IEEE Access on Machine Learning,Issue Date:04.February.2022

Malware Detection: A Framework for Reverse Engineered Android Applications through Machine Learning Algorithms

Beenish Urooj[1], Munam Ali Shah[2], Carsten Maple[3], Muhammad Kamran Abbasi[4], Sidra Riasat[5]

[1] Department of Computer Science COMSATS University, Islamabad, Pakistan (e-mail: ubee.arooj@gmail.com)

[2] Department of Computer Science COMSATS University, Islamabad, Pakistan (e-mail: mshah@comsats.edu.pk)

[3]WMG, University of Warwick, Coventry, UK. CV4 7AL (e-mail: cm@warwick.ac.uk)

[4]Institute for Research in Applicable Computing University of Bedfordshire (email: abbasikamran@usindh.edu.pk)

[5]Department of Computer Science COMSATS University, Islamabad, Pakistan (e-mail: sidrariasat4@gmail.com)

Corresponding Author: Carsten Maple (email: cm@warwick.ac.uk)

ABSTRACT Today, Android is one of the most used operating systems in smartphone technology. This is the main reason, Android has become the favorite target for hackers and attackers. Malicious codes are being embedded in Android applications in such a sophisticated manner that detecting and identifying an application as a malware has become the toughest job for security providers. In terms of ingenuity and cognition, Android malware has progressed to the point where they're more impervious to conventional detection techniques. Approaches based on machine learning have emerged as a much more effective way to tackle the intricacy and originality of developing Android threats. They function by first identifying current patterns of malware activity and then using this information to distinguish between identified threats and unidentified threats with unknown behavior. This research paper uses Reverse Engineered Android applications' features and Machine Learning algorithms to find vulnerabilities present in Smartphone applications. Our contribution is twofold. Firstly, we propose a model that incorporates more innovative static feature sets with the largest current datasets of malware samples than conventional methods. Secondly, we have used ensemble learning with machine learning algorithms such as AdaBoost, SVM, etc. to improve our model's performance. Our experimental results and findings exhibit 96.24% accuracy to detect extracted malware from Android applications, with a 0.3 False Positive Rate (FPR). The proposed model incorporates ignored detrimental features such as permissions, intents, API calls, and so on, trained by feeding a solitary arbitrary feature, extracted by reverse engineering as an input to the machine.

INDEX TERMS Android applications, Benign, Feature extraction, Malware Detection, Reverse Engineering, Machine Learning.

I. INTRODUCTION

To this degree, it is guaranteed that mobile devices are an integral part of most people's daily lives. Furthermore, Android now controls the vast majority of mobile devices, with Android devices accounting for an average of 80% of the global market share over the past years. With the ongoing plan of Android to a growing range of smartphones and consumers around the world, malware targeting Android devices has increased as well. Since it is an open-source operating system, the level of danger it poses, with malware authors and programmers implementing unwanted permissions, features and application components in Android apps. The option to expand its capabilities with third-party software is also appealing, but this capability comes with the risk of malicious device attacks. When the number of smartphone apps increases, so does the security problem with unnecessary access to different personal resources. As a result, the applications are becoming more insecure, and they are stealing personal information, SMS frauds, ransomware, etc.

In contrast to static analysis methods such as a manual assessment of AndroidManifest.xml, source files and Dalvik Byte Code and the complex analysis of a managed environment to study the way it treats a program, Machine Learning includes learning the fundamental rules and habits of the positive and malicious settings of apps and then dataenabling. The static attributes derived from an application are extensively used in machine learning methodologies and the tedious task of this can be relieved if the static features of reverse-engineered Android Applications are extracted and use machine learning SVM algorithm, logistic progression, ensemble learning and other algorithms to help train the model for prediction of these malware applications [1].

Machine learning employs a range of methodologies for data classification. SVM (Support Vector Machine) is a strong learner that plots each data item as a point in n-dimensional space (where n denotes the number of features you have), with the value of each feature becoming the vector value. Then it executes classification by locating the hyper-plane that best distinguishes the two groups, leading to an improvement identification property for any two parameters. Conversely, boosting or ensemble techniques like Adaboost are assigned higher weights to rectify the behavior of misclassified variables in conjunction with other machine algorithms. When combined alongside weak classifiers, our preliminary model benefits from deploying such models since they have a high degree of precision or classification. [2], [3], [4], supports classifiers in their system models to find the highest accuracy. Although using ensemble or strong classifiers can cause problems like multicollinearity, which in a regression model, occurs when two or more independent variables are strongly associated with one another. In multivariate regression, this indicates that one regression analysis may be forecasted from another independent variable. This scope of the study can be presented as a detection journal analysis itself and can present several experimentations and results based on machine learning models [5], [6].

When an app has access to a resource in the most recent versions of Android OS, it must ask the OS for approval, and the OS will ask the user if they wish to grant or refuse the request via a pop-up menu. Many reports have been performed on the success of this resource management approach. The studies showed consumers made decisions by giving all requested access to the applications to their privileges requests [7]. In contrast to this, over 70% of Android mobile applications seek extra access that is not needed. They also sought a permit that is not needed for the app to run. A chess game that asks for photographs or requests for SMS and phone call permits, or loads unwanted packages are an example of an extra requested authorization. So, trying to assess an app's vindictiveness and not understanding the app is a tough challenge. As a result, successful malicious app monitoring will provide extra information to customers to assist them and defend them from information disclosure [8]. Figure 1 elaborates the android risk framework through the Google Play platform, which is then manually configured by the android device developers.

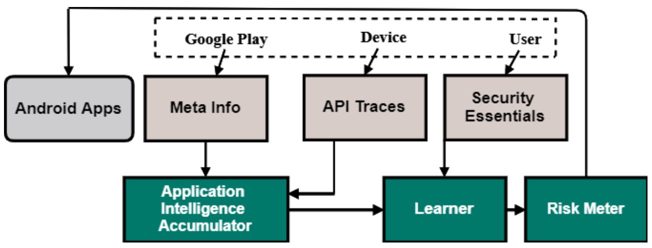


FIGURE 1. **Android Security Framework**

Contrary to other smartphone formats, such as iOS, Android requires users to access apps from untrusted outlets like file-sharing sites or third-party app stores. The malware virus problem has become so severe that 97 % of all Smartphone malware now targets Android phones. In a year, approximately 3.25 million new malware Android applications are discovered as the growth of smartphones increases. This loosely amounts to a new malware android version being introduced every few seconds [9]. The primary aim of mobile malware is to gain entrance to user data saved on the computer and user information used in confidential financial activities, such as banking. Infected file extensions, files received via Bluetooth, links to infected code in SMS, and MMS application links are all ways that mobile malware can propagate [10]. There are some strategies for locating apps that need additional features. Hopefully, by using these techniques, it would be possible to determine whether the applications that were flagged as questionable and needed additional authorization are malicious.

Static analysis methodologies are the most fundamental of all approaches. Until operating programs, the permissions and source codes are examined [11]. For many machine learning tasks, such as enhancing predictive performance or simplifying complicated learning problems, ensemble learning is regarded as the most advanced method. It enhances a single model's prediction performance by training several models and combining their predictions. Boosting, bagging, and random forest are examples of common ensemble learning techniques [12]. In summary, the main contributions of our study are as follows:

1) We present a novel subset of features for static detection of Android malware, which consists of seven additional selected feature sets that are using around 56000 features from these categories. On a collection of more than 500k benign and malicious Android applications and the highest malware sample set than any state-of-the-art approach, we assess their stability. The results obtain a detection increase in accuracy to 96.24 % with 0.3% false-positives.
2) With the additional features, we have trained six classifier models or machine learning algorithms and also implemented a Boosting ensemble learning approach (AdaBoost) with a Decision Tree based on the binary classification to enhance our prediction rate.
3) Our model is trained on the latest and large time aware samples of malware collected within recent years

including the latest Android API level than state-of-the-art approaches.

This research paper incorporates binary vector mapping for classification by allocating 0 to malicious applications and 1 for non-harmful and for predictive analysis of each application fed to the model implemented in the study. The technique eases the process by reducing fault predictive errors. Figure 2 shows the procedure for a better understanding of the concept applied later in our study. The paper passes both the categories of applications through static analysis and then is further processed for feature extraction. We presented features in 0's and 1's after extraction. Matrix displays the extraction characteristics of each application used in the dataset.

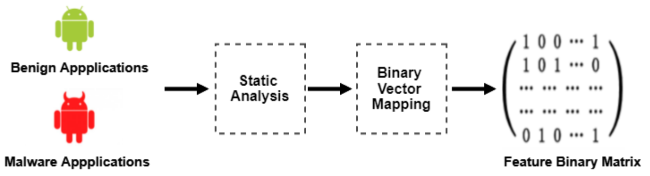


FIGURE 2. **Static Binary matrix extraction**

There are major issues to be addressed to incorporate our strategy. High measurements of the features will make it difficult to identify malware in many real-world Android applications. Certain features overlap with innocuous apps and malware [13]. In comparison, the vast number of features will cause high throughput computing. Therefore, we can learn from the features directly derived from Android apps, the most popular and significant features. The paper implements prediction models and various computer ensemble teaching strategies to boost and enhance accuracy to resolve this problem [14]. Feature selection is an essential step in all machine-based learning approaches. The optimum collection of features will not only help boost the outcomes of tests but will also help to reduce the compass of most machine-based learning algorithms [15].

Studies have extensively suggested three separate methods for identifying android malware: static, interactive meaning dynamically, and synthetic or hybrid. Static analysis techniques look at the code without ever running it, so they're a little sluggish if carried out manually and have to face a lot of false positives [16]. Data obfuscation and complex code loading are both significant pitfalls of the technique. That is why automated operation helps to achieve reliability, accuracy, and lesser time utilization [17]. Reverse engineer Android applications and extract features and do static analysis from them without having to execute them. This method entails examining the contents of two files: AndroidManifest.xml and classes.dex, and working on the file with the .apk extension. Feature selection techniques and classification algorithms are two crucial areas of feature-based types of fraudulent applications. Feature filtering methods are used to reduce the dimension size of a dataset. Any of the functions (attributes) that aren't helpful in the study are omitted from the data collection because of this. The remaining features are chosen by weighing the representational strength of all the dataset's features [18]. Parsing tools can help learn which permissions, packages or services an application offers by analyzing the AndroidManifest.xml file, such as permission android.permission.call phone, which allows an application to misuse calling abilities. The paper elaborates exactly what sort of sensitive API the authors could name by decoding the classes.dex file with the Jadx-gui disassembler [19]. In certain cases, including two permissions in a single app can signify the app's possible malicious attacks. For example, an application with RECEIVE SMS and WRITE SMS permissions can mask or interfere with receiving text messages [20] or applying sensitive API such as sendTextMessage() can also be harmful and lead to fraud and stealing.

Until we started our main idea of the project. The fact explained that Android applications pose a lot of threats to its user because of the unnecessary programs compiled inside them and explained why it is necessary to automate the process of static analysis for the efficient detection of malware applications based on the extracted features. The rest of the paper is planned as follows. Related works are examined in Section II. Section III will present the design and method of our model. Section IV elaborates the assessment findings and future threats. The experiments and results will be dilated and performed in Sections V and VI. Section VII includes our research issues, recommendations, and conclusions for the future.

II. RELATED WORKS

Linux (Android core) keeps key aspects of the security infrastructure of the operating system. The Android displays to the administrator a list of features, sought to reinstall an update. The program installs itself on the computer after they issue access. Figure 3 shows the integrated core parts of Android architecture. It comprises applications at the top layer and also includes an application framework, libraries or a Runtime layer, and a Linux kernel. These levels are further divided into their components, which make an Android Application. The Linux Kernel is the key part of Android that provides its OS functionality to phones, and the Dalvik Virtual Machine (DVM) is to manage a mobile device. Application is the Android architecture's highest layer. Native and third-party apps such as contacts, email, audio, gallery, clock, sports, and so on are located only in this layer. This framework gets the classes often used to develop Android apps. It also handles the user interface and device infrastructure and provides a common specification for hardware entry. To facilitate the development of Android, the Platform Libraries include many C/C++ core libraries and Java-based libraries such as SSL, libc, Graphics, SQLite, Webkit, Media, Surface Manager, OpenGL, and others. The taxonomy helps understand the viewer with a logical algorithmic approach for grasping the core surfaces and functionality of the operating system.

