Journal of Information Security and Applications 58 (2021) 102696

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa

NATICUSdroid: A malware detection framework for Android using native and custom permissions

Akshay Mathur [a], Laxmi Mounika Podila [a], Keyur Kulkarni [a], Quamar Niyaz [b], Ahmad Y. Javaid [a,*]

[a] The University of Toledo, 2801 W Bancroft St, Toledo, OH 43606, USA
[b] Purdue University Northwest, 2200 169th St, Hammond, IN 46323, USA

ARTICLE INFO

Keywords:
Android
Malware
Application security
Static features
Classification

ABSTRACT

The rapid growth of Android apps and its worldwide popularity in the smartphone market has made it an easy and accessible target for malware. In the past few years, the Android operating system (AOS) has been updated several times to fix various vulnerabilities. Unfortunately, malware apps have also upgraded and adapted to this evolution. The ever-increasing number of native AOS permissions and developers' ability to create custom permissions provide plenty of options to gain control over devices and private data. Therefore, newly created permissions could be of great importance in detecting current malware. Previous popular works on malware detection used apps collected during 2010–2012 to propose malware detection and classification methods. A majority of permissions used in those apps are not as widely used or do not exist anymore. In this work, we present a novel malware detection framework for Android called *NATICUSdroid*, which investigates and classifies benign and malware using statistically selected native and custom Android permissions as features for various machine learning (ML) classifiers. We analyze declared permissions in more than 29,000 benign and malware collected during 2010–2019 to identify the most significant permissions based on the trend. Subsequently, we collect these identified permissions that include both the native and custom permissions. Finally, we use feature selection techniques and evaluate eight ML algorithms for *NATICUSdroid* to distinguish benign apps from malware. Experimental results show that the Random Forest classifier based model performed best with an accuracy of 97%, a false-positive rate of 3.32%, and an f-measure of 0.96.

1. Introduction

The open-source nature of Android has made it the most widely adopted operating system (OS) in smart devices such as wearables, smartphones, and smart TVs [1,2]. As of 2019, Android stands as the major contributor to the mobile market with more than 2 billion active devices [3]. Android holds a little more than 74% of the world's total user share, making it the most popular mobile OS, out of which almost 87.4% of Android users are running Android 6.0 and above in their devices, as shown in Fig. 1 [4,5]. Such popularity and ease of availability, in turn, encourages malware developers to build sophisticated malware that can either compromise the device or the private data stored on them [6–10]. Malware compromise the security of a device using its resources (such as camera, microphone, Wi-Fi, and GPS), or accessing private data (such as contacts, emails, call logs, and messages) [11]. Similarly, repackaging a benign app with malicious code or an app update that requests more permissions than required may change the nature of the app from benign to malicious [12].

To limit malware infestation, Google launched *Play Protect* platform to identify malware in *Google Play Store* (referred to as Play Store from now onward) that helped in detecting 700,000 malicious apps [13]. More than 300 of these apps were responsible for various Distributed Denial of Service (DDoS) attacks [14]. Although Play Protect discovered a large number of malicious apps in the Play Store, there were several malicious apps that it could not detect. Among them, 145 malware were reported to be laden with Windows PC malware in 2018, and seven Stalkerware apps with more than 130,000 downloads were caught accessing user data without their consent in 2019 [15,16]. In addition, 85 Adware (with 8 million downloads) were found pretending to be gaming or photography apps [17]. Later, Google removed these apps from the Play Store after their discovery. While Play Store is the official place for Android apps, third-party app stores (such as Samsung, Sony, Huawei, Aptoide, GetJar, ACMarket, and Appchina) are also popular and prone to malware infection. This poses a significant threat to users' privacy and data when they download apps from such third-party app stores.

Android's Application Programming Interface (API) level uniquely identifies the framework API version of the AOS. The apps can use

* Corresponding author.
E-mail address: ahmad.javaid@utoledo.edu (A.Y. Javaid).

https://doi.org/10.1016/j.jisa.2020.102696

Available online 13 January 2021

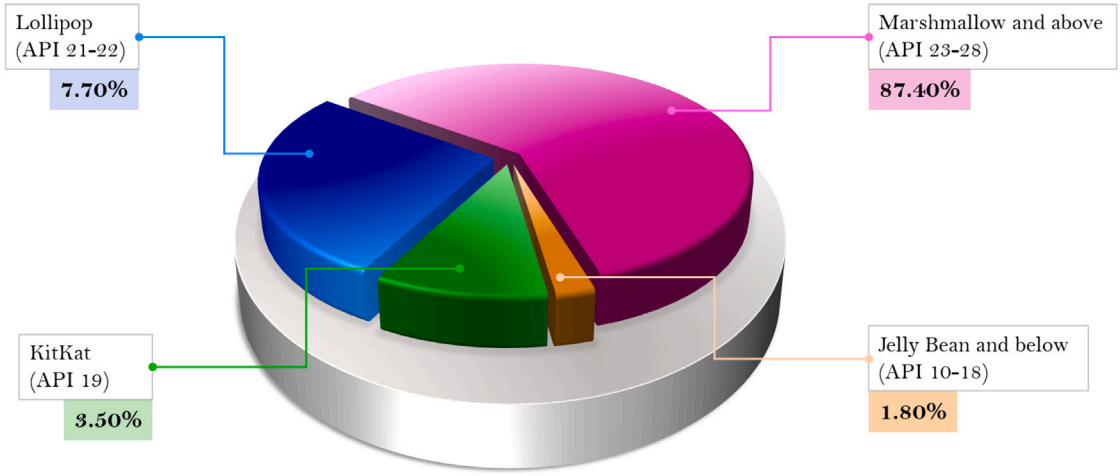


Fig. 1. Android market user-share for different APIs and versions [5].

this API to interact with the AOS that incorporates a permission-based model. This model allows users to control an app's access to device resources and data, preventing suspicious requests. These permissions are defined under four protection levels: (i) Normal, (ii) Dangerous, (iii) Signature, and (iv) Signature|Privileged [18]. *Normal permissions* are considered to be non-hostile and granted to the app by default, e.g., `SET_ALARM` permission. *Dangerous permissions*, on the other hand, control resources that can affect user data and app performance. Due to their sensitivity, these permissions are requested for user approval, e.g., `CALL_PHONE` that enables an app to make phone calls. All custom permissions defined by app developers fall under this protection level. *Signature permissions* have the highest privileges and are not granted to third-party apps, e.g., `REQUEST_INSTALL_PACKAGES`, that allows apps to request installation of packages such as updates. *Signature|Privileged permissions* are the most sensitive as they may allow altering of critical aspects of the AOS. As a result, they are granted to apps that are either installed in a specific folder or signed with the same certificate as the app that declared the permission, such as pre-installed system apps. For instance, `GET_ACCOUNTS_PRIVILEGED` allows access to the list of accounts being used on the device.

While such a model establishes protection levels for the permissions, it does not provide any insight to the user whether a request is coming from a malicious or benign app. It rather leaves it to the user's discretion to allow or deny any requested permission. In other words, it assumes that the user is well-informed of the nature of the apps and the validity of the requested permissions. To address this issue, several techniques have been proposed to identify the behavior of apps and can generally be categorized into *static* and *dynamic*. Static techniques analyze static features of an app, such as signatures [19–21], source code [22,23], and binary files (`.dex`) [24–27]. They work without running the app as they extract these features from app binaries. On the other hand, dynamic techniques analyze apps' behaviors by running them. These techniques consider various aspects during the run-time, such as CPU utilization [28–30], memory utilization [31,32], and network traffic [33–36] to distinguish benign apps from malicious ones.

Although attackers upgrade and adapt malware apps in parallel to the Android environment, several new techniques still use old apps for analysis. There are 325 permissions in Android API level 28 compared to 166 permissions in API level 15 [37]. Besides, several distributors build custom permissions to access Web APIs, hardware resources, and data. This results in the availability of more permissions and, consequently, more ways to exploit them. This necessitates the analysis of these custom permissions alongside the AOS native permissions. Although permissions defined in `AndroidManifest.xml` have been used in contemporary work to identify malware using ML, to the best of our knowledge, custom and native permissions together have not been considered. Several works have used DREBIN [26] and MalGenome [38] datasets for analysis [19,24,39–41]. However, these datasets contain apps collected between 2010 and 2012 with API levels 9 through 18. With the constantly changing Android environment, there is a need to build a malware detection architecture using more recent and robust data, considering the most critical permissions capable of evolving. Towards this direction, we have made the following contributions in this work:

1. We build NATICUSdroid (NATIve and CUStom permissions analysis for Android), a new malware detection framework that considers statistically significant permissions mentioned in the native Android OS (*native* permissions) and custom permissions built by app developers found in thousands of apps with API levels 23 and above.
2. We also create a malware detection framework that uses only native permissions (after selecting the most significant permissions) for comparing it with our proposed methodology to establish that native permissions alone are not sufficient in detecting malware.
3. We compare the performance of our work with state-of-the-art methods, providing evidence of higher accuracy, lower false-positive rate, and less detection time.

The organization of the paper is as follows. Section 2 discusses the related work in Android malware detection. Section 3 describes the implementation of NATICUSdroid framework. Further, Section 4 provides the results along with a detailed discussion. Finally, Section 5 concludes the paper.

2. Related work

As mentioned earlier, several static analysis based Android malware detection methods and systems have been proposed in the literature. Most of these apply ML techniques for malware detection, either using one of the old Android malware datasets, such as DREBIN, Malgenome, and Contagio for analysis or have low detection/high false-positive rates.

