MFF-AMD: Multivariate Feature Fusion for Android Malware Detection

Guangquan Xu[1,2], Meiqi Feng[1], Litao Jiao[2], Jian Liu[1(✉)], Hong-Ning Dai[3], Ding Wang[4], Emmanouil Panaousis[5], and Xi Zheng[6]

[1] College of Intelligence and Computing, Tianjin University, Tianjin, China
jianliu@tju.edu.cn
[2] Big Data School, Qingdao Huanghai University, Qingdao, China
[3] Faculty of Information Technology, Macau University of Science and Technology, Taipa, Macau SAR, China
[4] School of EECS, Peking University, Beijing, China
[5] University of Greenwich, London, UK
[6] Department of Computing, Macquarie University, Sydney, Australia

Abstract. Researchers have turned their focus on leveraging either dynamic or static features extracted from applications to train AI algorithms to identify malware precisely. However, the adversarial techniques have been continuously evolving and meanwhile, the code structure and application function have been designed in complex format. This makes Android malware detection more challenging than before. Most of the existing detection methods may not work well on recent malware samples. In this paper, we aim at enhancing the detection accuracy of Android malware through machine learning techniques via the design and development of our system called MFF-AMD. In our system, we first extract various features through static and dynamic analysis and obtain a multiscale comprehensive feature set. Then, to achieve high classification performance, we introduce the Relief algorithm to fuse the features, and design four weight distribution algorithms to fuse base classifiers. Finally, we set the threshold to guide MFF-AMD to perform static or hybrid analysis on the malware samples. Our experiments performed on more than 25,000 applications from the recent five-year dataset demonstrate that MFF-AMD can effectively detect malware with high accuracy.

Keywords: Malware detection · Hybrid analysis · Weight distribution · Multivariate feature fusion

1 Introduction

The proliferation of Android applications has been benefited from the rapid development of portable electronic devices, such as smartphones, tablets, and wearable smartwatches. As a byproduct, the number of malware has been constantly increasing over these years targeting at Android-based smartphones.

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2021
Published by Springer Nature Switzerland AG 2021. All Rights Reserved
H. Gao and X. Wang (Eds.): CollaborateCom 2021, LNICST 406, pp. 368–385, 2021.
https://doi.org/10.1007/978-3-030-92635-9_22

For Android users, there are several indications of this trend of ever-increasing threats of malware [1]. According to the Symantec research report [2], 23,795 Android malware on average were detected daily in 2017, an increment of 17.2% compared to that of 2016. While 360 Beaconlab [3] reported that approximately 12,000 malware samples per day on average were intercepted in 2018. Compared with that of 2017 or even three years ago, the malware detection rate has been reduced substantially. As AV-TEST [4] pointed out, this reduction is due to that malware is being designed and injected in a more complicated way, implying that attackers have focused on "better" quality of malware rather than the quantity. The corresponding malware detection methods should also be improved to adapt to the upgrade and development of malware.

In this work, we propose a model of MFF-AMD to address the above issues. Our model is built via a combination of dynamic and static techniques. Through the analysis of 25,000 samples obtained in the past five years, we extract features that can identify malicious applications from benign ones. More multiscale features can be used to describe an application better so as to detect malware more accurately. Meanwhile, to achieve a balance between efficiency and accuracy, our model automatically determines the analysis method. Specifically, we extract some basic static features including permissions, sensitive API calls, and other related features inferred from the basic features.

We also check the intent to see whether it is used to deliver sensitive messages. To extract more comprehensive dynamic features, we investigate the work related to detecting malware based on dynamic behaviors. We further implement a UI component testing scheme based on Android activity to trigger more malicious behaviors. In short, we extract the malware from more various dimensions, which can improve the malware detection consequently.

When training our model, we propose four weight distribution algorithms for base classifier fusion. We experimentally conclude that the overall performance of MFF-AMD is much better than those of a single-base classifier and other weight distribution algorithms. We also test the Android app samples from 2015 to 2019, to prove the robustness of our model. Experiments show that our model achieves good detection performance for Android malware samples from different years.

In summary, we highlight the major contributions of this paper as follows:

1. In order to solve the problem that existing methods cannot fully extract Android application features, we propose a model named MFF-AMD, which can achieve automated detection of Android malware, and our experimental results show that MFF-AMD can provide 96% accuracy on average.

2. MFF-AMD can automatically control the detection process during sample testing by training models that can adapt to static features or both static and dynamic features. Therefore, our model achieves higher accuracy with lower overhead.

3. In the process of training the model, in order to maximize the overall accuracy of MFF-AMD, we design a lightweight weight distribution algorithm to fuse the base classifiers. Experimental results show that our method can improve the overall accuracy by 3.53% on average.

The rest of the paper is organized as follows: In Sect. 2, we review the related work of Android malware detection based on machine learning. We introduce our feature extraction process in Sect. 3. We then describe our model implementation in Sect. 4. The experiment results and analysis are presented in Sect. 5. Finally, we conclude our paper and outlook future directions in Sect. 6.

