IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 28 September 2022, accepted 30 October 2022, date of publication 3 November 2022, date of current version 10 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3219047

RESEARCH ARTICLE

A Method for Automatic Android Malware Detection Based on Static Analysis and Deep Learning

MÜLHEM İBRAHIM[1], BAYAN ISSA[2], AND MUHAMMED BASHEER JASSER[3], (Member, IEEE)

[1]Department of Computer Engineering, Faculty of Engineering, Turkish-German University, Beykoz, 34820 Istanbul, Turkey
[2]Faculty of Informatics Engineering, University of Aleppo, Aleppo, Syria
[3]Department of Computing and Information Systems, School of Engineering and Technology, Sunway University, Petaling Jaya, Selangor 47500, Malaysia

Corresponding author: Bayan Issa (bayan.issa.b@gmail.com)

ABSTRACT The computers nowadays are being replaced by the smartphones for the most of the internet users around the world, and Android is getting the most of the smartphone systems' market. This rise of the usage of smartphones generally, and the Android system specifically, leads to a strong need to effectively secure Android, as the malware developers are targeting it with sophisticated and obfuscated malware applications. Consequently, a lot of studies were performed to propose a robust method to detect and classify android malicious software (malware). Some of them were effective, some were not; with accuracy below 90%, and some of them are being outdated; using datasets that became old containing applications for old versions of Android that are rarely used today. In this paper, a new method is proposed by using static analysis and gathering as most useful features of android applications as possible, along with two new proposed features, and then passing them to a functional API deep learning model we made. This method was implemented on a new and classified android application dataset, using 14079 malware and benign samples in total, with malware samples classified into four malware classes. Two major experiments with this dataset were implemented, one for malware detection with the dataset samples categorized into two classes as just malware and benign, the second one was made for malware detection and classification, using all the five classes of the dataset. As a result, our model overcomes the related works when using just two classes with F1-score of 99.5%. Also, high malware detection and classification performance was obtained by using the five classes, with F1-score of 97%.

INDEX TERMS Android, deep learning, malware, mobile security, static analysis.

I. INTRODUCTION

There is a growing need for an efficient malware detection tool for Android, since that Android is the most used mobile system [1] and there are hundreds of new applications for it that are released each day [2], thus scanning and checking them manually became a non-possible solution. More than six million mobile malware samples were encountered by McAfee in 2014 [3], and over 12 thousands Android malware samples were detected daily in 2018 [4]. Along with the rapid growth of malware, the new samples of Android malware are more sophisticated than the samples that were detected previously in regards to avoiding anti-virus detection through code obfuscation and encryption [4].

ML-based methods, and more specifically deep neural networks, have shown to be efficient at detecting malware, as they are able to learn features and patterns automatically and with multiple levels of abstraction [5] from a limited set of training examples, thus eliminating the need to explicitly define signatures when developing malware detectors [6].

Although there are lot of studies that tackle this subject, most of them utilize just the application permissions and API calls, which does not seem effective for new sophisticated samples and for a long term, as the benign applications are requiring more permissions and including more API calls than malware applications. So, those two most used features

The associate editor coordinating the review of this manuscript and approving it for publication was Mohamad Afendee Mohamed.

are not sufficient alone, and do not allow to learn the real characteristics of malware, thus, there is a need to handle more useful features and techniques for malware detection. It is also noticed that the used datasets in previous papers are outdated and not available anymore.

In addition, most of the existing studies do not consider malware classification, as they just train the models and classifiers to predict one of two classes, namely malware and benign, ignoring the malware classification problem, which is an important issue to tackle in this subject and essential for the cybersecurity community to help perform the right action.

Yet another limitation of the most existing studies is the usage of traditional machine learning algorithms, which proves inefficiency in feature engineering process by depending more on human intelligence and individual judgment [4]. On the other hand, the layered structure of deep learning based models improve the learning of abstract and highly non-linear patterns, which helps learn the features automatically and capture the substantial characteristics of complex data, which improves generality on new data.

Deep Learning methods, in particular, are typically better suited to capturing semantic knowledge within Android applications than classic ML methods, especially when enough data is available to build a meaningful semantic embedding [4].

In this paper, static analysis is used and a functional API deep learning model is proposed, which takes as inputs the most useful observed features of android applications, and those are: the file size, Permissions, services, API function calls, Broadcast receivers, Opcode sequences, and the fuzzy hash, which is used for similarity detection. They are automatically extracted using Bash script and Python3, which execute commands provided by the Androguard tool [7].

The structure of the model with the multiple layers, helps learn the most distinctive information from the given inputs. Additionally, the utilization of tokenization and embedding layers, helps in clustering and detecting the similarity in the discrete independent text features, such as permissions. The RNN part of the model helps learn the characteristics of the fuzzy hash values in order, and to the best of our knowledge, the usage of the fuzzy hash in this field is a novel technique, and along with Recurrent neural network, the model can detect the similarity in Android applications so efficiently, and this also enhances the classification of the new and modified samples. The model performance was evaluated using four metrics, namely the accuracy, F1 score, Recall, and Precision metric, and the obtained result is 96%.

The used dataset is 'CICMalDroid2020' [8] which is the newest and most diverse dataset that combines new APK samples as well as samples from the famous datasets that are used in previous studies, such as AMD and MalDozer [9]. The used dataset categorizes the android application samples into five classes: SMS malware, Banking malware, Riskware, Adware, and Benign. Using this diverse categorized dataset, our model outputs a prediction of the application's class, which helps in detecting as well as classifying the Android malware.

In summary, the contribution of our work is as follows:

- A new method for malware detection and classification was proposed with two new proposed features in the static analysis scope.
- Using two target classes, which are malware and benign, our model is able to achieve very high malware detection rates evaluated by precision, recall, F1-score, and accuracy metrics, with obtained value of 99.5% for all of them.
- Using the five classes of the dataset, our model is able to achieve high malware detection and classification performance evaluated by precision, recall, F1-score, and accuracy metrics, with obtained values of 97%, 96%, 97%, 97% respectively.

The rest of this paper is organized as follows: Section 2 lays out the background to this study. Section 3 subjects existing methods of Android malware detection, while Section 4 provides a detailed description of our proposed method with its all steps, with Section 5 exploring the results and discussions thereof, and Section 6 shows the results of some tested variations of our model to tune its hyperparameters, and Section 7 shows experimental evaluation of other models, including some traditional machine learning classifiers that are most used in previous studies. Finally, Section 8 concludes the paper.

II. BACKGROUND

In this section, we provide the necessary background that is relevant to our proposed method.

A. ANDROID

The most widely used mobile operating system is Android, which is built on the Linux kernel and uses the Java programming language. Additionally, a number of drivers and libraries have been altered to improve Android's performance on mobile devices. In 2005, Google began supporting Android Inc. financially, and in 2008, the operating system's first smartphones were released (HTC Dream). Because it is open source and distributed under the Apache License, the operating system has seen widespread and quick development. According to AppBrain [10], over 2.6 million Android apps exist in Google Play store as of the first quarter of 2022, with 37 percent identified as low-quality apps.

Android uses a special virtual machine, that is, the Dalvik virtual machine, which uses special bytecode. Therefore, standard Java bytecode cannot be run on Android. In order to convert Java Class files into ''dex'' (Dalvik executable) files, Android provides the ''dx'' tool. The ''aapt'' (Android Asset Packaging Tool) bundles Android applications into a ''APK'' (Android Package) file.

B. ANDROID APPLICATION BASICS

Android Applications have the extension APK which stands for Android Package Kit, the APK is made of a collection of components, these components are categorized into four types, where each application can be composed of one or more of these types [11]. The four types of android application components are:

- *Activities:* It is an interface component that implements interactions with the user. Each activity is designed to handle single user action. For instance, an appointments list in a task manager application is an activity, and showing the detail of an appointment is the role of a second activity. Each activity composes of one or more view objects, which are interface objects, such as buttons, labels, etc.
- *Services:* Or service components are background components that run independently of the user interface. They can operate for a long time even after the user switches to another application. A service can play music, download a file, or handle network transactions, all from the background. Services can also be used for interprocess communication (IPC) between Android applications [12].
- *Broadcast receivers:* System-wide broadcast events can occur when a device start or receive SMS or call, Broadcast receivers are made to listen to these events and interact with them. They run in background even when the app is closed.
- *Content providers:* Components that allow external apps and system components to access application data.

Android applications are typically written in Java or Kotlin and compiled into a single archive file (Android package or APK), along with data and resource files. The components of the APK include:

1) an XML manifest file that contains information such as app description, components declaration (i.e. Activities, permissions etc.)
2) A Classes.dex file(s) that is a Dalvik executable file that runs in its own instance of a Dalvik Virtual Machine (or Android RunTime for newer versions of Android).
3) A ''/res'' directory for indexed resources like images, icons, music etc.
4) A ''/lib'' directory for compiled code.
5) ''/META-INF'' folder including the app certificate and list of resources, SHA-1 digest etc.
6) Resources.arsc which is a compiled resource file

C. MALWARE TYPES

There are lot of malicious software (malware) types, here we define just the existing ones in the dataset that we used in this paper:

- Adware: Adware is a type of software that is designed to automatically deliver unwanted and annoying advertisements to the user.
- Riskware: Riskware is any legitimate program that poses a potential risk due to security vulnerabilities, software incompatibility, or legal violations. These applications are not designed for malicious purposes, but have features that can be used for malicious purposes. If used with malicious intent, the Riskware program can be considered as malware. This gray area of security makes Riskware a particularly difficult threat to deal with.
- Spyware: Monitors and sends information of victim's system by capturing keyboard typings, gaining access to microphone or webcam, etc. Spyware does this by modifying the security settings on users' devices. It often bundles itself with legitimate software. The SMS malware and the Banking malware categories that exist in the used dataset belongs to spyware, where they steal personal information from text messages, and banking account information from banking applications, respectively.

D. TYPES OF MALWARE ANALYSIS

There are two main approaches for automatic malware detection which are static analysis and dynamic analysis.

Static analysis detects the malicious application without the need to actually run it, and that is done by analyzing the packed files and the code which is obtained by using disassemblers or decompilers, and this process is called *Reverse Engineering*, or *Back Engineering*. However, Static analysis cannot detect some sophisticated malwares which have malicious runtime behavior, like for example generating a dynamic string which in turn downloads a malicious file. There are also approaches to complicate reverse engineering by using tools that obfuscate the code. The most common and free tool for that is Proguard [13], which obfuscate the code by renaming the classes, fields, and methods using short meaningless names. Another commercial tool built on it is called DexGuard [14], which is claimed to complicate static as well as dynamic analysis. Some other tools are Ijiami ApkProtect [15], and Bangcle [16].

From the perspective of the Android app developers, it is insecure to allow the decompiling process of their code, because reverse engineering is also used by attackers for many purposes, like infecting the apps or the servers that control the apps, searching for sensitive data hardcoded in the code, or reusing the code for their own benefit. So, we cannot consider the applications that contain obfuscated or encrypted code as malicious.

Another limitation of the static analysis is the external functions, which cannot be fetched by the static analysis, and while the benign apps use external functions to reduce their size on phone storage, we cannot assume that an app with lot of calls to external funtions is considered as malware.

Dynamic Analysis detects the malicious application by executing it in a virtual system so that the behavior of the application can be seen in action and without the risk of letting it infect a real used system or escape into the enterprise network. Dynamic Analysis requires thousands of

applications' runs to train and test the models; thus, it requires lot of resources and it is also slower than the static analysis. While it is considered in general a powerful solution for detecting malware applications and its accuracy is higher than the static analysis, it also cannot detect some sophisticated malwares which can discover that they are running in a virtual system, thus, they change their behavior to deceive the system.

