applied sciences

MDPI

Article

Visualising Static Features and Classifying Android Malware Using a Convolutional Neural Network Approach

Ömer Kiraz * and İbrahim Alper Doğru *

Department of Computer Engineering, Faculty of Technology, Gazi University, Ankara 06560, Turkey
* Correspondence: omer.kiraz1@gazi.edu.tr (Ö.K.); iadogru@gazi.edu.tr (İ.A.D.)

Abstract: Android phones are widely recognised as the most popular mobile phone operating system. Additionally, tasks like browsing the internet, taking pictures, making calls, and sending messages may be completed with ease in daily life because of the functionality that Android phones offer. The number of situations in which users are harmed by unauthorised access to data emerging from these processes is growing daily. Because the Android operating system is open source and generated applications are not thoroughly reviewed before being released onto the market, this scenario has been the primary focus of hackers. Therefore, technologies to distinguish between malware and benign Android applications are required. CNN-based techniques are proven to produce important and successful outcomes when applied to Android malware detection on images. The CICMalDroid 2020 dataset, which is currently utilised in the literature, was used for this purpose. The features of the apps in the dataset were obtained using the AndroPyTool tool, and faster analysis files of 17,089 Android applications were obtained using the parallel execution technique. Permissions, intents, receivers, and services were used as static analysis features in this article. After these features were obtained, as data preprocessing, the ones with a grand total equal to 1 for each feature in the whole dataset were excluded in order to exclude the features that were specially created by the applications themselves. For each of the features specified for each application, a comma-separated text was obtained according to the usage status of the application. The BERT method was used to digitise the pertinent texts in order to create a unique embedding vector for every feature. Following the digitisation of the vectors, picture files were produced based on the length of each feature. To create a single image file, these image files were combined side by side. Finally, these image files were classified with CNNs. Experimental results were obtained by applying CNNs to the dataset used in the study. As a result of the experiments, a CNN with two outputs provided the highest performance with an accuracy of 91%, an F1-score of 89%, a Recall of 90%, and a Precision of 91%.

Keywords: Android; CNN; Android malware; BERT; deep learning; machine learning; Android security; static analysis

check for updates

Citation: Kiraz, Ö.; Doğru, İ.A. Visualising Static Features and Classifying Android Malware Using a Convolutional Neural Network Approach. *Appl. Sci.* **2024**, *14*, 4772. https://doi.org/10.3390/app14114772

Academic Editor: Andrea Prati

Received: 24 April 2024
Revised: 24 May 2024
Accepted: 27 May 2024
Published: 31 May 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

One of the goals of Android is to create an open and secure platform for mobile devices and various other device form factors while allowing users to install applications from different sources [1]. As mentioned here, Android is open source and users can install apps from different stores other than the official app store. Furthermore, according to the statistics released by Statista, Android has the highest usage in the world with a market share of 70.1% while the Apple iOS operating system reported a market share of 29.2% in the fourth quarter of 2023 [2]. Due to the fact that Android is the most popular mobile operating system worldwide and for the reasons mentioned above, malicious applications are created with the intention of accessing user data or causing harm to people who use Android applications. Consequently, Google is also implementing security steps to prevent such malicious applications and lessen their effect on users. According to the information on the Google developer website, Google Play Protect is a tool that was created in 2017 and

is used to scan 125 billion applications every day. This tool enables Google to react swiftly to threats and limit the number of devices and users that are impacted by them [3]. This program did not have a real-time scanning feature before last year; thus, customers could not always analyse the application they planned to install. Google Security blog revealed that real-time app detection would be included in Google Play Protect in October 2023 [4]. In addition, as malware developers develop new methods and improvements to overcome these obstacles, these tools may also be insufficient. As a result, more potent and dynamic techniques for Android malware detection must be developed.

Android malware applications pose many risks for users. These risks generally include the seizure of users' personal data; financial losses; and the theft of usernames, passwords, and similar information. Furthermore, Android malware poses various security problems for self-driving vehicles [5]. The Financial Threats Report for 2023 by Kaspersky reveals significant increases in mobile banking malware and cryptocurrency-related phishing [6]. In light of these data, it is extremely important to develop strong and effective systems for detecting Android malware in order to protect users from these risks.

Different methods have been developed in the literature on Android malware detection and are available to users. There are three Android malware detection methods in the literature: static, dynamic, and hybrid [7–11]. In static analysis, features are extracted by unpacking or disassembling applications [12–14]. Static analysis is a behaviour-based approach to Android malware detection that extracts static features of Android applications without running them on an emulator or on a real device [15–17]. In this method, the features of Android applications are obtained by reverse engineering the .apk file [18–21]. Detection methods based on static analysis are further divided into several categories, including visualisation/bytecode sequence properties, permissions, opcode sequences, API calls, inter-component communication (ICC), class.dex files, and mixed static properties [9]. Dynamic analysis mainly studies the runtime behaviour of applications [11,13,16,22]. Dynamic analysis monitors the behaviour of applications (i.e., memory usage, network access, dynamic taint, etc.) and the effects of the application on this isolated environment [23–26]. Researchers often choose system calls, API calls, and network traffic as dynamic features [10]. Hybrid analysis utilises a combination of dynamic and static methods [7,10,13]. Static analysis methods often use permissions and API call features, while dynamic/hybrid analysis methods often use permissions, API calls, intents, and system calls, and rarely use CPU/memory/battery consumption data, META-INF, certificates, and HTTP streams [9]. A comparison of the pros and cons of static and dynamic analysis methods is shown in Table 1.

Table 1. Advantages and disadvantages of static and dynamic analysis methods.

Analysis Method	Advantages	Disadvantages
Static	Fast scanning and control of malicious applications [27].	Static analysis is ineffective as many malicious applications use a range of deformation technologies such as bytecode encryption, reflection, and local code execution [27].
	No need to tune execution environments and relatively low computational load for static analysis [28].	The signature database is limited and unable to detect zero-day malware [29].
	Complete code coverage [30]. Ability to reveal errors that do not manifest themselves [31].	
Dynamic	Detect malicious applications that use some obfuscation techniques such as code encryption or packaging [28].	It requires a lot of effort to build the environment and the code coverage may be insufficient to safely analyse malware [30]
	Monitor the behaviour of malware in real time and identify malware types more accurately [32]	It requires high automation and real time and requires more time and memory resources [33].

The contributions of the proposed study can be summarised as follows:

- Presenting a CNN model and image-based approach for Android malware analysis based on the values determined with hyperparameter optimisation in the classification model.
- Performing classification and learning operations on the current dataset.
- Using BERT as a text classification algorithm for numerical representation of textual features to generate images.
- Using the image of each feature side by side as a single image file to effectively detect Android malware.

This study has five sections. The remainder of this paper is structured as follows: The second section provides an overview of studies that leverage deep learning and machine learning-based artificial intelligence approaches for Android malware analysis, with a focus on the generation and presentation of the images of features in these studies. In the third section, we delve into the materials and methods employed in the process of analysing Android malware, elaborating on the techniques used and the performance metrics selected to gauge the effectiveness of these methods. The fourth section presents the experimental results obtained from the various models used in our analysis, accompanied by a comprehensive discussion of these findings. Finally, in the last section of this paper, we offer general evaluations and insights derived from the results presented, highlighting the implications and potential future directions in the field of Android malware analysis.

2. Related Studies

There are many studies in the literature on Android malware analysis and detection. As in the work we present under the title of this section, studies that generate images from features and then apply ML algorithms and deep learning algorithms will be discussed.

There are studies that only read the .apk file as binary and generate an image file. In Mercaldo and Santone's study, they created a grayscale image file using a vector of 8-bit values obtained by reading .apk files [34]. Five different traditional machine learning approaches based on decision trees were compared using a deep neural network classifier as a classification algorithm [34]. The algorithms used are J48, Random Forest, Random Tree, Bayesian Network, and AdaBoost [34]. The deep neural network classifier showed the best results with an Accuracy of 0.918, Precision of 0.859, Recall of 0.878, and F-Measure of 0.875 obtained when the number of hidden states was 10 [34]. Kural et al. produced a grayscale image file using the above-mentioned method and a CNN method was used for the classification and the results obtained an Accuracy of 94.13% [35]. In Nazir et al., a grayscale image file was created using this method and the results obtained an Accuracy of 90% via classification with a CNN [36]. Al-Fawa'reh et al. created a grayscale image file by applying this method in their study, and the results of the classification process with a CNN were obtained with an Accuracy of 95.9% with an unbalanced dataset [37]. In the study by Khan, Kumar, and Tripathi, a grayscale image is generated using the above model and four different types of algorithms are used to extract local features as Scale-invariant feature transform (SIFT), Speeded-up robust features (SURF), Oriented FAST, and Rotated BRIEF (ORB). The AdaBoost, K-Nearest Neighbours (KNN), Support Vector Machine (SVM), and Random Forest (RF) machine learning models were used to classify the local features extracted from greyscale images and an Accuracy of 96.86% was obtained with AdaBoost [38].