2 Related Work

There have been many approaches to detect malware based on machine learning combined with Android static analysis. Wu et al. [5] used the Kmeans and K nearest neighbor (K-NN) algorithms, which combined with static features including permissions, intents, and API calls. Their main contribution is to provide a static analysis paradigm for detecting Android malware and to develop a system called DroidMat. Arp et al. [6] proposed a lightweight detection method based on Support Vector Machine (SVM) running on the mobile terminal.

Unlike static detection, some methods use dynamic features combined with machine learning, for instance, AntiMalDroid [7] is a framework of detecting malware based on the analysis of dynamic behavior via the SVM algorithm, in which the features are extracted from the log behavior sequence. Saracino et al. [8] proposed MADAM, which can combine several classes of features, from distinct Android levels, and applied both anomaly-based and signature-based mechanisms. Afonso et al. [9] proposed a system to dynamically identify whether an Android application is malicious or not, based on the features extracted from Android API calls and system call traces.

The hybrid analysis includes both dynamic analysis and static analysis. The AndroPytool framework [10] is a new hybrid analysis-based work. It proposes a malware detection method based on static and dynamic feature fusion through the combination of ensemble classifiers. They mainly develop their tools based on Flowdroid [11]. Their experiment result shows that AndroPytool can achieve up to 89.7% detection accuracy. TrustDroid [12] is a hybrid approach that can both operate on the phone and on a server. It takes the Android byte code and converts it into a textual description using Jasmin syntax. MARVIN [13] uses machine learning based on hybrid static and dynamic features (SVM and L2 regularized linear classifiers). MARVIN evaluates the risks associated with unknown Android applications in a malicious scoring form from 0 to 10.

Some other works [14–19] combined static analysis and dynamic analysis. We find that many studies calculate dynamic behavior based on logs and some system parameters, which will introduce computational overhead. Different from those works, we utilize a hook framework to directly monitor the triggered dynamic behavior, which has lower overhead. Meanwhile, we parse Android UI components to trigger more behaviors to improve detection as much as possible.

Moreover, we combine more comprehensive dynamic features with static ones used in malware detection to obtain higher accuracy.

3 Multivariate Feature Extraction

The main purpose of extracting features is to distinguish malware from benign functions. According to the related studies, the behavior of the Android application mainly relates to the static code and dynamic behavior; thus, we intend to extract contributive features between malware and benign applications based on these two aspects. In order to improve our model's robustness across diverse malicious samples, we use Android malicious families or Android applications across four years. At the same time, we consider the upgrade of the Android SDK version and the popularity of the 4G network communication with the Android application.

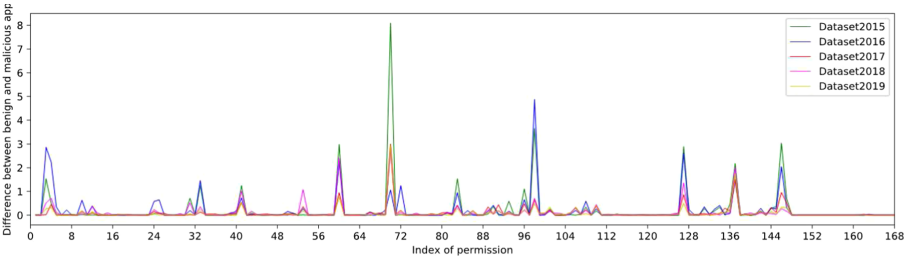

Fig. 1. The different distribution of the same permission on each data set.

3.1 Static Feature Extraction

Static features can be extracted without the need of running an application, which relies on concrete static analysis. We can perform automated extraction of some static features by using the python API provided by *Androguard* to analyze *apk* files. *Androguard* is an open source and supports extensions. It can implement automated reverse *apk* and easily extract static features of applications. We note that many related studies in the feature extraction part only give attention to whether exists a certain feature in the AndroidManifest.xml file or just describe the feature vector in binary code. The possible drawback is that the features of different codes of 1 are the same contribution for malware detection. Therefore, one improvement we made is to increase the frequency of utilization of features based on the original method. Features with large frequency may represent more important features. We also portray as many Android app details as possible.

– **Permissions.** The Android permission mechanism specifies operations that can be performed at different risk levels. With a more than 11,000 separate

sample set in the benchmark, our main idea is to extract the permission feature expressed in benchmarks and verify all the samples in our experimental dataset to see if the permission feature is effective in detecting malware. Figure 1 shows the different distribution of the samples we extracted from 2015 to 2019. From Fig. 1 we can see that almost 168 permission features we extracted are identical in distribution on each data set, though there are still some differences. Our assumption is that the greater the difference in permissions, the stronger the ability to distinguish malware.

