International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Revue d'Intelligence Artificielle
Vol. 37, No. 4, August, 2023, pp. 857-869
Journal homepage: http://iieta.org/journals/ria

Feature Selection for Android Malware Detection with Random Forest on Smartphones

Ibrahim Mahmood Ibrahim[1*], Amira Bibo Sallow[2]

[1] Technical College of Informatics-Akre, Duhok Polytechnic University, Duhok 42001, Iraq
[2] Technical College of Administration-Duhok, Duhok Polytechnic University, Duhok 42001, Iraq

Corresponding Author Email: ibrahim.mahmood@dpu.edu.krd

https://doi.org/10.18280/ria.370405

Received: 12 June 2023
Revised: 22 July 2023
Accepted: 28 July 2023
Available online: 31 August 2023

Keywords:
machine learning, random forest algorithm, malware, static analysis, dynamic analysis, hybrid analysis, permissions, API calls, feature selection, Android application

ABSTRACT

Android smartphones, integral to everyday life, offer a multifunctional platform for storing and managing sensitive personal data. However, the ubiquity of Android applications intensifies their vulnerability to malicious applications. This study presents the Static Dynamic Hybrid Feature Extraction (SDHFE) tool, a lightweight automation tool designed for the efficient analysis of Android applications by extracting features from a variety of sources. The research generated multiple datasets, each representing different feature categories and their combinations. A novel approach to improve Android malware detection on smartphones is introduced, leveraging the random forest algorithm. Multiple models were created and evaluated using metrics such as accuracy, precision, recall, and F1 score. The model trained on a dataset comprising permissions and intents achieved the highest average scores, 99.2%, thus outperforming other models. A comparative analysis was conducted to evaluate the efficiency of the SDHFE tool against two widely used tools, APKtool and Androguard, in static feature extraction. The results demonstrated that the SDHFE tool significantly reduced disassembly and analysis time, outperforming APKtool and Androguard by factors of 2.2 and 4.6, respectively. While this research provides valuable insights into Android malware detection, it is important to acknowledge potential limitations. The dynamic nature of malware behavior could affect the generalizability of our approach. Despite these potential limitations, the results underscore the effectiveness of our proposed method for enhancing malware detection in Android smartphones.

1. INTRODUCTION

The recent evolution in mobile device technology has led to the proliferation of smartphones, which operate on various operating systems (OS) such as Android, iOS, Windows, Symbian, and Blackberry [1]. Among these, the Android OS is the most prevalent, offering a multitude of services including calls, messaging, multimedia, shopping, banking, internet browsing, file storage, and file sharing. The growing popularity and open-source nature of Android have unfortunately resulted in a significant surge in malicious attacks on both official and third-party Android app stores [2].

Smartphones, due to their extensive service range and the storage of sensitive data, have become integral to daily life. This ubiquity has given rise to an increase in the development of various types of malware aimed at acquiring sensitive information without user consent [3]. Malware, or malicious code, encompasses several types of harmful or intrusive software such as viruses, spyware, worms, Trojan horses, backdoors, and rootkits. Such software is typically designed with the explicit intent of attacking systems, whether to destroy file systems, steal data, or execute other undesirable activities [4, 5].

In response to the growing threat of malicious code, researchers have developed various Intrusion Detection Systems (IDS) to protect mobile devices [6]. These systems aim to counteract the harmful actions of malware, which range from stealing sensitive information to damaging the system.

Many procedures for analyzing the behavior of Android apps have been proposed, and they can be broken down into three classes: static, dynamic, and hybrid analysis approaches. These approaches are mainly used to identify the behavior of the app as either normal or malicious [7]. The static approach scans the application's source code without executing it to detect the malware application. It employs reverse engineering methods to obtain the source code from the APK package. Several static features such as intents, permissions, activities, API calls, and opcodes features can be extracted and further used in the malware detection strategy [8]. The most crucial dynamic features are system calls and network traffic, which are retrieved from the executing app in the controlled environment. Every Android app relies on the operating system to deliver the most fundamental resources and services it needs to run. Due to the architecture of the Android system, applications cannot communicate directly with the operating system; therefore, the application has to use system calls to do specific operations, such as file opening, reading, writing, and closing [9]. The hybrid analysis utilizes both static and dynamic features [7].

In this study, we conducted the extraction and analysis of key features, including permissions, intents, API calls, and system calls. To facilitate this process, a lightweight tool was developed by the author specifically for extracting and analyzing these features. Many machine-learning models were created and trained on the extracted features to discover the optimal feature set for Android malware detection on the smartphone. The main contributions of this work can be summarized as follows:

1. We developed an automated and lightweight tool called Static Dynamic Hybrid Feature Extraction (SDHFE) to analyze the behaviors of Android applications. This tool enables researchers to analyze an unlimited number of apps sequentially without human intervention or restrictions. It offers three modes of analysis: static, dynamic, and hybrid, providing flexibility for researchers.
2. The performance of the SDHFE tool was evaluated by comparing it with two of the most commonly used tools for extracting static features: APKtool and Aandroguard. The results reveal that the SDHFE tool outperforms in terms of speed, providing quicker de-compilation and extraction of features from Android apps.
3. A total of 500 benign and 500 malware Android samples underwent analysis using the proposed tool, which successfully extracted four distinct types of features: permissions, intents, API calls, and system calls. Subsequently, ten distinct datasets were constructed based on the extracted features.
4. Based on the created datasets, we developed ten models utilizing the random forest algorithm to determine the most effective feature set for detecting malicious applications on smartphones.

The remaining sections of our paper are organized as follows. Section 2 provides related works. Section 3 shows the background theory, and Section 4 presents the methodology. Section 5 shows the experimental environment and result analysis. Sections 6 and 7 provide a discussion and conclusion.

2. RELATED WORKS

To do research in the area of Android security, we'll need to gather a set of features that Android apps have to analyze. The author grouped related studies into groups based on the type of features used in their studies.

2.1 Studies based on static features

Many researchers have developed and implemented their own tools for extracting features from Android apps. In 2019, Kapoor et al. [10] presented a method for the classification of Android applications based on permissions. The proposed system collects data from many sources. Benign samples are collected from the Google Play store while the malware samples are collected from different websites like virus share and zeltser. They have developed a Python script to extract static features (permissions) and save them in a CSV file. In 2020, Bibi et al. [11] proposed a deep learning model based on Gated Recurrent Unit GRU for the detection of malicious applications in the Android system. The proposed model can classify applications into benign, backdoor, and Trojans. The authors collected samples from AMD and Androzoo datasets. A Python script was developed to extract useful information and generate feature vectors from a manifest file. The developed script thoroughly analyses the manifest file and extracts many features like permissions, API calls, and intents. In 2020, Sirisha et al. [12] suggested a sequential neural network model in order to predict the presence of malware in Android APK files received from the web or play store. The dataset consisting of 398 APK files having 331 features was used for training the model in the proposed model. During the test, the neural network model will be tested using data from the Android play store as well as malicious sites, such as Droidbench, which includes malicious and benign APK files. The permissions were extracted from the APK file by using the Androguard tool. In 2019, Hr [13] developed a fully connected deep-learning model for detecting malicious applications. Benign data samples are collected from the Google Play store, while the malware samples are collected from the virus share repository. The Androguard tool was used for analyzing APK files to extract permissions from the application, and the random forest was used for feature selection. In 2018, Koli [14] presented the RanDroid method that uses several machine learning algorithms like random forest, naive Bayes, support vector machine, and decision tree for classifying malware applications. The RanDroid method depends on many features, such as API calls and requested permissions, along with other application features such as reflection and native code. The researchers used the Androguard static analysis tool to extract features like permissions and API calls. In 2020, Alsoghyer and Almomani [15] presented a model that deeply analyzed the extracted permissions that enabled them to differentiate ransomware from benign applications on the Android device. The important permissions are used and are comparable between ransomware and benign applications. The presented study takes a proactive approach to identifying ransomware before damaging the device. The present work used the APK tool to decompile APK files and then extract features. In 2018, Zhao et al. [16] suggested a deep neural network-based mobile Android malware detection system that employs optimized deep neural networks to learn from opcode sequences to detect malicious code. With the proposed method, the optimized CNN was trained several times on the operation code (Opcode) sequence extracted from the Android app after the de-compilation process by the APK tool. Opcodes represent specific instructions that the Android runtime executes to perform various tasks within the app. The suggested model collected data from many sources, such as Drebin for malicious applications and many Chinese app markets for benign applications. In 2019, Ma et al. [17] presented an ensemble model that detects malware applications based on the machine learning algorithms. The proposed model constructs a control flow graph from the source code after the application is de-compiled by using the flowdroid framework, then extracts API calls from the control flow graph and builds three datasets. In 2022, Kumar et al. [18] introduces a methodology for malware detection in Android applications by extracting features such as permissions, intent filters, activities, and services from the manifest file. The androguard utility is employed to disassemble the code and identify suspicious API calls from the dex code. To improve retrieval efficiency, the extracted features are serialized in feather data format. Finally, the XGBoost algorithm is utilized for effective malware detection.

In this section of the related works, it is observed that both the APK tool and the Androguard tool have been utilized in most studies for the analysis of Android apps. The APK tool is developed in the Java language used for reverse engineering Android apps. It has the capability to decode Android apps and create decompiled resources, encompassing components like the manifest and smali files. Androguard is a Python-based reverse engineering tool for Android apps, possesses the ability to decode resources and additionally perform bytecode disassembly to convert them into Java source code [19]. The

APK tool lacks built-in functions or APIs for feature extraction. Researchers utilizing this tool need to develop their own programs to extract features from files extracted from Android apps. In contrast, the Androguard tool provides functions to directly extract static features from files derived from Android apps. Both the APK tool and the Androguard tool only support static features.