In the literature, there are studies in which the DEX file contained in the .apk file is obtained, then these files are read in binary, an image file is created, and the image files are used as input and classified. Fang et al. used the file section features in the DEX file in their study. After the DEX file is extracted, the DEX file is divided into sections by parsing the header part of the DEX files [39]. The section information was used to convert the DEX files into RGB images and plain text files, respectively [39]. Finally, a Feature Fusion algorithm based on multiple kernel learning is used for classification. The KNN, SVM, and RF algorithms were also used for comparison [39]. A Precision of 0.96, a Recall of 0.96, and an F1-score of 0.96 were obtained with the multiple kernel learning

algorithm [39]. Xiao and Yang read DEX files and the image file to be created was in an RGB format [40]. The generated image files were classified using the CNN algorithm and the results were obtained with an Accuracy of 93%, a Precision of 93.6%, a Recall of 94.4%, an F-Measure of 94%, and an FPR of 9.7% [40]. Huang and Kao decompressed the applications and obtained the class.dex file and read these files as byte code [41]. The created images were transferred to the CNN model and a classification was performed; the results were obtained with a Detection Rate/False Positive of 96/9% and an Accuracy of 93% [41]. Mitsuhashi and Shinagawa developed a tool to convert malware files into images [42]. This tool is designed to accept Portable Executable (PE) format files for Windows malware and Dalvik Executable (DEX) format files for Android malware [42]. The developed tool checks the header information of the file to determine the file format and converts the binary data of the file into a greyscale image with 256 shades per pixel and classification with CNN [42]. The classification performance with an Accuracy of 93.65% and an F-score of 90.55% was obtained [42]. Alam and Demir extracted DEX files from the .apk file and stored the extracted DEX byte codes in a vector space of 8-bit unsigned integers [43]. The Gabor filter was applied to the resulting image file and the image features were obtained to find the similarities between the images [43]. Recursive Feature Elimination was applied to improve the accuracy of the classification and three different algorithms: Naive Bayes, AdaBoost, and linear SVM [43]. Naive Bayes had the best performance with an Accuracy of 99.43% [43]. Ding et al. created a greyscale image file using the classes.dex file in the .apk file [44]. The generated image files were classified using a CNN, and an Accuracy of 95.1%, TPR of 94.1%, and FPR of 0.7% were obtained [44]. In the Daoudi et al. study, DEX files are read in binary and converted into a greyscale "vector" image by considering each byte as an 8-bit pixel [45]. Image files are used as input for classification with CNNs and the results obtained had an Accuracy of 0.97, Precision of 0.97, Recall of 0.95, and F1-score of 0.96 [45].

In Bakour and Ünver, Android Manifest, Classes.dex, and .arsc files are read in binary and then converted into a 2D greyscale image file using the generated matrix [46]. In this study, the image was created with four different combinations: Manifest.xml; DEX files; Manifest and Resources.arsc; and Manifest, Resources.arsc, and DEX [46]. The created image files were classified with a CNN and reached a 98.96% Accuracy [46].

Khoa et al. obtained Permissions, Opcodes, API calls, System commands, activities, services, receivers, package names, and flowdroid properties using the AndroPyTool [47,48] tool [49]. Each integer value is converted to the corresponding binary value for the properties obtained [49]. Since the values of the properties can vary, in order to match a pixel in the image, it is ensured that they are only in the range of 0 to 255 [49]. The 8-bit elements are then concatenated into a single-bit array of length one array [49]. To simplify the image transformation process, a square RGB image was obtained to represent an Android application [49]. Finally, with the image files used as the input to 19 different CNN models, the highest scores were obtained with mobilenet_v2 with SGD optimisation algorithm with a Precision of 0.978, Recall of 0.981, F1 score of 0.979, and Accuracy of 0.99 [49].

Singh et al. created a greyscale image file by reading classes.dex, resource, manifest, and certificate files in binary and then extracted the handmade features from the image sections using Gray Level Co-occurrence Matrix-based (GLCM), Global Image deScripTor (GIST), and Local Binary Pattern (LBP) algorithms [50]. The extracted features are classified using machine learning algorithms such as K-Nearest Neighbours, Support Vector Machines, and Random Forests [50]. In the second stage, the handmade features are combined with CNN features to form the Feature Fusion strategy [50]. According to the results obtained using the Feature Fusion strategy, the accuracy of the Feature Fusion-SVM model using the combination of certificate and Android Manifest (certificate Android manifest) was 93.24% [50].

In the Zegzhda et al. study, .apk files were reverse-engineered to obtain .smali files, and then a list of individual APIs and protection level representations of the APIs was created [51]. An API sequence was created for each application according to API lists [51]. The APIs in these API sequences were mapped according to their protection level codes and

finally, an RGB image file was created [51]. Image files were input into CNNs, classification was performed, and the Accuracy obtained was 92.84 [51]. In this study, instead of reading the files in binary, the textual features obtained are digitised and an image file is created [51].

Darwaish and Naït-Abdesselam extracted permissions, activities, intents, services, providers, and receiver properties from the AndroidManifest.xml file using the .apk file Androguard [52] tool, as well as API calls, unique opcode sequences, and protected string properties [53]. Each character of all extracted application properties was converted into pixel values using ASCII code and filtering was performed with the help of a predefined dictionary [53]. In creating the RGB image file, the information was obtained as a result of the conversion of API calls and unique opcode sequences from the DEX file in the red channel; all suspected permissions and application components were extracted for values in the green channel, and the conversions of protected strings, suspected permissions, app components, and API calls in the blue channel were used [53]. The generated RGB image file was classified with a CNN and an Accuracy of 99.37% was obtained [53].

In Lan et al.'s study, API calls in the class.dex files were obtained using the Androguard [52] tool [54]. These API calls were summed by adding the ASCII value of each character, and the resulting value was modulo 256 to obtain the pixel value in the range of 0–255 [54]. At the end of the conversion, the pixel values of the standard API calls were set to the red channel, the suspicious and risky API calls were set to the blue channel, and each pixel value was set to 0 for the green channel to obtain an RGB image [54]. All suspicious and risky Android API calls were identified using a dictionary created based on various studies and datasets [54]. After the images were obtained, the random resizing layer resized the input image to a random size and the random padding layer filled the surroundings of the resized image with random zeros to increase the success of the developed model [54]. After this step, the classification process was performed using the autoencoder algorithm and an Accuracy of 96% was obtained [54].

Wang et al. obtained the API call sequences in the smali file using the Apktool [55] tool [56]. Each *.smali file contained ".class", ".super", ".implements", and ".invoke" related identifiers and object class, super class, interface, and API calls [56]. In this way, all internal function elements of the applications were obtained [56]. A feature database for the system APIs was created and used as a basis for the next feature sequence analysis and visualisation, which contains a total of 50,998 methods in 5858 Android API classes from level 1 to 30, with a fixed colour value setting for each system method [56]. The Depth-first search algorithm was used to generate a system-level API call sequence for all internal function call methods in the application one by one [56]. Their system obtained an accuracy of 98.9% and a robustness coefficient of 91.7% with the CNN and CGAN algorithms [56].

Zhao and Qian obtained RGB image files using Opcodes for the R channel, sensitive API packages for the G channel, and risky API functions for the B channel [57]. The Android operating system has a total of 255 Opcodes, which are encoded between 0x00 and 0xFF according to different functions [57]. By converting the hexadecimal value of the Opcode (in Android OS) to a decimal value, the R channel was digitised by matching pixels [57]. In order to extract API call features, it was divided into 58 classes according to their packages, with 18 classes (user privacy or device hardware such as camera and microphone) belonging to the high-risk level [57]. When this is reduced to the method level, they found that there are 41 high-risk methods [57]. After the image file was obtained, classification was performed with the CNN model and an Accuracy of 0.9067, Precision of 0.9336, and F1-score of 0.9356 were obtained [57].

Studies on detecting Android malware applications have been examined in the literature, and it has been observed that the CNN method is the most used method for classification among the studies examined. It has also been observed that machine learning algorithms such as Random Forest, Random Tree, J48, Naive Bayes, AdaBoost, KNN, and SVM were used in the studies. In studies in the literature, it has been observed that the static analysis method is generally used when obtaining images from features. The success

rates of the reviewed studies and this study in detecting Android malware are listed in n Section 4.2. In this way, a general review and comparison of the studies was provided.

3. Materials and Methods

In this section, general information about the dataset used in our study will be given. Then, the stages of the system developed for Android malware detection will be explained in detail. In the data processing phase, feature extraction operations, and in the data evaluation phase, classification operations and the method used will be explained in detail. The topics in this section are summarised in Figure 1.

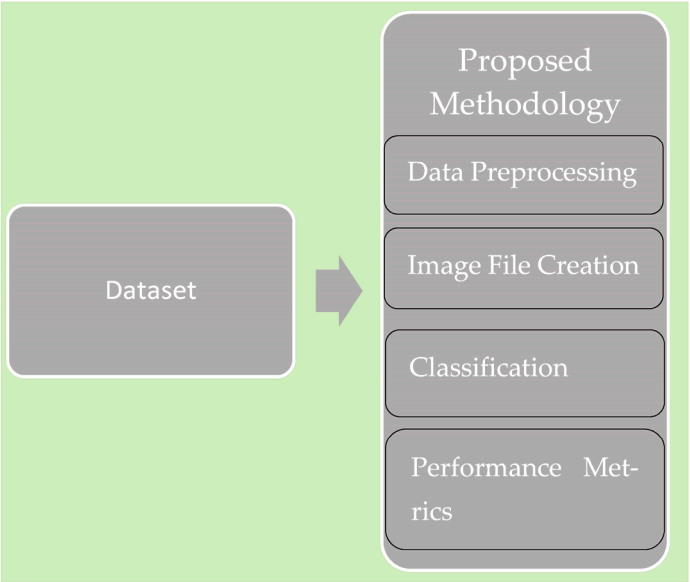


Figure 1. Organisational chart of the Materials and Methods section.

3.1. Dataset

The use of datasets has an important place in the studies conducted in the literature. When the studies conducted in the literature are examined, the Drebin [58] dataset was used in the [38,42,44–46,50] studies; the Malgenome [21] dataset was used in the [43,46] studies; the AMD [59] dataset was used in the [34,35,39,40,51] studies; the Androzoo [60] dataset was used in the [45,53,54] studies; the CIC-AndMal2017 [61] dataset was used in the [62] study; and the CICMalDroid 2020 [63,64] dataset was used in the [49,62] studies. In this study, the CICMalDroid 2020 [63,64] dataset was used. The main reason for using this dataset is that it has more up-to-date applications than other datasets. There are 17,341 applications in this dataset, and it consists of Adware, Banking, SMS, Riskware, and Benign categories.