Static Analysis cannot detect sophisticated malicious code, and sophisticated malwares can hide and deceive the virtual system with the dynamic analysis, so by combining those two techniques, hybrid analysis can provide security team the best of both approaches. For example, Hybrid analysis can apply static analysis to data generated by dynamic analysis, like when a malicious code runs and make changes in memory, dynamic analysis detects that and gives alert to security team to check that and perform static analysis on that memory dump. Even the most sophisticated malware threats can be found through hybrid analysis, but it is also so expensive and time-consuming solution.

E. TOOLS FOR MALWARE ANALYSIS

There are different tools and software that use reverse engineering and allow to decompile and debug android applications, some of them are Radare2 [17], Dex2Jar [18], JADX [19], and the one we use in this paper; Androguard [7], which is a complete framework developed in Python and allow to analyse APK files and extract lot of features from them, like services, resources, dex files and many others. Additionally, every Androguard feature can be added to customized Python scripts, making it simple to get comprehensive information on a file.

One limitation of Androguard is that it can be too slow in analysing APK files that are more than 10 MB in size. However, by testing other existing tools, nothing seems to give a better performance. Also, some tools, like ClassyShark [20] and ApkStudio [21] are limited to user interface, thus, automation process of the feature extraction is not applicable with them. Also, not all android decompilers allow the extraction of all the features of Android applications, some are limited to decompiling dex classes, and others are limited to manifest file.

There are also different websites for malware detection, the most known one is VirusTotal [22] which is multiscanners for antiviruses. Obviously, this can detect new samples.

For dynamic analysis, there is DroidBox [23] which use API call logs that can help explain APK behaviors. Another Tool is AppsPlayground [24]. This tool aims to automate the dynamic analysis of Android apps, however access to the tool requires registration. SandDroid [25] combines both static and dynamic analysis techniques.

F. CRYPTOGRAPHIC HASH FUNCTIONS

In information security, hash methods are used to generate ''fingerprints'' for documents, which characterize a possibly large document as unambiguously as possible by means of a short string with a fixed number of characters. Hash functions are also used to store passwords securely in an obfuscated, irreversible form. This process of hashing has two main properties. First, the output will be drastically altered if even one bit of the input is altered. Second, finding another input that generates the identical hash is computationally impossible given an input and its hash. Examples of hash algorithms are SHA-1 (1995 revision of SHA) which generates hash values of 160-bit length, SHA-2 family which includes SHA-224, SHA-256, SHA-384, SHA-512 where the number represents the length of the hash value in bits, and MD5 hash function.

Hash algorithms are used by forensic examiners to locate known files in collections of unknown files. An examiner compiles a list of known files, generates and maintains the cryptographic hash values for each of those files. During future investigations, Every file under inquiry can have its hash value calculated, and the examiner can then compare those hash values to previously computed known values. However, malicious individuals can defeat this strategy by altering known files even by one bit, which then totally changes the file's cryptographic hash. However, this limitation can be overcome with SSDeep, introduced in paper [26], SSDeep ''is a program for computing context triggered piecewise hashes (CTPH). Also called fuzzy hashes, CTPH can match inputs that have homologies. Such inputs have sequences of identical bytes in the same order, although bytes in between these sequences may be different in both content and length.'' This means that changing some bits, in other words modifying a file, will not change the complete hash value, and we will still be able to detect similarity between the original and the modified file, and that is why CTPH is a better option for detecting malwares and their variations.

III. EXISTING ANDROID MALWARE DETECTION TECHNIQUES

There are lot of published studies regarding Android malware detection based on machine learning, the majority of them use static analysis where the APK packages are analyzed and perceptive features are extracted, such as permissions, API calls, and opcode sequences. Other studies use dynamic analysis where the applications are executed in a virtual environment and their behaviors are analyzed, such as the consumption of CPUs and RAMs. There are also some studies that use a combination of the static and dynamic analysis, which is called hybrid analysis.

Some of these studies developed tools for the end user, which can detect malware directly on the Android device (on-device), other studies analyze and detect the malwares outside the Android devices (off-device).

In Figure 1, we can see a general overview of all the reviewed studies in this article, with machine learning approach as a common feature, and different analysis types.

A. STATIC ANALYSIS BASED MALWARE DETECTION

Opcode sequences were used in papers [27], [28] with convolutional neural network (CNN) which takes the sequences as

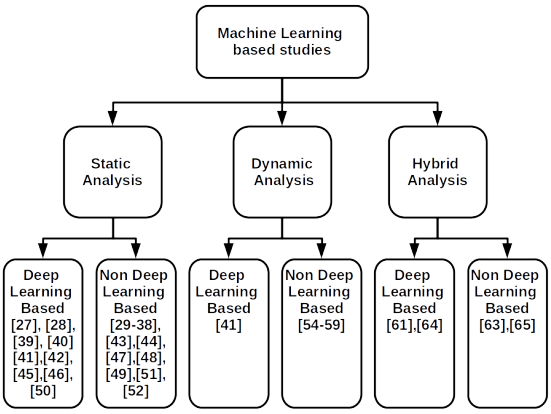

FIGURE 1. Overview of the existing studies.

one-hot vectors, the resulted accuracies were 88 and 99 percent respectively.

The papers [29], [30], [31], [37], [44], use both permissions and API calls, and pass them to different machine-learning algorithms such as random forest, SVM, and logistic regression. By utilizing different datasets, various accuracies were obtained in the range 87 – 98%.

The papers [32], [33], [34], [35], [36], use only permissions with clustering algorithms and cross validation. The latter proposed an on-device malware detection and remover, but its accuracy is not given.

The study [37] uses permissions and API calls and tries genetic algorithm in the feature selection process. After that, the features are passed to different ML algorithms such as J48, decision tree, random forest, and naive bayes. With 6000 total Android samples, the best obtained accuracy was about 97%.

The paper [38] extracts opcode sequences from selected datasets gathered between the years 2012-2015, and applies semantic simplification of the opcode sequences to enhance the resilience.

The study [39] uses opcode sequences and NLP (Natural Language Processing) algorithms. The obtained average accuracy was 83.56% using 5 malware families. It is noticed that training the model to classify more malware families, reduces the average accuracy significantly.

The paper [40] uses API call graphs with CNN and a lightweight classifier. the obtained accuracy was 91.27%.

The study [41] uses deep learning and ensemble classifiers for malware detection classification. Using multiple datasets, they train there model against static and dynamic features, each of them independently, and just two classes were used for static features; malware and benign.

The paper [42] uses deep learning algorithms, mainly Graph Neural Networks (GNN) and Generative Adversarial Network (GAN) for Android malware detection, again, this study also divides Android samples into just two class; malware and benign, ignoring the classes of malware.

The study [43] uses an out of the box technique, by converting the APK packages into audio files, they tested lot of different classifiers and obtained high performance for malware detection.

The study [44] concentrate on the malware category Ransomware, and uses swarm optimization algorithm to tune the classification algorithm's hyperparameters. They proposed a method based on SVM algorithm. They categorize their collected samples into ransomware and benign, and applied different oversampling algorithms to it to have a balanced dataset. Their method was able to achieve high performance for ransomware detection.

A paper [27] published in 2017 uses convolutional neural network (CNN) method for malware classification based on static analysis. They disassemble the dex classes in APK files to Smali files and from that they extract the opcode sequences discarding the operands. This sequence of opcode instructions then is encoded as one-hot vectors and passed to CNN layer. The last layer is a Multi-Layer Perceptron (MLP) which outputs the probability that the current example is malware. Three different datasets were used and the mean accuracy was 88%.

The paper [28] proposed a method that uses deep convolutional neural network to learn from opcode sequences. The experiments gave an accuracy of 99%.

DroidDeepLearner [45] uses static analysis with permissions and API function calls features, by testing different algorithms, the best obtained accuracy was 93.96% with DBN algorithm and F1-score metric.

DroidDeep [46] also uses static analysis and DBN algorithm. The overall obtained accuracies for different accepted ratios between benign and malware samples were in the range 92-97.5%.

Mlifdect approach is presented in the study [47] uses parallel machine learning classification method applied on 8,385 android samples. the obtained classification accuracy was about 98%.

Another study [29] uses parallel machine learning classifiers that utilize diverse algorithms with inherently different characteristics for early detection of Android malware. The extracted features are API calls, permissions, and the Standard OS and Android framework commands which are typically placed in hidden files within the APK. The used ML algorithms include: Decision Tree, Simple Logistic, Naïve Bayes, and others. They used supervised learning algorithms with McAfee dataset. They used a combined classification strategy, which involves combining the classification judgments made by each individual classifier in parallel. the best detection rate performance of 97.5% was obtained.

The paper [30] uses static analysis where the requested permissions and API calls are extracted and multihot encoded, then they are passed to different machine learning algorithms which are SVM, random forest, and logistic regression. The best accuracy obtained was about 98%. Three datasets were used which are AMD, Drebin, and UpDroid.

Another paper [31] uses risky permissions and API calls as features in the SVM algorithm. to evaluate the proposed method, experiments were made by utilizing Drebin dataset

and some of Google Play Applications. The overall obtained accuracy was 86%.

The study [33] also uses permissions and passes it to K-means clustering algorithm to detect Android malware. An experiment was made with 500 sample Android applications the best accuracy obtained was 91.75%.

The paper [32] presents an adaptive neuro-fuzzy inference system with fuzzy c-means clustering and depends on the permissions feature. This approach achieves the highest classification accuracy of 91%.

PUMA [34] uses permissions approach with k-fold cross validation to evaluate the performance of different machine-learning classifiers. Among them, SimpleLogistic showed best result with 84% accuracy.

Static analysis and Machine Learning was used by [48]. They extracted the APIs class under <used-permission> tag In the Manifest file, and used three algorithms: SVM, J48 and Random Forest. It is claimed that these three algorithms give the best result in research related to malware detection compared by many other algorithms like KNN, Naïve Bayes and Bagging Bootstrap. The highest accuracy obtained was 92.40%.

The study [49] uses static analysis that runs directly on smartphones to analyse and check the downloaded applications. They used Android Asset Packaging Tool to extract the following information from the Manifest file: permissions, activities, services, content providers, broadcast receivers, and filtered intents. And using a lightweight Android disassembler they extract API calls and network addresses contained in applications' disassembled code. Then they embedded these features in a vector space and trained a model offline to transfer then the trained model to the smartphones for direct analysing and predicting. The acquired accuracy was 93.90%. The drawbacks of this study are as follows:

- The used dataset is too unbalanced: 123,453 benign samples vs just 5,560 malware samples.
- Some collected features are not that useful: taking for instance IP addresses that are found in the disassembled code, it is hard to find two applications that use the same addresses. the activity names also represent the names of the windows and elements in the interface such as buttons and labels. And the collected intents from the manifest files will be empty for most of the samples (at least today's samples as I detected in this thesis). Thus, these features can be a waste of time and memory.
- It became now relatively old; the data were collected in a period from August 2010 to October 2012. Thus, the model can not perform well on our today's new samples.

The study [50] uses Android samples collected between 2015 and 2016, the perfectly balanced dataset with high number of samples, and the only two target classes, helped in giving high accuracy of 98%. The last two drawbacks discussed for the previous study applies to this study, too.

The study [51] uses static analysis and extracts only some of the information found in manifest files, which are permissions and intent filters. The used dataset consists of 30 malware and 30 benign apps. Then an experiment was made with 235 benign and 130 malware samples to test the effectiveness of the proposed method, the overall accuracy obtained was about 90%.

The paper [52] uses both permissions and Control Flow Graph to detect malware statically. The used algorithm is Support Vector Machine, and the best obtained accuracy was with Recall metric which gave about 85% accuracy, other metrics, such as F1 Score, showed far lower accuracies – below 30%.