2.2 Studies based on dynamic features

In 2018, Feng et al. [20] developed an approach that depends on a dynamic analysis called EnDroid for detecting malware apps based on multiple dynamic features. For feature extraction, the authors used Droidbox and the strace tool to extract dynamic features such as network traffic, file operations, and system calls from the executing app in the emulator. Based on these features, the developed method can achieve an accuracy of 96.32%. In 2019, Esmaeili and Shahriari [21] proposed a method named PODBot for detecting mobile phone botnets that utilize both network and host-based metrics. The detection is based on application features like permissions, APIs, and network traffic analysis. Androguard and MobSF tools were used to perform analysis. In 2020, Lê et al. [22] proposed a method of machine learning to identify Android malware apps. The features that are used to train machine learning are built based on the behavior, requisite permissions, and other features of malicious applications extracted by the Dexdump tool. In 2018, Kumar et al. [23] proposed a framework for machine learning (ML) to counter mobile threats rapidly. The model depends on the network's flow-based features. The proposed model utilizes 40 time-based network traffic features derived from malicious and benign apps in real time. The Cuckoo sandbox and the Anubis sandbox are publicly available and were used to generate network traffic. In 2023, Manzil and Naik [24] Introduced a novel approach for generating feature vectors by employing Huffman encoding. The experimental findings demonstrate that this innovative strategy significantly improves the effectiveness of the malware detection model. By extracting dynamic behavior patterns of malware through the utilization of system call frequencies as features, the proposed method contributes to the detection process. The performance of the model is evaluated by applying deep learning and machine learning techniques. In 2020, Mahdavifar et al. [25] introduced a simple and effective Android classification framework for malware on the basis of mining system-call dynamically observed behaviors. The authors depended on the CopperDroid virtual machine to extract dynamic features like system calls. The framework is built on a deep neural network that uses a semi-supervised technique to input behaviors that are dynamically observed and that allows the classification of the categories of malware.

2.3 Studies based on hybrid features

In 2018, Arora and Peddoju [26] introduced a hybrid system for detecting malware, named NTPDroid. This system effectively combines network traffic features and permissions extracted from applications. The approach employed the FP-Growth technique to identify prevalent patterns present in both malicious and benign datasets. The author utilize APK tool to extract static features. Importantly, the results indicated that integrating network traffic features with permissions resulted in improved detection rates, surpassing the efficacy of using either network traffic features or permissions in isolation. In 2019, Garg and Baliyans [27] suggested an ensemble approach comprising a number of algorithms, including support vector machine, multilayer perceptron, ripple down rule learner, and rule-based classification tree. This ensemble model works concurrently to create a resilient framework capable of accurately and efficiently distinguishing between benign and malware. The author employed many tools to extract features such as the APK tool, ADB, and strace tools. The extraction process involves capturing an array of static and dynamic attributes from applications and devices, encompassing API calls, permissions, and system components, network traffic, and battery usage. Experimental outcomes demonstrated exceptional performance, with the proposed system achieving an impressive accuracy level of 98.27%.

Upon reviewing the related works, the author observed that the previous works mainly focused on the accuracy of the models or developing new methods for Android malware detection, with less attention given to the complexities associated with the extraction of features. Several of the developed models have demonstrated promising results; however, when subjected to testing on mobile devices, they may encounter challenges in acquiring features from the installed applications. This can potentially impact the overall device's performance. Therefore, the main aim of this study is to analyze Android applications and extract features from many sources to get lightweight features that achieve high accuracy with less computation on mobile devices.

3. BACKGROUND THEORY

3.1 Android application package

Every Android application is distributed in the form of Application Package (APK) files. They are primarily ZIP files that contain all of the data and metadata essential for an application to run [28]. The main component of the APK file is illustrated in Figure 1 [27-29].

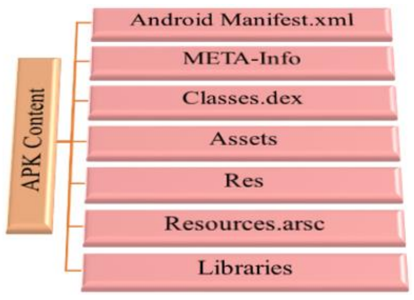


Figure 1. APK components [27]

AndroidManifes.xml is a critical file that provides the operating system with fundamental information about the application in order to ensure its proper execution. It includes a list of all the components required for the app to run, and permissions required to access critical resources. The **META-INF folder** stores the application's signature information. The signature information can be used to validate the APK's integrity. **classes.dex file** contains references to any classes or methods used by the application. Essentially, each activity, object, or fragment in the code base will be converted to bytes within a Dex file that can be executed as an Android application. **Assets folder**: used to keep static files that must be packed together with the APK, such as text, XML, and HTML. **Res folder**: contains external resources like images, videos, fonts, and languages to support the settings of a specific device. **resources.arsc file**: Compiled Android-generated application resource. For a precompiled binary translation resource, it stores value types of resources as well as other non-asset types of resource-related information. **Libraries folder**: contains compiled libraries that are particular to different types of processors and are intended for use with the appropriate application [29].

3.2 Android apps analysis techniques

There are three major techniques used in analyzing mobile apps, which are static analysis, dynamic analysis, and hybrid analysis techniques. Researchers use these techniques to analyze the behavior of mobile apps to determine malicious activity produced by mobile apps [30].

Static analysis is a method that analyzes mobile apps without executing them. The procedure aids in the understanding of code structure as well as the functions it will execute. Classes files and Android Manifest files are the most crucial files in the static analysis process. From these files, many features can be extracted, such as permissions, intents, API calls, and many other features. The process of extracting static features is very fast and doesn't need many resources [31]. In the dynamic approach, the application is executed in a controlled environment, monitors the running code, and inspects its interaction with the system. The dynamic analysis traces many features of an application and system, such as system calls, system components, network traffic, CPU consumption, memory usage, battery usage, and user interaction with the system [32]. Hybrid analysis gathers static and dynamic features together to detect malware applications. The benefit of using hybrid analysis is to improve the accuracy of the system, while the disadvantage is that the required time for hybrid analysis will be more than that required for static or dynamic analysis [33].

4. METHODOLOGY

Figure 2 depicts the framework of the Android malicious detection system. The system employed many types of features obtained from various sources, and many datasets (DS) are built based on a single feature category and a possible combination of two feature categories to identify the optimal feature sets suitable for devices with limited resources, such as smartphones. This approach aimed to effectively detect applications exhibiting malicious behavior while considering the constraints imposed by resource-constrained devices.

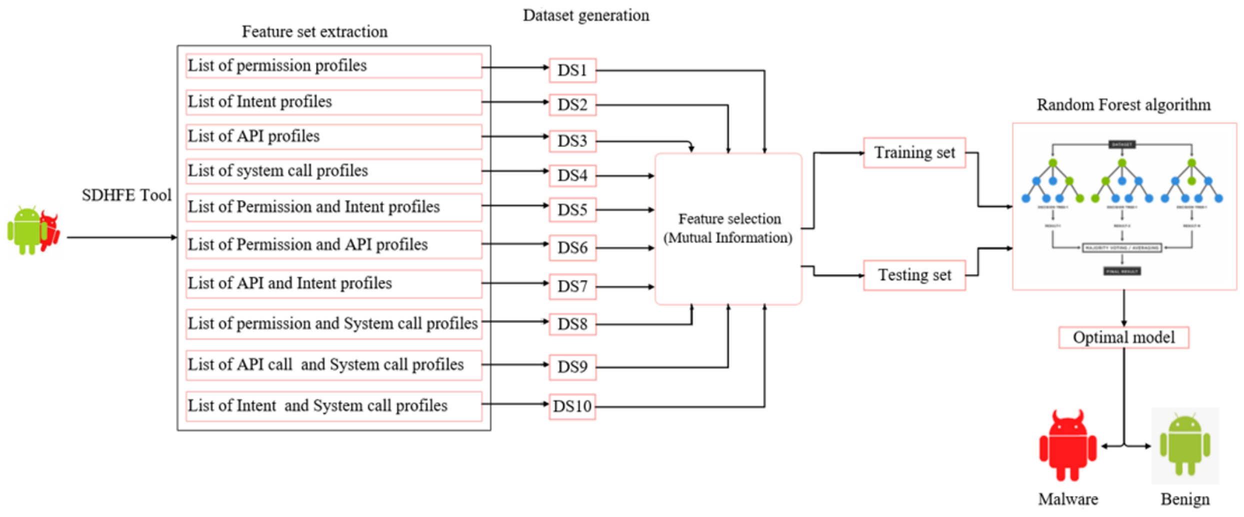

Dataset generation
Feature set extraction
SDHFE Tool
List of permission profiles
List of Intent profiles
List of API profiles
List of system call profiles
List of Permission and Intent profiles
List of Permission and API profiles
List of API and Intent profiles
List of permission and System call profiles
List of API call and System call profiles
List of Intent and System call profiles
DS1
DS2
DS3
DS4
DS5
DS6
DS7
DS8
DS9
DS10
Feature selection (Mutual Information)
Training set
Testing set
Random Forest algorithm
Optimal model
Malware
Benign

Figure 2. Schematic representation of the proposed system

4.1 Collecting samples

Numerous resources provide Android app samples specifically intended for research purposes. For the present study, 500 samples of malware were gathered from diverse databases, namely Drebin (https://www.sec.tu-bs.de/~danarp/drebin/), CICMalDroid 2020 (https://www.unb.ca/cic/datasets/maldroid-2020.html), the Android Botnet dataset (https://www.unb.ca/cic/datasets/android-botnet.html), and Malware Bazaar (https://bazaar.abuse.ch/browse/tag/apk/). For benign apps, this study collects samples from Google Play (https://play.google.com/store/apps), and the AndroZoo databases (https://androzoo.uni.lu/).