- **Intents.** Intent can launch activities, create services, and communicate with other Android apps. Figure 2 shows the code that carries sensitive data through an intent object.
- **API calls.** Android officially marks the API's risk level, and those sensitive APIs are often leveraged as a powerful feature in Android malware detection.
- **Components.** We calculate the number of Android components as a continuous feature, including *activity, service, broadcast receiver*, etc.
- **Code features.** Enhancing the robustness of malware detection requires exploring techniques that can confuse source codes. We indicate if there is a confusion technique by whether the code calls the relevant package.
- **Certificate.** Certificate signing is an Android protection mechanism, which can prevent *apk* files from being tampered by malicious developers.
- **APK file structure.** We extract the entire *apk* file directory structure and analyze it.

```
button1.setOnclickListenner(new setOnclickListenner){
    @override
    public void onClick(view v){
        String data = 'sensitive message';
        Intent intent = new Intent(MainActivity.this,TargetActivity.class);
        intent.putExtra('extra_data',data);
        startActivity(intent);
    }
}
```

Fig. 2. Easy intent: sensitive data are transformed to TargetActivity by binding explicit intent object.

3.2 Dynamic Feature Extraction

The acquisition of dynamic features relies on the installation and running of the Android application to detect malicious behavior that cannot be detected by static analysis. Through the *Inspeckage* based on the *Xposed* framework, we hook the application to be tested and enable dynamic monitoring of the application's various behaviors. We analyze the log files generated during the dynamic analysis process to achieve batch extraction of dynamic features.

- **Networks**. We capture the network data and analyze the data packet to determine whether the HTTP request contains sensitive data related to the user.
- **File and SQLite operations.** We focus on the file or database operations of the application recorded in the log. Our main research objects are the path and data of the file, which may include sensitive information.
- **Command execution.** Malware can call system programs in sensitive directories to execute commands aiming to achieve camouflage and malicious operations. We count the number of executions of the system command, coded as a continuous feature.

3.3 Application Coverage

Android applications are based on event-driven, and the execution process can be simply summarized as driven by various input events, and the application completes a variety of different logical functions. The key to malware detection based on dynamic analysis is the application's execution coverage. Our model does not intent to insert probes into the application to calculate the code coverage during testing, because modifying code, repackaging, and other operations are not conducive to automated detection. So, in order to improve the effect of dynamic analysis, we design an event testing scheme based on the Android UI view. Our main idea is not to modify the *apk* but to input as many as UI events as possible to the activity component information of the Android application. In this way, potential malicious behavior can be triggered as much as possible. Figure 3 shows the dynamic detection scheme based on the UI view. Android will render the UI view and layout information currently on the screen. We construct the socket script through the activity information obtained by static analysis and try to communicate with the View Server to request the analysis result of the current screen information in real-time. Then, we parse the results on the client and obtain a simplified view tree that can uniquely locate all layout information through the parsing algorithm. Finally, through analyzing the components to formulate corresponding test events, we aim to trigger as many input events as possible. Therefore, we obtain corresponding dynamic features by monitoring the running behavior of the application.

3.4 Feature Selection

To reduce the overhead of training models and improve the detection accuracy, we first select and reduce the dimensionality of both static features and dynamic features. The feature selection is used to filter out the features that are more suitable for detecting malware, so as to have a positive effect on studying malicious samples. We use the Relief algorithm for feature selection, and select features with larger weights to participate in training. The advantage of choosing Relief is to separate the feature selection from the training process, and the algorithm performs well in terms of time complexity.

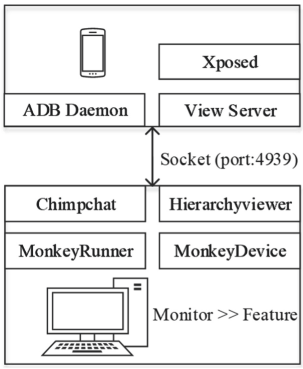

Fig. 3. Testing scheme based on UI view.

4 Implementation

4.1 Architecture

By analyzing the existing studies on Android malware detection, the method of using machine learning is generally divided into several stages: data filtering, feature extraction, model training, and model testing. We analyzed the steps of the malware detection process in detail and optimized it based on the traditional method. As shown in Fig. 4, during the training, our detection model extracts multi-dimensional static features and dynamic features from Android APK samples collected from different sources.

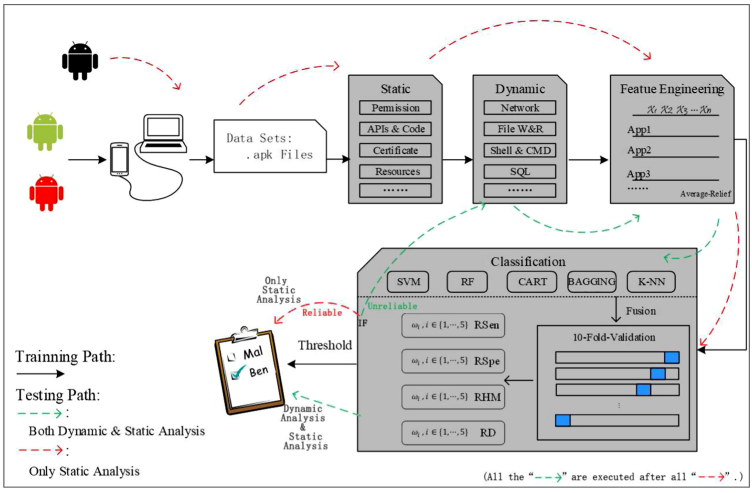

Fig. 4. MFF-AMD's architecture.