MalDozer was proposed [9] which use API Calls with convolutional neural network, their machine learning model was tested on different datasets and could detect malware samples and attribute them to their actual families with an F1-Score of 96% - 99%.

The paper [53] proposed a new method that uses creator information (serial number) as well as the API calls, permissions, intent filters, file hash, and system commands. The obtained average malware classification accuracy was 98%. In this paper, A serial number blacklist was made, which we believe that is not too useful because of new malwares will have definitely new serial numbers, so the detection and classification accuracy for new applications is questionable.

In Table 1, we see a comparison between studies that use static analysis and the different features they extract from APK packages.

B. DYNAMIC ANALYSIS BASED MALWARE DETECTION

The study [54] extracts system calls from the dataset CIC-ANDMAL2017, which are then fed to different machine learning algorithms, its declared that among them the K-Nearest Neighbor and the Decision Tree algorithms gave the best malware detection rates with F1-score metric, the rates are 85 and 72% respectively.

The study [55] proposes a parallel machine-learning model which used different classifiers such as J48, KNN, SVM, and random forest, to detect and classify Android malware using dynamic features.

The Andro-profiler system was proposed in the study [56] which depends on the system logs including the system calls that are extracted during dynamic analysis.

Another study [57] uses dynamic analysis to extract Sequences of System Calls to detect Android malwares. They assume that malicious behaviors such as sending high premium rate SMS or cyphering data for ransom are implemented by specific system calls sequences. So, this method is based on the "fingerprint" of the malwares. Implementing the trained model in a real device, the obtained detection accuracy was 97%.

Dynamic analysis is also used in the paper [58] to capture system calls during the applications' run-time interactions with the Android system. J48 and random forest algorithms were used to classify a dataset consists of 50 malware and 50 benign samples.

TABLE 1. The extracted Android applications' features in studies with static analysis.

Study/Feature	Opcode sequence	Filtered intents (from manifest)	permissions	API Calls	services	Broadcast receivers	Fuzzy Hash	File size	OS commands
[27]	√								
[29]			√	√					√
[30]			√	√					
[51]		√	√						
[32]			√						
[31]			√	√					
[33]			√						
[34]			√						
[9]				√					
[28]	√								
[35]			√						
[45]			√	√					
[37]			√	√					
[36]			√						
[38]	√								
[39]	√								
[44]			√	√					
Our proposed work	√		√	√	√	√	√	√	

Random Forest Classification was used in another study [59]. They worked with the free parameters of Random Forest algorithm (as the number of trees and the depth of each tree in the forest) on an Android feature dataset that was made using dynamic analysis, where they observed different features in battery, binder, CPU, memory, network and permission categories, they got a high accuracy with very tiny ratio of misclassification. They also found that in this case and for the used algorithm, more trees in random forest classification appears to be better, and also the depth of trees should be not less than 16. It is also observed that the Lower features per tree is better.

C. HYBRID ANALYSIS BASED MALWARE DETECTION

Hybrid analysis was used by Kabakus and Dogru in their paper [60], where they used two datasets and applied dynamic analysis on them using virtual Android device (emulator), followed by static analysis based on permissions approach and API Calls. Therefore, they discovered shared signs among malicious apps such as disabling the mobile data connection and the over-privileged permissions that are more common than in benign apps.

In another paper [61], deep learning is used to characterize and detect Android malwares. They have developed an online deep-learning-based Android malware detection engine ''DroidBox'' that can automatically detect whether an app is a malware or not. Their engine was based on TaintDroid, which is able to run a dynamic taint analysis with system hooking at the application framework level and keep an eye on various app operations like data leaks, cryptography operations and mobile phone calls. Hybrid analysis technique was conducted to extract features from each app, which fall under one of three types: required permissions, sensitive APIs, and dynamic behaviors. The total obtained features number was 192. They managed to understand the characteristics of malware and they reached a high classification accuracy of 96.76%.

The study [62] uses system calls to create data pattern sets for both malware and benign samples. new samples' patterns are compared with the created pattern sets to detect their category. The obtained detection accuracy was about 91%.

OmniDroid dataset was proposed in the paper [63], which consists of 22,000 malware and benign Android samples. Then, static and dynamic analysis were applied to this dataset, and ensemble classifiers were trained.

In paper [64], deep learning model was used for Android malware detection based on hybrid analysis, the model was tested on 165 benign and 146 malware samples. The test results shows that hybrid analysis could increase malware detection accuracy by 5%.

The paper [65] proposes a Tree Augmented naive Bayes based method using API calls, permissions and system calls features. The results showed that the malware detection with this method can take a long time with an accuracy of 97%.

D. LIMITATIONS OF EXISTING TECHNIQUES

From the previously mentioned information, we notice different limitations, for example, the accuracy in [27], [31], [34], and [52] was below 90%. Also, as mentioned previously, the used features may be insufficient, like the paper [28] which handles the opcode sequences alone, or [30] which considers just the API calls and permissions. The study [29], in addition to the few features limitation, uses multiple machine learning algorithms in parallel, which has two disadvantages, the first one is that the traditional machine learning algorithms are shallow and ineffective in learning complex data as discussed before, and the second one is that this technique, with all these parallel used algorithms, can be time-consuming and expensive regarding the used resources. The cost of time and resources expands significantly with the dynamic and hybrid types of analysis.

Another important limitation to mention, is that the most previous studies categorize the data into just two classes, namely malware and benign, and this helps in detecting malware samples, but not in classifying the malware into its

correct category, and this in turn reduces the chance of taking the appropriate precautions to protect Android devices [55].

Finally, the datasets that are used in most of the studies are from the years 2009-2015, even in new papers that are released after 2020 such as [38] and [37], these datasets contain samples that were for older versions of android system. Also, because of the rapid evolution of Android malware and its techniques, which definitely affects the extracted values, such datasets are considered outdated.

IV. THE PROPOSED ANDROID MALWARE DETECTION METHOD

In this paper, we propose a method that is based on static analysis, where we extract all the observed useful features from Android applications. There are very few studies that extract and use all those features together, and these features are namely the permissions, services, broadcast receivers, API calls, and opcode sequences. as the extraction of all these features is expensive regarding the time cost and resources like RAM and disk space usage. It is also hard to combine all these features together in one model, and this is the reason of our building of a functional API model, as it is, unlike the sequential models, a practical solution that gives the freedom to handle each input individually and independently.

Additionally, we propose two new features, which are the file size and the fuzzy hash. The former can be very useful feature in detecting malware, as it is strongly observed, that the sizes of malware samples are generally under 5 MB, while the sizes of the benign samples are more than 10 MB in general, so there is a big difference between the sizes of malware and benign applications. The fuzzy hash is a powerful technique that is used for similarity detection, and by using it along with a Gated Recurrent Unit (GRU) layer, any application that is modified can be effectively detected. The GRU layer lets our model learn the hash characters as a sequence in order, as the order of the characters in the hash values is important and meaningful, and it should not be discarded.

By combining all of these features together, there is no opportunity for data correlation, as the permissions, services, and broadcast receivers are extracted from the manifest xml file and each one represents different thing, so, they do not correlate with each other, whereas the API calls and opcode sequences are extracted from the classes.dex files. Although they are both extracted from the same sources, they are somehow independent from each other, as the opcodes represent only the operations to be performed, and do not include any information regarding the system functions that are called. Obviously, the two added features are also completely independent from the other features, as they are not extracted from the APK components at all, but calculated by using the APK as one file without extraction.

Figure 2 shows an overview of our method, where we see the input which is an APK file and the output is a prediction and classification, which can be one of five classes, namely benign, SMS malware, banking malware, adware and riskware. In the middle between the APK input and the prediction as an output, there are two processes, which are the feature extraction process, which uses the manifest.xml file and the decompiled classes.dex files as well as the APK file itself, after this process, data preprocessing is done to the extracted features, and then these data is passed to our trained model to give the prediction. More details about the feature extraction process and data preprocessing can be found in the next subsections. The data preparation and the constructed model blocks that take place in Figure 2 are shown more clearly and in more detail in Figure 3.

In the next sections, we talk about our method steps, where the feature extraction process is explained in detail, then the data preprocessing is explained in detail, too. After that, the construction and training process of our functional API deep learning model is explained.

A. FEATURE EXTRACTION

The first step after having the data, which are 14079 categorized Android applications, is to extract the features from them. After doing lot of research, and also from the Android components that are described previously, we found that the most useful features for Android malware detection are as follows:

1) Application size: It's noticed that the malware samples have in general low size, most benign apps are about three times (and more) the size of the malware apps, and this can be very logical because the malware developer does not make a real app with useful and lot of functionalities, but instead want to accomplish his purpose immediately and write any useless code along with malware functions, and this also can be clearly observed from the application's category, where most malware applications can be categorized as accessories, such as wallpaper changer, image beautifier, YouTube downloader, etc. Unfortunately, there is no dataset that gives the application's category in this way, otherwise, we believe that it could be a very important feature.
2) Permissions: which are extracted from the manifest.xml file. An app that does not have access to the internet, does not read or write to external storage, can be assumed as benign, on the other hand, what would a wallpaper changer app do with the location of the phone or with ''receive boot completed'' permission.
3) API calls: which are extracted from the decompiled code, and represent the functions of the system that the application is calling in the background during its execution.
4) Services: those are described previously and extracted from the manifest.xml file.
5) Receivers: same as the previous one.
6) Fuzzy Hash: which can detect similarity as explained previously in the second section.
7) Opcode sequences: which are extracted from the decompiled code. These represent the instructions that

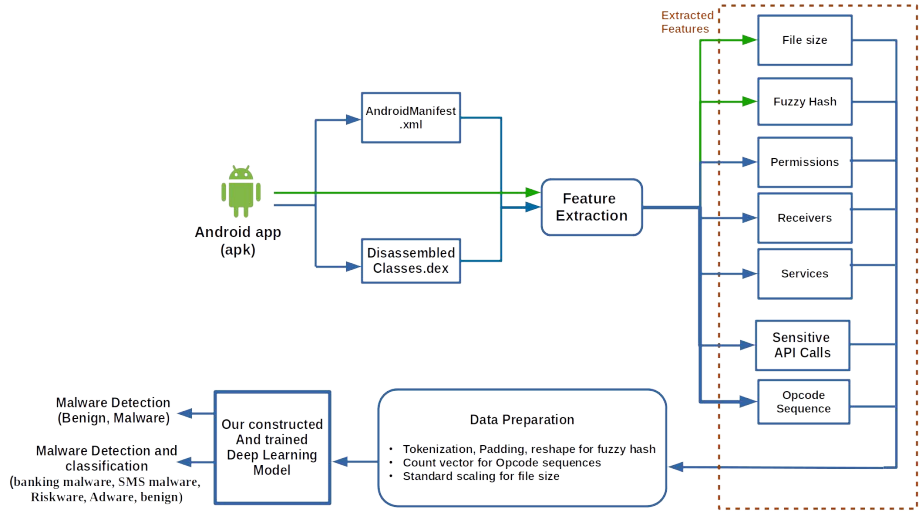

FIGURE 2. Method overview.

the application will execute during its runtime. These instructions are extracted in the same written order.

During the feature extraction process with the Androguard tool, two errors could occur due to the reading or extraction failure from some APK files, which was tackled by writing and executing extra codes to eliminate the error causing samples. Finally, the API calls, which was extracted independently, was merged with the other features into the generated csv file.