2.1. Permissions-only systems

Several studies have focused only on the analysis of native Android permissions for malware detection. Permission-based Malware Detection Systems (PMDS) applied ensemble learning methods on 2950 apps from the Drebin [26] and Android Malware Genome Project [38] datasets and detected malware with an accuracy of 92%–94% and a

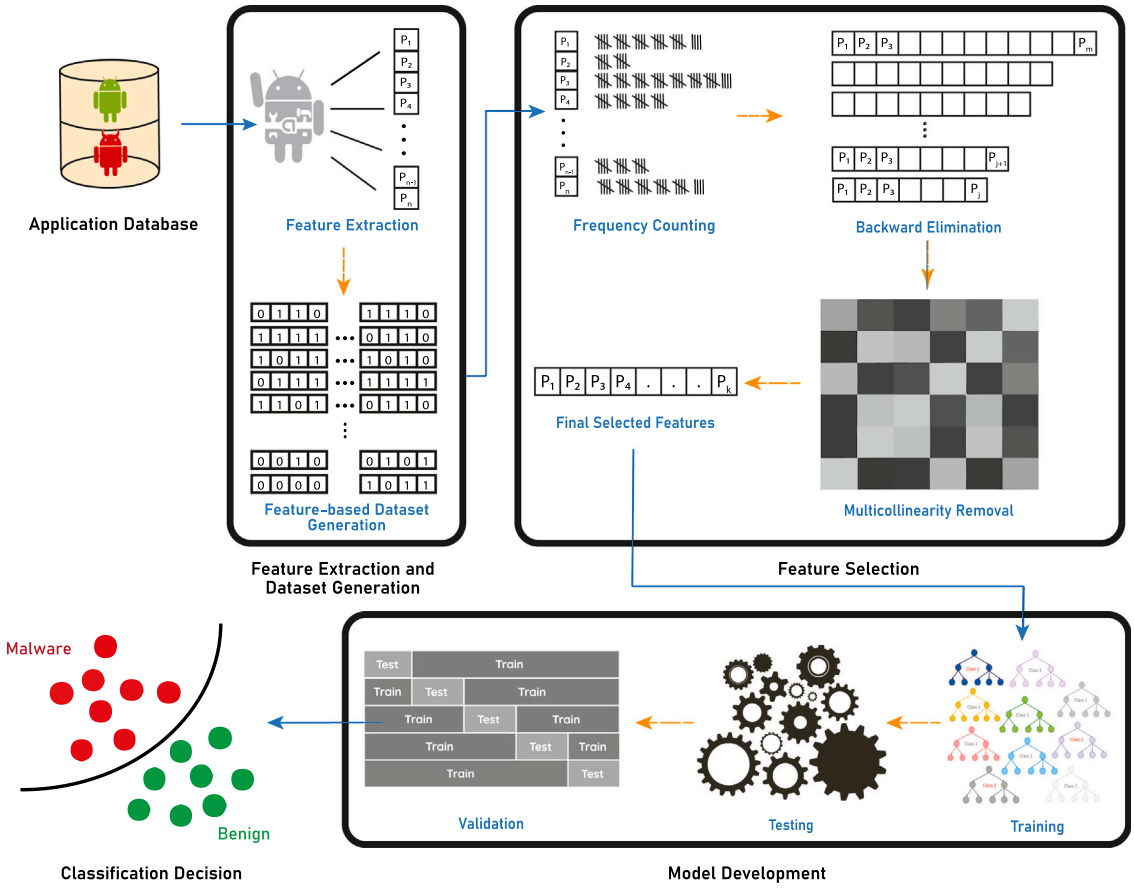


Fig. 2. NATICUSdroid development workflow.

false-positive rate between 1.52–3.93% [42]. Using Android Malware Genome Project and Contagio Mobile dataset [43], ApkAuditor, a client–server and Logistic Regression (LR) based system achieved an accuracy of 88% with 5% false-positive rate [44]. In [45], the authors applied LR, Naive–Bayes (NB), Decision Tree (DT), Random Forest (RF), and K* algorithms on dynamic permissions. A classification accuracy of 97% using LR was reported. Although these methods perform their intended tasks with high accuracy, they often have high false-positive rates. Moreover, they used legacy datasets, such as Malgenome, DREBIN, or Contagio, that have apps from the time when native permissions were prevalent, and custom permissions were not frequently used.

2.2. API-and-intent-based systems

Several works focused on other static features of apps to identify them as benign or malware. APIs and information passed through intents can also be critical in malware detection. By creating Control Flow Graphs (CFG) of API information of more than 20,000 apps, a group of researchers created three different datasets (Boolean, Frequency, and Sequence). A Deep Neural Network (DNN) model for each dataset was created with more than 120 features and a combined model that gave 98.8% accuracy [46]. In another work, systems call sequences, and their semantics were analyzed using natural language processing. Each system call was treated as a sentence in the language, and a model was built using Long Short-Term Memory (LSTM) network. The model exhibited an accuracy of 93.7% with a false-positive rate of 9.3% [47]. These methods classify with high accuracy and use the latest apps for analysis. However, they require high computational power because of high data dimensionality. Moreover, they require complete system call sequences for an app, which is only available when the app has run several times. Several malware infect the device during the very first run, making such approaches impractical.

2.3. Source code features based systems

DREBIN used static analysis on a collection of 123,000 benign and 5,500 malicious apps to detect malware using features such as permissions, API calls, and source-code. Upon testing, it was found that DREBIN could generate the analysis result in less than a minute for most apps with an accuracy of 93% [26]. A group of researchers proposed a framework that uses several app features, such as permissions, manifest, and source code keywords, as inputs to a *bag-of-words* representation model. The authors then used classifiers such as k-Nearest Neighbors (kNN), Support Vector Machines (SVM), and Decision Trees (DT), where SVM achieved the highest accuracy of 85.51% [48]. Another team proposed a hybrid classification system for detecting malware based on their features. The system extracts static and dynamic features (permissions, API calls, and network connections) from malware samples and feeds them to various ML techniques. Malware samples, such as Adware, Riskware, and Trojans from 20 categories, are then categorized with an accuracy of 92% using RF [49]. A serial Convolutional Neural Network (CNN-S) and Deep AutoEncoder - CNN (DAE-CNN) based approach was studied using 23,000 apps, and the model was compared with several ML algorithms. The detection accuracy was reported to be 98.60% for the DAE model with an FPR of 1.82% while it was 99.82% with an FPR of 0.21% for the CNN-S model. However, the detection time for the DAE-CNN-S and CNN-S models were reported as 59.6 and 332.9 min, respectively, making them impractical for real-world use. Since this was a hybrid method, a few features were extracted after the installation of apps on the devices [50].

Although these techniques conduct a thorough analysis of the apps to develop highly efficient malware detection systems, they tend to have high training and detection times, deeming them impractical in the real world. In addition, these techniques use *dated* datasets for model development and may fail to classify new malware as APIs and permissions are regularly added and removed. Therefore, we focus on resolving these issues to develop an efficient, adaptable, and implementable malware detection framework.

3. NATICUSdroid - An android malware detection framework

Building upon and to overcome the limitations of previous works, we propose NATICUSdroid by exploring the aspects that previous works may not have acknowledged to the best of our knowledge. We conducted an in-depth analysis of recent malicious and benign app datasets that we collected. We also analyzed the performance of ML classifiers that we used for building our detection model. The workflow for NATICUSdroid development is shown in Fig. 2.

3.1. App database

First, we collected a large number of Android apps from two sources. We utilized the apps available in the Androzoo project database to create a benign app dataset [51]. Androzoo has meta-data such as name, size, checksum, and VirusTotal ratings for more than 10 million Android apps available across 18 app markets, including Google Play Store. Although Play Store is known to have malware despite Play Protect scanning the Play Store apps, for simplicity, we assumed that the apps from Play Store exhibit a non-malicious behavior. In addition, to add an extra layer of assurance, we considered 15,000 apps of Play Store rated *benign* by VirusTotal. This database was further pruned by selecting apps with `targetSDKVersion`, i.e., API level 23 and above, resulting in 14,630 benign apps. To create the dataset for malware, we used Argus Lab's Android Malware Database (AMD) that contains more than 24,500 malware from 2010–2019, categorized in 135 varieties among 71 malware families [52]. We selected 14,700 random malware samples to keep the two datasets similar in size to avoid skewing the distribution of malware and benign app instances in the datasets.

3.2. Feature extraction and dataset generation

Most apps employ native permissions, usually declared as `android.permission.PERMISSION_NAME`. On the other hand, custom permissions are employed to utilize specialized hardware/software resources (such as web APIs) that are not defined in the AOS. These permissions are expected to assist in the functioning of the app and usually declared as `com.company.package.PERMISSION_NAME`.

Our analysis found that malware may use either native or native and custom permissions to access data or gain full control of the victim's device. Therefore, all such permissions should be considered in determining whether an app is benign or malicious. The permissions can be extracted from the app's manifest file, *AndroidManifest.xml*. It is possible to extract information from this file by traversing XML tree nodes and checking node tags to confirm that the node contains information for the components, which in our case are the declared permissions [53].

We created a permissions list by collecting (i) API level 28 native permissions [18], and (ii) custom permissions used by collected malicious and benign apps. This list had over 7000 different permissions, however, we observed that there were several native permissions which either do not exist anymore (such as `android.permission. GET_WIFI` and `android.permission.ACCESS_ALL_DOWNLOAD`), or had typographical errors (such as `android.permission.write_` and `android.permission.ACC_LOCATION`). Removing such permissions reduced the total number of permissions to 6761.

Further, we created two datasets by collecting the permissions vector for each app based on - (i) only native permissions (Native), and (ii) both native and custom permissions (Naticus). We created the Native dataset for comparison with the results of the Naticus dataset. Algorithm 1 describes the feature extraction and datasets generation process. We initialize two empty permissions vectors (all zeros) for each app where each vector element represents permission in the Native and Naticus permissions list. Next, all permissions extracted from an app are searched in these two vectors, and their occurrence is recorded as 1. Once the vectors for one app are created, they are appended to the respective dataset.

ALGORITHM 1: Pseudo code for dataset generation

```
Input:
App_DB: Application database
Native_Perms_List: List of native permissions
Naticus_Perms_List: List of native and custom permissions
Output:
native_dataset = [ ]
naticus_dataset = [ ]
for app in App_DB do
    perms_app = Androguard.get_all_permissions(app)

    native_perms_vector = [0, 0, ..., 0]
    naticus_perms_vector = [0, 0, ..., 0]

    for perm in perms_app do
        if perm in Native_Perms_List then
            ind = Native_Perms_List.index(perm)
            native_perms_vector[ind] = 1

        ind = Naticus_Perms_List.index(perm)
        naticus_perms_vector[ind] = 1

    native_dataset.append(native_perms_vector)
    naticus_dataset.append(naticus_perms_vector)
```

3.3. Feature selection

Feature selection is a necessary data pre-processing step when dealing with high-dimensional data. This step helps in creating models with low complexity while ensuring that the data is easily understandable. This, in turn, improves the performance of a model by eliminating noise in the dataset. In addition, it helps in preventing models from over-fitting and reduces the time and space complexity of data analysis methods, which enhances the model as a whole [54]. Hence, we performed feature selection in our work in three steps. First, we computed the frequency of each permission in all the apps of the dataset, i.e., how many apps have declared specific permission in their manifest files. This gave us an insight on which permissions are used more frequently than others, and could potentially work against the model. Second, we applied backward elimination to remove features that were not statistically significant in the classification process. Although this process reduces the number of required features and improves accuracy, it does not take care of the multicollinearity problem in the dataset that holds back the classification accuracy. Therefore, we performed a collinearity check by finding correlations among all possible pairs of features. After this check, we kept only one feature from the strongly correlated feature pairs. These steps are detailed in the following subsections.