4.2 SDHFE tool

Most researchers analyze Android applications for security purposes to determine whether the application is malware or benign. Therefore, researchers are looking for a tool that helps them analyze mobile applications and extract the essential features. Many reverse engineering tools on the market can perform static and dynamic analyses of Android APK files. Still, no supporting tools that offer three fundamental analyses (static, dynamic, and hybrid) and have the capability to automatically extract features from a bulk of APK files without the need for human intervention during the analysis process. This paper presents the development of the SDHFE tool, a comprehensive solution offering three distinct analysis modes. The SDHFE tool is a lightweight and automated tool, developed using shell script programming. It relies on essential libraries like AXMLPrinter and Baksmali. This tool is designed to the extraction of features and enable the analysis of a large number of APK files within a short time frame without the need for human intervention. Figure 3 shows the general mechanism of the SDHFE tool.

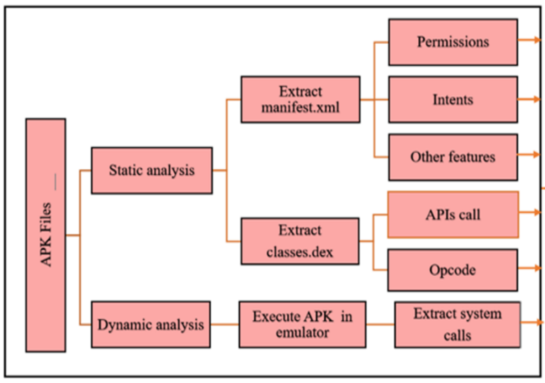


Figure 3. Overview of the SDHFE tool's organizational structure

4.3 Feature set extraction

This study involves the analysis of 1000 Android apps, which are evenly distributed between malware and benign applications. Our emphasis centers on permissions, intents, API calls, system calls, and possible combinations of two feature categories. The analysis is conducted using the SDHFE tool.

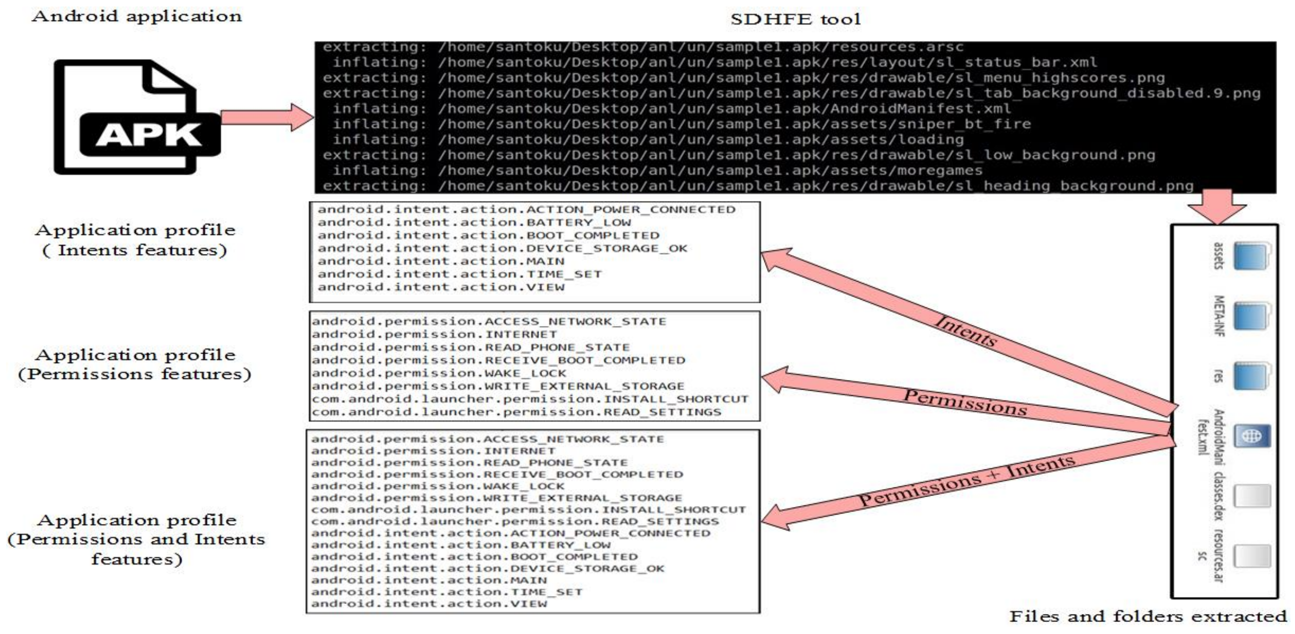
4.3.1 Feature sets extracted from the manifest file

Permissions and intent filters are important features in the Android manifest file that govern access to devise resources and define how the application interacts with other components. However, these features are susceptible to exploitation by malicious applications. For instance, the android. permission.SEND_SMS permission is commonly utilized in benign applications for sending and receiving SMS messages, it can be maliciously employed to send unauthorized premium-rate messages or utilized in phishing attacks, where deceptive SMS messages containing malicious links or requests for sensitive information are sent. Another exploit involves a malicious app registering itself to receive the "android.intent.action.PHONE_STATE" intent, enabling it to eavesdrop on phone conversations. This unauthorized access grants the app the ability to obtain call-related information, such as phone numbers, call durations, and even the audio content of the calls. Figure 4 illustrates the pseudocode algorithm for extracting permissions and intents, as well as generating profiles for applications by the SDHFE tool.

```
1. start
2. Input the path of APK files to the SDHFE tool.
3. For i=1 to length of APK files
    a. Extract files from the APK_i
    b. Search for AndroidManifest.xml in APK_i and convert it to human readable format
    c. Extract permissions from AndroidManifest.xml and add to permission profile of APK_i
    d. Extract intents from AndroidManifest.xml and add to intent profile of APK_i
4. If i < length of APK file got step 2 otherwise go to step 5
5. end
```

Figure 4. Generating profiles for APK files based on permissions and intents

The process of analyzing and extracting features from a manifest file is relatively fast and lightweight as the manifest file is typically small. Figure 5 depicts the steps of deriving features from an app using the SDHFE tool, which involves parsing the structured XML data present within the manifest file, resulting in the retrieval of crucial information.


Android application
SDHFE tool
APK
Application profile (Intents features)
Application profile (Permissions features)
Application profile (Permissions and Intents features)
Intents
Permissions
Permissions + Intents
assets
META-INF
res
AndroidManifest.xml
classes.dex
resources.arsc
Files and folders extracted

```
extracting: /home/santoku/Desktop/anl/un/sample1.apk/resources.arsc
 inflating: /home/santoku/Desktop/anl/un/sample1.apk/res/layout/sl_status_bar.xml
extracting: /home/santoku/Desktop/anl/un/sample1.apk/res/drawable/sl_menu_highscores.png
extracting: /home/santoku/Desktop/anl/un/sample1.apk/res/drawable/sl_tab_background_disabled.9.png
 inflating: /home/santoku/Desktop/anl/un/sample1.apk/AndroidManifest.xml
 inflating: /home/santoku/Desktop/anl/un/sample1.apk/assets/sniper_bt_fire
 inflating: /home/santoku/Desktop/anl/un/sample1.apk/assets/loading
extracting: /home/santoku/Desktop/anl/un/sample1.apk/res/drawable/sl_low_background.png
 inflating: /home/santoku/Desktop/anl/un/sample1.apk/assets/moregames
extracting: /home/santoku/Desktop/anl/un/sample1.apk/res/drawable/sl_heading_background.png
```

```
android.intent.action.ACTION_POWER_CONNECTED
android.intent.action.BATTERY_LOW
android.intent.action.BOOT_COMPLETED
android.intent.action.DEVICE_STORAGE_OK
android.intent.action.MAIN
android.intent.action.TIME_SET
android.intent.action.VIEW
```

```
android.permission.ACCESS_NETWORK_STATE
android.permission.INTERNET
android.permission.READ_PHONE_STATE
android.permission.RECEIVE_BOOT_COMPLETED
android.permission.WAKE_LOCK
android.permission.WRITE_EXTERNAL_STORAGE
com.android.launcher.permission.INSTALL_SHORTCUT
com.android.launcher.permission.READ_SETTINGS
```

```
android.permission.ACCESS_NETWORK_STATE
android.permission.INTERNET
android.permission.READ_PHONE_STATE
android.permission.RECEIVE_BOOT_COMPLETED
android.permission.WAKE_LOCK
android.permission.WRITE_EXTERNAL_STORAGE
com.android.launcher.permission.INSTALL_SHORTCUT
com.android.launcher.permission.READ_SETTINGS
android.intent.action.ACTION_POWER_CONNECTED
android.intent.action.BATTERY_LOW
android.intent.action.BOOT_COMPLETED
android.intent.action.DEVICE_STORAGE_OK
android.intent.action.MAIN
android.intent.action.TIME_SET
android.intent.action.VIEW
```

Figure 5. Process of sample analysis and feature extraction from the manifest file