B. DATA PREPROCESSING

The Android applications we have are classified into five classes. So, the features of applications were extracted from each class separately. Thus, five csv files were obtained, one for each class, those csv files were merged together, shuffled, and the rows that contain any null value in any column were deleted, then each feature were prepared individually to be passed to our deep learning model as the following:

- Application's size: using standard scaler, each size was converted to a float number in the range of -1 and $+1$.
- opcode sequences: this feature was planned to be passed to RNN layer(s). However, as each sequence was really quite long, with the longest sequence that had more than 2 million opcodes (which are represented in integers), so padding all other to that length and passing it to RNN layer did not seem to be useful and efficient, instead, the count of each opcode was calculated, and as we had the opcode represented in integers, the integers were in the range of 1 to 768, so a matrix of 768 columns was created.
- Fuzzy Hash: A tokenizer preprocessing layer with char level was used to represent each character in the hashes as an integer, taking into account upper and lower letters as well as the symbols, 65 unique characters were embedded. The hashes were converted completely to the integer sequences that resulted from the method "texts_to_sequences" of the tokenizer layer. Then, pre-padding was applied to make all the hash values in the same length, finally a reshape process was applied to convert the hash values to three dimensions, which is required for the next GRU layer.
- Permissions, Receivers, Services, API calls: TextVectorization layers are used to build vocabularies and map these features' values to integers. Because of the difference of the nmuber of unique values in each of these four features, different vocabulary sizes were given, which are shown within the brackets in Figure 3.
- Category: this is the target, which can be a number from 0 to 4 that represents the class of the application. which again, can be benign, SMS malware, riskware, banking malware, and adware. This column was onehot encoded in order to train the model to output a probability for each class.

C. THE PROPOSED MODEL CONSTRUCTION

Figure 3 shows the construction of the model, where functional API model was built in order to pass different types and dimensionalities of inputs to the model.

As we can see, three separate consecutive dense layers handle the Count Vectors of the opcode sequences. The value in brackets represents the number of neurons, which is a hyperparameter that can be tuned to get the best results.

The fuzzy hash, as it is treated as a sequence, is passed to Gated Recurrent Unit (GRU) layer to learn from the order of the characters in the hashes, then the output is passed to flatten layer to reduce the shape and make it convenient to be passed to a dense layer.

The permissions, services, receivers, and API calls inputs are passed to embedding layers to understand the features and cluster them (where similar words have similar embeddings), followed by flatten layers to make their shape appropriate. Note that we put the preprocessing text vectorization layers into the model, so that we can pass these last mentioned four inputs directly to the model without any processing and as a

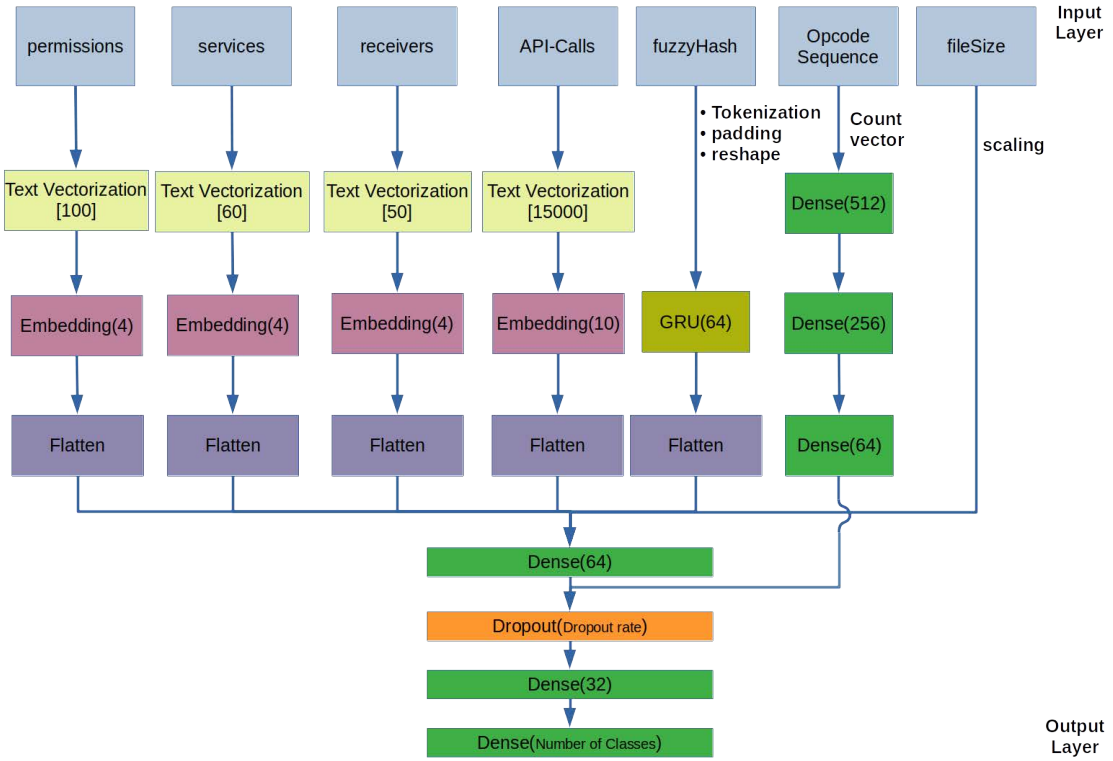

FIGURE 3. The proposed functional API Deep learning model for Android malware classification.

raw data, and this is called then end-to-end model, because it accepts unhandled raw data, but we could not do that for the other features, so the model is partially end-to-end.

The outputs from the flatten layers of the last four mentioned layers, as well as the output of the flatten layer of the fuzzy hash, and the scaled size input, are passed to a dense layer with 64 neurons, with Relu activation, which always outputs positive number, then the output of this layer along with the output of the last dense layer of the opcode sequence network are passed to a dropout layer, which helps preventing the model from overfitting. Then this output is passed to 32 neurons of a dense layer, whose output is passed to a final dense layer with five or two neurons, which is the number of the classes we have, and with Softmax activation, the neuron that has the maximum probability value in the range 0 and +1 will win and be given as the model prediction. With this topology, the total number of trainable parameters of the constructed model is 12,351,695.

D. MODEL TRAINING

During the training of the model, four performance metrics are used; the accuracy, F1 score, Precision, and Recall metric. Our model was trained on two variations of the dataset, the first one is with two target classes; malware and benign, and the other one is with the five clasees of the dataset. In Figures 4 and 5, it is evident that during training, the training rates of the evaluation metrics progressively rise, while the training loss fall. We can see that the model can achieve high accuracy directly after 2 epochs. We can tell

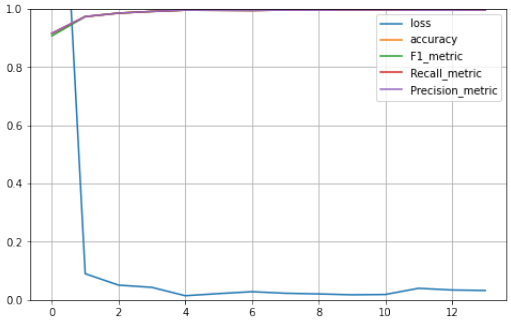

FIGURE 4. Learning curves: the mean training loss and evaluation metrics measured over each epoch with two target classes.

that the model has quite converged to the solution, with no overfitting, as the performance metrics give almost the same values both during training and when evaluating the model on the test set.

During training, 10% of the dataset was used for testing, and this was made to not loose lot of training data, and as we have a high number of samples, so the test set is more than 1400 samples, and this is already bigger than the test sets used in lot of previous papers.

The number of epochs was set to 30 and early stopping callback was used with patience value of 4, and this stops the training when the model does not perform better for 4 epochs. As a result, this clearly stopping callback stopped the training after 16 epochs.

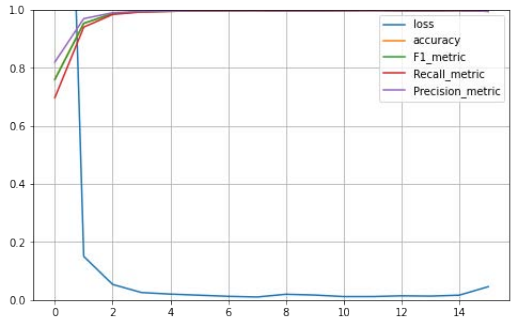

FIGURE 5. Learning curves: the mean training loss and evaluation metrics measured over each epoch with five target classes.

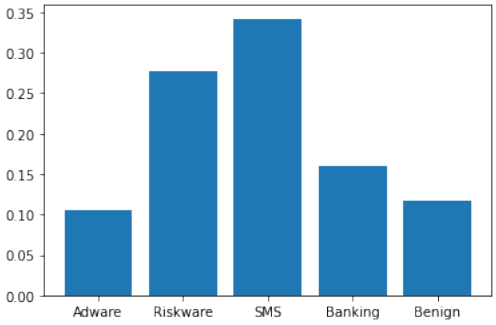

FIGURE 6. The percentage of each class in the five target classes.

We can tell that there is no overfitting from the evaluation results of the test set, which are shown in the next section.

V. RESULTS AND DISCUSSION

In this section, we talk about the characteristics of the dataset we used, then the obtained results are shown. Finally, the used tools for the preparation of the data and the training of the model are given.

A. DATASET

There are many datasets available from different sources. The used dataset in this paper is CICMalDroid 2020, which is provided by the University of New Brunswick and contains 17,341 current and advanced Android samples, categorized as Adware, Banking Malware, SMS Malware, Riskware, and Benign. These samples come from various sources, including VirusTotal [22], Contagio Security Blog, AMD, MalDozer and other datasets used in recent research papers. So, it is the most recent and diverse dataset available today. Full details of the dataset can be found in their research paper [8] and its underlying principles [66].

The total number of applications that were used and got their features extracted are 14079. The percentage of each class can be seen in Figure 6.

In order to be able to make a comparison with the most studies, which use two classes, namely malware and bengin, we also made another variation of the dataset, where we merged the four malware categories into just one, and called it simply "Malware". As a result, we got the two classes as shown with percentage distribution in Figure 7.

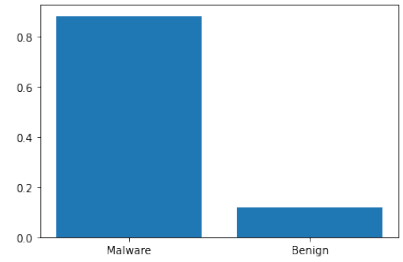

FIGURE 7. The percentage of each class in the two target classes.

TABLE 2. Classification report for our model with just two target classes.

	precision	recall	f1-score	#samples
Benign	0.98	0.97	0.98	157
Malware	1.00	1.00	1.00	1251
accuracy			1.00	1408
macro avg	0.99	0.99	0.99	1408
weighted avg	1.00	1.00	1.00	1408

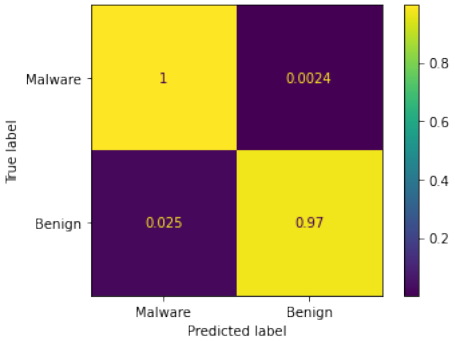

FIGURE 8. Confusion matrix of our deep learning model with two target classes.

B. MODEL EVALUATION

In this section, we evaluate the model with the two variations of the dataset, one with two target classes (malware, benign), which is just for malwre detection, and the other with five target classes, to detect as well as classify android malware.