3.3.1. Counting frequency of permissions

Computation of permission frequency is a common statistical analysis method used by researchers to analyze Android apps [38,55]. Direct frequency counting recognizes the widely used permissions. This step is robust in determining if certain permissions are used more frequently in benign or malicious apps. For our model, we extracted all

the permissions being used in the apps using *Androguard* and counted how many benign or malicious apps were using each permission. From the 6761 permissions, we found that certain permissions were used in more than 75% of the apps. On the other hand, a few custom permissions were used by only a single app. This step was helpful in determining which permissions to consider and which ones to ignore. Hence, after the analysis, we removed those permissions from the dataset that were being used by less than 100 apps. This step significantly reduced the number of permissions from our feature set, as thousands of permissions were removed during this process. It is noteworthy that most of these permissions were app-specific or custom, and were being used by less than 1% of the records in our dataset, which could have a potentially negative impact in the learning process. However, a permission with a high usage frequency may not necessarily be a relevant feature for the malware detection system. Hence, we utilized backward elimination in the feature selection process.

3.3.2. Backward elimination

Feature selection methods involve a search-and-select technique coupled with an evaluation measure for the different subsets of features. This evaluation measure divides feature selection into two broad categories: Filter methods and Wrapper methods [56,57]. Filter methods use information measure, consistency, or other measures to find the best subset of features [56]. These are generally computationally inexpensive and produce a general feature set for the model. Wrapper methods employ training of a classification algorithm to evaluate the effectiveness of several subsets of features. These methods are generally computationally expensive but provide classifier specific subsets of features [58]. Backward elimination, a filter method, focuses on considering a subset of the candidate features set [59]. It is an iterative process that starts with a full set of features, S. In each iteration, it removes a feature, or independent variable $s \in S$ with the highest p-value above the significance level (SL = 0.5), and whose exclusion maximizes the objective function. This process continues until there is a features set $D \subset S$, where all features have a p-value less than SL. In our detection model, we applied backward elimination to the set obtained from the initial permissions list after excluding the less frequent permissions. The backward elimination also removed several permissions and reduced the number of necessary and significant permissions.

3.3.3. Multicollinearity removal

High collinearity or multicollinearity can be observed between a pair of features or independent variables when they are highly correlated [60]. The value of the correlation lies between −1 and 1. The presence of highly collinear features could lead to issues such as instability in parameter estimation, counter-intuitive parameter signs in the model, high R^2 diagnostics despite few or no significant parameters, and high error in estimates of parameters [61–63]. Therefore, to eliminate collinearity from our data, we applied Pearson's correlation test by creating disjoint sets of highly correlated features using a correlation threshold value of ≥ 0.9. We considered only one feature from each pair of highly correlated features. Priority was given on the elimination of less dangerous permissions from the highly correlated feature sets. This step further helped in pruning the features and slightly brought down the number of features in our dataset, and improved the performance of the model.

3.4. Classification

The aspiration behind this step is to gauge and find the classifier that can identify apps as either malicious or benign with high confidence based on the permissions. In total, we experimented with three single classifiers and five ensemble classifiers for NATICUSdroid. Ensemble classifiers, such as Random Forests, are a combination of single learners intended to improve the classification performance. The combination can reduce the errors made by the individual classifiers on different regions of the input space. Therefore, the ensemble classifier's performance is likely to be better than single learner classifiers [64]. Their combination can reduce the variance of estimation errors and improve the overall classification accuracy [65]. We analyzed the performance of the classifiers for Native and Naticus datasets. Further, we evaluated the performance of the models using various metrics such as accuracy, f-score, ROC curves, training time, and detection time.

Table 1
Number of permissions in the datasets after each stage of feature selection/reduction.

Step	Naticus permissions	Native permissions
Feature extraction	6761	325
Permission frequency counting	86	52
Backward elimination	58	39
Collinearity check	55	39

The accuracy of a model is the percentage of correctly classified records in the dataset. Precision is the ratio of the number of true positives to the sum of true positives and false positives, while recall is the ratio of true positives to the sum of true positives and false negatives. F-score is the harmonic mean of precision and recall. False-positive rate (FPR) is also an important metric for evaluating a decision model. It is the ratio of false positives to the sum of false positives and true negatives classified by the model. When evaluating binary decision problems, we use the Receiver Operator Characteristic (ROC) curve that shows how the number of correctly classified positive records (TPR) on the y-axis, varies with the number of incorrectly classified negative records (FPR) on the x-axis [66]. The area under the curve (AUC) from ROC is used as a metric to determine how a classifier performs over the whole dataset [67,68].

Training time signifies the time taken by a classifier for building the model using the dataset. A model with the least training time is best-suited when it needs to be trained and updated regularly. Classification/detection time is the time required for the model to successfully classify or predict the class/category of a record. Models with low detection time are better as they can generate results faster and make the overall system reliable. Although detection time is tightly coupled with detection accuracy, specific scenarios require a trade-off between the two to choose an optimum model. Since we built a malware detection model that detects an app as benign or malicious, we will refer classification time as detection time. For this framework to be implemented in a real-world system, we would need a model with minimum training/detection time and high detection accuracy.

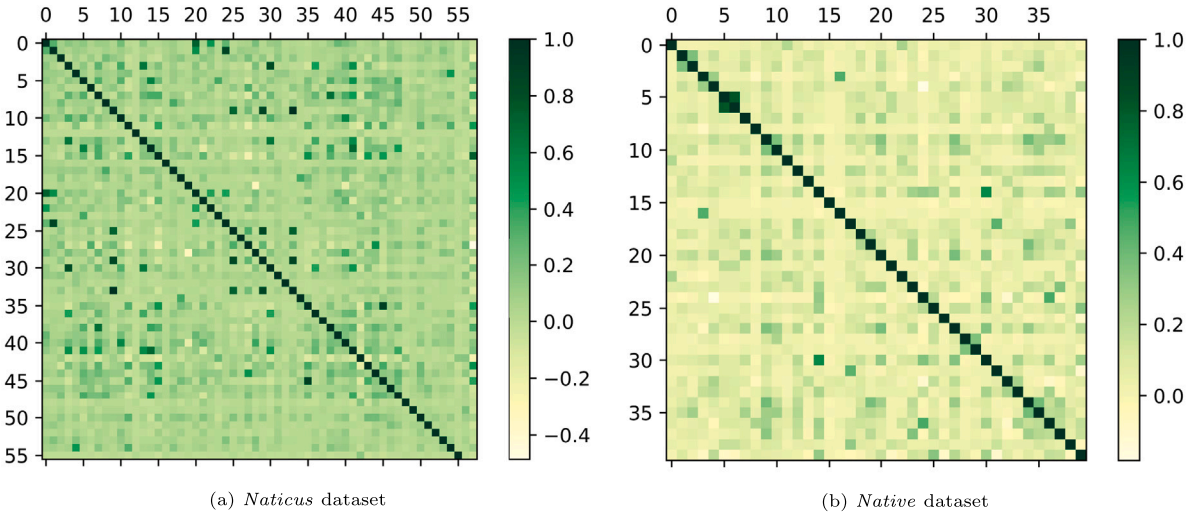
4. Experimental results and discussion

We performed a detailed analysis of the proposed scheme on the in-house collected Native and Naticus datasets. We used a machine running Windows 10 with Intel Core i7-8500 CPU and 16 GB of RAM. All the modules, methods, and experiments were developed using Python libraries. In terms of specific software, we used Androguard for feature extraction and Scikit-learn [69] for ML model development.

4.1. Feature selection

Here, we present results related to the feature selection stage that has been discussed in Section 3.2. Table 1 lists the number of permissions in the datasets after each stage of feature selection.

- **Permission Frequency Counting:** This analysis revealed that the apps in the dataset use 6761 different permissions. Interestingly, benign apps use as many as 6646 permissions (native and custom). Although most of the permissions were used in both benign and malicious apps, 159 permissions (native and custom) were specifically used in malicious apps. For reference, ten most frequently declared permissions in benign and malicious apps of both the datasets are listed in Table 2. It should



(a) *Naticus* dataset (b) *Native* dataset

Fig. 3. Correlation Heatmaps for the two datasets.

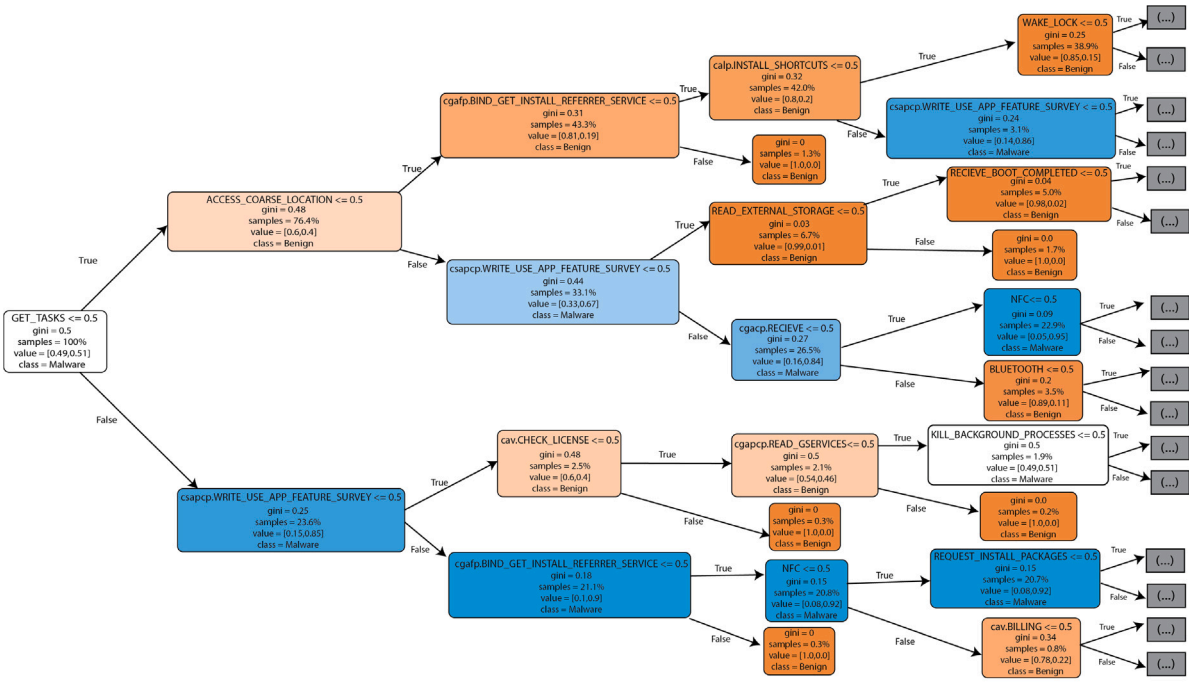
Table 2
Top 10 Permissions in Benign and Malicious Apps for Native and Naticus datasets (cgacp = com.google.android.c2dm.permission, calp = com.android.launcher.permission).