In the features pre-processing stage, we use the Relief algorithm to process the features, including the feature selection and feature dimensionality reduction. Based on the processed features, we use five base binary classification algorithms for supervised training, including SVM, RF, CART, BAGGING, and K-NN. Finally, we design four weight distribution algorithms including RSen, RSpe, RHM, and RD to allocate dual weights to the five base classifiers and obtain the combination of a pair of weights distribution strategies with the highest overall accuracy as the final algorithm to merge classifier.

For testing, given an arbitrary Android application, our model first performs static analysis on the application and then compares it with the threshold based on the score given by the classifier. If the score is higher than the threshold, the detection is ended, and the detection result is output. If the score is lower than the threshold, we need to perform hybrid analysis, and finally, compare the two classification results to determine whether the application is malicious. The advantage of this scheme is the ability to ensure accuracy and reduce the detection overhead.

4.2 Weight Distribution Algorithm

In order to improve the detection ability of our model, we adopt a strategy of emphasizing the base classifiers with weight pairs. We focus on how to assign a weight to different base classifiers. We know that different base classifiers have different classification performance. We denote True Positive Rate (TPR) as the detection accuracy of malicious samples by the fused classifier, which we named as sensitivity. We denote True Negative Rate (TNR) as the detection accuracy of benign samples by the fused classifier, which we named as specificity. Then, the overall accuracy of the weighted classifier can refer to the following equations, $P_{overall}$ can be computed as:

$$P_{overall} = \frac{N_M \cdot TPR + N_B \cdot TNR}{N_X} \tag{1}$$

where N_M and N_B represent the number of malicious and benign applications in the sample, respectively, and N_X represents the number of samples. In order to improve the overall generalization ability of the classifier to the sample, for a specific sample set, we can only improve TNR and TPR according to Eq. (1). By observing the performance of the classifier in a great number of malware detection processes, the overall classification accuracy of the classifier is directly proportional to its sensitivity and specificity, while both the sensitivity and the specificity of the most typical classifier are rarely equal in value. If one classifier has a higher specificity and sensitivity as well as a smaller difference between each other, it may have higher overall accuracy and robustness. Based on this, we propose four weight distribution algorithms, which can rank the classifier according to the specificity and sensitivity while assigning weights. These four algorithms are Ranking Algorithm Based on Sensitivity (RSen), Ranking Algorithm Based on Specificity (RSpe), Ranking Algorithm Based on Sensitivity and

Specificity Harmonic Mean (RHM), Ranking Algorithm Based on Sensitivity and Specificity difference (RD).

RSen Algorithm. In order to maximize the accuracy of the model's detection of malware, the simplest idea is to give the greatest weight to the classifier with the highest TPR. Thus, we design the first type of weight distribution algorithm. We use R_1 to represent RSen in this work.

Define e_k as the TPR of classifier k, $k \in \{1, \cdots, 5\}$. Make $E \leftarrow e_k$, we first rank each element of the set E in descending order and obtain $\bar{E}$. We assign weights of e_k according to the following equation:

$$\omega_i = 6 - i, i \in \{1, \cdots, 5\} \tag{2}$$

For instance, the weight of the classifier with the highest TPR is set to 5. The rules of subsequent algorithms are similar.

RSpe Algorithm. Similar to RSen algorithm, we set the maximum weight for the classifier with the highest sensitivity TNR, and the method is the same as the above, so we do not describe the detail. We use R_2 to represent RSpe in this work.

RHM Algorithm. We have known that the harmonic mean of two numbers tends to approach the one with a smaller value. Therefore, when the harmonic mean is large, and the two numbers are large. Classifier ranks are set to directly proportional to the sensitivity and specificity in this algorithm. We use R_3 to represent RHM in this work.

Define m_k as the harmonic mean of the sensitivity and specificity of classifier k, we have the following equation.

$$m_k = \frac{2 \cdot TPR_k \cdot TNR_k}{TPR_k + TNR_k}, \ k \in \{1, \cdots, 5\} \tag{3}$$

Make $M \leftarrow m_k$, and we let a set $M = \{m_k | k = \{1, \cdots, 5\}\}$ be the harmonic mean of classification accuracy of five classifiers. Descending the set M, and the symbol $\bar{M}$ represents the ranked set. We assign the weight for each classifier based on the element sequence in the set $\bar{M}$ according to Eq. (2) and we obtain the ω_i. Finally, the ω_i is going to be combined with classifier k and used to reclassify the category label of instance x.

RD Algorithm. In this algorithm, the weight of the classifier is designed to be inversely proportional to the absolute value of the difference between sensitivity and specificity. As this is a binary classification problem, the smaller the difference between the accuracy of the same classifier for different categories, the more stable the performance and the stronger the robustness. Such a classifier can perform better even on a dataset where the two samples are not balanced. We probably assign the weight to the classifier with stronger robustness by using this algorithm. This time, we use R_4 to represent this algorithm in this work.

