malware classifications that the selected studies were designed to identify. Adware, Trojan and Backdoor malware etc were the most frequently detected malware classifications, while Worm and Scareware etc were less frequently targeted. This data can assist researchers to identify which malware classes pose the most serious threats to Android devices and may help guide future research efforts.

Researchers have explored various malware classes in their studies. In the comprehensive analysis of malware classes conducted during the background research for this thesis, the *Figure 2.3* show the malware is classified into 22 distinct categories. We find all the studies considered benign, on the other hand 48 studies considered malware. A total of 8 studies conducted on Adware, 6 studies on SMS Trojan, 1 study on Phishing, 1 study on Data Stealer, 3 studies on Rootkit, 2 studies on Botnet, 1 study on ClickFraud, 1 study on DDOS, 8 studies on Ransomware, 7 studies on Trojan, 4 studies on Backdoor, 1 study on Riskware, 3 studies on Spyware, 3 studies on ScareWare, 2 studies on Worms, and 1 study each on Dialer, Downloader, Rouge, and Pws. This diverse representation of malware classes underscores the comprehensive nature of the research conducted in this domain.

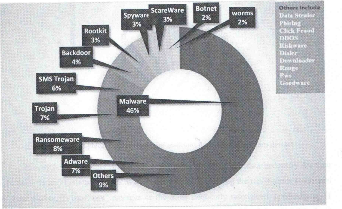


Figure 2.3: Classes used in primary studies

In a comprehensive analysis of studies that have employed various datasets in the field, it was observed that the "Drebin" dataset was the most frequently used, appearing in 17 research studies. "Genome" and "Private Dataset" datasets were also cited frequently, with 8 studies each making use of these resources. "AMD (CICMAL2017)" was utilized in 7 studies, while "CICInvesAndMal2019" was referenced in 2 research studies. Other datasets such as "Kaggel," "MoDroid," "Ember," "Microsoft Malware Classification Challenge Dataset," "KuafuDet," and "Omnidroid" were utilized in one research study each. Additionally, there were 8 instances where the dataset used was not explicitly mentioned, making it challenging to attribute the specific dataset utilized in those cases. As Figure 2.4 show the dataset used by researchers.

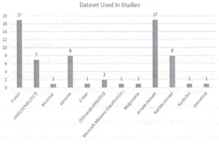


Figure 2.4: Graphical representations of malicious dataset in the primary dataset

Researchers often rely on various repositories to obtain the data necessary for their cybersecurity and malware studies as shown in Figure 2.5. Among the repositories mentioned in these studies, "VirusShare" emerges as the most frequently referenced, appearing in 11 research studies as a critical resource for malware samples. "Google Play," the official Android app store, follows closely with 6 mentions, emphasizing its significance in analyzing Android applications. "Contagio" serves as a valuable repository in 5 studies, while "Androozo" contributes to 3 research projects. Additionally, "Third-party app markets" were utilized in 3 studies, "Marvin" in 1 study, and in some instances, when the repository was not explicitly specified ("Not Mentioned"), researchers made use of the term "Android APK" in their studies, implying that these repositories were indeed the sources for their data.

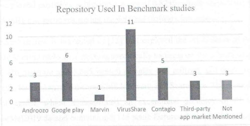


Figure 2.5: Repositories Considered in Benchmarks Studies

2.4. Our Analysis and Findings

2.4.1. Challenges in Dataset Quality

Author proposed a machine learning approach for Android malware detection, utilizing a static permission-based methodology[11]. 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.

Author in [12] aimed to 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 evaluated 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. Specifically, the Random Forest algorithm exhibited exceptional precision, achieving a score of 0.8249, which showcased its ability to accurately identify true positives among the detected malware instances.

A literature review is presented in [13] 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.

The author in [14] 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. To ensure robust feature selection, eight different feature selection algorithms were utilized, with a majority voting process leading to the selection of 19 significant features. 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.

Author worked on binary Android malware detection [15]. The proposed Deep Classify Droid detection system was a deep learning-based approach focused on distinguishing between 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 Deep Classify Droid 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.

In the research, a static analysis-based Android malware detection model was proposed using features from benign and malicious apps collected from Google Play and Virus Share [16]. 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 goodware and benign applications, contributing to a safer user experience on Android devices.

We analysed various models related to Machine Learning (ML) and Deep Learning (DL) that are being used for detecting Android malwares. These techniques are all use for detecting Android malwares. There exists a gap in the existing research landscape. While some studies are focusing on binary classification, others are exploring multi-class classification, usually with no more than 10 classes. However, the study[17] which use 10 classes lack detailed dataset descriptions and do not provide the names of the classes they are using. This gap emphasizes the need for a comprehensive exploration. Our aim is to address this gap by exploring additional classes, utilizing larger, more diverse datasets with more features. Our method makes use of deep neural networks (DNNs) and convolutional neural networks (CNNs) to efficiently manage more classes and extract important features. Therefore, our research has the potential for major improvements in Android device security and protect users from the malware attacks by fulfilling these goals. In s**Table 2.1** we show the comparison of existing literature using static analysis of application.

Table 2.1 Comparison of Existing Literature Using Static Analysis of Applications

ef	Year	Model	Dataset	Algorithums	Features	Classes	Limitation
18]	2023	dl	CICAndMal2017	(CNN and DNN	Permission , Intents	Adware, Radware, rootkit, SMS Malware, and ransomware	Limited classes consider
19]	2023	dl	Drebin and CICMaldroid200	RF and ET and DNN and 1D-CNN	permissions, intents, services, and API calls a	Adware ,Banking ,Benign,Riskware and SMS	Limited classes consider
20]	2022	dl	Derbin	logistics Regression ,Random Forest, SVM, Deep Neural Network	permission, APIs, app components and system calls (especially n-grams of system calls)	benign and malicious apps. GOOD WARE	Limited classes Used
21]	2022	dl	Private Dataset	Multilayer Perceptron (MLP)	Features not mention	Malware and Benign)	Limited size of dataset even not mention features
22]	2022	Tl	Private Dataset	GAN and quantum support vector machine (QSVM)	Static and dynamic analysis features	Malware and benign apps	Binary dataset
23]	2021	ml	Drebin	Naive Bayes (TAN) and Random forest	Permissions, API calls and intent	malware and benign	Limited number of classes

[24]	2021	ml	Drebin and Androzoo	Random forest	Permissions and API calls and Control flow graph	Malware	Consider limited feature.
[10]	2021	ml	CICAndMal19	Graph Convolutional Networks (GCN) and Multilayer Perceptrons (MLP)	Not mention	MaLware and benign	Need to incorporate more sophisticated use limited classes
[25]	2021	ml	Contagio and VirusShare and Microsoft Malware Classification Challenge	Boosted Learning and AdaBoost	System calls, API calls, Permissions, Opcode frequencies, String extraction	Zbot, Koobface, Virut, Sality, Vundo, Cutwail, Conficker, Zeus, Bredolab, and Kelihos	Small number of samples from each family, Imbalance in dataset, Use of only static analysis, Limited number of features used
	2021	ml	M0Droid dataset	Random Forest and SVM and Gaussian Naïve Bayes and K-Means	Permission	Malware and Benign	Only consider the permissions
[26]	2021	ml	APK Pure	KNN and Naive Bayes (NB) and Sequential Minimal Optimization and MLP, Random Forest and C4.5 and Logistic Regression	Permission	Trojan and Adware and Rootkit and Ransomware.	Limited feature set used
[27]	2021	ml	Private Dataset	CNN	Lines of code activates, services recivers, dangerous permissions, custom permissions, other	Malware and Benign	Work on Binary

					permission number of features		
]	2020	ml	Drebin and AMD and Genome and Malgenome	Hamming Distance (FNN) and all nearest neighbors (ANN) and weighted all nearest neighbors (WANN), and k-medoid based nearest neighbors (KMNN)	Static binary features extracted from the application binary code API, intent, and permission	Malware and Benign	Work on Binary
6]	2020	dl	Virus share,	MLP AND SVM	Permissions and API calls and manifest file features	Trojan and Adware and Ransomware, and Backdoor	Consider only 3 features permissions, Receivers, API calls
9]	2020	ml	Thirty Party app and VirusShare	Random Forest and K-Nearest Neighbor	API calls, permissions and intents. API calls	Malware and benign applications	Limited classes uses
5]	2020	ml	Not Mentioned	XGBoost algorithm	API calls, API-pair graphs, API call sequences, Execution behaviors, Permission ,and Intents	Malware and Benign	Limited number of classes
30]	2020	dl	Google play and Virus share	DNN	API calls and System calls and Permissions	Malware and Benign	only work on binary classification
14]	2019	dl	CICAndMal20	LSTM ,RNN	Permission Intents, API calls, Opcode sequences ,Bytecode sequences, System call sequences	Adware and ransomware and scareware and SMS	Used limited feature set

1]	2019	dl	Private Dataset	DNN	Permissions,Intent,API and system calls	Malware and benign	limited set of features
2]	2019	dl	VirusShare and Drebin and Contagio, and Androzoo, McAfee Labs	CNN	Permissions and API calls and Intents and System calls	Malware and bening	The main limitation of the study is that the model was only tested on a small dataset. The authors also did not evaluate the model's performance on new and unseen malware samples.
9]	2019	dl	Ember and VirusTotal and VirusShare and private dataset	Logistic Regression (LR) and Navie Bayes (NB) , K-Nearest Neighbor (KNN) ,Decision Tree (DT) ,Random Forest (RF),SVM and CNN and DNN	System calls	Dailer, Backdoor and worm and trojan and wormautoit and trojan and downloader and rouge and pws	Limited classes uses
3]	2019	dl	Drebin and Contagio, and Genome.	FalDroid and FNN and ANN and WANN and KMNN	Permissions and API calls and Intents and system calls.	malware and bening	Static analysis of binary files, limited to certain types of malware
4]	2018	ml	Google play	SVM	Permission	Malware and Benign	Use on one feature

[35]	2018	dl	Kaggle	RNN and CNN	not mention	malware Benign	Not mention features and only work on binary classes
[36]	2018	dl	Drebin and MARVIN	DNN	Permission and API calls	Malware and Benign	Consider only Binary classes
[7]	2018	dl	Drebin, Genome, Virus Share	ANN	API calls a, Permission s, Strings, Opcode	Adware and Ransomware and Rootkit and SMS Malware and Spyware and Trojan.	Limited family
[37]	2017	dl	Android Apk files	CNN and DNN	API calls and Permission s, and Third-party libraries.	Trojan and Adware and Riskware and Ransomware and Benign.	Limited feature set
[17]	2016	ml	Not Mentioned	SVM	Permission s and API calls and Intent	Adware and SMS malware and SMS phishing and Data theft and Rooting malware, and Botnet and Click fraud and DDoS malware and Ransomware, and Remote access Trojan (RAT)	It is unclear whether the system will be effective against new or previously unknown malware.
[38]	2016	ml	Android Apk files	DNN	Not Mentioned	Malware and Benign	The feature set is very limited.
[39]	2016	dl	Private Dataset	CNN	Opcode sequences and API calls	Malware and benign apps	The dataset is not publicly available
[40]	2016	ml	Google Play Store and Virus share and Third Party app	KNN and Logistic Regression and BN	API CALL	benign and malicious apps.	The paper does not discuss the scalability of the proposed approach or the performance on larger datasets.

1]	2016	dl	Virus Share and Maltrieve and private Dataset	Markov Models and SVM and NN	API call sequences and system call	Malware and Benign	Do not provide a detailed breakdown of the specific malware families or classes included in the dataset.
2]	2016	dl	Genome and Play store	RF and SVM and NN	API calls and Permission s	malware and Benign	Benign used 5000 and only use 1000 malware means limited malware
43]	2016	dl	Google playand Drebin and Genome and Contagio	SVM and CNN	Permission s, API calls,Intent s, library calls	Malware and Benign	Lack of explanation for the feature selection process
44]	2015	ml	Not Mentioned	SVM	App Permission s and API calls	Malware and Benign	Limited to certain types of malware
45]	2013	Not Menti oned	Not Mentioned	Boosted and J48 algorithm	High-dimensiona l static features	Malicious and benign applications	Work only on binanry

Variable data quality is an issue in the context of Android malware detection utilizing static feature datasets, according to the results of the systematic literature review. The choice of suitable datasets for training and testing models is one of the major issues in this domain. The major research' usage of various datasets highlights how crucial it is to test AI models on a range of datasets in order to verify their robustness and generalizability. However, this variation in dataset utilization might also result in inconsistencies that could reduce the accuracy of the algorithms' predictions.

As stated in Table 2.2 Dataset Used In Primary Studiesthe most frequently utilised datasets in the primary studies were Drebin, Private Datasets, VirusShare, and the AMD. Drebin appeared in 20 articles, [46], [24], [36], [7],[47], [23], [48], [33], [49], [50], [51], [20], [52], [53], [54], [55], [13], [56], [32], [57] while Private Datasets were utilised in 1 articles [58], [59], [41], [9], [49], [21], [39], [60], [27], [28], [22], [61], [62], [63], [64], [13], [57]. Virus Share, the third most frequently utilised dataset, was used in 12 studies. [16], [7], [30],

[47], [41], [9], [48], [65], [66], [29], [40], [56]. These dataset's variances in the amount of malicious and benign samples, however, can lead to issues when the model is being trained.

For instance, the model may be biased towards identifying malware if a dataset contains more malware samples than benign samples, and the other way around. This inconsistency can also cause the model to be over fitted or under-fitted which may reduce the predictability of the results.

Table 2.2 Dataset Used In Primary Studies

Dataset	Reference
Drebin	[46], [24], [36], [7],[47], [23], [48], [33], [49], [50], [51], [20], [52], [53], [54], [55], [13], [56], [32], [57]
Google play	[67], [30], [47], [42], [51], [40]
Androzoo	[24], [48], [4], [8]
M0Droid	[68]
Not Mentioned	[17], [25], [44], [45], [69], [70], [71]
APK Pure	[26]
AndroidApk files	[38], [37], [16]
MARVIN	[36]
Genome	[7], [42], [33], [49], [51], [52], [32]
Virus share	[16], [7], [30], [47], [41], [9], [48], [65], [66], [29], [40], [56]
CICAndMal2017	[14], [18]
Private Dataset	[58], [59], [41], [9], [49], [21], [39], [60], [27], [28], [22], [61], [62], [63], [64], [13], [57]
CICInvesAndMal2019	[72]
AMD	[47], [4], [73], [74], [52], [75], [53], [8]
Contagio	[7], [48], [33], [51], [66]
Maltrieve	[41]
VirusTotal ,Ember	[9]
McAfee Labs	[48]
KuafuDet,Omnidriod	[4]

Microsoft Malware Classification Challenge	[66]
Thirty Party app	[29], [40]
Kaggle	[35]
Malgenome	[52]
CICMaldroid2020	[54]
Minidump	[7]
MARVIN	[26]

These results illustrate that when analysing their suggested approaches for Android malware detection using static feature datasets, researchers only use a limited amount of datasets. It is essential to use relevant datasets that are typical of real-world situations and contain a balanced distribution of malicious and benign samples in order to effectively manage this problem. Standardised techniques for dataset selection and preparation can be used to achieve this. In order to deal with the problem of high-dimensional data, it is also crucial to utilise appropriate feature sets that are customised to the targeted datasets and to apply efficient feature selection and dimensionality reduction techniques.

We find feature sets and they are:

1. Application Tags
2. Feature Tags
3. Library Tags
4. Meta Data Tags
5. Permission Tags
6. Provider Tags
7. Receiver Tags
8. Service Tags

We consider the following for the analysis of each of these in following steps.

Step 1. Display Dataset Summary of Application/ Feature/ Library/ Meta/ Permission/ Provider/Receiver/ Service Tags.

Step 2. Calculate Summary Statistics of Application/ Feature/ Library/ Meta/ Permission/ Provider/Receiver/ Service Tags.

Step 3. Retrieve DataFrame Column Information of Application/ Feature/ Library/ Meta/ Permission/ Provider/Receiver/ Service Tags.

Step 4. Histogram of Malware Class Frequencies of Application/ Feature/ Library/ Meta/ Permission/ Provider/Receiver/ Service Tags.

Step 5. Visualization of Feature Usage within the Subset for Application/ Feature/ Library/ Meta/ Permission/ Provider/Receiver/ Service Tags.

However, it is pertinent to discuss the implications and insights gained for the Application Tags, Library Tags, Feature Tags, Meta Tags, Permission Tags, Provider Tags, Receiver Tags, Service Tags in each of these analysis steps. In Table 2.3 we show the feature set which we have considered for this research.

