applied sciences

MDPI

Article

Machine-Learning-Based Android Malware Family Classification Using Built-In and Custom Permissions

Minki Kim [1], Daehan Kim [1], Changha Hwang [2], Seongje Cho [3], Sangchul Han [4,*] and Minkyu Park [4]

[1] Department of Data and Knowledge Service Engineering, Dankook University, Yongin 16890, Korea; k_min_k24@dankook.ac.kr (M.K.); eogks0707@dankook.ac.kr (D.K.)
[2] Department of Statistics, Dankook University, Yongin 16890, Korea; chwang@dankook.ac.kr
[3] Department of Software Science, Dankook University, Yongin 16890, Korea; sjcho@dankook.ac.kr
[4] Department of Computer Engineering, Konkuk University, Chungju 27478, Korea; minkyup@kku.ac.kr
* Correspondence: schan@kku.ac.kr

Abstract: Malware family classification is grouping malware samples that have the same or similar characteristics into the same family. It plays a crucial role in understanding notable malicious patterns and recovering from malware infections. Although many machine learning approaches have been devised for this problem, there are still several open questions including, "Which features, classifiers, and evaluation metrics are better for malware familial classification"? In this paper, we propose a machine learning approach to Android malware family classification using built-in and custom permissions. Each Android app must declare proper permissions to access restricted resources or to perform restricted actions. Permission declaration is an efficient and obfuscation-resilient feature for malware analysis. We developed a malware family classification technique using permissions and conducted extensive experiments with several classifiers on a well-known dataset, DREBIN. We then evaluated the classifiers in terms of four metrics: macrolevel F1-score, accuracy, balanced accuracy (BAC), and the Matthews correlation coefficient (MCC). BAC and the MCC are known to be appropriate for evaluating imbalanced data classification. Our experimental results showed that: (i) custom permissions had a positive impact on classification performance; (ii) even when the same classifier and the same feature information were used, there was a difference up to 3.67% between accuracy and BAC; (iii) LightGBM and AdaBoost performed better than other classifiers we considered.

Keywords: Android malware; malware family classification; machine learning; built-in permission; custom permission; balanced accuracy; Matthews correlation coefficient

Citation: Kim, M.; Kim, D.; Hwang, C.; Cho, S.; Han, S.; Park, M. Machine-Learning-Based Android Malware Family Classification Using Built-In and Custom Permissions. *Appl. Sci.* **2021**, *11*, 10244. https://doi.org/10.3390/app112110244

Academic Editor: Arcangelo Castiglione

Received: 30 July 2021
Accepted: 26 October 2021
Published: 1 November 2021

1. Introduction

Android platforms are a major target of malware attacks. The statistics of Statista [1] showed that the total number of Android malware instances was 26.61 million on March 2018 and that 482,579 new Android malware samples were captured per month as of March 2020. G Data security experts announced that more than 1.3 million new malware samples were discovered every month in 2020 [2]. They also reported that, on average, new versions of malware were released every 1.3 s. Android malware analysts are overwhelmed by the large number of Android malware instances. To analyze malware instances efficiently and effectively, we need to classify them into malware families. Since the malware samples of a family have similar functionalities and share characteristics, malware family classification is crucial for understanding malware threat patterns and designing effective countermeasures against malware [3–5]. For example, if a malware sample is correctly categorized into a known family, we can easily identify any variants of the family, properly prescribe its solution, and instantly recover from the malware infection.

To tackle malware classification problems, many machine-learning-based techniques have been proposed. In machine-learning-based Android malware family classification,

an important issue is which features are selected and extracted [3,6–10]. There are static and dynamic features [3,6]. Static features can be obtained without executing malware. Static features include package name, app size, app component, permissions, application programming interface (API) calls, intent information, operation code, control flow graph, call graph, strings, etc. Dynamic features can be obtained by executing malware. Dynamic features include system calls, network traffic, resource consumption, SMS events, phone events, system logs, I/O operations, etc. Among those various features, permission information is the most widely used and effective feature [3,6–9]. The reasons are as follows. First, permission requests can be statically and easily extracted from the `Androidmanifest.xml` in an Android application package (APK) file [6,11–13]. Permission-based malware analysis needs to examine only the manifest file in each APK and does not need to parse/decompile/decrypt the `Dalvik` executable (DEX) file. Second, permission request are essential for Android malware to accomplish their malicious purposes. Android system controls each access to its resources with the associated permissions. When malware implements malicious behaviors by invoking some API calls, they require specific permissions related to the API calls. Third, the permission information is hard to deform by code obfuscation, which is often used by malware to evade analysis [10].

Another crucial issue is which metrics are useful for evaluating malware classification techniques. Malware family classification problems suffer from imbalanced datasets where the distribution of malware samples is unequal [14–16]. Different malware families normally contain different numbers of malware samples. In imbalanced datasets, a classifier tends to concentrate on correctly classifying large families while ignoring small ones. For a skewed class distribution, choosing proper metrics for imbalanced classification is challenging. The well-known metrics such as accuracy, F1-score, and receiver operating characteristic (ROC) may not be inappropriate especially when we are interested in minority sets [17,18]. Some studies [17–22] addressed this issue and proposed alternative metrics such as the area under the precision–recall curve (AUPRC), balanced accuracy (BAC), and the Matthews correlation coefficient (MCC).

This article presents a machine-learning-based Android malware family classification. Using only permission-based features, our method is lightweight and fast, but gives performance comparable to other methods that exploit multiple features. We carried out extensive experiments with several classifiers and a well-known dataset, DREBIN [23], to determine which classifier achieves better performance. The classifiers considered in this paper were random forest (RF) [24], artificial neural network (ANN) [25], deep neural network (DNN) [26], extremely randomized decision trees (Extra Trees) [27], adaptive boosting (AdaBoost) [28], XGBoost [29], and LightGBM [30]. We evaluated our Android malware classification method using the following metrics: accuracy (*ACC*) [18], F1-score [21], BAC [20], and the MCC [22]. BAC and the MCC are the proper metrics to evaluate imbalanced data classification. However, there is very little literature that has adopted BAC and the MCC in evaluating Android malware classification. We extracted both built-in permissions and custom permissions that are effective in malware family classification, analyzed the effect of the permissions in classifying DREBIN malware samples, and evaluated the classification models in terms of the four metrics.

The contributions of our work are as follows:

- We extracted built-in and custom permissions from malicious apps statically and used them as the feature for classifiers. The permissions were simply obtained from the `Manifest.xml` in each APK. Therefore, there was no need to reverse engineer, decrypt, or execute the Dalvik executable (DEX) file. In addition, our approach is resilient to code obfuscation and requires little domain knowledge;
- We applied seven machine learning classifiers for Android malware familial classification and compared their performance. The classifiers were ANN, DNN, random forest, Extra Trees, AdaBoost, XGBoost, and LightGBM;
- We evaluated the classification models with the following four metrics: ACC, macrolevel F1-score, BAC, and the MCC. The latter two metrics, BAC and the MCC, are suitable

metrics for the malware familial classification model whose datasets are imbalanced. When experimenting with 64 permissions (56 built-in + 8 custom permissions), the LightGBM classifier achieved 0.9173 (F1-score), 0.9512 (ACC), 0.9198 (BAC), and 0.9453 (MCC) on average;
- We inspected which Android permissions were primarily requested by a particular family or used only by a specific family. This can identify the permissions with which we can efficiently cluster malware instances into their families. For example, the three permissions ACCESS_SURFACE_FLINGER, BACKUP, and BIND_APPWIDGET are requested only by the `Plankton` malware family, and the two permissions BROADCAST_PACKAGE_REMOVED and WRITE_CALENDAR are requested only by the `Adrd` family;
- We considered all ninety-six permissions including nine custom ones and, then, considered eighty=seven built-in permissions alone, excluding custom ones. We then analyzed the impact of custom permissions on malware family classification.

The rest of this paper is organized as follows. Section 2 explains the Android security model based on permissions and describes the evaluation metrics for machine learning classifiers. Section 3 presents the schematic view of our approach and introduces the seven classifiers. In Sections 4 and 5, we present the setting for our experiments and evaluate the experimental results, respectively. Section 6 reviews related work, and Section 7 gives a discussion. Finally, Section 8 gives the conclusions.

2. Background

2.1. Android Permission Model

The APK file is an archive that contains all the contents of an Android app. An APK includes three files and four folders, as shown in Figure 1. In this article, our discussion is confined to two main files: `classes.dex` and `AndroidManifest.xml`. `Classes.dex` contains binary codes called the Dalvik bytecode that implements the app's functionality. The manifest file, `AndroidManifest.xml`, contains a set of information about the app: package name, version number, entry points, permissions, etc.

Android employs a permission-based security mechanism to restrict apps from accessing the resources of systems [6,11,31]. If an app tries to access hardware or software resources, it has to request the corresponding permissions through the `AndroidManifest.xml` file. The permissions are associated with an app's behaviors and capabilities with API calls. The permission mechanism is a key in the security model of Android. For instance, if an Android app needs to access the security-critical or privacy-sensitive methods of the Android application framework, the app must hold the corresponding permissions [32].

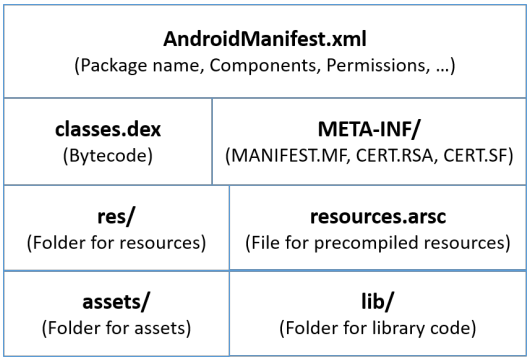

Figure 1. Structure of APK files.

There are two kinds of Android permissions: *built-in* and *custom* [13,33–36]. A built-in permission, also called a *system permission* or *native permission*, is a predefined one introduced by the native Android Operating System (OS), and thus is available by default on Android. An Android device contains many built-in permissions for its functionality such as monitoring incoming MMS message, accessing the camera device, connecting to

paired Bluetooth devices, broadcasting a notification, enabling/disabling location update notifications, etc. The constant values of the built-in permissions begin with the prefix `android.permission`. Examples of built-in permissions are `android.permission.BROADCAST_PACKAGE_REMOVED`, `android.permission.CONTROL_LOCATION_UPDATES`, etc. The number of built-in permissions is continuously increasing. Android API Level 15 supports 166 permissions, and API Level 28 supports 325 permissions [13]. As more permissions are available, there are more ways to exploit them.

A custom permission is defined by the developers of third-party apps to regulate access to their app-specific components by other apps [33–35]. Examples of custom permissions include `com.android.launcher.permission.UNINSTALL_SHORTCUT` and `com.google.android.googleapps.permission.GOOGLE_AUTH` [12]. The permission `com.android.launcher.permission.UNINSTALL_SHORTCUT` allows the app to delete home screen shortcuts without user involvement, but is no longer supported. The permission `com.google.android.googleapps.permission.GOOGLE_AUTH` allows apps to sign into Android services through the account saved on the device. All custom permissions defined by app developers are under the *dangerous* protection level [13].

Since the number of built-in permissions is increasing and app developers can commonly define more custom permissions in Android, malware has plenty of chances to gain control over sensitive data and devices. Thus, in analyzing recent Android malware samples, it is very important to consider newly introduced permissions. In addition, built-in and custom permissions are poorly separated, and they are treated the same by Android. This might allow malicious apps to obtain unauthorized access to system resources through custom permissions. The combination of multiple permissions may reflect some harmful behaviors. Requesting specific permissions is essential for Android malware to achieve its goals. Therefore, the permissions requested by apps can play a crucial role in malware analysis [10]. As a result, Android permissions are very popular features and the most used in many Android malware research works [6]. Moreover, they cannot be easily obfuscated by malware writers because scrambling them can destroy the Android programming model [37].

Another issue is the *runtime permissions*. Users can grant or revoke permissions at any time by changing the system settings since Android 6.0 (API Level 23). To access restricted data or perform restricted actions, Android apps need to check and request appropriate permissions at runtime. Users can approve or deny the runtime permission requests. This security enhancement makes permission handling nontrivial, resulting in various issues. Refs. [38,39] addressed these issues. Fortunately, the runtime permission issue affects the performance of our method very little because every Android app should declare at install time all permission requests (including runtime permissions) in `AndroidManifest.xml`, from which our method extracts permission request information. In some cases, an app may declare more permission requests in `AndroidManifest.xml` than it actually requests at runtime in the code [37,40]. This can be serious if a malicious app intentionally has dummy permissions to deceive a classifier into misclassifying it. However, malware rarely requests dummy permissions since apps requesting too many install-time permissions are likely to be denied by users.

2.2. Evaluation Metrics for Imbalanced Data Classification

Literature analysis suggests that most malware family classification studies work with imbalanced datasets. Their main performance evaluation metrics are precision, recall, ACC, F1-score, ROC, etc. [16–18,41]. In binary classification, ACC and F1-score calculated on confusion matrices have been the most popular metrics. A confusion matrix is a useful analytical tool to obtain the detail about what happened in the evaluation test and is the basis for calculating other performance measurements [42]. However, the two metrics can show excessively optimistic inflated results, particularly on imbalanced datasets [43]. It is also known that the ROC is inappropriate especially when we are interested in minority sets [17,18]. The interpretability of ROC plots from the viewpoint of imbalanced

datasets can be misleading with respect to conclusions about the credibility of classification performance, due to an intuitional, but incorrect interpretation of specificity [18].

Balanced accuracy (BAC) has some notional merits over the conventional accuracy while maintaining its simplicity. Accuracy treats each data point equally. Therefore, it has the effect of assigning different weights for each class. In the case of an imbalanced dataset, the overall performance can be distorted by classes with a large number of data points. Raff and Nicholas [15] used balanced accuracy as the metric. They said that BAC reweights the class with respect to the number of data points in each class and emphasizes the importance of learning low-frequency classes. The definition of BAC is given in Section 4.3.

The Matthews correlation coefficient (MCC) is said to explain the confusion matrix better than ACC, F1-score, and BAC [43]. The MCC is a balanced measure that yields a high value if the prediction achieves good results in all categories (true positives, false negatives, true negatives, and false positives). The definition of the MCC is shown in Section 4.3. If the MCC is one, this indicates that a classifier always forecasts correctly. A value of -1 indicates that a classifier always forecasts incorrectly. If the MCC is zero, it has the equivalent predictive ability as random prediction.

3. Malware Family Classification

Figure 2 illustrates the overview of our malware family classification. Our scheme extracts the permission requests from APK files using the Android asset packaging tool (AAPT) [8,44]. The extracted permissions are preprocessed and learned. Assuming that different malware families request different permissions, we trained seven machine learning classifiers with the permissions. We implemented our scheme using the Scikit-Learn library [45]. Using five-fold cross-validation, the classification performance was evaluated in terms of six metrics.

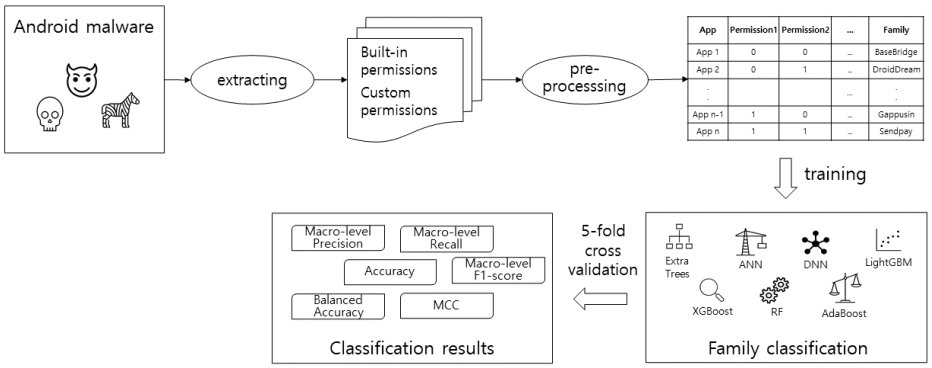

Figure 2. The schematic view of our malware family classification.

The seven machine learning classifiers we used in our experiments were ANN [25], DNN [26], random forest [24], Extra Trees [27], AdaBoost [28], XGBoost [29], and LightGBM [30]. All input variables used in our work have a binary value. Decision trees and their ensemble models generally work well with binary input data because input variables need not be binarized at all. Furthermore, decision trees and their ensemble models are also very effective at learning from unbalanced data, and in particular, ensemble models are popular due to their strong predictive performance [46–48]. This is the reason why we selected ensemble tree models.

ANN [25] is a machine learning algorithm created by mimicking the structure of a human neural network. ANN is composed of an input layer that receives multiple input data, an output layer in charge of outputting data, and a hidden layer that exists between them. A model is constructed by determining the number of nodes in the hidden layer. An activation function is used to find the optimal weight and bias. If there are many hidden layers, the accuracy of prediction increases, but the amount of computation increases exponentially. The disadvantages of ANN are: it is difficult to find the optimal parameter

values in the learning stage; there is a high possibility of overfitting; the learning time is relatively long.

DNN [26] improves the prediction by increasing the hidden layers in the model. DNN refers to a neural network structure with two or more hidden layers and is mainly used for iterative learning with many data. An error backpropagation technique has been devised and is widely used. CNN, RNN, LSTM, and GRU are representative algorithms.

The general idea of an ensemble algorithm is to combine several weak learners into a stronger one. A weak learner is a learning algorithm that only forecasts somewhat better than randomly.

Random forest [24] is an ensemble algorithm that learns using several decision trees. Random forest assigns input data sampled with replacement to a number of decision trees for training, collects the decision results of a target app, and determines the family with the most votes. As the tree grows, splitting is determined by considering only a subset of all characteristics at each node. The algorithm is simple and fast and does not cause overfitting. In general, it shows better performance than when using one good classifier.

Extremely randomized trees (Extra Trees) [27] randomly determines the threshold for each feature to split into subsets, while random forest searches for an optimal threshold. The learning time is significantly shorter than the random forest algorithm because finding the optimal threshold for each feature at all nodes is the most time-consuming task. As random forests, individual decision trees show some bias error, but overall, the bias errors and variation errors are reduced.

Adaptive boosting (AdaBoost) [28] is a general-purpose boosting algorithm. AdaBoost assigns equal weight to all instances when the first classifier learns. It adjusts the weight of instances for the next classifier according to the errors of the previous classifier: an increased weight for misclassified instances and a decreased weight for correctly classified ones. This modification makes the next classifier focus on the misclassified instance at the previous stage. AdaBoost repeats this recalculation until a desired number of classifiers are added. When all classifiers make a decision, AdaBoost makes a final decision by weighted voting.

EXtreme gradient boosting (XGBoost) [29] is an efficient implementation of the gradient boosting framework. XGBoost is also known as the normalized version of GBM, and normalization can prevent overfitting. The algorithm allows parallel processing on many CPU cores. The user can perform cross-validation at each iteration of the boosting process.

Light gradient boosting (LightGBM) [30] is a very efficient gradient boosting decision tree algorithm. It is similar to XGBoost and differs in how it builds a tree. LightGBM can reduce the learning time and memory usage by replacing continuous values with discrete bins. This algorithm can reduce the cost of calculating the gain for each partition. This algorithm supports not only GPU learning, but also parallel learning. LightGBM processes large-scale data more accurately.

4. Experiment Setup

We constructed a database of permission information extracted from malware of the DREBIN project [23]. To build a model for multiclass classification, a series of preprocessing was performed on this extracted data. Then, we trained the above-mentioned machine learning algorithms to identify the malware family.

4.1. Dataset

The DREBIN dataset contains 5560 malware samples from 179 different families. We dropped 61 samples that had no manifest file or structural errors; then, we used the remaining 5499 samples for the experiment. We ignored permissions that were not used in any malware samples. Then, we used the remaining 96 permissions (87 built-in permissions + 9 custom ones) as the features. The nine custom permissions are as follows:

com.android.alarm.permission.SET_ALARM,
com.android.browser.permission.READ_HISTORY_BOOKMARKS,
com.android.browser.permission.WRITE_HISTORY_BOOKMARKS,
com.android.launcher.permission.INSTALL_SHORTCUT,
com.android.launcher.permission.UNINSTALL_SHORTCUT,
com.android.vending.BILLING,
com.android.vending.CHECK_LICENSE,
com.google.android.c2dm.permission.RECEIVE,
com.google.android.googleapps.permission.GOOGLE_AUTH.

Since the malware of each family share codes, they also request similar permissions. Malware tends to belong to only one family, and some families have few samples to learn through machine learning. We sorted the families in descending order of the number of samples. We experimented with 4615 malware of the top 20 families. The family name, the number of samples, and permissions are shown in Table 1.

Table 1. The number of samples and permissions of the top 20 families.

Family	Rank	# of Apps	# of Permissions	
			Built-In	Custom
Fakeinstaller	1	920	42	1
DroidKungFu	2	660	59	5
Plankton	3	621	64	6
Opfake	4	601	35	2
GinMaster	5	336	43	5
BaseBRIDGE	6	327	36	1
Iconosys	7	152	21	0
Kmin	8	145	24	2
FakeDoc	9	132	36	4
Adrd	10	91	50	5
Geinimi	11	89	29	5
DroidDream	12	81	50	2
MobileTx	13	69	8	0
ExplitLinuxLotoor	14	66	48	4
Glodream	14	66	29	6
FakeRun	16	61	17	4
Gappusin	17	58	31	1
Sendpay	17	58	14	0
Imlog	19	43	8	0
SMSreg	20	39	25	5
		4615	669	58

We evaluated the algorithms using 5-fold cross-validation. We split the dataset into 5 folds. To proportionally distribute the samples of each family on 5 folds, we divided each family into 5 subsets and associated each subset with a fold.

4.2. Parameter Tuning

The parameters of the machine learning algorithms are shown in Table 2. The employed activation function of ANN and DNN was `ReLU`; the output function was `Softmax`; the optimization function was `Adam`.

Table 2. Parameters for machine learning algorithms.

	ANN	DNN		Random Forest	Extra Trees	Ada Boost	XGBoost	Light GBM
Hidden layers	0	10	**N estimators**	100	90	90	100	100
Epochs	100	100	**Depth**			100		
Batch size	128	128	**Max depth**	90	40			

4.3. Performance Metrics

A confusion matrix shows how many instances in each actual class are classified into each (predicted) class. An element m_{ij} of a confusion matrix M is the number of instances in actual class i that is predicted as class j. We can represent the confusion matrix as follows:

$$M_{K\times K} = [m_{ij}] \quad (1 \le i \le K, 1 \le j \le K), \tag{1}$$

where K is the total number of classes. With respect to class k, an instance is said to be positive if its predicted class is k and negative otherwise. Each instance belongs to one of the following categories with respect to class k:

- True positive: The actual class is k, and the predicted class is also k. The number of true positive instances is $TP_k = m_{kk}$;
- False positive: The actual class is not k, but the predicted class is k. The number of false positive instances is $FP_k = \left(\sum_{i=1}^{K} m_{ik}\right) - m_{kk}$;
- False negative: The actual class is k, but the predicted class is not k. The number of false negative instances is $FN_k = \left(\sum_{j=1}^{K} m_{kj}\right) - m_{kk}$;
- True negative: The actual class is not k, and the predicted class is also not k. The number of true negative instances is $TN_k = \left(\sum_{i=1}^{K}\sum_{j=1}^{K} m_{ij}\right) - (TP_k + FP_k + FN_k)$.

We can calculate the precision and recall of class k using Equation (2). Then, accuracy (ACC) is defined as the sum of true positives divided by the total number of instances. We also calculate balanced accuracy (BAC) using Equation (4) [20,21,49].

$$Precision_k = \frac{TP_k}{TP_k + FP_k}, \quad Recall_k = \frac{TP_k}{TP_k + FN_k}. \tag{2}$$

$$Accuracy = \frac{\sum_{k=1}^{K} TP_k}{\sum_{i=1}^{K}\sum_{j=1}^{K} m_{ij}}. \tag{3}$$

$$BAC = \frac{1}{K}\sum_{k=1}^{K}\frac{TP_k}{TP_k + FN_k}. \tag{4}$$

Micro- and macro-averages are two ways of interpreting confusion matrices in multi-class classification. The micro-average represents the performance of a model by focusing on each sample. On the other hand, the macro-average shows the model's performance by focusing on each class. The macro-average is therefore more suitable for data with an imbalanced class distribution. The macro-averages of precision, recall, and F1-score are calculated as follows:

$$Precision_{avg} = \frac{1}{K}\sum_{k=1}^{K} Precision_k, \quad Recall_{avg} = \frac{1}{K}\sum_{k=1}^{K} Recall_k, \tag{5}$$

$$F1\text{-}score = \frac{2 \times Precision_{avg} \times Recall_{avg}}{Precision_{avg} + Recall_{avg}}. \tag{6}$$

We used the following equation to calculate the *MCC* [21,22,43]:

$$MCC = \frac{c \times s - \sum_k^K p_k \times t_k}{\sqrt{(s^2 - \sum_k^K p_k^2) \times (s^2 - \sum_k^K t_k^2)}}, \quad (7)$$

where:

- $t_k = \sum_i^K m_{ik}$: the number of samples belonging to class k;
- $p_k = \sum_j^K m_{kj}$: the number of samples predicted as class k;
- $c = \sum_k^K m_{kk}$: the total number of samples correctly predicted;
- $s = \sum_i^K \sum_j^K m_{ij}$: the total number of samples.

5. Evaluation

5.1. Feature Sets

We used the AAPT to extract the permissions requested by malicious apps belonging to the top 20 malware families listed in Table 1. As mentioned in Section 4.1, the number of distinct permissions requested by at least one malicious app is 96. They consist of 87 built-in permissions and 9 custom ones.

We took advantage of the feature importance score of the LightGBM classifier, which represents how many times the feature is used to split the data while constructing trees. We calculated the feature importance score of the 96 permissions and normalized it to the sum of the importance scores of all features. Figure 3 shows the feature importance of the top 10 among the 96 permissions. Permissions related to the network, SMS, or external storage were highly ranked. This implies that such permissions play an important role in malware family classification. On the other hand, there were some permissions whose feature importance was 0. They were 31 built-in permissions and 1 custom permission. These permissions hardly contributed to the malware family classification. Notice that, however, they might be needed by malware to conduct malicious behavior.

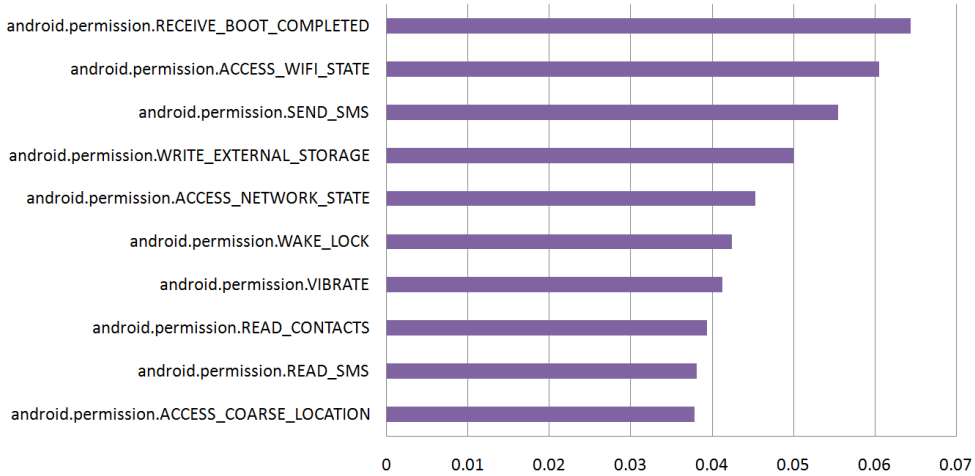

Figure 3. Feature importance of the top 10 permissions.

5.2. Effect of Classifiers and Custom Permissions

We trained several machine learning classifiers and measured their metrics, as suggested in Section 4.3. The experiments were performed using the four feature sets (permission sets) listed as below. The features sets were formed by excluding 0-importance permissions and/or custom permissions from the 96 permissions. These experiments were planned to find out which permission set was most effective and which classifier performed best.

- S96: 96 permissions;

- S87: 87 permissions (excluding custom permissions);
- S64: 64 permissions (excluding 0-importance permissions);
- S56: 56 permissions (excluding custom permissions and 0-importance ones).

We conducted *balanced* 5-fold cross-validation in our experiments. As explained in Section 4, we split each malware family into 5 subsets and distributed them to the folds evenly. Therefore, the ratio of a malware family in any fold was nearly equal to the ratio of the malware family in the whole dataset.

Figures 4–7 show the performance of the classifiers with permission sets S96, S87, S64, and S56, respectively (see Appendix A, Tables A1–A4 for the measured values). On the whole, boosting algorithms (AdaBoost, XGBoost, and LightGBM) performed better than other algorithms. AdaBoost achieved the highest precision, F1-score, ACC, and MCC for S96. LightGBM achieved the highest recall, F1-score, ACC, MCC, and BAC for S87. In many cases, XGBoost lied between AdaBoost and LightGBM. On the other hand, BAC is a metric suitable for imbalanced datasets as explained in Section 4.3. In all cases, i.e., for all classifiers and permission sets, BAC was lower than ACC. This implies that there were *small* families that included many misclassified instances. We look into the family-level performance in Section 5.3.

Table 3 summarizes the experiment results. It shows the most effective classifiers and their measured metrics for each permission set. AdaBoost achieved the highest precision, F1-score, ACC, and MCC, and LightGBM achieved the highest recall and BAC over all permission sets. LightGBM outperformed AdaBoost for permission sets S87, S64, and S56. For permission set S96, however, AdaBoost outperformed LightGBM. The achieved highest MCC was 0.9484 by AdaBoost, and the highest BAC was 0.9198 by LightGBM.

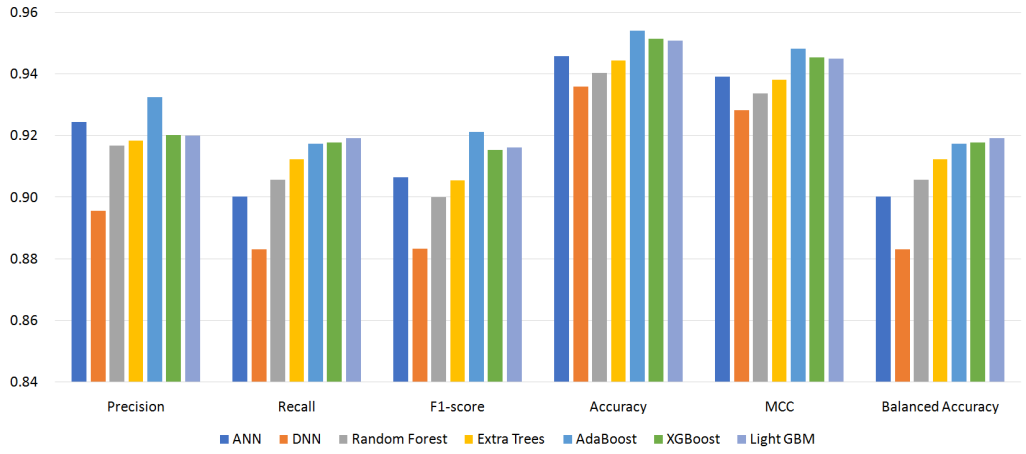

Figure 4. Performance of classifiers with permission set S96.

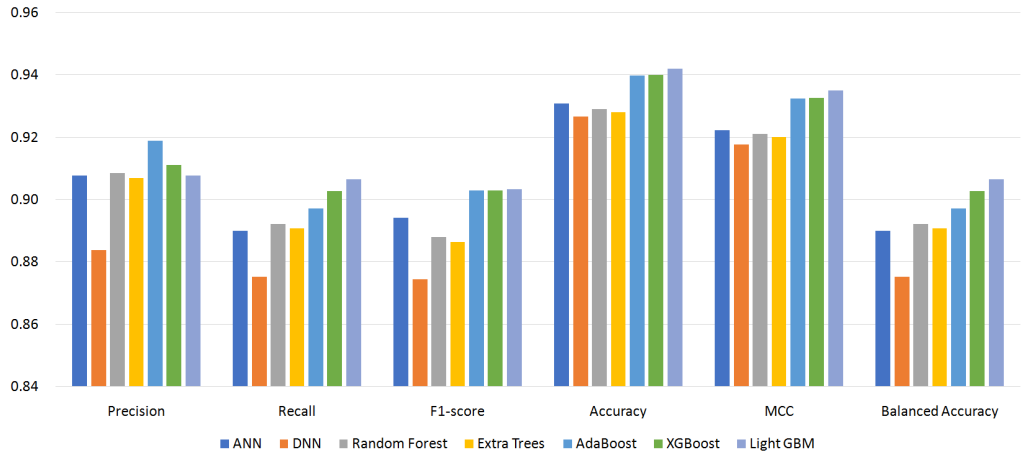

Figure 5. Performance of classifiers with permission set S87.

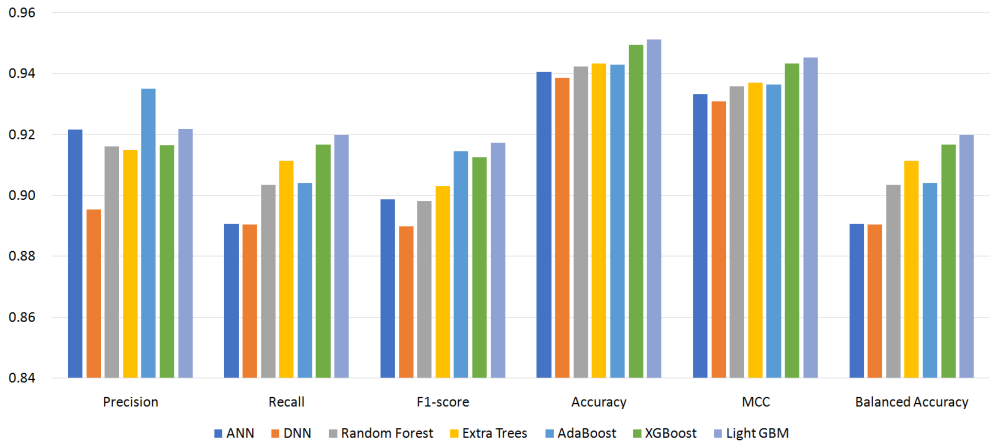

Figure 6. Performance of classifiers with permission set S64.

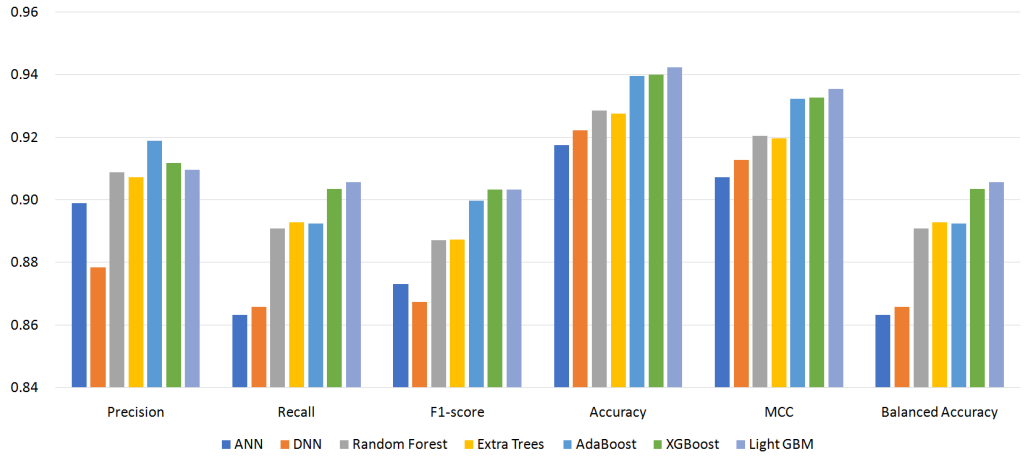

Figure 7. Performance of classifiers with permission set S56.

Table 3. Most effective classifiers and measured values. The highest value for each metric is emphasized as bold-face type.

	S96	**S87**	**S64**	**S56**
Precision	0.9326 (AdaBoost)	0.9190 (AdaBoost)	**0.9351** (AdaBoost)	0.9198 (AdaBoost)
Recall	0.9192 (LightGBM)	0.9066 (LightGBM)	**0.9198** (LightGBM)	0.9056 (LightGBM)
F1-score	**0.9212** (AdaBoost)	0.9033 (LightGBM)	0.9173 (LightGBM)	0.9033 (LightGBM)
ACC	**0.9541** (AdaBoost)	0.9421 (LightGBM)	0.9512 (LightGBM)	0.9424 (LightGBM)
MCC	**0.9484** (AdaBoost)	0.9351 (LightGBM)	0.9453 (LightGBM)	0.9354 (LightGBM)
BAC	0.9192 (LightGBM)	0.9066 (LightGBM)	**0.9198** (LightGBM)	0.9056 (LightGBM)

Note that the highest values were achieved with S96 and S64, which were the permission sets containing custom permissions. To check whether custom permissions contributed to Android malware family classification, we wanted to statistically compare the MCC and BAC when only built-in permissions were used and when both built-in

permissions and custom permissions were used. Thus, the null hypotheses are given as $H_0 : MCC(built\_in) = MCC(all)$ and $H_0 : BAC(built\_in) = BAC(all)$, where *all* includes both built-in and custom permissions. For the two-sided t-test, we conducted 5-fold cross-validation 30 times for LightGBM and then obtained 30 MCCs and BACs. The means of 30 MCCs were 0.9337700 and 0.9474933 for built-in and all permission sets, respectively. The relevant p-value was 5.931×10^{-39}, which is much less than 0.0001. Thus, $MCC(all)$ was significantly bigger than $MCC(built\_in)$. The means of 30 BACs were 0.9090167 and 0.9229400 for built-in and all permission sets, respectively. The relevant p-value was 4.518×10^{-20}, which is much less than 0.0001. Thus, $BAC(all)$ was significantly bigger than $BAC(built\_in)$. Therefore, we noticed that using custom permissions together with built-in permissions was more effective than using only built-in permissions, and then, custom permissions contributed to the Android malware family classification. We also performed the normality test by using the scipy.stats.shapiro() function for the Shapiro–Wilk test, which is known to be a reliable test for normality, on 30 MCCs and BACs of LightGBM. The result of the normality test showed that the data followed a normal distribution.

Figure 8 shows the performance of LightGBM for each permission set. The measured metrics for S96 and S64 (permission sets containing custom permissions) were higher than S87 and S56 (permission sets not containing custom permissions), respectively. This was the same for the other classifiers. Including custom permissions improved the classification performance. On the other hand, the effect of 0-importance permissions was not consistent. Excluding 0-importance permissions slightly improved the classification performance in the majority of cases. However, in some cases, excluding 0-importance permissions degraded the classification performance.

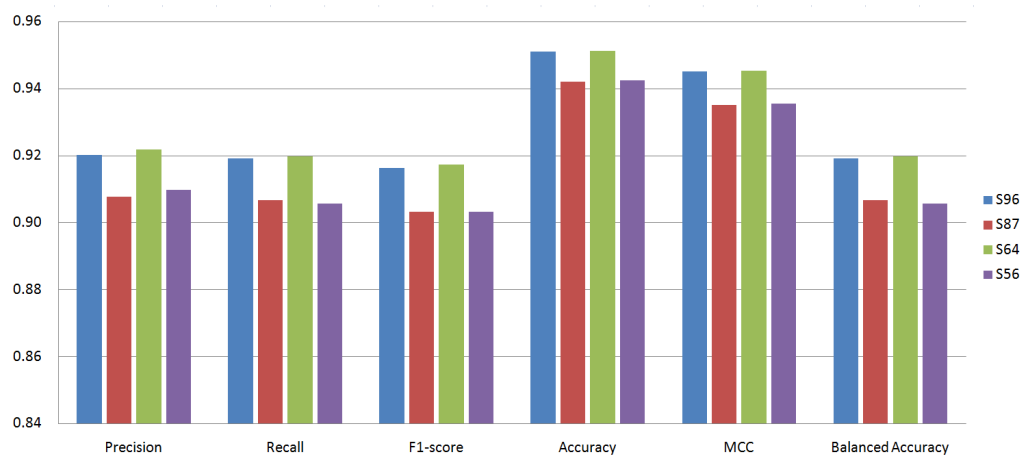

Figure 8. Performance of LightGBM for each permission set.

5.3. Family-Level Performance

This section analyzes the class-level (family-level) performance of our malware family classification. We examined the classification results of LightGBM with S64, which scored the highest BAC. Table A5 is a confusion matrix of LightGBM with S64. It is one of the test results of the 5-fold cross-validation. The family-level precision and recall calculated using this matrix are shown in Figure 9.

As presented in Section 4.3, overall accuracy (ACC) is the ratio of true positive instances to all instances. If *large* families are classified well, there will be many true positive instances. Overall accuracy can be high even if *small* families are classified very poorly. Figure 9 shows that most large families, such as DroidKungFu, FakeInstaller, Opfake, and Plankton (see Table 1), were classified well. Their measured family-level recall was higher than 0.96. Thus, the overall accuracy was very high, although some small families, such as ExploitLinuxLotoor and SMSreg, were classified poorly. The ACC of the confusion matrix (Table A5) was 0.9523. The poor classification of small families degraded the overall accuracy a little.

On the other hand, since BAC is the average of family-level recalls, the poor classification of small families had more effect on BAC than ACC. As we can see from Figure 9, the ExploitLinuxLotoor and SMSreg families had the lowest family-level recalls. Their measured recall was lower than 0.63. The BAC of the confusion matrix was 0.9289, which is lower than ACC. For malware family classification, we needed to measure both ACC and BAC. Furthermore, we also needed to identify poorly classified families and develop methods that can improve both ACC and BAC.

In Figure 9, the GinMaster family had a low recall. Since GinMaster is a relatively large family, many instances of GinMaster were misclassified, as shown in Table A5. This misclassification degraded not only the recall of GinMaster, but also the precision of other families. If we reduce the misclassified instances, both BAC and ACC can be improved.

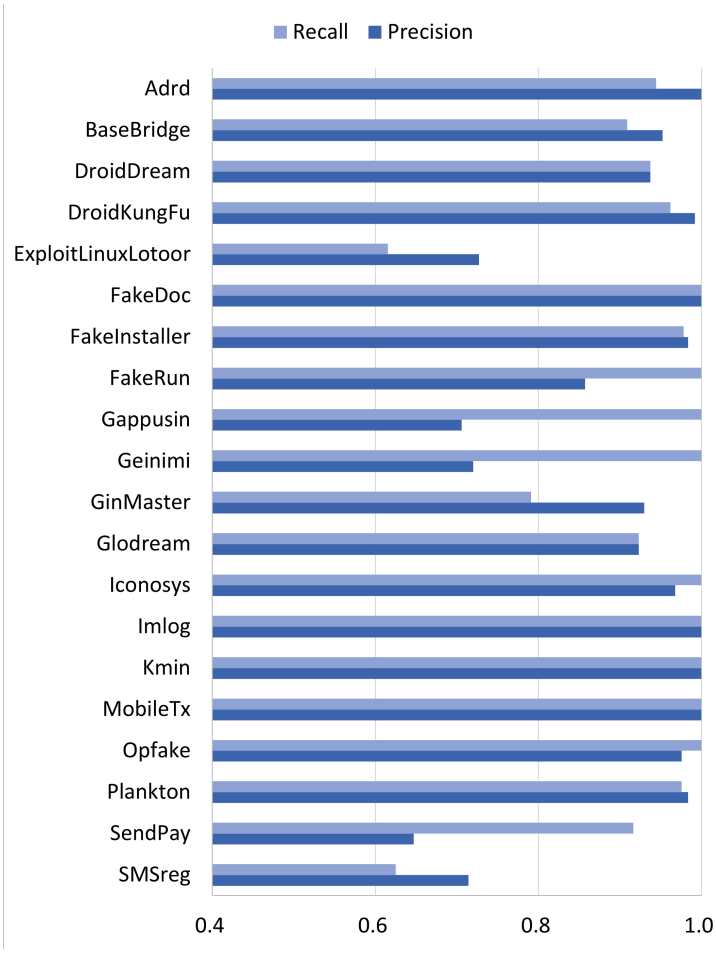

Figure 9. Family-level performance of LightGBM for permission set S64.

6. Related Work

This section describes the existing approaches to permission-based Android malware family classification using machine learning techniques. Alswaina and Elleithy [8] implemented a reverse engineering framework for malware family classification, which selected the permissions declared in malicious apps as static features. They introduced two approaches: the first one used the binary representation of the extracted permissions, and the second one used the features' importance based on the weighted value of each permission. Then, they conducted experiments with six machine learning classifiers using the dataset provided by [9]. The classifiers were k-nearest neighbor (k-NN), neural network (NN), random forest (RF), decision tree (DT), and support vector machine (SVM).

Some studies used other features, as well as permissions for Android malware family classification. Xie et al. [50] first extracted 149 static features, which consisted of 93 built-in

permissions, 41 hardware components, and 15 suspicious API calls. Then, the 20 key features were selected from the 149 features by eliminating less important features with the frequency-based algorithm. They classified malware samples into ten families using SVM. The malware samples were collected from Anzhi market in China.

Türker and Can [51] extracted API calls and permissions as static features. They selected 958 API calls and 42 permissions as important features using a feature ranking method. They used logistic regression (LR), DT, SVM, k-NN, RF, AdaBoost, major voting (MV), and multilayer perceptron (MLP).

Arp et al. [23] extracted various feature sets from `AndroidManifest.xml` and disassembled the bytecodes. The feature sets from `AndroidManifest.xml` include *requested permissions*, filtered intents, hardware components, and app components. The feature sets from disassembled bytecodes include *used permissions*, network addresses, and restricted and suspicious API calls. The *used permissions* refer to the permissions that are requested, as well as actually used during app execution. For the 20 largest malware families, they analyzed the performance to detect each of the families separately using linear SVMs.

Suarez-Tangil et al. [10] developed `DroidSieve`, a malware classification system that considered static and obfuscation resilient features. `DroidSieve` relied on invoked components, permissions, code structure, API calls, obfuscation artifacts, native components, and other obfuscation-invariant features.

Sedano et al. [52] employed the static features such as intents, API calls, permissions, network addresses, hardware components, etc. They proposed an evolutionary algorithm such as a genetic algorithm to select relevant features for characterizing malware families. They conducted experiments with the DREBIN dataset. The experiment results showed that the external information such as network addresses was more relevant than the characteristics of an app itself for identifying a malware family.

Qiu et al. [5] proposed a multilabel classification model that can annotate the malicious capabilities of suspicious malware samples. The model extracted permissions, API calls, and network addresses from malicious apps as the features. They performed experiments with the DREBIN and AMD datasets by applying the linear SVM, DT, and deep neural network (DNN) classifiers.

Bai et al. [16] constructed two kinds of feature sets: 250 manual features and 16,873 documentary features. Both feature sets consisted of attributes of intercomponent communication (ICC), permissions, and API calls. By using the features and applying k-NN, RF, DT, SVM, and basic MLP classifiers to the three different datasets, they investigated the influence of features, classifiers, and datasets. Their findings were that: (i) MLP slightly outperformed the four other classifiers by about 1–3% on the F1-score; (ii) API calls were more relevant features than permissions; (iii) the MLP-based transferability across different datasets was explored.

Chakraborty et al. [53] proposed EC2, which combines malware classification and clustering. They employed RF, SVM, naive Bayes (NB), LR, k-NN, and DT for supervised classification and MeanShift, K-means, affinity, and DBSCAN for unsupervised clustering. They used 190 static features including permissions and 2048 dynamic features including cryptographic usage, network usage, and file I/O. They conducted experiments on the DREBIN and Koodous dataset.

Atzeni et al. [54] introduced a semisupervised scalable framework with the goal of identifying similar apps and generating malware family signatures. The framework mined massive Android apps to automatically cluster malicious apps into families, while reducing the false positive rate. The framework extracted many features through static and dynamic analysis. Static features were obtained from the manifest file (permissions, filters, components) and the bytecode analysis. Dynamic features represented the app interaction with the operating system at the file system and networking module.

Table 4 shows the comparison of the studies on classifying or clustering Android malware families using permission information. Among the studies, Atzeni et al. [54] presented a framework that clusters apps into families and identifies them using formal

rules. To the best of our knowledge, our study is the first to adopt *custom permissions* and the *MCC* for Android malware classification. Moreover, we achieved high performance only using permissions and validated the effectiveness of custom permissions by applying the *p*-value approach to hypothesis testing.

Table 4. Comparison of the studies on Android malware family classification/clustering using permissions.

Study	Features	Classifiers	Metrics	Dataset
A3CM [5]	Requested permissions, used permissions, API calls, network address	SVM, DT, DNN	Precision, recall, F1-score Hamming loss, accuracy	DREBIN, AMD/ 4 capability types
Atzeni et al. [54]	Permissions, components, dynamic app interaction, etc.	Clustering algorithms	Adjusted Rand index, homogeneity, completeness, V-measure	1.5 million apps
Alswaina et al. [8]	Requested permissions	SVM, DT(ID3), RF, NN, k-NN, bagging	Accuracy	StormDroid [9]/ 28 families
EC2 [53]	Permissions, dynamic features (network, system information, etc.)	SVM, RF, DT, k-NN, LR, NB	Precision, recall, F1-score, AUC at macrolevel and microlevels	DREBIN, Koodous/ 44 families
DREBIN [23]	Requested permissions, used permissions, API calls, network address, HW components, etc.	SVM	Accuracy, detection rate	DREBIN/ 20 families
Bai et al. [16]	Permissions, API calls, ICC attributes	SVM, DT, RF, k-NN, MLP	Precision, recall, F1-score, accuracy	Genome, DREBIN, AMD/ 32, 131, 71 families
DroidSieve [10]	Permissions, API calls, code structure, invoked components	Extra Trees	Accuracy, F1-score detection rate	DREBIN / 108 families, PRAGuard/not specified
Xie et al. [50]	Built-in permissions, API calls, HW components	SVM	Accuracy	11,643 apps from Anzhi/ 10 families
AndMFC [51]	Requested permissions, API calls	SVM, DT, LR, k-NN, RF, AdaBoost, MLP, Majority voting	Macrolevel precision, recall, and F1-score, accuracy	AMD, DREBIN, UpDroid/ 71, 132, 20 families
Sedano et al. [52]	Permissions, API calls, HW components, intents, network address, etc.	Genetic algorithm for feature selection	Not specified	DREBIN/ 179 families
Our study	Requested permissions (built-in permissions + custom permissions)	ANN, DNN, RF, Extra Trees, AdaBoost, XGBoost, LightGBM	Precision, recall, and F1-score at macrolevel, accuracy balanced accuracy, MCC	DREBIN/ 20 families

To classify Android malware samples into their corresponding families, some existing studies utilized other features than permissions. Garcia et al. [37] leveraged obfuscation-resilient features such as API usage, reflection-related information, and native code features. They compared their method with the other existing methods and demonstrated that their method was resilient to obfuscation. They used accuracy for evaluating the proposed classification.

Fan et al. [4] constructed sensitive API-call-based frequent subgraphs that represented malicious behavior common to malware samples belonging to a family. They also developed a system called `FalDroid` to efficiently classify large-scale malware samples. The system offered useful knowledge for detecting and investigating Android malware. They evaluated the system in terms of precision, recall, F1-score, ROC, and accuracy.

Raff and Nicholas [15] proposed Stochastic Hashed Weighted Lempel–Ziv (SHWeL), which is an extension of the Lempel–Ziv Jaccard Distance (LZJD). Using the SHWeL vectors, the authors could make efficient algorithms for training and inference. The SHWeL approach was helpful to solve the class imbalance problem. It worked well with the LR classifier and improved balanced accuracy compared to the LZJD and SMOTE.

Kim et al. [55] extracted several features though static and dynamic analysis. They used the permissions, file name, and activity name as the static features and the API call sequence as the dynamic features. They represented the features using a social network

graph and calculated the similarity of malware samples using the weighted sum of feature similarities. Malware samples were clustered by the optimal weights based on social network analysis. They used accuracy as the evaluation metric.

Gao et al. [56] explored an approach for Android malware detection and familial classification using a graph neural network and developed a prototype system called GDroid. The approach mapped both apps and Android APIs into a heterogeneous graph, which was fed into the graph convolutional network model, and utilized a node classification problem for malware classification. GDroid achieved an average accuracy of almost 97% in the malware familial classification on the datasets AMGP, DB, and AMD. They used the precision, recall, and F-measure for the evaluation metrics.

Nisa et al. [57] proposed a feature fusion method that used distinctive pretrained models (AlexNet and Inception-V3) for feature extraction. The method converted binary files of malware to grayscale images and built a multimodal representation of malicious code that could be used to classify the grayscale images. The features extracted from malware images were classified by some variants of SVM, KNN, decision tree, etc. They also performed data augmentation on malware images. Their method was evaluated on a Malimg malware image dataset, achieving an accuracy of 99.3%. They employed the recall, accuracy, AUC, and error rate for the evaluation metrics

Suareze-Tangil et al. [10] proposed an Android malware classification approach, called DroidSieve. DroidSieve targets obfuscated malware and uses features missed by existing techniques. The approach uses features of native components, artifacts introduced by obfuscation, and invariants under obfuscation. The samples were from the DREBIN, MalGenome McAfee, Marvin, and PRAGuard (obfuscated) sets. They detected malware and identified families of them using Extra Trees with ranked features. The approach showed up to 99.82% accuracy for detection and 99.26% for classification.

Cai et al [58] proposed a dynamic app classification approach, called DroidCat. DroidCat first characterizes benign and malware samples and extracts features based on method calls and intercomponent communication (ICC) intents. These features represent the structure of app executions and are robust against reflection, resource obfuscation, system-call obfuscation, and malware evolution. The features include the distributions of method calls among user code, third-party libraries, and the SDK, as well as the percentage of ICCs that is implicit and external. They collected samples from AndroZoo, Google Play, VirsuShare, and MalGenome and used the random forest algorithm. DroidCat achieved a 97% F1-score for the detection and categorization of malware.

7. Discussion

In Android, while much research has utilized only permissions as the feature for malware detection, malware family classification studies that utilized only permissions as the feature are very few. Since our approach utilizes only the requested permissions contained in the `AndroidManifest.xml` of apps, it can be simply extended to identify a new family that was previously unknown. Android permissions are very significant features for machine learning models because they are obfuscation resilient [6,10,37]. In addition, the requested permissions are more easily extracted than the used permissions [59] and can be effective indicators to detect Android malware [7].

By analyzing the relationship between Android malware families and permissions, we found that certain permissions were requested by only one malware family. Table 5 shows the permissions requested by only one specific family. The malware samples with the permissions listed in the table can be simply classified into their corresponding family. However, permissions may have some limitations. According to [37], permissions are very granular. In this paper, to tackle the granularity issue of permissions, we introduced *custom permissions*, as well as *built-in permissions*. Another issue is that malicious apps can perform malicious behavior without any permission [40]. Actually, our investigation has shown that 19 of the 4615 malware samples did not request any permission. Twelve of the nineteen samples belonged to the `Geinimi` family. In our experiments, all nineteen samples

without any permission were classified into `Geinimi`, that is the seven samples without any permission were misclassified. To address this issue, we plan to consider the other features in `AndroidManifest.xml` such as intents, components, etc.

Table 5. Permissions requested only by a specific family.

Family Name	Permission List
Adrd	android.permission.WRITE_CALENDAR android.permission.BROADCAST_PACKAGE_REMOVED
DroidKunFu	android.permission.BROADCAST_WAP_PUSH android.permission.CALL_PRIVILEGED
ExploitLinuxLotoor	android.permission.REORDER_TASKS
FakeInstall	android.permission.ACCOUNT_MANAGER android.permission.BRICK android.permission.BROADCAST_SMS android.permission.CLEAR_APP_USER_DATA android.permission.READ_CALENDAR
Geinimi	com.google.android.googleapps.permission.GOOGLE_AUTH
Opfake	android.permission.GLOBAL_SEARCH android.permission.UPDATE_DEVICE_STATS com.android.alarm.permission.SET_ALARM
Plankton	android.permission.ACCESS_SURFACE_FLINGER android.permission.BACKUP android.permission.BIND_APPWIDGET android.permission.CHANGE_WIFI_MULTICAST_STATE

The next issue is the sustainability. Owing to the evolution of the Android framework and malware, the performance of machine-learning-based classification might be degraded. As the Android framework evolves, Android APIs and permissions are newly introduced or removed. Machine-learning-based classifiers trained using old features, such as old APIs and old permissions, may not correctly classify new malware without frequent retraining. Many researchers have addressed this issue [60–65]. In a recent research work [64], the authors defined sustainability metrics and compared state-of-the-art Android malware detectors. The authors also proposed a sensitive access distribution (SAD) profile and developed a SAD-based malware detection system, DroidSpan. The experiments using datasets over 8 y showed that DroidSpan outperforms other detectors in sustainability. The sustainability of our malware family classification can be affected by the evolution of Android permissions, the evolution of malware exploiting new permissions, and the emergence of new families. We plan to assess the sustainability of our classification in future work.

The samples in the DREBIN dataset were collected from August 2010 to October 2012 with API Levels 9 through 19. According to the findings in [13], the DREBIN dataset has apps from the time when *built-in permissions* were common and *custom permissions* were seldom used. Therefore, the custom permissions may not have a big impact on classifying Android malware families of the DREBIN dataset. In the near future, we will construct an Android malware family classification technique using more recent data and evaluate the effect of custom permissions. Furthermore, we will update the relevant permissions list for malware family classification by selecting significant built-in and custom permissions from the latest Android versions and newer malicious apps.

We utilized only the existence of *requested permissions* and did not consider the frequency of occurrence of individual requested permissions. As future work, we plan to consider the frequency of the real occurrences of individual permissions after extracting them from disassembled code.

Several existing research works on malware family classification utilized not only permissions, but also other features such as *API calls* [5,10,16,23,37,50–52], *hardware components* [23,50,52], *intents* [23,37,52,54], *network addresses* [5,23,52], etc. According to the

results of [16], API calls can be a more effective feature than permissions for Android malware family classification. However, it is hard to statically extract API calls if code packing and method hiding [66], encryption, or reflection [67] techniques are applied to malware samples.

As the source of the features to classify Android malware families, `AndroidManifest.xml` has three advantages [53]: (i) the features extracted from the code of a DEX file may bring in excessively detailed information, whereas `Manifest` has plentiful information about an app and its structure; (ii) the features extracted from the code may produce meaningless information due to code encryption or reflection, whereas `Manifest` is described in plaintext and includes many details about permissions, components, and interfaces; (iii) the `Manifest` file has significant information to identify malware families. According to those advantages, the classification performance would be improved if the features such as components and intents were extracted together from the `Manifest` file and utilized.

For the metrics to evaluate a classification model, most of the existing studies for Android malware family classification adopted *accuracy* [4,5,8,10,16,23,37,50,51,54,55] of the *F1-score* [4,5,10,16,51,53]. Only one previous study [15] used *balanced accuracy* for the evaluation metric. In this paper, we used the precision, recall, F1-score at the macrolevel, accuracy, balanced accuracy, and *MCC* for the metrics. The MCC is a good metric that evaluates classification tasks on imbalanced datasets. To the best of our knowledge, we are the first to adopt MCC for evaluating the performance of Android malware family classification. Our approach achieved a high MCC (up to 0.9484 in the case of S96 and AdaBoost) by using only the requested permissions as static features.

8. Conclusions

In this article, we proposed a new approach to classify Android malware families using only the requested permissions that consist of built-in and custom permissions. Those permission were directly extracted from `AndroidManifest.xml` of each malicious app. Some of them have the value of zero for the feature importance and did not play an important role in malware family classification. We constructed four kinds of permission sets as static features by including or excluding the custom permissions and the zero-importance permissions. We then conducted various experiments with seven classifiers on the top twenty largest malware families in the well-known DREBIN dataset.

We evaluated the performance of the classifiers in terms of the precision, recall, F1-score, accuracy, balanced accuracy, and MCC. Balanced accuracy and the MCC are known as good metrics to evaluate multiclass classification models on imbalanced datasets. The experiment results showed that LightGBM achieved the best balanced accuracy of 0.9198 with 64 permissions (56 built-in + 8 custom permissions), while the same classifier achieved a balanced accuracy of 0.9192 with 96 permissions (87 built-in + 9 custom permissions). On the other hand, AdaBoost achieved the highest MCC of 0.9484 with 96 permissions, while the same classifier achieved an MCC of 0.9364 with 64 permissions. The highest accuracy and F1-score was 0.9541 and 0.9212 when the AdaBoost classifier was applied with the 96 permissions.

We also analyzed the effect of the custom permissions. Using all the permissions including both the zero-importance permissions and custom permissions, LightGBM achieved a better F1-score, accuracy, balanced accuracy, and MCC by 1.29%, 0.89%, 1.26%, and 1.0%, respectively, compared to the excluding custom permissions. By excluding all the permissions with the zero-importance value, but including custom permissions, LightGBM also achieved a better F1-score, accuracy, balanced accuracy, and MCC by 1.4%, 0.88%, 1.42%, and 0.99%, respectively, compared to excluding custom permissions. Finally, the effectiveness of custom permissions was verified by applying a p-value approach to hypothesis testing.

In future work, we will try to seek other features that are obfuscation resilient and effective for malware family classification and enhance our technique. We will also collect

datasets with more recent malware samples and the small-sized families and investigate the effects of the custom permissions and the evaluation metrics.

Author Contributions: Software, M.K. and D.K.; data curation, M.K. and D.K.; writing—original draft preparation, S.C., S.H. and M.P.; writing—review and editing, C.H. and S.H.; supervision, S.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (No. 2021R1A2C2012574).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of the data; in the writing of the manuscript; nor in the decision to publish the results.

Appendix A

Tables A1–A4 show the measured metrics of the classifiers for permission sets S96, S87, S64, and S56, respectively. P_{avg} denotes $Precision_{avg}$, and R_{avg} denotes $Recall_{avg}$. Table A5 is a confusion matrix of one of the five-fold cross-validations using LightGBM for S64.

Table A1. Measured metrics for permission set S96.

	P_{avg}	R_{avg}	F1-Score	ACC	MCC	BAC
ANN	0.9246	0.9003	0.9065	0.9458	0.9392	0.9003
DNN	0.8956	0.8832	0.8834	0.9361	0.9284	0.8832
RF	0.9168	0.9058	0.9001	0.9404	0.9337	0.9058
ET	0.9184	0.9125	0.9055	0.9445	0.9382	0.9125
AdaBoost	**0.9326**	0.9174	**0.9212**	**0.9541**	**0.9484**	0.9174
XGBoost	0.9202	0.9178	0.9155	0.9515	0.9455	0.9178
LightGBM	0.9201	**0.9192**	0.9162	0.9510	0.9451	**0.9192**

Table A2. Measured metrics for permission set S87.

	P_{avg}	R_{avg}	F1-Score	ACC	MCC	BAC
ANN	0.9078	0.8900	0.8942	0.9309	0.9224	0.8900
DNN	0.8838	0.8753	0.8744	0.9268	0.9178	0.8753
RF	0.9086	0.8922	0.8880	0.9291	0.9212	0.8922
ET	0.9070	0.8908	0.8865	0.9281	0.9201	0.8908
AdaBoost	**0.9190**	0.8973	0.9029	0.9398	0.9324	0.8973
XGBoost	0.9111	0.9028	0.9029	0.9400	0.9326	0.9028
LightGBM	0.9077	**0.9066**	**0.9033**	**0.9421**	**0.9351**	**0.9066**

Table A3. Measured metrics for permission set S64.

	P_{avg}	R_{avg}	F1-Score	ACC	MCC	BAC
ANN	0.9217	0.8906	0.8988	0.9406	0.9333	0.8906
DNN	0.8953	0.8904	0.8899	0.9385	0.9309	0.8904
RF	0.9160	0.9034	0.8981	0.9424	0.9358	0.9034
ET	0.9149	0.9113	0.9030	0.9434	0.9370	0.9113
AdaBoost	**0.9351**	0.9040	0.9146	0.9430	0.9364	0.9040
XGBoost	0.9165	0.9167	0.9125	0.9495	0.9434	0.9167
LightGBM	0.9219	**0.9198**	**0.9173**	**0.9512**	**0.9453**	**0.9198**

Table A4. Measured metrics for permission set S56.

	P_{avg}	R_{avg}	F1-Score	ACC	MCC	BAC
ANN	0.8989	0.8632	0.8731	0.9174	0.9073	0.8632
DNN	0.8784	0.8658	0.8674	0.9222	0.9127	0.8658
RF	0.9089	0.8909	0.8871	0.9285	0.9205	0.8909
ET	0.9073	0.8928	0.8874	0.9276	0.9196	0.8928
AdaBoost	**0.9189**	0.8925	0.8997	0.9395	0.9322	0.8925
XGBoost	0.9117	0.9035	**0.9033**	0.9400	0.9326	0.9035
LightGBM	0.9097	**0.9056**	**0.9033**	**0.9424**	**0.9354**	**0.9056**

Table A5. Confusion matrix of LightGBM with S64.

	Adrd	**Base-Bridge**	**Droid-Dream**	**Droid-KungFu**	**Expl** [1]	**Fake-Doc**	**Fake-Inst** [2]	**Fake-Run**	**Gapp-Usin**	**Gei-Nimi**	**Gin-Master**	**Glod-Ream**	**Icon-Osys**	**Imlog**	**Kmin**	**Mobi-LeTx**	**Op-Fake**	**Plan-Kton**	**SMS-Reg**	**Send-Pay**
Adrd	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
BaseBridge	0	60	0	0	3	0	1	0	0	2	0	0	0	0	0	0	0	0	0	0
DroidDream	0	0	15	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
DroidKungFu	0	0	1	127	0	0	0	0	1	0	3	0	0	0	0	0	0	0	0	0
Expl [1]	0	0	0	0	8	0	0	0	2	3	0	0	0	0	0	0	0	0	0	0
FakeDoc	0	0	0	0	0	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0
FakeInst [2]	0	0	0	0	0	0	180	0	0	0	0	0	1	0	0	0	2	0	1	0
FakeRun	0	0	0	0	0	0	0	12	0	0	0	0	0	0	0	0	0	0	0	0
Gappusin	0	0	0	0	0	0	0	0	12	0	0	0	0	0	0	0	0	0	0	0
Geinimi	0	0	0	0	0	0	0	0	0	18	0	0	0	0	0	0	0	0	0	0
GinMaster	0	2	0	1	0	0	0	0	1	1	53	0	0	0	0	0	1	2	0	6
Glodream	0	1	0	0	0	0	0	0	0	0	0	12	0	0	0	0	0	0	0	0
Iconosys	0	0	0	0	0	0	0	0	0	0	0	0	30	0	0	0	0	0	0	0
Imlog	0	0	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0	0	0	0
Kmin	0	0	0	0	0	0	0	0	0	0	0	0	0	0	29	0	0	0	0	0
MobileTx	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	0	0	0	0
Opfake	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	120	0	0	0
Plankton	0	0	0	0	0	0	0	2	0	0	0	1	0	0	0	0	0	121	0	0
SMSreg	0	0	0	0	0	0	2	0	0	1	0	0	0	0	0	0	0	0	5	0
SendPay	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	11

[1] Expl: ExploitLinuxLotoor, [2] FakeInst: FakeInstaller.

References

1. Development of New Android Malware Worldwide from June 2016 to March 2020. Available online: https://www.statista.com/statistics/680705/global-android-malware-volume/ (accessed on 16 January 2021).
2. Mobile Malware Report—No Let-Up with Android Malware. Available online: https://www.gdatasoftware.com/news/2019/07/35228-mobile-malware-report-no-let-up-with-android-malware (accessed on 16 January 2021).
3. Alswaina, F.; Elleithy, K. Android Malware Family Classification and Analysis: Current Status and Future Directions. *Electronics* **2020**, *9*, 942. [CrossRef]
4. Fan, M.; Liu, J.; Luo, X.; Chen, K.; Tian, Z.; Zheng, Q.; Liu, T. Android malware familial classification and representative sample selection via frequent subgraph analysis. *IEEE Trans. Inf. Forensics Secur.* **2018**, *13*, 1890–1905. [CrossRef]
5. Qiu, J.; Zhang, J.; Luo, W.; Pan, L.; Nepal, S.; Wang, Y.; Xiang, Y. A3CM: Automatic Capability Annotation for Android Malware. *IEEE Access* **2019**, *7*, 147156–147168. [CrossRef]
6. Wang, W.; Zhao, M.; Gao, Z.; Xu, G.; Xian, H.; Li, Y.; Zhang, X. Constructing features for detecting android malicious applications: Issues, taxonomy and directions. *IEEE Access* **2019**, *7*, 67602–67631. [CrossRef]
7. Altaher, A. An improved Android malware detection scheme based on an evolving hybrid neuro-fuzzy classifier (EHNFC) and permission-based features. *Neural Comput. Appl.* **2017**, *28*, 4147–4157. [CrossRef]
8. Alswaina, F.; Elleithy, K. Android malware permission-based multi-class classification using extremely randomized trees. *IEEE Access* **2018**, *6*, 76217–76227. [CrossRef]
9. Chen, S.; Xue, M.; Tang, Z.; Xu, L.; Zhu, H. 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, Xi'an, China, 30 May–3 June 2016; pp. 377–388.
10. Suarez-Tangil, G.; Dash, S.K.; Ahmadi, M.; Kinder, J.; Giacinto, G.; Cavallaro, L. Droidsieve: Fast and accurate classification of obfuscated android malware. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY 2017), Scottsdale, AZ, USA, 22–24 March 2017; pp. 309–320.
11. Li, J.; Sun, L.; Yan, Q.; Li, Z.; Srisa-An, W.; Ye, H. Significant Permission Identification for Machine-Learning-Based Android Malware Detection. *IEEE Trans. Ind. Inform.* **2018**, *14*, 3216–3225. [CrossRef]
12. Park, J.; Kang, M.; Cho, S.J.; Han, H.; Suh, K. Analysis of Permission Selection Techniques in Machine Learning-based Malicious App Detection. In Proceedings of the 3rd IEEE International Conference on Artificial Intelligence and Knowledge Engineering (AIKE), Laguna Hills, CA, USA, 9–13 December 2020; pp. 13–22.
13. Mathur, A.; Podila, L.M.; Kulkarni, K.; Niyaz, Q.; Javaid, A.Y. NATICUSdroid: A malware detection framework for Android using native and custom permissions. *J. Inf. Secur. Appl.* **2021**, *58*, 102696.
14. Rossow, C.; Dietrich, C.J.; Grier, C.; Kreibich, C.; Paxson, V.; Pohlmann, N.; Steen, M. Prudent practices for designing malware experiments: Status quo and outlook. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 23–27 May 2012; pp. 65–79. [CrossRef]
15. Raff, E.; Nicholas, C. Malware classification and class imbalance via stochastic hashed LZJD. In Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security (AISec), Dallas, TX, USA, 3 November 2017; pp. 111–120.
16. Bai, Y.; Xing, Z.; Ma, D.; Li, X.; Feng, Z. Comparative analysis of feature representations and machine learning methods in Android family classification. *Comput. Netw.* **2020**, *184*, 107639. [CrossRef]
17. Davis, J.; Goadrich, M. The relationship between Precision-Recall and ROC curves. In Proceedings of the 23rd International Conference on Machine Learning, Pittsburgh, PA, USA, 25–29 June 2006; pp. 233–240.
18. Saito, T.; Rehmsmeier, M. The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets. *PLoS ONE* **2015**, *10*, e0118432. [CrossRef] [PubMed]
19. Roy, S.; DeLoach, J.; Li, Y.; Herndon, N.; Caragea, D.; Ou, X.; Ranganath, V.P.; Li, H.; Guevara, N. Experimental study with real-world data for android app security analysis using machine learning. In Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, 7–11 December 2015; pp. 81–90.
20. Carrillo, H.; Brodersen, K.H.; Castellanos, J.A. Probabilistic performance evaluation for multiclass classification using the posterior balanced accuracy. In Proceedings of the ROBOT2013: First Iberian Robotics Conference, Madrid, Spain, 28–29 November 2013; pp. 347–361.
21. Grandini, M.; Bagli, E.; Visani, G. Metrics for Multi-Class Classification: An Overview. *arXiv* **2020**, arXiv:2008.05756.
22. Jurman, G.; Riccadonna, S.; Furlanello, C. A comparison of MCC and CEN error measures in multi-class prediction. *PLoS ONE* **2012**, *7*, e41882. [CrossRef]
23. Arp, D.; Spreitzenbarth, M.; Hubner, M.; Gascon, H.; Rieck, K. DREBIN: Effective and explainable detection of android malware in your pocket. In Proceedings of the 2014 Network and Distributed System Security (NDSS) Symposium, San Diego, CA, USA, 23–26 February 2014; pp. 23–26.
24. Breiman, L. Random Forests. *Mach. Learn.* **2001**, *45*, 5–32. [CrossRef]
25. Russell, I. Neural Networks Module. 2012. Available online: https://scholar.google.co.jp/citations?view_op=view_citation&hl=zh-TW&user=Oy46FHsAAAAJ&sortby=pubdate&citation_for_view=Oy46FHsAAAAJ:_FxGoFyzp5QC (accessed on 29 July 2021).

26. Hinton, G.E.; Salakhutdinov, R.R. Reducing the dimensionality of data with neural networks. *Science* **2006**, *313*, 504–507. [CrossRef]
27. Geurts, P.; Ernst, D.; Wehenkel, L. Extremely randomized trees. *Mach. Learn.* **2006**, *63*, 3–42. [CrossRef]
28. Freund, Y.; Schapire, R.E.; Abe, N. A short introduction to boosting. *J. Jpn. Soc. Artif. Intell.* **1999**, *14*, 771–780.
29. Chen, T.; Guestrin, C. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 785–794.
30. Ke, G.; Meng, Q.; Finley, T.; Wang, T.; Chen, W.; Ma, W.; Ye, Q.; Liu, T.Y. Lightgbm: A highly efficient gradient boosting decision tree. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 3146–3154.
31. Xiao, J.; Chen, S.; He, Q.; Feng, Z.; Xue, X. An Android application risk evaluation framework based on minimum permission set identification. *J. Syst. Softw.* **2020**, *163*, 110533. [CrossRef]
32. Backes, M.; Bugiel, S.; Derr, E.; McDaniel, P.; Octeau, D.; Weisgerber, S. On demystifying the android application framework: Re-visiting android permission specification analysis. In Proceedings of the 25th USENIX Security Symposium (USENIX Security 16), Austin, TX, USA, 10–12 August 2016; pp. 1101–1118.
33. Tuncay, G.S.; Demetriou, S.; Ganju, K.; Gunter, C. Resolving the Predicament of Android Custom Permissions. In Proceedings of the Network and Distributed System Security Symposium (NDSS 2018), San Diego, CA, USA, 18–21 February 2018.
34. Bagheri, H.; Kang, E.; Malek, S.; Jackson, D. Detection of design flaws in the android permission protocol through bounded verification. In Proceedings of the International Symposium on Formal Methods, Oslo, Norway, 24–26 June 2015; pp. 73–89.
35. Bagheri, H.; Kang, E.; Malek, S.; Jackson, D. A formal approach for detection of security flaws in the android permission system. *Form. Asp. Comput.* **2018**, *30*, 525–544. [CrossRef]
36. Sadeghi, A.; Jabbarvand, R.; Ghorbani, N.; Bagheri, H.; Malek, S. A temporal permission analysis and enforcement framework for android. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018; pp. 846–857.
37. Garcia, J.; Hammad, M.; Malek, S. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Trans. Softw. Eng. Methodol.* **2018**, *26*, 1–29. [CrossRef]
38. Dilhara, M.; Cai, H.; Jenkins, J. 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, Gothenburg, Sweden, 27–28 May 2018; pp. 67–71.
39. Wang, Y.; Wang, Y.; Wang, S.; Liu, Y.; Xu, C.; Cheung, S.C.; Yu, H.; Zhu, Z. Runtime Permission Issues in Android Apps: Taxonomy, Practices, and Ways Forward. *arXiv* **2021**, arXiv:2106.13012.
40. Aafer, Y.; Du, W.; Yin, H. Droidapiminer: Mining api-level features for robust malware detection in android. In Proceedings of the International Conference on Security and Privacy in Communication Systems (SecureComm 2013), Sydney, Australia, 25–27 September 2013; pp. 86–103.
41. Hanley, J.A.; McNeil, B.J. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology* **1982**, *143* 29–36. [CrossRef] [PubMed]
42. Marsland, S. *Machine Learning: An Algorithmic Perspective*, 2nd ed.; CRC Press: Boca Raton, FL, USA, 2015.
43. Chicco, D.; Jurman, G. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genom.* **2020**, *21*, 6. [CrossRef] [PubMed]
44. Sanz, B.; Santos, I.; Laorden, C.; Ugarte-Pedrero, X.; Bringas, P.G.; Álvarez, G. Puma: Permission usage to detect malware in android. In Proceedings of the International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions, Ostrava, Czech Republic, 5–7 September 2012; pp. 289–298.
45. Scikit-Learn: Machine Learning in Python. Available online: https://scikit-learn.org (accessed on 20 October 2020).
46. Idrees, F.; Rajarajan, M.; Conti, M.; Chen, T.M.; Rahulamathavan, Y. PIndroid: A novel Android malware detection system using ensemble learning methods. *Comput. Secur.* **2017**, *68*, 36–46. [CrossRef]
47. Feng, P.; Ma, J.; Sun, C.; Xu, X.; Ma, Y. A novel dynamic Android malware detection system with ensemble learning. *IEEE Access* **2018**, *6*, 30996–31011. [CrossRef]
48. Zhang, Y.; Huang, Q.; Ma, X.; Yang, Z.; Jiang, J. Using multi-features and ensemble learning method for imbalanced malware classification. In Proceedings of the 2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, 23–26 August 2016; pp. 965–973.
49. Performance Measures for Multi-Class Problems. Available online: https://www.datascienceblog.net/post/machine-learning/performance-measures-multi-class-problems/ (accessed on 24 February 2021).
50. Xie, N.; Wang, X.; Wang, W.; Liu, J. Fingerprinting Android malware families. *Front. Comput. Sci.* **2019**, *13*, 637–646. [CrossRef]
51. Türker, S.; Can, A.B. Andmfc: Android malware family classification framework. In Proceedings of the 30th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC Workshops), Istanbul, Turkey, 8 September 2019; pp. 1–6.
52. Sedano, J.; Chira, C.; González, S.; Herrero, Á.; Corchado, E.; Villar, J.R. Characterization of android malware families by a reduced set of static features. In Proceedings of the International Joint Conference SOCO'16-CISIS'16-ICEUTE'16, San Sebastián, Spain, 19–21 October 2016; pp. 607–617.
53. Chakraborty, T.; Pierazzi, F.; Subrahmanian, V.S. EC2: Ensemble Clustering and Classification for Predicting Android Malware Families. *IEEE Trans. Dependable Secur. Comput.* **2017**, *17*, 262–277. [CrossRef]

54. Atzeni, A.; Díaz, F.; Marcelli, A.; Sánchez, A.; Squillero, G.; Tonda, A. Countering android malware: A scalable semi-supervised approach for family-signature generation. *IEEE Access* **2018**, *6*, 59540–59556. [CrossRef]
55. Kim, H.M.; Song, H.M.; Seo, J.W.; Kim, H.K. Andro-simnet: Android malware family classification using social network analysis. In Proceedings of the 16th Annual Conference on Privacy, Security and Trust (PST), Belfast, UK, 28–30 August 2018; pp. 1–8.
56. Gao, H.; Cheng, S.; Zhang, W. GDroid: Android malware detection and classification with graph convolutional network. *Comput. Secur.* **2021**, *106*, 102264. [CrossRef]
57. Nisa, M.; Shah, J.H.; Kanwal, S.; Raza, M.; Khan, M.A.; Damaševičius, R.; Blažauskas, T. Hybrid malware classification method using segmentation-based fractal texture analysis and deep convolution neural network features. *Appl. Sci.* **2020**, *10*, 4966. [CrossRef]
58. Cai, H.; Meng, N.; Ryder, B.; Yao, D. DroidCat: Effective Android Malware Detection and Categorization via App-Level Profiling. *IEEE Trans. Inf. Forensics Secur.* **2019**, *14*, 1455–1470. [CrossRef]
59. Liu, X.; Liu, J. A two-layered permission-based Android malware detection scheme. In Proceedings of 2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, Oxford, UK, 8–11 April 2014; pp. 142–148.
60. Cai, H. 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, Seoul, Korea, 13–15 July 2020; pp. 31–35.
61. Fu, X.; Cai, H. On the deterioration of learning-based malware detectors for Android. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering, Montreal, QC, Canada, 25–31 May 2019; pp. 272–273.
62. Zhang, X.; Zhang, Y.; Zhong, M.; Ding, D.; Cao, Y.; Zhang, Y.; Zhang, M.; Yang, M. Enhancing state-of-the-art classifiers with API semantics to detect evolved android malware. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, 30 October–3 November 2020; pp. 757–770.
63. Xu, K.; Li, Y.; Deng, R.; Chen, K.; Xu, J. DroidEvolver: Self-evolving Android malware detection system. In Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P), Stockholm, Sweden, 17–19 June 2019; pp. 47–62.
64. Cai, H. Assessing and improving malware detection sustainability through app evolution studies. *ACM Trans. Softw. Eng. Methodol.* **2020**, *29*, 1–28. [CrossRef]
65. Cai, H.; Jenkins, J. Towards sustainable android malware detection. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018; pp. 350–351.
66. Yang, W.; Zhang, Y.; Li, J.; Shu, J.; Li, B.; Hu, W.; Gu, D. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In Proceedings of the 18th International Symposium on Recent Advances in Intrusion Detection (RAID 2015), Kyoto, Japan, 2–4 November 2015; pp. 359–381.
67. Aonzo, S.; Georgiu, G.C.; Verderame, L.; Merlo, A. Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX* **2020**, *11*, 100403. [CrossRef]