	Benign		Malware	
	Permissions	%	*Permissions*	%
NATIVE	INTERNET	98.10	INTERNET	96.62
	ACCESS_NETWORK_STATE	93.94	ACCESS_NETWORK_STATE	95.86
	WRITE_EXTERNAL_STORAGE	60.54	READ_PHONE_STATE	94.31
	WAKE_LOCK	50.66	WRITE_EXTERNAL_STORAGE	73.27
	ACCESS_WIFI_STATE	42.00	RECEIVE_BOOT_COMPLETED	70.27
	VIBRATE	32.12	ACCESS_COARSE_LOCATION	67.84
	READ_EXTERNAL_STORAGE	25.52	ACCESS_FINE_LOCATION	64.80
	ACCESS_COARSE_LOCATION	24.84	ACCESS_WIFI_STATE	62.97
	ACCESS_FINE_LOCATION	24.21	WAKE_LOCK	45.52
	READ_PHONE_STATE	23.00	WRITE_SETTINGS	20.95
NATICUS	INTERNET	98.10	INTERNET	96.62
	ACCESS_NETWORK_STATE	93.94	ACCESS_NETWORK_STATE	95.86
	WRITE_EXTERNAL_STORAGE	60.54	READ_PHONE_STATE	94.31
	WAKE_LOCK	50.66	WRITE_EXTERNAL_STORAGE	73.27
	ACCESS_WIFI_STATE	42.00	RECEIVE_BOOT_COMPLETED	70.27
	cgacp.RECEIVE	41.17	ACCESS_COARSE_LOCATION	67.84
	VIBRATE	32.12	ACCESS_FINE_LOCATION	64.80
	READ_EXTERNAL_STORAGE	25.52	ACCESS_WIFI_STATE	62.97
	ACCESS_COARSE_LOCATION	24.84	WAKE_LOCK	45.52
	ACCESS_FINE_LOCATION	24.21	calp.INSTALL_SHORTCUT	39.88

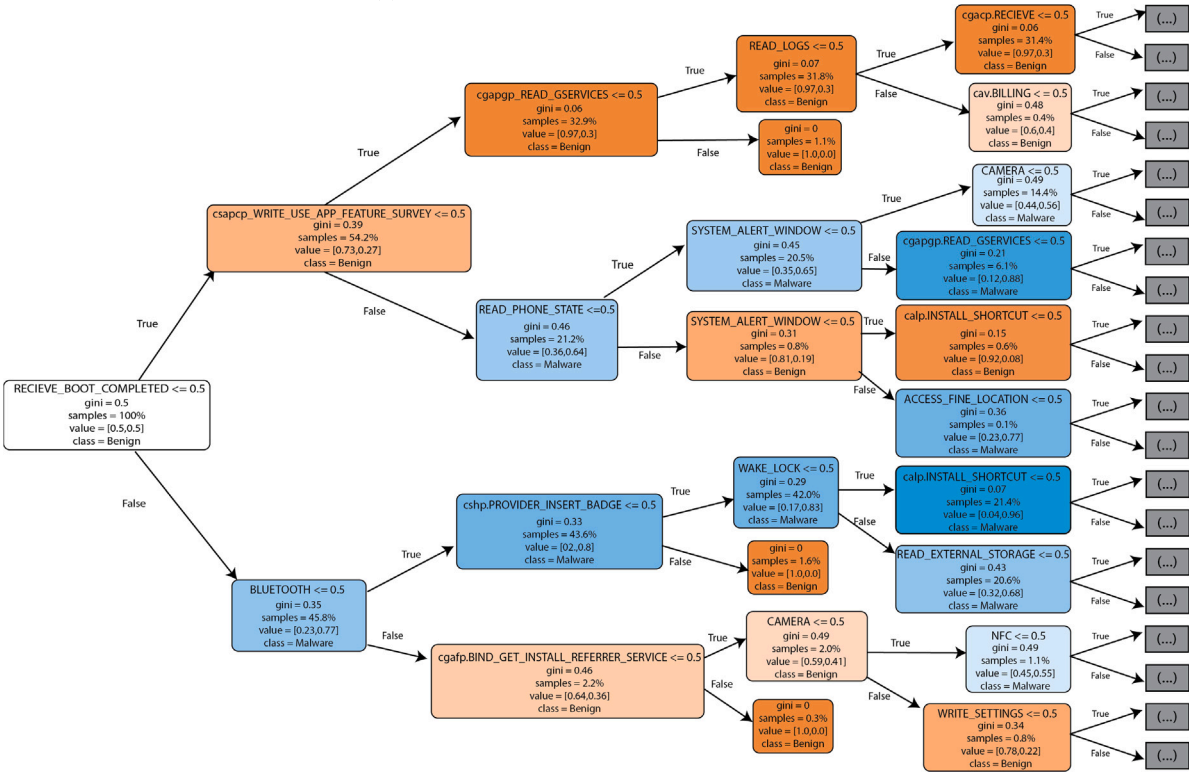
be noted that these permissions may or may not necessarily be included in the final features set of the ML model. The table just gives us an idea of the most frequently used permissions in the apps. While a few permissions were being used by more than 50% of the apps in our datasets (e.g., `INTERNET`, `ACCESS_NETWORK_STATE`), a few custom permissions were only used by a single app (e.g., `com.kiloo.subwaysurf.permission.C2D_MESSAGE`). The most used permissions were native permissions, but custom permissions were also used by a majority of benign apps. Malware, however, used more generic native permissions, with occasional use of custom permissions, such as `com.android.vending.BILLING`. These custom permissions were being used by as many as 3600 apps, of which 9.6% were malicious apps, to make in-app and out-of-app transactions using Google Pay [70]. Benign apps use these permissions to facilitate purchases inside the app. But several malware, disguised as benign, use them to bypass Google store's billing system and misuse the users' financial credentials. In conclusion, custom permissions are used more in benign apps than malware and play an important role in detection as they provide more information about benign apps and result in a more robust model.

Moreover, a few permissions were being used by both benign and malicious apps, e.g., `ACCESS_WIFI_STATE` that may be used to obtain the device MAC address to identify the device (and its user) uniquely, and `ACCESS_COARSE_LOCATION` allows access to the approximate location. Several malware request permissions such as `SEND_SMS` that fall under the *dangerous* protection level. It should be noted that a more extensive analysis was required for this step since a few permissions might appear harmless while they can be potentially exploited. However, many of them have been deprecated in previous API levels. For example, `GET_TASKS` refers to processes, not a user to-do list, and is often used for API calls such as `android.app.ActivityManager.getRunningTasks` and `android.app.ActivityManager.getRecentTasks`. These calls may be used in combination with `PackageManager()`, which will allow gathering app installation and usage statistics without any additional permissions.

Due to the above mentioned challenges and the vast differences in the occurrence frequencies of permissions, we selected the permissions which were used in more than 100 benign and malicious apps. This gave us 86 permissions for *Naticus* dataset and 52 permissions for *Native* dataset.



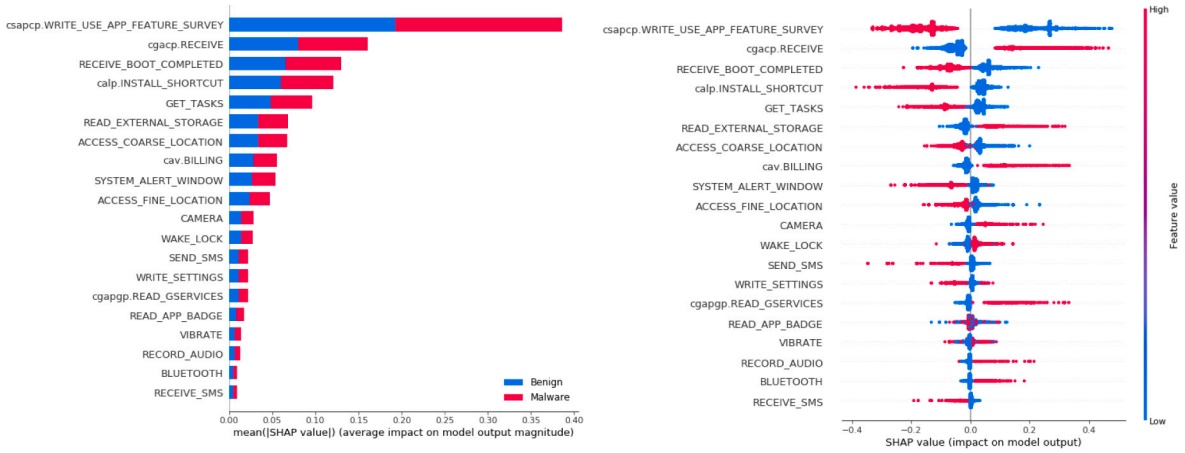
(a) Decision Tree 1



(b) Decision Tree 2

Fig. 4. The first five levels of two decision trees of the RF classifier.