Table 2.3: Our Data Analysis

Feature Set	Extracted Sub Features
Application Tags	7
Feature Tags	91
Library Tags	23
Meta Data Tags	14053
Permission Tags	4163
Provider Tags	5967
Receiver Tags	8586
Service Tags	10498

2.4.2. **Data Analysis:**

2.4.1.1. **Feature Set 1: Application Tags in AndroidManifest.xml**

Step 1: Display Dataset Summary Of Application Tag:

To provide an overview of the dataset, a summary of application tags can be created. This table will include relevant columns and statistics that capture key information about the dataset. In Table 2.4 show some of the dataset summary of Application tag.

Table 2.4: Display Dataset Summary Of Application Tag

	Allow Backup_@bool/customAllowBackup	Allow Backup_false	Allow Backup_true	-	Uses Clear text Traffic true	Activity Count	Malware Class
0	0	0	1	-	0	80	Adware
1	0	0	0	-	0	155	Adware
2	0	0	0	-	0	14	Adware
3	0	0	0	-	0	7	Adware
4	0	0	1	-	0	25	Adware

This summary table provides a quick overview of the dataset, showcasing the values in each column for the first few rows. It includes columns such as **allowBackup_@bool/customAllowBackup**, **allowBackup_false**, **allowBackup_true**, **usesCleartextTraffic_true**, **Activity Count**, and **Malware Class**. Each row represents an instance or record in the dataset related to the application tags in android manifest.xml

Step 2: Calculate Summary Statistics Of Application Tag:

In order to gain a better understanding of the dataset, summary statistics can be computed for the Data Frame. These statistics provide insights into the central tendencies, dispersion, and distribution of the dataset's numerical columns. Based on the provided statistics, the following summary statistics can be obtained. In Table 2.5 show the summary of statistics some of columns of application tag

Table 2.5: Calculate Summary Statistics Of Application Tag

	AllowBackup_ @bool/custom AllowBackup	AllowBackup _false	AllowBackup _true	Uses Cleartext Traffic_true	Activity Count
Count	14,376	14,376	14,376	14,376	14,376
Mean	0.000974	0.067752	0.421605	0.023929	34.81566 5
Std	0.031192	0.251328	0.493833	0.152833	43.82027
Min	0	0	0	0	0
25%	0	0	0	0	7
50%	0	0	0	0	19
75%	0	0	1	0	46
Max	1	1	1	1	485

Step 3: Retrieve DataFrame Column Information

To gain a better understanding of the DataFrame's columns, it is essential to retrieve information such as column names, non-null counts, and data types. In Figure 2.6 show the details of the retrieve data frame column information. Based on the provided details, the following table presents the column information for the DataFrame

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14376 entries, 0 to 14375
Data columns (total 6 columns):
 #   Column                              Non-Null Count  Dtype
---  ------                              --------------  -----
 0   allowBackup_@bool/customAllowBackup 14376 non-null  int64
 1   allowBackup_false                   14376 non-null  int64
 2   allowBackup_true                    14376 non-null  int64
 3   usesCleartextTraffic_true           14376 non-null  int64
 4   Activity Count                      14376 non-null  int64
 5   Malware Class                       14376 non-null  object
dtypes: int64(5), object(1)
memory usage: 674.0+ KB
```

Figure 2.6: Retrive Data Frame Column Information Of Application Tag

Below information provides a comprehensive overview of the DataFrame's column information. It includes the range of the index (from 0 to 14375) and the total number of columns (6). Each row represents a column and contains the following details:

- Column: The name of the column.

- Non-Null Count: The number of non-null values present in the column.
- Dtype: The data type of the column.

In this particular DataFrame, there are five integer columns (**Allow Backup_@bool/custom AllowBackup,-allowBackup_false,-allowBackup_true, Uses Clear text Traffic true**, and **Activity Count**) and one object column (**Malware Class**).

Step 4: Histogram of Malware Class Frequencies

We plot a histogram to examine the distribution of malware class frequencies within the dataset. The histogram provides a visual depiction of the frequency count for each of the 12 malware class present in the dataset. Remarkably, all the malware classes exhibit an equal frequency count of 1,198, indicating a balanced representation.

The x-axis of the histogram corresponds to the distinct malware class labels, while the y-axis represents the frequency count. The histogram plot conveys a symmetrical distribution of frequencies among the diverse malware classes, highlighting a proportional representation of each class within the dataset. For a comprehensive understanding of the histogram plot, please refer to the Figure 2.7. This visualization offers valuable insights into the distribution patterns and relative frequencies of the different malware class variants in the dataset, contributing to a comprehensive analysis of the dataset.

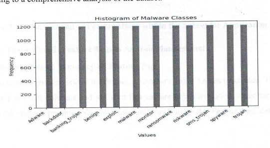


Figure 2.7: Histogram of Malware Class Frequencies Of Application Tag

Step 5: Visualization of Feature Usage within the Subset for Application Tag

A graphical representation was created to examine the usage of features within the subset of the dataset being analyzed. This graph focuses on the sum of integer columns while excluding the 'apkname', 'activity count', and 'malware class' columns to avoid redundancy. Moreover, a minimum threshold of 5 was applied to include only significant columns, ensuring a clear and concise analysis.

The x-axis of the graph denotes the different features present in the subset, while the y-axis represents the count of APK samples utilizing each specific feature. This visualization provides valuable insights into the prevalence and adoption of different features within the subset, enabling an understanding of which features are commonly employed by the APK samples. For a comprehensive understanding of the feature usage patterns within the analyzed subset, please refer to the Figure 2.8. This visualization offers a concise overview of the feature utilization landscape within the analyzed dataset subset, facilitating the identification of prominent features employed by the APK samples.

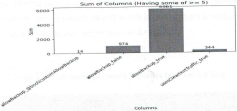


Figure 2.8: Visualization of Feature Usage within the Subset for Application Tag

2.4.1.2. Feature Set 2: Feature Tags in AndroidManifest.xml

Step 1: Display Dataset Summary Of Feature Tag

To provide an overview of the dataset, a summary Table 2.6 can be created. This table will include relevant columns and statistics that capture key information about the dataset. Based on the example provided, the following table summarizes the Feature Tags dataset.

Table 2.6: Display Dataset Summary Of Feature Tag

	FEATURE: Android. hardware. autofocus	FEATURE: Android. hardware. Location	...	FEATURE: Android.software. live_wallpaper	FEATURE :Android. software. mode	Activity Count	Malware Class
0	0	0	...	0	0	80	Adware
1	0	1	...	0	0	155	Adware
2	0	0	...	0	0	14	Adware
3	0	0	...	0	0	25	Adware

Step 2: Calculate Summary Statistics for Feature Tag

To help us understand the dataset better, we have added a Table 2.7 that summarizes the summary statistics of the features. This table is a significant resource for acquiring insights into the dataset, allowing us to study key statistical metrics and better understand the data's distribution, variability, and properties. we will be able to gain a better knowledge of the dataset and perform informed analysis and interpretation of the results.

Table 2.7: Calculate Summary Statistics for Feature Tag

	FEATURE: Android.hardware .autofocus	FEATURE: android.hardware. bluetooth	...	FEATURE: android.hardware. vulkan	FEATURE: android.software .vr.mode	Activity Count
OUNT	14376.000000	14376.000000	...	14376.000000	14376.000000	14376.000000
IEAN	0.001530	0.009878	...	0.000835	0.001878	34.923553
TD	0.039091	0.098897	...	0.028881	0.043298	43.701594
IIN	0.000000	0.000000	...	0.000000	0.000000	0.000000
5%	0.000000	0.000000	...	0.000000	0.000000	7.000000

50%	0.000000	0.000000	...	0.000000	0.000000	20.000000
75%	0.000000	0.000000	...	0.000000	0.000000	46.000000
MAX	1.000000	1.000000	...	1.000000	1.000000	485.000000

Step 3: Retrieve DataFrame Column Information Of Feature Tag

The Table 2.8 provides a quick overview of the DataFrame's column information. It includes the range of the index (from 0 to 14375) and the total number of columns (6).

Table 2.8: Retrieve Data Frame Column Information Of Feature Tag

#	Column	Non-Null Count	Dtype
0	FEATURE:android.hardware.autofocus	14376	int64
1	FEATURE:android.hardware.bluetooth	14376	int64
2	FEATURE:android.hardware.bluetooth_le	14376	int64
3	FEATURE:android.hardware.camera	14376	int64
4	FEATURE:android.hardware.camera.any	14376	int64
5	FEATURE:android.hardware.camera.autofocus	14376	int64
6	FEATURE:android.hardware.camera.flash	14376	int64
7	FEATURE:android.hardware.camera.front	14376	int64
8	FEATURE:android.hardware.camera2.full	14376	int64
9	FEATURE:android.hardware.location	14376	int64
10	FEATURE:android.hardware.location.gps	14376	int64
11	FEATURE:android.hardware.location.network	14376	int64
12	FEATURE:android.hardware.microphone	14376	int64
13	FEATURE:android.hardware.nfc	14376	int64
14	FEATURE:android.hardware.nfc.hce	14376	int64
15	FEATURE:android.hardware.screen.landscape	14376	int64
16	FEATURE:android.hardware.screen.portrait	14376	int64
17	FEATURE:android.hardware.sensor.accelerometer	14376	int64
18	FEATURE:android.hardware.sensor.compass	14376	int64
19	FEATURE:android.hardware.telephony	14376	int64

20	FEATURE:android.hardware.touchscreen	14376	int64
21	FEATURE:android.hardware.touchscreen.multitouch	14376	int64
22	FEATURE:android.hardware.touchscreen.multitouch.distinct	14376	int64
23	FEATURE:android.hardware.vulkan	14376	int64
24	FEATURE:android.hardware.vulkan.version	14376	int64
25	FEATURE:android.hardware.wifi	14376	int64
26	FEATURE:android.software.leanback	14376	int64
27	FEATURE:android.software.live_wallpaper	14376	int64
28	FEATURE:android.software.vr.high_performance	14376	int64
29	FEATURE:android.software.vr.mode	14376	int64
30	Activity Count	14376	int64
31	Malware Class	14376	object

Step 4: Histogram of Malware Class Frequencies Of Feature Tag:

We also plot a histogram for the feature tag to examine the distribution of malware class frequencies within the dataset. The histogram provides a visual depiction of the frequency count for each of the 12 malware class present in the dataset. Remarkably, all the malware classes exhibit an equal frequency count of 1,198, indicating a balanced representation. The x-axis of the histogram corresponds to the distinct malware class labels, while the y-axis represents the frequency count. The histogram plot conveys a symmetrical distribution of frequencies among the diverse malware classes, highlighting a proportional representation of each class within the dataset. For a comprehensive understanding of the histogram plot, please refer to the Figure 2.9 This visualization offers valuable insights into the distribution patterns and relative frequencies of the different malware class variants in the dataset, contributing to a comprehensive analysis of the dataset.

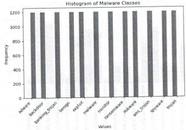


Figure 2.9: Histogram of Malware Class Frequencies Of Feature Tag:

Step 5: Visualization of Feature Usage within the Subset Of Feature Tag

A graphical representation was created to examine the usage of features within the subset of the dataset being analyzed. This graph focuses on the sum of integer columns while excluding the 'apk name', 'activity count', and 'malware class' columns to avoid redundancy. Moreover, a minimum threshold of 50 was applied to include only significant columns, ensuring a clear and concise analysis.

The x-axis of the graph denotes the different features present in the subset, while the y-axis represents the count of APK samples utilizing each specific feature. This visualization provides valuable insights into the prevalence and adoption of different features within the subset, enabling an understanding of which features are commonly employed by the APK samples.

For a comprehensive understanding of the feature usage patterns within the analyzed subset, please refer to the Figure 2.10 This visualization offers a concise overview of the feature utilization landscape within the analyzed dataset subset, facilitating the identification of prominent features employed by the APK samples.

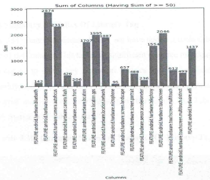


Figure 2.10: Visualization of Feature Usage within the Subset Of Feature Tag

2.4.1.3. Feature Set 3: Library_Tags_Dataset in AndroidManifest.xml

Step 1: Display Dataset Summary Library Tag

To provide an overview of the dataset, a summary table can be created. This table will include relevant columns and statistics that capture key information about the dataset as shown in

Table 2.9. Based on the example provided, the following table summarizes the Feature Tags dataset.

Table 2.9: Display Dataset Summary Library Tag

	Library:androidx.window.extensions	Library:androidx.window.sidecar	-	Library:org.apache.http.legacy	Library:org.simalliance.openmobileapi	Activity Count	Malware Class
0	1	0	-	1	0	80	adware
1	0	0	-	0	0	155	adware
2	0	1	-	0	0	14	adware
3	1	0	-	0	0	7	adware

Step 2: Calculate Summary Statistics Of Library Tag

To help us understand the dataset better, we have added a table that summarizes the summary statistics of the features. This table is a significant resource for acquiring insights into the dataset, allowing us to study key statistical metrics and better understand the data's distribution, variability, and properties. we will be able to gain a better knowledge of the dataset and perform informed analysis and Interpretation of the results. As Table 2.10 show the Calculate summary Statistics of Library Tag.

Table 2.10: Calculate Summary Statistics Of Library Tag

	Library:androidtest.runner	Library:androidx.window.extensions	-	Library:org.apache.http.legacy	Library:org.simalliance.openmobileapi	Activity Count
count	14376.00000	14376.000000	-	14376.000000	14376.000000	14376.0000
mean	0.007373	0.001391	-	0.023442	0.001252	34.654702
std	0.085554	0.037274	-	0.151307	0.035364	43.321236
min	0.000000	0.000000	-	0.000000	0.000000	0.000000
25%	0.000000	0.000000	-	0.000000	0.000000	7.000000
50%	0.000000	0.000000	-	0.000000	0.000000	20.000000
75%	0.000000	0.000000	-	0.000000	0.000000	46.000000
max	1.000000	1.000000	-	1.000000	1.000000	485.000000

Step 3: Retrieve Data Frame Column Information Of Library Tag

In Table 2.10 show retrieve data frame column information of library tag.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14376 entries, 0 to 14375
Data columns (total 10 columns):
 #   Column                                  Non-Null Count  Dtype
---  ------                                  --------------  -----
 0   Library:android.test.runner             14376 non-null  int64
 1   Library:androidx.window.extensions      14376 non-null  int64
 2   Library:androidx.window.sidecar         14376 non-null  int64
 3   Library:com.google.android.gcm.maps     14376 non-null  int64
 4   Library:com.google.android.maps         14376 non-null  int64
 5   Library:com.sec.android.app.multiwindow 14376 non-null  int64
 6   Library:org.apache.http.legacy          14376 non-null  int64
 7   Library:org.simalliance.openmobileapi   14376 non-null  int64
 8   Activity Count                          14376 non-null  int64
 9   Malware Class                           14376 non-null  object
dtypes: int64(9), object(1)
memory usage: 1.1+ MB
```

Figure 2.11: Retrieve Data Frame Column Information Of Library Tag

Step 4: Histogram of Malware Class Frequencies Of Library Tag:

In Figure 2.12 depict the histogram of malware class frequency of library tag,.

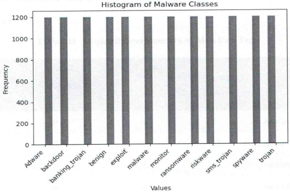


Figure 2.12:Histogram of Malware Class Frequencies Of Library Tag

Step 5: Visualization of Feature Usage within the Subset of Library Tag:

In Figure *2.13* show visualization of feature usage with in the subset of library tag.

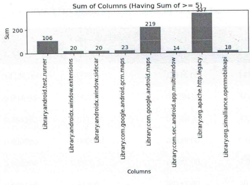


Figure 2.13: Visualization of Feature Usage within the Subset Of Library Tag

2.4.1.4. Feature Set 4: Meta Data Tags in AndroidManifest.xml:

Step 1: Display Dataset Summary:

The some of the dataset summary of meta data tag shown in the

Table *2.11*

Table 2.11 : Display Dataset Summary of Meta Data Tag

	meta-data:AA_DB_NAME	meta-data:APP_KEY	...	meta-data_value:true	Activity Count	Malware Class
0	0	0	..	0	80	Adware
1	0	0	..	1	155	Adware
2	0	0	..	0	14	Adware
3	0	0	..	0	7	Adware
4	0	0	..	1	25	Adware

Step 2: Calculate Summary Statistics Of Meta Data Tag:

In Table 2.12 show the calculate summary statistics of metadata tag.

