Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.xxxx.DOI

DroidEnsemble: Detecting Android Malicious Applications with Ensemble of String and Structural Static Features

WEI WANG[a,b], ZHENZHEN GAO[a], MEICHEN ZHAO[a], YIDONG LI[a], JIQIANG LIU[a], XIANGLIANG ZHANG[c]

[a]Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, 3 Shangyuancun, Beijing 100044, China
[b]Science and Technology on Electronic Information Control Laboratory, Chengdu 610036, China
[c]Division of Computer, Electrical and Mathematical Sciences & Engineering, King Abdullah University of Science and Technology (KAUST), Saudi Arabia

Corresponding author: Yidong Li (e-mail: ydli@bjtu.edu.cn).

ABSTRACT Android platform has dominated the Operating System of mobile devices. However, the dramatic increase of Android malicious applications (malapps) has caused serious software failures to Android system and posed a great threat to users. The effective detection of Android malapps has thus become an emerging yet crucial issue. Characterizing the behaviors of Android applications (apps) is essential to detecting malapps. Most existing work on detecting Android malapps was mainly based on string static features such as permissions and API usage extracted from apps. There also exists work on the detection of Android malapps with structural features, such as Control Flow Graph (CFG) and Data Flow Graph (DFG). As Android malapps have become increasingly polymorphic and sophisticated, using only one type of static features may result in false negatives. In this work, we propose DroidEnsemble that takes advantages of both string features and structural features to systematically and comprehensively characterize the static behaviors of Android apps and thus build a more accurate detection model for the detection of Android malapps. We extract each app's string features, including permissions, hardware features, filter intents, restricted API calls, used permissions, code patterns, as well as structural features like function call graph. We then use three machine learning algorithms, namely, Support Vector Machine (SVM), k-Nearest Neighbor (kNN) and Random Forest (RF), to evaluate the performance of these two types of features and of their ensemble. In the experiments, We evaluate our methods and models with 1386 benign apps and 1296 malapps. Extensive experimental results demonstrate the effectiveness of DroidEnsemble. It achieves the detection accuracy as 95.8% with only string features and as 90.68% with only structural features. DroidEnsemble reaches the detection accuracy as 98.4% with the ensemble of both types of features, reducing 9 false positives and 12 false negatives compared to the results with only string features.

INDEX TERMS Android malicious application analysis, malware analysis, software failure reduction, static analysis

I. INTRODUCTION

ANDROID continues to dominate the market of mobile devices. Gartner [1] indicates that Android market shares up to 86.1% in 2017. The total number of Android applications (apps) is rapidly increasing. Meanwhile, the proportion of malicious applications (malapps) is on the rise. From January to July in 2017, 360 Internet Security Center [2] cumulatively monitored 4.839 million new mobile malicious program samples added to end users. The malicious behaviors of these programs mainly include traffic consumption, stealing private information, malicious deductions, etc. Obviously, all the malicious behaviors have posed a great threat to users on both mentality and property. In addition, malapps often do not comply with the user's expectations and thus cause many serious software failures. For instance, a set of malapps consume lots of running memory, possibly leading to the failure of running benign apps. Consequently, developing effective approaches to vetting and detecting Android malapps is crucial to secure the Android markets and to reduce the failures of Android apps.

Most existing static analysis methods for Android malapp detection are based on string features [3]–[8] or structural features [9], [10]. String features, or the meta-data information, are descriptive information regarding the softwares/applications and their source code. They usually refer to features such as permissions, intents, API calls, etc. Structural features mainly refer to the structural relationships within the app, such as Control Flow Graph (CFG) or inter-component data flow graph as well as inter-procedural CFG. Although both of these two types of features can be used for detecting malapps, using only one of them may lead to false negatives or false positives. For example, string features may fail to detect most circumvention attacks or collusion attacks, while structural features are not effective on the detection of sophisticated malapps.

In order to better characterize the Android apps' behaviors and thus more accurately detect malapps, in this work, we propose a detection model called DroidEnsemble by taking advantages of both string features and structural features. As for string features, we extract permissions, hardware features, filtered intents, restricted API calls, used permissions and code patterns from each Android app. We further extract the structural features like function call graph to collaborate with string features for malapp detection. We then employ three methods, namely, Support Vector Machine (SVM), k-Nearest Neighbor (kNN) and Random Forest (RF), to evaluate the effectiveness of both types of features and of their ensemble of features. The extensive experimental results demonstrate the effectiveness of DroidEnsemble. It achieves the detection accuracy as 95.8% with only string features, and 90.68% with only structural features. The ensemble features acquire the highest accuracy of 98.4% and F-score of over 0.98. The experimental results also show that string features are more effective and efficient for malapp detection than structural features. However, the structural features are able to detect most samples that cannot be detected by string features, and they well make up for the deficiency of string features. As a consequence, the ensemble of both features outperforms any individual feature set within DroidEnsemble.

We make the following three contributions.