3.2. Proposed Methodology for Android Malware Detection

The schematic of the proposed research methodology for Android malware detection is shown in Figure 2.

The proposed approach aims to provide a fast, dynamic and effective solution to Android malware detection by applying deep learning using static analysis features. In the extraction of static features, permissions, intents, receivers, and service features of each application are obtained using the AndroPyTool tool [47,48]. After these features are obtained, the embedding vectors of each feature are created using the BERT algorithm and image files are obtained using these vectors. The image files used with the CNN algorithm and the classification process are then performed. Additionally, the workflow steps of the developed system are shown in Figure 3.

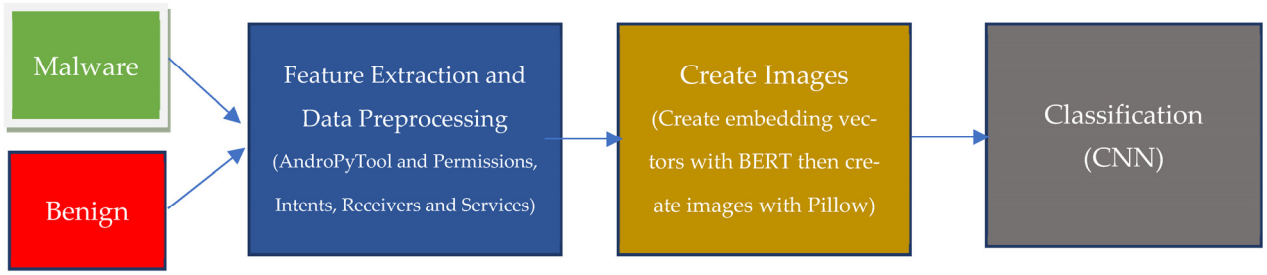


Figure 2. The architecture of the developed system.

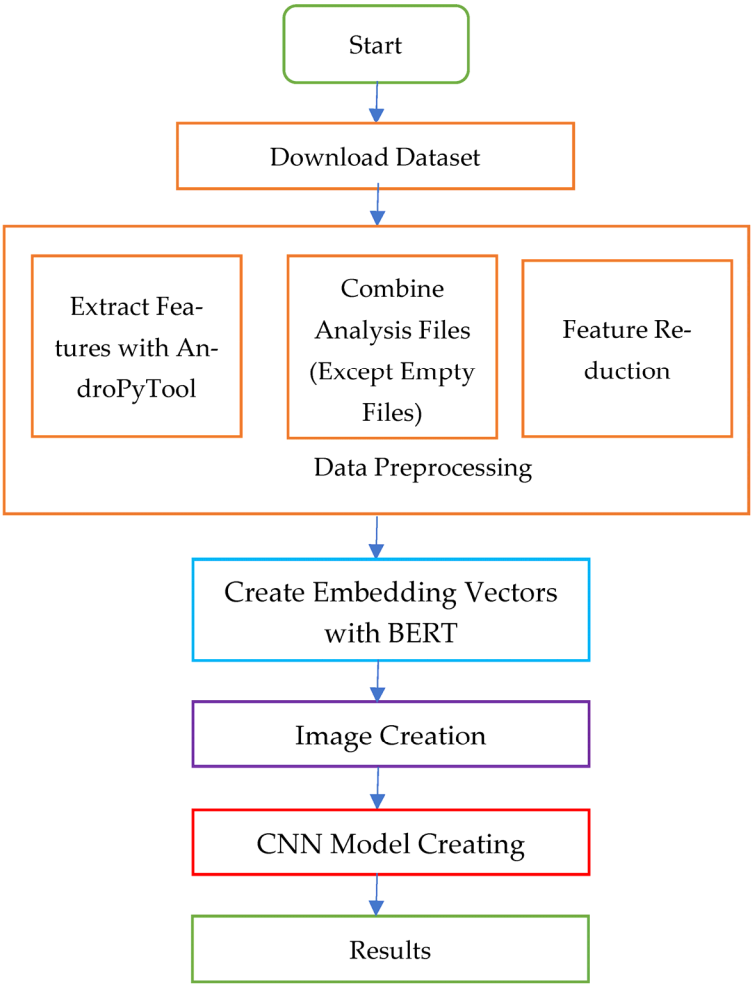


Figure 3. The workflow of the developed system.

As can be seen in Figure 3, the related methodology consists of 4 main stages: data preprocessing, image file creation, classification, and performance metrics for Android malware detection.

3.2.1. Data Preprocessing

In this process step, the features of Android applications are extracted using the AndroPyTool tool [47,48] of the applications in the dataset. AndroPyTool tool is a powerful framework for the automated extraction of static and dynamic features from Android applications. It integrates several well-known Android app analysis tools, such as DroidBox, FlowDroid, Strace, AndroGuard, and VirusTotal. Static features offer insights into the app's structure, permissions, components, and other characteristics without the need to execute the app. In this step, in order to run the AndroPyTool tool [47,48], the version of the

Docker application belonging to the Microsoft operating system installed on the computer is used. By running the "docker pull alexmyg/andropytool" command in the Docker application, the image of the AndroPyTool tool [47,48] is loaded via the Docker hub. In order to extract static analysis features with the AndroPyTool tool [47,48], the features are obtained by running the command "docker run --volume=D:\DATASET\Adware:/apks --cpus 2 --memory 4000 m --rm --name androPyToolAdwarealexmyg/andropytool -all -s/apks". In this command, --volume=D:\DATASET\Adware:/apks shows the location of the apk files. In addition, --cpus 2 --memory 4000 m commands are set as cpu 2 units and ram as 4 gb. These settings may vary depending on the computer being run. Docker images were run in parallel to finalise the analysis of applications more quickly. Apart from this, no parameter changes were made in AndroPyTool [47,48]. While these commands were running, it was observed that sometimes, errors were received during the static analysis of applications with the Androguard [52] tool. In these cases, the application causing the error was found and excluded and the relevant command was run again. Therefore, the number of applications in the dataset initially decreased from 17,341 to 15,725. In the Features_files folder, information about the static and dynamic features of each application was in a json format. The json file example is shown in Figure 4.


```
{
    "Pre_static_analysis": {
        "Filename": "81b21e758da65cbfdbaf707dae622ea60a95f9290c2dad722ad2ae97b331d955.apk",
        "md5": "248ae94179d79ae2d34e99507a2a946e",
        "sha256": "81b21e758da65cbfdbaf707dae622ea60a95f9290c2dad722ad2ae97b331d955",
        "sha1": "56cddf6c81f5c2fad652f9f0f39940c71b0a8399",
        "VT_positives": null
    },
    "Static_analysis": {
        "Package name": "com.androidemu.harvemario",
        "Permissions": [
        "Opcodes": {
        "Main activity": "com.androidemu.harvemario.MainActivity",
        "API calls": {
        "Strings": {
        "API packages": {
        "System commands": {
        "Intents": {
        "Activities": {
        "Services": {
        "Receivers": {
            "com.isw.third.app.CallReceiver": [
            "com.androidemu.harvemario.AlmService": [
        },
        "FlowDroid": {
    },
    "Dynamic_analysis": {
        "Droidbox": {
            "apkName": "/apks/samples//81b21e758da65cbfdbaf707dae622ea60a95f9290c2dad722ad2ae97b331d955.apk",
            "enfperm": [],
            "recvnet": {},
            "cryptousage": {},
            "recvsaction": {
                "com.androidemu.harvemario.AlmService": "push_alm",
                "com.isw.third.app.CallReceiver": "android.intent.action.PHONE_STATE_2"
            },
            "servicestart": {},
            "sendnet": {},
            "sendsms": {},
            "accessedfiles": {
            "fdaccess": {
            "dataleaks": {},
            "opennet": {},
            "dexclass": {
            "hashes": [
            "closenet": {},
            "phonecalls": {}
        },
        "Strace": "/apks/Dynamic/Strace/81b21e758da65cbfdbaf707dae622ea60a95f9290c2dad722ad2ae97b331d955.csv"
    },
    "VirusTotal": ""
}
```


Figure 4. Example of json files in the feature files folder.

The analysis files created separately for Adware, Banking, Benign, Riskware, and SMS were arranged as a single analysis file with the code developed in Python language. The file size is approximately 8 GB and each line in this file contains static and dynamic features and category information for each application. It consists of 23 columns in total, these are APK_ID, Pre_static_analysis_Filename, Pre_static_analysis_md5, Pre_static_analysis_sha256, Pre_static_analysis_sha1, Pre_static_analysis_VT_positives, Static_analysis_Package name,

Static_analysis_Permissions, Static_analysis_Opcodes, Static_analysis_Main activity, Static _analysis_System commands, Static_analysis_Intents, Static_analysis_Activities, Static _analysis_Services, Static_analysis_Receivers, Static_analysis_FlowDroid, Dynamic_analysis _DroidboxVirusTotal, Dynamic_analysis_Strace, and category.

In this study, permissions, intents, receivers, and services were used as static features. These features (permissions, intents, services, and receivers) were chosen because they are the most used for static analysis in studies in the literature. According to the analysis for permissions, the number of unique permissions was found to be 3489, while the number of permissions used more than once (Count > 1) was found to be 809 different permission features. In addition, 7 empty permissions were detected, and these were deleted as a result of the analysis. Table 2 shows the 10 most used permissions by benign and malicious applications.

Table 2. The 10 most used permissions.