1) EVALUATING OUR MODEL WITH TWO TARGET CLASSES

Since that the most previous studies classify android samples into just two classes, which are malware and benign, and since that more classes resulting in a reduction in the overall prediction accuracy, as was seen clearly in paper [39], we tested our model on our relabled dataset, where we merged the four malware classes mentiond previously, namely Banking, SMS, Riskware, Adware, into one class which is just malware, and as expected, we obtained significantly higher accuracy.

Table 2 shows the classification report obtained after evaluating our trained model on the independent test set.

Figure 8 shows the confusion matrix of the new trained model evaluated on the test set.

2) EVALUATING OUR MODEL WITH FIVE TARGET CLASSES

At the end of the training, the accuracy values obtained from the performance metrics and for each class are presented in Table 3. We can see that the values are significantly high with

TABLE 3. Classification report for our model with five target classes.

	precision	recall	f1-score	#samples
SMS	0.99	0.99	0.99	462
Benign	0.94	0.97	0.95	181
Banking	0.91	0.92	0.92	241
Adware	0.96	0.95	0.96	129
Riskware	0.97	0.96	0.97	395
accuracy			0.96	1408
macro avg	0.96	0.96	0.96	1408
weighted avg	0.96	0.96	0.96	1408

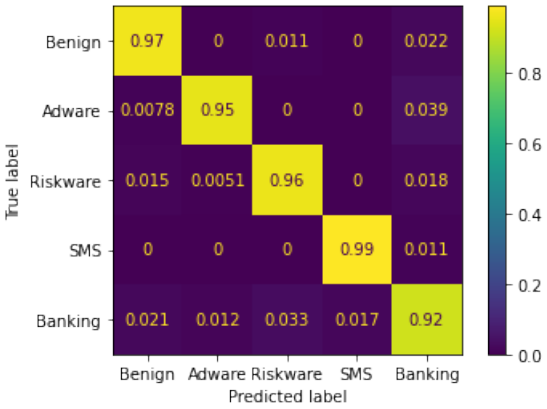

FIGURE 9. Confusion matrix of our deep learning model with five target classes.

a macro average of all classes of 96%, which is the average of the unweighted mean per label. The weighted average is also 96%. The number of the actual occurrences of each class in the given dataset (test dataset) can also be seen in the ''#samples'' column.

The confusion matrix in Figure 9 shows an acceptable and high effectiveness of the model, as we can see that the classes are predicted correctly in the range 92-99%, with the highest classification accuracy for SMS malware class with 99%, and the lowest one for Banking malware class with 92%. The percentages of false classifications are significantly low, with the highest false classification rate of Adware predicted as Banking malware with 3.9%.

C. TOOLS

Static analysis was used in our method, because in dynamic analysis and as discussed previously, the sophisticated malware samples can detect and deceive the virtual system. So, in static analysis we can detect the malware samples before installing and running them in a device, and this is also faster and require less resources, and allow more code coverage.

Androguard [7] was used for feature extraction process, although it is slow in handling big files, but with testing other existing tools, nothing seems to give a better performance. Additionaly, some tools like ClassyShark [20] and ApkStudio [21] are limited to user interface, thus, automation process of feature extraction is not applicable with them. Also, not all Android decompilers allow the extraction of all the features of Android applications, some are limited to decompile dex classes, and others are limited to manifest files.

TABLE 4. Evaluation metrics results with different dropout rates in our model and two target classes.

Dropout rate	Precision	Recall	F1-score	Accuracy
0.2	0.9950	0.9950	0.9950	0.9950
0.5	0.9929	0.9929	0.9929	0.9929
0.8	0.9936	0.9936	0.9936	0.9936
1.0	0.9908	0.9908	0.9908	0.9908

TABLE 5. Evaluation metrics results with different dropout rates in our model and five target classes.

Dropout rate	Precision	Recall	F1-score	Accuracy
0.2	0.9651	0.9616	0.9631	0.9631
0.5	0.9694	0.9673	0.9682	0.9680
0.8	0.9708	0.9695	0.9702	0.9702
1.0	0.9658	0.9624	0.9637	0.9638

Ssdeep tool [26] was used to calculate the fuzzy hashes, and the 'ls' terminal command was used to extract the features from the Android applications. To automate this process, bash script and python are used under Gnu-Linux operating system; Linux Mint 20 distribution, to extract the features from each APK file one by one and build the csv dataset.

For the data preparation and machine learning, Pandas which is used to read the data (the resulted csv files), numpy, sklearn, tenserflow and Keras libraries are used. Matplotlib library is also used to show the results visually in diagrams and confusion matrices.

The process of reading, processing the data, and training the model was carried out under google colab pro online, to get benefit from its powerful shared resources.

VI. TUNING THE MODEL's HYPERPARAMETERS

Although the obtained accuracy from our built model is high, we tried to enhance its accuracy by tuning the hyperparameters such as the optimizer, and the dropout rate, and we also tried to enhance its performance by altering the number of neurons, which in turn alters the number of the trainable parameters. All the results will be shown for both dataset variations (two classes, five classes).

Tables 4 and 5 show a comparison between different dropout rates and their related results of the four used evaluation metrics obtained from the test set. The last row in the table indicates that no dropout was used.

The dropout layer helps in preventing the model from overfitting the training data, by eliminating part of the outputs of the neurons in the previous layer. Dropout rate of 0.8 means that 80% of the data will be passed to the next layer. So, low dropout rate like 0.2 indicates that we loose much data, and this also reduces the model performance as seen in Table 5, while in Table 4, we see that the results of 0.2 and 0.8 dropout rates are almost same. Thus, the best experimented dropout rate according to the overall evaluation results is 0.8. So, we adopted this hyperparameter.

We tried then to reduce the number of trainable parameters by eliminating the two dense layers that have 64 neurons, which are shown in green color in the third and fourth hidden layer in Figure 3. As a result, the number of trainable

TABLE 6. Evaluation metrics results for the our model variations with two target classes.

	#Trainable parameters	Precision	Recall	F1-score	Accuracy
Our Basic Model	12,351,695	0.9950	0.9950	0.9950	0.9950
Variation A	7,510,319	0.9901	0.9901	0.9901	0.9901
Variation B	22,042,639	0.9950	0.9950	0.9950	0.9950
Variation C	12,352,767	0.9915	0.9915	0.9915	0.9915

TABLE 7. Evaluation metrics results for the our model variations with five target classes.

	#Trainable parameters	Precision	Recall	F1-score	Accuracy
Our Basic Model	12,351,695	0.9708	0.9695	0.9702	0.9702
Variation A	7,510,319	0.0.9679	0.9679	0.9651	0.9652
Variation B	22,042,639	0.9708	0.9688	0.9702	0.9702
Variation C	12,352,767	0.9658	0.9616	0.9624	0.9624

parameters decreased from 12,351,695 to 7,510,319, we call this "Variation A" of our model.

Another variation was tested by adding two dense layers with 128 neurons, one before each of those two dense layers with 64 neurons that was mentioned before, and by keeping those last two layers, the number of trainable parameters was increased to 22,042,639. Increasing the trainable parameters cause the model to overfit the training dataset with reduced prediction accuracy on new data, and this can be seen in the results, with the name "Variation B".

Lastly, with the same model shown in Figure 3 (without adding or deleting any layer), we wanted to increase the trainable parameters a little bit, to not cause overfitting like in the last variation, so we just increased the number of neurons in the last hidden layer (the dense layer), from 32 to 40, which increases the number of trainable parameters slightly to just 12,352,767. We call this "Variation C".

Tables 6 and 7 show a comparison between the evaluation metrics' results for our basic model structure and its tested variations.

As it is clearly observed, although reducing the trainable parameters reduces the training time of the model, but it also reduces its effectiveness. Also, increasing the trainable parameters does not help the model to perform better, as we see with Variation B and C.

We also did other experiments by tweaking the number of neurons in other layers, but all the obtained results were below the first previously obtained one.

Finally, we tried different optimizers, and the obtained results are shown in Tables 8 and 9. Note that the Adagrad optimizer required more training time (19 epochs) and its results were the lowest, where Adamax optimizer took the longest time (28 epochs).

We notice that adamax optimizer gave the same results as adam optimizer, but it takes lot more training time, as it required the 30 epochs to reach that accuracy.

TABLE 8. Evaluation metrics results for our model with different optimizers and two target classes.

Optimizer	Precision	Recall	F1-score	Accuracy
Adam	0.9950	0.9950	0.9950	0.9950
Nadam	0.9908	0.9908	0.9908	0.9908
Adagrad	0.9759	0.9759	0.9759	0.9759
Adamax	0.9950	0.9950	0.9950	0.9950

TABLE 9. Evaluation metrics results for our model with different optimizers and five target classes.

Optimizer	Precision	Recall	F1-score	Accuracy
Adam	0.9708	0.9695	0.9702	0.9702
Nadam	0.9642	0.9574	0.9602	0.9602
Adagrad	0.8834	0.6939	0.8088	0.8125
Adamax	0.9665	0.9638	0.9654	0.9652

TABLE 10. Overall accuracy of each tested classifier with only the first four features and two target classes.

Classifier	Overall Accuracy
SGD Classifier	0.821
DecisionTree Classifier	0.943
Random Forest Classifier	0.973
SVC	0.952
GaussianNB	0.118
XGBoost	0.973
KNN	0.955

VII. EXPERIMENTAL EVALUATION OF OTHER MODELS

In this section, we test different classifiers that are mostly used in previous studies, and compare them with our model, we also make a comparison between our model and other models in new studies that use the same dataset we used.

A. EXPERIMENTAL EVALUATION OF OTHER CLASSIFIERS

In order to be able to make a comparison between our proposed model and the models that were used in previous works, the most used models in previous works were tested on our obtained dataset. The classifiers that were tested are Stochastic Gradient Descent (SGD), Decision Tree, Random Forest, C-Support Vector Classifier (SVC), K-Nearest Neighbor (KNN), XGBoost, and Gaussian Naive Bayes (GaussianNB).

All the results will be shown for both dataset variations (two classes, five classes).

At the beginning, only the first four features were passed to the classifiers, namely permissions, receivers, services and API calls. The reason for this is that these features were the most commonly used in previous works, so we wanted to test whether the additional features we used provide better results or not. Tables 10 and 11 show the accuracy of the tested Classifiers with only these four features.

We passed then all the extracted features to these classifiers and re-checked the results, which are shown in Tables 12 and 13. It is obvious, according to the averaged accuracy of all classifiers, that using all the features we extracted gives better accuracy than using just the most commonly used features in previous papers.

Using all the features, and in both variations of two and five classes, it can be clearly seen that the performance of the best

TABLE 11. Overall accuracy of each tested classifier with only the first four features and five target classes.

Classifier	Overall Accuracy
SGD Classifier	0.522
DecisionTree Classifier	0.890
Random Forest Classifier	0.935
SVC	0.757
GaussianNB	0.370
XGBoost	0.907
KNN	0.860

TABLE 12. Overall accuracy of each tested classifier with all features and two target classes.

Classifier	Overall Accuracy
SGD Classifier	0.899
DecisionTree Classifier	0.974
Random Forest Classifier	0.975
SVC	0.952
GaussianNB	0.118
XGBoost	0.983
KNN	0.954
Our proposed Model	**0.950**

TABLE 13. Overall accuracy of each tested classifier with all features and five target classes.