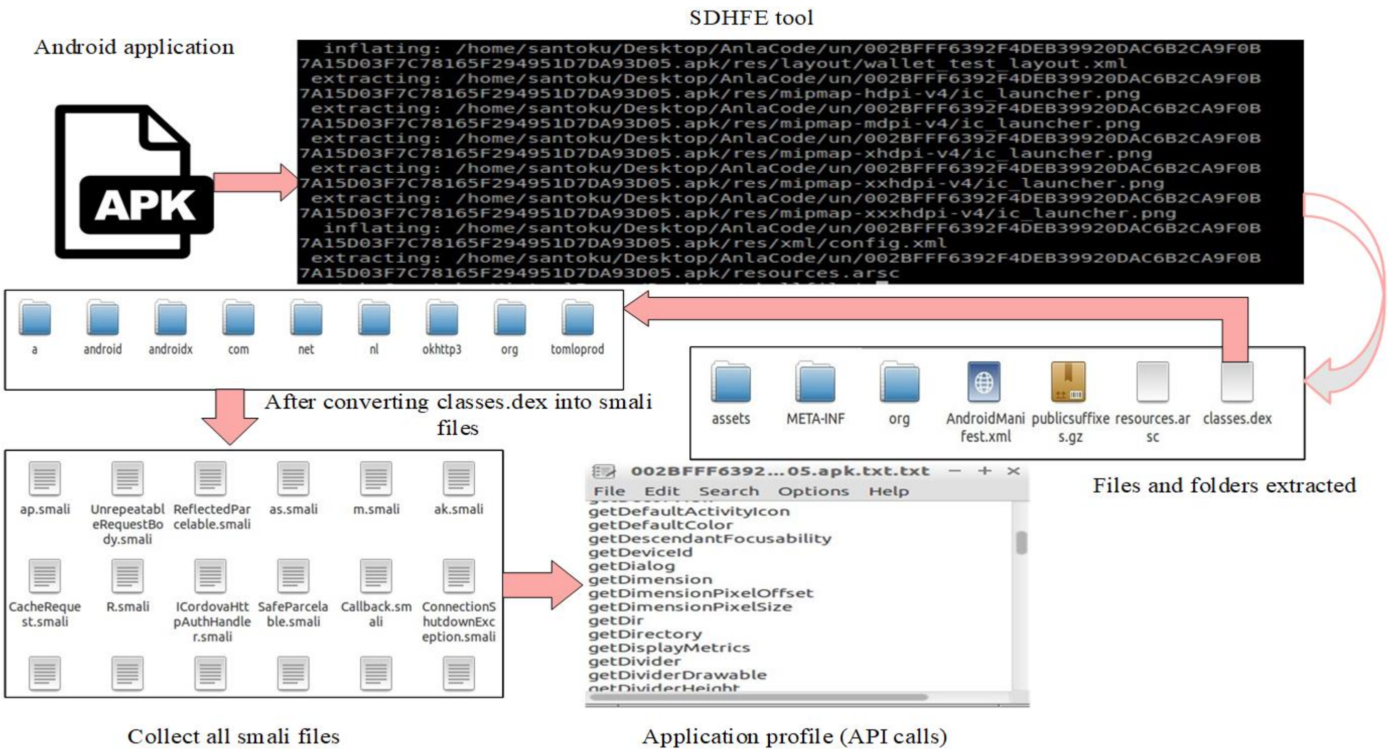
4.3.2 Feature sets extracted from the source code

Source code analysis provides deep insight into the inner workings of an Android application. By examining the source code, security analysts can understand how the app functions, accesses sensitive data, communicates with external services, and handles user inputs. The most important features in the source code are API calls. Malicious applications often incorporate API calls that enable unauthorized access to sensitive information or smartphone resources. A malicious app can exploit a certain API call to perform some action without user consent, such as getDeviceId(), getSubscriberId(), and getNetworkOperator(), from the telephony manager class. These APIs provide access to sensitive device information of the user and can lead to privacy breaches and unauthorized data collection. Another prominent API call that malware app developers frequently used is the sendTextMessage() method. This API is intended for sending SMS messages programmatically, but its misuse can lead to various malicious activities. The SDHFE tool plays a crucial role in identifying and extracting these API calls from the source code. Figure 6 provides an illustrative pseudocode algorithm that outlines the process of extracting API calls and generating profiles for applications using the SDHFE tool.

Extracting features from the source code (classes.dex file) after converting it into smali files demands more time and computational resources compared to extracting features from manifest files. This conversion process involves creating multiple folders and smali files that mirror the package structure of the Android application. The number of folders and smali files created varies based on the complexity and size of the Android application. The procedure of extracting API calls from a class.dex by the SDHFE tool is depicted in Figure 7.

```
1. start
2. Input the path of APK files to the SDHFE tool.
3. For i=1 to length of APK files
      a. Extract files from the APK_i
      b. Search for Classes.dex in APK_i and convert it to smali files.
      c. Search API calls from all smali files and add to API calls profile of APK_i
4. If i < length of APK file got step 2 otherwise go to step 5
5. End
```

Figure 6. Generating profiles for APK files based on API calls



```
  inflating: /home/santoku/Desktop/AnlaCode/un/002BFFF6392F4DEB39920DAC6B2CA9F0B
7A15D03F7C78165F294951D7DA93D05.apk/res/layout/wallet_test_layout.xml
 extracting: /home/santoku/Desktop/AnlaCode/un/002BFFF6392F4DEB39920DAC6B2CA9F0B
7A15D03F7C78165F294951D7DA93D05.apk/res/mipmap-hdpi-v4/ic_launcher.png
 extracting: /home/santoku/Desktop/AnlaCode/un/002BFFF6392F4DEB39920DAC6B2CA9F0B
7A15D03F7C78165F294951D7DA93D05.apk/res/mipmap-mdpi-v4/ic_launcher.png
 extracting: /home/santoku/Desktop/AnlaCode/un/002BFFF6392F4DEB39920DAC6B2CA9F0B
7A15D03F7C78165F294951D7DA93D05.apk/res/mipmap-xhdpi-v4/ic_launcher.png
 extracting: /home/santoku/Desktop/AnlaCode/un/002BFFF6392F4DEB39920DAC6B2CA9F0B
7A15D03F7C78165F294951D7DA93D05.apk/res/mipmap-xxhdpi-v4/ic_launcher.png
 extracting: /home/santoku/Desktop/AnlaCode/un/002BFFF6392F4DEB39920DAC6B2CA9F0B
7A15D03F7C78165F294951D7DA93D05.apk/res/mipmap-xxxhdpi-v4/ic_launcher.png
  inflating: /home/santoku/Desktop/AnlaCode/un/002BFFF6392F4DEB39920DAC6B2CA9F0B
7A15D03F7C78165F294951D7DA93D05.apk/res/xml/config.xml
 extracting: /home/santoku/Desktop/AnlaCode/un/002BFFF6392F4DEB39920DAC6B2CA9F0B
7A15D03F7C78165F294951D7DA93D05.apk/resources.arsc
```

```
getDefaultActivityIcon
getDefaultColor
getDescendantFocusability
getDeviceId
getDialog
getDimension
getDimensionPixelOffset
getDimensionPixelSize
getDir
getDirectory
getDisplayMetrics
getDivider
getDividerDrawable
getDividerHeight
```

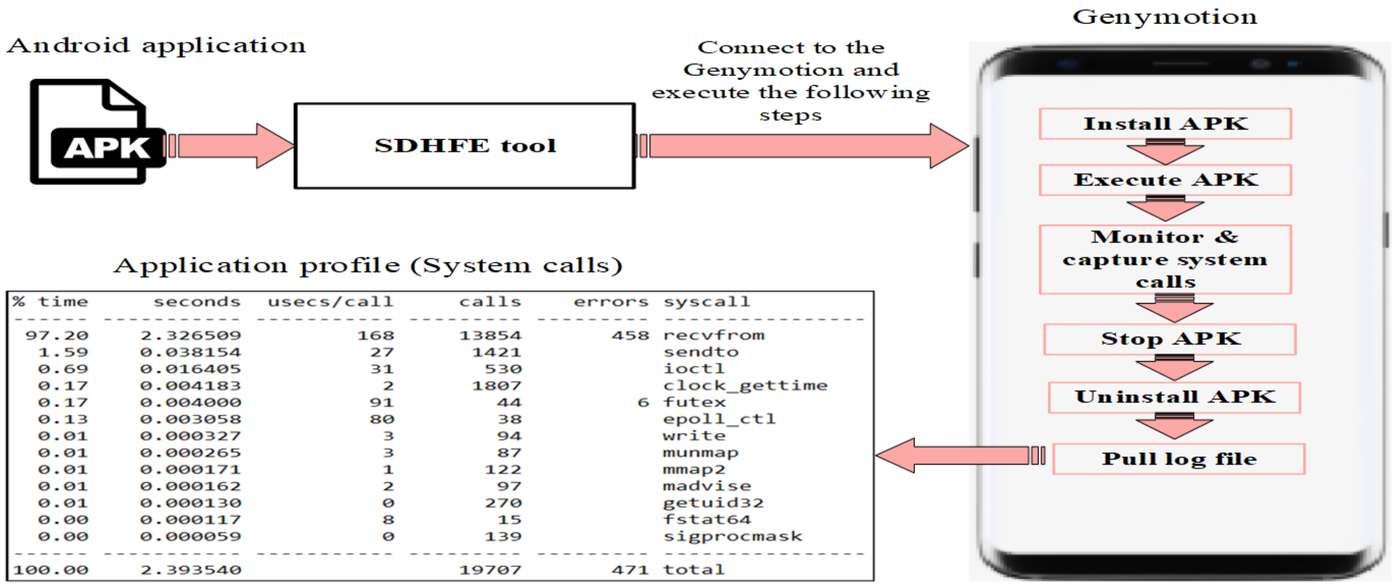
Figure 7. Process of sample analysis and feature extraction from the class file

4.3.3 Feature sets extracted from application behavior

Monitoring and analyzing system calls made by the application can reveal its interaction with the underlying operating system. Features such as the frequency, types, and sequences of system calls can provide insights into the app's resource utilization, file operations, network communication, and potentially malicious activities. For example, the "exec" system call is used to execute a command or launch an external program from within an application. A malicious app can abuse the "exec" system call to execute arbitrary commands or launch malicious programs without the user's knowledge or consent. The process of capturing system calls by the SDHFE tool passes through the following steps: 1) install the APK file on the emulated device (Genymotion) with the help of the ADB tool. 2) Start running the installed app. 3) Retrieve the process id of the currently running. 4) Trace the running process by process id for a specific time (here we used the monkey runner to send 500 pseudo-random events to the running application). 5) Terminate the running process using the kill process id. 6) Remove the installed application. 7) Pull the log file from the emulated device containing the application's system call and generate a profile for the application. Steps 1 to 7 are repeated for every application. Figure 8 provides a visual representation of dynamic analysis using the SDHFE tool to examine an application's behavior, including system calls. Dynamic analysis typically demands more time and computational resources compared to analyzing manifest and classes.dex files due to the following reasons:

1. Monitoring system calls involves tracking the interactions between the app and the operating system in real-time. This real-time analysis demands continuous monitoring, data collection, and processing, which can be resource-intensive.