Table 2.12: Calculate Summary Statistics Of Meta Data Tag

	data:AA_DB_NAME	meta-data:Adapter	...	meta-data_value:true	Activity Count
count	14376.000	14376.00	...	14376.00	14376.00
mean	0.004035	0.002295	...	0.079438	34.803214
std	0.063392	0.047858	...	0.270430	43.239226
min	0.000000	0.000000	...	0.000000	0.000000
25%	0.000000	0.000000	...	0.000000	7.000000
50%	0.000000	0.000000	...	0.000000	20.000000

75%	0.000000	0.000000	...	0.000000	46.000000
max	1.000000	1.000000	...	1.000000	485.000000

Step 3: Retrieve DataFrame Column Information Of Meta Data Tag:

The information of meta data tag which is retrieve data frame columns depict in Figure 2.14

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14376 entries, 0 to 14375
Columns: 457 entries, meta-data:AA_DB_NAME to Malware Class
dtypes: int64(456), object(1)
memory usage: 50.1+ MB
```

Figure 2.14: Retrieve DataFrame Column Information Of Meta Data Tag

Step 4: Histogram of Malware Class Frequencies Of Meta Data Tag:

As Figure 2.15 show the Histogram of Malware Class Frequencies Of Meta Data Tag.

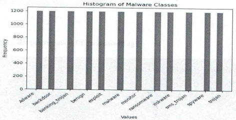


Figure 2.15: Histogram of Malware Class Frequencies Of Meta Data Tag

Step 5: Visualization of Feature Usage within the Subset Of Meta Data Tag:

Visualization of feature usage within the subset of meta data tag are shown in Figure 2.16.

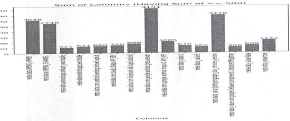


Figure 2.16: Visualization of Feature Usage within the Subset Of Meta Data Tag

2.4.1.5. Feature Set 5: Permission Tags in AndroidManifest.xml

Step 1: Display Dataset Summary Of Permission Tag:

In ***Table 2.3*** Display Dataset Summary Of Permission Tag.

Table 2.13 : Display Dataset Summary Of Permission Tag

	PERMISSION:android.hardware.camera.autofocus	PERMISSION:android.hardware.sensor.accelerometer	PERMISSION:android.permission.ACCESS_FINE_LOCATION	...	PERMISSION:telecom.mdesk.permission.WRITE_SETTINGS	PERMISSION_required:false	Activity Count	Malware Class
0	0	0	1	...	0	0	80	Adware
1	0	0	1	...	0	0	155	Adware
2	0	0	1	...	0	0	14	Adware
3	0	0	0	...	0	0	7	Adware
4	0	0	0	...	0	0	25	Adware

Step 2: Calculate Summary Statistics Of Permission Tag

Some of the Summary Statistics Of Permission Tag are shown in the Table 2.14.

Table 2.14: Calculate Summary Statistics Of Permission Tag:

	PERMISSION:android.hardware.camera.autofocus	PERMISSION:android.hardware.sensor	.	PERMISSION:org.adw.donut.permission.READ_SETTINGS	PERMISSION:telecom.mdesk.permission.WRITE_SETTINGS	PERMISSION_required:false	Activity Count
count	14376.00	14376.00	.	14376.00	14376.00	14376.00	14376.00
mean	0.001948	0.001530	.	0.002504	0.002156	0.003061	34.731
std	0.044091	0.039091	.	0.049981	0.046388	0.055240	43.135
Min	0.000000	0.000000	.	0.000000	0.000000	0.000000	0.00
25%	0.000000	0.000000	.	0.000000	0.000000	0.000000	7.000
50%	0.000000	0.000000	.	0.000000	0.000000	0.000000	20.000
75%	0.000000	0.000000	.	0.000000	0.000000	0.000000	47.00
max	1.000000	1.000000	.	1.000000	1.000000	1.000000	485.00

Step 3: Retrieve Data Frame Column Information Of Permission Tag :

In Figure 2.17 show the detail of the retrieve data frame column information of permission tag.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14376 entries, 0 to 14375
Columns: 225 entries, PERMISSION:android.hardware.camera.autofocus to Malware Cla
dtypes: int64(224), object(1)
memory usage: 24.7+ MB
```

Figure 2.17: Retrieve DataFrame Column Information Of Permission Tag

Step 4: Histogram of Malware Class Frequencies Of Permission Tag:

In Figure 2.18 show the histogram of malware class frequency of the permission tag

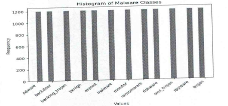


Figure 2.18: Histogram of Malware Class Frequencies Of Permission Tag

Step 5: Visualization of Feature Usage within the Subset Of Permission Tag

The visualization of the feature usage with in the subset of permission tag shown in Figure 2.19.

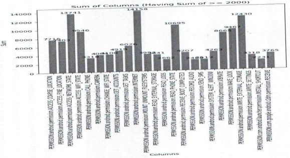


Figure 2.19: Visualization of Feature Usage within the Subset Of Permission

2.4.1.6. Feature Set 6: Provider Tags in AndroidManifest.xml

Step 1: Display Dataset Summary Of Provider Tag

To provide an overview of the dataset, a summary table can be created. This table will include relevant columns and statistics that capture key information about the dataset. Based on the example provided, the following table summarizes the application tags dataset. In Table 2.15 show some of the dataset summary of Application tag.

Table 2.15 : Display Dataset Summary Of Provider Tag

	provider:android.support.v4.content.FileProvider	provider:androidx.core.content.FileProvider	...	provider:mono.MonoRuntimeProvider	provider:nl.xservices.plugins.FileProvider	provider_readPermission:com.whatsapp.sticker.READ	Activity Count	Malware Class
0	0	0	...	0	0	0	80	Adware
1	0	1	...	0	1	0	155	Adware
2	1	0	...	0	0	1	14	Adware
3	0	0	...	0	0	0	7	Adware
4	0	1	...	0	0	0	25	Adware

Step 2: Calculate Summary Statistics Of Provider Tag

The summary of the statistics of provider tag show the count, mean, std , min, 25%, 50%, 75%, max of the provider tag which is shown in Table 2.16.

Table 2.16 : Calculate Summary Statistics Of Provider Tag

	provider:android.support.v4.content.FileProvider	provider:androidx.core.content.FileProvider	...	provider_grantUriPermissions:true	provider_readPermission:com.whatsapp.sticker.READ	Activity Count
count	14376.000000	14376.00000	.	14376.000000	14376.000000	14376.000000
mean	0.008069	0.013286	.	0.036380	0.000696	34.654772
std	0.089468	0.114501	.	0.187241	0.026366	43.067024
min	0.000000	0.000000	.	0.000000	0.000000	0.000000
25%	0.000000	0.000000	.	0.000000	0.000000	7.000000
50%	0.000000	0.000000	.	0.000000	0.000000	20.000000
75%	0.000000	0.000000	.	0.000000	0.000000	46.000000
max	1.000000	1.000000	.	1.000000	1.000000	485.000000

Step 3: Retrieve DataFrame Column Information Of Provider Tag:

In Figure 2.20show the details of data frame column of provider tag

```
 #   Column                                                                  Non-Null Count  Dtype
---  ------                                                                  --------------  -----
 0   provider:android.support.v4.content.FileProvider                        14376 non-null  int64
 1   provider:androidx.core.content.FileProvider                             14376 non-null  int64
 2   provider:androidx.lifecycle.ProcessLifecycleOwnerInitializer            14376 non-null  int64
 3   provider:androidx.startup.InitializationProvider                        14376 non-null  int64
 4   provider:androidx.work.impl.WorkManagerInitializer                      14376 non-null  int64
 5   provider:cl.json.RNShareFileProvider                                    14376 non-null  int64
 6   provider:cn.dictcn.android.digitize.contentprovider.SearchWordContentProvider  14376 non-null  int64
 7   provider:co.classplus.app.utils.ClassplusFileProvider                   14376 non-null  int64
 8   provider:com.RNFetchBlob.Utils.FileProvider                             14376 non-null  int64
 9   provider:com.android.vending.expansion.zipfile.APEZProvider             14376 non-null  int64
 10  provider:com.applovin.sdk.AppLovinInitProvider                          14376 non-null  int64
 11  provider:com.baidu.protect.StubProvider                                 14376 non-null  int64
 12  provider:com.crashlytics.android.CrashlyticsInitProvider                14376 non-null  int64
 13  provider:com.dotbiz.taobao.demo.m1.provider.LocalFileContentProvider    14376 non-null  int64
 14  provider:com.facebook.FacebookContentProvider                           14376 non-null  int64
 15  provider:com.facebook.NativeAppCallContentProvider                      14376 non-null  int64
 16  provider:com.facebook.ads.AudienceNetworkContentProvider                14376 non-null  int64
 17  provider:com.facebook.internal.FacebookInitProvider                     14376 non-null  int64
 18  provider:com.facebook.marketing.internal.MarketingInitProvider          14376 non-null  int64
 19  provider:com.freshchat.consumer.sdk.provider.FreshchatInitProvider      14376 non-null  int64
 20  provider:com.google.android.gms.ads.MobileAdsInitProvider               14376 non-null  int64
 21  provider:com.google.android.gms.measurement.AppMeasurementContentProvider  14376 non-null  int64
 22  provider:com.google.firebase.perf.provider.FirebasePerfProvider         14376 non-null  int64
 23  provider:com.google.firebase.provider.FirebaseInitProvider              14376 non-null  int64
 24  provider:com.igexin.download.DownloadProvider                           14376 non-null  int64
 25  provider:com.imagepicker.FileProvider                                   14376 non-null  int64
 26  provider:com.ironsource.lifecycle.IronsourceLifecycleProvider           14376 non-null  int64
 27  provider:com.kbeanie.multipicker.utils.AMPFileProvider                  14376 non-null  int64
 28  provider:com.qbiki.util.InternalFileContentProvider                     14376 non-null  int64
 29  provider:com.reactnative.ivpusic.imagepicker.IvpusicImagePickerFileProvider  14376 non-null  int64
 30  provider:com.reactnativecommunity.webview.RNCWebViewFileProvider        14376 non-null  int64
 31  provider:com.squareup.picasso.PicassoProvider                           14376 non-null  int64
 32  provider:com.urbanairship.UrbanAirshipProvider                          14376 non-null  int64
 33  provider:com.uzmap.pkg.uzapp.UProvider                                  14376 non-null  int64
 34  provider:com.vinzscam.reactnativefileviewer.FileProvider                14376 non-null  int64
 35  provider:de.appplant.cordova.emailcomposer.Provider                     14376 non-null  int64
 36  provider:de.appplant.cordova.plugin.notification.util.AssetProvider     14376 non-null  int64
 37  provider:droidninja.filepicker.utils.FilePickerProvider                 14376 non-null  int64
 38  provider:expo.modules.filesystem.FileSystemFileProvider                 14376 non-null  int64
 39  provider:expo.modules.imagepicker.ImagePickerFileProvider               14376 non-null  int64
 40  provider:io.flutter.plugins.share.ShareFileProvider                     14376 non-null  int64
 41  provider:io.github.pwlin.cordova.plugins.fileopener2.FileProvider       14376 non-null  int64
 42  provider:io.intercom.android.sdk.IntercomInitializeContentProvider      14376 non-null  int64
 43  provider:io.sentry.android.core.SentryInitProvider                      14376 non-null  int64
 44  provider:io.sentry.android.core.SentryPerformanceProvider               14376 non-null  int64
 45  provider:mono.MonoRuntimeProvider                                       14376 non-null  int64
 46  provider:nl.xservices.plugins.FileProvider                              14376 non-null  int64
 47  provider:org.apache.cordova.camera.FileProvider                         14376 non-null  int64
 48  provider:sdk.download.DownloadProvider                                  14376 non-null  int64
 49  provider_authorities:@string/classplus_provider_authority               14376 non-null  int64
 50  provider_authorities:@string/freshchat_file_provider_authority          14376 non-null  int64
 51  provider_grantUriPermissions:true                                       14376 non-null  int64
 52  provider_readPermission:com.whatsapp.sticker.READ                       14376 non-null  int64
 53  Activity Count                                                          14376 non-null  int64
 54  Malware Class                                                           14376 non-null  objec
```

Figure 2.20:Retrieve Data Frame Column Information Of Provider Tag

Step 4: Histogram of Malware Class Frequencies Of Provider Tag:

The histogram for each classes which we have used in our research in depict in the Figure 2.21.

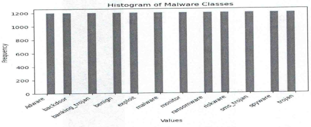
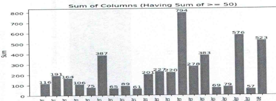


Figure 2.21:Histogram of Malware Class Frequencies Of Provider Tag:

Step 5: Visualization of Feature Usage within the Subset Of Provider Tag

In Figure 2.22 represent the features which are used with in the subset of the provider tag.

Figure 2.22:Visualization of Feature Usage within the Subset Of Provider Tag



2.4.1.7. Feature Set 7: Receiver Tags in AndroidManifest.xml

Step 1: Display Dataset Summary Of Receiver Tag:

In Table 2.17 display some of receiver tag dataset summary .

Table 2.17: Display Dataset Summary Of Receiver Tag

	receiver:androidx.work.impl.background.systemalarm.Constraint.StorageNotLowProxy	receiver:androidx.work.impl.background.systemalarm.ConstraintUpdateReceiver	..	receiver:pushsharp.clientsample.monoforandroid.SampleBroadcastReceiver	receiver_enabled:false	receiver_enabled:true	Activity Count	Malware Class
0	0	0	..	0	0	0	80	Adware
1	0	0	..	0	0	0	155	Adware
2	0	0	..	0	0	0	14	Adware
3	0	0	..	0	0	0	7	Adware
4	0	0	..	0	0	0	25	Adware

Step 2: Calculate Summary Statistics Of Receiver Tag

In Table 2.18 show the summary of statistics of receive tag

Table 2.18: Calculate Summary Statistics Of Receiver Tag

	receiver:androidx.work.impl.systemalarm.ConstraintProxy.BatteryNotLowProxy	receiver:androidx.work.impl.background.systemalarm.RescheduleReceiver	...	receiver_enabled:false	receiver_enabled:true	Activity Count
count	14376.000000	14376.000000	...	14376.000000	14376.000000	14376.00000
mean	0.011199	0.011269	...	0.018781	0.171119	34.34342
std	0.105236	0.105558	...	0.135757	0.376626	42.56421
min	0.000000	0.000000	...	0.000000	0.000000	0.00000
25%	0.000000	0.000000	...	0.000000	0.000000	7.00000
50%	0.000000	0.000000	...	0.000000	0.000000	19.00000
75%	0.000000	0.000000	...	0.000000	0.000000	46.00000
max	1.000000	1.000000	...	1.000000	1.000000	485.00000

Step 3: Retrieve Data Frame Column Information Of Receiver Tag:

The receiver tag show the following details which is mentioned below.

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 14376 entries, 0 to 14375
Columns: 290 entries, receiver:AlarmReceiver to Malware Class
dtypes: int64(289), object(1)
memory usage: 31.8+ MB

Step 4: Histogram of Malware Class Frequencies Of Receiver Tag:

The histogram show each malware class frequency for receiver tag in Figure 2.23.

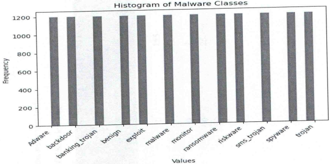


Figure 2.23:Histogram of Malware Class Frequencies Of Receiver Tag

Step 5: Visualization of Feature Usage within the Subset Of Receiver Tag:

In Figure 2.24 visualize the feature use with in the subset of receiver tag

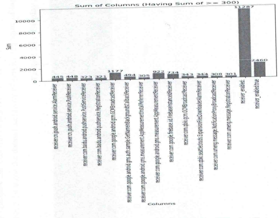


Figure 2.24: Visualization of Feature Usage within the Subset Of Receiver Tag:

2.4.1.8. Feature Set 8: Service Tags in AndroidManifest.xml

Step 1: Display Dataset Summary Of Service Tag

In Table 2.19 show the dataset summary of service tag some of the features has been shown in this table .

Table 2.19 : Display Dataset Summary Of Service Tag

	SERVICE:android.google.account	SERVICE:cn.com.pSerhchost.PhostServ	SERVICE:cn.com.pdua.cdu.Dliteinface	...	SERVICE_Enabled:false	SERVICE_Enabled:true	Activity Count	Malware Class
0	0	0	0	...	0	1	80	Adware
1	0	0	0	...	0	1	155	Adware
2	0	0	0	...	0	1	14	Adware
3	0	0	0	...	0	0	7	Adware
4	0	0	0	...	0	0	25	Adware

Step 2: Calculate Summary Statistics Of Service Tag

In Table 2.20 show the summary statistics such as count, mean, std, min, 25%, 50%, 75% for the service tag

Table 2.20 : Calculate Summary Statistics Of Service Tag

	SERVICE:android.gogle.accout	SERVICE:androidx.room.MultiInstanceInvalidationService	...	SERVICE_Enabled:@bool/enable_system_foreground_service_default	SERVICE_Enabled:false	SERVICE_Enabled:true	Activity Count
count	14376.0	14376.000	...	14376.00	14376.00	14376.0	14376.00
mean	0.01669	0.013147	...	0.007165	0.008069	0.309822	34.8723
std	0.40826	0.113908	...	0.084344	0.089468	0.462436	43.4702
min	0.0000	0.000000	...	0.000000	0.000000	0.000000	0.000000
25%	0.0000	0.000000	...	0.000000	0.000000	0.000000	7.000000
50%	0.0000	0.000000	...	0.000000	0.000000	0.000000	20.00000
75%	0.0000	0.00000	...	0.00000	0.00000	1.00000	46.0000
max	1.0000	1.0000	...	1.000000	1.000000	1.000000	485.000

Step 3: Retrieve DataFrame Column Information Of Service Tag

Retrieve data frame column information shown below of the service tag .