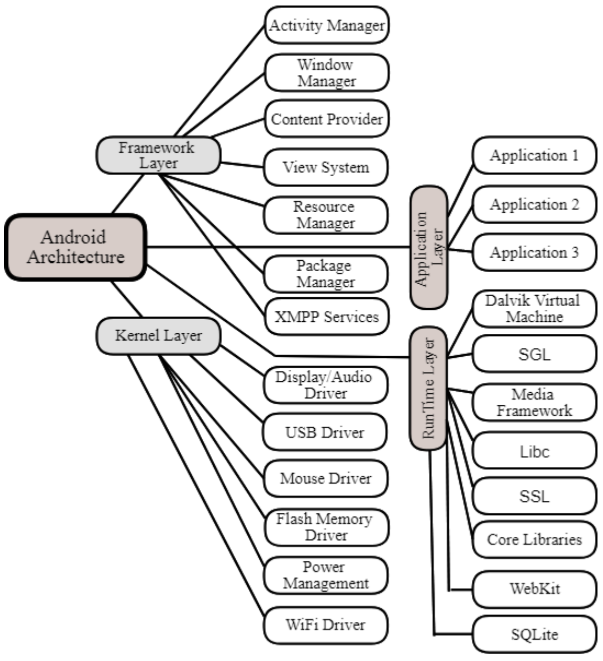


FIGURE 3. **Taxonomy of Android Architecture**

The methods proposed in this related work contribute to key aspects and a higher predictive rate for malware detection. Certain research has focused on increasing accuracy, while others have focused on providing a larger dataset, some have been implemented by employing various feature sets, and many studies have combined all of these to improve detection rate efficiency. In [21] the authors offer a system for detecting Android malware apps to aid in the organization of the Android Market. The proposed framework aims to provide a machine learning-based malware detection system for Android to detect malware apps and improve phone users' safety and privacy. This system monitors different permission-based characteristics and events acquired from Android apps and examines these features employing machine learning classifiers to determine if the program is goodware or malicious. The paper uses two datasets with collectively 700 malware samples and 160 features. Both datasets achieved approximately 91% accuracy with Random Forest (RF) Algorithm. [22] Examines 5,560 malware samples, detecting 94 % of the malware with minimal false alarms, where the reasons supplied for each detection disclose key features of the identified malware. Another technique [23] exceeds both static and dynamic methods that rely on system calls in terms of resilience. Researchers demonstrated the consistency of the model in attaining maximum classification performance and better accuracy compared to two state-of-the-art peer methods that represent both static and dynamic methodologies over for nine years through three interrelated assessments with satisfactory malware samples from different sources. Model continuously achieved 97% F1-measure accuracy for identifying applications or categorizing malware. [24] The authors present a unique Android malware detection approach dubbed Permission-based Malware Detection Systems (PMDS) based on a study of 2950 samples of benign and malicious Android applications. In PMDS, requested permissions are viewed as behavioral markers, and a machine learning model is built on those indicators to detect new potentially dangerous behavior in unknown apps depending on the mix of rights they require. PMDS identifies more than 92–94% of all heretofore unknown malware, with a false positive rate of 1.52–3.93%. The authors of this article [25] solely use the machine learning ensemble learning method Random Forest supervised classifier on Android feature malware samples with 42 features respectively. Their objective was to assess Random Forest's accuracy in identifying Android application activity as harmful or benign. Dataset 1 is built on 1330 malicious apk samples and 407 benign ones seen by the author. This is based on the collection of feature vectors for each application. Based on an ensemble learning approach, Congyi proposes a concept in [26] for recognizing and distinguishing Android malware. To begin, a static analysis of the Android Manifest file in the APK is done to extract system characteristics such as permission calls, component calls, and intents. Then, to detect malicious apps, they deploy the XGBoost technique, which is an implementation of ensemble learning. Analyzing more than 6,000 Android apps on the Kaggle platform provided the initial data for this experiment. They tested both benign and malicious apps based on 3 feature sets for a testing set of 2,000 samples and used the remaining data to create a training set of 6,315 samples. Additional approaches include [27], an SVM-based malware detection technique for the Android platform that incorporates both dangerous permission combinations and susceptible API calls as elements in the SVM algorithm. The dataset includes 400 Android applications, which included 200 benign apps from the official Android market and 200 malicious apps from the Drebin dataset. [28] Determines whether the program is dangerous and, if so, categorizes it as part of a malware family. They obtain up to 99.82 % accuracy with zero false positives for malware detection at a fraction of the computation power of state-of-the-art methods but incorporate a minimal feature set. The results of [29] demonstrate that deep learning is adequate for classifying Android malware and that it is much more successful when additional training data is available. A permission-based strategy for identifying malware in Android applications is described in [30], which uses filter feature selection algorithms to pick features and implements machine learning algorithms such as Random Forest, SVM, and J48 to classify applications as malware or benign. This research [31] provides a feature selection using the Genetic algorithm (GA) approach for identifying Android malware. For identifying and analyzing Android malware, three alternative classifier techniques with distinct feature subsets were built and compared using GA.

One of the important matters that has not been considered by any of the studies is the sustainability of the model after the advancement of applications. This issue is still a challenge for our research as well. The model's ability to classify will gradually decrease over time when new features or evolved applications are created. Only [28] and [25] specify this issue and introduce it as a drift concept, describing the low performance of their systems after some time. Our research doesn't implement this problem as well but we suggested some potential studies to initiate solutions for models sustainability in the research issues and challenges section. Another matter that could arise in the field of implementing machine learning algorithms is the "Multicollinearity Problem" which we have discussed in the introduction section. This subject arises due to the algorithms being dependent on multiple variables embedded in these machine learning or deep learning models. Although it is one of the rising issues in the area and could be present in our study it would constitute better as separate research. Our model is already incorporating a wide range of evaluations and analysis of Android applications features sets but this would be a great opportunity to further enhance the models for future use. There are relevant studies that support alleviating this challenge by detecting the model's dependencies in terms of comparing multiple models together and then calculating the greater impact of the highest given model. Authors in [32], [33], [34] consider different tales concerning different machine learning models to highlight and find out the measures for different model scenarios.

Tables 1 and 2 elaborates on the novelty of our approach and compare state-of-the-art methodologies in several categories. Table 1 focuses on the key novel categories in terms of malware samples, feature sets, the method proposed, accuracy, false-positive rate, the level of API (increased complex application behavior) and system environment for data processing. It also explains that our sample set and feature set is larger and achieve satisfactory accuracy with 0.3% FPR, depicting the lowest false positives other than DroidSieve. Our contribution lands on the upgraded API levels with large sample sizes including enhanced feature sets to detect malware. Table 2 elaborates a more in-depth approach and shows the key features present in the proposed and other approaches with also the time awareness of the data being collected.

TABLE I

Relative techniques analysis on basis of multiple factors in comparison to proposed approach (PER: Permissions, STR: String, API: Application Programming Interface, INT: Intents, PKG: Package, APP-C: App Components, SR: Services, RS: Receivers)

Year	Method	Models Trained	Feature Set	API Level	# Malware	FP%	Acc%	Environment
2013	PDMS	3	PER	19	700	-	91.75%	-
2014	Drebin	1	PER,STR,API,INT	20 , 21	5,560	1.0%	-	Core 2 Duo, 4G RAM
2015	RevealDroid	3	PER,API, INT,PKG	22, 23	9,054	18.7%	95.2	8-Core, 64G RAM
2016	DroidDetector	Deep Learning	PER, API	24, 25	1760	-	96.76%	-
2017	DroidSieve	3	API, PER, INT	26, 27	16,141	0.0%	99%	40-Core Xeon, 378G RAM
2018	DroidCat	1	API	22, 23	16,978	-	97%	-
2018	Permission based malware detection in android devices	4	PER	28	673	6%	93%	8 G in memory, i5-4300U CPU, LINUX CENTOS 7 OS
2019	Permission-based Android Malware Detection System (FS with GA)	2	PER	29	1119	-	98.45%	-
2020	Method for Detecting Android Malware Based on Ensemble Learning	2	PER, INT, APP-C	28, 29	4,011	-	95%	Intel ® core (TM) i7-8750h CPU @ 2.20GHz 2.21ghz, 16.0GB memory and Windows 10 OS.
2021	Proposed Approach	6	PER, API, PKG, INT, SR, RS, App-C	29, 30	18,578	0.3%	96.24%	Intel ®, Core i5, 2.5 GHz, Windows 8, 4GB Ram

TABLE II
Relative techniques analysis on basis of features and sample collected in comparison to proposed approach

Year	Sample Collected	Method	Function		Static Features Extracted							
			Clas	Det	Permissions	Intent	M-Tags	Packages	API Calls	Receivers	Services	APP-C
2013	2012	PDMS	×	✓	✓	×	×	×	×	×	×	×
2014	-	Drebin	×	✓	✓	✓	×	×	✓	×	×	×
2015	2013-2014	RevealDroid	✓	✓	✓	✓	×	✓	✓	×	×	×
2016	2016	DroidDetector	✓	✓	✓	×	×	×	✓	×	×	×
2017	Past nine years	DroidSieve	✓	✓	✓	✓	×	×	✓	×	×	×
2018	2017	Permission based malware detection in android devices	×	✓	✓	×	×	×	×	×	×	×
2019	2018	Permission-based Android Malware Detection (FS with GA)	×	✓	✓	×	×	×	×	×	×	×
2020	2018	Detecting Android Malware Based on Ensemble Learning	×	✓	✓	✓	×	×	×	×	×	✓
2021	2017-2021	Proposed Approach	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

A. *Reverse Engineered Applications characteristics*

As for Android apps, various apps have various functionalities. If the app is to use the device tools, you must specify the corresponding allowances in the Android Manifest format. Different program forms, therefore, have different declarations of prior approval *[35] [36]*. System static analysis also identifies an application as malicious or benevolent. In classification, they make rational choices using features. The article shows the taxonomy diagram for the features present in Android applications *[37]*. It comprises all the components present in the APK files and how they are when they are reverse engineered by using a disassembler, in our case Jadx-gui. Fig.4 represents the process of apk file disassembly.

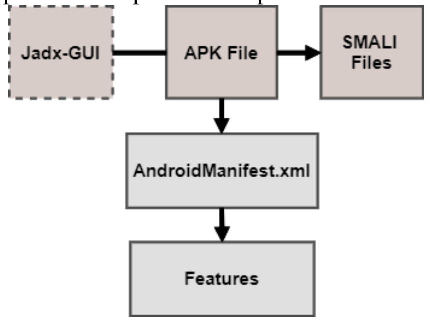


FIGURE 4. **Reverse Engineering APK files architecture**

1) ANDROIDMANIFEST.XML

In the root folder of any reverse-engineered application, there must be an android Manifest.xml file. The Manifest file gives essential information to the Mobile application, which is required by the framework before executing any code for the app. The authorization process should protect the application's key elements, which include the Operation, Service, Content Provider, and Broadcast Receivers. These results mainly accomplished by affiliating these components with the relevant element in its manifest definition and making Android dynamically implement the features in the closely associated contexts [27].

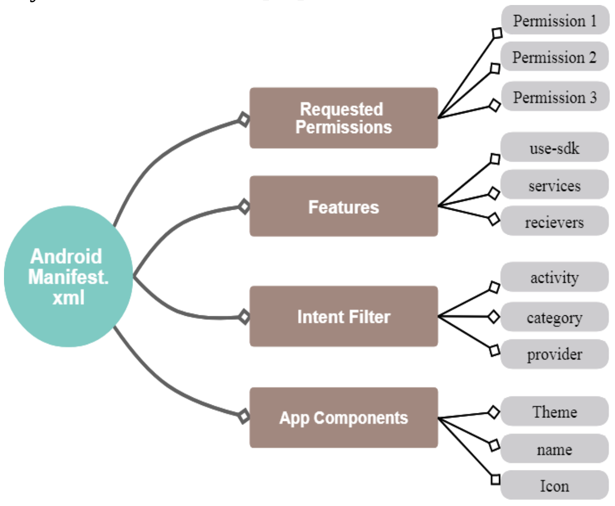


FIGURE 5. **Taxonomy of Android Manifest**

Fig. 5 shows the taxonomy of the Android manifest components, which contain all the requested permissions, packages, intents and features for extraction.

B. FEATURE SET EXTRACTION

Using feature filtering decreases the dimensions of data collection by deleting functions that are not useful for study. We chose the characteristics based on their capability to display all data sets. Enhanced efficiency by reducing the dataset size and the hours wasted on the classification process introduces an effective function selection process. Our process does not support a revamped Android emulator, because it's not a convenient approach and we preferred our system for physical devices in the future. Jadx carries out the modification and evaluation of source code. The system concentrated on trying to hook the byte-level API calls [38]. For our dataset, features from over 1, 00,000 applications are extracted containing around 56000 extracted features. Functions and processes of opcode API features are removed from the disassembled *Smali* and *Manifest* files of an APK file. The *Smali* file, segmented by the process and the opcode frequency of Dalvik for every method, is determined by scanning Dalvik Bytecodes. To verify invocation of a hazardous API in that form, it is also possible to determine the hazardous frequency of an API invocation for each method during the byte code search. For string functions, strings are selected without the method of isolation from the entire *Smali* archives [39].

