Hindawi
Wireless Communications and Mobile Computing
Volume 2022, Article ID 1830201, 12 pages
https://doi.org/10.1155/2022/1830201

Research Article

An Android Malware Detection Leveraging Machine Learning

Ahmed S. Shatnawi,[1] **Aya Jaradat**,[2] **Tuqa Bani Yaseen**,[2] **Eyad Taqieddin**,[2] **Mahmoud Al-Ayyoub**,[3] **and Dheya Mustafa**[4]

[1]*Department of Software Engineering, Jordan University of Science & Technology, Irbid 21110, Jordan*
[2]*Department of Network Engineering and Security, Jordan University of Science & Technology, Irbid 21110, Jordan*
[3]*Department of Computer Science, Jordan University of Science & Technology, Irbid 21110, Jordan*
[4]*Department of Computer Engineering, Faculty of Engineering, The Hashemite University, Zarqa 13133, Jordan*

Correspondence should be addressed to Ahmed S. Shatnawi; ahmedshatnawi@just.edu.jo

Received 4 December 2021; Revised 23 March 2022; Accepted 8 April 2022; Published 6 May 2022

Academic Editor: Mohamed Elhoseny

Android applications have recently witnessed a pronounced progress, making them among the fastest growing technological fields to thrive and advance. However, such level of growth does not evolve without some cost. This particularly involves increased security threats that the underlying applications and their users usually fall prey to. As malware becomes increasingly more capable of penetrating these applications and exploiting them in suspicious actions, the need for active research endeavors to counter these malicious programs becomes imminent. Some of the studies are based on dynamic analysis, and others are based on static analysis, while some are completely dependent on both. In this paper, we studied static, dynamic, and hybrid analyses to identify malicious applications. We leverage machine learning classifiers to detect malware activities as we explain the effectiveness of these classifiers in the classification process. Our results prove the efficiency of permissions and the action repetition feature set and their influential roles in detecting malware in Android applications. Our results show empirically very close accuracy results when using static, dynamic, and hybrid analyses. Thus, we use static analyses due to their lower cost compared to dynamic and hybrid analyses. In other words, we found the best results in terms of accuracy and cost (the trade-off) make us select static analysis over other techniques.

1. Introduction

Cybersecurity has become a primary area of immediate concern to computer scientists and network engineers that satisfactory solutions to many issues are in order. As a result of the rapid evolutions in technology developments and their inherent integrations in all aspects of our lifestyles, a variety of malware applications and their intended targets have become well studied and identified. Among the malware variety that has received attention lately is Android malware which is found to occupy considerable attention in the web world. Android is one of the most common operating systems that dominates the operating system market. In 2020 [1], the Android system accounted for 85% of the total number of smartphones that harness the Android system as the operating system of choice.

In general, an Android system inherently supports a whole host of applications. At the end of April 2020, the number of these applications exceeded the 2.8 million mark [2]. The variety of these applications are commonly found on Google Store, which is home to most Android applications. Moreover, the increasing reliance on this system by application developers and users together with its friendly nature has made it more prone for malware to sneak into these applications [3]. This is by nature of the fact malicious programs execute their functionalities in a seamless manner and are assisted by several key factors. These include, amongst others, calling a third-party code, the environment under which an Android application operates [4], in addition to the level of permissions allowed.

Malware invasive techniques continue to evolve with the aim of evading detection [5], as some malware applications contain more than 50 variables that make detecting them a great challenge. Therefore, it is necessary to work on finding methods capable of dealing with the continuous evolution of Android malware to detect it, where it happens, and deactivate/remove it effectively [6]. All such challenges did indeed preoccupy researchers in the area and prompted them to do research to discover malware and tackle it in a proper manner [7]. As a result, researchers were able to develop three mechanisms to detect Android malware that fell under static, dynamic, and hybrid analyses techniques. Static analysis is harnessed with the objective of extracting the features that help us identify harmful behavior for an application without a need for actual application deployment. However, this type of analysis was found to suffer from code obfuscation techniques that help malware authors to evade static detection techniques [8]. Dynamic analysis, however, is leveraged to determine the malware behavior of an application during its runtime such as during system calls. Generally, the static analysis feature provides the ability to locate the malware component through the source code, as the dynamic analysis feature provides the ability to identify the malware location through in a runtime environment [4].

In this paper endeavor, we propose a model for Android malware detection based on a combination of static and two dynamic analyses (hybrid analysis) together with machine learning classifiers. In the hybrid analysis part, we set out by conducting experiments through which we would extract those features that can provide the maximum possible amount of important information about the application's behavior by rank selection. After that, we apply different machine learning (ML) algorithms and compare the performance of each to identify the most accurate algorithm. In this, ML is one of the modern technologies currently widely used in many areas, including malware detection, as it is used in the classification process and the selection of features.

The process of selecting and extracting features is one of the most critical steps in an Android malware detection process. The efficiency in feature extraction readily determines the detection quality. Hence, extracting and selecting certain features from the hotspots must be given particular attention in the malware detection process [9–11]. Nonetheless, there are some concerning obstacles that could be encountered in the feature selection process:

(i) The feature extraction process may consume considerable time due to the large size of the Android data file; as the number of features may be in millions, the entire process of extracting features could pose some serious burdens. Furthermore, it is also possible for the malware detection process to become ineffective depending on the choice of some of the features that may have a low impact on the accurate Android malware detection

(ii) As a result of the rapid development and evolution of Android applications, the process of extracting and selecting features is observing continuous improvement due to a large reservoir of different behaviors of the applications and the various forms that have emerged; it is quite possible to reach a feature that had not been met/identified before which ultimately could interject significant bearing towards a more accurate detection process

(iii) It is often that a feature could get selected undercutting the algorithm's ability to skip unnecessary program execution paths, rendering the detection accuracy ineffective for all potential cases. Therefore, one aim of this study would be so to choose the features accurately towards the "best" detection accuracy

Therefore, the main contribution of our paper is to detect malware based on a hybrid analysis scenario, whereby we combine several dynamic and static features together. Here, the permissions represent the static feature that is extracted from the manifest file, which can either be the port and/or the IP address, whereas the action repetition represents the dynamic features that were extracted during the implementation of the application. The action repetition feature is one of the features that is being explored for the first time, including its use with the permissions. In the work presented here, we used four machine learning classifiers with particular abilities to classify correctly when selecting the features appropriately. The results of our evaluation have shown that our system is fairly competitive in terms of the accuracy attained in the malware detection process.