Define d_k as the absolute value of the sensitivity and specificity of classifier k, we have:

$$d_k = |TPRe_k - TNR_k|, k \in \{1, \cdots, 5\} \tag{4}$$

Make $D \leftarrow d_k$, and we denote a set $D = \{d_k | k = \{1, \cdots, 5\}\}$ as the difference of classification accuracy of five classifiers. Unlike before, we ascend the set D and the symbol $\bar{D}$ represents the ranked set. Finally, we assign weights for each classifier based on the previous strategy and reclassify instances.

After we assign weights to each classifier using these four weight distribution algorithms, we use the combination of any two algorithms to weight the classifier again to get the optimal weight combination scheme. For example, if R_1 assigns each classifier a weight set as $\{1, 2, 4, 5, 3\}$ and R_2 assigns each classifier a weight set as $\{2, 3, 4, 1, 5\}$. Then the combined scheme R_1R_2 assigns each classifier a final weight set as $\{3, 5, 8, 6, 8\}$. This weight set is used to reclassify the sample. We will evaluate each scheme in the next section to find the optimal solution.

5 Evaluation

The purpose of our experiments is multifaceted. In this section, we analyze the selection of parameters and evaluate the performance and overhead of our model.

5.1 Dataset and Setup

Environment. Our experiments are all done under Windows 10 Enterprise Edition, and the PC is equipped with Intel (R) Core I5-4460 CPU@ 3.20 GHz. We leverage *androguard* (v3.4.0) and python 3.6 to extract static features. For dynamic feature extraction, we used *Google Nexus 5* (Android 5.0, SDK 21) based on the *Xposed* framework using *Genymotion* (v3.0.2). The *MonkeyRunner* and the *HierarchyViewer* that come with Android are used for dynamic event testing. In the feature engineering stage, we use python3 to implement the Relief algorithm and use it for feature extraction and fusion.

Dataset. To perform an effective analysis, we select more than 25,000 samples from multiple sources. It includes benign and malicious Android apps from 2015 to 2019. Table 1 lists all datasets used in our experiments. We describe data set in alphanumeric format for simplicity. For example, samples from 2017 are represented to Dataset-17 respectively. All benign applications were downloaded from *Google Play* (GP) and confirmed by the *Virus Total* platform. We collected applications in Google play using the third-party website *apkCombo*. We did not pay attention to the category of the applications when collecting benign samples, because we aim for a generic solution. Our malicious applications were collected from *VirusShare*. After downloading malicious sample files from the platform, we filter out those non-Android related malicious samples. In order to avoid the biases of the unbalance of samples, the number of benign samples and malicious samples in each dataset is roughly equal.

Metrics. We use common metrics in machine learning to evaluate the classification performance. In our research, precision is expressed as the correct rate of malware detection. The recall rate (TPR) reflects the sensitivity of our model to malware. TNR is also this case for benign samples. F1-score (F1) represents

Table 1. Datasets.

DATASET	Benign apps		Malicious apps		Total
	Source	#App	Source	#App	
Dataset-15	GP	2968	VS	2793	5761
Dataset-16	GP	2992	VS	2837	5829
Dataset-17	GP	2991	VS	2780	5771
Dataset-18	GP	2974	VS	2861	5835
Dataset-19	GP	2784	VS	2963	5747

a comprehensive evaluation indicator of the classifier performance on malware detection. In addition, accuracy (ACC) is defined as the correct classification rate of our model for both benign and malicious samples.

5.2 Results and Analysis

Optimal Combination of Weights Distribution Algorithm. We select five base classifiers in this work, including SVM, Random Forest (RF), Classification and Regression Tree (CART), BAGGING, and K-Nearest Neighbor (K-NN). These classifiers have shown good performance in malware detection, so to demonstrate the effectiveness of our weight distribution algorithms, we still use these base classifiers. We choose to use Dataset-17 to evaluate the five base classifiers. For Dataset-17, we first test the performance of each base classifier in static analysis and hybrid analysis. The results are shown in Table 2. Then, we use four ranking algorithms to assign weights to the five base classifiers. The results are shown in Table 3. Each column represents the weights assigned to the classifier by each algorithm.

Table 2. Performance of base classifier on dataset-17 (%).

Classifier	Static analysis			Hybrid analysis		
	TPR	TNR	ACC	TPR	TNR	ACC
RF	92.81	93.75	93.28	95.35	98.14	96.77
CART	89.87	95.39	92.62	91.47	97.03	94.31
BAG	91.50	92.43	91.97	93.41	96.65	95.07
K-NN	81.70	92.11	86.89	91.86	92.94	92.41
SVM	85.29	92.76	89.02	84.88	94.05	89.56

From Table 2 and Table 3, we can see that in the static analysis, RF is more sensitive to malware, while CART has better performance in identifying benign applications. In the hybrid analysis, the overall performance of RF is better, with

Table 3. Each algorithm combination's performance on dataset-17 (%).