We will never have a predictable response when the number of features inside a dataset exceeds the number of occurrences. In other terms, when we don't have enough data to train our machine on, generating a structure that could capture the association between both the predictive variables and responses variable appears problematic.

The system used in this study also incorporates larger feature sets for classification. Although this problem arises in machine learning quite often to some extent choosing the type of model for detection or classification can highly impact the high dimensionality of the data being used. Support vector machine and AdaBoost can handle relatively well than other algorithms because of their high dimensional space/hyperplane sectioning. Another suspension for our datasets was the tool used for extracting the given datasets. Androguard implements parsing and analyzing automation to further break down components of application apk's after decompiling and encourages weighting of the data into binary, making it easy to use relevant data for classification. It uses certain functionality to get useful features from manifest files of these Android applications reducing the acquiring irrelevant features. Although the data in this study works significantly well for evaluation, however, the datasets will be needed to upgrade in terms of forthcoming evolving measures.

Certain other authors have presented many tools and proposals to deal with high dimensional data such as [40], [41], inducing multiple methods such as filtering wrapping to enhance robustness.

The feature set of our model includes:

$F_1 \rightarrow$ *Permissions*
$F_2 \rightarrow$ *API Calls*
$F_3 \rightarrow$ *Intents*
$F_4 \rightarrow$ *App Components*
$F_5 \rightarrow$ *Packages*
$F_6 \rightarrow$ *Services*
$F_7 \rightarrow$ *Receivers*

2) PERMISSIONS

Permission is a security feature that limits access to certain information on smartphone, with the role of preserving sensitive data and functions that might be exploited to harm the user's experience. A unique label is assigned for every permit, which typically denotes a limited operation. The permissions are further categorized into four parts by Google: normal, dangerous, signature, and SignatureOrSystem. For evaluating Android permissions, researchers take a variety of methods [42]. Standard (also called secure) levels of security permissions, such as VIBRATE and SET WALLPAPER, are permissions without risk. Android kit installer will not allow the user to approve these permissions. The dangerous security standard will pose warnings to the user before implementation and will require the user's consent. The signature and symbol Security stages of SignatureOrSystem cover the most risky permits. Only applications with the same certificate, as the certificate used to sign the request declaring approval, are allowed to sign signature permissions [43]. It also acts as a buffer in the middle of hardware and the rest of the stack. A variety of different C/C++ core libraries, such as libc and SSL, are being used in libraries. Dalvik virtual machines and key libraries are part of the Android Run Time. App Model defines classes for developing Android applications, as well as a standardized structure for hardware control and the management of user experience and app property. API libraries are used for both proprietary and third-party users [44]. Table 3 shows some dangerous permissions that pose problems to the reverse engineered applications.

TABLE III
Dangerous Permissions (Malware Probability)

DANGEROUS PERMISSIONS			
SMS	SEND_SMS	STORAGE	READ_EXTERNALSTORAGE
	RECEIVE_SMS		WRITE_EXTERNALSTORAGE
	READ_SMS	SENSORS	BODY_SENSORS
	RECEIVE_WAPPUSH	CALENDAR	READ_CALENDAR
MICROPHONE	RECORD_AUDIO		WRITE_CALENDAR
CONTACTS	READ_CONTACTS	LOCATION	ACCESS_FINELOCATION
	WRITE_CONTACTS		ACCESS_COARSELOCATION
	GET_ACCOUNTS	PHONE	READ_PHONESTATE
CALLLOG	RECEIVE_MMS		READ_PHONENUMBERS
	READ_CALLLOG		CALL_PHONE
	WRITE_CALLLOG		USES_IP
	PROCESS_OUTGOINGCALLS		READ_PHONESTATE

3) INTENTS

The message delivered among modules such as activities, content providers, broadcast receivers, and services is known as Android Intent. It's commonly used alongside the startActivity() function to start activities, broadcast receivers, and other things. Individual intent counts are exploited as a continuous feature in categorization. To provide more specificity, we divide the list of intents into further sections, such as intentions including the phrases (android.net), which are linked to the network manager, intents including (com.android.vending), for billing transactions, and intents addressing framework components (com.android) and proving to be harmful elements in these apps.

4) API CALLS

Safe APIs are tools that are only available by the operating system. GPS, camera, SMS, Bluetooth, and network or data are some examples. To make use of such resources, the application must identify them in its manifest [45]. The Cost-sensitive APIs are those that can increase cost through their usages, such as SMS, data or network, and NFC. Each version includes these APIs in the OS-controlled set of protected APIs that require the device's user's sole permission. API calls that grant sensitive information or device resources are commonly detected in malicious codes. These calls are isolated and compiled in a different feature set so they might contribute to harmful activity. Table 4 elaborates dangerous API features:

TABLE IV

Sensitive APIs	
getDeviceId()	execHttpRequest()
getSubscriberId()	sendTextMessage()
setWifiEnabled()	Runtime.exec()

5) API COMPONENTS

The program that requires access or activity e.g. a path from point A to point B on a route predicated on a user's location from another application makes a call to its API, stating the data/functionality demands. The other software includes the data/functionality that the first program requested. For privacy reasons, some API features must be declared and not used in these apps. These components relate to broadcast features present in these applications.

6) PACKAGES, SERVICES AND RECEIVERS

The package manifest has always been found in the package's root and provides information about the package, such as its registered name and sequence number. It also specifies crucial data to convey to the user, such as a consumer name for the program that displays in the User Interface (UI). The file format is in *.json* for packages.
According to a publication process model, Android apps can transmit and receive messages from the Android system and other Android apps. When a noteworthy event occurs, these broadcasts are sent out. The Android system, for example, sends broadcasts when different system events occur, such as the system booting up or the smartphone charging. Individuals can sign up to receive certain broadcasts [46]. When a broadcast is sent, the system automatically directs it to applications that have signed up to receive that sort of broadcast. Services, unlike activities, do not have a graphical user interface. They're used to build long-running background processes or a complex communications API that other programs may access. In the manifest file, all services are represented by <*service*> elements and they allow the developer to invalidate the structure of the application.

C. Classification

The collection of chosen features in the signature database, separated into training and test data, and is used to recognize android malware apps by traditional machine learning techniques [47]. There are three different computer frameworks: supervised learning, unsupervised learning, and reinforcement learning. The supervised learning method is the focus of this paper, comprises algorithms that learn a model from externally provided instances of known data and known results to produce a theoretical model so that the learned model predicts feedback about previous occurrences over new data [48]. The deployment of ensemble techniques and strong learning classifiers helps classification of our binary feature sets, resulting in correctly categorized malware and benign samples. We believe that these classification mechanics produces efficient outputs because of their sorting nature. Fig. 6 explains the learning model process.

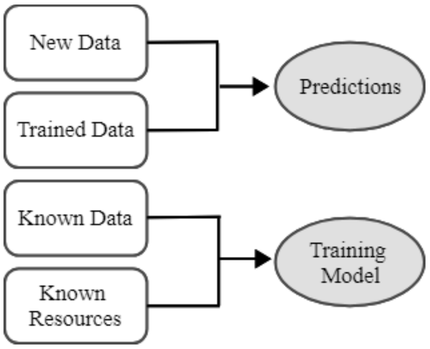


FIGURE 6. **Machine learning Process**

A comparative algorithm selection for our model based on AdaBoost, Naive Bayes, Decision Tree classifier, K-Neighbor, Gaussian NB, Random forest classifier, and Support Vector Machine performing a relative review which will give an accurate analysis of the algorithm for the prediction of our model.

1) ALGORITHM CHARACTERISTICS APPRAISAL

The assessment of suggested algorithms was carried out using Python. The use of FPR and Accuracy assess our comparative algorithms trials [49]. These estimates, derived from the following basic factors, are listed further down:

- **Accuracy:** Accuracy is one criterion being used to evaluate classification techniques. TP refers to the number of malicious apps which were misclassified as malicious, and FN identifies the number of safe applications which were misidentified as malicious. The number TN measures the truly benign applications and FN denotes the number of irregular apps that were wrongly labelled as normal [50].
- **False Positive Rate:** Determines the measuring factor of a model's ability to identify correct apps or the model's ability to generate FP.

$$(Acc)_{m,b} = \frac{(TP)_{m,b} + (TN)_{m,b}}{All\ samples} \qquad [1]$$

$$(FPR)_{m,b} = \frac{(FP)_{m,b}}{(TP)_{m,b} + (FP)_{m,b}} \qquad [2]$$

Equations (1) and (2) demonstrate the accuracy of the false detection rate measuring formula applied to calculate the Detection Rate (DR) and precision. Accuracy of the classification dataset, which contains both benevolent and malicious applications, our models define a hyperplane that divides both categories with the largest probability. One class is synonymous with ransomware and the other with friendly applications [51]. The authors then assumed the research data to be unknown applications, which are classified by projecting them to subspace to determine if they are on the malicious or friendly side of the hyperplane [52]. Then, using our model will correlate all the regression findings to their original reports to assess the proposed model's malware identification accuracy [53]. Static features make for a pleasing accuracy and precision of more than 90%. What's more noteworthy is that defining the usage of API calls in a single part of the Android platform allows for the creation of the most representative function space or the resources where malicious and benign can be distinguished more easily [54], [55]. If the amount of the classification target is greater than the probability estimates, the classification target of the testing data is then calculated as that label [56]. The objects are Blue or Red; the dividing lines identify the border, so an object on the right side is called blue, meaning benign, a general scenario and likewise. This is an example of linear classification, but not all classifications are this basic, and functional groups are needed to differentiate between groups [57], [58].

III. PROPOSED METHODOLOGY

The major goal of our research is to determine which criteria are most helpful in detecting malware in cell phones, particularly those running Android. We have taken up the task to train up to six machine learning algorithms such as AdaBoost, Support Vector Machine, Decision Tree, KNN, Navies Bayes and Random Forest techniques and classify these machine learning algorithms accurately. The methodology section is elaborated in two sections; Pre-Processing (explaining the pre-requisite processing), Proposed Model (Model functionalities and components).

D. Pre-Processing

APK files from numerous apps were included in the resulting datasets (containing malware and benign characteristics). A *Jadx-Gui* decompiler is then used to reverse engineer the apk files to extract features from the Android manifest file's feature set for further processing. These stages are regarded as pre-processes from before real assessments and are essential parts before any kind of testing and training using any predictive models.

Androguard, an open-source tool that extracts prioritised features from files and converts them into binary values, is used to extract features. For labelling the false or accurate android application, we employ binary search techniques, i.e. 1 or 0 for benign and 1 or 0 for malware. Figure 7 shows our technique's pre-processing framework and flow structures, which must be accomplished before the classifiers are tested.

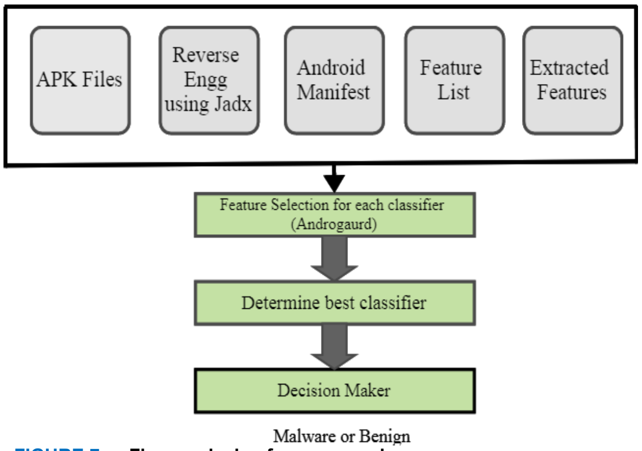


FIGURE 7. Flow analysis of our research

The operations embedded in the rectangle are to be determined beforehand, ensuring efficient data collection. The main role in this is by the decompiler and extractor which improves and eases the model's data classification efficiency for detection of malware applications. Although our study discusses the challenge of multi-collinearity and the use of high dimensional data being implemented, we have discussed the better output for high-dimensional data in our feature extracted section but the issue of collinearity still stands and can be done as a novel contribution as future work.