In short, the contribution in this research work fulfills several objectives:

(i) Introduce a new feature, which was never addressed in the literature. This is particularly manifested in the action repetition, whereby actions are monitored and tracked, with the repetition number leading to the detection of a harmful behavior

(ii) Contrive a new, practical and highly effective system capable of detecting malware, leveraging a set of features, the most important of which are action repetition and permissions. The objective, here, entails arriving at experimental results that reflect high accuracy and sufficient efficacy, in an effort to demonstrate the effectiveness of our proposed system

The rest of this paper is organized as follows: Sections 2 and 3 shed some light on related background work. Section 4 describes the methodology proposed in this paper. Sections 5 and 6 discuss the experimental setup used and evaluation of results, respectively. Section 7 arrives at key conclusive remarks of the work addressed in this paper.

2. Background

Malware is a term used to refer to software with inherent malicious objectives. They present illegal activity targeted by attackers towards theft of data or user credentials [12].

Applications form an attack vector through which malware can be delivered on mobile devices. Most users have limited expertise to identify the permissions required by an application nor the exploits that may appear through them. Furthermore, installing the process of selecting and extracting features is one of the most critical steps in an Android malware detection process.

Many researchers use traditional malware analysis techniques wherein they are restricted between two practical approaches to detect such malware. The Android malware detection analysis-based approaches are static, dynamic, and hybrid. The following subsections introduce these analysis methods, briefly summarizing their employed features. Static-based malware binary classification static analysis involves unpacking the application to analyze the code for any malicious content. By decompiling the files, it is possible to identify critical parts of the code [13]. This may also involve extracting features from the disassembly of the underlying Java code and the AndroidManifest.xml file [13].

This type of analysis does not involve any execution of the application. Accordingly, it does not require large resources and is considered faster in comparison to other approaches [14]. However, with the explosive growth in the number of Android applications, it has become rather challenging to rely solely on such an approach as some malicious applications use repackaging or obfuscation to bypass detection. Meanwhile, dynamic-based malware classification analysis leverages activities associated with an application as the application is being executed. It may involve some interaction from the user or other processes in order to trigger the malicious behavior [15]. Here, the actions of the application are monitored to detect abnormalities in the system call, network activity, processor load, phone calls initiated, or SMS sent in order to extract the dynamic features. Dynamic analysis has an advantage of recording the applications' behavior and detecting any dynamic code being loaded at runtime, which makes it favorably popular for malicious code analysis [16]. However, a major drawback of this form of analysis is the added overhead on the operating system, which makes it harder to implement. Furthermore, it takes a long duration to complete due to the fact that monitoring the activities may not yield useful results without proper triggers.

Hybrid analysis malware classification is a type of analysis that mix of both the static and dynamic analyses to benefit from their combined advantages. It involves analyzing the source files of the application along with monitoring the behavior at runtime. Here, although it achieves higher detection accuracy, it has been found to suffer from the drawbacks of both previous approaches in terms of long duration for detection and consumption of the operating system resources [17].

3. Related Work

Several efforts related to Android malware detection have been addressed in the literature. In the sequel in this section, we present a rundown of the most relevant papers with respect to static, dynamic, and hybrid malware analysis.

The work presented in [18] introduced a static-based malware detection method. There, the authors consider the API call feature by extracting it using correlative analysis. Furthermore, three machine learning algorithms, support vector machine (SVM), random forest (RF), and k-nearest neighbor (KNN), were used to train the feature in order to classify the application as either benign or malicious. The RF classifier produced the best results in terms of accuracy. In [19], Ratyal et al. introduced a model in which they use the features of Android permissions and Android destinations to detect malware. According to the authors, these two features resulted in improved detection. Another approach for static analysis was introduced in [20]. The features were determined based on three steps: graph creation, sensitive node extraction, and method creation. The authors employed four classifiers, of which RF was the best in terms of classification accuracy.

Static analysis was also addressed in [21] by checking open sockets, message digest, and reflection code. The paper develops a process of identifying malware which resulted in 96% accuracy leveraging RF machine learning methodology.

Cai [22] studied the sustainability problem for ML-based app classifiers. He defined sustainability metrics and compared them among five state-of-the-art malware detectors for Android. He also built an infrastructure to mine a mobile software ecosystem with three elements; the mobile platforms, user apps built on the platforms, and users associated with the apps [23]. Fu and Cai investigated the deterioration problem of defense solutions against malware using machine learning classification for four state-of-the-art Android malware detectors [24].

The work in [25] proposes an estimation method that is based on an entropy approach in choosing the best features, thereby averting the need for a standard procedure that chooses all the features to differentiate between benign and harmful malware, particularly that APK files possess many features, making any standard process rather cumbersome and infeasible. The approach in [25] was shown to achieve an accuracy of 96.9% in KNN and SVM machine learning.

Meanwhile, several works in the literature have addressed different approaches of classification harnessing dynamic analysis. In [26], three input generation techniques were used: state-based, random-based, and hybrid techniques. There, the authors use the DroidBot and Monkey tools to extract features from the generated log files. Their results indicate that the best classification results were achieved using RF. The works in [27, 28] also use dynamic analysis to create the permissions then use WEKA to collect the permissions. The main difference between the two papers lies in the number of classifiers used. The former uses five classifiers, with the best results reported were achieved when using the simple logistic (SL), J48, and RF techniques. On the other hand, the work in [28] uses seven classifiers and shows the best classification outcomes when employing RF techniques.

Authors in [29] extract the dynamic analysis features by using the structure, safety, and intercomponent communication dimensions. They found that the features that were extracted using the dimensions of the structure were more important than the other two.

Cai and Jenkins targeted a sustainable Android malware detector that, once trained on a dataset, can continue to effectively detect new malware without retraining [30]. DroidEvolver [31] is an Android malware detection system that can automatically and continually update itself during malware detection without any human involvement.