The 10 Most Used Permissions in Benign Applications	The 10 Most Used Permissions in Malware Applications
android.permission.INTERNET	android.permission.INTERNET
android.permission.ACCESS_NETWORK_STATE	android.permission.WRITE_EXTERNAL_STORAGE
android.permission.WRITE_EXTERNAL_STORAGE	android.permission.READ_PHONE_STATE
android.permission.WAKE_LOCK	android.permission.ACCESS_NETWORK_STATE
com.google.android.c2dm.permission.RECEIVE	android.permission.SEND_SMS
android.permission.ACCESS_WIFI_STATE	android.permission.RECEIVE_SMS
android.permission.VIBRATE	android.permission.READ_SMS
android.permission.GET_ACCOUNTS	android.permission.ACCESS_WIFI_STATE
com.android.vending.BILLING	android.permission.CHANGE_WIFI_STATE
android.permission.READ_PHONE_STATE	android.permission.ACCESS_COARSE_LOCATION

As can be seen in Table 2, android.permission.INTERNET and android.permission. ACCESS_NETWORK_STATE are in the top 2 in both benign and malicious applications. Also, malicious applications tend to use android.permission.SEND_SMS, android.permission. RECEIVE_SMS, and android.permission.READ_SMS permissions related to SMS. According to the number of permissions used more than once (Count > 1), the text is obtained for each application by printing the permissions used by the applications one by one with commas. In the analysis of intents, the number of unique intents is 10,896 and the number of intents used more than once (Count > 1) is 2512. Seven empty ones were detected and deleted as a result of the analysis. Table 3 shows the 10 most used intents by benign and malicious applications.

Table 3. The 10 most used intents.

The 10 Most Used Intents in Benign Applications	The 10 Most Used Intents in Malware Applications
android.intent.action.MAIN	android.intent.action.MAIN
com.google.android.c2dm.intent.RECEIVE	android.intent.action.BOOT_COMPLETED
android.intent.action.VIEW	android.provider.Telephony.SMS_RECEIVED
com.android.vending.INSTALL_REFERRER	android.net.conn.CONNECTIVITY_CHANGE
com.google.android.c2dm.intent.REGISTRATION	android.intent.action.USER_PRESENT
android.intent.action.BOOT_COMPLETED	android.intent.action.SCREEN_ON
com.google.android.gms.measurement.UPLOAD	android.net.wifi.STATE_CHANGE
android.appwidget.action.APPWIDGET_UPDATE	android.intent.action.BATTERY_CHANGED
android.net.conn.CONNECTIVITY_CHANGE	android.intent.action.DATA_SMS_RECEIVED
com.google.firebase.INSTANCE_ID_EVENT	android.intent.action.PACKAGE_ADDED

As can be seen in Table 3, the intent android.intent.intent.action.MAIN is in the first place for both the benign and malicious applications. In addition, as in permissions, it was found that intent information related to SMS was used in malware applications. According to the number of intents used more than once (Count > 1), the intents used by the applications are printed one by one with a comma and the text is obtained for each

application. According to the analysis of services, the number of unique services is 15,573 and the number of services used more than once (Count > 1) is 3474. Table 4 shows the 10 most used services by benign and malicious applications.

Table 4. The 10 most used services.

The 10 Most Used Services in Benign Applications	The 10 Most Used Services in Malware Applications
com.google.android.gms.measurement.AppMeasurementService	com.pay.sdk.msg.PayService
com.google.android.gms.analytics.CampaignTrackingService	com.pay.sdk.msg.PayListenerService
com.google.firebase.iid.FirebaseInstanceIdService	com.ast.sdk.server.ServerM
com.google.android.gms.auth.api.signin.RevocationBoundService	com.software.application.C2DMReceiver
com.google.android.gms.analytics.AnalyticsService	com.b.ht.IDD
com.google.firebase.messaging.FirebaseMessagingService	com.b.ht.JAA
com.parse.PushService	com.door.pay.sdk.sms.SmsService
com.google.firebase.crash.internal.service.FirebaseCrashReceiverService	com.wyzf.service.InitService
com.google.firebase.crash.internal.service.FirebaseCrashSenderService	com.mj.jar.pay.SmsServices
com.digits.sdk.android.ContactsUploadService	com.mj.sms.service.InitService

According to the number of services used more than once (Count > 1), the services used by the applications are printed one by one with commas and the text is obtained for each application. According to the analysis of receivers, the number of unique receivers is 14,691 and the number of receivers used more than once (Count > 1) is 3656. Thirty-nine empty values were deleted. Table 5 shows the top 10 receivers used by benign and malicious applications.

Table 5. The 10 most used receivers.

The 10 Most Used Receivers in Benign Applications	The 10 Most Used Receivers in Malware Applications
com.google.android.gms.measurement.AppMeasurementReceiver	com.google.android.c2dm.C2DMBroadcastReceiver
com.google.firebase.iid.FirebaseInstanceIdInternalReceivercom.google.	com.ast.sdk.receiver.ReceiverM
firebase.iid.FirebaseInstanceIdReceiver	com.software.application.Notificator
com.google.android.gms.gcm.GcmReceiver	com.software.application.SmsReceiver
com.google.android.gms.analytics.CampaignTrackingReceiver	com.software.application.Checker
com.google.android.gms.analytics.AnalyticsReceiver	com.b.ht.JDR
com.appsflyer.MultipleInstallBroadcastReceiver	com.door.pay.sdk.sms.SmsReceiver
com.google.android.gms.measurement.AppMeasurementInstallReferrerReceiver	com.mj.jar.pay.InSmsReceiver
com.parse.GcmBroadcastReceiver	top.cure.rece
com.google.android.gcm.GCMBroadcastReceiver	com.emag.yapz.receiver.BootReceiver

According to the number of receivers used more than once (Count > 1), the receivers used by the applications were printed one by one with commas and the text was obtained for each application. After textualising the features, the "bert-base-uncased" model was used to digitise the texts. BERT is a self-supervised pre-trained transformer model on a large English dataset (BERT base model (uncased)). The BERT model was pre-trained using a dataset of 11,038 unpublished books of the BookCorpus name and English Wikipedia (excluding lists, tables, and titles) [65]. Python code was developed to translate the generated textual expressions into a numeric vector. These steps were also applied for the Matching_Permissions, Matching_Intents, Matching_Receivers, and Matching_Services columns and the embedding vectors were obtained. The flow chart created for the developed Python code is shown in Figure 5.

This code performs several tasks related to natural language processing (NLP) and data manipulation as can be seen in Figure 5. The first step is to import libraries and read features as dataframes. Also start loading static features in a CSV file into a Pandas DataFrame and trimming the DataFrame (specific columns). The second step is to load a pre-trained language model. It loads a pre-trained BERT (Bidirectional Encoder Representations from Transformers) model (bert-base-uncased) and its tokeniser using the Hugging Face

transformers library. The third step is to create a new DataFrame based on unique categories. It creates a new DataFrame (new_dataframe) containing rows from the original DataFrame (df) grouped by unique values in the "category" column. The fourth step is to compute embeddings for features. It defines a function (get_embedding) that takes features as input, tokenises them using the BERT tokeniser, feeds them through the BERT model, and computes the mean of the last hidden states to obtain a fixed-size embedding for each feature. The fifth step is to apply the embedding function. It applies the get_embedding function to the features (permissions, intents, receivers, and services) columns. Finally, embedding vectors are obtained for each feature.

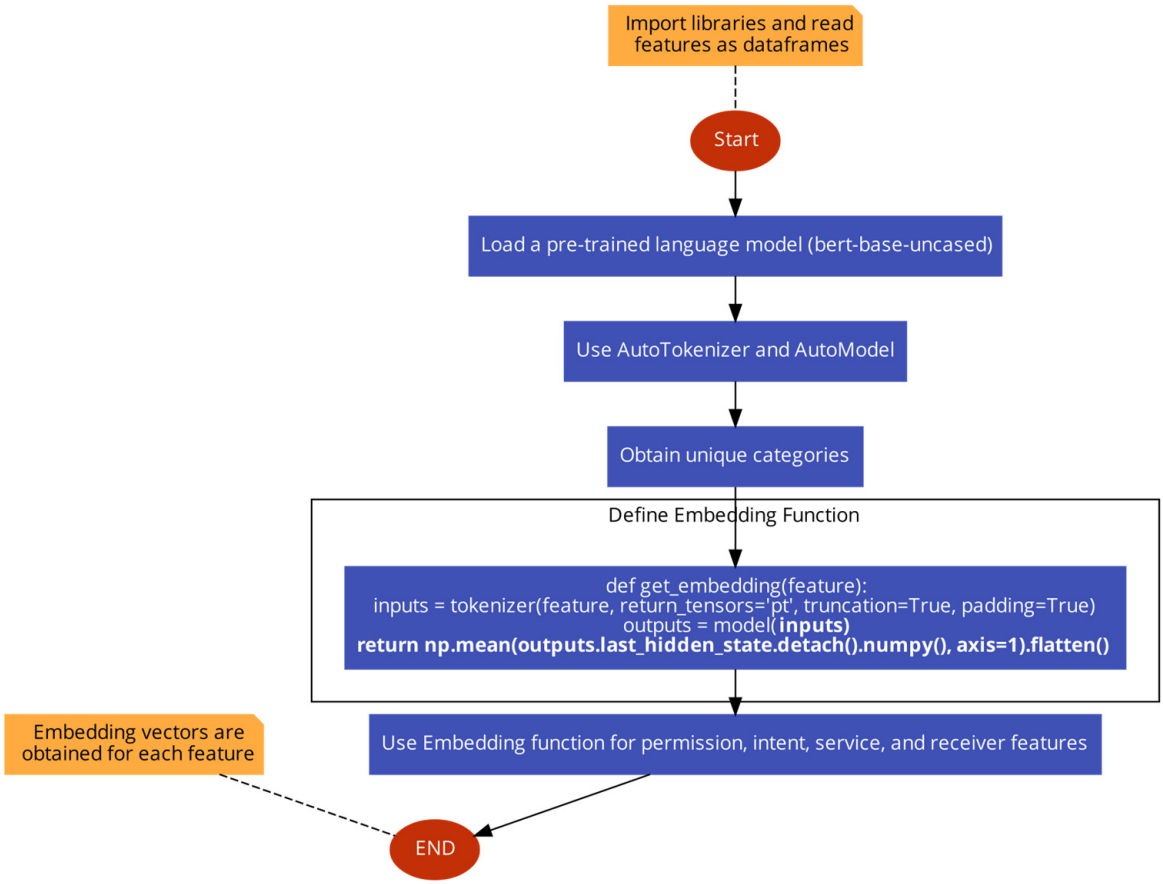


Figure 5. The flow chart illustrates the developed Python code for embedding vectors.