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 14376 entries, 0 to 14375
Columns: 308 entries, SERVICE:android.google.account to Malware Class
dtypes: int64(307), object(1)
memory usage: 33.8+ MB

Step 4: Histogram of Malware Class Frequencies Of Service Tag

Histogram of each malware class which is considered for the service tag is depict in Figure 2.25.**Figure *2.25*:Histogram of Malware Class Frequencies Of Service Tag**

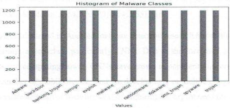


Figure 2.25:Histogram of Malware Class Frequencies Of Service Tag

Step 5: Visualization of Feature Usage within the Subset Of Service Tag

In Figure 2.26 shows the visualization of feature usage within the subset of service tag.

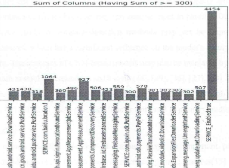


Figure 2.26: Visualization of Feature Usage within the Subset Of Service Tag

2.4.3. Modelling Techniques

SVM was extensively utilised in Android malware detection research [46], [67], [68], [17], [16], [59], [41], [9], [42], [65], [4], [44], [51], [20], [27] from (2013-2015). During this period, Random Forest was also employed in a few researches [70], [56], [64] and [64] Also often employed were K-Means and KNN [20], [64], [55], [16].

In recent years, the trend in Android malware detection research has shifted towards the use of deep learning approaches, particularly CNNs. Due to the increased accessibility of massive datasets and computing power, CNNs have grown in popularity in recent years. The increase in the number of mobile devices and the development of mobile computing has also resulted in an increase in Android malware attacks, requiring the urgent need for more accurate and efficient detection methods.

According to studies, CNNs can identify Android malware with a high degree of accuracy [26] , [9], [65], [40], [20], [27], [55]. This is because of their capacity to determine complex properties from unstructured data, which is particularly helpful in identifying newly developed and unknown malware classes. Additionally, CNNs are more computationally efficient than typical machine learning techniques because they can analyse input in parallel. Despite their efficiency, CNN-based Android malware detection methods may yet be improved. The selection of input features, which has a significant influence on the model's accuracy, is one area for development. The accuracy of CNN-based models may also be increased by the use of ensemble models, such as stacking ensembles and bagging [4], [49], [45], [27], [64]. The ability to adapt of CNN-based models to malware attacks can also be increased through the use of concurrent training. In Table 2.21 show the Algorithms mentioned in primary studies.

2.4.4. Dataset Quality

Research has shown that the selection of datasets plays an important part in the performance of Android malware detection models when using static features[15]. Although the Drebin dataset has been extensively utilised in past studies, it has certain drawbacks, which include being very small and mainly including outdated malware samples. Therefore, researchers have begun utilizing larger and more diverse datasets such as Androzoo and VirusShare, which offer a broader range of malware samples.

For instance, a recent study trained a deep learning-based model on the Androzoo dataset and obtained an accuracy of 99.53% in identifying Android malware using static features. This dataset offers a rich supply of data for training machine learning models and comprises over

23 million Android apps including both benign and malware samples. Similar to this, another study trained a Random Forest model using the VirusShare dataset to detect Android malware with a 98.9% accuracy using static features.

In addition to the dataset choice, the model architecture selection can also significantly affect how well malware is detected. Recent studies have demonstrated that deep learning-based models such as Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) have shown promising results in identifying Android malware using static features. However, the fact that these models need a lot of data for training highlights how essential having high-quality datasets are. The performance of detecting Android malware using static features can be enhanced by combining consistent, large datasets with the most recent models. One research, for instance, trained a deep learning-based model using the datasets from Androzoo and VirusShare to obtain an accuracy of 99.27%. Another study trained a Random Forest model using numerous datasets, including AMD, VirusShare, and Androzoo and attained an accuracy of 97.8%. These findings highlight the value of having high-quality datasets and the possibility of utilising them in alongside advanced algorithms for accurate malware detection.

Table 2.21: Algorithms mentioned in primary studies

ALGORITHM	STUDIES
SVM	[46], [67], [68], [17], [16], [59], [41], [9], [42], [65], [4], [44], [51], [20], [27]
RANDOM FOREST	[70], [56], [64], [64]
K-MEANS	[24], [68], [9], [9], [42], [23], [65], [4], [49], [29] , [20], [27], [70]
KNN	[64], [55], [16]
NAÏVE BAYES	[68]
SEQUENTIAL MINIMAL OPTIMIZATIO	[26], [9], [65], [29], [40], [27], [70], [56]
ENSEMBEL AND MLP,	[26] , [9] , [23], [65], [4], [27], [70], [61], [64]
C4.5	[26]

ALGORITHM	STUDIES
LOGISTIC REGRESSION	[26] , [16], [65]
DNN	[26]
CNN	[26] , [9], [65], [40], [20], [27], [55]
ANN	[38], [37] , [36], [30], [58], [9], [20] , [75], [32], [18], [54]
RNN	[37],[9], [48], [30], [21],, [50], [35], [71], [63], [18], [54], [8] , [53
LSTM	[7], [33], [52], [57], [13]
BILSTM	[14], [72], [35]
GRU	[14], [72], [8]
ADABOOST	[72], [47], [8]
MARKOV MODEL	[72], [8]
NEURAL NETWORK	[59], [9], [65], [66], [27]
DECISION TREE	[41]
FNN/ HAMMING DISTNACE	[41], [42]
WANN/WEIGHTED ALL NEAREST NEIGHBORS	[9], [65], [4], [61], [56], [64]
K-MEDOID BASED NEAREST NEIGHBORS (KMNN)	[33], [52]
BAGGING	[33], [52]
GRADIENT BOOSTING	[33], [52]
VOTING CLASSIFIER	[65], [27]
J48	[65], [66], [45]
JRIP	[65]
STACKING ENSEMBLE	[4], [49], [45], [27], [64]

ALGORITHM	STUDIES
MULTI-MODEL VISUAL REPRESENTATION	[49]
MULTILAYER PERCEPTRON (MLP)	[50]
DFS	[74]
BN	[60]
HYPERPARAMETERS	[69]
BAYESNET	[40]
GAN	[20]
QUANTUM SUPPORT VECTOR MACHINE (QSVM)	[21]
DEEP BELIEF NETWORK (DBN	[60]
MULTIMODAL DEEP LEARNING	[29]
CLASSIFICATION AND REGRESSION TREES (CART),	[20]
ET	[27]

2.4.1.9. Performance Metrics:

Performance metrics are critical for assessing the efficacy of machine learning and deep learning models for Android malware detection. Accuracy, precision, recall, F1-score, and area under the curve (AUC) are all common performance indicators. Several published research in Android malware detection have reported these performance metrics from 2013 to 2023 to evaluate the performance of their suggested models. In Table 2.22 show the performance measures use in the primary studies. Among these measures, accuracy is generally employed as a major performance measure to assess the model's overall performance, whilst precision and recall are used to assess the model's efficacy in detecting malware samples and benign samples, respectively. The analysis of the collected data showed that the most commonly used performance measures for investigating the impact of data quality issues on static analysis of malware detection in Android are accuracy, recall, and precision. Based on the information provided in the table, the top three performance metrics utilized to examine the effects of data quality concerns on static analysis of malware detection in Android were Accuracy, Recall (Sensitivity, True Positive Rate TPR), and Precision/Correctness. Accuracy measures the overall correctness of the classification model and is defined as the ratio of correctly classified instances to the total number of instances. These performance measures can help address data quality issues by providing insights into the effectiveness of the static analysis approach in detecting malware. For e.g accuracy can give an overall assessment of the data quality used to train and test the model. Low accuracy might be a sign of poor data quality, such as missing or inaccurate data. Recall can help identify false negatives, which are instances of malware that were not detected by the model. Recall can be used to identify false negatives, or instances of malware that the model missed. This could help researchers in identifying the varieties of malware that are more challenging to find and enhancing the model's capacity to perform. Precision can be used to spot false positives, or non-malware cases that were mistakenly labelled as such. This can assist researchers in identifying the characteristics that the model is using to detect malware and improving the model to lower false positives. Overall, using appropriate performance measures in evaluating the impact of data quality issues on static analysis of malware detection in Android can help researchers and practitioners improve the accuracy, effectiveness, and efficiency of malware detection systems.

Table 2.22: Performance Metrics mentioned in primary studies

EVALUATION METRIC	STUDIES
ACCURACY	[46], [67],[24], [68],[37], [36], [16], [30], [14], [72], [47], [59], [9], [42], [25], [23], [48], [33], [4], [49], [21], [50], [73], [74], [40], [35], [28], [70], [61], [55], [18], [54], [8], [53]
RECALL (SENSITIVITY, TRUE POSITIVE RATE TPR	[46], [67],[24], [68],[17], [26], [38], [37], [16], [30], [14], [58], [72], [41], [9], [25], [23], [48], [33], [65], [4], [49], [21], [74], [44], [51], [45], [69], [40], [35], [27], [27], [27], [28], [70], [61], [75], [32], [32], [57], [13], [71], [62], [63], [64], [55], [18], [54], [8], [53]

EVALUATION METRIC	STUDIES
SPECIFICITY (TNR)	[46], [16], [14]
PRECISION/CORRECTNESS	[46], [67], [24] , [68], [26], [58], [37] , [16], [14], [42], [72], [41], [9], [25], [23], [48],[[65], [4], [49], [21], [74], [44], [51], [45], [69], [40], [20], [27], [28], [70], [61], [75], [32], [32], [57], [13], [71], [63], [55], [18], [54], [8], [53]
FALSE POSITIVE RATE (FPR)	[46], [24], [17], [38], [47]z [59], [25], [27], [28], [71], [55], [13], [57]
FALSE NEGATIVE RATE (FNR)	[25], [66], [53]
AUC (AREA UNDER CURVE)	[38], [30], [25], [74], [52], [57], [71]
ROC	[38], [25], [74], [52], [8]
F MEASURE	[46], [67], [24], [26], [47], [27], [70], [64]
COHEN'S KAPPA	[46]
F1-SCORE	[38],[36],, [16], [14], [58], [72], [9], [66], [60], [29], [40], [20], [35], [27], [71], [62], [63], [27], [70], [61], [75], [64], [55], [18], [54], [8], [53]
MATTHEWS CORRELATION COEFFICIENT	[16]
CONFUSION METRIC	[9], [42], [60], [28], [54]
DETECTION RATE	[4], [66], [29], [28]
LOG AND LOSS	[54]
MISCLASSIFICATION RATE Đ	[53]

2.4.1.10. Transfer Learning:

After conducting an extensive search on the topic of Android malware detection using static features, it was found that only a limited number of studies have explored the role of transfer learning in improving the accuracy of detecting Android malware. Out of the three relevant papers [5], [63], [74] identified, all of them used transfer learning techniques to enhance the performance of the malware detection models.Therefore, while the limited number of studies on the topic suggests that transfer learning has not been extensively explored in the context of Android malware detection using static features, the studies that have been conducted demonstrate the potential of transfer learning techniques to improve the accuracy of detection models. Further research in this area is warranted to fully explore the effectiveness of transfer learning in addressing the challenges of detecting Android malware.

2.4.5. Discussion of Limitations and Conclusions:

We performed a systematic literature review to analyse the performance of machine learning, and deep learning techniques for Android Malware Detections using Static Features. We select the 62 studies out of 120 studies from different sources by applying inclusion and exclusion criteria. We found out which of the techniques are preferred by researchers in each category i.e., machine learning, and deep learning. We also compared performance reported in each of the selected primary studies and we reported which of the performance measure is used in each of these studies. We report that deep learning and machine learning are widely used for Android malware detection. However, researchers used ensemble techniques and transfer learning methods less frequently for android malware detection for static analysis of applications. There is need to work on using same techniques on combinations of different datasets having large number of classes and there is a requirement of larger datasets in public domain. The most often used dataset is the Drebin and Virus share dataset, and studies indicate that it is a trustworthy and valuable resource for detecting android malware. The total 47 from our selected primary studies are on the on the binary classes (Most of the published paper work) and remaining selected primary studies are on multi class classification. The studies reviewed show that this method is successful in detecting numerous forms of malware with high accuracy rates. We analysed from or study that the Adware, Ransomware, Trojan and Backdoor are mostly considered by the researchers. The data also reveal that SVM is the most successful AI model for this purpose, while API calls, Permissions, and Strings are the most relevant elements for identifying android malware. One major limitation of this SLR is that it primarily concentrated on static feature analysis research for Android malware detection. Other methodologies, such as dynamic analysis or hybrid approaches, might be investigated in future studies. Another drawback is that this evaluation only included papers published in English, which may have eliminated some important investigations.

3. RESEARCH METHODOLOGY

3.1. Research Design:

We use Applied Research design principles and provide research design that includes data collection, measurement and data analysis steps. We further include data preprocessing, feature extraction, model choice, model assessment, and performance measures for a detailed discussion of our research process. We go through the data collection and preprocessing steps, feature selection, and deep neural network (DNN) model development processes. We assess our model's performance by, such as accuracy, confusion matrix, F1 score, recall, and ROC-AUC. Finally, we provide details on how we divided the dataset into training and testing sets, and how we used the testing set to validate our model. In Figure *3.1* we depict our research design which we follow in our research.

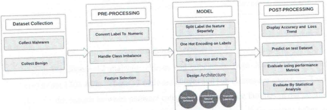


Figure 3.1: Our Research Methodology

3.2. Data Collection Methods:

1. **Defining the starting point:** We begin by clearly defining the research objectives and the types of malware classes we want to collect data on. We followed a series of steps to determine which classes we utilize to train our model as depict in **Figure 3.2**.

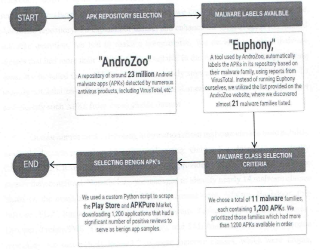


Figure 3.2: Selection of Classes

We choose 12 malware classes and subclasses from ANDROZOO which containing 1201 APks. These 12 malware classes and sub classes are shown in **Table 3.1**. 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. It is pertinent to note that extracting APK files from publicly available datasets allows the download of a sub- set of totally available APK files from these repositories. It is important to highlight that datasets such as CICmal2020, CICmal2019, and CICmal2017, provided details about the malware classes they contained. We identify 14 malware classes that include Backdoor, File Infector, PUA, Ransomware, Riskware, Scareware, Trojan, Trojan- Banker, Trojan-Dropper, Trojan-SMS, Trojan-Spy, Zero-day, and SMs Malware based on the avail From the Androzoo repository, we successfully located 7 of these malware classes, which were Trojan, Banking, Backdoor, SMS Trojan, Ransomware, Adware, and Riskware. However, for the remaining 7 malware classes mentioned earlier, we couldn't find suitable data in the Androzoo repository. To expand our dataset and enhance the effectiveness of Android malware detection, we had to make a compromise. We decided to select malware classes that had more than "1201" APKs available in the Androzoo repository. Among these, we included three additional classes, namely spyware, monitor, and exploit. To identify potential zero-day threats, we utilized the K-means algorithm to detect patterns and identify such APKs from the available dataset

During our research, obtaining information about malware classes from publicly available datasets proved to be quite challenging. Only a few datasets, specifically CICmal2020, CICmal2019, and CICmal2017, provided details about the malware classes they contained. Nonetheless, we managed to identify nearly 14 malware classes based on the available information. These classesincluded Adware, Backdoor, File Infector, PUA, Ransomware, Riskware, Scareware, Trojan, Trojan-Banker, Trojan-Dropper, Trojan-SMS, Trojan-Spy, Zero-day, and SMs Malware. From the androzoo repository, we successfully located 7 of these malware classes, which were Trojan, Banking, Backdoor, Sms Trojan, Ransomeware, Adware, and Riskware. However, for the remaining 7 malware classes mentioned earlier, we couldn't find suitable data in the androzoo repository. To expand our dataset and enhance the effectiveness of android malware detection, we had to make a compromise. We decided to select malware classesthat had more than 1200 APKs available in the androzoo repository. Among these, we included three additional classes, namely spyware, monitor, and exploit. To identify potential zero-day threats, we utilized the K-means algorithm to detect patterns and identify such APKs from the available dataset.