Succeeding the extraction process and the use of efficient datasets accommodating useful features, the testing and training are administered. For our model, a comparative approach will be adopted based on Naive Bayes, Decision Tree classifier, K-Neighbour, Gaussian NB, Random Forest classifier, Support Vector Machine and AdaBoost. The comparison evaluation will provide an accurate assessment of the algorithm used to forecast our model. The installation package is a ZIP-compressed bundle of files that includes the manifest file (AndroidManifest.xml) and classes.dex. The manifest file describes an Android application, namely the activities, services, broadcast receivers, and content providers that make up the system. The methodology and the classification are explained before in the related work section. The next section describes the model functionality.

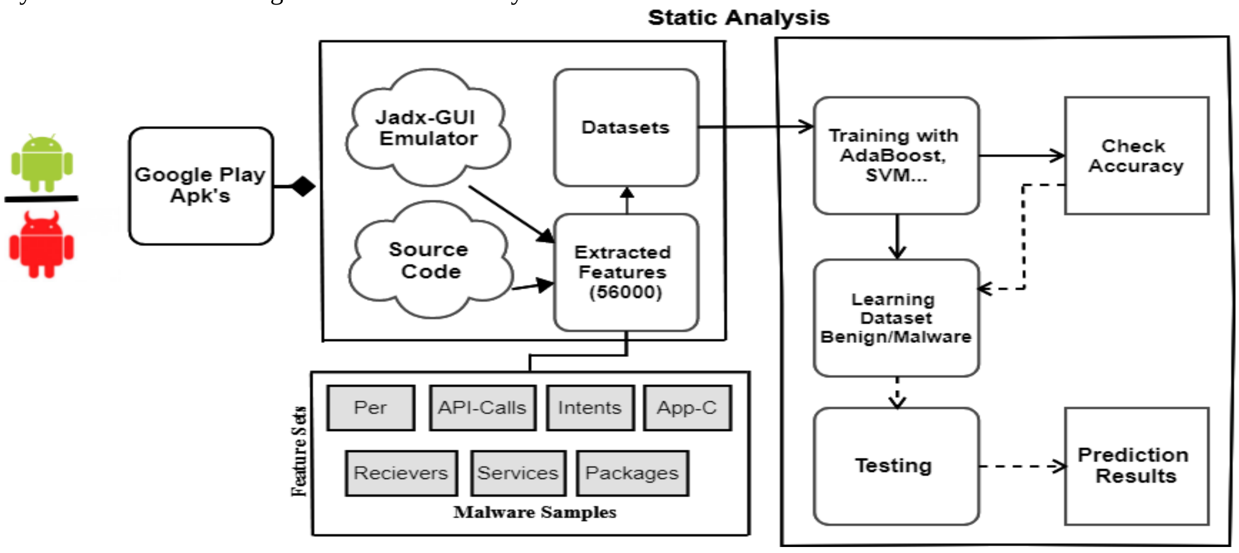


FIGURE 8. Proposed Methodology of our system

E. Proposed Model

The model gathers information from many Android applications (Google Play). These reverse-engineered (decompiled through *Jadx-Gui*) apps are then subjected to static analysis to extract features. Our suggested approach in figure 8, for the training phase, uses the retrieved characteristics to create vector mapping parsed through *Androguard*. The contribution is indicated by the proposed feature section that encompasses nearly 56,000 extracted features from the feature set seen in figure 8. Those collected features are then composed in a form of a dataset *.csv* file, stating the benign and malware properties in 1 or 0. After we generate the datasets, the features are ready for classification by predictive models. We adopted Python to create a machine algorithm classification performance program after collecting the dataset, and then we'll employ the best accurate algorithms to train our models for malware and benign detection. The system's approach and its operation are detailed in figure 8, which depicts the whole methodology of our model and algorithm learning phase with the training model processing for detection.

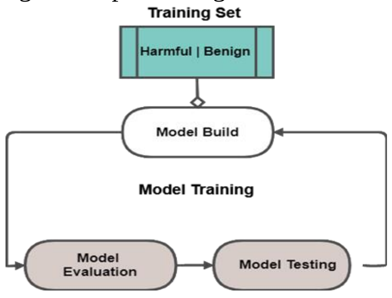


FIGURE 9. Training Model Processing

Figure 9 shows us the training cycle of the program and how the model first is constructed and then evaluated. Then further on the data is cycled towards testing and that is the data fed to the trained model for further prediction analysis of the android applications.

The future threats and predictions pointed out in the next section state insecure android applications which contain unnecessary permissions, and opt for an easy way for an attacker to steal private data or launch major attacks, and later on, present the methodology of our research.

IV. FUTURE THREATS AND PREDICTION

By 2020, mobile applications will be installed onto consumer devices over 205 billion times. Statistics by Marketing Land suggest that 57 percent of the overall digital content time is spent on mobile devices. Our daily activities always depend on social networking, bank transfers, business operations, and mobile managed services applications. Accommodating over two billion individuals, almost 40% of the world's total population, Juniper Sources point to the number of those using mobile banking services.

Developers devote close attention to the development of software to provide us a comfortable and seamless experience and when someone enthusiastically installs these mobile applications requiring personal information, the user stops thinking about the security consequences. This is the reason people don't even look closely at the permissions or the feature updates being asked by the applications [59]. They simply download the application they want and, when asked for installation, they overlook everything else and start using the app. Most of these applications never even ask the consent of the consumer and the hackers are using their information without their knowledge. The future threat rises, at the end of 2020 and beginning of 2021:

- 70% of Google Play Store applications require access to one more "dangerous permission and packages, up from 66.6% in Q12020, which is a 5 percent raise". 69.4% of applications for children (13 years of age) claim at least one risky permit up from 68.8% in 2020 (a 1 percent rise).
- Over 2.3 million applications altogether, over 2.1 million applications for children need at least one harmful authorization.

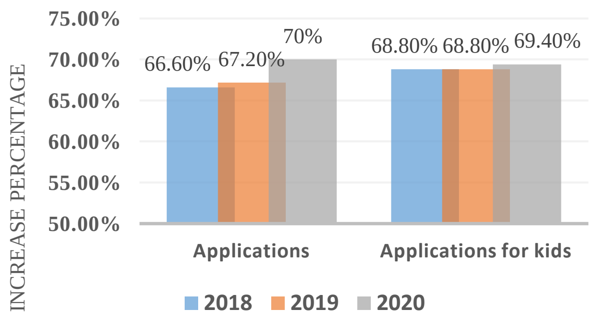


FIGURE 10. Graph of application threat increase by 5%

Figure 10 shows the percent hit in 2020, proceeding to 2021 on both the application for permission criteria. As per these statistics, the predicted rate in the coming years (till 2025) proposes that there could be a grave danger because of these unnecessary access as per each level of the Android API. Graph 1 shows the representation of both the factors, application for everyone and the other for application kids for the year 2019-2020. The graph shows the increase in 5% of the applications with dangerous malware. This takes a great deal of application security and also depicts the futuristic way that if nothing is done on time, these applications will increase up to a higher number in the future. According to multiple tech reviews, each one published in 2021, states that according to research of 2,500 top-of-the-line and rising applications, over two portions of the most popular Android applications on Google Play request excessive user permissions and access. These allow apps, among other unwanted behaviors, to launch harmful scripts and access messages unnecessarily with unwanted features inbuilt [60]. They stated that with the increase in usage of application components and features and also the release of new Android frameworks and APIs each year. It is most likely that threats are surely to increase by 15% from 5%. The average Android user has roughly 80 applications loaded, thus at least one app on the phone demands additional authorization on the phone. It is likely that excessive authorizations may jeopardize user data and privacy or even allow device hacks.

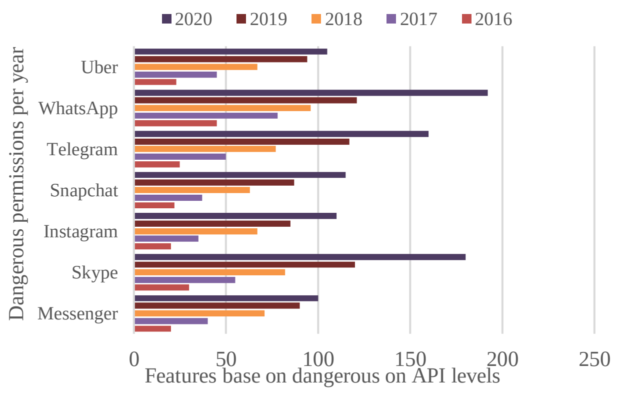


FIGURE 11. Increase in Android Malware Statistics

Figure 11 elaborates the dangerous malware increase till 2020 with every newer version of API Level. Figure 12 shows the most rising apps from 2016 to 2021 and the percentage of dangerous permissions, packages these applications gain [61]. These applications are used daily and if they are involved in unnecessary and third-party access, then there is a special need to apply countermeasures on these applications, as this is going to be a major threat in the future. Also, the Figure depicts the need to measure these threats and devise countermeasures or at least present models to provide more encoded procedures to carry out for these well-known applications. These apps provide a lot of opportunities, but with an increase in private and intellectual property stored in these apps, certain anecdotes need to be proposed.

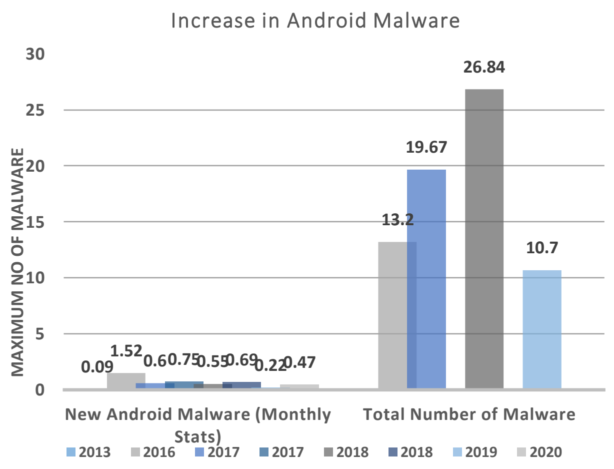


FIGURE 12. Third-party well-known dangerous apps increase from 2016 to 2020

V. Experimental Results

In this section, the results of our experimentation are stated. To start our experimentation discussion, we will elaborate on the basic criteria for performing our implementation successfully and will also briefly discuss the data collection or the dataset that we got and then further converse about the actual contribution part.

A. Experiment Setup

Our environment is based on *Windows 8.1 Pro* with Intel®, Core (MT) i5-2450 CPU @ 2.50 GHz as a processor. The installed memory (RAM) of the system is 4.00 GB with a 64-bit Operating System (OS), x-64 based processor.
For the generated dataset *Androguard 3.3.5* (latest release) is used for decompiling and feature extraction, deployed in regulated *.csv* files in binary vectors. We have installed *Python 3.8.12* (version 3.8) on our system for the implementation and execution of training and testing scripts of imported machine learning models.

B. Dataset

Three different datasets are used for our implementation, mainly apps belonging to Google Play. The static features of our first two datasets containing API calls, permissions, intents, packages, receivers and services were collected from MalDroid [62] and DefenseDroid [63] which includes around 14,000 malware samples. The model also uses a third dataset of 5000 malware samples using our own generated applications dataset. Applications in the datasets were reverse-engineered by the *Jadx-GUI* tool and the features are extracted using Androguard into binary data. All the datasets from different platforms are combined to incorporate our multiple features sets more than state-of-the-art approaches (explained in table 5) in a single training to achieve higher accuracy and classification of malware. The datasets are first trained on every algorithm for comparative classification analysis. After the accuracy of the algorithms are evaluated, the datasets is again trained and tested on the higher-performing algorithms to use as a feed, based on the features, inserted into the database and our model will then forecast the output for a given android application extracted features.

TABLE V
Sample Datasets

Datasets	Period	Source	Benign Applications	Malware Applications
MalDroid	2017-2020	UNB	1795	10,516
DefenseDroid	2021	Kaggle	1500	3062
GD	2021	GD	2421	5000

The next subsection elaborates the discussion and presentation of the programs for our machine learning algorithms.

C. MACHINE LEARNING ALGORITHM AND ENSEMBLE LEARNING

Six models have been selected to experiment with two strong classifiers (AdaBoost, SVM and Random Forest). The model executes upon KNN, NB, RBF, Decision Tree, SVM and we have also performed AdaBoost with Decision Tree by calculating the weighted error of the Decision tree based on its data points. As the input parameters are not jointly optimized, Adaboost is less prone to overfitting. Adaboost can help you to increase data performance of existing weak classifiers. After the higher weight of all the wrongly misclassified data points is rightly classified, the model can enhance model accuracy. Fig 13 shows the functioning of the boosting technique.