Classifier	Static analysis				Hybrid analysis			
	RSen	RSpe	RHM	RD	Rsen	RSpe	RHM	RD
RF	5	4	5	4	5	5	5	3
CART	3	5	4	3	2	4	3	2
BAG	4	2	3	5	4	3	4	4
K-NN	1	1	1	1	3	1	2	5
SVM	2	3	2	2	1	2	1	1

an accuracy rate of 96.77%. The difference between KNN's TRP and TNR is the smallest, only 1.08%, indicating that the classifier is relatively stable. Overall, the classification performance of SVM and KNN is not good, hence a low weight is assigned when the classifiers are fused. There are performance differences between each classifier, but at the same time, each has its own advantages. Finally, we use a dual weight distribution strategy to combine different algorithms and fuse the classifiers. The results are shown in Table 4 and Table 5, respectively.

Table 4. Each algorithm combination's performance on dataset-17 in the static analysis (%).

Combination	TPR	TNR	F1-score	ACC
R_1R_2	90.85	95.39	95.17	93.11
R_1R_3	92.48	94.74	94.62	93.61
R_1R_4	91.83	95.07	94.90	93.44
$\boldsymbol{R_2R_3}$	**93.02**	**95.73**	**95.61**	**94.35**
R_2R_4	90.85	95.39	95.17	93.11
R_3R_4	93.27	94.97	94.88	94.10

From the data in Table 4 and Table 5, specifically, in the static analysis, the ACC of the optimal algorithm combination R_2R_3 reaches 94.35%, which is 1.07% higher than the best base classifier RF. After the detection of classifier fusion and hybrid analysis, it can achieve 97.15% accuracy and 98.83% F1-score. Overall, no matter it is static analysis or hybrid analysis, most of the dual weight distribution combinations have played a positive role, indicating the effectiveness of our weight distribution algorithm. For static analysis and hybrid analysis, we selected the optimal algorithm combination that is suitable for each detection method and displayed it in bold in the table.

The Threshold for Static Analysis. In order to achieve a better detection effect with less time overhead, we set a threshold for static analysis. Firstly,

Table 5. Each algorithm combination's performance on dataset-17 in the hybrid analysis (%).

Combination	TPR	TNR	F1-score	ACC
$\boldsymbol{R_1R_2}$	**95.34**	**98.88**	**98.83**	**97.15**
R_1R_3	93.80	98.88	98.82	96.39
R_1R_4	93.80	98.88	98.82	96.39
R_2R_3	94.95	96.29	96.24	95.63
R_2R_4	93.41	98.88	98.82	96.20
R_3R_4	94.17	96.29	96.21	95.25

the static analysis of the application is performed. If the probability that the model detects the sample as malicious is less than the preset threshold, then the application will be subjected to mixed analysis; otherwise, the detection will be ended. Such a detection method can reduce the overall internal overhead of the model, and at the same time can obtain a higher detection accuracy.

The key issue is how to determine this threshold according to our needs. We noticed using the ROC curve to help us solve the problem. In machine learning, the AUC calculated by the ROC curve is an indicator for evaluating different classifiers. The ROC curve of each base classifier and the fused model during static analysis is shown in Fig. 5. The label in the figure represents the AUC value corresponding to each classifier. It can be seen from the figure that the fused classifier performs better than the base classifier. At the same time, we can calculate the point closest to (0, 1) in the coordinate system, which has a value of 94.83%, and we mark it with an arrow. This point means that when this probability threshold is used to judge as malware, it can make a lower FPR and a higher TPR. At this time, it has the best performance for the entire model, which can reduce the model overhead.

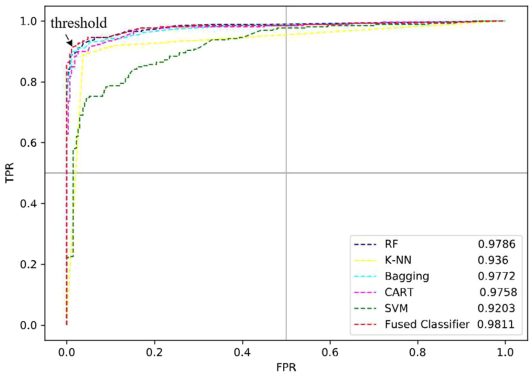

Fig. 5. ROC curve of each classifier.

Performance Evaluation. We evaluate the model performance in terms of execution time. We set up two sets of experiments to count the average detection time of samples that only require static analysis and the average detection time of samples that require static and dynamic analysis. The results are shown in Table 6. We select *apk* files with a size of 5 to 25 MB from the original sample set for evaluation. Due to the large differences between each *apk*, we count the average detection time. As can be seen from the table, for a small sample size (5 to 10 MB), our model only needs to perform static analysis to detect malware, the average time cost is 103.08 s. For small samples that require further dynamic analysis, the average detection increases to 223.33 s. The main time consumed in the process of dynamic analysis consists of two parts, including dynamic test time and feature extraction time. No matter for large samples or small samples, compared to dynamic analysis, the static analysis only needs to perform feature extraction and sample prediction, which can save the detection time.