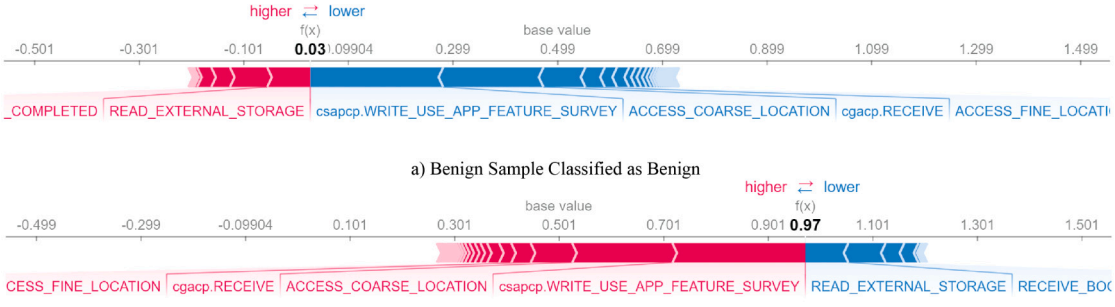
- **Backward Elimination:** After this step, a few permissions being used by almost 95% benign, and 61% malicious apps were removed as they were not statistically significant in the classification task. Examples of such permissions include `ACCESS_NETWORK_STATE` and `WRITE_EXTERNAL_STORAGE`. On the one hand, benign apps use `ACCESS_NETWORK_STATE` to access the connection status between the device and the network. On the other hand, malicious apps use it along with `CHANGE_NETWORK_STATE` to switch the Wi-Fi status without the user's consent. This allows them to contact their server whenever they desire and leak the user's data in the background without the user's consent. `WRITE_EXTERNAL_STORAGE` enables malware to download malicious files, packages, and/or payloads on the device and store them on external storage without the user's consent. Although this permission is used in several genres of benign apps, malware use it to download malicious packages



(a) Contribution of top 20 features during training

(b) Contribution of top 20 features during detection

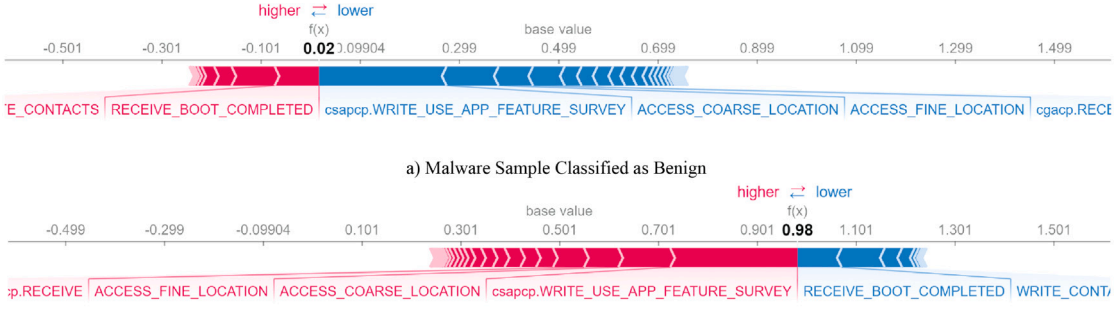
Fig. 5. Feature importance in train and test datasets.



a) Benign Sample Classified as Benign

b) Benign Sample Classified as Malware

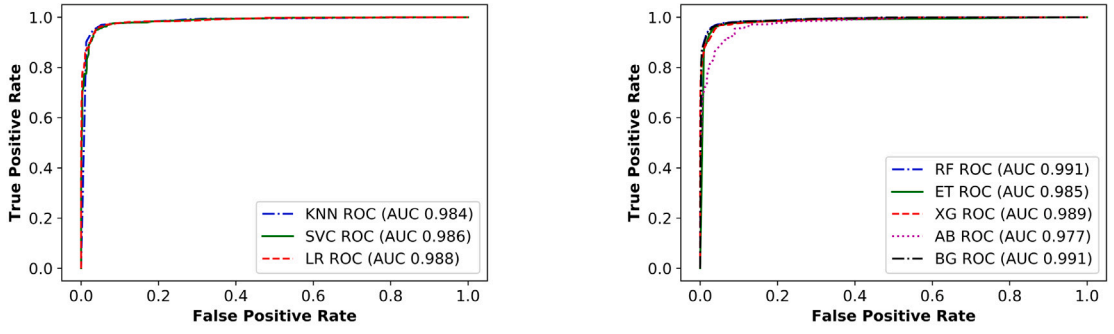
Fig. 6. Classification probability of Benign samples.



a) Malware Sample Classified as Benign

b) Malware Sample Classified as Malware

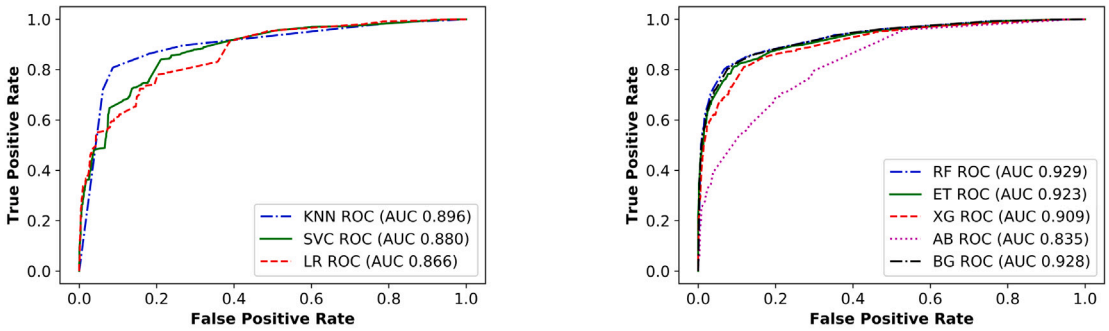
Fig. 7. Classification probability of malware samples.



(a) Single Learners

(b) Ensemble Learners

Fig. 8. ROC curves for the *Naticus* dataset.



(a) Single Learners (b) Ensemble Learners

Fig. 9. ROC curves for the *Native* dataset.

on the storage space. Permissions, such as `MOUNT_UNMOUNT_FILESYSTEMS` and `RESTART_PACKAGES`, which are frequently used in malicious apps, and sometimes also by benign apps, were also removed in this step as they had high R^2 values. Apart from these native permissions, several custom permissions were removed as well. This step reduced the number of features from 86 to 57 in the *Naticus* dataset and from 51 to 39 in the *Native* dataset.

- **Multicollinearity Removal:** Although backward elimination gives us the most significant features, it does not eliminate highly correlated features. If a pair of permissions had a correlation above the threshold of 0.9 after Pearson's correlation test, one of the permissions from the pair was removed, and the correlation was calculated again. This process was performed recursively until there was no permission pair above the correlation threshold. As an example, `android.permission.READ_APP_BADGE` was removed from the *Naticus* dataset as it had a correlation of 0.98 and 0.96 with `com.oppo.launcher.READ_SETTINGS` and `com.oppo.launcher.WRITE_SETTINGS`, respectively. Similarly, `com.oppo.launcher.READ_SETTINGS` was removed as it had a correlation of 0.99 with `com.oppo.launcher.WRITE_SETTINGS`. It was inferential to remove a less dangerous feature from many of these pairs. This brought down the number of features to 55 in the *Naticus* dataset. However, no highly correlated features were found in this step for the *Native* dataset and so there was no need for removing any features. The correlation heat-maps of the features for both the datasets, after this elimination, are presented in Figs. 3a and 3b. The final sets of features for both the datasets can be found in Appendix A in Table 5.

As a result of the feature selection process, 55 permissions were selected for the Naticus dataset (40 native, 15 custom) and 39 permissions for the Native dataset. We also made the following interesting observations for the used permissions:

- Few custom permissions were created by popular smartphone manufacturers and played significant roles in differentiating between benign and malicious apps. These custom permissions are used in malware developed for specific manufacturers' smartphones who may have a large user base. These permissions could also be a part of apps repackaged with malicious code.
- A few malware is known to create or remove shortcuts in user devices, and hence, frequently use `INSTALL_SHORTCUT` or `UNINSTALL_SHORTCUT` permissions. Such malware can hide in plain sight by uninstalling the shortcut of the app displayed in the launcher.
- `BILLING` permission turned out to be another significant permission that malware frequently used.

4.2. Classification

After feature selection, we evaluated various single and ensemble classifiers. Each classifier was trained with both the datasets independently. We split each dataset into training (to train classifier), cross-validation (to check over- and under-fitting), and test (for evaluation) sets. We used the split of 70% training (including 10% cross-validation) - 30% testing. Metrics such as validation accuracy, detection accuracy, f-measure, training time, and detection time for each classifier were recorded. After training, we performed k-fold cross-validation (k = 10) and calculated the validation accuracy for each classifier. Further, testing was performed to report the detection accuracy and detection time — total time taken by a model to detect all the records in the test set. These results are detailed in Table 3.

We found that the performance metrics for all classifiers were significantly better on *Naticus* dataset than *Native* dataset, showing evidence that using only native permissions may not be the most robust way of detecting malicious apps today. Random Forest (RF) performed the best on both datasets with a detection accuracy, f-score, and detection time of 96.95%, 0.9662, and 0.11 s, respectively on the *Naticus* dataset; and 85.98%, 0.8835, and 0.12 s, respectively on the *Native* dataset. The detection accuracy of AdaBoost (AB) was the worst on both datasets. Bagging (BG) was the second-best classifier after RF in terms of model accuracy and f-score, while it performed far worse in terms of training and detection times. On the other hand, Extra Trees (ET) classifier performance was comparable to RF on all metrics. Both ET and RF have the same detection time of 0.11 s. However, the average validation accuracy for RF was greater than that of ET, and hence, we selected RF as the classifier for our detection model. Among single learners, KN performed comparable to ensemble learners, with a detection accuracy of 93.13% on *Naticus* dataset, but takes 91 times more time in detection as it is a lazy learner.

Our dataset comprised of more than 29,000 samples with 55 features in *Naticus* dataset and 39 features in the *Native* dataset with binary data values (0 and 1). Since both the datasets are high-dimensional, binary-valued, and large, all tree-based algorithms worked better than any non-tree based algorithms [71]. The random forest (RF) classification algorithm creates several decision trees by randomly choosing a feature subset for each tree. N number of trees (estimators) are created using different subsets of features, and then a classification decision is made through majority voting. Each of these decision trees in the model has a distinct structure in terms of the root, child, and leaf feature nodes. Due to space constraints, the first five levels of two example decision trees for the RF classifier of NATICUSdroid are shown in Fig. 4.

Each tree selects a feature with the minimum 'gini' impurity for the root node from a random set of features, considering random data samples. The root node has two child nodes based on the selected feature's decisions at the root node. New nodes are further added as child nodes with minimum gini impurity until all the records are classified as 'Benign' or 'Malware' at the leaf nodes with zero gini

Table 3
Classification Results for Naticus and Native datasets (KN = k-Nearest Neighbor, SVM = Support Vector Machines, LR = Logistic Regression, RF = Random Forest, ET = Extra Trees, XG = XGBoost, AB = AdaBoosting, BG = Bagging).

Classifier	Validation accuracy (%)		Detection accuracy (%)		F-Score		Training time (s)		Detection time (s)	
	Naticus	*Native*	*Naticus*	*Native*	*Naticus*	*Native*	*Naticus*	*Native*	*Naticus*	*Native*
KN	96.13	84.85	96.13	84.65	0.9617	0.8742	0.75	0.98	10.91	5.04
SVM	95.31	80.79	95.32	81.06	0.9537	0.8518	34.31	165.96	1.46	6.30
LR	95.93	77.75	95.95	77.9	0.9598	0.8158	0.09	0.08	0.01	0.001
RF	**97.10**	**86.03**	**96.95**	**85.98**	**0.9662**	**0.8835**	**0.17**	**0.12**	**0.11**	**0.11**
ET	96.45	85.06	96.49	84.67	0.9650	0.8704	0.13	0.12	0.11	0.11
XG	96.02	82.85	96.17	82.95	0.9620	0.8635	0.68	0.69	0.02	0.01
AB	92.87	77.34	92.18	77.05	0.9225	0.8378	1.27	1.38	0.15	0.15
BG	96.49	85.81	96.58	85.84	0.9659	0.8817	24.35	14.09	2.34	1.68

Table 4
Comparison of our work with state-of-the-art (CP: Custom Permissions, FPR: False Positive Rate, DT: Detection Time).