Classifier	Overall Accuracy
SGD Classifier	0.841
DecisionTree Classifier	0.907
Random Forest Classifier	0.934
SVC	0.701
GaussianNB	0.504
XGBoost	0.924
KNN	0.859
Our proposed Model	**0.970**

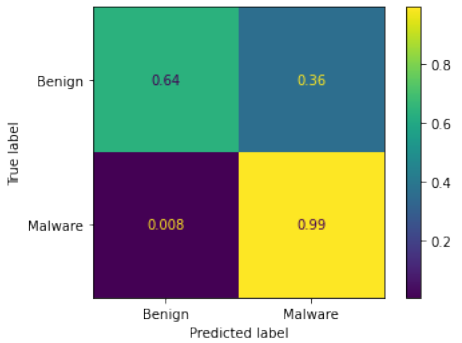

FIGURE 10. Confusion matrix of SVC classifier with two target classes.

classifier was lower than the performance of our proposed model. We also see clearly and as expected, that reducing the number of classes, the results become significantly better.

In the next sections, the results of each tested classifier are discussed in detail (with the use of all features).

1) EVALUATION OF SVC CLASSIFIER

Figure 10 shows the confusion matrix of SVC classifier, which was trained just to predict two classes; malware and benign. We can see that in this case, the classifier gives high prediction accuracy for malware class, but also 36% of benign samples are being classified as malware.

FIGURE 11. Confusion matrix of SVC classifier with five target classes.

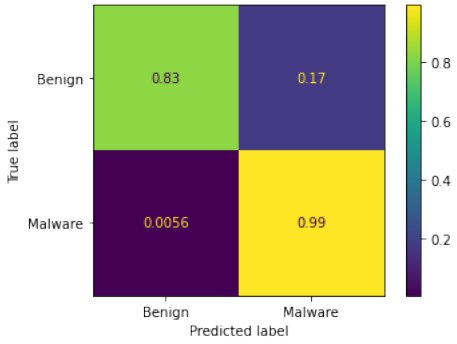

FIGURE 12. Confusion matrix of RandomForest classifier with two target classes.

In the case of five target classes, we see from the confusion matrix in Figure 11, that 4.4% of the benign class samples were incorrectly classified as adware and 1.7% of them was classified as riskware. As we most care about the classification of the benign class against malware, the accuracy of classification for benign class was 88% in the confusion matrix, which is not so high.

2) EVALUATION OF RANDOM FOREST CLASSIFIER

In Figure 12, and in case of two target classes, we see that 17% of benign samples are being classified as malware, which is a lower false negative rate than the previous classifier, while the prediction accuracy for malware class is still 99%.

With the five target classes, the true classification rate of the benign class was not better from SVC classifier, as the confusion matrix in Figure 13 shows the same rate for benign class with 88%, and the class that is most wrongly classified as benign is the adware class with 7.7%, which is relatively high false classification rate.

3) EVALUATION OF SGD CLASSIFIER

With two target classes, Figure 14 shows very low classification accuracy for benign class with just 20%.

Till now, Stochastic Gradient Descent classifier's rates were in the middle between SVC and Random Forest in the case of five target classes. The confusion matrix in Figure 15

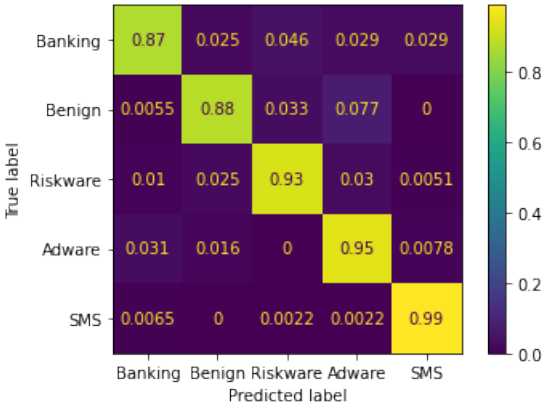

FIGURE 13. Confusion matrix of RandomForest classifier with five target classes.

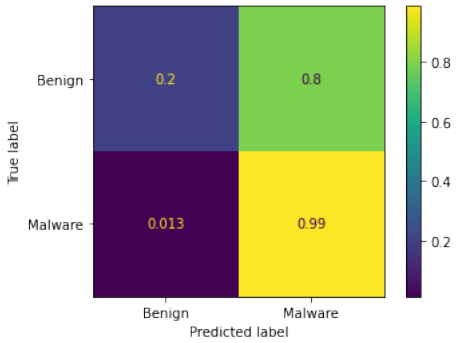

FIGURE 14. Confusion matrix of SGD classifier with two target classes.

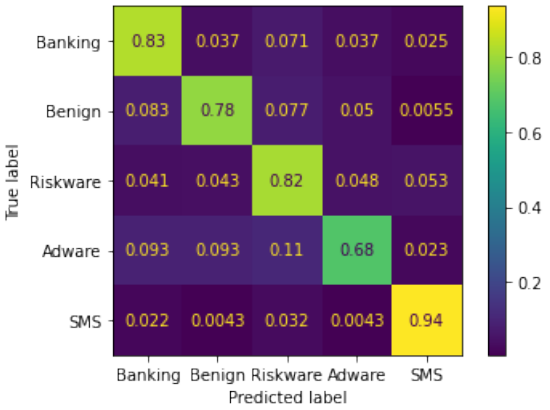

FIGURE 15. Confusion matrix of SGD classifier with five target classes.

shows that the benign class was mostly wrong classified as banking with the percentage of 8.3%.

4) EVALUATION OF DECISION TREE CLASSIFIER

The confusion matrix for two target classes in Figure 16 shows that Decision Tree Classifier gives in this case high prediction accuracy for both malware and benign classes, and this is the best classifier in case of two classes, and compared to the previously shown classifiers till now.

The confusion matrix for five target classes in Figure 17 shows that the benign samples were mostly classified as

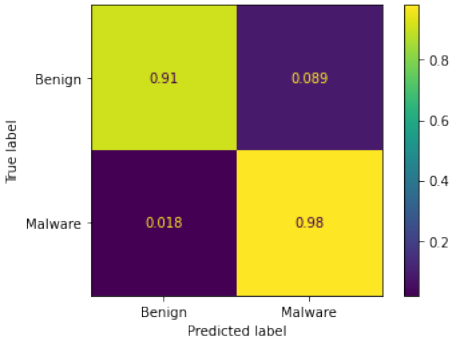

FIGURE 16. Confusion matrix of Decision Tree classifier with two target classes.

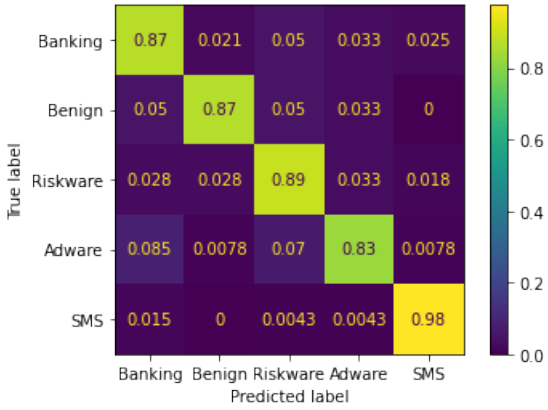

FIGURE 17. Confusion matrix of Decision Tree classifier with five target classes.

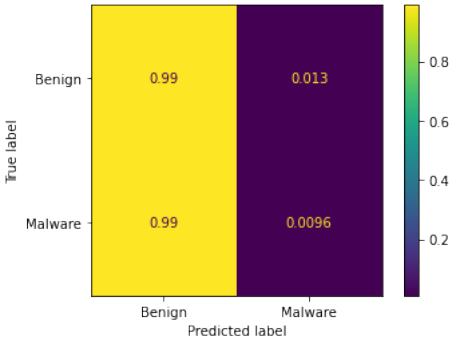

FIGURE 18. Confusion matrix of GaussianNB classifier with target classes.

banking and riskware with 5% both, and never wrongly classified as SMS malware, as the SMS malware class has the highest true classification rate of 98%.

5) EVALUATION OF GaussianNB CLASSIFIER

Figure 18 shows that in the case of a two target classes, the Gaussian Naive Bayes classifier classifies almost all samples as benign, in spite of the fact that the benign samples are a lot less than the malware samples.

With five target classes, Gaussian Naive Bayes classifier gave the lowest results among the tested classifiers. The confusion matrix in Figure 19 shows that the percentage of the true classification of benign class (48%) is lower than

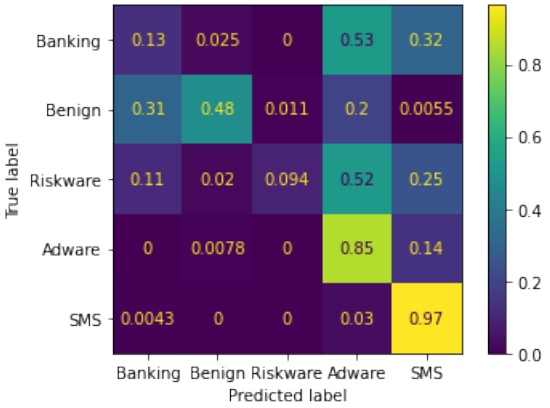

FIGURE 19. Confusion matrix of GaussianNB classifier with five target classes.

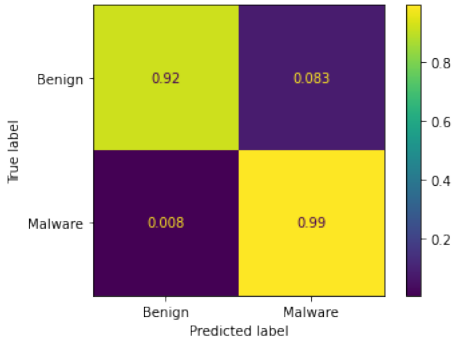

FIGURE 20. Confusion matrix of XGBoost classifier with two target classes.

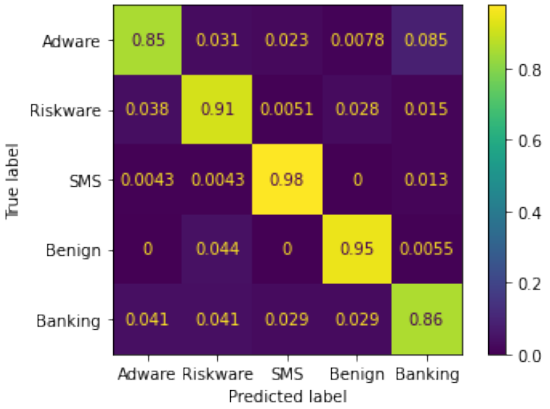

FIGURE 21. Confusion matrix of XGBoost classifier with five target classes.

its wrong classification as SMS malware with 55%. and this is the only tested classifier that gave higher rate of false classification than the rate of true classification.

6) EVALUATION OF XGBoost CLASSIFIER

In the case of two target classes, XGBoost Classifier performs really well, with prediction accuracy of 99% for malware class, and 92% for benign class, as seen in Figure 20.

In the case of five target classes, the confusion matrix in Figure 21, shows that XGBoost Classifier has an overall good performance, with the lowest accuracy of 85% for adware class, and highest one of 98% for SMS class.

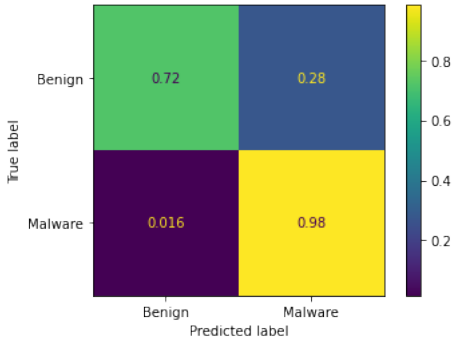

FIGURE 22. Confusion matrix of KNN classifier with two target classes.

FIGURE 23. Confusion matrix of KNN classifier with five target classes.