Table 6. Application's average detection time.

Method	Sample size	App size (MB)	Average detection time (s)			Total (s)
			Testing	Extraction	Prediction	
Static	200	5–10	–	103.08	0.9×10^{-3}	103.08
Static	200	10–25	–	199.86	0.37×10^{-3}	199.86
Hybrid	200	5–10	116	107.33	0.44×10^{-3}	223.33
Hybrid	200	10–25	252	202.27	0.35×10^{-3}	454.27

Model Robustness. In order to verify the robustness of MFF-AMD, we tested different datasets and the results are shown in Table 7. The experimental results show that our model performs well for other datasets. This means that our model can handle almost all Android applications without considering whether the application is up to date. It is worth noting that the accuracy of our model for dataset-15 and dataset-19 is higher than 96%, indicating that our features have a better prevalence. Meanwhile, our average precision and FRP are 97.12%

Table 7. Performance in each dataset (%).

Dataset	TPR	TNR	Precision	ACC	F1
Dataset-15	95.44	97.22	97.16	96.33	96.30
Dataset-16	94.17	96.47	96.23	95.35	95.19
Dataset-17	94.56	96.47	96.24	95.54	95.37
Dataset-18	94.96	98.14	98.00	96.58	96.46
Dataset-19	94.18	98.14	97.98	96.20	96.04

and 2.72%, respectively, indicating that the detection results of our model have a high degree of confidence.

Comparison with Related Work. We selected 5 related studies that have similar features or similar analytical methods. The results are shown in Table 8. The work in the table may use a static method [5], the dynamic method [9,20], or hybrid method [21,22]. We found that static and hybrid methods select features related to permissions and sensitive APIs as their static features, which indicates the prevalence of such features for static Android malware detection. While the difference is that we have added some features such as component features, certificates, etc. At the same time, for some samples, our method can perform dynamic analysis based on static analysis. Most of the dynamic methods are based on dynamic behavior monitoring, by building feature sets from different aspects. Some of these scores in the table look slightly better than us. That is because they were tested in different datasets (our samples are more abundant), which is explained in the following to prove the superiority of our method.

Table 8. Comparison with other approaches (%).

Method	Type		Feature							ACC/F1
	St	Dy	P	A	M	C	N	O	E	
Droidmat	✓	–	✓	✓	✓	–	–	–	–	97.87 91.83
Andromaly	–	✓	–	–	–	–	✓	✓	✓	91.13 –
Afonso	–	✓	–	✓	–	–	✓	✓	–	96.82 96.79
M. Su	✓	✓	✓	✓	–	✓	✓	✓	–	97.4 –
stormDroid	✓	✓	✓	✓	–	–	✓	✓	–	93.80 93.80
MFF-AMD	✓	✓	✓	✓	✓	✓	✓	✓	✓	96.00 95.87

St: static method, Dy: dynamic method, P: permission-based feature, A: API-based feature, M: meta-information-based feature, including component information, intent, package information, file md5, file size, certificate, etc., C: code-feature-based feature, including java reflection, dynamic loading, etc., N: network- information-based feature, O: sensitive-file-based or database-operation-based feature, E: shell-based or command-based feature.

Each accuracy in Table 8 is obtained from its own dataset, because we didn't get all their source code, and some of their datasets are not such comprehensive. Compared with the dataset size of the three works of Andromaly (10800), Afonso (7520), and StormDroid (7970), our dataset samples are more abundant.

Therefore, experiment results show that the advantages exist in comparison with other schemes. The Droidmat's datasets (1738) and M. Su's datasets (1200) have a smaller size, and the obtained accuracy rates are 97.87% and 97.4%, respectively. We also test our scheme on their dataset (Contagio Mobile dataset) separately. In the case of the same sample, our method has an advantage in accuracy, and the detection accuracy reaches 98.37% and 98.34%, which is superior to these two schemes.

6 Conclusion and Future Work

We proposed a high accuracy-oriented detection model of Android malware - MFF-AMD, based on multiscale feature extraction and classifier fusion. It extracts dynamic and static features and is proved to be effective in distinguishing between benign applications and malicious applications. We balanced the contradiction between overhead and accuracy in hybrid analysis via selective dynamic analysis. Our research shows that by using our designed weight distribution algorithm to fuse base classifiers, we can make up the unreliable performance of base classifiers, and effectively improve the overall accuracy of the model. Finally, MFF-AMD performs a better detection rate and robustness based on our data from the past five years. Our work can provide a solid solution to complement current malware detection.

There are some interesting works following this line of research. The process of extracting dynamic features is still costly. Despite we use UI views for dynamic analysis, it may be not able to perform code-level analysis, which may result in the failure to trigger a logical relationship and deeper malicious behavior. One possible solution is to use semantic analysis to enhance our model, which we will study in future work.