2. Monitoring system calls may involve tracking resource usage such as memory, file access, network activity, and more. Collecting and analyzing this information further contributes to the computational load.


Android application
APK
SDHFE tool
Connect to the Genymotion and execute the following steps
Genymotion
Install APK
Execute APK
Monitor & capture system calls
Stop APK
Uninstall APK
Pull log file
Application profile (System calls)

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 97.20    2.326509         168     13854       458 recvfrom
  1.59    0.038154          27      1421           sendto
  0.69    0.016405          31       530           ioctl
  0.17    0.004183           2      1807           clock_gettime
  0.17    0.004000          91        44         6 futex
  0.13    0.003058          80        38           epoll_ctl
  0.01    0.000327           3        94           write
  0.01    0.000265           3        87           munmap
  0.01    0.000171           1       122           mmap2
  0.01    0.000162           2        97           madvise
  0.01    0.000130           0       270           getuid32
  0.00    0.000117           8        15           fstat64
  0.00    0.000059           0       139           sigprocmask
------ ----------- ----------- --------- --------- ----------------
100.00    2.393540                 19707       471 total
```

Figure 8. Process of extracting system call features from Android application

4.4 Preparing dataset

For each feature set discussed in the preceding section, a feature vector is generated for every application using a Python program specifically developed by the author for this purpose. This program is responsible for producing a file in the Comma-Separated Values (CSV) format, wherein the feature names are arranged as columns and their respective values are recorded as rows. To accomplish this, the program employs a one-dimensional array to store the names of the columns. During execution, the Python program compares the elements of the array with each line of the application's profile. If a feature name is found within the profile, the corresponding value in the array is replaced with 1 in the case of the feature type belonging to (permission, intent, API call). However, if the feature type corresponds to system calls, the value in the array is replaced with its frequency call. Conversely, if a feature name does not match, the value is replaced with 0. Subsequently, the feature vector of the application is appended to the CSV file. This process is repeated for all applications.

4.5 Mutual information

For feature selection, we employed the mutual information feature selection algorithm. The mutual information between feature F and class C can be employed to quantify their level of relevance.

$$MI\,(F,C) = \sum_{f_i}\sum_{c_j} p(F = f_i, C = c_j) * \log\frac{p(F = f_i, C = c_j)}{p(F = f_i\;) * p(C = c_j)} \quad (1)$$

where, P(C = cj) is the frequency count of class C with value cj, P(F = fi) is the frequency count of feature F with value fi, and P(F = fi, C = cj) is the frequency count of F with value fi in class cj. The values of the class are 0 or 1, where 0 indicates a benign sample and 1 indicates a malware sample. And feature values (boolean for permissions, intents, and API; frequency count for system calls). The mutual information is a non-negative value ranging between 0 and 1. A mutual information value of 1 indicates a strong correlation between the feature and the class, while a value of 0 signifies no correlation between the feature and the class.

5. EXPERIMENTAL ENVIRONMENT AND RESULT ANALYSIS

The findings presented in this paper can be categorized into three sub-sections: Firstly, we conducted thorough testing and evaluation of our proposed tool on a selection of Android applications, comparing its performance with well-known tools used for similar purposes. Secondly, we computed the time required for feature extraction using SDHFE. And finally, we employed the random forest algorithm to train and test on ten different datasets in order to identify the most effective feature set for detecting malicious apps in Android devices. The evaluation of the SDHFE tool was performed on a Linux Santoku operating system 64-bit, version 0.5 which was installed within VirtualBox 7.0.8 on a Windows 10 64-bit host system equipped with an Intel(R) Core(TM) i5-2320 CPU @ 3.00 GHz, 4 CPU cores and 4 logical processor, NVIDIA Quadro 4 GB, and 16 GB of RAM. The hardware configuration chosen for the Santoku operating system includes 6 GB of RAM, 1 CPU, and 100 GB of storage. The implementation of our machine learning models was carried out on the Anaconda platform version 1.7.2, utilizing Python version 3. Key libraries utilized in this work include Pandas version 1.0.5, NumPy version 1.18.5, and Scikit-learn version 0.23.1.

5.1 SDHFE tool evaluation and comparison

The APK package is a compressed file containing various components as shown in Figure 1. The static analysis involves two steps: decompiling the APK package and extracting the main files, followed by the conversion of the manifest file to its original XML format and the conversion of classes to smali files, which contain human-readable APIs and opcodes. Static features such as permissions, intents, and APIs are then extracted from these files. To evaluate the performance of the proposed tool, the authors downloaded ten Android apps from the Androzoo database and compared the time required for the first step of static analysis using SDHFE, with two other commonly used tools in reverse engineering, APKTool, and Androguard. Tables 1 and 2 present the hash codes, alternative names of downloaded applications, and the respective time taken by each tool for decompiling apps and converting the extracted files into readable formats.

Table 1. Application hash codes (SHA-256) and alternative names

No	Application Hash Codes	Alternative Name
1	00F437B674B208055122A1465AE385F0BB0B68A7D6C25BE182028FEDD88B3B5A.APK	App1
2	001E621FE5BF38AEC2F29762A3235A45B6C6D2A946EFBE5DB846369D6827475E.APK	App2
3	0013437B05773AFBC48F1B0422A262DD925690A765CE43377CD0C6E8F1A379AF.APK	App3
4	0029566D768782F2B8A713A4C3C3C2FB266A86930D4396DE9E97A6EA76B2BEE1.APK	App4
5	001C5184A1578728C810EC5227EB7B2BC07F488D98886A8835CFD7323A449572.APK	App5
6	00069DD64B3D5D97C63B1ACF3FEA0C5BEA909C8D19A4EA42989ECB0A02125DB3.APK	App6
7	000B39F83E95385C35ED8C36438939D1B7BD2F1AEA67F70B2FCF424D713C3666.APK	App7
8	0023F64B6A69F24E8AF6E9F12835A93E4C23347796B10B9DADDD1A681BFEAA26.APK	App8
9	001E492C1DCFEE402802A0A6D957FA6981FD9A0285694AB8A297795D4045270C.APK	App9
10	00F3FCF02941D4A40FEAF22853FEE8695134058184113BC15DDFF67D5D55FEF9.APK	App10

Table 2. Time required for APK package extraction and decompilation by APKTool, Androguard, and the SDHFE tool

App No.	App Name	App Size	Required Time in Seconds to Analyze and Decompile APK Package by:		
			APK Tool	Androguard	SDHFE
1	App1	1.05 MB	21.96	20.48	6.354
2	App2	12.1 MB	25.96	36.26	10.132
3	App3	22.4 MB	30.67	43.49	10.068
4	App4	33.9 MB	25.15	48.90	9.835
5	App5	41.3 MB	25.78	70.02	13.650
6	App6	53.8 MB	26.98	73.28	14.711
7	App7	60.2 MB	35.45	75.45	16.248
8	App8	70.1 MB	34.94	75.04	19.453
9	App9	82.7 MB	38.31	74.85	15.724
10	App10	91.3 MB	34.71	103.85	19.493

Table 2 presents the analysis results, consisting of 10 apps with a total size of 468.85 MB. The time taken to analyze this dataset using APKTool was 299.91 seconds, while Androguard required 621.22 seconds. In contrast, our proposed tool completed the analysis in just 135.668 seconds. Notably, the SDHFE tool demonstrated a substantial time reduction, approximately 2.2 times faster than APKTool and 4.6 times faster than Androguard. This significant improvement enables researchers to efficiently analyze a large number of apps within a shorter timeframe. Figure 9 provides a visual representation of how long it took APKTool, Androguard, and the SDHFE tools to extract and decompile each APK package.

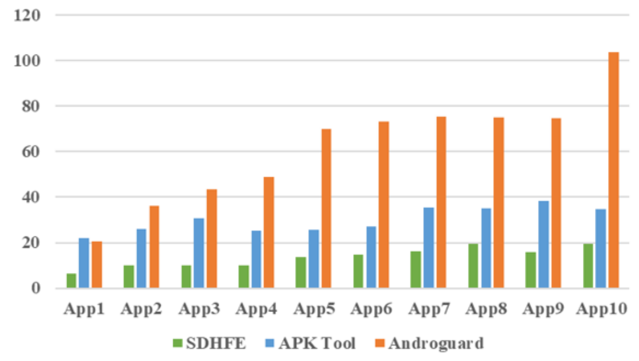


Figure 9. Extraction and decompilation time comparison of APKTool, Androguard, and the SDHFE tool

Another important point is the size of each tool, APKTool has a size of 18.4 MB, Androguard occupies 5.88 MB, while our proposed tool occupies only 0.84 MB. Figure 10 provides a visual representation of the tool sizes, including their corresponding dependencies, in megabytes.

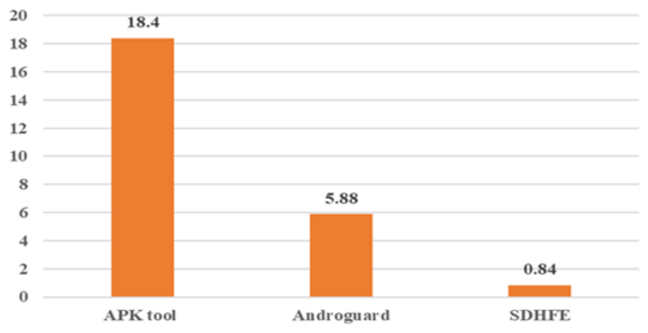
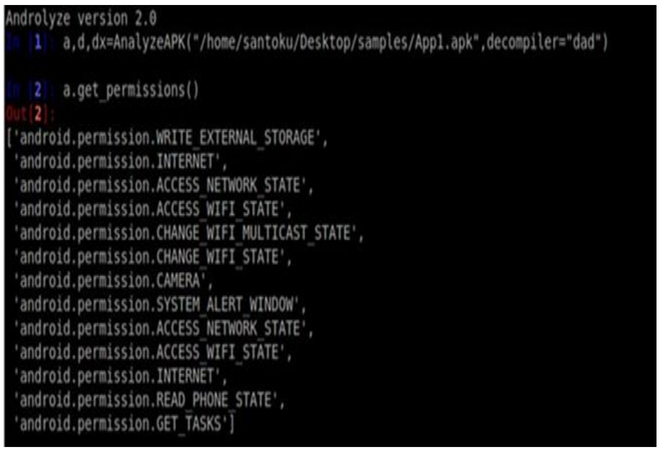


Figure 10. APKTool, Androguard, and SDHFE tool size comparison

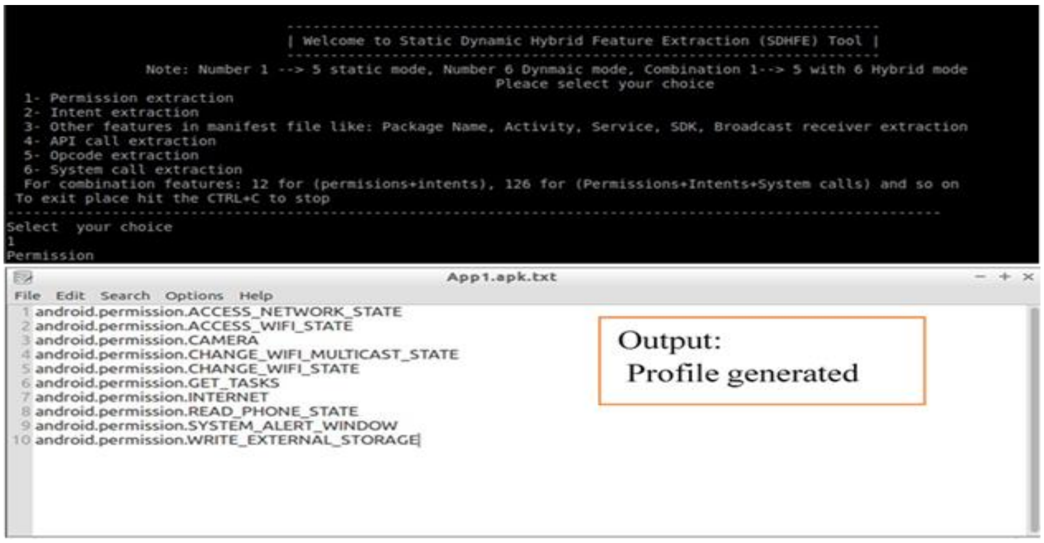
In terms of usability, the SDHFE tool doesn't require programming experience. A researcher simply needs to select the path of applications that wants to analyze and selects the type of features to extract. The APK tool offers robust analysis capabilities. However, it does necessitate programming expertise for feature extraction. It decompiles Android applications, converting their files into readable formats. Subsequently, a researcher needs to write specific programming statements to extract the desired features. In addition to the decompilation of the application, the Androguard tool offers APIs and functions for feature extraction; utilizing these APIs and functions to extract desired features and create the application's profile necessitates familiarity and experience with the tool. Given that the APK tool lacks APIs or functions to extract features after the decompilation process, a comparison between the accuracy of the SDHFE tool and the Androguard tool was conducted for feature extraction. During analyzing the applications listed in Table 1, the author observed that the Androguard tool allows the extraction of repeated features from the manifest file. This requires additional analysis by the researcher to eliminate these redundant features. For example, as depicted in Figure 11, the Androguard tool extracts a total of thirteen permissions from App1. However, it is noteworthy that three permissions are duplicated: 'android.permission.ACCESS_NETWORK_STATE', 'android.permission.ACCESS_WIFI_STATE', and 'android.permission.INTERNET'. In contrast, the SDHFE tool efficiently retrieved ten permissions without duplication from the same app and recorded them alphabetically in a text file as shown in Figure 12. This comparison underscores the higher accuracy of the SDHFE tool over the Androguard tool.