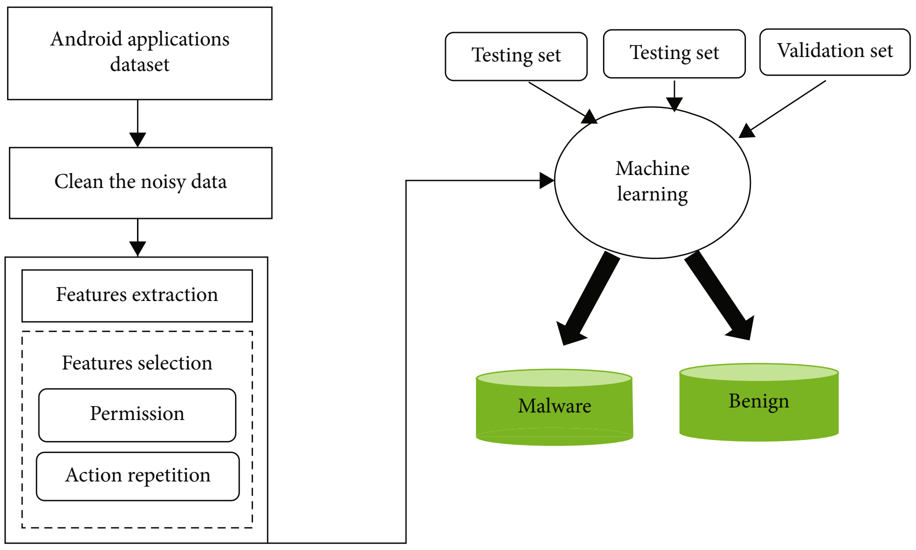

Figure 1: Proposed hybrid analysis approach.

In [16], the authors introduce a malware detection mechanism in which they rely on hybrid analysis features, namely, API calls, permissions, and system calls. They use the tree augmented naive Bayes to capture the interrelation between static and dynamic features. The mechanism used in [29] was shown to achieve a detection accuracy of 97.

The SAMADroid and StormDroid techniques presented in [32, 33],, respectively, offer hybrid analysis tools. The SAMADroid approach combines static and dynamic analysis on both local and remote hosts. The tool uses SVM, RF, and naïve Bayes (NB) and achieves high accuracy in detecting malicious programs. It further adds the advantage of low energy consumption and improved storage efficiency. The StormDroid approach, on the other hand, extracts static and dynamic features from permissions and sensitive API calls by sequencing them directly in the source code.

Another more recent hybrid approach is addressed in [16]; the malware detection model presented is based on a tree augmented naive Bayes (TAN), which uses conditional dependencies against both static and dynamic features such as API permissions and system calls. It then detects malicious behavior by combining the outputs of both of these features as gained from the classifier. Although the model shows 97% accuracy, it does not show the android version or user input generated during dynamic analysis.

In [34], a different hybrid detection system is proposed. The model uses SVM and a linear classifier based on a new open source framework called CuckooDroid. The framework enables Cuckoo Sandbox's features by a misuse detector—"an approach in which attack patterns or unauthorized and suspicious behaviors are learned based on past activities and then the knowledge about the learned patterns is used to detect or predict similar subsequent such patterns in a network.". Misuse detection is commonly adopted to detect well-known malware and classify android malware by combining static and dynamic analysis techniques together. The proposed model benefits from the low false-positive rate of misuse detection and the ability of anomaly detection to detect zero-day malware and is evaluated using 5560 malware and 12000 benign samples. This model was shown to achieve 98.79% accuracy detection rates when classifying 98.32% of the malware samples.

In [35], the authors extract the API and permissions from the source code to access the static analysis features. They also consider the total time needed for the system to extract the dynamic analysis features. The detection accuracy reached based on this came out to be 93.33%. Along the same context, authors in [17] apply the static, dynamic, and hybrid analysis techniques and draw their comparisons. They conclude that the hybrid analysis approach was the one to achieve the better results.

Note, however, that although the hybrid analysis approach was shown to be superior to static or dynamic, the feature selection process remains the most critical factor in determining the detection accuracy.

Several studies investigated Android application permission system. REAPER [36] is a tool that traces the permissions requested by apps in real time and discriminates them from other permissions requested by third-party libraries linked with the app. Fasano et al. [37] proposed a formal method to detect the exact point in the code of an Android application where a permission is invoked at runtime. Dilhara et al. [38] proposed a technique to automatically detect the incompatible permission used in a given app and repairs them when found.

4. Methodology

This section presents our proposed methodology to the classification process. The general methodology of the proposed malware detection Android systems is shown in Figure 1. Commensurate with the figure, the hybrid approach is divided into two stages: (1) static analysis and (2) dynamic analysis. In the first phase of the static analysis stage, APK files are converted from XML to JSON. After that, the process of scanning JSON files begins to extract the permissions pertinent to the application involved. Once these permissions are made available via static analysis leveraging machine learning, the application is classified as either malicious or suspicious. With this, the static analysis stage terminates, as the dynamic analysis stage commences. The dynamic analysis phase aims to do more checks for any suspicious application. By analyzing the application's runtime behaviors, the application in question is executed to determine the observed dynamic features such as system call, score, intent, and action. The dynamic features obtained are made available for dynamic analysis based on machine learning for final classification. In the process, suspicious applications are classified as either malicious or benign. The applications that get classified as malware are added to the list of malware, as applications that got classified as benign get added to the list of benign applications for future reference.

Each step will, in turn, be separately explained below.

4.1. Dataset. Several studies presented and tested a large-scale datasets of android applications including runtime traces for research community, such as AndroCT [39] and AndroZoo [40]. In this paper, we use a dataset that consists of benign, malware, and Greyware applications. Palo Alto Networks collected these data over a month period back in 2017 [41]. It was derived from a series of mixed Android APK files containing an XML report representing the metadata built during the static and dynamic analyses.

4.2. Extraction of Features. In recent times, it has become common practice to come across datasets containing hundreds of thousands of features. Over time, it became more noticeable that the number of features often converges to the number of observations stored in the database, something that results in attaining a machine learning model that suffers from overfitting [42].To avoid this problem, the tendency is to reduce the number of features by creating a new set of features deduced from the original set, as we create a set of features capable of reducing most of the data's original features. Therefore, in this proposed work, we aim to extract the most prominent features out of dynamic and static analysis while marginalizing the less valued features to create a new set of dynamic and static features.