Work	Dataset		CP	Accuracy	FPR	DT
	Year	*Apps*				
PMDS [42]	2010–12	2950	✗	92 - 94	1.52–3.93	✗
ApkAuditer [44]	2010–12	8762	✗	88	✗	✗
CFG based detection [46]	2017	20 693	✗	98.8	2.9–9.1	✗
System calls and LSTM based detection [47]	2010–17	7005	✗	93.4	9.3	1 s
Drebin [26]	2010–12	129 013	✗	93	1	0.75 s
Signature and Heuristic based detection [48]	2015	401	✗	85	6.45	85 s
NATICUSdroid	**2010-19**	**29 330**	✔	**96.95**	**3.32**	**0.11 s**

impurity. In Fig. 4a, the root node with feature `GET_TASKS` shows a gini impurity of 0.5, where all the randomly selected data samples for the corresponding decision tree were considered. For this decision tree, the root node has a larger proportion of 'Malware' samples (0.51) than 'Benign' samples (0.49). Therefore the node is marked with the 'Malware' class label. Similarly, the right child node with feature `ACCESS_COARSE_LOCATION` has a gini impurity of 0.48. It received 76.4% data samples after the root node's decision, the majority of which are 'Benign' (0.6). Therefore the node is marked as 'Benign.' At level 3, we see that the left child of the rightmost node of level 2 has zero gini impurity, and 1.3% of the samples are classified as 'Benign.' Two other example decision trees can be found in Figs. 10 and 11 in Appendix B. The other tree-based classifiers are different in terms of their data sampling methods, cut-point selection criteria, generalizing techniques, and branch pruning techniques. They either have lower accuracy and better training/detection time or comparable accuracy and more training/detection time. RF gave the best trade-off between accuracy and training/detection time in our case.

Whenever a model's performance is important, it is imperative to understand the model correctly as most non-linear models are hard to interpret. Such 'BlackBox' models can be understood with the help of eXplainable Artificial Intelligence (XAI), which provides explanations and factors for the model's decisions. To explain the working of our RF model, we used SHapely Additive exPlanations (SHAP) that provides both local and global explanations of the results through graphical visualizations [72]. Fig. 5a shows a global influence of each feature in learning about benign and malware apps. For readability and simplicity, only the top 20 features are shown in the figures. Although each feature's influence seems to be somewhat similar for both the classes, `csapcp.WRITE_USE_APP_FEATURE_SURVEY` was the most influential permission in distinguishing between benign and malware apps. Similarly, in Fig. 5b the same permission is promising in classifying malware and benign apps. A similar trend is seen for `cgacp.RECIEVE` during training and detection phases. This gives us an overall idea of which features are contributing to the classification process. A local explanation on the probability for a benign and malware samples being correctly or incorrectly classified are shown in Figs. 6 and 7. This also shows the impact the features have on a sample during the classification process. The figure shows that the features interact with each other, pushing the probability value right or left until it reaches a final value to classify the record as benign or malicious.

The ROC curves for both the datasets are presented in Figs. 8 and 9. For the *Naticus* dataset, we observe that the True Positive Rate is above 95%, and False Positive Rate is below 2%, and the area under the curve (AUC) is approaching towards 1 as shown in Fig. 8. RF and ET have the highest AUC of 0.991, while AB's AUC is the least (0.977). However, for *Native* dataset, the best value of AUC is for RF (0.929) and the least is for AB (0.835), as seen in Fig. 9.

In addition, we also compared our work with state-of-the-art, shown in Table 4. The table clearly indicates that the detection time of our framework is the lowest while showing a comparable accuracy and FPR with other recent works. It is noteworthy that our work has utilized recent malware datasets along with the inclusion of custom permissions created by the app developers.

5. Conclusion

An adaptive, light-weight, fast, and robust malware detection framework, *NATICUSdroid*, has been developed to detect Android malware spreading from third-party Android markets and phishing attacks. We analyzed both the native and custom permissions from extensive and recent collections of benign and malicious Android apps to develop NATICUSdroid. We found that most native Android permissions were used in apps, while several custom Android permissions were used quite frequently. Based on our analysis, we selected permissions that significantly contributed to detection and discarded extraneous and non-significant permissions from the dataset, which played a significant role in improving the performance of the detection model. After selecting the most significant permissions from the dataset, we trained eight ML classifiers. Among them, the Random Forest classifier performed the best with a detection accuracy of 96.95%, f-score of 0.9962, FPR of 3.32%, and detection time of 0.11 s. Along with this, we did a comparative study by testing our framework on native-only permissions and other state-of-the-art methods and found that native Android permissions alone are not sufficient for malware detection, and custom permissions play an important role today.

The proposed detection framework has the capability of being upgraded with time to detect new malware efficiently. The challenge is updating the permissions list through the feature extraction process with newer Android versions and malware. Android developers can easily integrate this framework as a core Android feature, which can be updated with each Android release. When integrated into the Android

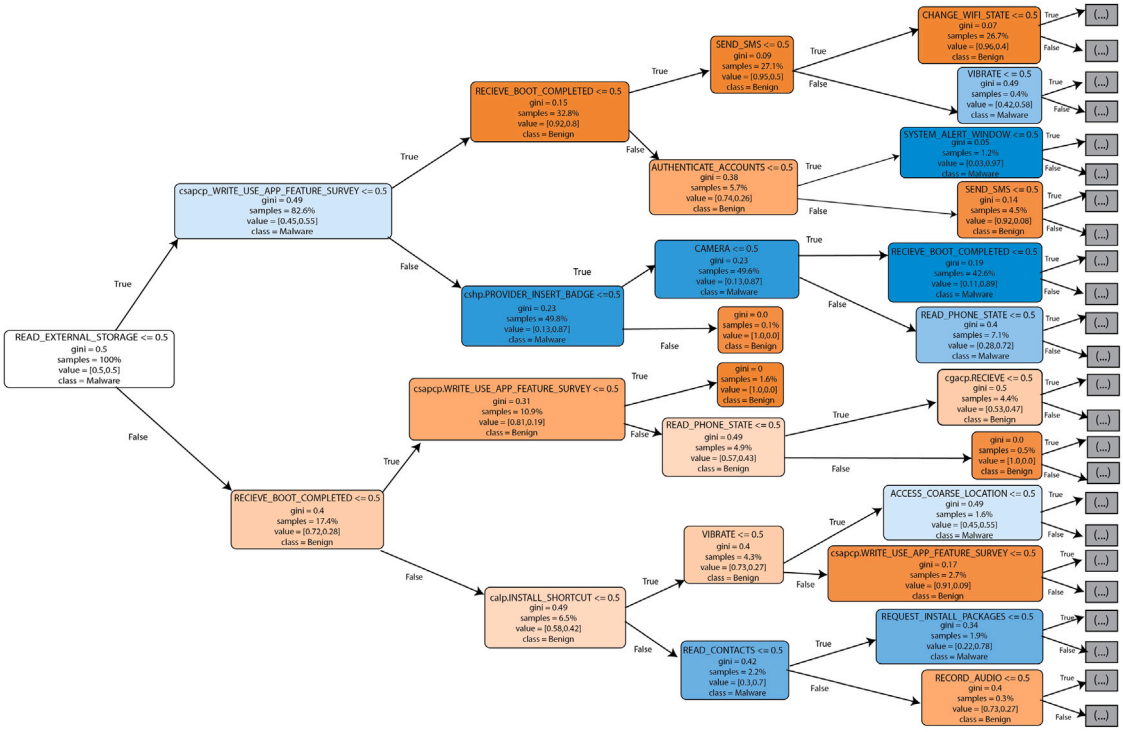


Fig. 10. First five levels of decision Tree 3 of the RF classifier.

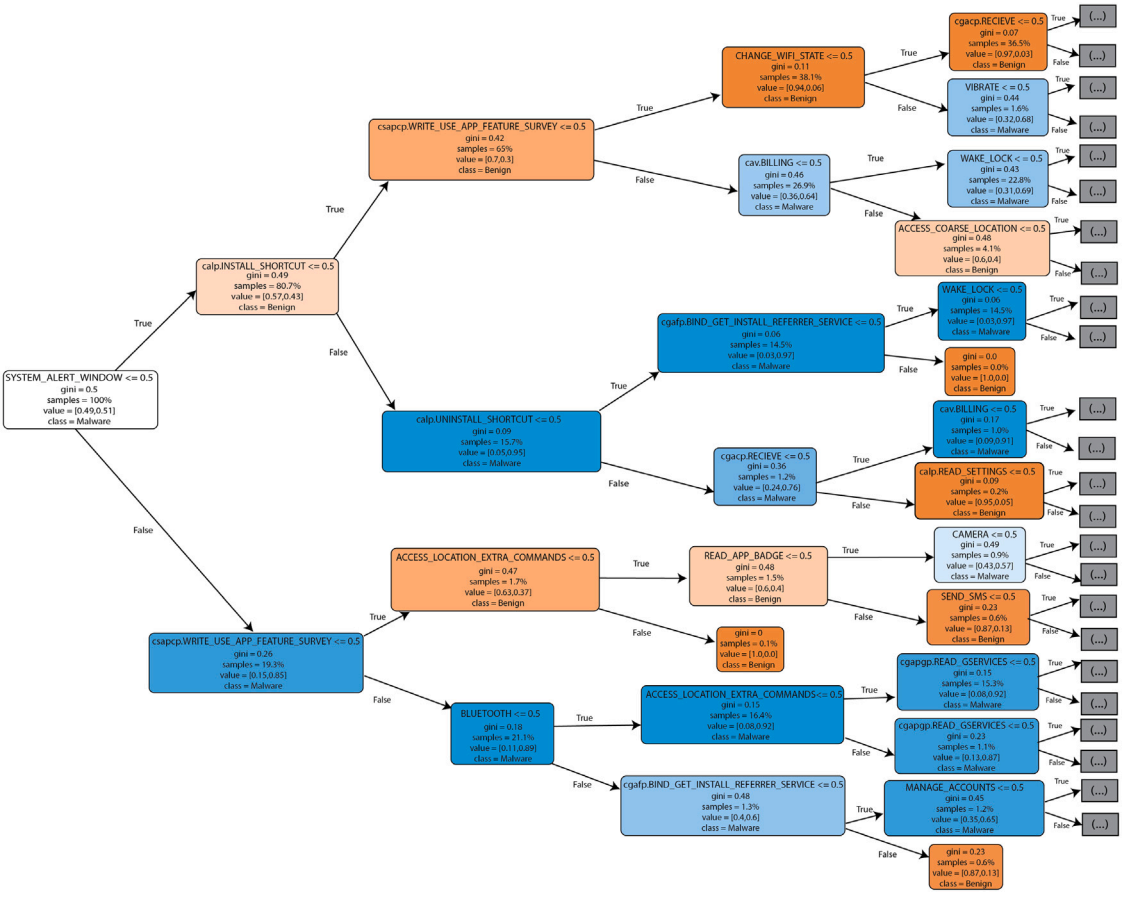


Fig. 11. First five levels of decision Tree 4 of the RF classifier.