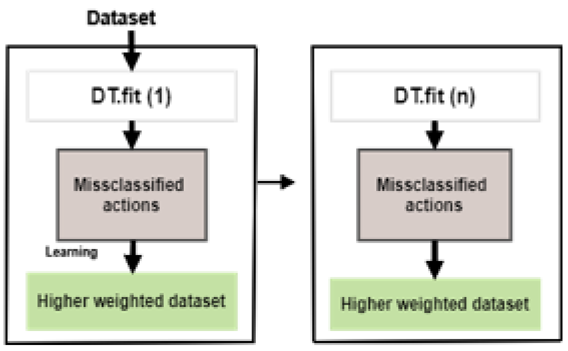


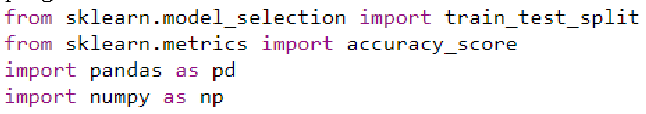
FIGURE 13. Boosting mechanism

Since there is a distinct boundary between two categories, ensemble methods and SVM perform rather well enough when dealing with clear aligned datasets following adequate extraction processes. Another significant benefit of the SVM Algorithm is that it can handle high-dimensional data, which comes in handy when it comes to its use and application in the Machine Learning sector. As seen in the diagram above, AdaBoost's greater weighted property aids our weak learner (Decision Trees) with achieving higher accuracy and wider consumption for misclassified binary feature inputs.

D. PROGRAM PARAMETERS

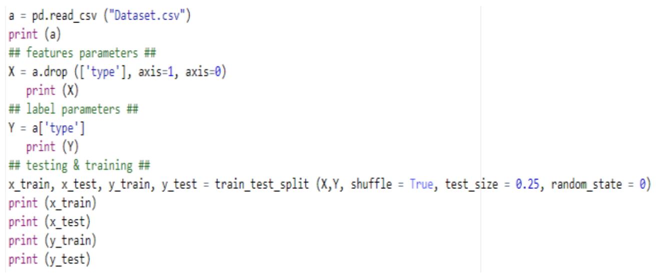
Our project is based on Python 3.9.7 and divided our execution into two programs. The first program, written to compare the algorithms for the accuracy check of respective models, based on AdaBoost, Decision Tree, KNN, SVM, Naive Bayes, and Random Forest for the comparative analysis. The program uses different import and split functions to train the models and then stores the result in a variable embedded for the testing model. The function *sklearn.model_selection*, used for accessing the bundles of algorithms, *accuracy_score* for accuracy readings, *pandas* to read the database, and *NumPy* to convert the testing model data into *rr* format.

The parameter on the x-axis is the features of the algorithms and on the y-axis is its label, meaning the accuracy percentage for these algorithms. The x and y parameters of the program are configured *to shuffle=True* using the *test_train_split* function, so each algorithm takes a random permission value from the dataset. Figures 14 and 15 show the import modules and parameters values set in our program.



```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import pandas as pd
import numpy as np
```

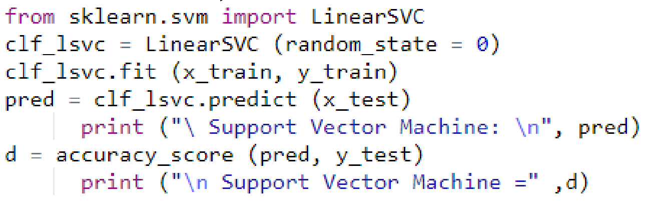
FIGURE 14. Representation of the modules of our program.



```
a = pd.read_csv ("Dataset.csv")
print (a)
## features parameters ##
X = a.drop (['type'], axis=1, axis=0)
   print (X)
## label parameters ##
Y = a['type']
   print (Y)
## testing & training ##
x_train, x_test, y_train, y_test = train_test_split (X,Y, shuffle = True, test_size = 0.25, random_state = 0)
print (x_train)
print (x_test)
print (y_train)
print (y_test)
```

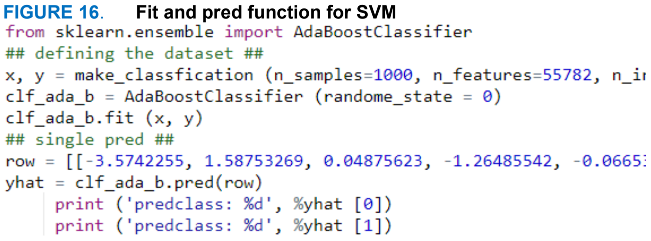
FIGURE 15. Program Parameters and split functions.

First, all the algorithms are imported into the program to implement the training data for the model, meaning the machine is training based on the given datasets. The program will work as each algorithm will take up random binary value of an app from the dataset and execute its feature's accuracy score in another variable. After training the data, the program passes the testing data to store into a predictive function. The program is designed to identify the normal and harmful permissions features through the dataset binary values (0.1) and specifies those results in function *pred ().* As you can see in the code below, the program uses a *fit ()* function, which takes the training data as arguments that are fitted using the x and y parameters into testing data for our two models (AdaBoost and SVM). All the variables were specified at the end that was given to each of our algorithms in the program to the variable *acc*. After executing the program, every algorithm will start accessing the dataset and start predicting the dataset value for the android permissions. Figures 16 and 17 represent the main key functions for our models AdaBoost and SVM, which are discussed above.



```
from sklearn.svm import LinearSVC
clf_lsvc = LinearSVC (random_state = 0)
clf_lsvc.fit (x_train, y_train)
pred = clf_lsvc.predict (x_test)
      print ("\ Support Vector Machine: \n", pred)
d = accuracy_score (pred, y_test)
      print ("\n Support Vector Machine =" ,d)
```

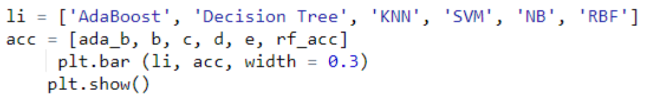
FIGURE 16. Fit and pred function for SVM



```
from sklearn.ensemble import AdaBoostClassifier
## defining the dataset ##
x, y = make_classfication (n_samples=1000, n_features=55782, n_i
clf_ada_b = AdaBoostClassifier (randome_state = 0)
clf_ada_b.fit (x, y)
## single pred ##
row = [[-3.5742255, 1.58753269, 0.04875623, -1.26485542, -0.0665
yhat = clf_ada_b.pred(row)
    print ('predclass: %d', %yhat [0])
    print ('predclass: %d', %yhat [1])
```

FIGURE 17. Predictive measures for AdaBoost

Figure 17 also explains the predictive procedure of the ensemble model with 1000 malware sample runs and given features to train for a single predictive classification output. The same *fit()* function is used for dataset training. The model is placed for higher weights of decision trees algorithm within row values and executed in *yhat.* Accuracy is then accomplished by declaring the mean and standard deviation (mean *(n_acc_scores), std (n)acc_score)))* for the binary classification output of malware. Further ahead, figure 18 shows the plotted assigned value for accuracy after the data is trained on the models.



```
li = ['AdaBoost', 'Decision Tree', 'KNN', 'SVM', 'NB', 'RBF']
acc = [ada_b, b, c, d, e, rf_acc]
     plt.bar (li, acc, width = 0.3)
    plt.show()
```

FIGURE 18. Results stored to acc variable and plotted by plt.bar function

Figure 19 shows the accuracy percentage for our models which is 96.24% and the graph displays the highest correct predictive frequency out of all the algorithms, professing the research work for greater validity. This graph is plotted by training the algorithms on the datasets to verify which algorithm can classify the application's features accurately. Program 1 is scripted to import all of the algorithms and execute them one by one on these datasets to train the algorithms, producing the most precise values after testing. In the case of AdaBoost, we trained Decision Tree first on the dataset and then used those classified values to train on the higher weights using AdaBoost. AdaBoost takes those classified samples and features used by decision trees and generates higher weights for correct results after training on those features again. *(x,y)* are the stored values by decision trees which are given as input values for AdaBoost to enhance accuracy, hence the model with the highest accuracy in fig 19. This program performs in a way that when all the models are done training, the script generates a graph using the *plt.bar* command to display the algo that classifies most applications correctly. Figure 19 and Table 6 show the accuracy and the label value that depicts the training data each algorithm randomly took and trained its model for.

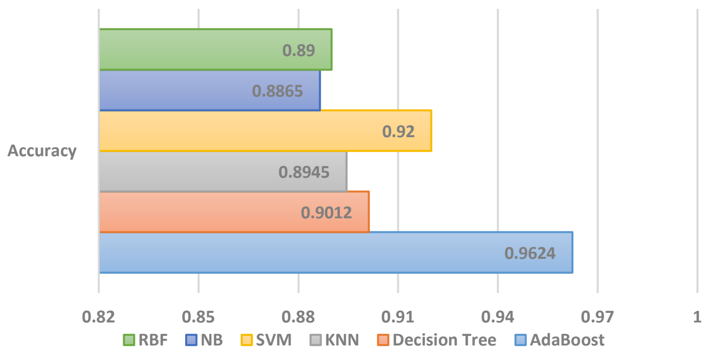


FIGURE 19. **Models accuracy percentage w.r.t label**

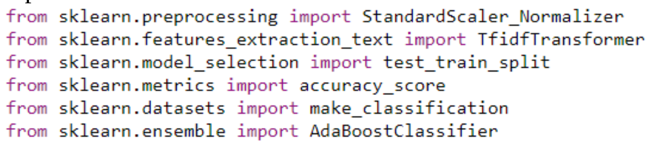
TABLE VI
Shows the label values for each algorithm and their accuracy percentage

ML Algorithm	**Testing Label Values**	**Algorithm Percentage**
AdaBoost	[01011110110000000000100111100001001 101100111100110110101111111111111 11101101101011101010000111100011111]	96.24%
Decision Tree	[11111110110111100001001111100001001 10110011110011011010111100000000111 1110110100111110101000011110001111]	90.12%
SVM	[11111101100111000010011110000010011 011001111101101101011110000000011111 110110100111110101000011110001111]	92%
KNN	[11111101101111000110011110000010010 01100011110110110101111000010011111 11011010011011010000000110100011]	89.45%
Navies Bayes	[11101110010101100001101010000001001 00111011111001001010011000010001111 01101101001101101000000011000001011]	88.65%
Random Forest	[11111110110111100001001111100001001 0011001111101101101011110000000011111 1101101001101101010000111100011111]	89%

VI. Model Precision Evaluation

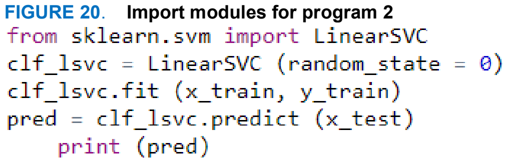
After training the datasets on algorithms and achieving accuracy percentage, individually developed another program that uses the properties of the previous code to help execute and predict the application state according to the input from the dataset. For this program, the algorithm with greater prediction capabilities is imported, i.e. AdaBoost and SVM using the function *sklearn* imports *linear_svc* and *sklearn.ensemble import AdaBoost*. The database stores input permissions *into the rr python module as a feeding factor for the* trained models and designated [1] for the benign applications and [0] for the malware application, meaning the app which uses unnecessary features. This will work in a way that, when the program executes, the algorithms will take the input from the database and then categorize the features based on what we trained the algorithm upon. So, if there are malware applications fed as an input to the database, the trained model will predict the outcome and label the state of the application.

Following the import of the trained models, the *random_state = 0* and the testing data = 0.25 for the algorithms. The import of *sklearn.preprocessing_normalize* function, which takes samples separately according to the Normalize unit. Every set of data with one component or perhaps more (each data matrix row), rescaled separately from other samples to the standard. The program also imports the function *sklearn.features_extraction.text* which transforms a text data array into a token count matrix and at the very end declares the accuracy score of these algorithms by using *sklearn.metrics* function, implementing loss, score, and utility functions to quantify performance in the categorization of the feature sets. Parameters for this program are the same as the previous program, but to fix features on every algorithm, the *x type* is dedicated to the trained models for features and *y type* for the prediction of the applications. So when the program executes it will work in the same manner and this time gives us the precision value instead of the plotted accuracy percentage of the algorithms and at last, the program will print out the *pred ()* function value which was declared to the model's testing data. Figures 20 and 21 and 22 indicate the consideration of AdaBoost and SVM prediction for features extracted for single feature input.