In reviewing various dataset files emanating from a multitude of applications, it was observed that certain features (over others), both dynamic and static, could particularly impact the classification process altogether. More precisely, we noticed that certain actions were frequently repeated in multiple malware files. We also noticed that three ports were the most that were being used: 443, 80, and 5200. Additionally, we noticed certain IP addresses that were often used, which were sources for further attention and suspicion.

In our database there are two types of analysis:

(1) In the database files subject of our investigation, we conducted two types of analysis: static analysis: the features emanating from this were extracted directly from the source code of the applications. Moreover, among the features that could, also, be extracted from APK applications include the following:

   (i) Permissions: on Android systems, the permission mechanism is relied upon to protect the privacy of users, with the particular objective of requesting permission from the application in order to access data and system features such as calls and camera as most of the time these programs want to request a special set of permissions. Hence, it is good to scrutinize this with care when extracting the anticipated features

   (ii) Intent: usually, some malware families use intent to activate their malicious activity directly after restarting smart devices

   (iii) Suspicious API calls: it is a set of application interfaces that can be used to access sensitive resources and information, as it sometimes leads to harmful behavior without any permission request

   (iv) Restricted API calls: this type of API call aims to achieve roughly the same goal as would suspicious API calls, save for the fact that it is protected by permissions

(2) Dynamic analysis: this type of analysis addresses the features that can be extracted during the implementation of an application under a safe environment where the implementation process is recorded. In our research, there is a number of dynamic features that can be extracted from Android applications. Features that can be extracted during the implementation of the applications include the following:

   (i) DNS query: it is a request which requires information sent from the user to the server. It is considered one of the solutions used to discover and block DNS queries for malicious behavior

   (ii) IP address: there is a number of factors that can make an IP address a suspicious address. Amongst these factors is the sending a lot of spam and linking to devices full of malware and other different suspicious patterns of behavior. Therefore, it is rather essential to be able to differentiate between the IP addresses are they benign or those that arouse suspicion

   (iii) Port: the port feature is a virtual site that establishes connection to the network from the beginning right to the end of the interval of interest, as when sending a package or a group of specific IP address packets. The computer would identify the port to which the packets will be directed commensurate with the components of the application or package

   (iv) Action: this feature refers to an application's activity and is represented in the number of diverse behaviors and activities, which are represented by patterns of access that may be normal or suspicious. Therefore, identifying these activities is very important because they lead to identify the particular suspicious activity and its source, thereby mitigating any damage caused by such activities

4.3. Feature Selection. After completing the stage of features extraction and identifying new features, we move on to selecting the features stage. The feature selection process is one of the most important steps with which we aim to choose the features from the pool of newly identified features in such a way as to achieve an increase in accuracy, a reduction in complexity, and, in the meantime, avoid any overfitting. Researchers have traditionally applied a number of feature classification approaches to detect malware in applications. In this endeavor, we particularly employ the feature rank approach, as this method leverages certain crucial elements in the arrangement of features due to its ability to choose the appropriate features needed to build the malware detection models.

The feature rank process is based on the random forest [43], as it is one of the ML methods, due to its relatively good accuracy, simplicity, ease of use, and strength. All of these harvested advantages have made it a powerful way to choose features. The random forest (RF) approach consists of a group of decision trees. Each node in the decision tree represents one particular feature, which makes measuring the importance of the feature a fairly simple process. When training a decision tree, we calculate the amount of each feature that reduces impurities in the tree. When applied to the forest, the average impurity reduction for each feature is calculated to arrange the features according to the average number of impurities.

The feature selection process, depending on random forests, can be summarized as follows:

(i) As a first step, we work towards classify each feature separately by implementing a random forest algorithm

(ii) As a second step, we work on classifying features based on minimizing impurities, as the features of less importance are removed, while other features are preserved

(iii) In the third step, we basically reapply the random forest algorithm to the retained features, with the objective of accessing the final essential features

Our dataset contains several features, as we may use all of these features in the process of detecting malware.

After completing the feature selection process based on RF, we obtain the top dynamic and static features:

(1) Dynamic: action repetition

(2) Static: permissions

4.4. Machine Learning Techniques. Machine learning is an integral part of artificial intelligence. It is based on self-learning and development depending on vast amounts of data without clear programming. Machine learning is divided into supervised machine learning, unsupervised machine learning, semisupervised machine learning algorithms, and reinforcement machine learning algorithms. Supervised machine learning algorithms are the most widely used because they depend on the classification of a dataset. This is particularly so due to their ability to analyze the training dataset, and they can classify them into classes according to their characteristics [42].

In this paper, we opted to implement four machine learning algorithms: naïve Bayes (NB) that is based on the principle of maximum likelihood; random forest (RF), which is based on the initiation of packing and the random characteristic for decision making; the decision tree (DT), which is based on the principle of computing all traits and then making the decision; and XGBoost which enhances parallel tree boosting with a fast and accurate mechanism together with gradient boosting. In the sequel, we discuss the features for each algorithm separately, as follows:

(1) The random forest (RF) algorithm: RF is one of the standard algorithms used for the intended purpose; it is based on decision trees, leading to better prediction accuracy. The nomenclature (forest) draws on the fact that it contains a group of decision trees, and it is used with the objective to develop the trees based on independent subsets of the dataset. The algorithm stands out due to its ease of calculation and ability to adapt quickly, and it is considered a stable algorithm

(2) The decision tree (DT) algorithm: DT is one algorithm that leverages the training process to build data structures resembling a tree structure. It is used to make predictions about the test data. The tree that is established as such consists of decision nodes and leaf nodes. This algorithm has gained wide popularity due to its simplicity. Further, its achieves higher accuracy based on fewer decisions taken [44]

(3) XGBoost: this is an algorithm that is considered one of the improved algorithms. It was first derived with the objective to set up an algorithm that achieves high efficiency, portability, and fast calculations. It is now one of the most widely used algorithms in machine learning due to its ability to deal effectively in handling a whole host of problems, be they regression or classification type problems; and it has proven itself well in terms of speed and robustness

(4) The gradient boosting algorithm: the idea behind this algorithm arose from the idea of reinforcement. It functions based on iterative learning, and how many weak learners there exists, in order to obtain a robust model capable of dealing with classification and regression problems. It represents an algorithm capable of improving the cost function [45]

5. Experiment Setup

In this section, we will discuss the steps that were followed in this work to achieve detection of malware in more elaborate detail.

```
1.  Being
2.  Create array of action from apk_ api.
3.  Create array,it have malware permissions which are used by applications.
4.  Read the File dynamic and Static analysis.
5.  For X in range Files
6.  Path File _Dynamic= files [dynamic _json_ file_ path][x]
7.  Path File _Static= files [static _ json_ file_path][x]
8.  open path file_ dynamic
9.  dynamic_feature= load data
10.  open path file_ static
11.  static_feature= load data
12.  Insert [id = dynamic_feature,
13.  port = dynamic_feature,
14.  Ip=dynamic_feature,
15.  If dynamic features.get(action)==action[i]
16.  action =dynamic _features,
17.  If static features.get(permission)==permission[i]
18.  permission=static _feature
19.  end
```

Algorithm 1: Pseudocode of the selection features.

5.1. Data Preparation. At the onset of this research endeavor, we reviewed numerous files for benign and malware-infected applications. Particularly, we had to deal with a total of 195623 applications, consisting of 104747 malware-containing applications and 90876 benign applications. However, the application files would contain some unwanted/unnecessary files that must be removed before pursuing any of the steps prescribed herein as they would not offer adequate pertinent information. Consequently, we ended up having to deal with 166710 files.

With that, we had to split out the static benign files pursuant with their size as the variation in file size ranged from 2.17 KB to 4.12 MB. Files got divided it into two groups; the first group contained files the sizes for which were found to exceed 1 MB; the other group contained data that was less than 1 MB in size. Meanwhile, there was no need to make a similar split for the dynamic benign files. This was not particularly needed as the variations in the file sizes exhibited a smaller range: 498 bytes to 1 MB. Finally, the malware files were split out according to the family labels, which were made up of five families: Trojan, Pup, Adware, Spyware, and Riskware.

5.2. Extract and Select Features. In the next phase, the most important features were extracted and selected based on the classification process prescribed earlier. Algorithm 1 illustrates the feature selection process where we first read all the dynamic and static files within the field. Then, we load the essential features and rely on the following dynamic feature: being action repetition. In the process, we extracted all actions for malware and benign applications with a total number of 176 actions. We also relied on the static permissions feature and extracted the most important permissions that referred to malware applications, where we extracted 668 permissions from a total of 4276 permissions. In the sequel, we will elaborate further on the importance of the permission and the action repetition features. Additionally, we will scrutinize some of the permissions and the action repetitions with the highest bearing on the classification process:

Table 1: Permission for static analysis.

Importance	Features name
0.587484	android.permission.WRIT_EXTERNAL_STORAGE
0.088250	com.google.android.c2dm.permission.RECEIVE
0.049034	android.permission.READ_LOGS
0.033341	android.permission.vending.BILLING
0.018864	android.permission.USE_CREDENTIALS
0.014858	android.permission.GET_PACKET_SIZE
0.013001	android.permission.ACCESS_NETWORK_STATE
0.012275	android.permission.GET_TASKS
0.011658	android.permission.SYSTEM_ALERT_WINDOW
0.010321	android.permission.CAMERA

(1) Permissions: Android permissions are one of the most important security features that can be provided in Android systems as most applications that are downloaded through the Android Play Store are accompanied by a set of permissions that each application requires. Therefore, it is one of the important features that has a great impact on the process of detecting malware. In our experience, a number of permissions were extracted that had the top impact on the classification process, as shown in Table 1

(2) Action repetition: monitoring the actions of Android applications is an important process, as it is one of the features that helps to identify harmful procedures from others, but we have depended on our work on the frequency of these procedures in the

Table 2: Action repetition for dynamic analysis.

Importance	Features name
0.302175	APK file removed the launcher icon
0.197239	File tried to connect to a malicious URL
0.090943	APK file deleted a file
0.080813	APK file displayed a float window
0.026626	APK file uploaded geolocation information to the remote server
0.025072	APK file fetched the information of apps installed on the device
0.023656	File wrote a file on the device
0.021470	APK file tried to connect to the URL
0.020729	File wrote an ELF library file on the device
0.017581	APK file sent out an SMS message

classification process, being that sometimes the procedure may not indicate the presence of malware programs, but repetition may lead to the discovery of these programs. Table 2 shows ten of the top actions relied in classification

5.3. Training of Machine Learning-Based Classifiers. During this stage, we split the dataset into training dataset (70%), testing dataset (15%), and validation dataset (15%). The training dataset is used to build the model. Once the model is established, we use the test dataset to determine whether the model was trained properly in order to provide the most accurate results. The training dataset used consists of 104747 malware applications and 90,876 benign applications. The training process for a machine learning-based static and dynamic analyzer begins by analyzing an Android application into JSON files to extract the permission-related features; the extracted permissions are out of a total of about 668 unique permissions. Furthermore, about 176 actions are collected by the dynamic analyzer. As the training process commences both static and dynamic features are identified concurrently.

Meanwhile, to handle the malware families, we broke them down commensurate with the data split ratios as shown in Table 3.

6. Results and Discussion

In this section, we provide the details of the steps that were taken to investigate the results that were obtained. We, also, analyze the performance of the various classifiers used in the study, showing the results for four different machine learning algorithms (RF, DT, XGBoost, and gradient boosting) used in detecting malware with the grid search mechanism being employed for hyperparameter tuning. This is illustrated in Table 4.

We applied classifiers to dynamic and static features, each applied independently to the appropriate type of features involved. Our results have been categorized commensurate with results based on dynamic features without permissions, results based on static features without action repetition, and results based on dynamic and static features.

Table 3: Malware family details.

Family_ Label	Number	Traning (70%)	Testing (15%)	Validating (15%)
Adware	4661	3263	699	699
Riskware	5	3	1	1
Trojan	82425	57697	12364	12364
Spyware	54	38	8	8
Pup	17603	12323	2640	2640

Equations (1), (2), (3), and (4) show the measures that were taken in the process of testing the performance for each machine learning algorithm. Pursuant with the equations shown, FP (false positive) refers to the number of applications classified as malware when, in reality, they are benign. FN (false negative) refers to the number of applications classified as benign but, in reality, can readily be classified as malware. TP (true positive) indicates the number of applications that are properly classified as malware, while TN (true negative) indicates the amount of applications that are properly classified as benign.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad (1)$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad (2)$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}, \quad (3)$$

$$F1 - Score = \frac{2 * \text{Precious} * \text{Recall}}{\text{precious} + \text{Recall}}. \quad (4)$$

The static, hybrid, and dynamic models would normally be expected to achieve high F1 score, accuracy, recall, and precision. We trained the analysis tool used on some dataset and used optimized parameters for machine learning. In the proposed mechanism, we firstly do the static analysis part comprising the manifest file, which includes permission. Following this, the mechanism, as prescribed, invokes the dynamic classification model and detects it under a control

Table 4: Grid search setting.

Classifier	Parameter

XGBoost	Permission Only:{'learning rate':0.1,'max_depth':10,'n_estimators':500,'subsample':0.5}
	Action Only:{'learning rate':0.1,'max_depth':10,'n_estimators':200,'subsample':1.0}
	Permission & Action:{'learning rate':0.1,'max_depth':5,n_estimators':500,'subsample':0.5}

Gradient boosting	Permission Only: {'learning rate':0.1,'max_depth':10,'n_estimators':500,'subsample':0.5}
	Action Only:{'learning rate':0.1,'max_depth':10,'n_estimators':200,'subsample':1.0}
	Permission & Action:{'learning rate":0.1,max_depth':5,'n_estimators':500,'subsample':0.5}

DT	Permission Only:{'criterion':'gini' ,'max_depth':10 ,'max_features':'auto','n_estimators':100};
	Action Only:{'criterion': 'gini', 'max*depth'* : 7,' *maxf eatures'* :' *log2*',' *nestimators'* : 10}
	Permission & Action:{'criterion':'entropy','max_depth":7,'max_features':'auto','n_estimators':10}

RF	Permission Only:{'criterion':'gini','max_depth':10,'max_features':'auto','n_estimators":100}
	Action Only:{'criterion': 'gini', 'max*depth'* : 7,' *maxf eatures'* :' *log2*',' *nestimators'* : 10}
	Permission & Action:{'criterion':'entropy','max_depth':7,'max_features':'auto','n_estimators':10}

Table 5: The summary of results of validation file.

Classifier	(a) Static feature result				(b) Dynamic feature result				(c) Hybrid feature result			
	Accuracy	Precision	Recall	F1_ score	Accuracy	Precision	Recall	F1_ score	Accuracy	Precision	Recall	F1_ score
Gradient boosting	0.9954	0.996	0.9967	0.9964	0.993	0.997	0.992	0.994	0.997	0.998	0.996	0.997
XGBoost	0.9956	0.9959	0.9971	0.9965	0.993	0.997	0.992	0.994	0.998	0.999	0.997	0.998
Decision tree	0.9946	0.99501	0.9964	0.9947	0.987	0.991	0.989	0.987	0.996	0.997	0.996	0.996
Random forest	0.962	0.947	0.994	0.970	0.943	0.967	0.941	0.954	0.976	0.994	0.967	0.980

Table 6: The summary of results of test file.

Classifier	(a) Static feature result				(b) Dynamic feature result				(c) Hybrid feature result			
	Accuracy	Precision	Recall	F1_ score	Accuracy	Precision	Recall	F1_ score	Accuracy	Precision	Recall	F1_ score
Gradient boosting	0.995	0.994	0.9969	0.996	0.993	0.997	0.991	0.994	0.997	0.999	0.997	0.998
XGBoost	0.995	0.9946	0.997	0.9958	0.993	0.997	0.991	0.994	0.998	0.997	0.997	0.998
Decision tree	0.9941	0.994	0.9969	0.9942	0.987	0.991	0.988	0.987	0.995	0.997	0.996	0.996
Random forest	0.946	0.951	0.994	0.972	0.943	0.965	0.942	0.953	0.974	0.993	0.966	0.979

environment. Finally, the proposed method would apply the hybrid model. In this way, we would have tested the application against the three different models explained.

The static, hybrid, and dynamic models would normally be expected to achieve high F1 score, accuracy, recall, and precision. We trained the analysis tool used on some dataset and used optimized parameters for machine learning. In the proposed mechanism, we firstly do the static analysis part comprising the manifest file, which includes permission. Following this, the mechanism, as prescribed, invokes the dynamic classification model and detects it under a control environment. Finally, the proposed method would apply the hybrid model. In this way, we would have tested the application against the three different models explained.

Table 5 shows a summary of validation results file. The results show comparable results when XGBoost and gradient boosting are used while reflecting favorably better results than when both DT and RF are implemented. Further, Table 6 shows a summary of the test file results. The results reveal comparable results when both XGBoost and gradient boosting are used and better results than when both DT and RF are applied.

From the results obtained, note that all the classifiers have achieved good results. In our work, we particularly relied on the action repetition feature in which the actions are related to each other. Moreover, the results indicate that we, in fact, have identified a feature which had not been targeted directly, namely, action repetition. Finally, we observe from the our work done on each of the features separately, or when applying the hybrid analysis method, that we have achieve good results in the classification process. Now, in order to analyze the required cost needed in terms of device memory and CPU for every model we evaluated in our experiments, we captured the cost needed for both training and execution using the IPython command which was particularly used to capture these factors:

Table 7: Permission.

Classifier	Train			Test			Validate		
	CPU times		Wall time	CPU times		Wall time	CPU times		Wall time
	User	System		User	System		User	System	
Gradient boosting	9 min 17 s	549 ms	17 min 23 s	1.07 s	20 ms	1.09 s	1.09 s	24 ms	1.1 s
XGBoost	53 min 45 s	1.81 s	2 min 44 s	1.64 s	1 ns	91.8 ms	1.76 s	4.401 ms	142 ms
Decision tree	16.5 s	996 ms	6.33 s	515 ms	36 ms	121 ms	530 ms	20.1 ms	123 ms
Random forest	2.41 s	56 ms	2.49 s	57.7 ms	1 ns	55.6 ms	57 ms	7 Ms	54.9 ms

Table 8: Action.

Classifier	Train			Test			Validate		
	CPU times		Wall time	CPU times		Wall time	CPU times		Wall time
	User	System		User	System		User	System	
Gradient boosting	56 s	696 ms	10 min 40 s	707 ms	64 ms	898 ms	731 ms	68 ms	918 ms
XGBoost	2 min 23 s	292 ms	16.8 s	1.94 s	60 ms	357 ms	1.91 s	64 ms	356 ms
Decision tree	969 ms	852 ms	1.84 s	222 ms	136 ms	385 ms	217 ms	140 ms	385 ms
Random forest	1.77 s	100 ms	1.98 ms	130 ms	64 ms	342 ms	108 ms	104 ms	357 ms

Table 9: Action and permission.

Classifier	Train			Test			Validate		
	CPU times		Wall time	CPU times		Wall time	CPU times		Wall time
	User	System		User	System		User	System	
Gradient boosting	8 min 48 s	792 ms	16 min 4 s	797 ms	116 ms	1.04 s	814 ms	112 ms	1.06 s
XGBoost	38 min 53 s	2.06 s	1 min 58 s	2.86 s	128 ms	414 ms	2.62 s	96 ms	399 ms
Decision tree	1.83 s	1.38 s	2.94 s	309 ms	184 ms	521 ms	326 ms	176 ms	539 ms
Random forest	5.04 s	76 ms	5.25 s	174 ms	120 ms 4	464 ms	177 ms	160 ms	486 ms

(i) %%time: time of the execution for the entire cell. From this command, we were able to get the CPU times for the amount of time the CPU takes for running exclusively the code, the amount of time the CPU takes on system calls, combined user and system times, and the actual time taken by a computer to complete a task (it is the sum of three terms: CPU time, I/O time, and the communication channel delay)

(ii) %memit: measure the memory use of a single statement. From this command, we were able to get the background memory usage from the Python interpreter itself and also the memory needed each line of code affects the total memory needed

According to data revealed in Tables 7–9, the time took for the model to be trained was shown not to exceed a few seconds (3.21 s and 5.12 s) for the random forest and decision tree models and a few minutes (8 min 49 s) for the gradient boosting model. Moreover, for the XGBoost, it was observed that XGBoost consumed longer times due to the formations of trees and parameters involved in the process (38 min 54 s). Meanwhile, however, the time needed in the process is seen as an "optimal" amount of time considering the large number of applications on which models are trained. As for the testing phase, the time observed has been rather short. In particular, when calculated for one application, it was noted that the time needed would not exceed few microsecond (199.26 μs). Furthermore, it was noted that the amount of memory used in the action model is reasonably small commensurate with the number of features used in the model. By running a comparison upon the classifier models, XGBoost was found to consume a significant amount of memory due to the repetition of building the tree, pursuant with what was elaborated earlier(3721.53 MiB).

7. Conclusion

Recent research endeavors across the literature and current technologies deployed that seek to detect malicious programs have lacked a more complete study addressing the collective impact of both action repetition and permissions deployed together to classify Android applications as being malicious or otherwise. In this paper, we investigated the performance of four machine learning classifiers that seek to detect malware depending on the dynamic (action repetition) and static (permissions) features. These features were found to have significant bearing and a key role in the classification process. In particular, we applied the four classifiers in three stages. In the first stage, we leveraged the dynamic features in the dataset. In the second stage, we used the static features involved, while the third stage encompassed a combination of both dynamic and static features.

In the three stages used, the results indicate that classification was done to high accuracy. As we implemented the proposed methodology, we, leveraging the results obtained, have shown that new features, never addressed before, can be extracted in the process. We have also demonstrated that a combination of the permissions and action repetition features has achieved good results in detecting malware in Android applications. Also, the results showed that accuracy achieved from static, dynamic, and hybrid analyses was above 94%, so using static analyses alone should be efficient and less in cost for classification in our case.

As future work, we are planning to investigate and get one step further to develop a novel method based on machine learning to classify malware families rather than binary classification.

Data Availability

The datasets generated during and/or analysed during the current study are available upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research received partial funding from the Jordan University of Science and Technology.

References

[1] "Smartphonemarket," 2020, https://www.idc.com/promo/smartphone-market-share/os.

[2] "Number of android apps on google play," 2020, https://www.appbrain.com/stats/number-of-android-apps.

[3] G. Suarez-Tangil and G. Stringhini, "Eight years of rider measurement in the android malware ecosystem: evolution and lessons learned," 2018, https://arxiv.org/abs/1801.08115.

[4] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, "A review of android malware detection approaches based on machine learning," *IEEE Access*, vol. 8, pp. 124579–124607, 2020.

[5] H. Cai, F. Xiaoqin, and A. Hamou-Lhadj, "A study of run-time behavioral evolution of benign versus malicious apps in android," *Information and Software Technology*, vol. 122, article 106291, 2020.

[6] H. Cai and B. Ryder, "A longitudinal study of application structure and behaviors in android," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2934–2955, 2021.

[7] M. Noman, M. Iqbal, and A. Manzoor, "A survey on detection and prevention of web vulnerabilities," *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 6, pp. 521–540, 2020.

[8] B. Baskaran and A. Ralescu, *A study of android malware detection techniques and machine learning*, In MAICS, 2016.

[9] W. Zhang, H. Wang, H. He, and P. Liu, "DAMBA: detecting android malware by ORGB analysis," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 55–69, 2020.

[10] S. Alam, S. A. Alharbi, and S. Yildirim, "Mining nested flow of dominant APIs for detecting android malware," *Computer Networks*, vol. 167, article 107026, 2020.

[11] O. Olukoya, L. Mackenzie, and I. Omoronyia, "Towards using unstructured user input request for malware detection," *Computers & Security*, vol. 93, article 101783, 2020.

[12] P. Ravi Kiran Varma, K. P. Raj, and K. V. S. Raju, "Android mobile security by detecting and classification of malware based on permissions using machine learning algorithms," in *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*, pp. 294–299, Palladam, India, 2017.

[13] J. Song, C. Han, K. Wang, J. Zhao, R. Ranjan, and L. Wang, "An integrated static detection and analysis framework for android," *Pervasive and Mobile Computing*, vol. 32, pp. 15–25, 2016.

[14] S. Hahn, M. Protsenko, and T. Müller, "Comparative evaluation of machine learning-based malware detection on android," in *Sicherheit 2016-Sicherheit, Schutz und Zuverlässigkeit*, Gesellschaft für Informatik e.V, 2016.

[15] Q. Fang, X. Yang, and C. Ji, "A hybrid detection method for android malware," in *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pp. 2127–2132, Chengdu, China, 2019.

[16] R. Surendran, T. Thomas, and S. Emmanuel, "A TAN based hybrid model for android malware detection," *Journal of Information Security and Applications*, vol. 54, article 102483, 2020.

[17] W.-C. Kuo, T.-P. Liu, and C.-C. Wang, "Study on android hybrid malware detection based on machine learning," in *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)*, pp. 31–35, Singapore, 2019.

[18] H. Zhang, S. Luo, Y. Zhang, and L. Pan, "An efficient android malware detection system based on method-level behavioral semantic analysis," *IEEE Access*, vol. 7, pp. 69246–69256, 2019.

[19] N. J. Ratyal, M. Khadam, and M. Aleem, "On the evaluation of the machine learning based hybrid approach for android malware detection," in *2019 22nd International Multitopic Conference (INMIC)*, pp. 1–8, Islamabad, Pakistan, 2019.

[20] C. Yang, Z. Xu, G. Gu et al., "Droidminer: automated mining and characterization of fine-grained malicious behaviors in android applications," in *European symposium on research in computer security*, pp. 163–182, Cham, 2014.

[21] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti, "Anastasia: android malware detection using static analysis of applications," in *2016 8th IFIP international conference on new technologies, mobility and security (NTMS)*, pp. 1–5, Larnaca, Cyprus, 2016.

[22] H. Cai, "Assessing and improving malware detection sustainability through app evolution studies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 2, pp. 1–28, 2020.

[23] H. Cai, "Embracing mobile app evolution via continuous ecosystem mining and characterization," in *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pp. 31–35, Seoul, Republic of Korea, 2020.

[24] F. Xiaoqin and H. Cai, "On the deterioration of learning-based malware detectors for android," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE- Companion)*, pp. 272-273, Montreal, QC, Canada, 2019.

[25] L. Cai, Y. Li, and Z. Xiong, "Jowmdroid: android malware detection based on feature weighting with joint optimization of weight-mapping and classifier parameters," *Computers & Security*, vol. 100, article 102086, 2021.

[26] S. Y. Yerima, M. K. Alzaylaee, and S. Sezer, "Machine learning-based dynamic analysis of android apps with improved code coverage," *EURASIP Journal on Information Security*, vol. 2019, Article ID 4, 2019.

[27] A. Mahindru and P. Singh, "Dynamic permissions based android malware detection using machine learning techniques," in *Proceedings of the 10th innovations in software engineering conference*, pp. 202–210, Jaipur, India, 2017.

[28] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Emulator vs real phone: android malware detection using machine learning," in *Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics*, pp. 65–72, Jaipur, India, 2017.

[29] H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: effective android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2019.

[30] H. Cai and J. Jenkins, "Towards sustainable android malware detection," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 350-351, Gothenburg, Sweden, 2018.

[31] X. Ke, Y. Li, R. Deng, K. Chen, and X. Jiayun, "Droidevolver: self-evolving android malware detection system," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 47–62, Stockholm, Sweden, 2019.

[32] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song, and H. Yu, "SAMADroid: a novel 3-level hybrid malware detection model for android operating system," *Access*, vol. 6, pp. 4321–4339, 2018.

[33] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "Stormdroid: a streaminglized machine learning-based system for detecting android malware," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pp. 377–388, Xi'an, China, 2016.

[34] X. Wang, Y. Yang, Y. Zeng, C. Tang, J. Shi, and K. Xu, "A novel hybrid mobile malware detection system integrating anomaly detection with misuse detection," in *Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services*, pp. 15–22, Paris, France, 2015.

[35] Y. Liu, Y. Zhang, H. Li, and X. Chen, "A hybrid malware detecting scheme for mobile android applications," in *2016 IEEE International Conference on Consumer Electronics (ICCE)*, pp. 155-156, Las Vegas, NV, USA, 2016.

[36] M. Diamantaris, E. P. Papadopoulos, E. P. Markatos, S. Ioannidis, and J. Polakis, "Reaper: real-time app analysis for augmenting the android permission system," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pp. 37–48, New York, NY, USA, 2019.

[37] F. Fasano, F. Martinelli, F. Mercaldo, and A. Santone, "Android run-time permission exploitation user awareness by means of formal methods," in *ICISSP*, pp. 804–814, Valletta, Malta, 2020.

[38] M. Dilhara, H. Cai, and J. Jenkins, "Automated detection and repair of incompatible uses of runtime permissions in android apps," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pp. 67–71, Gothenburg, Sweden, 2018.

[39] W. Li, F. Xiaoqin, and H. Cai, "Androct: ten years of app call traces in android," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 570–574, Madrid, Spain, 2021.

[40] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: collecting millions of android apps for the research community," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 468–471, Austin, TX, USA, 2016.

[41] I. Amit, J. Matherly, W. Hewlett, Z. Xu, Y. Meshi, and Y. Weinberger, "Machine learning in cyber-security-problems, challenges and data sets," 2018, https://arxiv.org/abs/1812.07858.

[42] "Towards data science," 2019, https://towardsdatascience.com/feature-extraction-techniques-d619b56e31be.

[43] M. S. Rahman, M. K. Rahman, S. Saha, M. Kaykobad, and M. S. Rahman, "Antigenic: an improved prediction model of protective antigens," *Artificial Intelligence in Medicine*, vol. 94, pp. 28–41, 2019.

[44] A. J. Myles, R. N. Feudale, Y. Liu, N. A. Woody, and S. D. Brown, "An introduction to decision tree modeling," *Journal of Chemometrics: A Journal of the Chemometrics Society*, vol. 18, no. 6, pp. 275–285, 2004.

[45] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of Statistics*, vol. 29, no. 5, 2001.