Table 5
The final feature set for Native and Naticus datasets.

Naticus	Native
MANAGE_ACCOUNTS	ACCESS_DOWNLOAD_MANAGER
WRITE_SYNC_SETTINGS	ACCESS_LOCATION_EXTRA_COMMANDS
READ_EXTERNAL_STORAGE	ACCESS_MOCK_LOCATION
RECEIVE_SMS	ACCESS_NETWORK_STATE
calp.READ_SETTINGS	ACCESS_WIFI_STATE
WRITE_SETTINGS	BLUETOOTH
cgapgp.READ_GSERVICES	BLUETOOTH_ADMIN
GET_TASKS	BROADCAST_STICKY
RECORD_AUDIO	CHANGE_CONFIGURATION
CHANGE_NETWORK_STATE	CHANGE_NETWORK_STATE
calp.INSTALL_SHORTCUT	CLEAR_APP_CACHE
CALL_PHONE	DELETE_PACKAGES
WRITE_CONTACTS	DISABLE_KEYGUARD
READ_PHONE_STATE	FLASHLIGHT
csapcp.WRITE_USE_APP_FEATURE_SURVEY	GET_TASKS
MODIFY_AUDIO_SETTINGS	INTERACT_ACROSS_USERS_FULL
ACCESS_LOCATION_EXTRA_COMMANDS	INTERNET
INTERNET	MANAGE_ACCOUNTS
AUTHENTICATE_ACCOUNTS	MODIFY_AUDIO_SETTINGS
ACCESS_WIFI_STATE	MODIFY_PHONE_STATE
USE_CREDENTIALS	MOUNT_UNMOUNT_FILESYSTEMS
CHANGE_CONFIGURATION	NFC
READ_SYNC_SETTINGS	PACKAGE_USAGE_STATS
cdlp.UPDATE_COUNT	READ_LOGS
caap.SET_ALARM	READ_PROFILE
cgacp.RECEIVE	RECEIVE_BOOT_COMPLETED
KILL_BACKGROUND_PROCESSES	REQUEST_INSTALL_PACKAGES
cshp.PROVIDER_INSERT_BADGE	RESTART_PACKAGES
SEND_SMS	SET_WALLPAPER
REQUEST_INSTALL_PACKAGES	SET_WALLPAPER_HINTS
SET_WALLPAPER_HINTS	SYSTEM_ALERT_WINDOW
colp.WRITE_SETTINGS	USE_CREDENTIALS
ACCESS_MOCK_LOCATION	VIBRATE
ACCESS_COARSE_LOCATION	WAKE_LOCK
READ_LOGS	WRITE_APN_SETTINGS
cgagp.ACTIVITY_RECOGNITION	WRITE_SECURE_SETTINGS
SYSTEM_ALERT_WINDOW	WRITE_SETTINGS
DISABLE_KEYGUARD	WRITE_SMS
CHANGE_WIFI_STATE	WRITE_SYNC_SETTINGS
READ_CONTACTS	
cav.BILLING	
RECEIVE_BOOT_COMPLETED	
WAKE_LOCK	
ACCESS_FINE_LOCATION	
BLUETOOTH	
CAMERA	
cav.CHECK_LICENSE	
FOREGROUND_SERVICE	
VIBRATE	
NFC	
RECEIVE_USER_PRESENT	
CLEAR_APP_CACHE	
calp.UNINSTALL_SHORTCUT	
csaip.BILLING	
cgafp.BIND_GET_INSTALL_REFERRER_SERVICE	

[calp] com.android.launcher.permission
[cgapgp] com.google.android.providers.gsf.permission
[csapcp] com.samsung.android.providers.context.permission
[cdlp] com.anddoes.launcher.permission
[caap] com.android.alarm.permission
[cgacp] com.google.android.c2dm.permission
[cshp] com.sonymobile.home.permission
[colp] com.oppo.launcher.permission
[cgagp] com.google.android.gms.permission
[cav] com.android.vending
[csaip] com.sec.android.iap.permission
[cgafp] com.google.android.finsky.permission.

OS, this feature or app may be given permission to scan every app during installation and alert the user of suspicious behavior, if any. The development of such an app or Android-integrated feature could be a promising future work. Upon completion of the peer-review process, the native and custom permissions based dataset created by us will be released in the UC Irvine machine learning dataset repository to help the Android security research community engaged in the malware detection system.

CRediT authorship contribution statement

Akshay Mathur: Investigation, Methodology, Software, Formal analysis, Validation, Writing - original draft, Visualization. **Laxmi Mounika Podila:** Investigation, Validation. **Keyur Kulkarni:** Methodology, Writing - original draft. **Quamar Niyaz:** Conceptualization, Supervision, Writing - review & editing. **Ahmad Y. Javaid:** Conceptualization, Project administration, Resources, Supervision, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Final feature set

Table 5 lists the final feature set for both the *Naticus* and *Native* datasets, in the order in which they were considered during the selection process. This also implies that the first permission listed in the table had the lowest significance level, and the last permission had a significance level of 0.5 during the backward elimination process. Moreover, out of the 55 permissions in Naticus and 39 permissions in Native dataset, only 23 native permissions occur in both the datasets. This signifies that these permissions play a crucial role in the classification process, but presents the best result only when used with the other relevant native and/or custom permissions.

Appendix B. Other decision trees

See Figs. 10 and 11.

References

[1] Google. Wear os by google. Google Inc.; 2018, https://wearos.google.com/ (Accessed: 2020-04-24).

[2] Google. Android tv. Google Inc.; 2018, https://www.android.com/tv/ (Accessed: 2020-04-24).

[3] The Verge. There are now 2.5 billion active android devices. 2019, https://www.theverge.com/2019/5/7/18528297/google-io-2019-android-devices-play-store-total-number-statistic-keynote (Accessed: 2020-04-24).

[4] Statcounter - GlobalStats. Mobile operating system market share worldwide applications. Statcounter - GlobalStats; 2019, https://gs.statcounter.com/os-market-share/mobile/worldwide (Accessed: 2020-04-24).

[5] AppBrain. Top android os versions. 2020, https://www.appbrain.com/stats/top-android-sdk-versions (Accessed: 2020-04-24).

[6] Google. The google android security team's classifications for potentially harmful applications. Google Inc.; 2017, https://source.android.com/security/reports/Google_Android_Security_PHA_classifications.pdf (Accessed: 2020-04-24).

[7] Dent Steve. Sophisticated android malware tracks all your phone activities. engadget; 2018, https://www.engadget.com/2018/05/07/zoopark-android-malware-exfiltration/ (Accessed: 2020-04-24).

[8] Palmer Danny. Sophisticated android malware spies on smartphones users and runs up their phone bill too. ZDnet; 2018, https://www.zdnet.com/article/sophisticated-android-malware-spies-on-smartphones-users-and-runs-up-their-phone-bill-too/ (Accessed: 2020-04-24).

[9] O'Donnell Lindsey. Sophisticated reddrop malware targets android phones. Threat Post; 2018, https://threatpost.com/sophisticated-reddrop-malware-targets-android-phones/130170/ (Accessed: 2020-04-24).

[10] Goodin Dan. Found: New android malware with never-before-seen spying capabilities. Ars Technica; 2018, https://arstechnica.com/information-technology/2018/01/found-new-android-malware-with-never-before-seen-spying-capabilities/ (Accessed: 2020-04-24).

[11] Wang Yang, Zheng Jun, Sun Chen, Mukkamala Srinivas. Quantitative security risk assessment of android permissions and applications. In: IFIP annual conference on data and applications security and privacy. Springer; 2013, p. 226–41.

[12] Jiang Xuxian, Zhou Yajin. A survey of android malware. In: Android malware. Springer; 2013, p. 3–20.

[13] Google. Play protect. Google Inc.; 2018, https://www.android.com/play-protect/ (Accessed: 2020-04-24).

[14] Lardinois Frederic. Google says it removed 700K apps from the play store in 2017, up 70% from 2016. TechCrunch; 2018, https://techcrunch.com/2018/01/30/google-says-it-removed-700k-apps-from-the-play-store-in-2017-up-70-from-2016/ (Accessed: 2020-04-24).

[15] Smith Chris. Google just removed 145 Android apps that contained malware for Windows PCs. BGR; 2018, https://bgr.com/2018/08/06/android-malware-windows-google-removes-145-apps/ (Accessed: 2020-04-24).

[16] The Next Web. Google has removed 7 'stalkerware' apps from its play store. 2019, https://thenextweb.com/code-word/2019/07/18/google-removed-7-stalkerware-apps-from-its-play-store/ (Accessed: 2020-04-24).

[17] Android Central. Google pulls 85 adware-laden photography and gaming apps from play store. 2019, https://www.androidcentral.com/google-removes-85-adware-apps-were-installed-over-8-million-users (Accessed: 2020-04-24).

[18] GoogleDev. Android developers. 2019, https://developer.android.com/guide/topics/manifest/permission-element (Accessed: 2020-04-24).

[19] Feng Yu, Anand Saswat, Dillig Isil, Aiken Alex. Apposcopy: Semantics-based detection of android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM; 2014, p. 576–87.

[20] Feng Yu, Bastani Osbert, Martins Ruben, Dillig Isil, Anand Saswat. Automated synthesis of semantic malware signatures using maximum satisfiability. 2016, arXiv preprint arXiv:1608.06254.

[21] Zheng Min, Sun Mingshen, Lui John CS. Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware. In: 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications. IEEE; 2013, p. 163–71.

[22] Sanz Borja, Santos Igor, Laorden Carlos, Ugarte-Pedrero Xabier, Bringas Pablo Garcia, Álvarez Gonzalo. Puma: Permission usage to detect malware in android. In: International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions. Springer; 2013, p. 289–98.

[23] Wu Dong-Jie, Mao Ching-Hao, Wei Te-En, Lee Hahn-Ming, Wu Kuo-Ping. Droidmat: Android malware detection through manifest and api calls tracing. In: Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on. IEEE; 2012, p. 62–9.