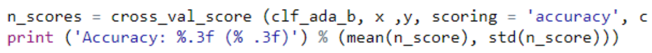
```
from sklearn.preprocessing import StandardScaler_Normalizer
from sklearn.features_extraction_text import TfidfTransformer
from sklearn.model_selection import test_train_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification
from sklearn.ensemble import AdaBoostClassifier
```

FIGURE 20. **Import modules for program 2**



```
from sklearn.svm import LinearSVC
clf_lsvc = LinearSVC (random_state = 0)
clf_lsvc.fit (x_train, y_train)
pred = clf_lsvc.predict (x_test)
    print (pred)
```

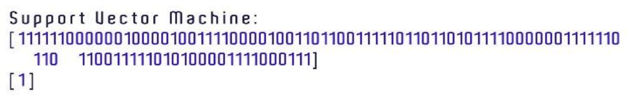
FIGURE 21. **Prediction function for SVM for testing data for the database**



```
n_scores = cross_val_score (clf_ada_b, x ,y, scoring = 'accuracy', c
print ('Accuracy: %.3f (% .3f)') % (mean(n_score), std(n_score)))
```

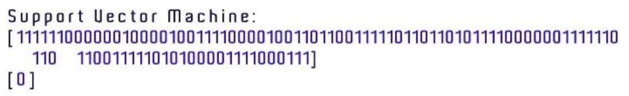
FiGURE 22. **Prediction function for AdaBoost for testing data for the database**

Further ahead, the prediction results of the program are discussed. As the code executes, the models will take the features from the dataset that was provided for a single application. The result displayed in Figure 23 shows that it's a benign application. When permission features, again fed as input the Figure 23 shows that it is a malware application based on the features the highly trained models draw out. In the same manner, the database is fed with feature binary values and the model will predict the result in 1 or 0. Figures 16 and 17 elaborate on the predictive function which will allow AdaBoost and SVM to predict the basis of the applications on the feeding input. Figures 23, 24, 25 and 26 are output screenshots of [1] showing benign and [0] for harmful applications with random application features for respective models.



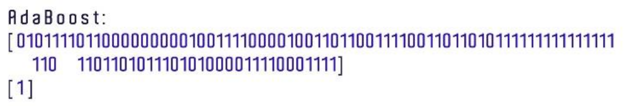
```
Support Vector Machine:
[111111000000100001001111000010011011001111101101101011110000001111110
  110   1100111110101000011111000111]
[1]
```

FIGURE 23. Output [1] representing the Benign Application (SVM).



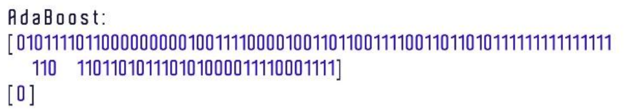
```
Support Vector Machine:
[111111000000100001001111000010011011001111101101101011110000001111110
  110   1100111110101000011111000111]
[0]
```

FIGURE 24. Output [0] representing the Malware Application (SVM).



```
AdaBoost:
[0101111011000000000010011110000100110110011110011011010111111111111111
  110   1101101011101010000111110001111]
[1]
```

FIGURE 25. Output [1] representing the Benign Application (AdaBoost)



```
AdaBoost:
[0101111011000000000010011110000100110110011110011011010111111111111111
  110   1101101011101010000111110001111]
[0]
```

FIGURE 26. Output [0] representing the Malware Application (AdaBoost

A. Results

After the forecast of our models, results show that the accuracy for our highest predictive systems is 96% and 92%. The proposed model doesn't peak in higher accuracy or predictive rate but it contributes by introducing enhanced and large feature sets (containing around 56000 newly extracted features) with the latest API level applications datasets collected in recent years than state-of-the-art approaches. Another point of view for a less predictive rate is the limitation of our sources/environment to process and generate these datasets on our models. The novelty and contributions are explained in Tables 1 and 2.

Figures 27, 28, 29 and 30 show the runs performed on the datasets on our trained model. The applications in orange indicate not harmful apps and only passes sensitive features over the line, which doesn't pose that much of a threat for the application, but it still shows the model issue for indicating true negatives for zero apps. The applications in black indicate harmful applications and the false positive rate (FPR) of this category which falls over the non-harmful apps is about 3-4 applications in case of AdaBoost and 6-7 in case of SVM in our system for 1000 runs, as shown in figures above achieved with 96% and 92% accuracy of AdaBoost and SVM.

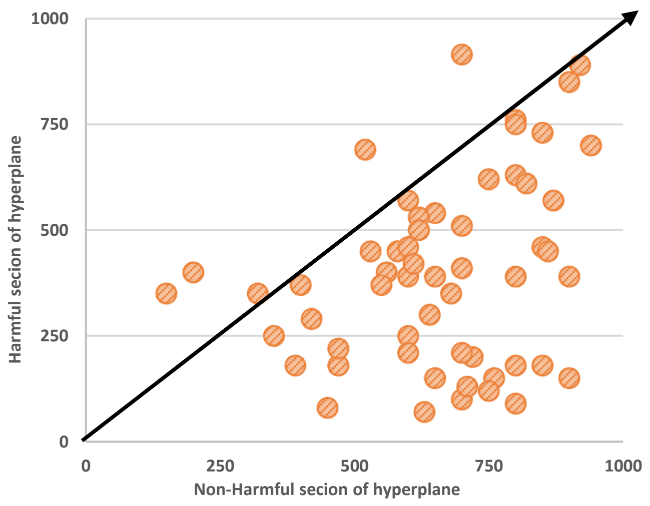


FIGURE 27. Orange entries for Non-Harmful applications in AdaBoost

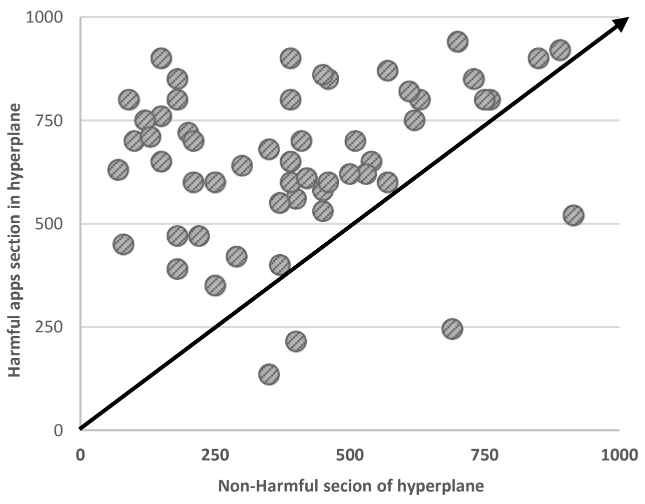


FIGURE 28. Black entries for Harmful applications in AdaBoost

All four figures are plotted in a hyperplane which describes the applications classifications in two sections i.e. Harmful and Non-harmful applications. The above line represents the harmful apps section (Black and Red) and applications lying below the line indicated non-harmful applications. The plotted hyperplanes help in understanding the prediction applications perspective as shown in fig 28 and 30 showing successful classification above the line and 3-4 apps below line indicating misclassifications. The same process is for non-harmful apps in orange colors (fig 27, 29) and the above line shows misclassifications but they don't pose serious threats.

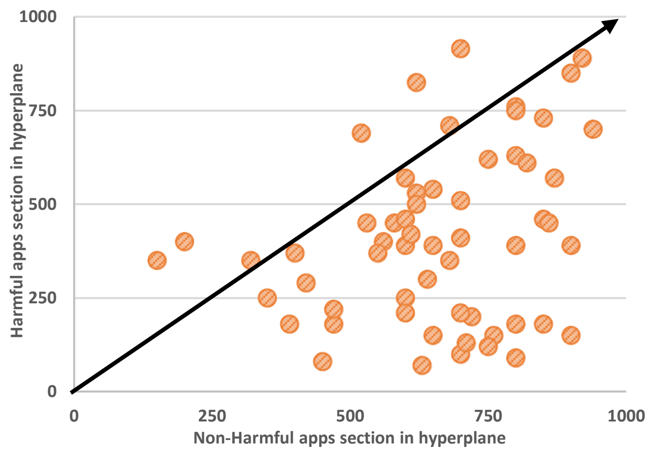


FIGURE 29. Orange entries for Non-Harmful applications in SVM

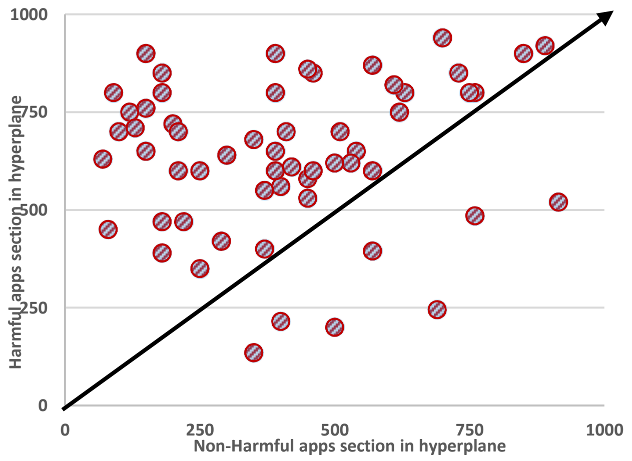


FIGURE 30. Black entries for Harmful applications in SVM

The Forthcoming is the comparative review of both malicious and benign applications of our models and experimental results with accumulative accuracy and FPR. The purpose to plot a comparative graph of malware detection is to understand the relative perspective of both our parameters. Figure 31 represents a comparative analysis of both models in terms of malicious and benign applications. Triangles in red represent the classification and detection of AdaBoost and in the square, the SVM is displayed. The graph show a malware section angle for the executive runs performed and the values above the hyperplane shows the category of Non-Harmful apps. The 0.7 misclassification rate of SVM and 0.3 of AdaBoost is plotted with malware applications falling into the true positive category. Nevertheless, the models perform with 96.24% accuracy by accurately predicting the applications categories.

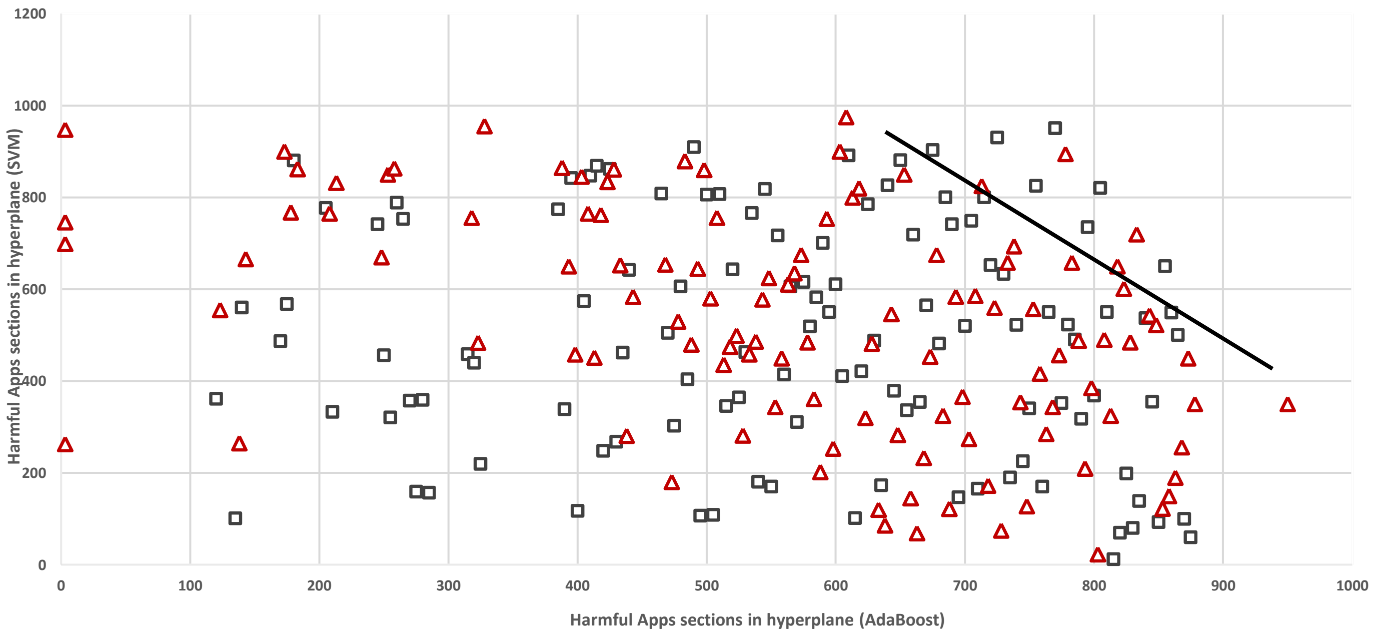


FIGURE 31. Comparative Analysis of Malicious and Benign in Adaboost and SVM

We use Accuracy and FPR as evaluation markers in this project. Precision is computed as the percentage of true harmful samples in the malware tagged by the detection system, showing the system's capacity to discriminate malware properly in the field of malware detection. False Positive Rate (FPR) is the criteria to judge the model's performance in terms of establishing how many true indications a model gives. Below are the experimental results in quantitative measures, presented in table 7, which explains the points based on accuracy, false positive rate and their predictive measures after testing on binary input for 1000 runs on our 2 higher predictive models depending on testing and training of mixed datasets containing features and malware samples. The operational speed advantage of AdaBoost is not apparent when adopting the datasets for classification and prediction. However, given AdaBoost structural features with parallel learning, we anticipate it will perform better while computing bigger data sets. We reached the same conclusion after we analyzed a much bigger data set with over 500,000 apps.

In table 7, both models are compared and trained on datasets and specify the accuracy, FPR and features used and selected corresponding to the composing samples. The FPR is also presented in figures 26 to 29 above, specifying the calculative measures through a hyperplane. The accuracy and false positives have been measured by the equation described in section IV in algorithm characteristics for the number of runs of the model. Results show 96.24% as the highest accuracy for the model after experimentation and false-positive rate of 0.3% in the case of the ensemble approach.

TABLE VII

Experimental results (AdaBoost and SVM), Selected, specify features selected in the model, MalD (MalDroid), DefenseD (DefenseDroid), GD (Generated Dataset), FPR (False Positive Rate)

Model	Features #	Datasets	Mal (S)	Acc [1]	FPR [2]
AdaBoost	55821 (Selected)	MalD + DefenseD + GD	18578	96.24%	0.301%
AdaBoost	55821 50621 + 331 (Selected)	MalD + GD	12931	95.74%	0.416%
SVM	55821 331 + 56471 (Selected)	MalD + DefenseD + GD	18578	92.04%	0.731%
SVM	331 (Selected)	GD	5877	90.1%	0.970%

Related works explain the originality of our model and exhibit the novel features and sample size. To conclude our model still lack fewer percentages in terms of accurate detection. To justify this fact, table 8 presents some properties of similar studies with higher performance rates, indicating such elements which elaborated the efficiency of our system.

[28] This model has exceptional computational/processing power with a much stronger environment to test and train their datasets. [23] Has somewhat of a similar resource with higher processing but their sample size is very limited in comparison to our model. A few other studies describe similar technical advantages, thus, leaving us to work with restrictive measures. Table 8 presents some key properties to elaborate on similar systems' components.

TABLE VIII

Relative resources (Pro, Processing)

Model	Accuracy	RAM	Pro	Samples	FPR
DroidSieve	99.82%	378GB	40 core	16,141	0%
DroidCat	97%	-	-	16,978	No FPR
PMDS (FS with GA)	98.45%	-	-	1119	No FPR
Proposed Model	96.24%	4GB	Core-i5	18,578	0.3%

VII. Research Issues and Challenges

This section highlights our experiment's prevalent and crucial topics. These hurdles are based on various stages of our work and maybe gradually rectified in the work to be undertaken in the future.

1. Features declared mostly on the device are more durable than the features specific to the applications and therefore can usually automate malware detection. The range of android parameters for processing is rather big and difficult to detect properly if someone does not extract the permissions properly.
2. There is still a fast increase in the number of apps. Malware apps can always be identified in potential in combination with methods based on AI or machine learning, such as inept learning, to make the detection more sophisticated to make it easier to identify and regulate app prediction rate.
3. Application behaviours in the malware ecosystem encourage non-emerging threats. Our study doesn't incorporate the rider analysis or behaviour of repackaged malware. The study simply uses the reverse-engineered apk files and extracts the given context to the AndroGuard and extracts features in binary vectors. Although this is a major issue and a key challenge with the advancement in Android malware. This approach will be our advanced project to perform differential or effective analysis on reverse applications, determining the effects of these applications and their results.
4. The applications with time induce new features with enhanced malware abilities which is why we would have to upgrade the system whenever the model's FPR rate after execution increases. The simplest explanation for how to identify if the model is degrading on evolved features is that our datasets are designed in binary matrix extracted from features that are currently implemented in these applications and not features that will be present in evolved apps in coming years. With new features, we would have to reverse and extract those features to form an updated dataset again to train on these classifiers. [64], [65], [66] and [67] discuss the possible solutions for this key issue and propose some possible solutions but for our model and given the resource we have only performed for current features. For future work, we will consider model sustainability and how to classify the malware that our system will be able to detect even if the features are not yet implemented.
5. The research mentions the problem of multicollinearity in the introduction, depicting the rise of dependent variables in-between machine learning algorithms which cause interpretation in results. However, this field of study can be taken as a future work for further testing of several models handling multicollinearity because our model itself is already performing high processing detection schemes to generate accuracy for Android applications features malware. We will foresee this issue and incorporate it to produce an efficient solution to the problem. Authors in [68], [69], [70] proposes some

solutions to tackle this challenge and can help understand viewers queries.

B. LIMITATIONS

The technique in this paper is based on binary classification of lightweight code of static feature sets present in the Android manifest file. The three major limitations of our method is:

1. The research doesn't include dynamic or runtime application features. We will consider the potential dynamic aspects of Android applications in the future, including real-time permissions requests and essential API requests. We will evaluate the behavioural traits of the app using a mixture of dynamic and static evaluation to discover harmful tendencies.
2. Our system lags in future sustainable operative measures, meaning the system will need to be upgraded in terms of forthcoming API levels and malware collection or terms of new innovative features present in these Android applications.
3. The constraint of a slow and low processing environment is another motive for less accuracy and predictive measures of our model in comparison to a few other peak detection techniques achieving higher accuracy.

VIII. Conclusion

In this research, we devised a framework that can detect malicious Android applications. The proposed technique takes into account various elements of machine learning and achieves a 96.24% in identifying malicious Android applications. We first define and pick functions to capture and analyze Android apps' behavior, leveraging reverse application engineering and AndroGuard to extract features into binary vectors and then use python build modules and split shuffle functions to train the model with benign and malicious datasets. Our experimental findings show that our suggested model has a false positive rate of 0.3 with 96% accuracy in the given environment with an enhanced and larger feature and sample sets. The study also discovered that when dealing with classifications and high-dimensional data, ensemble and strong learner algorithms perform comparatively better. The suggested approach is restricted in terms of static analysis, lacks sustainability concerns, and fails to address a key multicollinearity barrier. In the future, we'll consider model resilience in terms of enhanced and dynamic features. The issue of dependent variables or high intercorrelation between machine algorithms before employing them is also a promising field.

IX. References

[1] A. O. Christiana, B. A. Gyunka, and A. Noah, “Android Malware Detection through Machine Learning Techniques: A Review,” Int. J. Online Biomed. Eng. IJOE, vol. 16, no. 02, p. 14, Feb. 2020, doi: 10.3991/ijoe.v16i02.11549.
[2] D. Ghimire and J. Lee, “Geometric Feature-Based Facial Expression Recognition in Image Sequences Using Multi-Class AdaBoost and Support Vector Machines,” Sensors, vol. 13, no. 6, pp. 7714–7734, Jun. 2013, doi: 10.3390/s130607714.
[3] R. Wang, “AdaBoost for Feature Selection, Classification and Its Relation with SVM, A Review,” Phys. Procedia, vol. 25, pp. 800–807, 2012, doi: 10.1016/j.phpro.2012.03.160.
[4] J. Sun, H. Fujita, P. Chen, and H. Li, “Dynamic financial distress prediction with concept drift based on time weighting combined with Adaboost support vector machine ensemble,” Knowl.-Based Syst., vol. 120, pp. 4–14, Mar. 2017, doi: 10.1016/j.knosys.2016.12.019.
[5] A. Garg and K. Tai, “Comparison of statistical and machine learning methods in modelling of data with multicollinearity,” Int. J. Model. Identif. Control, vol. 18, no. 4, p. 295, 2013, doi: 10.1504/IJMIC.2013.053535.
[6] C. P. Obite, N. P. Olewuezi, G. U. Ugwuanyim, and D. C. Bartholomew, “Multicollinearity Effect in Regression Analysis: A Feed Forward Artificial Neural Network Approach,” Asian J. Probab. Stat., pp. 22–33, Jan. 2020, doi: 10.9734/ajpas/2020/v6i130151.
[7] W. Wang et al., “Constructing Features for Detecting Android Malicious Applications: Issues, Taxonomy and Directions,” IEEE Access, vol. 7, pp. 67602–67631, 2019, doi: 10.1109/ACCESS.2019.2918139.
[8] B. Rashidi, C. Fung, and E. Bertino, “Android malicious application detection using support vector machine and active learning,” in 2017 13th International Conference on Network and Service Management (CNSM), Tokyo, Nov. 2017, pp. 1–9. doi: 10.23919/CNSM.2017.8256035.
[9] 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. Inform., vol. 14, no. 7, pp. 3216–3225, Jul. 2018, doi: 10.1109/TII.2017.2789219.
[10] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, “Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families,” Expert Syst. Appl., vol. 41, no. 4, pp. 1104–1117, Mar. 2014, doi: 10.1016/j.eswa.2013.07.106.
[11] M. Magdum, “Permission based Mobile Malware Detection System using Machine Learning Techniques,” vol. 14, no. 6, pp. 6170–6174, 2015.
[12] M. Qiao, A. H. Sung, and Q. Liu, “Merging Permission and API Features for Android Malware Detection,” in 2016 5th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI), Kumamoto, Japan, Jul. 2016, pp. 566–571. doi: 10.1109/IIAI-AAI.2016.237.
[13] D. O. Sahin, O. E. Kural, S. Akleylek, and E. Kilic, “New results on permission based static analysis for Android malware,” in 2018 6th International Symposium on Digital Forensic and Security (ISDFS), Antalya, Mar. 2018, pp. 1–4. doi: 10.1109/ISDFS.2018.8355377.
[14] 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.
[15] X. Su, D. Zhang, W. Li, and K. Zhao, “A Deep Learning Approach to Android Malware Feature Learning and Detection,” in 2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, Aug. 2016, pp. 244–251. doi: 10.1109/TrustCom.2016.0070.
[16] K. A. Talha, D. I. Alper, and C. Aydin, “APK Auditor: Permission-based Android malware detection system,” Digit. Investig., vol. 13, pp. 1–14, Jun. 2015, doi: 10.1016/j.diin.2015.01.001.
[17] A. Mahindru and P. Singh, “Dynamic Permissions based Android Malware Detection using Machine Learning Techniques,” in Proceedings of the 10th Innovations in Software Engineering Conference, Jaipur India, Feb. 2017, pp. 202–210. doi: 10.1145/3021460.3021485.
[18] U. Pehlivan, N. Baltaci, C. Acarturk, and N. Baykal, “The analysis of feature selection methods and classification algorithms in permission based Android malware detection,” in 2014 IEEE Symposium on Computational Intelligence in Cyber Security (CICS), Orlando, FL, USA, Dec. 2014, pp. 1–8. doi: 10.1109/CICYBS.2014.7013371.

[19] M. Kedziora, P. Gawin, M. Szczepanik, and I. Jozwiak, “Malware Detection Using Machine Learning Algorithms and Reverse Engineering of Android Java Code,” Int. J. Netw. Secur. Its Appl., vol. 11, no. 01, pp. 01–14, Jan. 2019, doi: 10.5121/ijnsa.2019.11101.

[20] X. Liu and J. Liu, “A Two-Layered Permission-Based Android Malware Detection Scheme,” in 2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, Oxford, United Kingdom, Apr. 2014, pp. 142–148. doi: 10.1109/MobileCloud.2014.22.

[21] “Permission-Based Android Malware Detection | Semantic Scholar.” https://www.semanticscholar.org/paper/Permission-Based-Android-Malware-Detection-Aung-Zaw/c8576b5df33813fe8938cbb19e35217ee21fc80b (accessed Oct. 31, 2021).

[22] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, “Drebin: Effective and Explainable Detection of Android Malware in Your Pocket,” presented at the Network and Distributed System Security Symposium, San Diego, CA, 2014. doi: 10.14722/ndss.2014.23247.

[23] H. Cai, N. Meng, B. Ryder, and D. Yao, “DroidCat: Effective Android Malware Detection and Categorization via App-Level Profiling,” IEEE Trans. Inf. Forensics Secur., vol. 14, no. 6, pp. 1455–1470, Jun. 2019, doi: 10.1109/TIFS.2018.2879302.

[24] P. Rovelli and Ý. Vigfússon, “PMDS: Permission-Based Malware Detection System,” in Information Systems Security, vol. 8880, A. Prakash and R. Shyamasundar, Eds. Cham: Springer International Publishing, 2014, pp. 338–357. doi: 10.1007/978-3-319-13841-1_19.

[25] M. S. Alam and S. T. Vuong, “Random Forest Classification for Detecting Android Malware,” in 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing, Beijing, China, Aug. 2013, pp. 663–669. doi: 10.1109/GreenCom-iThings-CPSCom.2013.122.

[26] D. Congyi and S. Guangshun, “Method for Detecting Android Malware Based on Ensemble Learning,” in Proceedings of the 2020 5th International Conference on Machine Learning Technologies, Beijing China, Jun. 2020, pp. 28–31. doi: 10.1145/3409073.3409084.

[27] W. Li, J. Ge, and G. Dai, “Detecting Malware for Android Platform: An SVM-Based Approach,” in 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, New York, NY, USA, Nov. 2015, pp. 464–469. doi: 10.1109/CSCloud.2015.50.

[28] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, “DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware,” in Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Scottsdale Arizona USA, Mar. 2017, pp. 309–320. doi: 10.1145/3029806.3029825.

[29] Z. Yuan, Y. Lu, and Y. Xue, “Droiddetector: android malware characterization and detection using deep learning,” Tsinghua Sci. Technol., vol. 21, no. 1, pp. 114–123, Feb. 2016, doi: 10.1109/TST.2016.7399288.

[30] S. Ilham, G. Abderrahim, and B. A. Abdelhakim, “Permission based malware detection in android devices,” in Proceedings of the 3rd International Conference on Smart City Applications, Tetouan Morocco, Oct. 2018, pp. 1–6. doi: 10.1145/3286606.3286860.

[31] O. Yildiz and I. A. Doğru, “Permission-based Android Malware Detection System Using Feature Selection with Genetic Algorithm,” Int. J. Softw. Eng. Knowl. Eng., vol. 29, no. 02, pp. 245–262, Feb. 2019, doi: 10.1142/S0218194019500116.

[32] A. Senawi, H.-L. Wei, and S. A. Billings, “A new maximum relevance-minimum multicollinearity (MRmMC) method for feature selection and ranking,” Pattern Recognit., vol. 67, pp. 47–61, Jul. 2017, doi: 10.1016/j.patcog.2017.01.026.

[33] R. Tamura, K. Kobayashi, Y. Takano, R. Miyashiro, K. Nakata, and T. Matsui, “BEST SUBSET SELECTION FOR ELIMINATING MULTICOLLINEARITY,” J. Oper. Res. Soc. Jpn., vol. 60, no. 3, pp. 321–336, 2017, doi: 10.15807/jorsj.60.321.

[34] A. Farrell et al., “Machine learning of large-scale spatial distributions of wild turkeys with high-dimensional environmental data,” Ecol. Evol., vol. 9, no. 10, pp. 5938–5949, May 2019, doi: 10.1002/ece3.5177.

[35] S. Niu, R. Huang, W. Chen, and Y. Xue, “An Improved Permission Management Scheme of Android Application Based on Machine Learning,” Secur. Commun. Netw., vol. 2018, pp. 1–12, Oct. 2018, doi: 10.1155/2018/2329891.

[36] “Permission Based Malware Protection Model for Android Application,” presented at the International Conference on Advances in Engineering and Technology, Mar. 2014. doi: 10.15242/IIE.E0314102.

[37] G. L. Scoccia, S. Ruberto, I. Malavolta, M. Autili, and P. Inverardi, “An investigation into Android run-time permissions from the end users’ perspective,” in Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, Gothenburg Sweden, May 2018, pp. 45–55. doi: 10.1145/3197231.3197236.

[38] P. Topark-ngarm, “Identifying Android Malware Using Machine Learning Based Upon Both Static and Dynamic Features,” 2014.

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

[40] V. Bolón-Canedo, N. Sánchez-Maroño, and A. Alonso-Betanzos, Feature Selection for High-Dimensional Data. Cham: Springer International Publishing, 2015. doi: 10.1007/978-3-319-21858-8.

[41] B. Pes, “Ensemble feature selection for high-dimensional data: a stability analysis across multiple domains,” Neural Comput. Appl., vol. 32, no. 10, pp. 5951–5973, May 2020, doi: 10.1007/s00521-019-04082-3.

[42] A. Hamidreza and N. Mohammed, “Permission-based Analysis of Android Applications Using Categorization and Deep Learning Scheme,” MATEC Web Conf., vol. 255, p. 05005, 2019, doi: 10.1051/matecconf/201925505005.

[43] T. Boksasp and E. Utnes, “Android Apps and Permissions: Security and Privacy Risks,” p. 143.

[44] N. Yadav, A. Sharma, and A. Doegar, “A Survey on Android Malware Detection,” vol. 2, no. 12, p. 7, 2016.

[45] F. I. Abro, “Investigating Android Permissions and Intents for Malware Detection,” p. 169.

[46] M. Magdum and S. K. Wagh, “Permission Based Android Malware Detection System using Machine Learning Approach,” vol. 14, no. 6, p. 6, 2016.

[47] 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.

[48] H. A. Alatwi, “Android Malware Detection Using Category-Based Machine Learning Classifiers,” p. 62.

[49] P. Basavaraju and A. S. Varde, “Supervised Learning Techniques in Mobile Device Apps for Androids,” p. 12.

[50] R. N. Romli, M. F. Zolkipli, and M. Z. Osman, “Efficient feature selection analysis for accuracy malware classification,” J. Phys. Conf. Ser., vol. 1918, no. 4, p. 042140, Jun. 2021, doi: 10.1088/1742-6596/1918/4/042140.

[51] J. Abah, W. O.V, A. M.B, A. U.M, and A. O.S, “A Machine Learning Approach to Anomaly-Based Detection on Android Platforms,” Int. J. Netw. Secur. Its Appl., vol. 7, no. 6, pp. 15–35, Nov. 2015, doi: 10.5121/ijnsa.2015.7602.

[52] I. K. Aksakalli, “Using convolutional neural network for Android malware detection,” p. 7, 2019.

[53] I. Martín, J. A. Hernández, A. Muñoz, and A. Guzmán, “Android Malware Characterization Using Metadata and Machine Learning Techniques,” Secur. Commun. Netw., vol. 2018, pp. 1–11, Jul. 2018, doi: 10.1155/2018/5749481.

[54] S. Fallah and A. J. Bidgoly, “BENCHMARKING MACHINE LEARNING ALGORITHMS FOR ANDROID MALWARE DETECTION,” Jordanian J. Comput. Inf. Technol., vol. 05, no. 03, p. 15.

[55] X. Jiang, B. Mao, J. Guan, and X. Huang, “Android Malware Detection Using Fine-Grained Features,” Sci. Program., vol. 2020, pp. 1–13, Jan. 2020, doi: 10.1155/2020/5190138.

[56] H. Yuan, Y. Tang, W. Sun, and L. Liu, “A detection method for android application security based on TF-IDF and machine learning,” PLOS ONE, vol. 15, no. 9, p. e0238694, Sep. 2020, doi: 10.1371/journal.pone.0238694.

[57] A. M. García, "Machine learning techniques for Android malware detection and classification," p. 170.
[58] S. Y. Yerima, M. K. Alzaylaee, and S. Sezer, "Machine learning-based dynamic analysis of Android apps with improved code coverage," EURASIP J. Inf. Secur., vol. 2019, no. 1, p. 4, Dec. 2019, doi: 10.1186/s13635-019-0087-1.
[59] Y. Dong, "Android Malware Prediction by Permission Analysis and Data Mining," p. 71.
[60] V. P. D and V. P, "Detecting android malware using an improved filter based technique in embedded software," Microprocess. Microsyst., vol. 76, p. 103115, Jul. 2020, doi: 10.1016/j.micpro.2020.103115.
[61] A. Hemalatha and D. S. S. Brunda, "DETECTION OF MOBILE MALWARES USING IMPROVED DEEP CONVOLUTIONAL NEURAL NETWORK," vol. 7, no. 14, p. 7, 2020.
[62] S. Mahdavifar, A. F. Abdul Kadir, R. Fatemi, D. Alhadidi, and A. A. Ghorbani, "Dynamic Android Malware Category Classification using Semi-Supervised Deep Learning," in 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), Calgary, AB, Canada, Aug. 2020, pp. 515–522. doi: 10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00094.
[63] "Android Malware Detection | Kaggle." https://www.kaggle.com/defensedroid/android-malware-detection (accessed Nov. 14, 2021).
[64] X. Fu and H. Cai, "On the Deterioration of Learning-Based Malware Detectors for Android," in 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Montreal, QC, Canada, May 2019, pp. 272–273. doi: 10.1109/ICSE-Companion.2019.00110.
[65] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "DroidEvolver: Self-Evolving Android Malware Detection System," in 2019 IEEE European Symposium on Security and Privacy (EuroS&P), Stockholm, Sweden, Jun. 2019, pp. 47–62. doi: 10.1109/EuroSP.2019.00014.
[66] H. Cai, "Assessing and Improving Malware Detection Sustainability through App Evolution Studies," ACM Trans. Softw. Eng. Methodol., vol. 29, no. 2, pp. 1–28, Apr. 2020, doi: 10.1145/3371924.
[67] X. Zhang et al., "Enhancing State-of-the-art Classifiers with API Semantics to Detect Evolved Android Malware," in Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event USA, Oct. 2020, pp. 757–770. doi: 10.1145/3372297.3417291.
[68] A. Katrutsa and V. Strijov, "Comprehensive study of feature selection methods to solve multicollinearity problem according to evaluation criteria," Expert Syst. Appl., vol. 76, pp. 1–11, Jun. 2017, doi: 10.1016/j.eswa.2017.01.048.
[69] R. Grewal, J. A. Cote, and H. Baumgartner, "Multicollinearity and Measurement Error in Structural Equation Models: Implications for Theory Testing," Mark. Sci., vol. 23, no. 4, pp. 519–529, Nov. 2004, doi: 10.1287/mksc.1040.0070.
[70] M. Shyamala Devi, A. Poornima, J. Kosanam, and T. Hari Sathya Prashanth, "Outlier multicollinearity free fish weight prediction using machine learning," Mater. Today Proc., p. S2214785321019271, Mar. 2021, doi: 10.1016/j.matpr.2021.02.773.

BEENISH UROOJ received her Bachelor's in Computer Science from COMSATS University Islamabad, Wah Campus, Pakistan, 2015-2019. Currently, she is pursuing her Masters in Information Security from Department of Computer Science, COMSATS University Islamabad, Pakistan. She is working part time as a Graphic designer and a freelancer. Her conference paper about Security in SCADA Systems was declared runner up in best developmental research in 2021 (soon to be published). Her research interests lie in the domain of Cyber Security, Threat Hunting and Security in Industrial Control Systems (ICS).

MUNAM ALI SHAH received B.Sc. and M.Sc. degrees, both in Computer Science from University of Peshawar, Pakistan, in 2001 and 2003 respectively. He completed his M.S. degree in Security Technologies and Applications from University of Surrey, UK, in 2010, and has passed his Ph.D. from University of Bedfordshire, UK in 2013. Since July 2004, he has been a Lecturer, Department of Computer Science, COMSATS Institute of Information Technology, and Islamabad, Pakistan. His research interests include MAC protocol design, QoS and security issues in wireless communication systems. Dr. Shah received the Best Paper Award of the International Conference on Automation and Computing in 2012. Dr. Shah is the author of more than 50 research articles published in international conferences and journals.

CASTREN MAPLE is Professor of Cyber Systems Engineering at WMG's Cyber Security Centre (CSC), University of Warwick. He is the director of research in Cyber Security working with organizations in key sectors such as manufacturing, healthcare, financial services and the broader public sector to address the challenges presented by today's global cyber environment. He is a member of several professional societies including the Council of Professors and Heads of Computing (CPHC) whose remit is to promote public education in Computing and its applications and to provide a forum for those responsible for management and research in university computing departments. He is an elected member to the Committee of this body. He is an Education Advisor for TIGA â˘A ¸S the trade association representing the UK's games industry. He is also a Fellow of the British Computer Society, the Chartered Institute for IT and is a Chartered IT professional. He also holds two Professorships in China, including a position at one of the top two control engineering departments in China. His interests include Information Security and Trust and Authentication in Distributed Systems.

MUHAMMAD KAMRAN ABBASI received his PhD in the field of Computer Science from the University of Bedfordshire, UK. He is currently serving as Associate Professor in the Department of Distance Continuing and Computer Education, University of Sindh. His research focuses on Unsupervised Machine Learning, Informatics and Educational Technology.

SIDRA RIASAT completed her Bachelor's in Computer Science from Fatima Jinnah Women University. She is currently doing her Masters in Information Security (Department of Computer Science) from COMSATS University, Islamabad. Her research interest includes Cyber Security, Block chain smart cities and SCADA networks.