TABLE 14. Overall comparison of our proposed model and the tested classifiers with two target classes.

class	classifier	precision	recall	f1-score	#Samples
Benign	SVC	0.91	0.64	0.75	157
	SGD	0.67	0.20	0.31	
	Random Forest	0.95	0.83	0.88	
	GaussianNB	0.11	0.99	0.20	
	Decision Tree	0.87	0.91	0.89	
	XGBoost	0.94	0.92	0.93	
	KNN	0.85	0.72	0.78	
	Our proposed Model	**0.98**	**0.97**	**0.98**	
Malware	SVC	0.96	0.99	0.97	1251
	SGD	0.91	0.99	0.95	
	Random Forest	0.98	0.99	0.99	
	GaussianNB	0.86	0.01	0.02	
	Decision Tree	0.99	0.98	0.99	
	XGBoost	0.99	0.99	0.99	
	KNN	0.97	0.98	0.97	
	Our proposed Model	**1.00**	**1.00**	**1.00**	
macro avg	SVC	0.93	0.81	0.86	1408
	SGD	0.79	0.60	0.63	
	Random Forest	0.96	0.91	0.94	
	GaussianNB	0.48	0.5	0.11	
	Decision Tree	0.93	0.95	0.94	
	XGBoost	0.96	0.95	0.96	
	KNN	0.91	0.85	0.88	
	Our proposed Model	**0.99**	**0.99**	**0.99**	
weighted avg	SVC	0.95	0.95	0.95	1408
	SGD	0.88	0.90	0.88	
	Random Forest	0.98	0.98	0.98	
	GaussianNB	0.77	0.12	0.04	
	Decision Tree	0.98	0.97	0.97	
	XGBoost	0.98	0.98	0.98	
	KNN	0.95	0.95	0.95	
	Our proposed Model	**1.00**	**1.00**	**1.00**	

TABLE 15. Overall comparison of our proposed model and the tested classifiers with five target classes.

class	classifier	precision	recall	f1-score	#Samples
SMS	SVC	0.63	0.97	0.77	462
	SGD	0.93	0.94	0.94	
	Random Forest	0.98	0.99	0.98	
	GaussianNB	0.7	0.97	0.81	
	Decision Tree	0.97	0.98	0.97	
	XGBoost	0.97	0.98	0.98	
	KNN	0.94	0.96	0.95	
	Our proposed Model	**0.99**	**0.99**	**0.99**	
Benign	SVC	0.83	0.88	0.85	181
	SGD	0.78	0.78	0.78	
	Random Forest	0.9	0.88	0.89	
	GaussianNB	0.85	0.48	0.61	
	Decision Tree	0.9	0.87	0.88	
	XGBoost	0.90	0.95	0.92	
	KNN	0.79	0.75	0.77	
	Our proposed Model	**0.91**	**0.98**	**0.94**	
Banking	SVC	0.74	0.56	0.64	241
	SGD	0.79	0.83	0.81	
	Random Forest	0.95	0.87	0.91	
	GaussianNB	0.24	0.13	0.17	
	Decision Tree	0.85	0.87	0.86	
	XGBoost	0.90	0.86	0.88	
	KNN	0.79	0.88	0.83	
	Our proposed Model	**0.95**	**0.91**	**0.93**	
Adware	SVC	0.6	0.64	0.62	129
	SGD	0.69	0.68	0.69	
	Random Forest	0.78	0.95	0.86	
	GaussianNB	0.22	0.85	0.35	
	Decision Tree	0.79	0.83	0.81	
	XGBoost	0.80	0.85	0.83	
	KNN	0.72	0.68	0.70	
	Our proposed Model	**0.96**	**0.97**	**0.97**	
Riskware	SVC	0.88	0.41	0.56	395
	SGD	0.84	0.82	0.83	
	Random Forest	0.95	0.93	0.94	
	GaussianNB	0.95	0.09	0.17	
	Decision Tree	0.92	0.89	0.91	
	XGBoost	0.94	0.91	0.93	
	KNN	0.89	0.84	0.86	
	Our proposed Model	**0.97**	**0.97**	**0.97**	
macro avg	SVC	0.74	0.69	0.69	1408
	SGD	0.81	0.81	0.81	
	Random Forest	0.91	0.92	0.92	
	GaussianNB	0.59	0.5	0.42	
	Decision Tree	0.88	0.89	0.89	
	XGBoost	0.90	0.91	0.91	
	KNN	0.83	0.82	0.82	
	Our proposed Model	**0.96**	**0.96**	**0.96**	
weighted avg	SVC	0.74	0.7	0.68	1408
	SGD	0.84	0.84	0.84	
	Random Forest	0.94	0.93	0.94	
	GaussianNB	0.67	0.5	0.45	
	Decision Tree	0.91	0.91	0.91	
	XGBoost	0.93	0.92	0.92	
	KNN	0.86	0.86	0.86	
	Our proposed Model	**0.97**	**0.97**	**0.97**	

7) EVALUATION OF KNN CLASSIFIER

In the case of two target classes, the confusion matrix of KNN Classifier shows a prediction accuracy of 98% for malware class, and 72% for benign class, as seen clearly in Figure 22.

In the case of five target classes, K-Nearest Neighbor classifier gave relatively a good prediction accuracy. We can see from the confusion matrix in Figure 23 that Adware class has the lowest percentage of the true classification of 68%, while SMS malware has the highest true classification rate with 96%. The true classification accuracy of benign class is 75%.

TABLE 16. Experimental Comparison between our model and related works with two target classes.

Study	Precision	Recall	F1-score	Accuracy
[41]	0.9816	0.9818	0.9815	0.9818
[42]	0.9918	0.9860	0.9889	0.9833
[43]	0.9840	0.9885	0.9862	0.9787
Ours	**0.9950**	**0.9950**	**0.9950**	**0.9950**

Based on the results of these classifiers and the results of our proposed model, and in case of five target classes, we can see that the banking and adware classes have the lowest classification accuracy, which means that the samples used were insufficient and more samples are needed for these classes to improve the classification accuracy.

8) EXPERIMENTAL COMPARISON WITH THE PROPOSED MODEL

In Tables 14 and 15, we see an overall comparison between all the classifiers and our proposed model for both two classes and five classes cases respectively. It is clearly observed, that our proposed model always gives the highest accuracy among the tested classifiers with all metrics and for all classes.

B. EXPERIMENTAL COMPARISON WITH RELATED WORKS

In this section, we show a comparison between our model and the studies that use the same dataset that we used, considering just the two target classes case, which are namely malware and benign, as these studies concentrate just on these two classes for malware detection problem, without giving importance to malware classification issue.

As we see in Table 16, our model performs better than the few existing new studies that use the same dataset.

VIII. CONCLUSION

Because of the rapidly increasing number of malware samples that target the android operating system with most new and sophisticated techniques, lot of studies were performed and published in order to try to develop a tool or system to automatically detect Android malware. Most of these studies do not cover all Android applications' features and information, and there are lot of them with accuracy below 90%, with the others' performance questioned for high accuracy with new samples.

In this paper, we proposed a new method for Android malware detection, where we built and trained a new functional API deep learning model that handles each feature we used individually, as we combined lot of features with different types and dimensionalities. We extracted the most useful features we observed, which are namely the permissions, API calls, services, broadcast receivers, and opcode sequences, all of them were combined in one study and one model, and to the best of our knowledge, there are very few studies that use all of these features together. Additionally, we proposed completely two new static features, namely application size and fuzzy hash. The former is a strong marker that can indicate the malware, as the observed big difference between the size of benign samples, which is generally becoming bigger

than 10 MB, and the size of malware samples which is less than 5 MB in general. The fuzzy hash is also an important added feature, that is used for similarity detection, which is in turn a very essential feature to be tackled in this subject.

We used the most recent and diverse dataset, CICMalDroid 2020, to train our model, where we extracted the mentioned features from 14079 samples that belong to 5 classes, which is an added advantage, as the model not just detect the malware, but also predict its class from among four classes, namely adware, banking malware, SMS malware, and riskware.

To compare our work with other studies, which have the limitation of concentrating on malware detection without considering the malware classification problem, we also trained our model and performed all other experiments on another variation of the dataset, where we merged the malware classes into one ''Malware'' class, to have only two target classes; malware and benign, like the most related works.

Using four metrics to evaluate the performance of our model, namely F1 score, recall, precision, and accuracy metric, all of them gave more than 96% accuracy, which is significantly high, and this is believed to be the result of the combination of the used features in previous studies, along with our new proposed ones, which are the file size and fuzzy hash.

We also tested some algorithms that are used in previous papers which are random forest, Stochastic Gradient Descent, Gaussian Naive Bayes, C-Support Vector, K-Nearest Neighbor, XGBoost, and decision tree classifier. The experimental results show that our model overcomes all tested classifiers with all the evaluation metrics.

REFERENCES

[1] *Operating System Market Share Worldwide Aug. 2020—Sep. 2021*. Accessed: Sep. 22, 2022. [Online]. Available: https://gs.statcounter.com/os-market-share#monthly-202009-202109

[2] S. Avinash. (Jul. 26, 2022). *Top Google Play Store Statistics 2022 You Must Know*. Accessed: Sep. 22, 2022. [Online]. Available: https://appinventiv.com/blog/google-play-store-statistics/

[3] T. Kimberly, ''The evolution of Android malware and Android analysis techniques,'' *ACM Comput. Surv.*, vol. 49, no. 4 pp. 1–41, 2017.

[4] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, ''A survey of Android malware detection with deep neural models,'' *ACM Comput. Surveys*, vol. 53, no. 6, pp. 1–36, Nov. 2021.

[5] S. Liu, G. Lin, Q.-L. Han, S. Wen, J. Zhang, and Y. Xiang, ''DeepBalance: Deep-learning and fuzzy oversampling for vulnerability detection,'' *IEEE Trans. Fuzzy Syst.*, vol. 28, no. 7, pp. 1329–1343, Jul. 2020.

[6] J. Senanayake, H. Kalutarage, and M. O. Al-Kadri, ''Android mobile malware detection using machine learning: A systematic review,'' *Electronics*, vol. 10, no. 13, p. 1606, Jul. 2021.

[7] *Androguard: A Full Python Tool to Play With Android Files*. Accessed: Sep. 22, 2022. [Online]. Available: https://androguard.readthedocs.io/en/latest/

[8] S. Mahdavifar, A. F. A. Kadir, R. Fatemi, D. Alhadidi, and A. Ali, ''Dynamic Android malware category classification using semi-supervised deep learning,'' in *Proc. 18th IEEE Int. Conf. Dependable, Autonomic, Secure Comput. (DASC)*, Aug. 2020, pp. 515–522.

[9] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, ''MalDozer: Automatic framework for Android malware detection using deep learning,'' *Digit. Invest.*, vol. 24, pp. 48–59, Mar. 2018.

[10] (Jul. 19, 2022). *Android Statistics > Number of Android Apps on Google Play*. Accessed: Sep. 22, 2022. [Online]. Available: https://www.appbrain.com/stats/number-of-android-apps

[11] B. Peter and P. Crowley, *Modern Embedded Computing: Designing Connected, Pervasive*. Amsterdam, The Netherlands: Elsevier, 2012.