[24] Aafer Yousra, Du Wenliang, Yin Heng. Droidapiminer: Mining api-level features for robust malware detection in android. In: International Conference on Security and Privacy in Communication Systems. Springer; 2013, p. 86–103.

[25] Mariconti Enrico, Onwuzurike Lucky, Andriotis Panagiotis, De Cristofaro Emiliano, Ross Gordon, Stringhini Gianluca. Mamadroid: Detecting android malware by building markov chains of behavioral models. 2016, arXiv preprint arXiv:1612.04433.

[26] Arp Daniel, Spreitzenbarth Michael, Hubner Malte, Gascon Hugo, Rieck Konrad, Siemens CERT. Drebin: Effective and explainable detection of android malware in your pocket. In: Ndss, Vol. 14. 2014, p. 23–6.

[27] Arzt Steven, Rasthofer Siegfried, Fritz Christian, Bodden Eric, Bartel Alexandre, Klein Jacques, Le Traon Yves, Octeau Damien, McDaniel Patrick. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Acm Sigplan notices, Vol. 49. ACM; 2014, p. 259–69.

[28] Shabtai Asaf, Kanonov Uri, Elovici Yuval, Glezer Chanan, Weiss Yael. "Andromaly": a behavioral malware detection framework for android devices. J Intell Inf Syst 2012;38(1):161–90.

[29] Saracino Andrea, Sgandurra Daniele, Dini Gianluca, Martinelli Fabio. Madam: Effective and efficient behavior-based android malware detection and prevention. IEEE Trans Dependable Secure Comput 2016;15(1):83–97.

[30] Chaba Sanya, Kumar Rahul, Pant Rohan, Dave Mayank. Malware detection approach for android systems using system call logs. 2017, arXiv preprint arXiv:1709.08805.

[31] Reina Alessandro, Fattori Aristide, Cavallaro Lorenzo. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. EuroSec 2013. April.

[32] Wu Chiachih, Zhou Yajin, Patel Kunal, Liang Zhenkai, Jiang Xuxian. Airbag: Boosting smartphone resistance to malware infection.. In: NDSS. 2014.

[33] Enck William, Gilbert Peter, Han Seungyeop, Tendulkar Vasant, Chun Byung-Gon, Cox Landon P, Jung Jaeyeon, McDaniel Patrick, Sheth Anmol N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans Comput Syst (TOCS) 2014;32(2):5.

[34] Rastogi Vaibhav, Chen Yan, Enck William. Appsplayground: automatic security analysis of smartphone applications. In: Proceedings of the third ACM conference on data and application security and privacy. ACM; 2013, p. 209–20.

[35] Boukhtouta Amine, Mokhov Serguei A, Lakhdari Nour-Eddine, Debbabi Mourad, Paquet Joey. Network malware classification comparison using DPI and flow packet headers. J Comput Virol Hack Techn 2016;12(2):69–100.

[36] Rahmat Safia, Niyaz Quamar, Mathur Akshay, Sun Weiqing, Javaid Ahmad Y. Network traffic-based hybrid malware detection for smartphone and traditional networked systems. In: 2019 IEEE 10th annual ubiquitous computing, electronics & mobile communication conference (UEMCON). IEEE; 2019, p. 0322–8.

[37] Felt Adrienne Porter, Chin Erika, Hanna Steve, Song Dawn, Wagner David. Android permissions demystified. In: Proceedings of the 18th ACM conference on computer and communications security. ACM; 2011, p. 627–38.

[38] Zhou Yajin, Jiang Xuxian. Dissecting android malware: Characterization and evolution. In: 2012 IEEE symposium on security and privacy. IEEE; 2012, p. 95–109.
[39] Feizollah Ali, Anuar Nor Badrul, Salleh Rosli, Suarez-Tangil Guillermo, Furnell Steven. Androdialysis: Analysis of android intent effectiveness in malware detection. Comput Secur 2017;65:121–34.
[40] Altaher Altyeb, BaRukab Omar. Android malware classification based on ANFIS with fuzzy c-means clustering using significant application permissions. Turk J Electr Eng Comput Sci 2017;25(3):2232–42.
[41] Arshad Saba, Shah Munam A, Wahid Abdul, Mehmood Amjad, Song Houbing, Yu Hongnian. Samadroid: a novel 3-level hybrid malware detection model for android operating system. IEEE Access 2018;6:4321–39.
[42] Rovelli Paolo, Vigfússon Ýmir. Pmds: Permission-based malware detection system. In: International conference on information systems security. Springer; 2014, p. 338–57.
[43] Mila. Contagio mobile dataset. 2012, http://contagiominidump.blogspot.com/.
[44] Talha Kabakus Abdullah, Alper Dogru Ibrahim, Aydin Cetin. Apk auditor: Permission-based android malware detection system. Digit Investig 2015;13:1–14.
[45] Mahindru Arvind, Singh Paramvir. Dynamic permissions based android malware detection using machine learning techniques. In: Proceedings of the 10th innovations in software engineering conference. ACM; 2017, p. 202–10.
[46] Ma Zhuo, Ge Haoran, Liu Yang, Zhao Meng, Ma Jianfeng. A combination method for android malware detection based on control flow graphs and machine learning algorithms. IEEE Access 2019;7:21235–45.
[47] Xiao Xi, Zhang Shaofeng, Mercaldo Francesco, Hu Guangwu, Sangaiah Arun Kumar. Android malware detection based on system call sequences and LSTM. Multimedia Tools Appl 2019;78(4):3979–99.
[48] Rehman Zahoor-Ur, Khan Sidra Nasim, Muhammad Khan, Lee Jong Weon, Lv Zhihan, Baik Sung Wook, Shah Peer Azmat, Awan Khalid, Mehmood Irfan. Machine learning-assisted signature and heuristic-based detection of malwares in android devices. Comput Electr Eng 2018;69:828–41.
[49] Pektaş Abdurrahman, Acarman Tankut. Ensemble machine learning approach for android malware classification using hybrid features. In: International conference on computer recognition systems. Springer; 2017, p. 191–200.
[50] Wang Wei, Zhao Mengxue, Wang Jigang. Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. J Ambient Intell Humaniz Comput 2019;10(8):3035–43.
[51] Li Li, Gao Jun, Hurier Médéric, Kong Pingfan, Bissyandé Tegawendé F, Bartel Alexandre, Klein Jacques, Traon Yves Le. Androzoo++: Collecting millions of android apps and their metadata for the research community. 2017, arXiv preprint arXiv:1709.05281.
[52] Wei Fengguo, Li Yuping, Roy Sankardas, Ou Xinming, Zhou Wu. Deep ground truth analysis of current android malware. In: International conference on detection of intrusions and malware, and vulnerability assessment. Springer; 2017, p. 252–76.
[53] Kim TaeGuen, Kang BooJoong, Rho Mina, Sezer Sakir, Im Eul Gyu. A multimodal deep learning method for android malware detection using various features. IEEE Trans Inf Forensics Secur 2019;14(3):773–88.
[54] Li Jundong, Cheng Kewei, Wang Suhang, Morstatter Fred, Trevino Robert P, Tang Jiliang, Liu Huan. Feature selection: A data perspective. ACM Comput Surv 2018;50(6):94.
[55] Sharma Mohit, Chawla Meenu, Gajrani Jyoti. A survey of android malware detection strategy and techniques. In: Proceedings of international conference on ICT for sustainable development. Springer; 2016, p. 39–51.
[56] Dash Manoranjan, Liu Huan. Feature selection for classification. Intell Data Anal 1997;1(1–4):131–56.
[57] Kohavi Ron, John George H. Wrappers for feature subset selection. Artif Intell 1997;97(1–2):273–324.
[58] Nguyen Hoai Bach, Xue Bing, Liu Ivy, Zhang Mengjie. Filter based backward elimination in wrapper based pso for feature selection in classification. In: 2014 IEEE congress on evolutionary computation (CEC). IEEE; 2014, p. 3111–8.
[59] Meyer Patrick, Marbach Daniel, Roy Sushmita, Kellis Manolis. Information-theoretic inference of gene networks using backward elimination.. In: BioComp. 2010, p. 700–5.
[60] Johnston Ron, Jones Kelvyn, Manley David. Confounding and collinearity in regression analysis: a cautionary tale and an alternative procedure, illustrated by studies of british voting behaviour. Qual Quant 2018;52(4):1957–76.
[61] Fotheringham AStewart, Oshan Taylor M. Geographically weighted regression and multicollinearity: dispelling the myth. J Geogr Syst 2016;18(4):303–29.
[62] O'brien Robert M. A caution regarding rules of thumb for variance inflation factors. Qual Quant 2007;41(5):673–90.
[63] Kuh Edwin, Welsch Roy E. Regression diagnostics: identifying influential data and sources of collinearity, Vol. 163. Wiley-Interscience; 1980.
[64] Tsai Chih-Fong, Hsiao Yu-Chieh. Combining multiple feature selection methods for stock prediction: Union, intersection, and multi-intersection approaches. Decis Support Syst 2010;50(1):258–69.
[65] Kim Myoung-Jong, Min Sung-Hwan, Han Ingoo. An evolutionary approach to the combination of multiple classifiers to predict a stock price index. Expert Syst Appl 2006;31(2):241–7.
[66] Davis Jesse, Goadrich Mark. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on machine learning*, 2006, p. 233–40.
[67] Davis Jesse, Burnside Elizabeth S, de Castro Dutra Inês, Page David, Ramakrishnan Raghu, Costa Vitor Santos, Shavlik Jude W. View learning for statistical relational learning: With an application to mammography. In: IJCAI. Citeseer; 2005, p. 677–83.
[68] Bradley Andrew P. The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern Recognit 1997;30(7):1145–59.
[69] Pedregosa Fabian, Varoquaux Gaël, Gramfort Alexandre, Michel Vincent, Thirion Bertrand, Grisel Olivier, Blondel Mathieu, Prettenhofer Peter, Weiss Ron, Dubourg Vincent, et al. Scikit-learn: Machine learning in python. J Mach Learn Res 2011;12(Oct):2825–30.
[70] GoogleDev. Android developers documentation. 2019, https://developer.android.com/google/play/billing (Accessed: 2020-04-24).
[71] Varghese Danny. Comparative study on classic machine learning algorithms. 2018, https://towardsdatascience.com/comparative-study-on-classic-machine-learning-algorithms-24f9ff6ab222.
[72] Lundberg Scott M, Lee Su-In. A unified approach to interpreting model predictions. In: Guyon I, Luxburg UV, Bengio S, Wallach H, Fergus R, Vishwanathan S, Garnett R, editors. Advances in neural information processing systems, Vol. 30. Curran Associates, Inc.; 2017, p. 4765–74, http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf.