

Article

MFDroid: A Stacking Ensemble Learning Framework for Android Malware Detection

Xusheng Wang¹, Linlin Zhang^{2,*}, Kai Zhao¹, Xuhui Ding¹ and Mingming Yu² 

¹ School of Cyber Science and Engineering, College of Information Science and Engineering, Xinjiang University, Urumqi 830046, China; wang_xs98@foxmail.com (X.W.); zhaokk@xju.edu.cn (K.Z.); xhding2021@163.com (X.D.)

² School of Software, Xinjiang University, Urumqi 830046, China; yumm0408@foxmail.com

* Correspondence: zllnadasha@xju.edu.cn

Abstract: As Android is a popular a mobile operating system, Android malware is on the rise, which poses a great threat to user privacy and security. Considering the poor detection effects of the single feature selection algorithm and the low detection efficiency of traditional machine learning methods, we propose an Android malware detection framework based on stacking ensemble learning—MFDroid—to identify Android malware. In this paper, we used seven feature selection algorithms to select permissions, API calls, and opcodes, and then merged the results of each feature selection algorithm to obtain a new feature set. Subsequently, we used this to train the base learner, and set the logical regression as a meta-classifier, to learn the implicit information from the output of base learners and obtain the classification results. After the evaluation, the F1-score of MFDroid reached 96.0%. Finally, we analyzed each type of feature to identify the differences between malicious and benign applications. At the end of this paper, we present some general conclusions. In recent years, malicious applications and benign applications have been similar in terms of permission requests. In other words, the model of training, only with permission, can no longer effectively or efficiently distinguish malicious applications from benign applications.

Keywords: Android malware; ensemble learning; machine learning; static analysis; feature selection



Citation: Wang, X.; Zhang, L.; Zhao, K.; Ding, X.; Yu, M. MFDroid: A Stacking Ensemble Learning Framework for Android Malware Detection. *Sensors* **2022**, *22*, 2597. <https://doi.org/10.3390/s22072597>

Academic Editors: Alexios Mylonas and Nikolaos Pitropakis

Received: 14 February 2022

Accepted: 21 March 2022

Published: 28 March 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Android is the mobile operating system with the highest market share in the world. As of December 2021, the market share of Android was as high as 70% [1]. As the number of Android users has risen in recent years, malware, such as financial losses and privacy disclosure, have become more common. In many Asian countries, the risk of being infected with malware is much higher. There are many app stores provided by various third-party vendors and many smartphones have been rooted. There are about 1.61 billion active mobile devices in China, of which, about 78.6% run Android as the operating system [2]. Therefore, from the perspective of Android information security, it is necessary to research malware detection technology and improve detection performance.

At present, there are two mainstream malware detection methods, static detection and dynamic detection [3,4]. The techniques involved in static detection include decompilation, reverse analysis, and static system call analysis. Static analysis does not need to run the application, it uses decompilation tools to perform lexical analysis, semantic analysis, etc., on the static code to extract features. APK files contain many features, such as permissions, API calls, signatures, network addresses, and hardware structures. All features can be used as a basis for judging whether an Android application is malicious. Another detection technique is dynamic detection, which places the application in a sandbox isolated from the outside world and observes the behavior of the application. Dynamic analysis involves analyzing the behavior characteristics of the application without disturbing the external software and hardware environment. If the behavior of the application is found to be

inconsistent with the routine, it is judged as malware. Dynamic features are mainly extracted from the “behavior” that occurs when an Android application is running, and the most important of these features are the underlying system call information and network traffic interaction information. Every application requires the operating system to provide the most basic resources and services to achieve its functions. At the same time, due to the characteristics of Android system architecture, applications belonging to the application layer cannot directly interact with the operating system, so the application also needs to use system calls to perform certain tasks, such as file read, write, and open [5].

With the development of machine learning technology, more researchers have applied it to Android malware detection [6]. They extract features of Android application through static analysis or dynamic analysis and utilize machine learning classification models to detect Android malware. Although malware detection based on a single model has been clearly studied, the performance of each classifier varies due to differences in training data and feature selection algorithms. To overcome the shortcoming of single-model-based methods, an ensemble learning method is proposed. It combines multiple individual classifiers and usually achieves better generalization performance than a single model. Therefore, we built a stacking ensemble framework, called MFDroid, for malware detection. We collected 1664 real applications from 4 app markets in China to evaluate the detection performance of MFDroid. At the same time, we performed statistical analysis on our dataset. The results show that MFDroid proposed in this study is effective, which can significantly distinguish malicious applications from benign applications. The main contributions of this paper are as follows:

1. We constructed a real-world Android malware dataset and conducted a comprehensive analysis for the dataset, showing the differences between malicious and benign applications.
2. We combined the results of seven feature selection algorithms and fed them into the model for training. The results show that the F1-score of our method is significantly higher than other feature selection algorithms.
3. We propose an Android malware detection framework based on stacking ensemble learning—MFDroid.

The remainder of this paper is organized as follows: Section 2 presents related work and the current state of research on dataset construction, static analysis methods, machine learning, and ensemble learning. The details of the dataset are described in Section 3. In Section 4, we present the proposed framework and method in detail, and then in Section 5, we analyze the experimental results on our dataset with our proposed method. Our conclusions are summarized in Section 6.

2. Related Work

Due to the increase of malware on smart devices, Android malware detection has become an important research topic, and extensive studies have been produced. Concerning the general process of Android malware detection, including dataset construction, static feature extraction, feature selection, machine learning methods, and ensemble learning methods, we review the related work in the field of malware detection.

2.1. Construction of Dataset

In recent years, the permissions requested by Android malware have been very close to benign applications. Older datasets can no longer meet the needs of accurate detection, so it is necessary to build a newer dataset for malware detection.

Arp et al. [7] presented a dataset of real Android applications and real malware, and they collected an initial dataset of 131,611 apps. The samples were collected between August 2010 and October 2012.

Allix et al. [8] constructed AndroZoo, an Android application dataset of over 17,854,943 apps, including Google play, each of which was analyzed by dozens of different antivirus products to mark malware.

Martín et al. [9] constructed a large comprehensive feature dataset consisting of 22,000 real malware and benign applications. Their work could help anti-malware tool researchers develop new mechanisms and tools for Android malware detection.

Given the early creation of the above datasets, it is not possible to conduct research on the latest Android application security situation. Therefore, we create a new dataset containing APKs collected during 2019–2020.

2.2. Static Analysis

Static analysis involves utilizing decompilation tools to extract features through lexical analysis, semantic analysis, etc., of applications. APK files contain many features, such as permissions, API calls, signatures, network addresses, and hardware structures, etc., which can be used as a basis for judging whether an Android application is malicious.

Li et al. [10] proposed a malware detection system SIGPID based on permission usage analysis, and developed three levels of pruning by mining permission data to identify important permissions that could effectively distinguish benign applications from malicious applications. SIGPID utilized machine learning-based classification methods to classify different families of malware and benign applications.

Tao et al. [11] proposed MalPat, an automatic malware detection system that automatically extracts key features from tens of thousands of applications to help the Android app market fight malware. Using statistical analysis techniques, they delved into different usages of permissions and APIs and revealed their ability to distinguish malware from benign applications. MalPat took full advantage of the characteristics of machine learning methods and used it earlier in the system architecture to learn the latent features of existing data.

Alazab et al. [12] proposed an efficient classification model that combined permission requests and API calls. Three different grouping strategies were proposed to select the most valuable API calls to maximize the likelihood of identifying Android malware applications. An in-depth analysis of different public and private packages, classes, and methods was performed to evaluate the effectiveness and accuracy of the proposed method when dealing with large datasets.

Although the above research uses the fine-grained feature of API call, it does not use Opcode, which is also a fine-grained feature, as one of the features. Some studies have shown that incorporating Opcode into the feature sets for experiments is helpful to understand the complexity and behavior of applications [13]. Therefore, this paper adopts three features: permission, API call, and Opcode.

2.3. Machine Learning

A machine learning classification model can further discover features and hidden rules between features and make predictions by fitting known data.

Cai et al. [14] proposed a novel feature-weighted-based Android malware detection scheme that combined the optimization of weight mapping functions and classifier parameters, called JOWMDroid. First, eight categories of features were extracted from the Android application package, and then the information gain was used to select a certain number of the most important features for malware detection. Next, initial weights were calculated for each selected feature through three machine learning models, and then five weight mapping functions were designed to map the initial weights to the final weights. Finally, the parameters of the weight mapping function and the classifier were jointly optimized by the differential evolution algorithm.

Mahindru et al. [15] proposed an effective and efficient web-based Android malware detection solution, MLDroid, capable of detecting malware applications based on permissions and API calls. They implemented different supervised, unsupervised, semi-supervised, and hybrid machine learning algorithms to train MLDroid, enabling it to achieve higher detection rates.

Jannat et al. [16] proposed a system to analyze and detect Android malware using machine learning. They solved problems in two ways—dynamic analysis and static analysis. The best results were obtained in dynamic analysis using the random forest (RF) algorithm, which is an extended variant of the decision tree (DT) algorithm. In addition, researchers used various datasets for static and dynamic studies.

Utilizing only one machine learning algorithm makes the experiment subject to skewed data, and the computational overhead is relatively large. In addition, it may be affected by the initial setting, is sensitive to noise, and cannot handle high-dimensional features well. Therefore, we used an ensemble learning approach to overcome the limitations of a single machine learning algorithm, thereby improving the accuracy of malware detection.

2.4. Ensemble Learning

Ensemble learning involves combining multiple weakly supervised models to get a better (and more comprehensive) strongly supervised model. The key idea of ensemble learning is that, even if a weak classifier gets a wrong prediction, other weak classifiers can correct the error.

Aboaoja et al. [17] proposed an ensemble behavior-based early evasion malware detection framework. The developed framework consists of three main stages, evasion behavior collection, correlation-based feature extraction and selection, and model building stage. The framework was able to efficiently identify complex malware behaviors using ensemble learning methods and make final decisions based on the results of the majority voting strategy.

Zhu et al. [18] proposed a stacking integration framework, SEDMDroid, to identify Android malware. Principal component analysis was performed on each feature subset to detect accuracy by retaining all principal components and using the entire dataset to train each base learner multilayer perceptron (MLP). Then, a support vector machine (SVM) was used as a fusion classifier to learn implicit supplementary information from the outputs of ensemble members and produce final predictions.

Idress et al. [19] proposed a new permission- and intent-based framework, PIndroid, to identify Android malware applications. PIndroid was the first solution to accurately detect malware using a combination of permissions and intent, complemented by an ensemble approach. The authors applied statistical significance tests to investigate the correlation between permissions and intent and found statistical evidence of a strong correlation between permissions and intent that could be leveraged to detect malware applications.

Rana et al. [20] proposed and evaluated various machine learning algorithms by applying an ensemble-based learning approach to identify Android malware associated with a substring-based classifier feature selection (SBFS) strategy. They used the DREBIN dataset and achieved better results with an ensemble learning approach.

Given that the ensemble method combines the results of multiple machine learning algorithms, it can improve the prediction performance considerably. We construct a stacking ensemble learning framework composed of five base learners and a meta classifier to achieve effective and efficient malware detection. The stacking ensemble method utilized in our paper uses all of the training data to perform k-fold cross-validation for multiple models in the first layer, so that each model has a predicted value, and then it trains the models in the second layer with these predicted values as new features.

3. Dataset

This study explored the statuses of Android malware applications, so we constructed a real-world Android malware dataset. From 2019 to 2020, we collected a total of 1664 Android apps in 15 categories from 4 app markets in China: Huawei application market (<https://appstore.huawei.com/> accessed on 30 November 2019), Xiaomi application market (<https://app.mi.com/> accessed on 1 February 2020), 360 application market (<https://ext.se.360.cn/> accessed on 2 March 2020), and the Wandoujia application market (<https://www.wandoujia.com/> accessed on 5 May 2020), including 806 malicious apps and

858 benign apps (Table 1). The 15 categories were finance and economics, home and life, chatting and socializing, travel and transportation, photography and videography, fashion and shopping, practical tools, video and music, sports, book reading, efficient office, news and information, learning and education, health care and entertainment. We identified whether the collected Android applications were malicious or benign by using the VirusTotal web tool, which was an online malware detection tool that used multiple anti-virus engines to detect uploaded APK files, and returned detection reports to determine whether the files were infected by viruses, worms, trojans, and various types of malware [21]. We marked APK files according to the number of engines in the report provided by VirusTotal, when an APK file was detected as malicious by at least three engines, it was marked as malicious, otherwise it was benign. The specific features of the dataset are shown in Table 2.

Table 1. Number of applications.

Category	Count
Malware	806
Benign	858
Total	1664

Table 2. Number of dataset features.

Feature	Count
Permission	146
Opcode	218
API call	3183
Activity	4071
Service	3219
Receiver	4633
apiPackage	212

4. MFDroid

In this section, we introduce the proposed Android malware detection framework in detail, as is shown in Figure 1, which is divided into three stages: feature preprocessing, feature selection, and stacking ensemble method.

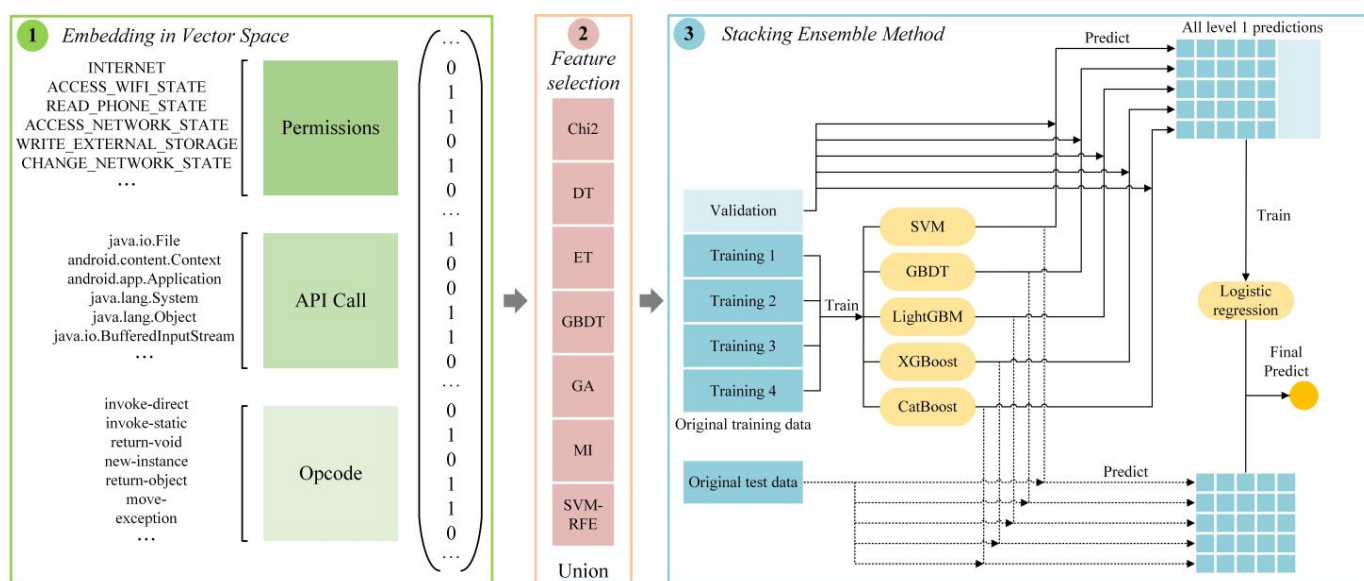


Figure 1. The architecture of MFDroid.

4.1. Data Preprocessing

We preprocess the data by utilizing AndroPyTool, a tool for extracting static and dynamic features from APK, which combines various well-known Android application analysis tools such as DroidBox, FlowDroid, Strace, and AndroGuard. AndroPyTool uses the above tools to perform pre-static analysis, static analysis, and dynamic analysis on APK files, and generate feature files in JSON format [9]. As shown in Figure 2, the first step is to check whether the sample is a valid APK file and filter out the corrupted APK files that cannot be decompiled. The second step uses AndroPyTool to extract the feature information of each sample and store it in JSON format for subsequent feature extraction and vectorization. The third step uses regular expressions to extract features, such as permissions, opcodes, and API calls from the JSON file generated by AndroPyTool. The last step is to vectorize the extracted features. If a certain bit of the feature vector is 0, it means that the feature does not appear in the APK file, and if it is 1, the opposite is true. A total of 3547 features, including permissions, opcodes and API calls, were mapped to the vector space for subsequent feature selection. Detailed feature information is shown in Table 3.

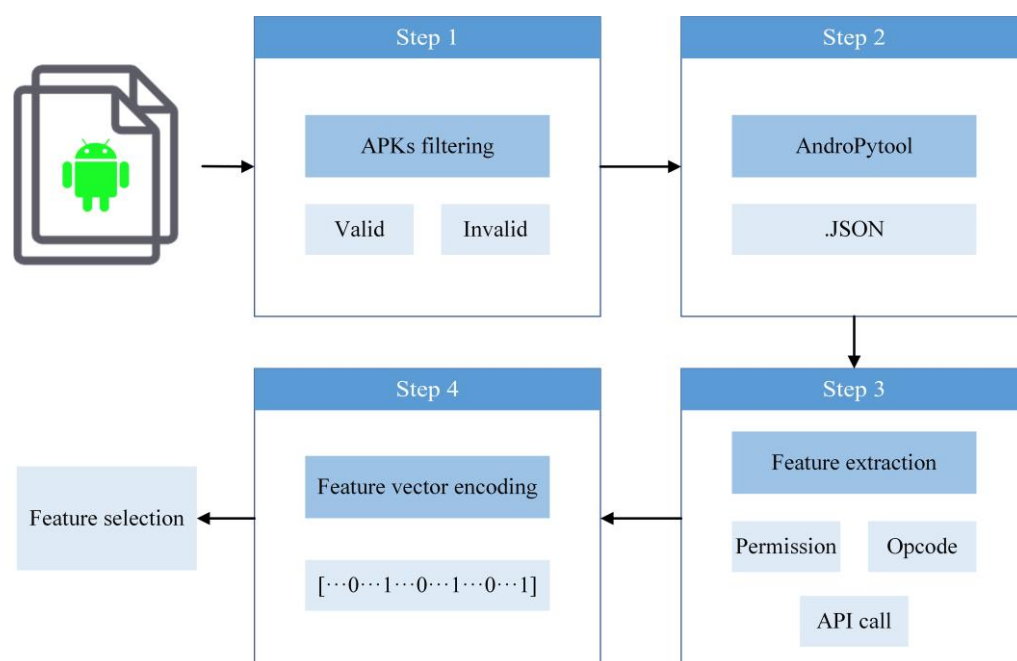


Figure 2. The process of data preprocessing.

Table 3. Representation of feature vectors.

Feature	Category	Amount	Combined Vector
Permission	0–1	146	The three features formed a 3547-dimensional vector
Opcode	0–1	218	
API call	0–1	3183	

4.2. Feature Selection Methods

In this section, we elaborate on the seven feature selection algorithms utilized by MFDroid and introduce the feature selection strategy used in this study.

4.2.1. Feature Selection Algorithms

The tree-based feature selection method is an information gain algorithm. The principle is that—the more the same types of leaf nodes are contained in a feature, the more significant the feature is in training. The tree model can calculate the feature value through

learning and training importance and calculate the feature contribution index, so it can be used to remove irrelevant features. In this study, we used three tree-based feature selection algorithms: decision tree (DT), gradient boosting decision tree (GBDT), and extra-trees (ET).

Chi-square test (Chi2) is a hypothesis testing method in which the distribution of a statistic approximately obeys the chi-square distribution when the null hypothesis is established, and it is used to determine whether two variables are independent [22]. The correlation between categorical variables can be judged by calculating the chi-square value of two categorical variables. The larger the chi-square value, the greater the relationship between the two categorical variables and the lower the independence; otherwise, the smaller the relationship between the two categorical variables, the higher the independence. When the chi-square value reaches 0, it means that the factors are the same.

Genetic algorithm (GA) is a search and optimization technology inspired by the biological evolution process [23,24]. Based on the rule of survival of the fittest, it searches for the optimal solution through various genetic operations. The population in the genetic algorithm represents the possible solution set of the problem, and a population is composed of a certain number of individuals encoded by genes, and each individual corresponds to a possible solution to the problem. The genetic algorithm will randomly generate the initial population according to different coding methods, and iteratively evolve according to the principle of survival of the fittest until a suitable solution set is found.

The support vector machine based on recursive feature elimination (SVM-RFE) feature selection algorithm is a backward recursive elimination feature selection algorithm that uses the classification performance of the support vector machine as the feature selection evaluation standard, and has high recognition performance [25]. The SVM-RFE method removes the features with the smallest ranking coefficient through continuous iteration, then uses SVM to retrain the remaining features to obtain new feature rankings, and finally obtains a sorted list of features. Using the sorted list, define several nested feature subsets $F_1 \subset F_2 \subset \dots \subset F$ to train the SVM, and evaluate the pros and cons of these subsets with the SVM prediction accuracy rate, to obtain the optimal feature subset.

The degree of correlation between different variables can be measured by mutual information (MI) [26]. The smaller the mutual information, the less common information, and the more independent the two variables are; on the contrary, the more common information, the more related the two variables are. According to the definition of mutual information, if and only when two random variables are independent of each other, the mutual information is 0; when the two variables are highly correlated, the mutual information of the two variables will also be large.

4.2.2. Union of Feature Selection Results

On the feature vector space after data preprocessing, seven feature selection algorithms, including ET, GBDT, DT, Chi2, GA, SVM-RFE, and MI were applied. We extracted the feature index dictionary stored by each feature selection algorithm, took the union to combine the selection results of these single feature selection algorithms, stored the combined result as a dictionary, and selected from the original feature vector space according to the feature index in the dictionary. The final feature set was used in this study.

As shown in Table 4, each feature selection algorithm selected a different number of features. Therefore, the single feature selection algorithm has limitations, such as some important features may be omitted. This study utilizes the union strategy to merge the results of the 7 feature selection algorithms, and finally 2376 features remain, which are feed to the model for training. In addition, we also use the intersection strategy to test, and the results show that the feature set generated by the intersection strategy was not effective in the experiment. Because different feature selection algorithms have different internal mechanisms, the results of each feature selection algorithm are quite different. Therefore, it is necessary to choose union strategy.

Table 4. The number of features selected by each feature selection algorithm.

Algorithm	ET	GBDT	DT	Chi2	GA	SVM-RFE	MI
Amount	1452	1319	1500	1500	1687	1893	1366

4.3. Stacking Ensemble Method

4.3.1. Base Learners

This study used SVM, GBDT, XGBoost, LightGBM, and CatBoost as base learners, and used the new feature set generated by the union strategy to train each base learner to guarantee accuracy.

SVM is often used to solve binary classification problems [27]. Given a sample training set (x_i, d_i) , $i = 1, 2, \dots, N$, $x_i \in R^n$, $d_i \in \{\pm 1\}$, the principle of SVM is to try to find a hyperplane $(w \cdot x) + b = 0$, $x, w \in R^n$, $b \in R$, so that the hyperplane satisfies the classification requirements; that is, find the optimal value of the weight vector w and the bias b , so as to satisfy the formula:

$$d_i(w^T x_i + b) \geq 1 - \xi_i \quad i = 1, 2, \dots, N \quad (1)$$

where slack variable $\xi_i \geq 0$, $i = 1, 2, \dots, N$.

GBDT is a gradient boosting integration method, which is a method that adopts boosting technology. The main idea of GBDT is to follow the gradient descent direction of the loss function of the previously established model every time a model is built. If the classifier model can be built in the direction that the loss function continues to decrease, the performance of the classifier model improves. By minimizing the loss function $L(\theta)$, parameter θ is obtained, and its calculation formula is shown in the formula.

$$\theta = \theta - \alpha \cdot \frac{\partial}{\partial \theta} L(\theta) \quad (2)$$

In the m -th iteration of GBDT, the classifier parameters of the first $m-1$ iterations are fixed. Therefore, in the m -th iteration, only the loss function of the m -th classifier needs to be minimized to obtain the corresponding classifier.

The XGBoost algorithm is an evolutionary form of the GBDT algorithm [28]. Its base learner usually selects a decision tree model and generates a new tree through continuous iteration to learn the residual between the actual value and the predicted value of all current trees and accumulate the results of all trees as the result, to get the highest possible classification accuracy. The objective function of the XGBoost algorithm is shown in Formula (3):

$$obj = \sum_{i=1}^n l(y_i, \hat{y}_i) + \sum_{k=1}^K \theta(f_k) = \sum_{i=1}^n \left[f_t(x_i) g_i + \frac{1}{2} (f_t(x_i))^2 h_i \right] + \theta(f_t) \quad (3)$$

where g_i and h_i are the first and second derivatives of the loss function, respectively, and $\theta(f_t)$ is the structure of the t -th tree.

The LightGBM algorithm is another evolutionary form of the GBDT algorithm [29]. The algorithm uses a depth-limited leaf-wise strategy to find the node with the largest gain value from the current leaf nodes to split; it limits the depth of the tree to prevent overfitting and reduces the time to find the optimal depth tree. At the same time, when the number of splits is the same, the error can be reduced and a higher precision can be obtained. In the process of building a tree, the process of finding the optimal splitting node is the most wasteful in terms of time and computer resources. For this, LightGBM uses histogram algorithm, gradient based one-side sampling, mutually exclusive features, and the binding algorithm (exclusive feature building, EFB) to improve operating efficiency.

CatBoost is a machine learning framework based on GBDT, which is designed to solve the discrete feature problem existing in GBDT [30]. CatBoost uses the oblivious tree as the base predictor, and the index of the leaf node of the oblivious tree is converted into a binary

vector whose length is the depth of the tree. CatBoost performs binarization operations on various types of features and statistical information, and then inputs the binary features into the model to calculate the predicted value. The binary features are stored in vector B , and the value of leaf nodes is stored in a vector of size 2^d , d is the depth of the tree. For sample x , its binary vector representation is:

$$\sum_{i=0}^{d-1} 2^i \cdot B(x, f(t, j)) \quad (4)$$

where $B(x, f)$ represents the binary-type feature value of the sample x obtained from the vector B , and $f(t, j)$ represents the numerical value of the binary feature in the t -th tree.

4.3.2. Meta-Classifer

In our study, logistic regression is used as a meta-classifier, and logistic regression is proposed to deal with classification problems. Traditional linear regression only implements regression, and its basic form is as follows:

$$y = \omega^T x + b \quad (5)$$

The predicted value produced by linear regression is a series of real values, but to solve a classification problem, the predicted value should be 0 or 1, so it is further transformed with the help of the continuous function Sigmoid:

$$y = \frac{1}{1 + e^{-x}} \quad (6)$$

The sigmoid function can convert the prediction result of linear regression into the conditional probability that the sample belongs to a certain category, and then with the help of the preset classification threshold, the classification result can be obtained. For the two-class problem, by incorporating b into the weight vector ω , the probability that the sample belongs to the positive and negative classes can be obtained:

$$P(Y = 1|x) = \frac{\exp(\omega \cdot x)}{1 + \exp(\omega \cdot x)} \quad (7)$$

$$P(Y = 0|x) = \frac{1}{1 + \exp(\omega \cdot x)} \quad (8)$$

Equations (7) and (8) are logistic regression models. According to Equation (7), when the predicted value of the linear function tends to positive infinity, the probability value of the predicted sample being a positive class is close to 1, otherwise it is close to 0.

4.3.3. Ensemble of Classifiers

The stacking ensemble learning prediction method is shown in Figure 1. The base learner is composed of each single prediction model. First, the original dataset is used to train multiple base learners. In the training process, to reduce the risk of model overfitting, K -fold cross-validation is generally used to train base learners. Then, the prediction results of the base learner are formed into a new dataset, and then the secondary learners are trained to obtain the final prediction results [18]. The specific steps of the algorithm are as follows:

1. Divide the original dataset into two parts: the original training set D and the original testing set T .
2. Perform K -fold cross-validation on the base learner: randomly divide the original training set D into K equivalents ($D_1, D_2, \dots, D_k$), and each base learner uses one of them as the K -fold test set, and the rest the $K-1$ copies are used as the K -fold training set. Each base learner is trained using the K -fold training set, and predictions are made

on the K -fold test set, and the prediction results of each base learner are combined as the training set $\tilde{D}$ of the secondary learner.

3. Each base learner makes predictions on the original test set T , and the prediction results are averaged as the validation set $\tilde{T}$ of the secondary learners.
4. The secondary learner obtains the new dataset generated from the base learner: the training set $\tilde{D}$ and the validation set $\tilde{T}$, and then performs learning and training, and outputs the final prediction result.

The stacking ensemble learning prediction method uses K -fold cross-validation to reduce the risk of overfitting of the model and uses the prediction results of multiple base learners to perform secondary training (see Algorithm 1). This method can overcome the limitations of a single learner and integrate variously; it is a machine learning method that improves the accuracy and generalization of prediction results.

In stacking the ensemble learning prediction method, the base learner can choose a strong learner, and the secondary learner can choose a simple learner, which could make the fusion effect better and avoid overfitting.

Algorithm 1 Stacking with K -fold Cross Validation

Input: training data $\mathcal{D} = \{X_i, y_i\}_{i=1}^m$ ($X_i \in \mathbb{R}^n, y_i \in Y$)

Output: an ensemble classifier H

1: Step 1: adopt cross validation approach in preparing a training set for second-level classifier

2: Randomly split $\mathcal{D}$ into K equal-size subsets: $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_K\}$

3: for $k = 1$ to K do

4: Step 1.1: learn first-level classifiers

5: for $t = 1$ to T do

6: Learn a classifier h_{kt} from $\mathcal{D} \setminus \mathcal{D}_K$

7: end for

8: Step 1.2: construct a training set for second-level classifier

9: for $X_i \in \mathcal{D}_K$ do

10: Get a record $\{X'_i, y_i\}$, where $X'_i = \{h_{k1}(X_i), h_{k2}(X_i), \dots, h_{kT}(X_i)\}$

11: end for

12: end for

13: Step 2: learn a second-level classifier

14: Learn a new classifier h' from the collection of $\{X'_i, y_i\}$

15: Step 3: relearn first-level classifiers

16: for $t = 1$ to T do

17: Learn a classifier h_t

18: end for

19: return $H(X) = h'(h_1(X), h_2(X), \dots, h_T(X))$

4.4. Evaluation Metrics

To train the model, we split the data, of which 75% was used for training and 25% for testing. To evaluate the performance of MFDroid, some evaluation criteria were introduced, such as confusion matrix, accuracy, precision, recall, F1-score, and ROC curves. In this study, the binary classification machine learning results were classified by the confusion matrix, as shown in Table 5. The confusion matrix includes information about the predicted classification results based on machine learning and the actual classification results.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \quad (9)$$

$$Precision = \frac{TP}{TP + FP} \quad (10)$$

$$Recall = \frac{TP}{TP + FN} \quad (11)$$

$$F1 - score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (12)$$

where TP is the amount of malware apps correctly detected, FP is the amount of benign apps wrongly predicted as malware, FN is the amount of malware cases misclassified as benign software, and TN is the amount of benign apps correctly detected.

Table 5. Confusion matrix.

Predicted	Actual	Positive	Negative
Positive		True Positive (TP)	False Positive (FP)
Negative		False Negative (FN)	True Negative (TN)

5. Experiments and Results

5.1. Dataset Analysis

In this section, we analyze different types of features. Evaluating datasets from different feature perspectives is a useful mechanism to identify differences in features between malicious and benign applications and to draw general conclusions.

5.1.1. Permissions Required Analysis

Table 6 shows the top 20 permissions used by malicious applications and benign applications. Most samples of different types need to apply for INTERNET, ACCESS_NETWORK_STATE and WRITE_EXTERNAL_STORAGE permissions, so there is no obvious difference. The WRITE_SETTINGS permission allows programs to read or write system settings. Moreover, 79% of malicious apps have applied for this permission, and 64% of benign apps have applied for this permission. In particular, our research analyzes 24 dangerous permissions officially announced by Android and counts the application frequency of dangerous permissions in malicious applications and benign applications. As can be seen from Figure 3, there are three permissions related to SMS messages, the SEND_SMS permission allows the application to send SMS messages, the RECEIVE_SMS permission allows the application to receive SMS messages, and the READ_SMS permission allows the application to read SMS messages. These three permissions show obvious differences in different types of applications, and the frequency of malicious applications applying for SMS message-related permissions is significantly higher than that of benign applications. As we all know, some malicious applications may cause serious property damage or privacy leakage to users by stealing SMS messages, such as verification codes and other private messages. At the same time, the PROCESS_OUTGOING_CALLS permission, which allows applications to view the number being dialed during an outgoing call and can choose to redirect the call to another number or abort the call entirely, is significantly more frequently applied in malicious apps than in benign ones. The official Android documentation marks its protection level as dangerous, and it can be used to access very sensitive information.

It is worth noting that, whether it is ordinary permissions or dangerous permissions, the frequency of permission applications for malicious applications and benign applications is not significantly different, and further analysis shows that the use of specific permissions is also very similar. Therefore, it is difficult to achieve the detection effect in the past only by relying on permissions as a feature of machine learning algorithms. More fine-grained API calls, Opcode, etc., are required as features to participate in the detection. Various types of application features are conducive to improving the performance of detection.

5.1.2. API Calls Analysis

The mechanisms inherent in API calls make it possible to identify relevant differences between applications with good or illegal intents. In this study, we selected some typical API calls for statistical analysis according to the common attack types of Android malware. As shown in Table 7, among the Android/telephony/TelephonyManager API calls, 87% of malicious applications call this package, on the contrary, only 46% of benign applications call this package. In permissions analysis, we observed that phone-related services were used more frequently in malware samples.

Table 6. Top 20 most common permissions in malicious and benign apps.

Malware		Benign	
Permission	%	Permission	%
INTERNET	100%	INTERNET	99%
ACCESS_NETWORK_STATE	100%	WRITE_EXTERNAL_STORAGE	99%
WRITE_EXTERNAL_STORAGE	100%	ACCESS_NETWORK_STATE	99%
ACCESS_WIFI_STATE	100%	ACCESS_WIFI_STATE	94%
READ_PHONE_STATE	99%	READ_PHONE_STATE	90%
READ_EXTERNAL_STORAGE	94%	READ_EXTERNAL_STORAGE	86%
WAKE_LOCK	90%	WAKE_LOCK	82%
ACCESS_COARSE_LOCATION	89%	VIBRATE	81%
GET_TASKS	88%	CAMERA	80%
VIBRATE	88%	ACCESS_COARSE_LOCATION	77%
CHANGE_WIFI_STATE	87%	ACCESS_FINE_LOCATION	76%
ACCESS_FINE_LOCATION	86%	CHANGE_WIFI_STATE	72%
CAMERA	82%	GET_TASKS	71%
WRITE_SETTINGS	79%	RECEIVE_BOOT_COMPLETED	65%
SYSTEM_ALERT_WINDOW	76%	REQUEST_INSTALL_PACKAGES	64%
MOUNT_UNMOUNT_FILESYSTEMS	74%	WRITE_SETTINGS	64%
RECEIVE_BOOT_COMPLETED	72%	RECORD_AUDIO	61%
CHANGE_NETWORK_STATE	70%	SYSTEM_ALERT_WINDOW	60%
REQUEST_INSTALL_PACKAGES	67%	MOUNT_UNMOUNT_FILESYSTEMS	59%
RECORD_AUDIO	66%	CHANGE_NETWORK_STATE	58%

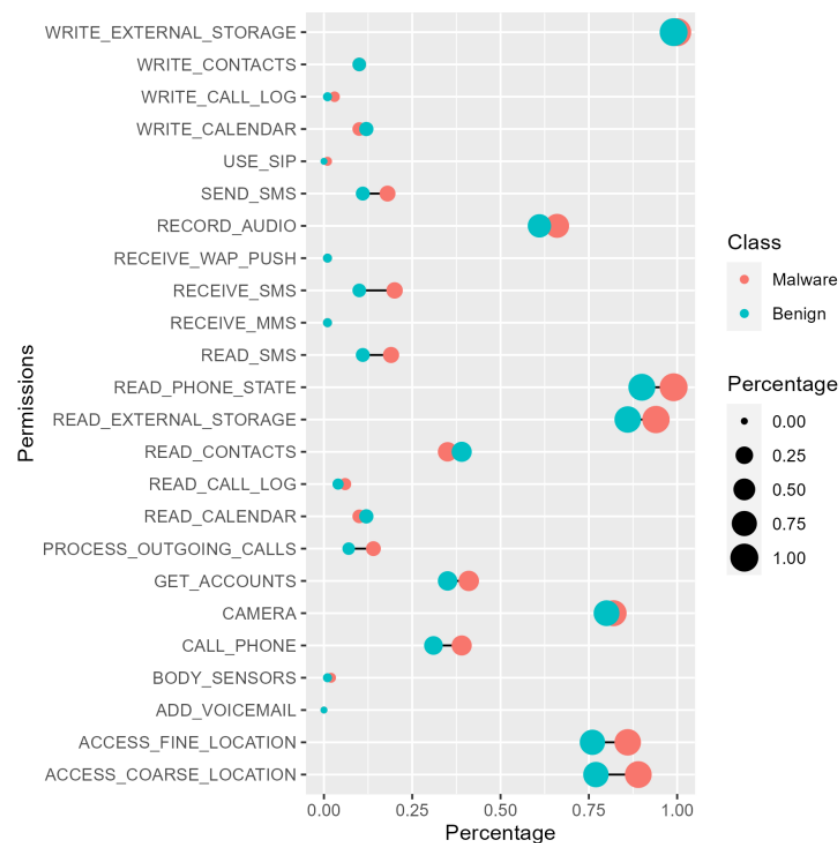
**Figure 3.** A total of 24 dangerous permissions in the percentages of malicious applications and benign applications, respectively.

Table 7. Percentage of common API calls by attack type in malicious and benign applications.

Attacks	API Calls	Malware	Benign
Telephone	Android/telephony/TelephonyManager	87%	46%
Location	Android/telephony/gsm/GsmCellLocation	62%	34%
	Android/location/Address	80%	38%
	Android/location/Location	80%	39%
	Android/location/LocationManager	77%	39%
Camera	Android/hardware/Camera	57%	29%
Storage	Android/os/Environment	90%	45%
	Android/content/Context/getExternalFilesDir	75%	36%
	Java/IO/ByteArrayOutputStream	94%	47%

5.1.3. Opcodes Analysis

In this subsection, we examine the use of opcodes. Table 8 is the top 20 opcodes used by malicious applications and benign applications. Typically, an opcode-based analysis of the frequency of use does not yield a relevant conclusive assessment. Our dataset demonstrates that there is no large difference between the top 20 frequently used opcodes for malicious and benign applications. However, opcodes are instructions executed by applications that must be used by all applications to perform scheduled activities. Therefore, it is of great significance to take opcode as one of the characteristics of detecting Android malware.

Table 8. Top 20 most common opcodes in malicious and benign apps.

Malware		Benign	
Opcode	%	Opcode	%
const/16	100%	const-wide/16	100%
const/4	100%	const/16	100%
goto	100%	const/4	100%
if-eqz	100%	goto	100%
if-nez	100%	if-eqz	100%
invoke-direct	100%	if-nez	100%
invoke-static	100%	invoke-direct	100%
invoke-super	100%	invoke-static	100%
invoke-virtual	100%	invoke-super	100%
move-exception	100%	invoke-virtual	100%
move-result	100%	move-exception	100%
move-result-object	100%	move-result	100%
new-array	100%	move-result-object	100%
new-instance	100%	new-array	100%
return-object	100%	new-instance	100%
return-void	100%	return-void	100%
sget-object	100%	sget-object	100%
sput-object	100%	sput-object	100%
aput-object	100%	throw	100%
const-wide/16	100%	move-object	100%

5.2. Analysis of Feature Selection Results

Our study used eight feature selection methods, namely ET, GBDT, DT, Chi2, GA, SVM-RFE, MI, and non-selection (NS), experiments with MFDroid, each feature selection algorithm selected a different number of features. It can be seen from Figure 4 and Table 9 that the experimental results of the single feature selection algorithm are not significantly different (see Figures 5 and 6). It is worth noting that the number of features selected by ET is small but its experimental results are high. GA has the largest number of features selected among the seven feature selection algorithms, and the experimental results are the best. Considering the limitations of the single feature selection algorithm, some important features may be missed. Therefore, this study adopts the union strategy to merge the results of the seven feature selection algorithms, and finally retains 3547 features and sends

them to the model for training. Experiments show that the F1-score of the union strategy reaches 96.0%, which is 2.3% higher than that of GA, and the accuracy value is 2.4% higher. Therefore, the feature selection method using the union strategy is accurate and effective.

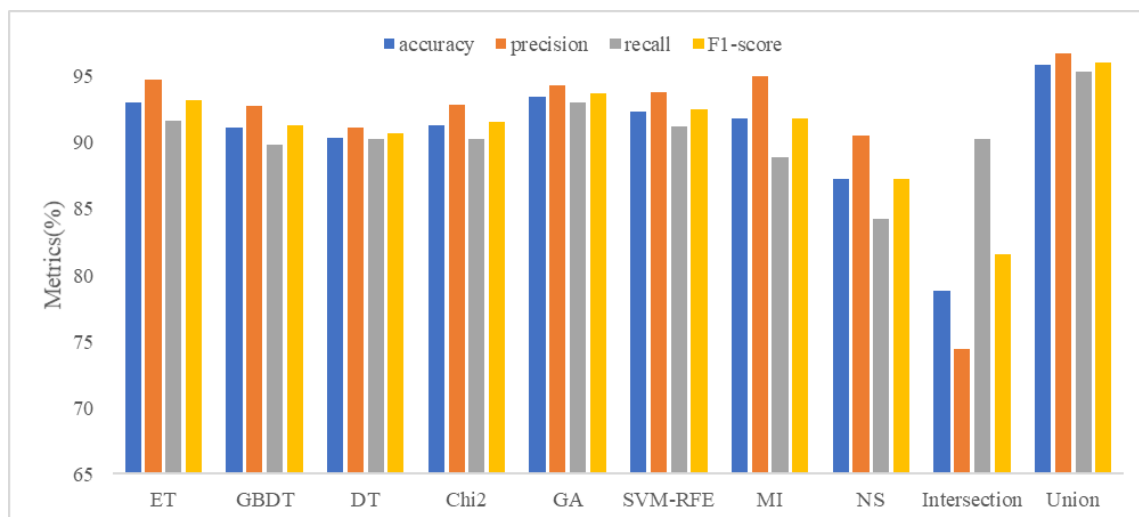


Figure 4. Feature selection results comparison on our dataset.

Table 9. Evaluation metrics for different feature selection methods on ours and OmniDroid.

No.	Algorithm	Accuracy		Precision		Recall		F1-Score	
		Ours	OmniDroid	Ours	OmniDroid	Ours	OmniDroid	Ours	OmniDroid
1	ET	0.930	0.901	0.947	0.896	0.917	0.907	0.932	0.902
2	GBDT	0.911	0.899	0.928	0.905	0.898	0.892	0.913	0.898
3	DT	0.904	0.883	0.911	0.886	0.903	0.879	0.908	0.882
4	Chi2	0.914	0.902	0.929	0.893	0.903	0.913	0.916	0.903
5	GA	0.935	0.920	0.944	0.922	0.931	0.916	0.937	0.919
6	SVM-RFE	0.923	0.883	0.938	0.874	0.927	0.895	0.925	0.884
7	MI	0.918	0.865	0.951	0.863	0.889	0.867	0.919	0.865
8	NS	0.873	0.844	0.905	0.851	0.843	0.835	0.873	0.843
9	Intersection	0.788	0.809	0.744	0.758	0.903	0.908	0.815	0.882
10	Union	0.959	0.934	0.967	0.941	0.954	0.926	0.960	0.933

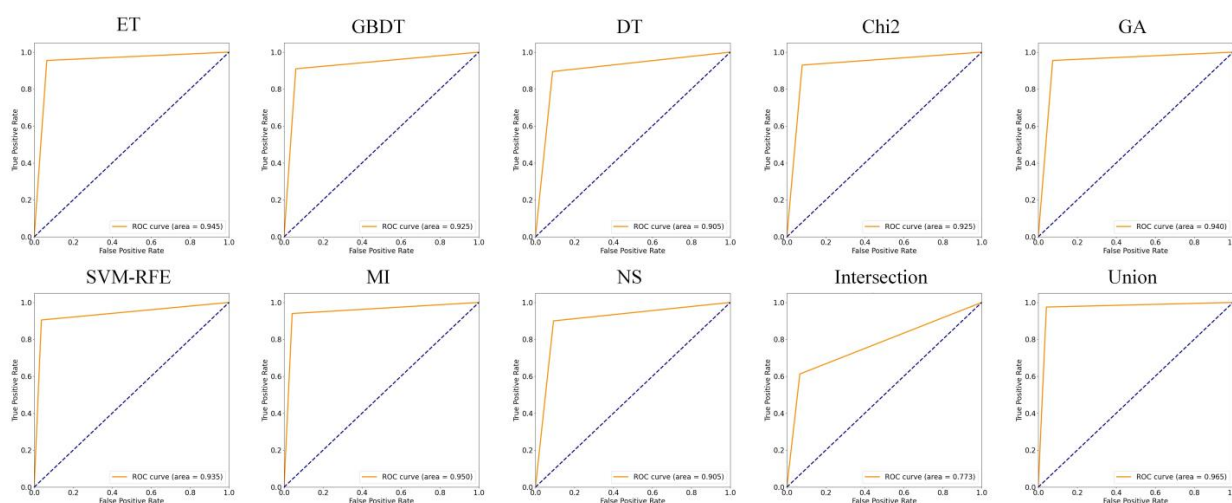


Figure 5. ROC curves of different feature selection methods on our dataset.

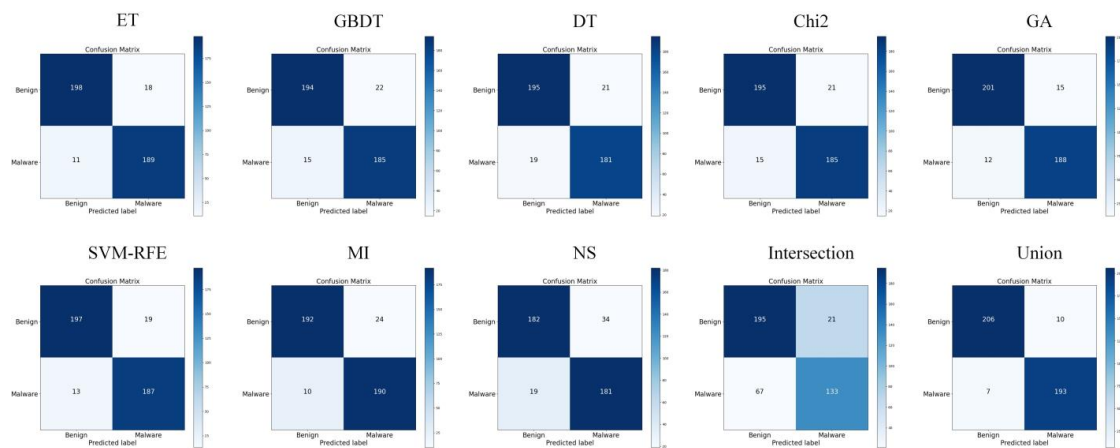


Figure 6. Confusion matrix of different feature selection methods on our dataset.

To compare our work more easily with other existing studies, we introduce the OmniDroid dataset to test our model (see Figures 7–9). As shown in Table 9, our adopted feature selection union strategy outperforms other feature selection algorithms and our model outperforms other comparative methods on OmniDroid. Compared with [31], our method performs better on the static features of OmniDroid.

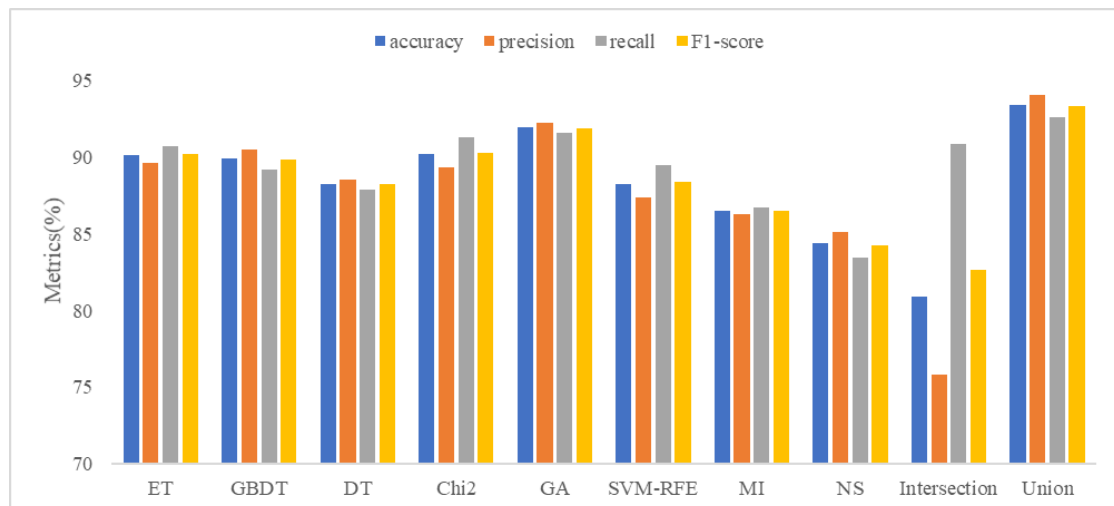


Figure 7. Feature selection results comparison on OmniDroid dataset.

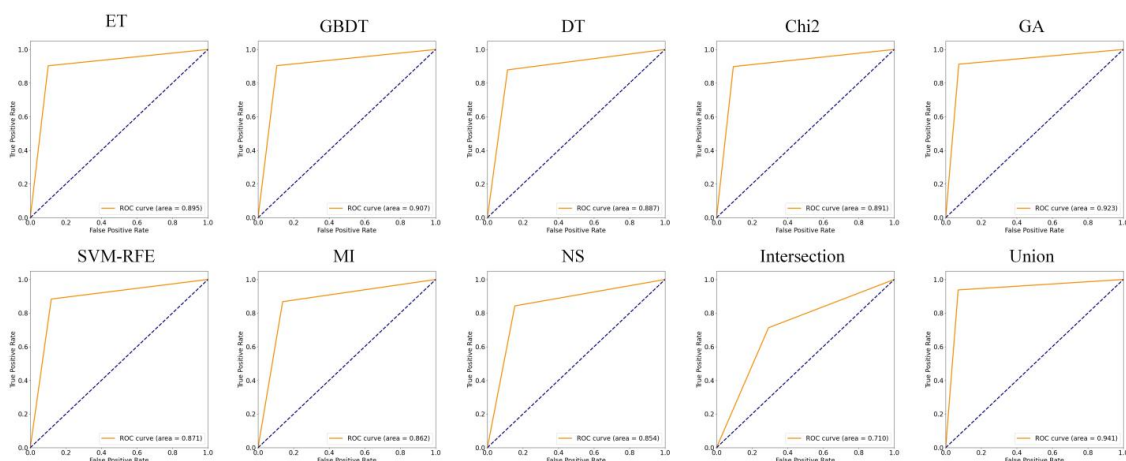


Figure 8. ROC curves of different feature selection methods on OmniDroid dataset.

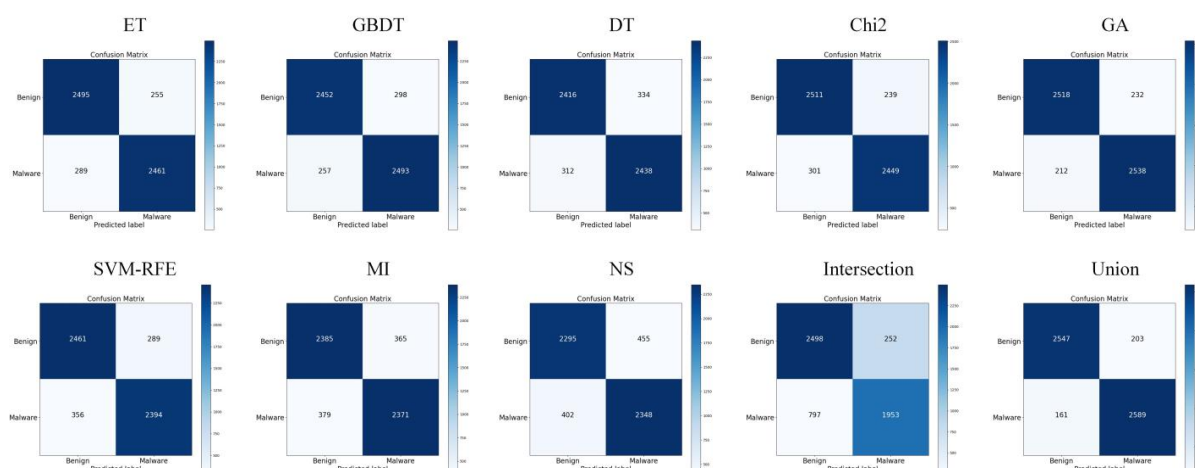


Figure 9. Confusion matrix of different feature selection methods on OmniDroid dataset.

5.3. Detection Performance Evaluation of MFDroid

To verify the effect of the multi-model fusion framework MFDroid proposed in this study in the field of Android malware detection, combined with the five-fold cross-validation method, SVM, GBDT, XGBoost, LightGBM, CatBoost, and MFDroid proposed in this study were compared. The experimental results are shown in Figure 10 and Table 10.

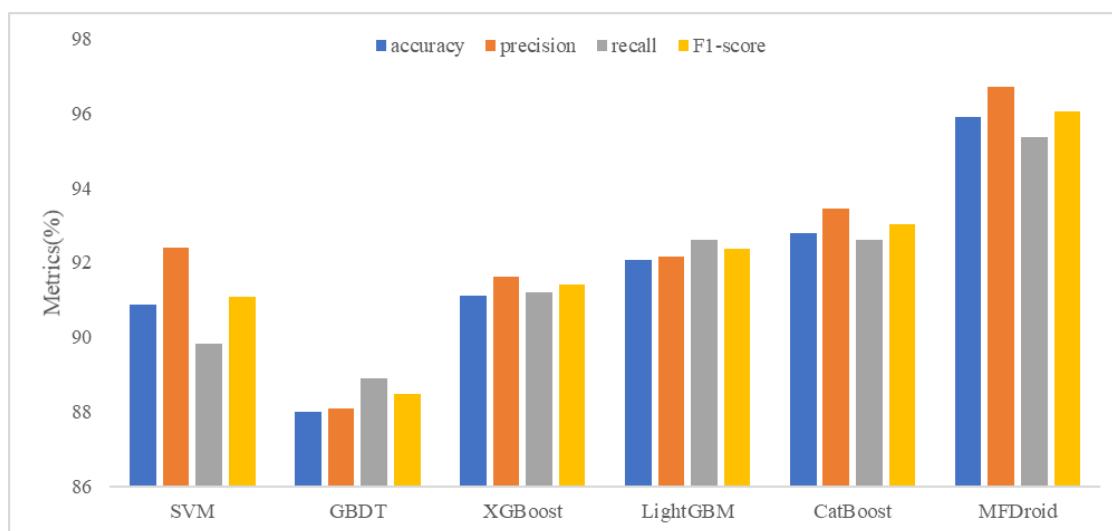


Figure 10. Comparison of results between MFDroid and single classifier.

Table 10. Evaluation metrics of MFDroid and single classifier.

No.	Model	Accuracy	Precision	Recall	F1-Score
1	SVM	0.908	0.924	0.898	0.911
2	GBDT	0.880	0.881	0.889	0.886
3	XGBoost	0.911	0.916	0.912	0.914
4	LightGBM	0.921	0.922	0.926	0.924
5	CatBoost	0.928	0.935	0.926	0.930
6	MFDroid	0.959	0.967	0.954	0.960

Through comparison, it is found that the evaluation index of MFDroid is significantly higher than that of other single models. Compared with SVM, GBDT, XGBoost, LightGBM and CatBoost, the F1-score of MFDroid is increased by 4.9%, 7.4%, 4.6%, 3.6%, and 3.0%, and the accuracy value is increased by 5.1%, 7.9%, 4.8%, 3.8% and 3.1%, respectively. It

was proven that MFDroid has high accuracy and stability in the field of Android malware detection (see Figures 11 and 12).

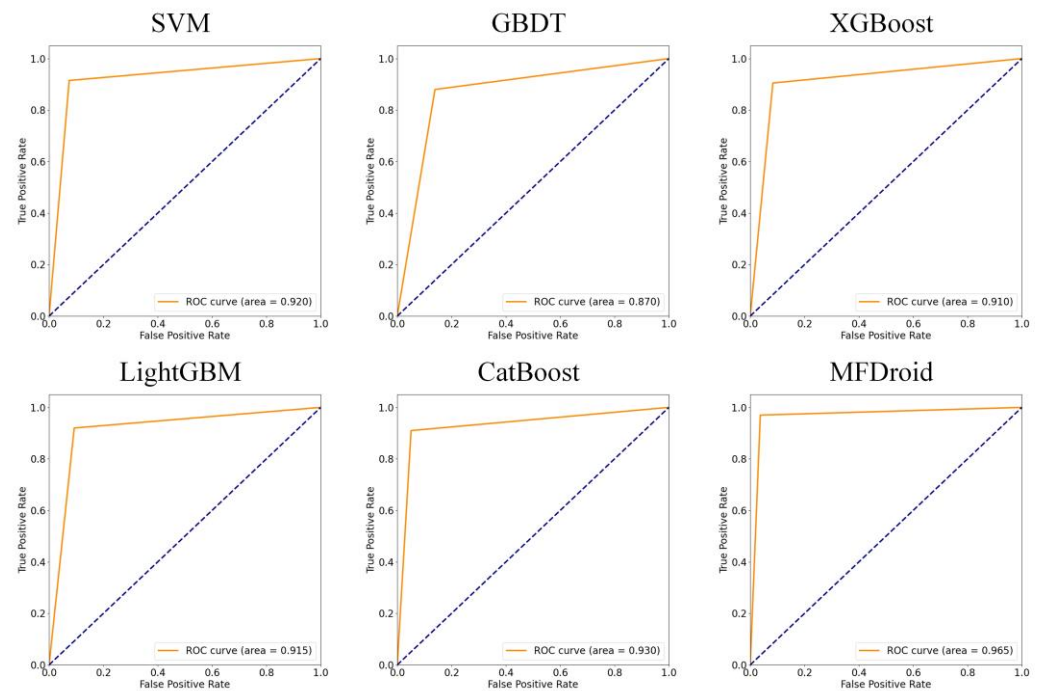


Figure 11. ROC curve of MFDroid and single classifier.

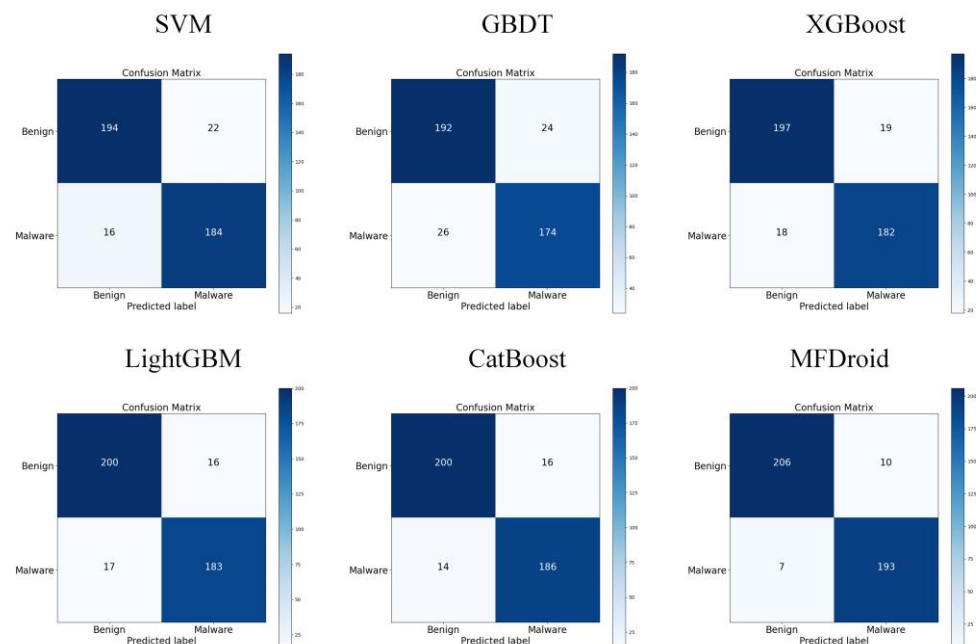


Figure 12. Confusion matrix of MFDroid and single classifier.

6. Conclusions and Future Work

We built a real-world Android malware dataset and proposed an Android malware detection framework based on stacking ensemble learning, called MFDroid, to accurately and effectively detect the application software on the Android operating system, and filter out the malware, to maintain the software quality of the Android application market and the information security of users. First, the framework combined the results of seven feature selection algorithms to obtain a new feature set. Then, combined with the advantages

and characteristics of various learners, we set five base learners in the first layer of the stacking ensemble learning framework, and the second layer used logistic regression as a meta-classifier. The results showed that MFDroid was an effective Android malware detection framework.

Regarding the limitations of a single feature selection algorithm, we conducted experiments on each feature selection algorithm, and the performance index of our method was higher than that of a single feature selection algorithm, which also indirectly proves that a single feature selection algorithm will miss some features. Regarding the limitations of a single machine learning algorithm—a single machine learning algorithm will be affected by data skew, and the computational overhead is relatively large. In addition, it may be affected by the initial setting, is sensitive to noise, and cannot handle high-dimensional features well. In the future, we will explore static and dynamic features that can describe application behavior more comprehensively, by using multiple types of features to improve detection accuracy.

Author Contributions: Conceptualization, L.Z., X.W.; methodology, X.W.; software, X.W.; validation, X.W., M.Y.; formal analysis, X.W., K.Z.; investigation, X.D.; resources, X.W., M.Y., X.D.; data curation, X.W.; writing—original draft preparation, X.W.; writing—review and editing, L.Z.; visualization, X.W., M.Y.; supervision, L.Z.; project administration, L.Z.; funding acquisition, L.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded in part by the Natural Science Foundation of Xinjiang Uygur Autonomous Region under grants 2019D01C062, 2019D01C041, 2019D01C205, and 2020D01C028; in part by the National Natural Science Foundation of China under grant 12061071; in part by the Higher Education of Xinjiang Uygur Autonomous Region under grant XJEDU2020Y003, and XJEDU2019Y006; in part by the Tianshan Innovation Team Plan Project of Xinjiang Uygur Autonomous Region under grant 202101642; in part by the National College Student Innovation Training Project under grant 202010755020; in part by the National College Student Innovation Training Project under grant 202010755021.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The author has no relevant interest to disclose.

References

1. Statcounter. Mobile Operating System Market Share Worldwide. Available online: <https://gs.statcounter.com/os-market-share/mobile/worldwide/> (accessed on 13 January 2022).
2. AV-Comparatives. Mobile Security Review 2021. Available online: <https://www.av-comparatives.org/tests/mobile-security-review-2021/#google-android> (accessed on 13 January 2022).
3. Yuan, W.; Jiang, Y.; Li, H.; Cai, M. A Lightweight On-Device Detection Method for Android Malware. *IEEE Trans. Syst. Man Cybern. Syst.* **2021**, *51*, 5600–5611. [CrossRef]
4. Vinod, P.; Zemmari, A.; Conti, M. A machine learning based approach to detect malicious android apps using discriminant system calls. *Future Gener. Comput. Syst.* **2019**, *94*, 333–350.
5. Wong, M.Y.; Lie, D. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 21–24 February 2016; pp. 21–24.
6. McDonald, J.; Herron, N.; Glisson, W.; Benton, R. Machine Learning-Based Android Malware Detection Using Manifest Permissions. In Proceedings of the 54th Hawaii International Conference on System Sciences, Hawaii, HI, USA, 4–8 January 2021; pp. 69–76.
7. 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 Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 23–26 February 2014; pp. 1–12.
8. Allix, K.; Bissyandé, T.F.; Klein, J.; Le Traon, Y. Androzoo: Collecting millions of android apps for the research community. In Proceedings of the 13th IEEE/ACM Working Conference on Mining Software Repositories (MSR), Austin, TX, USA, 14–15 May 2016; pp. 468–471.
9. Martín, A.; Lara-Cabrera, R.; Camacho, D. Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset. *Inf. Fusion* **2019**, *52*, 128–142. [CrossRef]

10. 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. [\[CrossRef\]](#)
11. Tao, G.; Zheng, Z.; Guo, Z.; Lyu, M.R. MalPat: Mining patterns of malicious and benign Android apps via permission-related APIs. *IEEE Trans. Reliab.* **2017**, *67*, 355–369. [\[CrossRef\]](#)
12. Alazab, M.; Alazab, M.; Shalaginov, A.; Mesleh, A.; Awajan, A. Intelligent mobile malware detection using permission requests and API calls. *Future Gener. Comput. Syst.* **2020**, *107*, 509–521. [\[CrossRef\]](#)
13. Gaviria de la Puerta, J.; Sanz, B. Using Dalvik opcodes for malware detection on android. *Logic. J. IGPL* **2017**, *25*, 938–948. [\[CrossRef\]](#)
14. Cai, L.; Li, Y.; Xiong, Z. JOWMDroid: Android malware detection based on feature weighting with joint optimization of weight-mapping and classifier parameters. *Comput. Secur.* **2021**, *100*, 102086. [\[CrossRef\]](#)
15. Mahindru, A.; Sangal, A.L. MLDroid—Framework for Android malware detection using machine learning techniques. *Neural Comput. Appl.* **2021**, *33*, 5183–5240. [\[CrossRef\]](#)
16. Jannat, U.S.; Hasnayeem, S.M.; Shuhan, M.K.B.; Ferdous, M.S. Analysis and Detection of Malware in Android Applications Using Machine Learning. In Proceedings of the 2019 International Conference on Electrical, Computer and Communication Engineering (ECCE), Cox's Bazar, Bangladesh, 7–9 February 2019; pp. 1–7.
17. Aboaoja, F.A.; Zainal, A.; Ghaleb, F.A.; Al-rimy, B.A.S. Toward an Ensemble Behavioral-based Early Evasive Malware Detection Framework. In Proceedings of the 2021 International Conference on Data Science and Its Applications (ICoDSA), Bandung, Indonesia, 6–7 October 2021; pp. 181–186.
18. Zhu, H.; Li, Y.; Li, R.; Li, J.; You, Z.; Song, H. SEDMDroid: An Enhanced Stacking Ensemble Framework for Android Malware Detection. *IEEE Trans. Netw. Sci. Eng.* **2021**, *8*, 984–994. [\[CrossRef\]](#)
19. 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. [\[CrossRef\]](#)
20. Rana, M.S.; Sung, A.H. Evaluation of Advanced Ensemble Learning Techniques for Android Malware Detection. *Vietnam. J. Comput. Sci.* **2020**, *7*, 145–159. [\[CrossRef\]](#)
21. VirusTotal. VirusTotal: Free Online Virus, Malware and URL Scanner. Available online: <https://www.virustotal.com/> (accessed on 13 January 2022).
22. Wang, X.; Wang, W.; He, Y.; Liu, J.; Han, Z.; Zhang, X. Characterizing Android apps' behavior for effective detection of malapps at large scale. *Future Gener. Comput. Syst.* **2017**, *75*, 30–45. [\[CrossRef\]](#)
23. Lee, J.; Jang, H.; Ha, S.; Yoon, Y. Android Malware Detection Using Machine Learning with Feature Selection Based on the Genetic Algorithm. *Mathematics* **2021**, *9*, 2813. [\[CrossRef\]](#)
24. Fatima, A.; Maurya, R.; Dutta, M.K.; Burget, R.; Masek, J. Android Malware Detection Using Genetic Algorithm based Optimized Feature Selection and Machine Learning. In Proceedings of the 42nd International Conference on Telecommunications and Signal Processing (TSP), Budapest, Hungary, 1–3 July 2019; pp. 220–223.
25. Takahashi, T.; Ban, T. Android Application Analysis Using Machine Learning Techniques. *AI Cybersecur.* **2019**, *151*, 181–205.
26. Macedo, F.; Oliveira, M.R.; Pacheco, A.; Valadas, R. Theoretical foundations of forward feature selection methods based on mutual information. *Neurocomputing* **2019**, *325*, 67–89. [\[CrossRef\]](#)
27. Han, H.; Lim, S.; Suh, K.; Park, S.; Cho, S.-J.; Park, M. Enhanced android malware detection: An svm-based machine learning approach. In Proceedings of the IEEE International Conference on Big Data and Smart Computing (BigComp), Busan, Korea, 19–22 February 2020; pp. 75–81.
28. 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.
29. 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.
30. Dorogush, A.V.; Ershov, V.; Gulin, A. CatBoost: Gradient boosting with categorical features support. *arXiv* **2018**, arXiv:1810.11363.
31. Rodrigo, C.; Pierre, S.; Beaubrun, R.; El Khoury, F. BrainShield: A Hybrid Machine Learning-Based Malware Detection Model for Android Devices. *Electronics* **2021**, *10*, 2948. [\[CrossRef\]](#)