3.2.2. Image Creation

After these steps, the process of converting the embedding vectors into image files is conducted. The Pillow (PIL Fork) library [66] was used to create images. Pillow (PIL Fork) [66] is a Python library used to work with image files. It converts the embedding vector to a NumPy array and normalises the embedding vector to the range of [0, 255]. If there are missing data for the given target dimension, the dimension is padded with zeros to reach the corresponding dimension. The normalised and padded vector is reshaped into a square matrix of the target dimension. For permissions, 28 × 28; intents, 42 × 42; receivers, 45 × 45; and services, 42 × 42 sized images are created. Then, the separately generated images are merged side by side. As a result of combining the pictures side by side, a 157 × 45 sized picture is obtained. The examples of images for each category are shown in Figure 6.

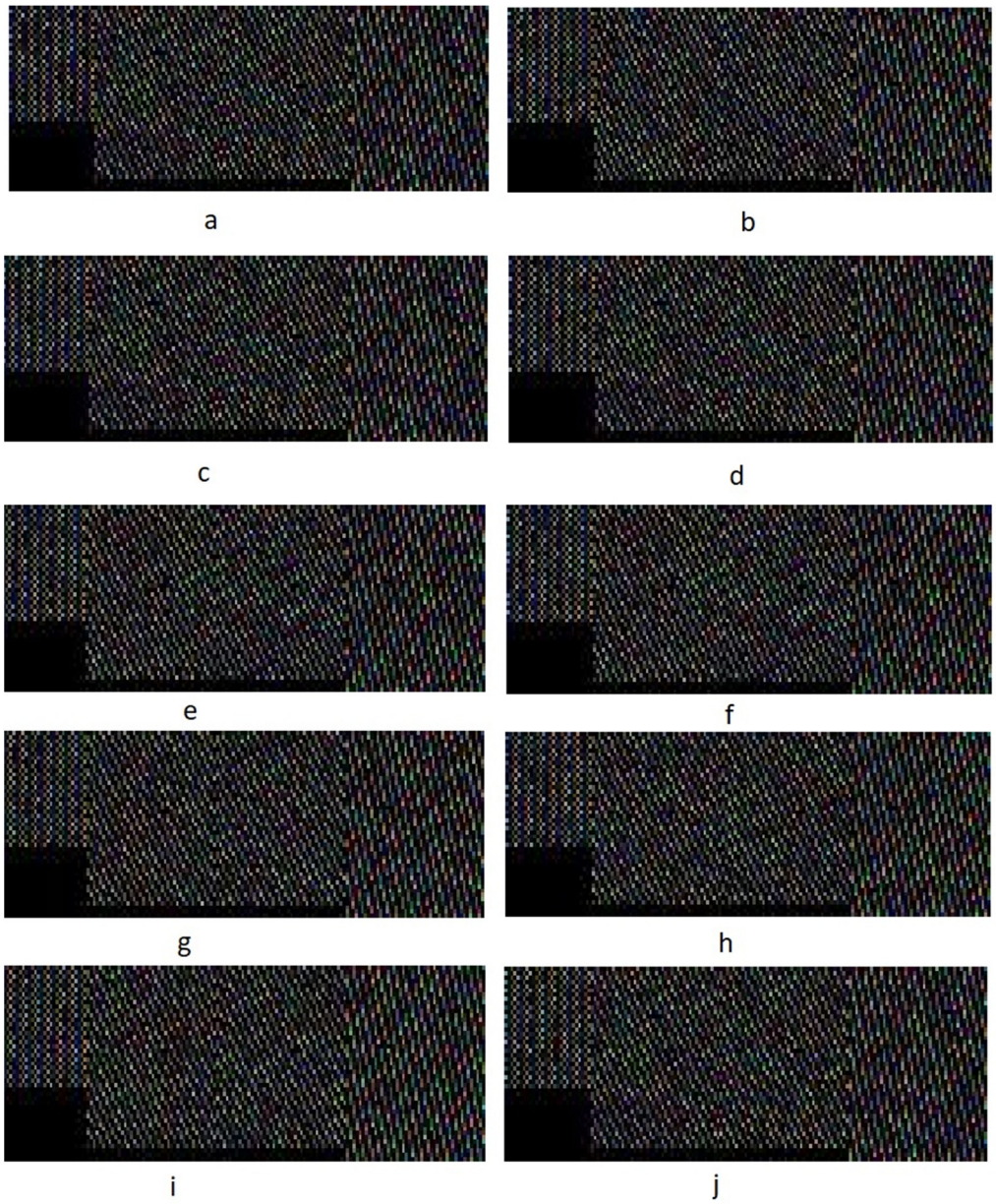


Figure 6. Examples of created images, ((**a**,**b**) Adware category example, (**c**,**d**) Banking category example, (**e**,**f**) Riskware category example, (**g**,**h**) SMS category example, and (**i**,**j**) Benign category example).

3.2.3. Classification

The CNN model was selected to use the obtained images as the input and perform the classification process. The CNN model has achieved tremendous success in many research areas such as image recognition, semantic segmentation, and natural language processing [7,67,68]. The CNN architecture created for the study is shown in Figure 7.

As can be seen in Figure 7, there are 2 convolutional layers in CNN, and after each convolutional layer, max-pooling is used to reduce the number of parameters and calculations in the network. Then, with the Flatten layer, the matrices obtained with the previous layers are reduced to a single-plane vector and the data obtained in the last layer are used. With the last layer (Fully Connected Layer), single-plane vectors are obtained, and the learning process is performed with artificial neural networks. Finally, the classification process is finalised with the output layer. The values of hyperparameters such as filters, kernel_size, pool_size, units, epochs, batch_size, validation_split, and verbose hyperparameters used in

this CNN architecture are obtained as a result of 35 trials using the optuna library [69], in which value range the hyperparameters are tested as shown in Table 6.

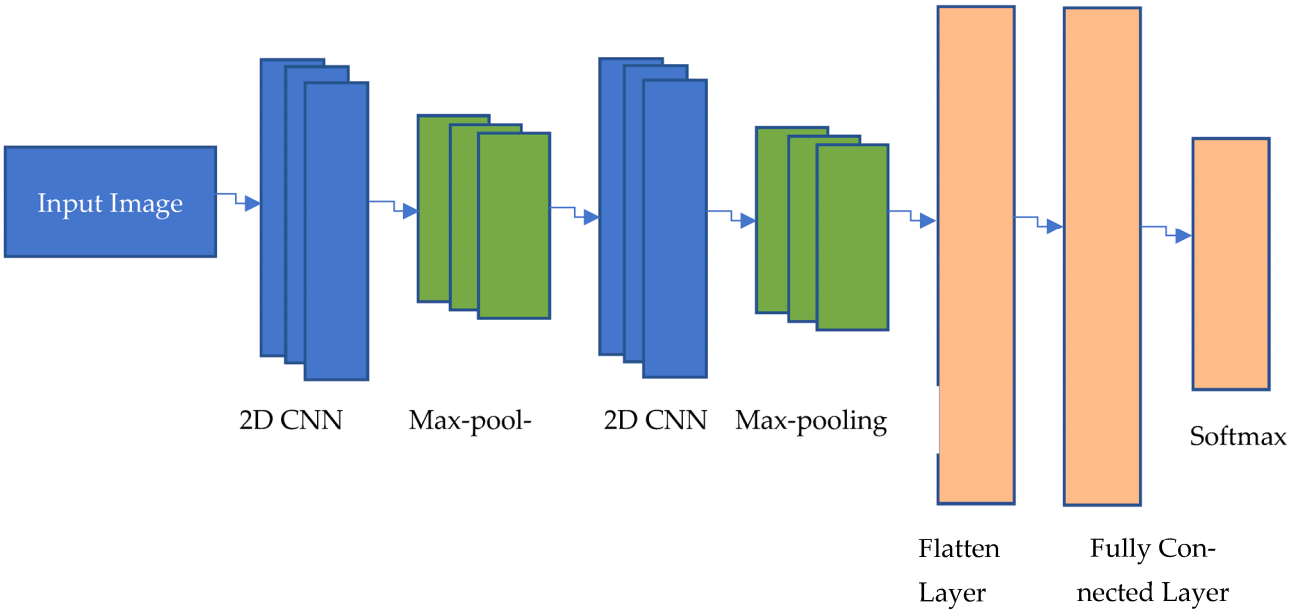


Figure 7. CNN architecture.

Table 6. Value ranges for CNN model hyperparameters.

Parameters	Values
Filters	32–256
Kernel size	3–7
Pool size	2–4
Units	64–256
Epochs	15–30
Batch size	16, 32, 64
Validation split	0.1, 0.3
Verbose	0, 1

Accordingly, the value of "filters" is 221, "kernel_size" is 7, "pool_size" is 4, "units" is 209, "epochs" is 19, "batch_size" is 64, "validation_split" is 0.11426968692124594, and "verbose" is 0, as obtained in Trial 31.

3.2.4. Performance Metrics

Performance metrics serve to evaluate and compare deep learning and machine learning algorithms for the model [70]. For the evaluation of the developed model, the confusion matrix is frequently used in the literature. The use of the 2 × 2 matrix for the confusion matrix is shown in Table 7.

Table 7. Confusion matrix.

		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	True Positive (TP)	False Negative (FN)
	Negative (0)	False Negative (FN)	True Negative (TN)

TP indicates positively predicted and true, TN indicates negatively predicted and true, FP indicates positively predicted and false, and FN indicates negatively predicted and false.

Recall, Precision, Accuracy, and F-measure values are calculated using these values. The Recall value should be as high as possible and shows how many of all positive classes are correctly predicted. The calculation used for Recall is shown in Equation (1).

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{1}$$

Precision should be as high as possible and indicates how many of all classes predicted as positive are actually positive. The calculation used for Precision is shown in Equation (2).

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2}$$

The Accuracy value should be as high as possible and indicates how many of all classes (positive and negative) were correctly predicted. The calculation used for Accuracy is shown in Equation (3).

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} \tag{3}$$

The F-measure value is used to compare the Precision and Recall values at the same time. The calculation used for the F-measure is shown in Equation (4).

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

In this study, the success of the classification process was measured using Recall, Precision, Accuracy, and F1-score performance metrics.

4. Experimental Results

In this section, the analysis results of the proposed CNN-based, DenseNet121, MobileNetV2, and ResNet50 models are discussed. Additionally, a comparison between previous studies and the current study in terms of attack detection success is included in the Discussion Section.

4.1. Results

Experiments were conducted on the developed CNN model in two different ways. In the first CNN model trial, five (Adware, Banking, SMS, Riskware, and Benign) categories were classified in the output layer. According to the classification with the CNN model with five categories, a weighted average Accuracy of 89%, Precision of 89%, Recall of 89%, and F1-score of 89% were obtained. In the second trial, the CNN model was tested as two outputs (Benign and Malware) and the results showed a weighted average Accuracy of 91%, Precision of 90%, Recall of 91%, and F1-score of 89%. In order to compare the success of the study, classification was performed with ResNet50, DenseNet121, and MobileNetV2 CNN models and the obtained results are shown in Table 8.

Table 8. Experiment results of our models and ResNet50, DenseNet121, and MobileNetV2 models.

Model Name	Accuracy (%)	F1-Score (%)	Precision (%)	Recall (%)
Our model with 5 outputs	89	89	89	89
Our model with 2 outputs	91	89	90	91
DenseNet121 with 2 outputs	79	80	88	79
MobileNetV2 with 2 outputs	76	66	82	76
ResNet50 with 2 outputs	88	88	88	88

As can be seen in Table 8, the best obtained results were with the CNN model we developed. The closest obtained results to the developed model were with the ResNet50 model with an Accuracy of 88%, an F1-Score of 88%, a Precision of 88%, and a Recall of 88%.

The second closest results were obtained with the DenseNet121 model, with an Accuracy of 79%, an F1-Score of 80%, a Precision of 88%, and a Recall of 79%. The worst results were obtained with the MobileNetV2 model: an Accuracy of 76%, an F1-Score of 66%, a Precision of 82%, and a Recall of 76%.

4.2. Discussion

The analysis results of the two models with the highest success rates among the models proposed in the study (two outputs) were compared with the analysis results of the studies in the literature. The results obtained are given in Table 9.

Table 9. Comparison of studies in the literature and our model.

References	Datasets	Analysis Method	Features	Creating Image Method	Algorithms	Classification Performance
[34]	AMD	static	apk file	Binary to an image	deep neural network, J48 (J48), Random Forest (RF), Random Tree (RT), Bayesian Network (BN), and AdaBoost	Deep Learning Accuracy 91.8%
						F-Measure 87.5%
						Precision 85.9%
						Recall 87.8%
[36]	Virus-Share	static	apk file	Binary to an image	CNN	Accuracy 90%
[37]	-	static	apk file	Binary to an image	CNN	Unbalanced dataset Accuracy 74%
						Balanced dataset Accuracy 95.9%
[40]	AMD	static	classes.dex file	Binary to an image	CNN	Accuracy 93%
						F-measure 94%
						Precision 93.6%
						Recall 94.4%
[41]	Cheetah Mobile Taiwan Agency	static	classes.dex file	Binary to an image	CNN	Accuracy 93%
[42]	Drebin	static	classes.dex file	Binary to an image	CNN	Accuracy 93.65%
						F-measure 90.55
						Precision 90.16
						Recall 90.96%
[44]	Drebin	static	classes.dex file	Binary to an image	CNN	Accuracy 95.1%
[49]	CICMalDroid 2020	static	Permissions, Opcodes, API Calls, System Command, Activities, Services, Receivers, API package, Flowdroid	Binary to an image (if feature exists: 1; otherwise: 0)	CNN	Accuracy 99%
						F1 Score 97.9%
						Precision 97.8%
						Recall 98.1%
[50]	Drebin	static	Certificate file, Android manifest file, resources file, and classes.dex	Binary to an image	SVM, KNN, and RF for malware images. Texture descriptors GLCM, GIST, and LBP	Certificate and Android Manifest (Feature Fusion-SVM) Accuracy 93.24%

Table 9. *Cont.*

References	Datasets	Analysis Method	Features	Creating Image Method	Algorithms	Classification Performance
[51]	AMD	static	.smali file in API sequence with protection levels	API code to image	CNN	Accuracy 92.84%
[56]	-	static	API calls	System call sequence into an RGB image (colour values matching from the constructed feature database)	CNN	Accuracy 93.5% F-Score 92.9% Precision 92.1% Recall 90.1%
[57]	Drebin	static	Opcodes and API calls	ASCII code and dictionary	CNN	Accuracy 90.67% F1 93.56% Precision 93.36% Recall 93.95%
[71]	AndroZooCICMalDroid2020	static	DEX file	Binary to an image	Vision Transformer (ViT)	CICMalDroid2020 Accuracy 86.81% F1 Score 87.01% Precision 87.53% Recall 86.81%
Our CNN method with 2 outputs	CICMalDroid2020	static	Permissions, intents, receivers and services	Create embedding vectors with BERT then create images with Pillow	CNN	Accuracy 91% F-Score 89% Precision 90% Recall 91%

The comparison of the studies in the literature on classification by generating images from Android features and the developed system information are shown in Table 9. In the studies carried out, it was seen that the analyses were carried out on ready-made datasets. In the studies examined, the Drebin [58] and AMD [59] datasets were widely used. In our studies, the CICMalDroid 2020 [63,64] dataset was used because it has more up-to-date Android applications.

It was observed that static features were generally used in the studies conducted by generating images and the CNN method was used for classification. Furthermore, in the examined studies, machine learning algorithms and the Vision Transformer (ViT) algorithm were used for classification. In the study by Al-Fawa'reh et al. [37], three convolutional layers were followed by a max-pooling layer after each convolutional layer, and a total parameter count of 37,975,105 was achieved. Xiao et al. [40] utilised three convolutional layers, each followed by a max-pooling layer. Ding et al. [44] employed two convolutional layers, each followed by a max-pooling layer. In the study by Singh et al. [50], three convolutional layers followed by max-pooling layers were used. Zegzhda et al. [51] employed four convolutional layers followed by one max-pooling layer. Wang et al. [56] utilised three convolutional layers and two pooling layers. In our study, we employed two convolutional layers followed by a max-pooling layer after each convolutional layer, resulting in a total parameter count of 3,258,799. It is observed that the developed model has low complexity. Unlike the CNN architecture used in other studies, our model is developed

with fewer convolutional layers and is designed to be simpler and more understandable. This aims to run the model with less GPU and CPU usage.

The studies showed that the images used varied in size. The image dimensions for each study are presented in Table 10.

Table 10. Image size information about studies in the literature.

References	Image Sizes
[34]	256 width and height depend on file size
[37]	50 width 50 height
[42]	Width depends on file size
[44]	512 width 512 height
[49]	210 width 210 height
[50]	108 width 108 height
[51]	384 width 384 height
[56]	32 width 32 height
[57]	64 width 64 height
[71]	224 width 224 height
Our study	157 width 45 height

The features were used for the classification process, and the classification success varies between 90% and 99% according to the Accuracy value, the classification success varies between 87.01% and 97.9% according to the F1-score value, the classification success varies between 85.9% and 97.8% according to the Precision value, and the classification success varies between 86.81% and 98.1% according to the Recall value. It was seen that the success of the study was close to or higher than the success of other studies. In the studies given in Table 9, it is generally seen that the file itself is produced as a binary reading.

In most of the studies, feature selection is not performed by generating the image as a binary read of the file itself, and it is thought that it is aimed to achieve high classification success since the file itself is processed. In this study, unlike other studies, four features were selected, textual expressions were converted into embedding vectors with BERT, image files were created with these embedding vectors, and results were obtained by classifying them with the CNN model. When compared with the studies using the same dataset used in our study, it was seen that Khoa et al. [49] obtained the most successful result classification results, our study was the second, and Jo et al. [71] was the last.

5. Conclusions

Today, the popularity of Android phones has led to a significant increase in malware applications in this field. Android malware applications present various risks for users, including the seizure of personal data; financial losses; and the theft of usernames, passwords, and similar information. Due to the prevalence of Android malware in the official app market and other app stores, these risks are further exacerbated. The methods developed to protect Android device users are insufficient due to both malware diversity and changes in malware behaviour.