[12] *Services Overview*. Accessed: Sep. 22, 2022. [Online]. Available: https://developer.Android.com/guide/components/services

[13] *ProGuard*. Accessed: Sep. 22, 2022. [Online]. Available: https://stuff.mit.edu/afs/sipb/project/android/sdk/android-sdk-linux/tools/proguard/docs/index.html

[14] *DexGuard*. Accessed: Jul. 26, 2022. [Online]. Available: https://www.guardsquare.com/dexguard

[15] *ApkProtect*. Accessed: Jul. 26, 2022. [Online]. Available: https://github.com/ijiami/ApkProtect

[16] *The second generation Android Hardening Protection*. Accessed: Sep. 22, 2022. [Online]. Available: https://github.com/woxihuannisja/Bangcle

[17] *Libre and Portable Reverse Engineering Framework*. Accessed: Sep. 22, 2022. [Online]. Available: https://rada.re/n/

[18] *Tools to Work With Android.Dex and Java.Class Files*. Accessed: Sep. 22, 2022. [Online]. Available: https://github.com/pxb1988/dex2jar

[19] *Dex to Java Decompiler*. Accessed: Sep. 22, 2022. [Online]. Available: https://github.com/skylot/jadx

[20] *Android-ClassyShark- Android and Java Bytecode Viewer*. Accessed: Sep. 22, 2022. [Online]. Available: https://github.com/google/Android-classyshark

[21] Apkstudio. *Open-Source, Cross Platform Qt Based IDE for Reverse-Engineering Android Application Packages*. Accessed: Sep. 22, 2022. [Online]. Available: https://github.com/vaibhavpandeyvpz/apkstudio

[22] *VirusTotal*. Accessed: Sep. 22, 2022. [Online]. Available: https://www.virustotal.com

[23] *Droidbox*. Accessed: Sep. 22, 2022. [Online]. Available: https://github.com/pjlantz/droidbox

[24] *AppsPlayground*. Accessed: Sep. 22, 2022. [Online]. Available: http://list.cs.northwestern.edu/mobile/

[25] *SandDroid*. Accessed: Sep. 22, 2022. [Online]. Available: http://sanddroid.xjtu.edu.cn

[26] J. Kornblum, ''Identifying almost identical files using context triggered piecewise hashing,'' *Digit. Invest.*, vol. 3, pp. 91–97, Sep. 2006.

[27] M. Niall, ''Deep Android malware detection,'' in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, 2017, pp. 301–308.

[28] D. Li, L. Zhao, Q. Cheng, N. Lu, and W. Shi, ''Opcode sequence analysis of Android malware by a convolutional neural network,'' *Concurrency Comput., Pract. Exper.*, vol. 32, no. 18, Sep. 2020.

[29] S. Y. Yerima, S. Sezer, and I. Muttik, ''Android malware detection using parallel machine learning classifiers,'' in *Proc. 8th Int. Conf. Next Gener. Mobile Apps, Services Technol.*, Sep. 2014, pp. 37–42.

[30] S. Turker and A. B. Can, ''AndMFC: Android malware family classification framework,'' in *Proc. IEEE 30th Int. Symp. Pers., Indoor Mobile Radio Commun. (PIMRC Workshops)*, Sep. 2019, pp. 1–6.

[31] W. Li, J. Ge, and G. Dai, ''Detecting malware for Android platform: An SVM-based approach,'' in *Proc. IEEE 2nd Int. Conf. Cyber Secur. Cloud Comput.*, Nov. 2015, pp. 464–469.

[32] A. Altaher and O. Barukab, ''Android malware classification based on ANFIS with fuzzy C-means clustering using significant application permissions,'' *TURKISH J. Electr. Eng. Comput. Sci.*, vol. 25, no. 3, pp. 2232–2242, 2017.

[33] Z. Aung and W. Zaw, ''Permission-based Android malware detection,'' *Int. J. Sci. Technol. Res.*, vol. 2, no. 3, pp. 228–234, 2013.

[34] S. Borja, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Alvarez, ''Puma: Permission usage to detect malware in Android,'' in *Proc. Int. Joint Conf. (CISIS)*. Berlin, Germany: Springer, 2013, pp. 289–298.

[35] S. B. Almin and M. Chatterjee, ''A novel approach to detect Android malware,'' *Proc. Comput. Sci.*, vol. 45, pp. 407–417, Dec. 2015.

[36] B. Tahtaci and B. Canbay, ''Android malware detection using machine learning,'' in *Proc. Innov. Intell. Syst. Appl. Conf. (ASYU)*, Oct. 2020, pp. 1–6.

[37] J. Lee, H. Jang, S. Ha, and Y. Yoon, ''Android malware detection using machine learning with feature selection based on the genetic algorithm,'' *Mathematics*, vol. 9, no. 21, p. 2813, 2021.

[38] V. Sihag, M. Vardhan, and P. Singh, ''BLADE: Robust malware detection against obfuscation in android,'' *Forensic Sci. Int., Digit. Invest.*, vol. 38, Sep. 2021, Art. no. 301176.

[39] D. Dang, F. Di Troia, and M. Stamp, ''Malware classification using long short-term memory models,'' 2021, *arXiv:2103.02746*.

[40] J. Kim, Y. Ban, E. Ko, H. Cho, and J. H. Yi, "MAPAS: A practical deep learning-based Android malware detection system," *Int. J. Inf. Secur.*, vol. 21, no. 4, pp. 725–738, Aug. 2022.
[41] P. Musikawan, Y. Kongsorot, I. You, and C. So-In, "An enhanced deep learning neural network for the detection and identification of Android malware," *IEEE Internet Things J.*, early access, Jul. 29, 2022, doi: 10.1109/JIOT.2022.3194881.
[42] R. Yumlembam, B. Issac, S. M. Jacob, and L. Yang, "IoT-based Android malware detection using graph neural network with adversarial defense," *IEEE Internet Things J.*, early access, Jul. 5, 2022, doi: 10.1109/JIOT.2022.3188583.
[43] P. Tarwireyi, A. Terzoli, and M. O. Adigun, "BarkDroid: Android malware detection using bark frequency Cepstral coefficients," *Indonesian J. Inf. Syst.*, vol. 5, no. 1, pp. 48–63, 2022.
[44] I. Almomani, R. Qaddoura, M. Habib, S. Alsoghyer, A. A. Khayer, I. Aljarah, and H. Faris, "Android ransomware detection based on a hybrid evolutionary approach in the context of highly imbalanced data," *IEEE Access*, vol. 9, pp. 57674–57691, 2021.
[45] Z. Wang, J. Cai, S. Cheng, and W. Li, "DroidDeepLearner: Identifying Android malware using deep learning," in *Proc. IEEE 37th Sarnoff Symp.*, Sep. 2016, pp. 160–165.
[46] X. Su, D. Zhang, W. Li, and K. Zhao, "A deep learning approach to Android malware feature learning and detection," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, Aug. 2016, pp. 244–251.
[47] X. Wang, D. Zhang, X. Su, and W. Li, "Mlifdect: Android malware detection based on parallel machine learning and information fusion," *Secur. Commun. Netw.*, vol. 2017, pp. 1–14, Aug. 2017.
[48] Y. Rosmansyah and B. Dabarsyah, "Malware detection on Android smartphones using API class and machine learning," in *Proc. Int. Conf. Electr. Eng. Informat. (ICEEI)*, Aug. 2015, pp. 294–297.
[49] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. E. R. T. Siemens, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*. vol. 14, 2014, pp. 1–15.
[50] 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 Security*, vol. 14, no. 3, pp. 773–788, Aug. 2018.
[51] R. Sato, D. Chiba, and S. Goto, "Detecting Android malware by analyzing manifest files," in *Proc. Asia–Pacific Adv. Netw.*, vol. 36, Dec. 2013, p. 17.
[52] J. Sahs and L. Khan, "A machine learning approach to Android malware detection," in *Proc. Eur. Intell. Secur. Informat. Conf.*, Aug. 2012, pp. 141–147.
[53] H. Kang, J.-W. Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying Android malware using static analysis along with creator information," *Int. J. Distrib. Sensor Netw.*, vol. 11, no. 6, 2015, Art. no. 479174.
[54] S. Shakya and M. Dave, "Analysis, detection, and classification of Android malware using system calls," 2022, *arXiv:2208.06130*.
[55] A. H. El Fiky, M. A. Madkour, and A. El Shenawy, "Android malware category and family identification using parallel machine learning," *J. Inf. Technol. Manag.* vol. 14, no. 4, pp. 19–39, 2022.
[56] J.-W. Jang, J. Yun, A. Mohaisen, J. Woo, and H. K. Kim, "Detecting and classifying method based on similarity matching of Android malware behavior with profile," *SpringerPlus*, vol. 5, no. 1, p. 273, 2016.
[57] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Detecting Android malware using sequences of system calls," in *Proc. 3rd Int. Workshop Softw. Develop. Lifecycle Mobile*, Aug. 2015, pp. 13–20.
[58] T. Bhatia and R. Kaushal, "Malware detection in Android based on dynamic analysis," in *Proc. Int. Conf. Cyber Secur. Protection Digit. Services (Cyber Security)*, Jun. 2017, pp. 1–6.
[59] M. S. Alam and S. T. Vuong, "Random forest classification for detecting Android malware," in *Proc. IEEE Int. Conf. Green Comput. Commun. IEEE Internet Things IEEE Cyber, Phys. Social Comput.*, Aug. 2013, pp. 663–669.
[60] A. T. Kabakus and I. A. Dogru, "An in-depth analysis of Android malware using hybrid techniques," *Digit. Invest.*, vol. 24, pp. 25–33, Mar. 2018.
[61] 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.
[62] F. Tong and Z. Yan, "A hybrid approach of mobile malware detection in Android," *J. Parallel Distrib. Comput.*, vol. 103, pp. 22–31, May 2017.
[63] A. Martín, R. Lara-Cabrera, and D. Camacho, "Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset," *Inf. Fusion*, vol. 52, pp. 128–142, Dec. 2019.
[64] R. B. Hadiprakoso, H. Kabetta, and I. K. S. Buana, "Hybrid-based malware analysis for effective and efficiency Android malware detection," in *Proc. Int. Conf. Informat., Multimedia, Cyber Inf. Syst. (ICIMCIS)*, Nov. 2020, pp. 8–12.
[65] R. Surendran, T. Thomas, and S. Emmanuel, "A TAN based hybrid model for Android malware detection," *J. Inf. Secur. Appl.*, vol. 54, Oct. 2020, Art. no. 102483.
[66] S. Mahdavifar, D. Alhadidi, and A. A. Ghorbani, "Effective and efficient hybrid Android malware classification using pseudo-label stacked auto-encoder," *J. Netw. Syst. Manag.*, vol. 30, no. 1, pp. 1–34, Jan. 2022.

MÜLHEM İBRAHIM was born in Aleppo, Syria, in 1994. He received the bachelor's degree in computer engineering from Turkish-German University, Istanbul, Turkey, in 2022. He did an internship in Frontend Development with Infina Software Company, Istanbul, in Summer 2019. He also made an internship in backend and frontend web development and testing with Sunway Construction Company, Malaysia, in Summer 2021. He was granted a scholarship from DAAD organization for his bachelor's study.

BAYAN ISSA received the bachelor's degree in informatics engineering from the University of Aleppo, where she is currently pursuing the master's degree in artificial intelligence. She worked as a Machine Learning Engineer with experience building, training and testing modern models, particularly in the area of natural language processing and image processing.

MUHAMMED BASHEER JASSER (Member, IEEE) received the master's and Ph.D. degrees in software engineering from University Putra Malaysia (UPM). He is currently a Senior Lecturer and the Program Leader of the B.Sc. degree (Hons.) in information technology with the Department of Computing and Information Systems, School of Engineering and Technology, Sunway University. He was granted the Malaysian Technical Cooperation Program Scholarship (MTCP) from the Ministry of Higher Education (Malaysia) for his postgraduate studies. His research interests include optimization algorithms, evolutionary computation, model-driven software engineering, formal specification, verification and theorem proving, artificial intelligence, and machine learning. He is also working on several fundamental and industrial research projects in the area of artificial intelligence and software engineering funded by several companies and universities. Several postgraduate students are working under his supervision on these projects. He is also a member of several professional academic bodies, including the Institute of Electronics, Information and Communication Engineers (IEICE), and Formal Methods Europe Organization.