Acknowledgment. This work is partially sponsored by National Key R&D Program of China (No. 2019YFB2101700), National Science Foundation of China (62172297, 61902276), the Key Research and Development Project of Sichuan Province (No. 21SYSX0082), Tianjin Intelligent Manufacturing Special Fund Project (20201159).

References

1. SophosLabs 2018 Malware Forecast. https://www.sophos.com/de-de/medialibrary/PDFs/technical-papers/malware-forecast-2018.ashx. Accessed 20 Mar 2019
2. Symantec 2018 Internet Security Threat Report. https://symantec.com/security-center/threat-report. Accessed 2 Mar 2019
3. Core Security. http://blogs.360.cn/post/review_android_malware_of_2018.html. Accessed 20 Mar 2019
4. AV-TEST Antivirus for Android. https://www.av-test.org/en/news/test-20-protection-apps-for-android/. Accessed 26 Sept 2019

5. Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., Wu, K.-P.: DroidMat: Android malware detection through manifest and API calls tracing. In: 2012 Seventh Asia Joint Conference on Information Security, pp 62–69, Tokyo. https://doi.org/10.1109/asiajcis.2012.18
6. Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K.: DREBIN: effective and explainable detection of Android malware in your pocket. In: Proceedings 2014 Network and Distributed System Security Symposium, vol. 14, pp. 23–26 (2014). https://doi.org/10.14722/ndss.2014.23247
7. Zhao, M., Ge, F., Zhang, T., Yuan, Z.: AntiMalDroid: an efficient SVM-based malware detection framework for Android. In: Liu, C., Chang, J., Yang, A. (eds.) ICICA 2011, Part I. CCIS, vol. 243, pp. 158–166. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-27503-6_22
8. Saracino, A., Sgandurra, D., Dini, G., Martinelli, F.: MADAM: effective and efficient behavior-based Android malware detection and prevention. IEEE Trans. Dependable Secure Comput. **15**(1), 83–97 (2018). https://doi.org/10.1109/tdsc.2016.2536605
9. Afonso, V.M., de Amorim, M.F., Grégio, A.R.A., Junquera, G.B., de Geus, P.L.: Identifying Android malware using dynamically obtained features. J. Comput. Virol. Hacking Tech. **11**(1), 9–17 (2014). https://doi.org/10.1007/s11416-014-0226-7
10. 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 **52**, 128–142 (2019). https://doi.org/10.1016/j.inffus.2018.12.006
11. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., et al.: FlowDroid. ACM SIGPLAN Not. **49**(6), 259–269 (2014). https://doi.org/10.1145/2666356.2594299
12. Zhao, Z., Colon Osono, F.C.: TrustDroid: preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking. In: 2012 7th International Conference on Malicious and Unwanted Software. https://doi.org/10.1109/malware.2012.6461017
13. Lindorfer, M., Neugschwandtner, M., Platzer, C.: MARVIN: efficient and comprehensive mobile app classification through static and dynamic analysis. In: 2015 IEEE 39th Annual Computer Software and Applications Conference. https://doi.org/10.1109/compsac.2015.103
14. Bichsel, B., Raychev, V., Tsankov, P., Vechev, M.: Statistical deobfuscation of Android applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (2016). https://doi.org/10.1145/2976749.2978422
15. Wang, W., Li, Y., Wang, X., Liu, J., Zhang, X.: Detecting Android malicious apps and categorizing benign apps with ensemble of classifiers. Future Gener. Comput. Syst. **78**, 987–994 (2018). https://doi.org/10.1016/j.future.2017.01.019
16. Wang, W., et al.: Constructing features for detecting Android malicious applications: issues, taxonomy and directions. IEEE Access **7**, 67602–67631 (2019). https://doi.org/10.1109/access.2019.2918139
17. Chen, K., Wang, P., Lee, Y., Wang, X., Zhang, N., Huang, H., et al.: Finding unknown Malice in 10 seconds: mass vetting for new threats at the Google-play scale. In: USENIX Security, vol. 15 (2015). https://doi.org/10.5555/2831143.2831185

18. Gascon, H., Yamaguchi, F., Arp, D., Rieck, K.: Structural detection of Android malware using embedded call graphs. In: Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security. https://doi.org/10.1145/2517312.2517315
19. Perdisci, R., Dagon, D., Wenke Lee, Fogla, P., Sharif, M.: Misleading worm signature generators using deliberate noise injection. In: 2006 IEEE Symposium on Security and Privacy (S&P). https://doi.org/10.1109/sp.2006.26
20. Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y.: "Andromaly": a behavioral malware detection framework for Android devices. J. Intell. Inf. Syst. **38**(1), 161–190 (2011). https://doi.org/10.1007/s10844-010-0148-x
21. 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, pp. 377–388, Xi'an (2016). https://doi.org/10.1145/2897845.2897860
22. Su, M.-Y., Chang, J.-Y., Fung, K.-T.: Machine learning on merging static and dynamic features to identify malicious mobile apps. In: 2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN). https://doi.org/10.1109/icufn.2017.7993923