In this study, a comprehensive literature review is provided to showcase existing studies, an Android malware detection system based on image generation from static analysis features and classification with CNN is proposed as an effective and powerful approach to overcome the mentioned problem, and the effectiveness of this model is evaluated with performance measurements. The recent dataset named CICMalDroid 2020 was used in the study and static features were extracted from this dataset with the AndroPyTool tool. The static features used in the study were determined as permissions, intents, receivers, and services. The features of Android applications were converted into numerical vectors using BERT, and then an image was created for each Android application. The images obtained were used as input to the CNN developed for the classification process. The filters, kernel_size, pool_size, units, epochs, batch_size, validation_split, validation_split, and verbose hyperparameters in the developed CNN model were determined automatically

with the help of optuna to give the best result. The results evaluated with the CNN-based model proposed in this study were an Accuracy of 91%, an F1-score of 89%, a Precision of 90%, and a Recall of 91% obtained.

In future studies, the aim will be to develop a hybrid analysis method using dynamic analysis and static analysis together. Dynamic analysis refers to the process of running Android applications (APKs) on real or virtual Android devices for a certain period and analysing the behaviours the applications exhibit during this runtime. It is planned to use the features obtained through dynamic and static analyses together using specific feature selection algorithms. The combination of static and dynamic analyses is envisaged for the detection of malicious applications. It is also planned to obtain better results using hybrid methods in the classification stage. It is also planned to develop a model allowing users to upload their desired applications to the system through a developed web interface for deeper analysis.

Author Contributions: The first author (Ö.K.) conducted the experiments, wrote the manuscript, and executed the software process. The second author (İ.A.D.) was responsible for supervising and correcting the direction of the work. All authors have read and agreed to the published version of the manuscript.

Funding: The authors received no financial support for the research, authorship, or publication of this paper.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original data data regarding Android malware dataset (CICMalDroid 2020) presented in the study is openly available at https://www.unb.ca/cic/datasets/maldroid-2020.html (accessed on 20 May 2024) and the raw data supporting the conclusions of this study are available on request from the corresponding author due to privacy.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Android Security Paper 2023. Available online: https://services.google.com/fh/files/misc/android-enterprise-security-paper-2023.pdf?utm_medium=blog&utm_source=keyword&utm_content=cta-txt&utm_campaign=2023-oct-global-android_14_security-eng&utm_term=security (accessed on 26 February 2024).
2. Market Share of Mobile Operating Systems Worldwide 2009–2023. Available online: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (accessed on 26 February 2024).
3. On-Device Protections. Available online: https://developers.google.com/android/play-protect/client-protections?hl=en (accessed on 26 February 2024).
4. Enhanced Google Play Protect Real-Time Scanning for App Installs. Available online: https://security.googleblog.com/2023/10/enhanced-google-play-protect-real-time.html (accessed on 26 February 2024).
5. Hammood, L.; Doğru, İ.A.; Kılıç, K. Machine Learning-Based Adaptive Genetic Algorithm for Android Malware Detection in Auto-Driving Vehicles. *Appl. Sci.* **2023**, *13*, 5403. [CrossRef]
6. Kaspersky. Global Mobile Banking Malware Grows 32 Percent in 2023. Available online: https://www.kaspersky.com/about/press-releases/2024_global-mobile-banking-malware-grows-32-percent-in-2023 (accessed on 15 May 2024).
7. Zhu, H.-J.; Gu, W.; Wang, L.-M.; Xu, Z.-C.; Sheng, V.S. Android malware detection based on multi-head squeeze-and-excitation residual network. *Expert Syst. Appl.* **2023**, *212*, 118705. [CrossRef]
8. Guerra-Manzanares, A. Machine Learning for Android Malware Detection: Mission Accomplished? A Comprehensive Review of Open Challenges and Future Perspectives. *Comput. Secur.* **2024**, *138*, 103654. [CrossRef]
9. Manzil, H.H.R.; Naik, S.M. Detection approaches for android malware: Taxonomy and review analysis. *Expert Syst. Appl.* **2024**, *238*, 122255. [CrossRef]
10. Liu, K.; Xu, S.; Xu, G.; Zhang, M.; Sun, D.; Liu, H. A Review of Android Malware Detection Approaches Based on Machine Learning. *IEEE Access* **2020**, *8*, 24579–124607. [CrossRef]
11. Alamro, H.; Mtouaa, W.; Aljameel, S.; Salama, A.S.; Hamza, M.A.; Othman, A.Y. Automated Android Malware Detection Using Optimal Ensemble Learning Approach for Cybersecurity. *IEEE Access* **2023**, *11*, 72509–72517. [CrossRef]
12. Vinayakumar, R.; Soman, K.P.; Poornachandran, P. Deep Android Malware Detection and Classification. In Proceedings of the 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Udupi, India, 13–16 September 2017; pp. 1677–1683.

13. Arora, A.; Peddoju, S.K.; Conti, M. PermPair: Android Malware Detection Using Permission Pairs. *IEEE Trans. Inf. Forensics Secur.* **2020**, *15*, 1968–1982. [CrossRef]
14. Alzaylaee, M.K.; Yerima, S.Y.; Sezer, S. DL-Droid: Deep learning based android malware detection using real devices. *Comput. Secur.* **2020**, *89*, 101663. [CrossRef]
15. Idrees, F.; Rajarajan, M.; Conti, M.; Chen, T.M.; Rahulamathavan, Y. PIndroid: A novel Android malware detection system using ensemble learning methods. *Comput. Secur.* **2017**, *68*, 36–46. [CrossRef]
16. Surendrana, R.; Thomas, T.; Emmanuel, S. A TAN based hybrid model for android malware detection. *J. Inf. Secur. Appl.* **2020**, *54*, 102483. [CrossRef]
17. Bhat, P.; Dutta, K. A multi-tiered feature selection model for Android malware detection based on feature discrimination and information gain. *J. King Saud Univ.-Comput. Inf. Sci.* **2022**, *34*, 9464–9477. [CrossRef]
18. Kabakus, A.T. Hybroid: A Novel Hybrid Android Malware Detection Framework. *Erzincan Univ. J. Sci. Technol.* **2021**, *14*, 331–356. [CrossRef]
19. Alzaylaee, M.K.; Yerima, S.Y.; Sezer, S. Improving Dynamic Analysis of Android Apps Using Hybrid Test Input Generation. In Proceedings of the IEEE International Conference on Cyber Security and Protection of Digital Services (Cyber Security 2017), London, UK, 19–20 June 2017; pp. 1–8.
20. Grace, M.; Zhou, Y.; Zhang, Q.; Zou, S.; Jiang, X. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services—MobiSys'12, New York, NY, USA, 25–29 June 2012; pp. 281–294.
21. Zhou, Y.; Jiang, X. Dissecting Android Malware: Characterization and Evolution. In Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland 2012), San Francisco, CA, USA, 20–23 May 2012; pp. 95–109.
22. Wang, D.; Chen, T.; Zhang, Z.; Zhang, N. A survey of Android malware detection based on deep learning. In *Machine Learning for Cyber Security (ML4CS 2022)*, 5th ed.; Xu, Y., Yan, H., Teng, H., Cai, J., Li, J., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2022.
23. Chandramohan, M.; Tan, H.B.K. Detection of Mobile Malware in the Wild. *Computer* **2012**, *45*, 65–71. [CrossRef]
24. Liang, S.; Du, X. Permission combination-based scheme for Android mobile malware detection. In Proceedings of the 2014 IEEE International Conference on Communications (ICC), Sydney, Australia, 10–14 June 2014; pp. 2301–2306.
25. Singh, P.; Tiwari, P.; Singh, S. Analysis of Malicious Behavior of Android Apps. *Procedia Comput. Sci.* **2016**, *79*, 215–220. [CrossRef]
26. Suarez-Tangil, G.; Tapiador, J.E.; PerisLopez, P.; Blasco, J. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Syst. Appl.* **2014**, *41*, 1104–1117. [CrossRef]
27. Feng, P.; Ma, J.; Sun, C.; Xu, X.; Ma, Y. A Novel Dynamic Android Malware Detection System with Ensemble Learning. *IEEE Access* **2018**, *6*, 30996–31011. [CrossRef]
28. Kim, T.; Kang, B.; Rho, M.; Sezer, S.; Im, E.G. A Multimodal Deep Learning Method for Android Malware Detection Using Various Features. *IEEE Trans. Inf. Forensics Secur.* **2019**, *14*, 773–788. [CrossRef]
29. El Fiky, A. Visual Detection for Android Malware using Deep Learning. *Int. J. Innov. Technol. Explor. Eng.* **2020**, *10*, 152–156. [CrossRef]
30. Jung, J.; Choi, J.; Cho, S.-J.; Han, S.; Park, M.; Hwang, Y. Android malware detection using convolutional neural networks and data section images. In Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems (RACS'18), New York, NY, USA, 9–12 October 2018; pp. 149–153.
31. Lekssays, A.; Falah, B.; Abufardeh, S. A Novel Approach for Android Malware Detection and Classification using Convolutional Neural Networks. In Proceedings of the 2018 Eleventh International Conference on Contemporary Computing (IC3), Noida, India, 2–4 August 2018; pp. 1–4.
32. Zhang, W.; Luktarhan, N.; Ding, C.; Lu, B. Android Malware Detection Using TCN with Bytecode Image. *Symmetry* **2021**, *13*, 1107. [CrossRef]
33. Zhao, C.; Wang, C.; Zheng, W. Android Malware Detection Based on Sensitive Permissions and APIs. In *Security and Privacy in New Computing Environments (SPNCE 2019)*, 3rd ed.; Li, J., Liu, Z., Peng, H., Eds.; Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering; Springer: Cham, Switzerland, 2019.
34. Mercaldo, F.; Santone, A. Deep learning for image-based mobile malware detection. *J. Comput. Virol. Hacking Tech.* **2020**, *16*, 157–171. [CrossRef]
35. Kural, O.E.; Şahin, D.Ö.; Akleylek, S.; Kılıç, E.; Ömüral, M. Apk2Img4AndMal: Android Malware Detection Framework Based on Convolutional Neural Network. In Proceedings of the 6th International Conference on Computer Science and Engineering (UBMK), Ankara, Turkey, 15–17 September 2021; pp. 731–734.
36. Nazir, F.; Khan, M.U.S.; Khan, N.; Fayyaz, A. Examining Malware Patterns in Android Platform using Sufficient Input Subset (SIS). In Proceedings of the 2023 International Multi-Disciplinary Conference in Emerging Research Trends (IMCERT), Karachi, Pakistan, 4–5 January 2023; pp. 1–5.
37. Al-Fawa'reh, M.; Saif, A.; Jafar, M.T.; Elhassan, A. Malware Detection by Eating a Whole APK. In Proceedings of the 15th International Conference for Internet Technology and Secured Transactions (ICITST), London, UK, 8–10 December 2020; pp. 1–7.
38. Khan, M.A.R.; Kumar, N.; Tripathi, R.C. Detection of Android Malware App through Feature Extraction and Classification of Android Image. *Int. J. Adv. Comput. Sci. Appl. (IJACSA)* **2022**, *13*, 906–914. [CrossRef]