Table 3.1: Type Of Classes Select For Our Research

MALWARE CLASS	DESCRIPTION	TYPE/SUBTYPE
Malware	General term used for malicious software in Android OS	General Category
Trojan	Malicious Apps disguised as legit, e.g. GB Whatsapp	General Category
Banking Trojan	Kind of Trojan disguised specifically for the intent of stealing info from banking apps in phone	Sub Category – Trojan
Backdoor	A type of Malware which allows hidden access to a Trojan in the system	Sub Category – Trojan
SMS Trojan	A type of Trojan used to send premium SMS without user consent	Sub Category – Trojan
Spyware	A malicious software which secretly gathers user information	General Category
Monitor	A type of spyware with a specific intent of tracking user behavior which can allow attacker to apply social engineering attack on the user	Sub Category – Spyware
Exploit	A type of malware which targets vulnerabilities of the OS or apps	General Category
Ransomware	A type of attack which encrypts the personal data of user and demand financial ransom for decryption	General Category
Adware	A type of malicious software which displays unwanted ads in android OS	General Category
Riskware	Potentially risky apps but may/may not be a malware	General Category
Zero-day Attack	A zero-day attack is a cyber threat exploiting undiscovered vulnerabilities, striking before software developers can release a fix	Cyber Attack
Benign	Apps that are legit and proven that they do not posse a threat to the android OS	Not a Malware

2. **Searching for malware classes:** After determining the malware classifications, we search for them in web databases such as VirusTotal and Androzoo. We may find a large number of malware samples for our study from these databases.
3. **Converting data to CSV format:** After locating the malware samples, we construct a Python script to convert the data from the Virus Total JSON format to a more practical CSV format. As a result of this step, critical data such as the virus's name, file type, file size, number of detections, and download URL are simpler to extract from the data.

4. **Organizing the data:** Finally, by dividing the CSV file according to the various malware types selected, we organize the data. We may achieve this using a Python script or other tools to make it simple to obtain the data for analysis. We save the data locally for quick access and as a backup in case of data loss.

All these steps depict in Figure 3.3.

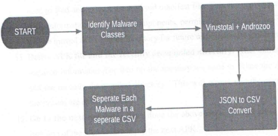


Figure 3.3: Collection Of Malware Classes

5. **Load CSV of each malware class:** After splitting the CSV file based on malware types, we need to load each CSV file into memory to extract the APK links. This process involves reading the CSV files and getting the data that is needed by processing.
6. **Inquire the user for the total number of APKs to Download:** After loading the CSV files, we require to show the message for the user how many APKs they want to download. Processing APKs might be time-consuming so it is important to restrict the quantity of APKs downloaded. This approach ensures that even in cases where the program unexpectedly closes, users do not have to restart the download of all 1200 APKs from the beginning, making the tool more user-friendly and efficient.
7. **Create a loop:** 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 the following steps:

8. **Download APK:** In each iteration, we will download an APK from the specified malware class using its corresponding link from the previously loaded CSV file.
9. **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.
10. **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.
11. **Delete APK file and the recently decompiled directory:** After extracting the required information,For free up the memory we need to delete the APK file and the recently decompiled directory . This will ensure that the resources of the system are used efficently.
12. **Go to the next iteration:** After done the above steps, we will go to the next iteration of the loop to download the next APK.

The whole process of download malware apks shown in Figure 3.4.

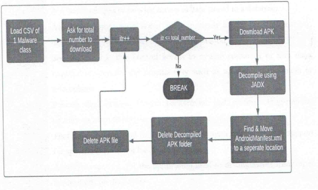


Figure 3.4: Process Of Malware APKS Download

13. **From extracts directory find AndroidManifest.xml files:** After decompiling the downloaded APKs, we shift the AndroidManifest.xml files to a separate directory. In this step for extract the content we will search for all the AndroidManifest.xml files in the directory .
14. **Construct a loop to iterate through all files found:** We create a loop to iterate through each file found after we have get all the AndroidManifest.xml files. From this loop will ensure that we extract the required information from all the files.

In each iteration, script read XML of each file and store the contents mentioned in the table below as features of AndroidMalware to a CSV For each file. The Table 3.2 shows the information that we extracted.

Table 3.2: Feature Used In Our Research

Feature	Description
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 execute 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 are in the applications.
Labels	Names 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 will gather above information from each file in a CSV file. We will be used this CSV file in the future for further analysis. Once all the files have been processed, the feature extraction process is complete. As Figure 3.5 show the process of malware feature extraction.

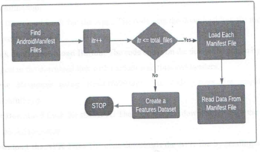


Figure 3.5: Malware Feature Extraction

Above mentioned processes are only for the collection of Malware samples so we had to construct a benign malware sample collector as well which is depict in Figure 3.6, so we designed a web scrapper to scrape through the Playstore and other third-party app stores to acquire benign samples, as shown in Figure 3.6 and as steps described below:

1. **Import required libraries:** The first step is to import the required libraries such as **requests**, **BeautifulSoup**, **urllib**, **time**, **random**, **csv**, and **os**.
2. **Read App IDs from CSV file:** The code reads the list of benign app IDs from a CSV file and stores them in a list.
3. **Loop through the App IDs:** The code loops through each app ID in the list of benign app IDs.
4. **Create Search URL:** The code creates a search URL using the app ID.

5. **Make Request to the Search URL:** The code sends a request to the search URL with random wait time and headers.
6. **Parse Response using BeautifulSoup:** The code parses the response using BeautifulSoup.
7. **Get Link for the App Page:** The code gets the link for the app page from the parsed response.
8. **Create URL for the App Page:** The code creates the URL for the app page.
9. **Make Request to the App Page:** The code sends a request to the app page with random wait time and headers.
10. **Parse Response using BeautifulSoup:** The code parses the response using BeautifulSoup.
11. **Get Download Link for the App:** The code gets the download link for the app from the parsed response.
12. **Click on the Download Button:** The code clicks on the download button by sending a request to the download link with random wait time and headers.
13. **Parse Response using BeautifulSoup:** The code parses the response using BeautifulSoup.
14. **Get Download Link for the APK:** The code gets the download link for the APK from the parsed response.
15. **Download the APK File:** The code downloads the APK file by sending a request to the APK download link with random wait time and headers.
16. **Save the APK File:** The code saves the downloaded APK file in the specified directory path.

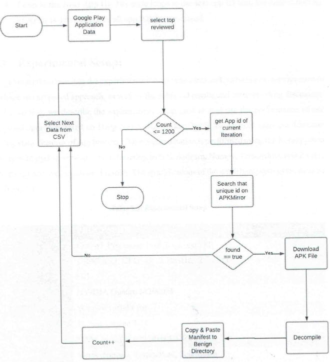


Figure 3.6: Benign Downloader

1. **Decompile the APK:** The code decompiles the APK file using JADX by constructing the command and running it.
2. **Copy Manifest File:** The code find the manifest file in the decompiled directory, renames it to the app ID, and saves it in a separate directory for the extracted files.
3. **Remove APK File and Decompiled Directory:** The code removes the downloaded APK file and the decompiled directory.

4. **Loop to the Next App ID**: The code loops to the next app ID until the desired number of APKs is downloaded or all app IDs are processed.

3.3. Experimental Setup:

This section describes the experimentation arrangements and performance metrics used to evaluate the proposed approach, as well as the achieved results and corresponding discussion. In this section, we describe the experimental setup used to evaluate the performance of our proposed approach based on Deep Neural Networks (DNN) for Android malware detection having static features of Application. The experimentation is performed using the Keras python library with python version 3.11.3, by using with Scikitlearn, Numpy, Tensorflow, and Pandas libraries to achieved the desired results. The specifications of the underlying system are defined in Table 3.3.

Table 3.3: Experimental Setup

	PROCESSOR	MODEL	GENRATION
CPU	Core-i7 Processor Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz, 2.71 GHz	G3	6th
GPU	NVIDIA Quadro M2000M.		
OS	Windows 10-64 Bit		
Language	Python version 3.11		
RAM	RAM 16 GB- 3600 MHz DDR4		
Software	Keras ,Numpy, Tensorflow, Scikitlearn		

To evaluate the performance of our proposed approach, we use several performance metrics, including accuracy, recall, F1 score, and ROC, AUC curve. A test set of Android malware samples is used to generate these metrics; this test set is separate from the training set. The experiment's findings are explained in the parts that follow along with a comparison to state-of-the-art methods.

3.4. Data Preprocessing:

Preprocessing is necessary in machine learning and deep learning applications because raw data often contains irrelevant or redundant information that can negatively impact model performance. Preprocessing can help with problems including missing data, class imbalance, and dataset noise. We can make sure that the model is only taking into account necessary features and that the data is in a format that the model can comprehend by cleaning, transforming, and normalizing the data. In the end, this may result in predicts that are more accurate and reliable.

3.4.1.1. Data Loading:

Loading data is the first step in any deep learning project. In this step, we used the pandas library in Python to read the CSV file containing the Android application dataset. The collection included details on a range of features, including API calls, intents, and permissions, utilized by various Android applications, along with information on the related malware class. It is simple to handle and examine the data by loading it into a pandas DataFrame.

3.4.1.2. Data Cleaning:

Data cleaning is an important step in any deep learning project. 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 identify for any missing or null values in the dataset and eliminate them as needed. For building a good model it is important in data cleaning step which help to to ensure that the data is accurate and reliable.

3.4.1.3. Handling Class Imbalance:

Class imbalance is a common problem in deep learning, where the number of samples in one class is much higher than the other class. This can lead to biased models that perform poorly on the minority class. To solve this problem, we used oversampling techniques like SMOTE to generate synthetic samples of the minority class or under sampling techniques like random under sampling in order to reduce the number of samples in the majority class. This helps to balance the classes in the dataset and ensure that the model is not biased towards the majority class. While initially considering 1200 APKs for each class, it became evident during the decompilation process that certain APKs couldn't be successfully decompiled due to various errors. Consequently, for certain classes, we found ourselves with a deficit of 10 to 20 APKs. In response to this challenge, we had to implement balancing techniques to ensure that each class had a sufficient number of representative samples for our analysis and model training.

3.5. Feature Selection:

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 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. Therefore, model work better because it reduce the number of dimensions in the dataset. So, we used the pandas concat() function to join the chosen features with the target variable (Malware Class) and the df.to_csv() function to save the preprocessed dataset as a CSV file. Feature selection helps improve the accuracy of the model by reducing the number of irrelevant or duplicate of attributes that can add noise to the data and make it harder for the model to learn the underlying patterns.

Initially, we had a total of 43,377 features, but following the preprocessing stage, we were left with only 10,523 features for further analysis and modeling.

We aim to make our deep neural network model work better and be more accurate by using the above pre-processing steps. The process of adding to and cleaning the dataset helps to make sure that the data is correct and consistent, which is important for building a reliable model. Handling class imbalance and selecting relevant features help to reduce bias in the model and improve its ability to make accurate predictions on unseen data. By taking these steps, we are able to extract the most useful information from the dataset and train a model that is robust and efficient in detecting malware in Android applications. This not only benefits the field of cybersecurity but also has practical applications in protecting users from potential harm and threats.

3.6. Evaluation Metrics:

To evaluate the performance of our deep learning model, we utilized a variety of metrics. These metrics included training accuracy, confusion matrix, F1 score, recall, and ROC-AUC. All evaluation metrics results are shown in

***Table** 3.4.*

3.6.1.1. Training Accuracy:

Training accuracy is a identify how well the model fits the training data. It is determined as the number of correctly classified samples divided by the total number of samples in the training set. Taking a look at the model's training quality may make it easier to figure out how well it can learn from the data and whether it is overfitting or under fitting.

3.6.1.2. TP/FP/TN/FN:

We used the confusion matrix to represent true positives, false positives, true negatives, and false negatives. A confusion matrix determine how well a classification model works by comparing the actual labels of the data to the predicted labels. The confusion matrix can be helpful in determining which classes the model is having problem accurately classifying and may suggest further model improvements.

3.6.1.3. F1 Score:

The F1 score is a measure of a model's accuracy that considers both precision and recall. It is the harmonic mean of precision and recall, and it ranges from 0 to 1. As F1 score provides a single score that summarizes the model's performance score can be helpful in comparing models or tuning hyper-parameters.

3.6.1.4. Recall:

Recall is a way to measure how well a model can find true positives, or how many true positive cases it correctly identified. Recall can be helpful when it's important to find positive cases for example in medical diagnoses or fraud detection.

3.6.1.5. ROC-AUC:

The ROC-AUC (Receiver Operating Characteristic - Area Under the Curve) is a measure of a model's ability to distinguish between positive and negative classes. It is calculated by plotting the true positive rate against the false positive rate at various classification thresholds. ROC-AUC can be helpful in assessing the overall performance of the model and comparing different models.

Table 3.4: Evaluation Metric

Evaluation Metric	Equation
Accuracy	(TP + TN) / (TP + TN + FP + FN)
Confusion Matrix	A confusion matrix is obtained by comparing the predicted labels of a model with the true labels of a dataset.
F1 Score	2 * (precision * recall) / (precision + recall)
Recall	TP / (TP + FN)

ROC-AUC	The ROC curve is obtained by plotting the true positive rate (TPR) against the false positive rate (FPR) at different threshold values.

Note: TP = True Positive, TN = True Negative, FP = False Positive, FN = False Negative.

3.7. Hypotheses and Research Questions:

Our study aims to address the following research questions which are describe in Table 3.5.

Table 3.5: Research Questions and Hypothesis

Research Question	Hypothesis
How can we construct datasets with larger sets of examples by considering more features and families/classes?	By including a larger number of features and malware families in the dataset, we hypothesize that the DNN will have a better understanding of the malware landscape and will be able to classify new, unseen samples with higher accuracy.
How can we develop a deep learning approach based models that make use of a bigger range of malware classes and features in the dataset and improve efficiency of existing systems?	We hypothesize that specific deep learning models, DNN model outperforms the CNN model in terms of accuracy and generalization for malware detection tasks, according to our preliminary evaluation. By conducting a thorough comparison and evaluation of various deep learning models, including DNN and CNN models, on balanced datasets, we hope to validate our hypothesis and identify the model with the highest performance for malware detection.
What is comparison of deep learning-based classifiers that can be employed to identify malware with and without the possibility of handling zero-day attacks?	Hypothesis is done using deep learning pre-trained models, dataset is fine-tuned for known threats, it will enhance the identification of zero-day attacks. To analyze static application features transfer learning is used in order to identify zero-day attacks. It will enable the model to effectively detect

	previously unknown attack patterns and improve the overall performance of zero-day attack identification by utilizing the knowledge gained from the known threats.

In order to answer these research questions and test our hypotheses, we followed a methodology that involved data preprocessing, model training, and evaluation using various performance metrics such as training accuracy, confusion matrix, F1 score, recall, and ROC-AUC. The details of our methodology and experimental setup are described in the following sections

3.8. Software and Tools Used:

Description of any software tools or frameworks used in the study are shown in Table 3.6.

Table 3.6 : Software and Tools

Task	Software/Tools	Purpose	Reference
Data Acquisition	Custom Tool	Extracting APK files and their static features	Mentioned in Data Acquisition section
Feature Extraction	Custom Tool	Extracting features from the collected applications	Mentioned in Data Acquisition section
Data Preprocessing	Pandas	Cleaning and preparing data for model training	[76]
Model Development	Keras with TensorFlow backend	Building and training deep learning models	[77]
Model Evaluation	Scikit-learn	Evaluating model performance using metrics such as accuracy, confusion matrix, F1 score, recall, and ROC-AUC	[78]
Visualization	Matplotlib, Seaborn	Creating visualizations of data and model performance	[79]

Other	Jupyter Notebook, Python 3.11.8	Environment for coding, analysis, and report writing	[80]

3.9. Implementation Details:

To implement our proposed approach,, we made a set of Python tools that handle the steps of getting data and training the DNN. We performed the following steps in order to get the information we require:

1. We downloaded a large number of malware samples from different sources through internet and collect them in on our local machine.
2. After that we used a Python script that used the Apktool tool to decode the apps and get the AndroidManifest.xml files out of these samples to get the APK files.
3. We extracted the AndroidManifest.xml files from the samples of malware and stored them in a separate directory.

Similarly, for the benign samples, we followed these steps:

1. We downloaded a large number of benign APK files from the Google Play Store using third party app store such as APK Pure and stored them in a directory on our local machine.
2. We used a Python script that manipulate the Apktool tool to extract the AndroidManifest.xml files from these malware and benign samples.
3. We collected the AndroidManifest.xml files from the benign samples and stored them in a separate directory.

For the DNN training phase, we performed the following steps:

1. **Preprocessing:** We preprocessed the data by converting the collected AndroidManifest.xml files into a format that can be fed into the DNN model. We also performed data cleaning and filtering to remove irrelevant or duplicate information.
2. **Building the architecture:** We designed and built a DNN model using a Python deep learning library such as Keras or Tensor Flow. The architecture was designed to take the preprocessed data as input and output the predicted malware/benign label.
3. **Training the model:** We used the preprocessed data to train the DNN model, and we changed the hyper parameters to get the best results. We also used methods like cross-validation to check how well the model worked.
4. **Visualizing the performance:** We used different visualizing performance metrics to measure how well the learned model worked so the used performance metric are Confusion metric, accuracy, precision, recall, and F1 score. We also visualized the performance using graphs and charts.
5. **Adjusting hyper-parameters:** Adjusting hyper-parameters: If needed, we changed the DNN model's hyper-parameters to make it work even better. We did the training and testing steps over and over until we got the desired level of accuracy and performance

For handling the zero-day attack detection, we transferred the knowledge of previously trained DNN model to another DNN Model:

1. **Loading Pre-trained DNN:** In this step, we loaded a previously trained Deep Neural Network (DNN) model using a Python deep learning library such as Keras. This pre-trained model had been trained on a related task or dataset and contained valuable knowledge that we wanted to transfer to our new model for zero-day attack detection.
2. **Detecting anomalies:** We applied anomaly detection techniques such as K-Means clustering method to identify potential anomalies or deviations in our dataset. These anomalies might represent unknown or zero-day attacks that do not conform to the expected patterns of benign or known malicious apps.
3. **Updating the anomalies as potential zero day attacks:** After detecting potential anomalies in the dataset, we updated the labels or annotations of these instances to mark them as "potential zero-day attacks." This label modification allowed us to differentiate these instances during the training of the new DNN model.

4. **Building a new DNN architecture:** We designed and built a DNN model using a Python deep learning library such as Keras or Tensor Flow. The architecture was designed to take the preprocessed data as input and output the predicted malware/benign label.

5. **Transferring knowledge of pre-trained DNN to newly designed DNN:** We performed knowledge transfer by initializing the weights and architecture of our newly designed DNN model with those from the pre-trained DNN. This process allowed our new model to inherit valuable features and patterns learned from the related task, providing it with a strong starting point for zero-day attack detection.

6. **Training the model:** We used the preprocessed data to train the DNN model, and we changed the hyper parameters to get the best results. We also used methods like cross-validation to check how well the model worked.

7. **Visualizing the performance:** We used different visualizing performance metrics to measure how well the learned model worked so the used performance metric are Confusion metric, accuracy, precision, recall, and F1 score. We also visualized the performance using graphs and charts.

8. **Adjusting hyper-parameters:** Adjusting hyper-parameters: If needed, we changed the DNN model's hyper-parameters to make it work even better. We did the training and testing steps over and over until we got the desired level of accuracy and performance we needed XML, Java, and resources. In our Python tools, we used APKTOOL to get the AndroidManifest.xml files from the APK files and decode them. APKTOOL is an useful open- source tool to reverse engineer Android apps and extract various files such as XML, Java, and resources.

3.10. Model Selection Criteria:

We experimented with different DNN architectures, such as simple feedforward neural networks and more complex ones such as convolutional neural networks (CNNs). After looking at each design accuracy, we decided to use a simple feedforward DNN for our approach. We chose a feedforward DNN because it is simple and easy to train, yet powerful enough to learn complex patterns in the AndroidManifest.xml files. Moreover, we found that a feedforward DNN was able to achieve high accuracy on our dataset without overfitting or requiring excessive computational resources. We also implemented a transfer learning approach by leveraging the knowledge gained from one Deep Neural Network (DNN) model and applying it to another DNN model. To achieve this, we provided 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.

The selected DNN architecture consists of several fully connected layers with ReLU activation functions, followed by a final output layer with a softmax activation function. We used the categorical cross-entropy loss function and the Adam optimizer to train the DNN.

3.11. Feature Extraction and Selection:

In our approach, we extracted features from the AndroidManifest.xml file of the APK samples. We downloaded both benign and malware APK samples and collect the required features from the AndroidManifest.xml file of each APK sample by executing the feature extractor tool that we developed. These extracted features were then used to train our deep neural network model.

After the feature extraction, to select only the relevant features to use in our model, we performed feature selection. This step is necessary to make the model less complicated and stop to avoid overfitting We counted how many 1s were in each column of the feature matrix to find out how often each feature shows up in the dataset. We then set a threshold to remove features with fewer than a certain number of ones. For example, we set a threshold of 2, which means that any feature is less than 2 should be eliminated from the dataset. This process focus on the most important features and helps to remove features that are less important. The selected subset of features was then used to test and train the deep neural network model.

3.12. Algorithmic Details:

The architecture of our DNN consists of four layers, including an input layer, three hidden layers with ReLU activation, and an output layer with a softmax activation, as presented in Algorithm 1. The input layer receives nine features, which are extracted from the AndroidManifest.xml files of the apk samples.

The ReLU activation function is defined as $f(x) = max(0,x)$, which means that it outputs the maximum between 0 and the input value. The ReLU activation is used in the three hidden layers to introduce non-linearity to the model and help it learn complex patterns in the data.

The output layer has 12 nodes, where 11 nodes correspond to the 11 malware subclasses, and the last node corresponds to the benign class. The softmax activation function is applied to the output layer, which outputs a probability distribution over the 12 classes. The softmax function is defined as follows:

$$softmax(a\text{-}i) = exp(a\_i) / sum(exp(a\_i)) \text{ for all } j$$

as in equation ai is the input to the i-th output node, and j is an index overall output nodes. The softmax function ensures that the sum of the outputs of all nodes is equal to 1, which gives us a probability distribution.

We used ***categorical_crossentropy*** as the loss function for the model. The categorical_cross entropy measures the difference between the predicted probability distribution and the true probability distribution. Mathematically, the categorical_crossentropy is defined as:

$$Categorical\_Crossentropy = -sum(y\_true * log(y\_pred))$$

where y_true is the true distribution of probabilities and y_pred is the distribution of probabilities that was expected.

ALGORITHM 1:

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

function DNN_Model(T, t, E, B):

1: Normalize T (0~1)

2: Reshape T

3: Initialize DNN architecture:

4 * Dense(unit = 2^n, activation=relu, kernal_regularizer =l2(0.0001) + Dropout(rate = 0.2)

output layer = Dense(units =12, activation = softmax)

4: Optimisation Settings: optimiser = Adam(learning_rate=0.0001),

loss = categorical crossentropy, metrics = accuracy

5: for epoch in range(E**):**

6: for i in range(0, len(T**), B):**

6: batch = T[i:i+B]

7: Train w.r.t B from T

9: Calculate loss for the B

8: if Wrong Prediction then

9: Update W

10: end if

11: end for

12: end for

13: Save the trained model

14: Result = Predict on t

15: Save Result

end function

 # Usage example:
T = X_train #training features
t = X_test #testing features
E = 200 # Number of epochs
B = 64 # Batch size
input_dim = X_train.shape[1] # Input dimensionality
DNN_Model(T, t, E, B)

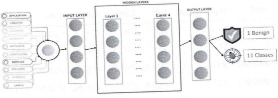


Figure 3.7:Deep Neural Network Architecture

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. Mathematically, the L2 regularization term is defined as:

L2_Regularization = 0.0001 * sum(w^2) where w is the weight of the model.

In our approach, we used the early stopping functionality to stop the training process when the validation loss stopped 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 3. and the proposed algorithm is presented in Algorithm 1.

3.13. Architecture of DNN:

The Figure 3.7 show the details of the each layer, including the number of input nodes, the number of nodes in each hidden layer, the corresponding weights, and biases. The total number of parameters is calculated by summing the individual components, resulting in a value of 5,560,576. Additionally, a total of 972 biases are considered in the network. We present the architecture of our CNN based model in Algorithm 2 where the architecture of our proposed model using DNN is shown in Figure 3.8.

		HL1	HL2	HL3	HL4	Output Layer	Total
0	Input	10523	512	256	128	64	*
1	Nodes	512	256	128	64	12	*
2	Weights (%)	96.89%	2.36%	0.59%	0.15%	0.01%	100.00%
3	Biases (%)	52.67%	26.34%	13.17%	6.58%	1.23%	100.00%

Figure 3.7 : Architecture of DNN

ALGORITHM 2:

-- **Require:** Dataset-Training X_train, Dataset-Training Labels y_train, Epochs E, Batch-Size B

Function CNN_Model(X_train, y_train, E, B):
1: Normalize X_train (0~1)
2: Reshape X_train
3: Initialize CNN model:
Initialize CNN model:
4 * Conv1D(2^n, 3, activation='relu', input_shape=(size, 1)) + MaxPooling1D(2) + Flatten() BatchNormalization() + Dropout(0.3)
Output Layer = Dense(12, activation='softmax')
4: **Optimisation Settings:** optimiser = Adam(learning_rate=0.0001), loss = categorical crossentropy, metrics = accuracy 5: for epoch in range(E):
5: **for i in range(0, len(X_train), B):**
6: batch = X_train[i:i+B]
7: Train the model on the batch
8: Calculate loss for the batch
9: **if Wrong Prediction:**
10:Update the model weights
11: **end if**
12: **end for**
13: **end for** 14: Save the trained model to a file **end function**

 # Usage example:
X_train = ... # Training data
y_train = ... # Training labels
E = 200 # Number of epochs
B = 64 # Batch size
CNN_Model(X_train, y_train, E, B)

3.14. Architecture of CNN:

We construct a CNN model. It consists of four Conv1D layers, each with a different filter size determined by 2^n, followed by a ReLU activation function. After each convolutional layer, there's 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 2.

3.15. Architecture of TL based Model:

We construct a new deep neural network model by leveraging 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 zero-day attack detection, we make a crucial modification to the output layer. Assuming we have 13 classes, including one for zero-day attacks, we tailor the network accordingly. 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. In this evaluation, we primarily measure accuracy as an indicator of how well the model can classify data into its respective classes, which includes identifying zero-day attacks when they occur. We present the architecture of our TL based model in Algorithm 3.

ALGORITHM 3:

```
# Required Dataset-Training X_train, Dataset-Training Labels y_train, Epochs E, Batch-Size B,
pretrained_model

# Function for DNN Model with Transfer Learning
def DNN_Model_Transfer_Learning(T, t, E, B, pretrained_model):
  # Normalize T (0~1) and reshape
  T = T / np.max(T),    T = T.reshape(T.shape[0], -1)
  # Initialize DNN architecture
  model = Sequential()
  for layer in pretrained_model.layers:
     model.add(layer)
  # Modify the output layer for zero-day detection (assuming 13 classes)
  model.layers[-1].output_dim = 13  # Assuming 13 classes (12 original + 1 zero-day)
  model.layers[-1].activation = 'softmax'  # Adjust activation function
  # Optimization settings
  optimizer = Adam(learning_rate=0.0001)
  model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
  # Training loop
  for epoch in range(E):
     for i in range(0, len(T), B):
        batch = T[i:i+B]
        # Training w.r.t B from T
        # Calculate loss for the batch
        loss = model.train_on_batch(batch, batch)  # Assuming reconstruction loss
        # If loss is above a threshold, update weights (indicating potential zero-day attack)
        threshold = 0.1  # Adjust this threshold as needed
        if loss > threshold:
           # Update weights or perform further actions as needed
           model.layers[-1].set_weights(new_weights)
           pass
  Save the trained model
  model.save("zero_day_detection_model.h5")

  Evaluation on testing data
  t = t / np.max(t)
  t = t.reshape(t.shape[0], -1)
  y_pred = model.predict(t)
  y_true = t

  Calculate accuracy (you can use other metrics as well)
  accuracy = accuracy_score(np.argmax(y_true, axis=1), np.argmax(y_pred, axis=1))
  print("Test Accuracy:", accuracy)
```

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. In practice, this function can be applied to a specific dataset (X_train and y_train) by specifying the number of training epochs (E) and the batch size (B) to create and train a custom CNN model.

3.16. Evaluation Procedure for DNN models:

To assess the performance of our DNN model, we divided the dataset into a training set and a testing set using an 80:20 ratios. We used the testing set to evaluate its performance and the training set to train the DNN model. Specifically, we used 80% for training and 20% of the dataset for testing and validation. During the training phase, we used to adjust the hyper-parameters and the training set to update the weights of the model. Then, we evaluated the model using the testing set to ensure that it could generalize well to new, unseen data. We validated the performance of our model using the evaluation metrics mentioned earlier, including accuracy, confusion matrix, F1 score, recall, and ROC-AUC. We utilized the 20% testing set to validate our DNN using these metrics, which allowed us to measure the model's performance on unseen data. This approach ensured that our model was not overfitting to the training data and was able to generalize to new data.

4. RESULTS AND DISCUSSION

The evaluation and analysis of the proposed anomaly detection system's efficiency and accuracy rely on a set of established and standardized performance metrics. These metrics serve as a benchmark to measure the system's performance and provide a quantitative basis for evaluating its effectiveness [1].

In our work, we have implemented several performance metrics for evaluation, including Accuracy, Recall, Precision, ROC, AUC, and F1-Score. The Confusion Matrix is utilized to present the actual values of True Negative (TN), True Positive (TP), False Negative (FN), and False Positive (FP). When dealing with balanced 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. To elaborate on some of the performance metrics shown in **Table *4.1***.

Table 4.1 : Metrics

Metric	Formula
Accuracy	(TP + TN) / (TP + TN + FP + FN)
Precision	TP / (TP + FP)
Recall	TP / (TP + FN)
F1-Score	2 * ((Precision * Recall) / (Precision + Recall))
AU-ROC	Area Under the Receiver Operating Characteristic (computed graphically)

By utilizing these performance metrics, we gain a comprehensive understanding of the effectiveness and efficiency of our anomaly detection system in accurately detecting and classifying malware in the Android malware detection domain.

4.1. Comparative Analysis:

In this section, we compare our approach to the base approach [8] used in android malware detection. The following Table 4.2 summarizes the key differences between the two approaches.

Table 4.2 : Comparative Analysis

S#	Our Approach	Base Approach [8]
1	Our model utilizes a CPU-powered, DL-based detection algorithm that is efficient enough to operate on devices without GPU compatibility.	The base model relies on CUDA-empowered DL technology and requires GPU hardware for optimal efficiency.
2	The developed algorithm is capable of detecting up to 12 distinct variants of malware, showcasing advanced knowledge. This expertise enables the identification of newly introduced malware models, as they often incorporate functionalities from existing malware types or employ similar attack methods.	The base model is based on only 3 different variants and may not detect as many malware types as our model. In reality, there are hundreds of diverse malware types, some of which are entirely dissimilar from one another.
3	In order to assess the efficiency of our model, we utilize our proprietary dataset, consisting of a substantial number of carefully collected APKs. This unique dataset not only surprises potential attackers but also poses a challenge for them to comprehend the underlying concept of our developed approach.	On the other hand, the base model evaluates its efficiency using publicly available datasets, which may be familiar to attackers who possess knowledge of these datasets.
4	Number of application for each class balanced used so advantage is that each malware class and benign apks equal it will increase accuracy, Enhanced generalizability, balanced representation	Number of application for each class unbalanced so it will lead to Bias in the results, Reduced generalizability, minority classes can result in limited understanding of their characteristics and

	of each class enables a thorough understanding of the characteristics, behaviors, and patterns exhibited by different malware classes and benign applications, robust evaluation, ultimately leading to more reliable	behaviors. Lead to incomplete evaluation. When evaluating the performance of the model, an unbalanced dataset can skew the assessment metrics, such as accuracy, precision, and recall. This distortion can obscure the true effectiveness and efficiency of the model's performance on the various classes.
5	Given the limited amount of data samples, our approach of using a 80:20 train-test split with model. Fit() is superior to the base paper's cross-validation approach using 10 folds. By allocating 80% of the data for training, our approach allows the model to learn from a more representative and diverse set of samples, enabling better generalization and capturing complex patterns. Additionally, using a single, well-defined test set eliminates the variability introduced by different folds in cross-validation. This approach also saves computational resources by training the model once instead of multiple times in cross-validation.	The base model's use of a 10-fold cross-validation approach may be less effective with limited data due to several reasons. For first of all, splitting the dataset into many folds might result in smaller training sets, reducing the model's capacity to capture the full range of variations and complexity.
7	Our model utilizes an extensive feature set of 43,377, even after implementing preprocessing techniques. Through careful selection, we have reduced the number of features to around 10,524. This surpasses the base model's utilization of only 190 features for training. Our model's extensive feature	The base model has limited feature representation, utilizing only 190 features for training. This restricted feature coverage may hinder its ability to capture the full range of information in the dataset. The base model's limited feature representation may limit its effectiveness in detecting and classifying malwares.

selection and retention give it a significant advantage over the base model in terms of accurately detecting and classifying malwares.

Our approach demonstrates several advantages over the base approach, including efficient CPU-powered detection, the ability to detect a greater number of malware variants, utilization of a proprietary dataset, balanced representation of each class, consideration of a diverse feature set, and an optimized training-test split. These advantages contribute to improved accuracy, enhanced generalizability, and increased robustness in detecting and classifying android malwares.

4.1.1.1. DNN vs CNN vs TL:

In this section, 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 4.3 We evaluate the models based on several key metrics, including training time, accuracy, loss, confusion matrix, AUC-ROC, recall, F1 score, and precision.

Table 4.3: DNN vs CNN vs TL

S#	ITEM	DNN	CNN	TL
1	TRAINING-TIME (aprox)	53.33 Minutes	2 Hours & 53 Minutes	05 Minutes
2	EPOCH	194	200	50
3	ACCURACY	97.62%	96.44%	94.45
4	Loss	0.1424	0.1278	0.3550
5	CONFUSION MATRIX	Figure 4.1	Figure 4.2	Figure 4.6
6	AUC-ROC	**0.98 (*Figure 4.1*)**	**0.97 (*Figure 4.2*)**	**1.0 (*Figure 4.3*)**
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

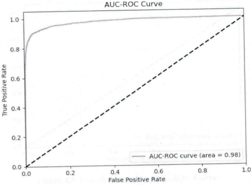


Figure 4.1 DNN AUC_ROC Curve

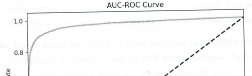


Figure 4.2 CNN AUC-ROC Curve

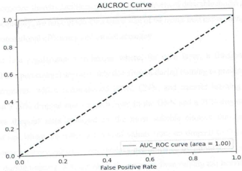


Figure 4.3 Transfer Learning AUC-ROC Curve

4.1.1.2. Selection of Hyper-Parameters:

We employ various hyper parameters to enhance the performance of our deep learning algorithm, as highlighted in the study[81]. Our approach involves the following components:

1. Epochs: Epochs denote the training iterations performed 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 most promising experiment.

2. Batch-size: The batch size denotes the number of data samples processed together during each training iteration. In our series of experiments, we systematically 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 DNN). Importantly, the preference for a batch size of 64 aligns with our dataset's substantial nature, encompassing over 10,000 features. In this context, larger batch sizes expedite training while mitigating the risk of slower convergence, thereby facilitating the accomplishment of desirable model performance levels. Consequently, we have opted for a batch size of 64 in our best experiments, balancing between computational efficiency and model accuracy.

3. Dropout: is a regularization technique where, for each layer, a fraction of neurons (represented as a percentage) are randomly deactivated during training to prevent overfitting. In our experiments, which encompassed DNN, CNN, and transfer learning models, 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%. This 20/30 percent dropout strategy enhances our model's performance by introducing an element of uncertainty during training. It discourages the network from relying too heavily on specific neurons, promoting the learning of more robust and generalized features. By preventing overfitting, it allows our models to better adapt to new, unseen data, ultimately improving their ability to make accurate predictions and classifications.

4.Optimizer function: The optimizer function plays a crucial role in adjusting weights to minimize error rates. Our study utilizes the 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 transfer learning ensures stable and efficient fine-tuning for new tasks. This approach accelerates convergence, prevents overly slow learning rates, and mitigates the risk of overfitting, resulting in improved and quicker model adaptation.

5.Activation Function: In our neural network, we deliberately 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. For the output layer, we made a deliberate choice to utilize the softmax activation function. 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, aiding in the straightforward identification of the most likely class among several possibilities.

Overall, the DNN methodology outperformed the CNN and TL method 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. The confusion metrics of them three models depict in Figure 4.4, Figure 4.5 and Figure 4.6.

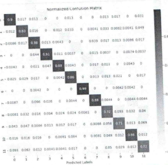


Figure 4.4: DNN Confusion Matrix

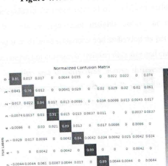


Figure 4.5: CNN Confusion Matrix

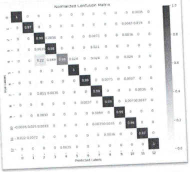


Figure 4.6: TL Confusion Matrix

4.2. Robustness and Sensitivity Analysis:

In this section, 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 differed 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 understanding of the model's capabilities and uncover important considerations for its deployment in real-world scenarios.

4.2.1.1. Sensitivity Analysis by unbalancing the dataset:

```
5     1152
4     1102
7     1095
1     1015
12    1011
6      934
11     933
2      930
9      911
8      880
3      851
10     847
Name: Malware Class, dtype: int64
```

Figure 4.7: Count of samples after unbalancing

We conducted a sensitivity analysis by changing the dataset from balanced to unbalanced. For the purpose of our analysis, we deliberately introduced an element of randomness into the dataset by randomly excluding certain APKs, leading to an intentionally unbalanced dataset. The ***Figure 4.7*** below visually represents this dataset, where each class is denoted by a numerical representation. In the adjacent column, we provide the specific count of APKs included in this analysis for each respective class. 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 ***Table 4.4***

Table 4.4: Results of sensitivity Analysis by unbalancing the dataset

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
Confusion Matrix	Figure 4.8

The sensitivity analysis results demonstrate the resilience and adaptability of our DNN model to unbalanced datasets. Despite the inherent challenges associated with uneven class distributions, our model maintained high accuracy, precision, recall, ROC and AUC, and F1 score. This underscores the effectiveness and reliability of our approach in detecting and classifying android malware instances, regardless of the class distribution challenges. As it shown in the tabular form in Table 4.4.

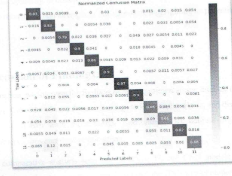


Figure 4.8: Confusion Matrix

4.2.1.2. Cross Validation

In addition, we also conducted a sensitivity analysis results of our DNN model by changing the test and train splits to 80% for testing and 20% for training. 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 4.5 20% and 80% split results are shown.

Table 4.5: 20% training and 80% testing split:

Epoch	200
Training Time	17 Minutes
Loss	0.1985
Accuracy	96.15%
Precision	0.72
Recall	0.71
ROC and AUC	0.94
F1 Score	0.71

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.

4.3. Discussion of Findings:

In this section, we interpret the findings of our experimentations, analyze the results within the context of Android malware detection using static feature analysis, and discuss the insights and conclusions drawn from the findings. Additionally, we address any unexpected or interesting observations and their implications.

Our comparative analysis between the DNN, TL and CNN models revealed several significant findings. Firstly, in terms of training time, the DNN model outperformed the CNN model, requiring only 53.33 minutes compared to the CNN model's 2 hours and 53 minutes whereas TL model requires 5 minutes. While our results undeniably showcase that Transfer Learning (TL) achieved comparable performance with just 50 epochs and a mere 5 minutes of training, it's imperative to emphasize that transfer learning is an augmentation to the training time and epochs initially considered by the pre-trained Deep Neural Network (DNN). This underscores the fact that the DNN model provides a notably time-efficient solution for Android malware detection when compared to the additional training time and resources required for transfer learning. Moreover, the DNN model achieved higher accuracy, with a rate of 97.62%, surpassing the CNN model's accuracy of 96.44% and TL model's accuracy is 94.45%. This suggests that the DNN model excels in accurately classifying Android applications as malicious or benign.

The AUC-ROC scores for these models were quite high, with the DNN model achieving a score of 0.98 and the CNN model achieving 0.97 and TL model achieving 1.00. This demonstrates the models' ability to effectively differentiate between positive and negative instances, further validating their effectiveness in detecting Android malware and zero-day attack.

Interestingly, despite the differences in architecture and training approach, both models performed comparably in terms of recall, precision, and F1 score, showcasing their robustness in capturing true positives, false positives, and achieving a balanced performance.

4.3.1.1. Statistical Analysis:

P-Value Analysis of Analysis of Model:

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% further supports the rejection of H0 and strengthens the claim that the DNN model is effective in identifying malware. These findings emphasize the potential practical value of the DNN model in combating malware threats and provide compelling evidence for its inclusion in malware detection systems.

P-Value Analysis of Analysis of Dataset:

In this research paper, we delve into a rigorous statistical analysis of our dataset, employing P-value analysis as a powerful tool for assessment. Our investigation comprises two distinct tests designed to unveil the relationship between the dataset columns and the malware label column.

> **Test 1:** Top 50 Correlated Columns For our initial examination, we picked the top 50 columns exhibiting the highest correlation with the malware label column. 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 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.
>
> **Test 2:** All 10,523 Columns Our second test extended the analysis to encompass all 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 4.9***, 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 4.10*** Failed P-Value Analysis, highlighting the complexity and diversity of factors present in the dataset.

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 4.9: Passed 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

Figure 4.10 Failed P-Value Analys

CONCLUSION

In this chapter, we present our conclusions. We proposed a deep learning-based approach for addressing the security concerns associated with Android-based applications, specifically focusing on malware detection. It's worth noting that while Deep Neural Networks (DNNs) have been explored in this domain previously, our algorithm represents a significant advancement. 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. Our proposed detection method is intended to be effective and scalable, and protect against complex multi-threats and attacks. Transfer Learning, Convolutional neural networks, and deep neural networks are used by the system to combat the increasing cyber threats and attacks posed on by Android malware. In essence, this chapter highlights our commitment to enhancing Android security. We aim to protect users from malware effectively while keeping our approach adaptable to evolving threats.

We evaluate the performance of our proposed mechanism using our own datasets which contain more number of feature and more number of classes and benchmark deep learning algorithms. The results obtained from the evaluation process are rigorously validated, providing clear and unbiased insights into the system's performance. We employ various performance metrics such as Recall, F1 score, Confusion Matrix, Accuracy, AUC (Area Under the Curve), ROC (Receiver Operating Characteristic), P-value analysis to assess the effectiveness of our multi-threat malware detection techniques, considering both detection accuracy and time efficiency.

We gained substantial insights into the robustness and adaptability of our model through its application in various scenarios and subsequent performance analysis under different settings. Our investigation into the effects of alterations in training data size and class distribution has illuminated the model's stability and efficacy across diverse conditions. These comprehensive analyses have provided us with a profound comprehension of the model's capabilities and have revealed crucial insights for its practical implementation in real-world situations. The experimental results demonstrate that our approach achieves high detection accuracy with DNN while maintaining efficient processing times. The deep neural networks

(DNNs) train model give accuracy of 97.62%,. Whereas 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 (TL), where the accuracy reaches 94.45% with an addition of detecting zero day attacks alongside the other 12 malware families we considered. This highlights the proficiency of our system in addressing the complex challenges posed by multi-threat malware and detecting the zero-day attacks in Android environments.

As a future work, we plan to execute the same incorporating develop and make public large datasets so that the research fraternity may benefit and develop more robust models for Android malware detection. We aspire to leverage the insights gained from our work to develop cutting-edge antivirus software or productivity tools for the market. This proactive approach serve the broader audience by enhancing their digital security and productivity as well as give benefit to the researchers. By bridging the gap between academic research and practical applications, we aim to make a meaningful impact in the cybersecurity and productivity software industries, ultimately contributing to a safer and more efficient digital ecosystem.

REFERENCES

[1] P. Faruki *et al.*, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Commun. Surv. Tutorials*, vol. 17, no. 2, pp. 998–1022, Apr. 2015, doi: 10.1109/COMST.2014.2386139.

[2] "M. Labs, 'McAfee Labs 2020 Threats Predictions Report | McAfee Blog,' McAfee Blog, Dec. 05, 2019. https://www.mcafee.com/blogs/other-blogs/mcafee-labs/mcafee-labs-2020-threats-predictions-report/ (accessed Jul. 07, 2023)."

[3] https://www.statista.com/statistics/687566/number-of-android-users/, "Statista. (2022). Number of active Android users worldwide from 2016 to 2022 (in millions).," Jan. 2022.

[4] A. Al Zaabi and D. Mouheb, "Android Malware Detection Using Static Features and Machine Learning," in *Proceedings of the 2020 IEEE International Conference on Communications, Computing, Cybersecurity, and Informatics, CCCI 2020*, Institute of Electrical and Electronics Engineers Inc., Nov. 2020. doi: 10.1109/CCCI49893.2020.9256450.

[5] N. Bhodia, P. Prajapati, F. Di Troia, and M. Stamp, "Transfer Learning for Image-Based Malware Classification," Jan. 2019, [Online]. Available: http://arxiv.org/abs/1903.11551

[6] H. Farhat and V. Rammouz, "Malware Classification Using Transfer Learning," Jul. 2021, [Online]. Available: http://arxiv.org/abs/2107.13743

[7] E. M. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: Automatic framework for android malware detection using deep learning," in *DFRWS 2018 EU - Proceedings of the 5th Annual DFRWS Europe*, Digital Forensic Research Workshop, 2018, pp. S48–S59. doi: 10.1016/j.diin.2018.01.007.

[8] I. U. Haq, T. A. Khan, and A. Akhunzada, "A Dynamic Robust DL-Based Model for Android Malware Detection," *IEEE Access*, vol. 9, pp. 74510–74521, 2021, doi: 10.1109/ACCESS.2021.3079370.

[9] R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, and S. Venkatraman, "Robust Intelligent Malware Detection Using Deep Learning," *IEEE Access*, vol. 7, pp. 46717–46738, 2019, doi: 10.1109/ACCESS.2019.2906934.

[10] Z. Liu, R. Wang, N. Japkowicz, D. Tang, W. Zhang, and J. Zhao, "Research on unsupervised feature learning for Android malware detection based on Restricted Boltzmann Machines," *Futur. Gener. Comput. Syst.*, vol. 120, pp. 91–108, 2021, doi: 10.1016/j.future.2021.02.015.

[11] N. Milosevic, A. Dehghantanha, and K. K. R. Choo, "Machine learning aided Android malware classification," *Comput. Electr. Eng.*, vol. 61, pp. 266–274, 2017, doi: 10.1016/j.compeleceng.2017.02.013.

[12] D. Kiefer, M. Bauer, F. Grimm, and C. van Dinther, "53rd Hawaii International Conference on System Sciences (HICCS), online, January 5-8, 2021," in *54th Hawaii International Conference on System Sciences*, University of Hawai'i at Manoa, 2021.

[13] E. C. Bayazit, O. K. Sahingoz, and B. Dogan, "A Deep Learning Based Android Malware Detection System with Static Analysis," in *HORA 2022 - 4th International Congress on Human-Computer Interaction, Optimization and Robotic Applications, Proceedings*, Institute of Electrical and Electronics Engineers Inc., 2022. doi: 10.1109/HORA55278.2022.9800057.

[14] I. Bibi, A. Akhunzada, J. Malik, G. Ahmed, and M. Raza, "An Effective Android Ransomware Detection Through Multi-Factor Feature Filtration and Recurrent Neural Network," *2019 UK/China Emerg. Technol. UCET 2019*, pp. 1–4, 2019, doi: 10.1109/UCET.2019.8881884.

[15] Y. Zhang, Y. Yang, and X. Wang, "A novel android malware detection approach based on

convolutional neural network," *ACM Int. Conf. Proceeding Ser.*, pp. 144–149, 2018, doi: 10.1145/3199478.3199492.

[16] H. Zhu, Y. Li, R. Li, J. Li, Z. You, and H. Song, "SEDMDroid: An Enhanced Stacking Ensemble Framework for Android Malware Detection," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 984–994, Apr. 2021, doi: 10.1109/TNSE.2020.2996379.

[17] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti, "ANASTASIA: ANdroid mAlware detection using STatic analySIs of applications," *2016 8th IFIP Int. Conf. New Technol. Mobil. Secur. NTMS 2016*, pp. 1–5, 2016, doi: 10.1109/NTMS.2016.7792435.

[18] A. Djenna, A. Bouridane, S. Rubab, and I. M. Marou, "Artificial Intelligence-Based Malware Detection, Analysis, and Mitigation," *Symmetry (Basel).*, vol. 15, no. 3, p. 677, Mar. 2023, doi: 10.3390/sym15030677.

[19] C. D. Nguyen, N. H. Khoa, K. N. D. Doan, and N. T. Cam, "Android Malware Category and Family Classification Using Static Analysis," *Int. Conf. Inf. Netw.*, vol. 2023-Janua, pp. 162–167, 2023, doi: 10.1109/ICOIN56518.2023.10049039.

[20] M. L. Anupama *et al.*, "Detection and robustness evaluation of android malware classifiers," *J. Comput. Virol. Hacking Tech.*, vol. 18, no. 3, pp. 147–170, Sep. 2022, doi: 10.1007/s11416-021-00390-2.

[21] R. Poorvadevi, N. B. Keerthi, and N. V. Lakshmi, "Android Malware Identification and Detection using Deep Learning," in *2022 6th International Conference on Trends in Electronics and Informatics, ICOEI 2022 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., 2022, pp. 1266–1270. doi: 10.1109/ICOEI53556.2022.9776920.

[22] Y. Huang *et al.*, "Android-SEM: Generative Adversarial Network for Android Malware Semantic Enhancement Model Based on Transfer Learning," *Electron.*, vol. 11, no. 5, Mar. 2022, doi: 10.3390/electronics11050672.

[23] R. Surendran, T. Thomas, and S. Emmanuel, "A TAN based hybrid model for android malware detection," *J. Inf. Secur. Appl.*, vol. 54, Oct. 2020, doi: 10.1016/j.jisa.2020.102483.

[24] J. Mohamad Arif, M. F. Ab Razak, S. Awang, S. R. Tuan Mat, N. S. N. Ismail, and A. Firdaus, "A static analysis approach for Android permission-based malware detection systems," *PLoS One*, vol. 16, no. 9, p. e0257968, 2021, doi: 10.1371/journal.pone.0257968.

[25] S. Yang, S. Li, W. Chen, and Y. Liu, "A Real-Time and Adaptive-Learning Malware Detection Method Based on API-Pair Graph," *IEEE Access*, vol. 8, pp. 208120–208135, 2020, doi: 10.1109/ACCESS.2020.3038453.

[26] D. Ö. Şahin, O. E. Kural, S. Akleylek, and E. Kılıç, "A novel permission-based Android malware detection system using feature selection based on linear regression," *Neural Comput. Appl.*, vol. 35, no. 7, pp. 4903–4918, Mar. 2023, doi: 10.1007/s00521-021-05875-1.

[27] A. T. Kabakus, "What static analysis can utmost offer for android malware detection," *Inf. Technol. Control*, vol. 48, no. 2, pp. 235–249, 2019, doi: 10.5755/j01.itc.48.2.21457.

[28] R. Taheri, M. Ghahramani, R. Javidan, M. Shojafar, Z. Pooranian, and M. Conti, "Similarity-based Android Malware Detection Using Hamming Distance of Static Binary Features," Aug. 2019, [Online]. Available: http://arxiv.org/abs/1908.05759

[29] M. Dhalaria and E. Gandotra, "A Framework for Detection of Android Malware using Static Features," in *2020 IEEE 17th India Council International Conference, INDICON 2020*, Institute of Electrical and Electronics Engineers Inc., Dec. 2020. doi: 10.1109/INDICON49873.2020.9342511.

[30] Z. Ren, H. Wu, Q. Ning, I. Hussain, and B. Chen, "End-to-end malware detection for android

IoT devices using deep learning," *Ad Hoc Networks*, vol. 101, Apr. 2020, doi: 10.1016/j.adhoc.2020.102098.

[31] J. Booz, J. McGiff, W. G. Hatcher, W. Yu, J. Nguyen, and C. Lu, "Towards Deep Learning-Based Approach for Detecting Android Malware," *Int. J. Softw. Innov.*, vol. 7, no. 4, pp. 1–24, 2019, doi: 10.4018/IJSI.2019100101.

[32] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "DL-Droid: Deep learning based android malware detection using real devices," *Comput. Secur.*, vol. 89, Feb. 2020, doi: 10.1016/j.cose.2019.101663.

[33] R. Taheri, M. Ghahramani, R. Javidan, M. Shojafar, Z. Pooranian, and M. Conti, "Similarity-based Android malware detection using Hamming distance of static binary features," *Futur. Gener. Comput. Syst.*, vol. 105, pp. 230–247, Apr. 2020, doi: 10.1016/j.future.2019.11.034.

[34] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An, and H. Ye, "Significant Permission Identification for Machine Learning Based Android Malware Detection."

[35] G. Sun and Q. Qian, "Deep Learning and Visualization for Identifying Malware Families," *IEEE Trans. Dependable Secur. Comput.*, vol. 18, no. 1, pp. 283–295, 2021, doi: 10.1109/TDSC.2018.2884928.

[36] H. Alshahrani, H. Mansourt, S. Thorn, A. Alshehri, A. Alzahrani, and H. Fu, "DDefender: Android application threat detection using static and dynamic analysis," *2018 IEEE Int. Conf. Consum. Electron. ICCE 2018*, vol. 2018-Janua, pp. 1–6, 2018, doi: 10.1109/ICCE.2018.8326293.

[37] R. Nix and J. Zhang, "Classification of Android apps and malware using deep neural networks," *Proc. Int. Jt. Conf. Neural Networks*, vol. 2017-May, pp. 1871–1878, 2017, doi: 10.1109/IJCNN.2017.7966078.

[38] E. M. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Cypider: Building community-based cyber-defense infrastructure for android malware detection," in *ACM International Conference Proceeding Series*, Association for Computing Machinery, Dec. 2016, pp. 348–362. doi: 10.1145/2991079.2991124.

[39] Z. Yuan, Y. Lu, and Y. Xue, "DroidDetector: Android Malware Characterization and Detection Using Deep Learning," 2016.

[40] S. Wu, P. Wang, X. Li, and Y. Zhang, "Effective detection of android malware based on the usage of data flow APIs and machine learning," *Inf. Softw. Technol.*, vol. 75, pp. 17–25, Jul. 2016, doi: 10.1016/j.infsof.2016.03.004.

[41] B. H. Kang and Q. Bai, Eds., *AI 2016: Advances in Artificial Intelligence*, vol. 9992. in Lecture Notes in Computer Science, vol. 9992. Cham: Springer International Publishing, 2016. doi: 10.1007/978-3-319-50127-7.

[42] M. Qiao, A. H. Sung, and Q. Liu, "Merging permission and api features for android malware detection," in *Proceedings - 2016 5th IIAI International Congress on Advanced Applied Informatics, IIAI-AAI 2016*, Institute of Electrical and Electronics Engineers Inc., Aug. 2016, pp. 566–571. doi: 10.1109/IIAI-AAI.2016.237.

[43] X. Su, D. Zhang, W. Li, and K. Zhao, "A deep learning approach to android malware feature learning and detection," *Proc. - 15th IEEE Int. Conf. Trust. Secur. Priv. Comput. Commun. 10th IEEE Int. Conf. Big Data Sci. Eng. 14th IEEE Int. Symp. Parallel Distrib. Proce*, pp. 244–251, 2016, doi: 10.1109/TrustCom.2016.0070.

[44] H. Kang, J. W. Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," *Int. J. Distrib. Sens. Networks*, vol. 2015, 2015, doi: 10.1155/2015/479174.

[45] A. Saad, *ACMSE'13 : 51st ACM Southeast Conference : Savannah, Georgia, April 4-6, 2013*. ACM, 2013.

[46] V. Syrris and D. Geneiatakis, "On machine learning effectiveness for malware detection in Android OS using static analysis data," *J. Inf. Secur. Appl.*, vol. 59, Jun. 2021, doi: 10.1016/j.jisa.2021.102794.

[47] A. Mahindru and A. L. Sangal, "MLDroid—framework for Android malware detection using machine learning techniques," *Neural Comput. Appl.*, vol. 33, no. 10, pp. 5183–5240, May 2021, doi: 10.1007/s00521-020-05309-4.

[48] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "DL-Droid: Deep learning based android malware detection using real devices," *Comput. Secur.*, vol. 89, 2020, doi: 10.1016/j.cose.2019.101663.

[49] R. S. Arslan, I. A. Doğru, and N. Barişçi, "Permission-Based Malware Detection System for Android Using Machine Learning Techniques," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 29, no. 1, pp. 43–61, Jan. 2019, doi: 10.1142/S0218194019500037.

[50] H. R. Sandeep, "Static analysis of android malware detection using deep learning," *2019 Int. Conf. Intell. Comput. Control Syst. ICCS 2019*, no. Iciccs, pp. 841–845, 2019, doi: 10.1109/ICCS45141.2019.9065765.

[51] X. Su, D. Zhang, W. Li, and K. Zhao, "A Deep Learning Approach to Android Malware Feature Learning and Detection," 2016, doi: 10.1109/TrustCom/BigDataSE/ISPA.2016.69.

[52] "s61".

[53] I. Almomani, M. Ahmed, and W. El-Shafai, "Android malware analysis in a nutshell," *PLoS One*, vol. 17, no. 7 July, Jul. 2022, doi: 10.1371/journal.pone.0270647.

[54] C. Nguyen, N. H. Khoa, K. N. Doan, and N. T. Cam, "Android Malware Category and Family Classification Using Static Analysis," *2023 Int. Conf. Inf. Netw.*, pp. 162–167, 2023, doi: 10.1109/ICOIN56518.2023.10049039.

[55] S. Acharya, U. Rawat, and R. Bhatnagar, "A Comprehensive Review of Android Security: Threats, Vulnerabilities, Malware Detection, and Analysis," *Secur. Commun. Networks*, vol. 2022, 2022, doi: 10.1155/2022/7775917.

[56] B. Gencaydin, C. N. Kahya, F. Demirkiran, B. Duzgun, A. Cayir, and H. Dag, "Benchmark Static API Call Datasets for Malware Family Classification," *Proc. - 7th Int. Conf. Comput. Sci. Eng. UBMK 2022*, pp. 137–141, 2022, doi: 10.1109/UBMK55850.2022.9919580.

[57] G. Ye, J. Zhang, H. Li, Z. Tang, and T. Lv, "Android Malware Detection Technology Based on Lightweight Convolutional Neural Networks," *Secur. Commun. Networks*, vol. 2022, 2022, doi: 10.1155/2022/8893764.

[58] J. Booz, J. McGiff, W. G. Hatcher, W. Yu, J. Nguyen, and C. Lu, "Towards Deep Learning-Based Approach for Detecting Android Malware," *Int. J. Softw. Innov.*, vol. 7, no. 4, pp. 1–24, Oct. 2019, doi: 10.4018/IJSI.2019100101.

[59] B. Urooj, M. A. Shah, C. Maple, M. K. Abbasi, and S. Riasat, "Malware Detection: A Framework for Reverse Engineered Android Applications Through Machine Learning Algorithms," *IEEE Access*, vol. 10, pp. 89031–89050, 2022, doi: 10.1109/ACCESS.2022.3149053.

[60] S. Aluru, Jaypee Institute of Information Technology University, University of Florida. College of Engineering, IEEE Computer Society, IEEE Computer Society. Technical Committee on Parallel Processing, and Institute of Electrical and Electronics Engineers, *2018 Eleventh International Conference on Contemporary Computing (IC3) : 2-4 August 2018, Jaypee Institute of Information Technology, Noida, India*.

[61] R. Ali, A. Ali, F. Iqbal, M. Hussain, and F. Ullah, "Deep Learning Methods for Malware and Intrusion Detection: A Systematic Literature Review," *Security and Communication Networks*, vol. 2022. Hindawi Limited, 2022. doi: 10.1155/2022/2959222.

[62] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," *IEEE Trans. Inf. Forensics Secur.*, vol. 14, no. 3, pp. 773–788, Mar. 2019, doi: 10.1109/TIFS.2018.2866319.

[63] S. Acharya, U. Rawat, and R. Bhatnagar, "A Low Computational Cost Method for Mobile Malware Detection Using Transfer Learning and Familial Classification Using Topic Modelling," *Appl. Comput. Intell. Soft Comput.*, vol. 2022, 2022, doi: 10.1155/2022/4119500.

[64] S. Diniz *et al.*, "Editorial Board Members." [Online]. Available: http://www.springer.com/series/7899

[65] G. Sun and Q. Qian, "Deep Learning and Visualization for Identifying Malware Families," *IEEE Trans. Dependable Secur. Comput.*, vol. 18, no. 1, pp. 283–295, Jan. 2021, doi: 10.1109/TDSC.2018.2884928.

[66] H. Kang, J. W. Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," *Int. J. Distrib. Sens. Networks*, vol. 2015, 2015, doi: 10.1155/2015/479174.

[67] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An, and H. Ye, "Significant Permission Identification for Machine-Learning-Based Android Malware Detection," *IEEE Trans. Ind. Informatics*, vol. 14, no. 7, pp. 3216–3225, Jul. 2018, doi: 10.1109/TII.2017.2789219.

[68] *53rd Hawaii International Conference on System Sciences (HICCS), online, January 5-8, 2021.* University of Hawai'i at Manoa, 2021.

[69] M. Junaid, D. Liu, and D. Kung, "Dexteroid: Detecting malicious behaviors in Android apps using reverse-engineered life cycle models," *Comput. Secur.*, vol. 59, pp. 92–117, Jun. 2016, doi: 10.1016/j.cose.2016.01.008.

[70] O. Aslan and R. Samet, "Investigation of possibilities to detect malware using existing tools," in *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA*, IEEE Computer Society, Mar. 2018, pp. 1277–1284. doi: 10.1109/AICCSA.2017.24.

[71] P. Kaushik and P. K. Yadav, "A Novel Approach for Detecting Malware in Android Applications Using Deep Learning," *2018 11th Int. Conf. Contemp. Comput. IC3 2018*, pp. 1–4, 2018, doi: 10.1109/IC3.2018.8530668.

[72] E. C. Bayazit, O. K. Sahingoz, and B. Dogan, "A Deep Learning Based Android Malware Detection System with Static Analysis," *HORA 2022 - 4th Int. Congr. Human-Computer Interact. Optim. Robot. Appl. Proc.*, pp. 1–6, 2022, doi: 10.1109/HORA55278.2022.9800057.

[73] J. Pavithra and S. Selvakumara Samy, "A Comparative Study on Detection of Malware and Benign on the Internet Using Machine Learning Classifiers," *Math. Probl. Eng.*, vol. 2022, 2022, doi: 10.1155/2022/4893390.

[74] F. Ullah, A. Alsirhani, M. M. Alshahrani, A. Alomari, H. Naeem, and S. A. Shah, "Explainable Malware Detection System Using Transformers-Based Transfer Learning and Multi-Model Visual Representation," *Sensors*, vol. 22, no. 18, Sep. 2022, doi: 10.3390/s22186766.

[75] J. wook Jang, J. Yun, A. Mohaisen, J. Woo, and H. K. Kim, "Detecting and classifying method based on similarity matching of Android malware behavior with profile," *Springerplus*, vol. 5, no. 1, Dec. 2016, doi: 10.1186/s40064-016-1861-x.

[76] W. Mckinney, "Data structures for statistical computing in Python McKinney () Statistical Data Structures in Python," 2010.

[77] "GitHub - fchollet/deep-learning-models: Keras code and weights files for popular deep learning models." https://github.com/fchollet/deep-learning-models (accessed Aug. 16, 2023).

[78] F. Pedregosa FABIANPEDREGOSA *et al.*, "Scikit-learn: Machine Learning in Python," *J. Mach. Learn. Res.*, vol. 12, no. 85, pp. 2825–2830, 2011, Accessed: Aug. 16, 2023. [Online]. Available: http://jmlr.org/papers/v12/pedregosa11a.html

[79] J. D. Hunter, "Matplotlib," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90–95, May 2007, doi: 10.1109/MCSE.2007.55.

[80] T. Kluyver *et al.*, "Jupyter Notebooks – a publishing format for reproducible computational workflows," *Position. Power Acad. Publ. Play. Agents Agendas - Proc. 20th Int. Conf. Electron. Publ. ELPUB 2016*, pp. 87–90, 2016, doi: 10.3233/978-1-61499-649-1-87.

[81] H. J. P. Weerts, A. C. Müller, and J. Vanschoren, "Importance of Tuning Hyperparameters of Machine Learning Algorithms," 2018.

Saima Akbar MS Thesis Plagiarism Report

ORIGINALITY REPORT

13%	10%	7%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Iqra Batool, Tamim Ahmed Khan. "Software fault prediction using data mining, machine learning and deep learning techniques: A systematic literature review", Computers and Electrical Engineering, 2022 Publication	1%
2	ebin.pub Internet Source	1%
3	link.springer.com Internet Source	<1%
4	Submitted to Rochester Institute of Technology Student Paper	<1%
5	www.researchgate.net Internet Source	<1%
6	www.mdpi.com Internet Source	<1%
7	Ikram Ul Haq, Tamim Ahmed Khan, Adnan Akhunzada, Xuan Liu. "MalDroid: Secure DL–	<1%

enabled intelligent malware detection framework", IET Communications, 2021
Publication

8	Submitted to Nottingham Trent University Student Paper	<1%
9	www.medrxiv.org Internet Source	<1%
10	iq.opengenus.org Internet Source	<1%
11	pandas.pydata.org Internet Source	<1%
12	Iram Bibi, Adnan Akhunzada, Jahanzaib Malik, Muhammad Khurram Khan, Muhammad Dawood. "Secure Distributed Mobile Volunteer Computing with Android", ACM Transactions on Internet Technology, 2022 Publication	<1%
13	ozenero.com Internet Source	<1%
14	doi.org Internet Source	<1%
15	ijirset.com Internet Source	<1%
16	etd.astu.edu.et Internet Source	<1%