```
Androlyze version 2.0
In [1]: a,d,dx=AnalyzeAPK("/home/santoku/Desktop/samples/App1.apk",decompiler="dad")

In [2]: a.get_permissions()
Out[2]:
['android.permission.WRITE_EXTERNAL_STORAGE',
 'android.permission.INTERNET',
 'android.permission.ACCESS_NETWORK_STATE',
 'android.permission.ACCESS_WIFI_STATE',
 'android.permission.CHANGE_WIFI_MULTICAST_STATE',
 'android.permission.CHANGE_WIFI_STATE',
 'android.permission.CAMERA',
 'android.permission.SYSTEM_ALERT_WINDOW',
 'android.permission.ACCESS_NETWORK_STATE',
 'android.permission.ACCESS_WIFI_STATE',
 'android.permission.INTERNET',
 'android.permission.READ_PHONE_STATE',
 'android.permission.GET_TASKS']
```

Figure 11. Permissions extraction by the Androguard tool



```
                    ------------------------------------------------------------
                    | Welcome to Static Dynamic Hybrid Feature Extraction (SDHFE) Tool |
                    ------------------------------------------------------------
          Note: Number 1 --> 5 static mode, Number 6 Dynmaic mode, Combination 1--> 5 with 6 Hybrid mode
                                             Please select your choice
 1- Permission extraction
 2- Intent extraction
 3- Other features in manifest file like: Package Name, Activity, Service, SDK, Broadcast receiver extraction
 4- API call extraction
 5- Opcode extraction
 6- System call extraction
 For combination features: 12 for (permisions+intents), 126 for (Permissions+Intents+System calls) and so on
To exit place hit the CTRL+C to stop
--------------------------------------------------------------------------------------------
Select  your choice
1
Permission
```

```
App1.apk.txt
File  Edit  Search  Options  Help
android.permission.ACCESS_NETWORK_STATE
android.permission.ACCESS_WIFI_STATE
android.permission.CAMERA
android.permission.CHANGE_WIFI_MULTICAST_STATE
android.permission.CHANGE_WIFI_STATE
android.permission.GET_TASKS
android.permission.INTERNET
android.permission.READ_PHONE_STATE
android.permission.SYSTEM_ALERT_WINDOW
android.permission.WRITE_EXTERNAL_STORAGE
```

Output:
Profile generated

Figure 12. Permissions extraction by the SDHFE tool

Table 3. Duration of static feature extraction (in seconds)

App No.	App Name	Permissions	Intents	API Calls	Permissions + Intent	Permissions + API Calls	Intent + API Calls
1	App1	0.61	0.574	6.632	0.647	6.932	6.805
2	App2	0.777	0.752	10.674	0.821	15.322	14.989
3	App3	1.212	1.231	11.799	1.257	14.068	13.682
4	App4	1.198	1.198	11.249	1.199	12.884	12.132
5	App5	3.766	3.793	15.302	3.798	17.481	17.273
6	App6	5.063	5.121	17.641	5.170	20.967	20.087
7	App7	6.302	6.236	19.066	6.316	22.522	22.653
8	App8	8.535	8.356	20.656	8.719	28.369	28.088
9	App9	4.314	3.591	19.465	4.394	22.194	21.590
10	App10	8.205	8.295	21.303	8.301	29.865	29.545

5.2 The run-time overhead of feature extraction

The process of static analysis fundamentally differs from dynamic analysis. The applications listed in Table 1 are utilized for the analysis and extraction of static and dynamic features using the SDHFE tool in two different cases.

5.2.1 Case one static feature

The process of extracting static features involves two steps. Firstly, the application is decompiled, enabling access to its internal structure. Subsequently, a targeted search is conducted within a specific file(s) to extract the desired features. Permissions and intents are extracted from the manifest file, whereas the API features are extracted from the smali files. Table 3 presents the time required in seconds to extract features from the manifest and smali files for ten applications of varying sizes. The obtained data in Table 3 are plotted to get its related graph, as shown in Figure 13.

During the analysis process, it is sometimes observed that smaller APK packages may require a longer duration compared to larger APK packages, despite the disparity in their file sizes. The reason behind this is that other factors affect the delay in the analysis process, such as structural intricacies and complexity of the underlying code, the optimization techniques employed to compress certain files within the APK package, and the sources that the analysis tool focuses on.

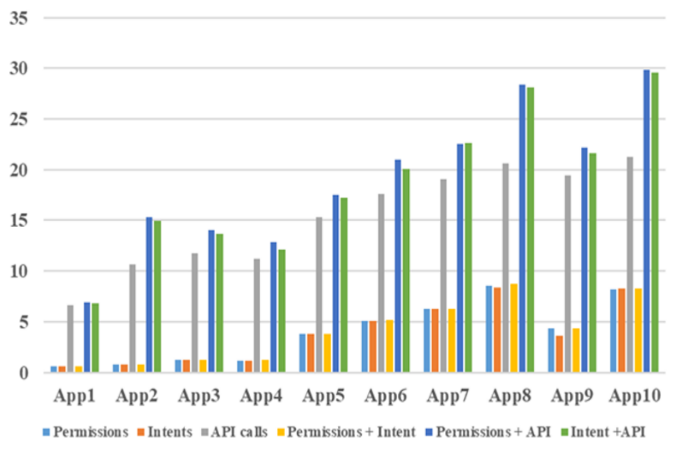


Figure 13. Time required for extraction features from manifest and smali files for ten Android applications

5.2.2 Case two dynamic features

The time required to extract system call features from an application depends on the duration of the application's execution by the user. In this study, all applications were executed in the Genymotion emulator for an equal period (60 seconds) and subjected to 500 pseudo-random events. The author utilized the Monkey tool to generate pseudo-random events to simulate user interactions or actions that mimic real user behavior when interacting with an Android application running on Genymotion. These events are generated to test the application's functionality. Some examples of pseudo-random events generated by Monkey include touch events, keystrokes, random navigation between activities and menus, scrolling, sending SMS, and a number of system-level events such as changing the settings of the system. Generally, dynamic features demand more time compared to static features due to the additional steps involved, such as the installation and uninstallation of applications within the emulator, alongside the actual execution time.

5.3 Contribution of the feature set to detect malware

This study focuses on single-feature categories and possible combinations of the two-feature categories while excluding the consideration of a multi-feature category. The higher computational requirements of the multi-feature category, which could potentially affect the model's performance when deployed and tested in real-time on smartphones.

Table 4. Description of datasets

Dataset	Source	Feature Type	No. of Samples	No. of Features
DS1	Manifest file	Permission	500 Benign 500 Malware	146
DS2	Manifest file	Intent	500 Benign 500 Malware	114
DS3	Smali files	API calls	500 Benign 500 Malware	246
DS4	Behavior execution	System calls	500 Benign 500 Malware	80
DS5	Manifest file	Permissions and Intents	500 Benign 500 Malware	260
DS6	Manifest file + smali files	Permissions and APIs	500 Benign 500 Malware	392
DS7	Manifest file + smali files	APIs and Intents	500 Benign 500 Malware	360
DS8	Manifest file + Behavior execution	Permissions and system class	500 Benign 500 Malware	226
DS9	smali files + Behavior execution	APIs and system calls	500 Benign 500 Malware	326
DS10	Manifest file + Behavior execution	Intent and system calls	500 Benign 500 Malware	194

Table 5. Results of 10 random forest models

Dataset	Random Forest Models	Accuracy (%)		Precision (%)	Recall (%)	F1 Score (%)	Average Test Scores (%)
		Train	Test	Test	Test	Test	Test
DS1	Model 1	98.26	98	96.87	99.2	98.02	98.02
DS2	Model 2	90.66	91.6	95.61	87.2	91.21	91.41
DS3	Model 3	99.46	95.6	93.84	97.6	95.68	95.68
DS4	Model 4	100	96.8	98.34	95.19	96.74	96.77
DS5	Model 5	98.26	99.2	100	98.4	99.19	99.2
DS6	Model 6	100	98.8	99.19	98.4	98.79	98.8
DS7	Model 7	95.06	90.8	96.36	84.8	90.21	90.54
DS8	Model 8	100	94	100	88	93.61	93.9
DS9	Model 9	100	91.6	96.42	86.4	91.13	91.39
DS10	Model 10	100	97.2	97.58	96.8	97.18	97.19

As illustrated in Figure 2, our study comprises ten distinct datasets (DS), denoted as DS1, DS2, DS3, DS4, DS5, DS6, DS7, DS8, DS9, and DS10. The details of each dataset are shown in Table 4. For each dataset, a split of 75% was allocated for training purposes, while the remaining 25% was reserved for testing. To facilitate the selection of meaningful features, we employed the mutual information algorithm, which identified 75 significant features from each dataset. These selected features were subsequently utilized for training and testing the random forest algorithm separately. Various evaluation metrics were employed to discern the optimal model for the detection of Android malware. The results obtained by the random forest algorithm on all datasets are presented in Table 5. The average scores of each model in Table 5 are plotted to generate related graphs as shown in Figure 14.

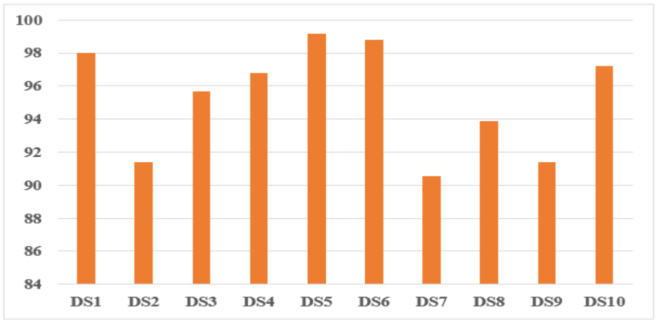


Figure 14. Average scores of 10 random forest models

Illustrated in Figure 14, the model trained on DS5, which is a combination of permissions and intents, displayed remarkably elevated average scores. In contrast, the model trained on DS7, comprised of APIs and Intents, showcased the least favorable outcomes. Furthermore, Figure 15 illustrates the Receiver Operating Characteristic (ROC) curve, derived from an ensemble of ten random forest models, presenting a comprehensive visualization of the model's performance across different thresholds.

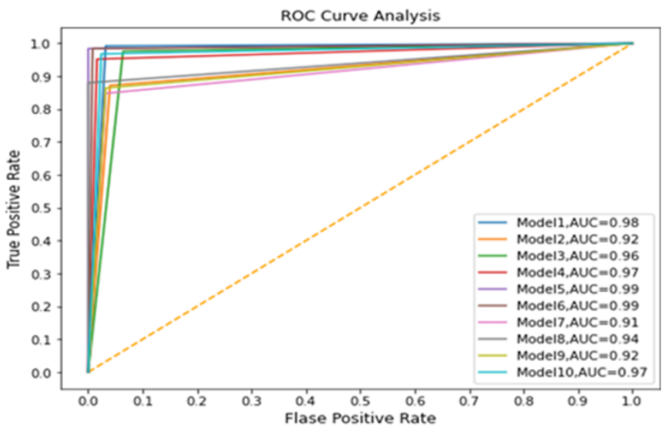


Figure 15. ROC curve of 10 random forest models

6. DISCUSSION

The findings of this study can be categorized into two parts. The initial part primarily centers on examining the time and complexity necessary to extract each specific type of feature. The process of extracting features from the Manifest file generally entails parsing the XML structure and retrieving significant information. This procedure is typically lightweight and computationally efficient due to the Manifest file's small size and uncomplicated nature. While extracting features from source code requires converting the class.dex file into smali files. This process demands more time due to the following reasons:

1. Smali files contain low-level bytecode representations of the app's source code. Decompiling and analyzing bytecode is inherently more complex and resource-intensive than parsing structured XML data.
2. Extracting features from multiple smali files requires additional processing time and computational resources.

Figures 5 and 7 also confirm that the process of extracting features from the source code is more complicated than from the manifest file. The obtained results in Table 3 as well support the fact that the permission and intent features from the manifest file need less time than API calls from the source code. The computational effort required for analyzing system calls can vary based on factors such as app execution duration, the number of system calls made, and the complexity of the interactions. Profiling dynamic features (system calls) can potentially be more computationally intensive than compared to profiling static features due to the reasons mentioned in Section 4.3.3.

The second part of the study focuses on machine learning models. Actually, the author evaluate each model by many metrics including accuracy, precision, recall, and F1 score. The accuracy provides an overall measure of correct predictions across all classes. The recall emphasizes the ability to capture positive instances; this metric is useful when missing positive cases has serious consequences, minimizing false negatives. Precision focuses on the accuracy of positive predictions, minimizing false positives. F1 scores strike a balance between precision and recall. In this study, the positive class represents malware samples, and the negative class represents benign samples. The accuracy of the overall model is very important. However, it needs to check the value of other metrics, especially recall, which focuses on the rate of detecting malware samples. During experiments, certain models demonstrated 100% accuracy during the training phase. Nevertheless, their performance in the testing phase was less reliable, particularly in detecting malware samples. For example, models 8 and 9 exhibited lower recall scores in the testing phase, which means these models produce a lower true positive rate. Model 7, which was trained using APIs and intents features, attained a notably lower average score. Similarly, models 2, 8 and 9, trained on intents and the combination of (permissions and system calls) and (API calls and system calls) respectively, also exhibited comparatively lower performance. This is an indicator of a weak relationship between features. Some models got acceptable average test scores, such as models 1, 6, and 10. However, based on the results presented in Table 5, it is evident that model 5, which combines permissions and intents, achieved notably higher average scores during the testing phase.

Another metric called AUC-ROC curve was utilized to visually represent the performance of all models on a single curve. Figure 15 demonstrates that models 5 and 6 achieved higher AUC scores, which are 99%, indicating superior performance. However, considering the results in Table 3, the feature extraction process for model 5 requires less time compared to model 6. Actually, permissions and intents are closely related. An app might require specific permissions to perform certain actions triggered by intents. For example, the Banking Trojan application is appeared as a legitimate banking application for users but is designed to steal sensitive user information, such as login credentials and financial data. This example will illustrate how the permissions with intents feature sets might be strongly related. The malicious app requests permission to access SMS (android.permissions.RECEIVE_SMS) and contacts (android.permissions.READ_CONTACTS). The app registers an intent filter to intercept incoming SMS messages containing keywords related to banking transactions or authentication codes. It uses intercepted SMS data to extract sensitive information and send it to a remote server.

Overall, model 5, which combines permissions and intents, is the optimal feature set based on the majority of criteria employed in this study for real-time malware detection on smartphones. However, like any other machine learning model, this model can have weaknesses. The possible weakness of this model appears when both malware and benign apps utilize similar permissions and intents. This might increase the risk of false positives and false negatives, flagging benign apps as malware and flagging malware as benign.

7. CONCLUSION

In this study, we conducted a comprehensive analysis of 1000 Android applications, comprising both malicious and benign samples, utilizing the SDHFE tool. This tool enabled the extraction of features from diverse sources, including manifest files, smali files, and runtime behavior within an isolated environment. Subsequently, we generated multiple datasets (DS1 to DS10) encompassing various feature categories, both individually and in combinations.By harnessing the power of the random forest algorithm, we developed numerous machine-learning models to discern the optimal feature set for detecting malicious apps on Android smartphones. Through meticulous evaluation using accuracy, precision, recall, and F1 score metrics, we consistently achieved impressive average scores surpassing the 90% score. Notably, the DS5 dataset exhibited the highest average scores, while DS7 demonstrated relatively lower scores.

An important observation emerged from our study: the DS5 dataset, enriched with manifest-based features, showcased reduced computational demands, rendering it particularly suitable for resource-constrained devices like smartphones. This discovery underscores the potential advantages of leveraging manifest-based features for efficient malware detection in such environments.

Moving forward, this research paves the way for several avenues of exploration. To advance the field of Android malware detection, we recommend an in-depth investigation into the integration of hybrid feature sets, combining static and dynamic attributes. Furthermore, the exploration of ensemble learning techniques and the integration of more advanced malware behavior analysis mechanisms could contribute to even higher accuracy levels. Overall, this study not only sheds light on effective feature selection for malware detection but also opens doors for innovative enhancements to address emerging challenges in the ever-evolving landscape of mobile security.

REFERENCE

[1] Qamar, A., Karim, A., Chang, V. (2019). Mobile malware attacks: Review, taxonomy & future directions. Future Generation Computer Systems, 97: 887-909. https://doi.org/10.1016/j.future.2019.03.007

[2] Qiu, J., Zhang, J., Lou, W., Nepal, S., Wang, Y., Xiang, Y. (2019). A3CM: Automatic capability annotation for android malware. IEEE Access, 7: 147156-147168. https://doi.org/10.1109/ACCESS.2019.2946392

[3] Choudhary, M., Kishore, B. (2018). Haamd: Hybrid analysis for android malware detection. In 2018 International Conference on Computer Communication and Informatics (ICCCI), Coimbatore, India, pp. 1-4. https://doi.org/10.1109/ICCCI.2018.8441295

[4] Ding, Y., Zhang, A., Hu, J., Xu, W. (2020). Android malware detection method based on bytecode image. Journal of Ambient Intelligence and Humanized Computing, 1-10. https://doi.org/10.1007/s12652-020-02196-4

[5] Salih, H.M., Mohammed, M.S. (2020). Spyware injection in android using fake application. 2020 International Conference on Computer Science and Software Engineering (CSASE), Duhok, Kurdistan Region, Iraq. https://doi.org/10.1109/CSASE48920.2020.9142101

[6] Wang, H., Si, J., Li, H., Guo, Y. (2019). Rmvdroid: Towards a reliable android malware dataset with app metadata. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada. https://doi.org/10.1109/MSR.2019.00067

[7] Almomani, I., Qaddoura, R., Habib, M., Alsoghyer, S., Khayer, A.A., Aljarah, I., Faris, H. (2021). Android ransomware detection based on a hybrid evolutionary approach in the context of highly imbalanced data. IEEE Access, 9: 57674-57691. https://doi.org/10.1109/ACCESS.2021.3071450

[8] Khan, K.N., Ullah, N., Ali, S., Khan, M.S., Nauman, M., Ghani, A. (2022). OP2VEC: An opcode embedding technique and dataset design for end-to-end detection of android malware. Security and Communication Networks. https://doi.org/10.1155/2022/3710968

[9] Wang, X., Zhang, L., Zhao, K., Ding, X., Yu, M. (2022). MFDroid: A stacking ensemble learning framework for Android malware detection. Sensors, 22(7): 2597. https://doi.org/10.3390/s22072597

[10] Kapoor, A., Kushwaha, H., Gandotra, E. (2019). Permission based android malicious application detection using machine learning. In 2019 International Conference on Signal Processing and Communication (ICSC), Noida, India. https://doi.org/10.1109/ICSC45622.2019.8938236

[11] Bibi, I., Akhunzada, A., Malik, J., Iqbal, J., Musaddiq, A., Kim, S. (2020). A dynamic DL-driven architecture to combat sophisticated Android malware. IEEE Access, 8: 129600-129612. https://doi.org/10.1109/ACCESS.2020.3009819

[12] Sirisha, P., Kamala, P.B., Aditya, K.K., Anuradha, T. (2019). Detection of permission driven malware in android using deep learning techniques. In 2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India. https://doi.org/10.1109/ICECA.2019.8821811

[13] Hr, S. (2019). Static analysis of android malware detection using deep learning. In 2019 International Conference on Intelligent Computing and Control Systems (ICCS), Madurai, India. https://doi.org/10.1109/ICCS45141.2019.9065765

[14] Koli, J.D. (2018). RanDroid: Android malware detection using random machine learning classifiers. In 2018 Technologies for Smart-City Energy Security and Power (ICSESP), Bhubaneswar, India, pp. 1-6. https://doi.org/10.1109/ICSESP.2018.8376705

[15] Alsoghyer, S., Almomani, I. (2020). On the effectiveness of application permissions for Android ransomware detection. In 2020 6th conference on data science and machine learning applications (CDMA), Riyadh, Saudi Arabia, pp. 94-99. https://doi.org/10.1109/CDMA47397.2020.00022

[16] Zhao, L., Li, D., Zheng, G., Shi, W. (2018). Deep neural network based on android mobile malware detection system using opcode sequences. In 2018 IEEE 18th International Conference on Communication Technology (ICCT), Chongqing, China, pp. 1141-1147. https://doi.org/10.1109/ICCT.2018.8600052

[17] Ma, Z., Ge, H., Liu, Y., Zhao, M. Ma, J. (2019). A combination method for android malware detection based on control flow graphs and machine learning algorithms. IEEE Access, 7: 21235-21245. https://doi.org/10.1109/ACCESS.2019.2896003

[18] Kumar, U.S., Yadav, A., Singh, V. (2022). Detecting malware in android applications by using Androguard tool and XGBoost Algorithm. In 2022 IEEE 9th Uttar Pradesh Section International Conference on Electrical, Electronics and Computer Engineering (UPCON), Prayagraj, India, pp. 1-6. https://doi.org/10.1109/UPCON56432.2022.9986470

[19] Feng, R., Chen, S., Xie, X., Ma, L., Meng, G., Liu, Y., Lin, S.W. (2019). Mobidroid: A performance-sensitive malware detection system on mobile platform. In 2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS), Guangzhou, China, pp. 61-70. https://doi.org/10.1109/ICECCS.2019.00014

[20] Feng, P., Ma, J., Sun, C., Xu, X., Ma, Y. (2018). A novel dynamic Android malware detection system with ensemble learning. IEEE Access, 6: 30996-31011. https://doi.org/10.1109/ACCESS.2018.2844349

[21] Esmaeili, S., Shahriari, H.R. (2019). PodBot: a new botnet detection method by host and network-based analysis. In 2019 27th Iranian Conference on Electrical Engineering (ICEE), Yazd, Iran, pp. 1900-1904. https://doi.org/10.1109/IranianCEE.2019.8786432

[22] Lê, N.C., Nguyen, T.M., Truong, T., Nguyen, N.D., Ngô, T. (2020). A Machine learning approach for real time Android malware detection. In 2020 RIVF International Conference on Computing and Communication Technologies (RIVF), Ho Chi Minh City, Vietnam, pp. 1-6. https://doi.org/10.1109/RIVF48685.2020.9140771

[23] Kumar, S., Ari., V., Timo, H. (2018). A network-based framework for mobile threat detection. In 2018 1st International Conference on Data Intelligence and Security (ICDIS), South Padre Island, TX, USA, pp. 227-233. https://doi.org/10.1109/ICDIS.2018.00044

[24] Manzil, H.H.R., Naik, S.M. (2023). Android malware category detection using a novel feature vector-based machine learning model. Cybersecurity, 6(1): 6. https://doi.org/10.1186/s42400-023-00139-y

[25] Mahdavifar, S., Kadir, A.F.A., Fatemi, R., Alhadidi, D., Ghorbani, A.A. (2020). Dynamic android malware category classification using semi-supervised deep learning. In 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), Calgary, AB, Canada, pp. 515-522. https://doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00094

[26] Arora, A., Peddoju, S. (2018). NTPDroid: A hybrid Android malware detector using network traffic and system permissions. In 2018 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE). New York, NY, USA. https://doi.org/10.1109/TrustCom/BigDataSE.2018.00115

[27] Garg, S., Baliyan, N. (2019). A novel parallel classifier scheme for vulnerability detection in android. Computers & Electrical Engineering, 77: 12-26. https://doi.org/10.1016/j.compeleceng.2019.04.019

[28] Somasundaram, S., Kasthurirathna, D., Rupasinghe, L. (2019). Mobile-based malware detection and classification using ensemble artificial intelligence. In 2019 International Conference on Advancements in Computing (ICAC), Malabe, Sri Lanka, pp. 351-356. https://doi.org/10.1109/ICAC49085.2019.9103424

[29] Wang, Z., Liu, Q., Chi, Y. (2020). Review of android malware detection based on deep learning. IEEE Access, 8: 181102-181126. https://doi.org/10.1109/ACCESS.2020.3028370

[30] Wu, Q., Li, M., Zhu, X., Liu, B. (2020). Mviidroid: A multiple view information integration approach for android malware detection and family identification. IEEE MultiMedia, 27(4): 48-57. https://doi.org/10.1109/MMUL.2020.3022702

[31] Chen, L., Xia, C., Lei, S., Wang, T. (2021). Detection, traceability, and propagation of mobile malware threats. IEEE Access, 9: 14576-14598. https://doi.org/10.1109/ACCESS.2021.3049819

[32] Lei, T., Qin, Z., Wang, Z., Li, Q., Ye, D. (2019). EveDroid: Event-aware android malware detection against model degrading for IoT devices. IEEE Internet of Things Journal, 6(4): 6668-6680. https://doi.org/10.1109/JIOT.2019.2909745

[33] Agrawal, P., Trivedi, B. (2019). A survey on android malware and their detection techniques. In 2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT), Coimbatore, India, pp. 1-6. https://doi.org/10.1109/ICECCT.2019.8868951