39. Fang, Y.; Gao, Y.; Jing, F.; Zhang, L. Android Malware Familial Classification Based on DEX File Section Features. *IEEE Access* **2020**, *8*, 10614–10627. [CrossRef]
40. Xiao, X.; Yang, S. An Image-Inspired and CNN-Based Android Malware Detection Approach. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 1259–1261.
41. Huang, T.H.-D.; Kao, H.-Y. R2-d2: Color-inspired convolutional neural network (CNN)-based android malware detections. In Proceedings of the 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 10–13 December 2018.
42. Mitsuhashi, R.; Shinagawa, T. Exploring Optimal Deep Learning Models for Image-based Malware Variant Classification. In Proceedings of the IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC), Los Alamitos, CA, USA, 27 June–1 July 2022; pp. 779–788.
43. Alam, S.; Demir, A.K. Mining Android Bytecodes through the Eyes of Gabor Filters for Detecting Malware. *Int. Arab J. Inf. Technol. (IAJIT)* **2023**, *20*, 180–189. [CrossRef]
44. Ding, Y.; Zhang, X.; Hu, J. Android malware detection method based on bytecode image. *J. Ambient. Intell. Humaniz. Comput.* **2020**, *14*, 6401–6410. [CrossRef]
45. Daoudi, N.; Samhi, J.; Kabore, A.K.; Allix, K.; Bissyandé, T.F.; Klein, J. DEXRAY: A Simple, yet Effective Deep Learning Approach to Android Malware Detection Based on Image Representation of Bytecode. In *Deployable Machine Learning for Security Defense (MLHat 2021)*, 3rd ed.; Wang, G., Ciptadi, A., Ahmadzadeh, A., Eds.; Communications in Computer and Information Science; Springer: Cham, Switzerland, 2021.
46. Bakour, K.; Ünver, H.M. DeepVisDroid: Android malware detection by hybridizing image-based features with deep learning techniques. *Neural Comput. Appl.* **2021**, *33*, 11499–11516. [CrossRef]
47. Martín, A.; Lara-Cabrera, R.; Camacho, D. Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset. *Inf. Fusion* **2019**, *52*, 128–142. [CrossRef]
48. Martín, A.; Lara-Cabrera, R.; Camacho, D. A new tool for static and dynamic Android malware analysis. In Proceedings of the Data Science and Knowledge Engineering for Sensing Decision Support (FLINS 2018), Belfast, UK, 21–24 August 2018; pp. 509–516.
49. Khoa, N.H.; Cam, N.T.; Pham, V.-H.; Nguyen, A.G.-T. Detect Android malware by using deep learning: Experiment and Evaluation. In Proceedings of the 2021 5th International Conference on Machine Learning and Soft Computing (ICMLSC'21), Association for Computing Machinery, New York, NY, USA, 29–31 January 2021; pp. 129–134.
50. Singh, J.; Thakur, D.; Gera, T.; Shah, B.; Abuhmed, T.; Ali, F. Classification and Analysis of Android Malware Images Using Feature Fusion Technique. *IEEE Access* **2021**, *9*, 90102–90117. [CrossRef]
51. Zegzhda, P.D.; Pavlenko, E.; Ignatev, G.M. Applying deep learning techniques for Android malware detection. In Proceedings of the 11th International Conference on Security of Information and Networks (SIN'18), New York, NY, USA, 10–12 September 2018.
52. Androguard. Available online: https://github.com/androguard/androguard (accessed on 26 February 2024).
53. Darwaish, A.; Naït-Abdesselam, F. RGB-based Android Malware Detection and Classification Using Convolutional Neural Network. In Proceedings of the GLOBECOM 2020—2020 IEEE Global Communications Conference, Taipei, Taiwan, 7–11 December 2020; pp. 1–6.
54. Lan, T.; Darwaish, A.; Naït-Abdesselam, F.; Gu, P. Defensive Randomization Against Adversarial Attacks in Image-based Android Malware Detection. In Proceedings of the 2023 IEEE International Conference on Communications (ICC): Communication and Information System Security Symposium, Rome, Italy, 28 May–1 June 2023; pp. 5072–5077.
55. Apktool. A Tool for Reverse Engineering Android Apk Files. Available online: https://apktool.org/ (accessed on 26 February 2024).
56. Wang, C.; Zhang, L.; Zhao, K.; Ding, X.; Wang, X. AdvAndMal: Adversarial Training for Android Malware Detection and Family Classification. *Symmetry* **2021**, *13*, 1081. [CrossRef]
57. Zhao, Y.; Qian, Q. Android Malware Identification Through Visual Exploration of Disambly Files. *Int. J. Netw. Secur.* **2018**, *20*, 1061–1073.
58. Arp, D.; Spreitzenbarth, M.; Hubner, M.; Gascon, H.; Rieck, K.; Siemens, C. Drebin: Effective and explainable detection of android malware in your pocket. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 23–26 February 2014.
59. Wei, F.; Li, Y.; Roy, S.; Ou, X.; Zhou, W. Deep ground truth analysis of current android malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2017)*, 2nd ed.; Polychronakis, M., Meier, M., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2017.
60. Allix, K.; Bissyandé, T.F.; Klein, J.; Traon, Y.L. Androzoo: Collecting millions of android apps for the research community. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR'16), New York, NY, USA, 14–22 May 2016; pp. 468–471.
61. Lashkari, A.H.; Kadir, A.F.A.; Taheri, L.; Ghorbani, A.A. Toward Developing a Systematic Approach to Generate Benchmark Android Malware Datasets and Classification. In Proceedings of the 52nd IEEE International Carnahan Conference on Security Technology (ICCST), Montreal, QC, Canada, 22–25 October 2018; pp. 1–7.
62. Ksibi, A.; Zakariah, M.; Almuqren, L.A.; Alluhaidan, A.S. Deep Convolution Neural Networks and Image Processing for Malware Detection. Preprint (Version 1). 27 January 2023. Available online: https://www.researchsquare.com/article/rs-2508967/v1 (accessed on 20 May 2024).

63. Mahdavifar, S.; Kadir, A.F.A.; Fatemi, R.; Alhadidi, D.; Ghorbani, A.A. Dynamic Android Malware Category Classification using Semi-Supervised Deep Learning. In Proceedings of the 2020 IEEE International Conference on Dependable, Autonomic and Secure Computing, International Conference on Pervasive Intelligence and Computing, International Conference on Cloud and Big Data Computing, International Conference on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), Calgary, AB, Canada, 17–22 August 2020; pp. 515–522.
64. Mahdavifar, S.; Alhadidi, D.; Ghorbani, A.A. Effective and Efficient Hybrid Android Malware Classification Using Pseudo-Label Stacked Auto-Encoder. *J. Netw. Syst. Manag.* **2022**, *30*, 1–34. [CrossRef]
65. Bert Base Model (Uncased). Available online: https://huggingface.co/bert-base-uncased (accessed on 26 February 2024).
66. Pillow (PIL Fork). Available online: https://pillow.readthedocs.io/en/stable/index.html (accessed on 26 February 2024).
67. Park, J.H.; Lee, J.; Lee, K.; Min, J.; Ko, H. FBRNN: Feedback recurrent neural network for extreme image super-resolution. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops, Seattle, WA, USA, 14–19 June 2020; pp. 2021–2028.
68. Zhuang, B.; Shen, C.; Tan, M.; Liu, L.; Reid, I. Structured binary neural networks for accurate image classification and semantic segmentation. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Long Beach, CA, USA, 15–20 June 2019; pp. 413–422.
69. Optuna. Optimize Your Optimization. Available online: https://optuna.org/ (accessed on 26 February 2024).
70. Sogut, E.; Erdem, O.A. A Multi-Model Proposal for Classification and Detection of DDoS Attacks on SCADA Systems. *Appl. Sci.* **2023**, *13*, 5993. [CrossRef]
71. Jo, J.; Cho, J.; Moon, J.A. Malware Detection and Extraction Method for the Related Information Using the ViT Attention Mechanism on Android Operating System. *Appl. Sci.* **2023**, *13*, 6839. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.