1) We propose DroidEnsemble that effectively detects Android malapps with ensemble of both string and structural static features. The ensemble of both features can characterize the static behaviors of apps more systematically and comprehensively than any individual feature. Moreover, DroidEnsemble is more helpful to reduce the Android software failures. In details, string features suit for vetting malapps in general while structural features suit for inspecting instruction-level obfuscation in apps in particular.
2) We extract 6 types of string features and structural features like function call graph (FCG) to characterize the behaviors of Android apps. The number of string features is as large as 34552. The FCG features of each app contain several 15-bit function node encodings. We employ three supervised classifiers, namely, Support Vector Machine (SVM), k-Nearest Neighbor (kNN) and Random Forest (RF), to vet malapps. We compare the detection performance with different types of features and with different classifiers.
3) We conduct experiments on a data set containing 1386 benign apps collected from four app markets and 1296 malapps collected in the wild. Extensive experimental results demonstrate the effectiveness of DroidEnsemble. It achieves the detection accuracy as 95.8% with string features and 90.68% with structural features. DroidEnsemble reaches the detection accuracy of 98.4% with ensemble of both features, outperforming any individual feature.

The rest of this paper is organized as follows. We review related work in section II. Section III introduces DroidEnsemble, and explains in details both types of features. Section IV describes the data sets and experiments. We illustrate the limitations of DroidEnsemble in Section V. Section VI concludes this paper.

II. RELATED WORK

The issue of information security has been receiving widespread attention. There exists work on authentication security [11]–[13] or data security [14]–[17]. Due to the popularity of mobile devices, they have become major targets of attacks with malapps. The detection of malapps is thus essential to securing Android app markets and to reducing apps' failures.

Static analysis is often used for vetting and detecting malapps. String features and structural features are two typical features in static analysis of apps. String features are straightforward and easy to extract from most apps. If an app wants to execute some specified operations or applies for some resources, it must declare corresponding string information in the manifest file, such as permissions, hardwares, etc. Much work extracts this kind of information to discriminate malicious apps from benign ones. Wu et al. [18] extracted permissions as well as other features as features and utilized machine learning methods to detect malapps. Feizollah et al. [19] evaluated the effectiveness of Android Intents (explicit and implicit) for identifying malicious apps. They also conducted experiments with Android Intent in conjunction with permissions. Idrees et al. [20] used a combination of permissions and intents for identifying Android malicious apps, and optimized the results with ensemble methods. Previous work [21]–[26] also employed API calls as features in malapp detection. Hou et al. [25] further categorized the API calls that belong to the same method in the smali code into a block, which is the so-called API call block. In our previous work [4], [8], [27], we also used permissions and other string features to detect malapps.

Although string features are generally effective for detecting malapps, they are easily circumvented or exploited by sophisticated attacks in many cases. For example, Salehi et al. [28] built a kernel-level attack model that did not need to apply for relevant permissions. This attack model can obtain all the sensitive information it wants. Fortunately, the majority of new malapps are variants of existing malapps. Structural features are more suitable to detect these types of malapps. Crussell et al. [29] proposed a tool using a technique based on program dependency graphs (PDGs) to acquire the similarity between the malicious and legitimated apps. Gascon et al. [30] adopted a method for malapp detection based on efficient embedding of function call graphs with an explicit feature map. This method strove to identified subgraphs of the function call graph representing known malicious code. Alam et al. [31] used control flow with patterns, and implemented and adapted two techniques including Annotated Control Flow Graph (ACFG) to reduce the effect of obfuscations. This method was conducive to detecting the variants of malapps. By extracting the CFG, Kim et al. [32] formed structural information of methods in an Android app called '4-tuple' and clustered the apps with the computed similarity of apps. Dam and Touili [33] constructed API call graphs by applying a kind of control point reachability analysis on the CFG, so as to carry out further experiments.

In summary, previous work used either string or structural features to characterize the Android apps' behaviors and accordingly detect malapps. However, many malapps evade the detection based on only string features through technologies like code obfuscation or encryption. Whilst structural features may perform poorly in detecting malapps from new malicious families. Furthermore, as malapps have become increasingly polyphomic and sophisticated, the detection merely based on one type of features cannot meet the needs. In this work, we propose DroidEnsemble that considers 6 string features and structural features to improve the detection performance. In DroidEnsemble, the detection with string features runs fast and accurately, and the detection with structural features identifies more specific anomalies like instruction-level obfuscation. It thus improves the detection performance with the ensemble of string and structural features.

III. METHOD

DroidEnsebmle works with four steps, as shown in Fig. 1. First, we collect a number of apps from four app markets and malapps in the wild. Second, we extract 6 types of string features and function call graph as structural features from each apk. Third, we construct three supervised learning models to evaluate the performance of our methods with both types of feature sets. Fourth, we validate the effectiveness of DroidEnsemble with ensemble of string and structural features. Finally, we conduct extensive experiments including: (1) comparing the performance of our methods with each type of features; (2) classifier comparison and (3) optimizing the detection results with ensemble of both types of features.

In this section, we firstly explain in details the string features and structural features used, and then describe the machine learning models we employ in this work. We then describe the methods with the ensemble of both types of features.

A. FEATURE SETS

1) String Features

We extract 6 types of string features from each app. All the features are described as follows.

FS1. *Requested Permissions*: If an app needs to execute some specified operations, it must request corresponding permissions in the manifest file. Each app contains a manifest file providing meta-information supporting the installation and later execution of the app. Valuable information can be extracted from this file. However, some apps request permissions that are unnecessarily needed in their functions, which may indicate malicious intents. In another case, the combination of multiple permissions may reflect some harmful behaviors. For example, if an app applies for network connecting permission as well as SMS accessing permission, the app may acquire users' SMS information and then spread it out through the Internet. In this work, we use all the permissions declared in the manifest file, with <uses-permission> elements, as a feature set. Previous related work [4], [8], [19], [27], [34]–[36] demonstrated the effectiveness of this feature.

FS2. *Hardware Features*: In Android system, hardware and software requirements indicate the demands of the apps for system resources. For instance, if an app accesses 4G and GPS, it may imply that it reports the location of the user to the attacker, which reveals the malicious behavior of an app. Hence, we extract the hardware and software information defined in the manifest file, with <uses-feature> elements, as the second feature set. There exists related work [27], [37] that used this type of features.

FS3. *Filtered Intents*: Intents handle the communication between components by sending intent objects on Android. Intent filters help app components reject the unwanted intents as well as leave the desired intents. We extract the filtered intents as another feature set for malapp detection and they are signal with <intent-filter> elements. Feizollah et al. [19] evaluated the effectiveness of Android Intents (explicit and implicit) as the feature for identifying malicious apps and the detection rate reached 91%.

We extract above three feature sets from the manifest file with androguard tools and Android Asset Packaging Tool (aapt). Furthermore, we extract another three static features from disassemble code (FS4-FS6).

FS4. *Restricted API Calls*: The Android permission system restricts access to a series of critical API calls, presenting how an app interacts with Android framework. Restricted API Calls are protected by permissions. According to the API-permission mapping provided by PScout, it is easy to identify which APIs are protected by permissions. We then define a dictionary that maps the relationship between permissions and corresponding restricted API Calls. We scan the disassembled code of the app samples and record whether they invoke API calls protected by some permissions to acquire this feature set.

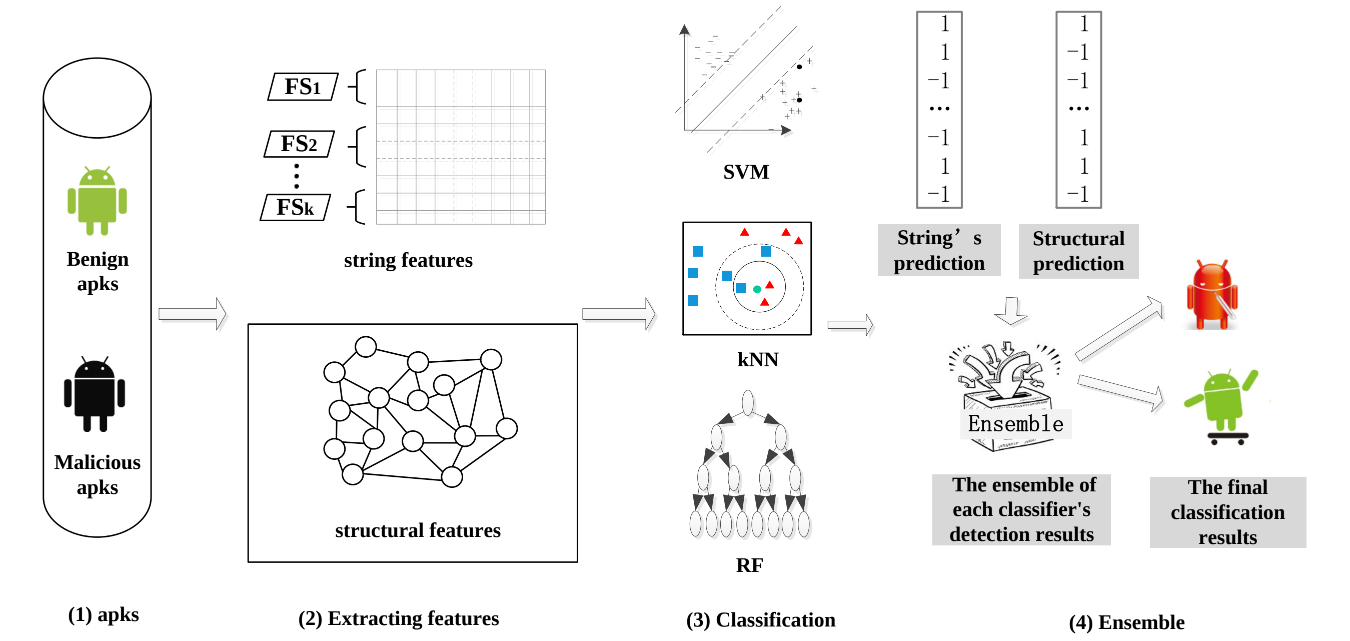

FIGURE 1. Overview of DroidEnsemble.

FS5. *Used Permissions*: Requesting a permission does not mean that the app actually accesses to the corresponding resources. Hence, we extract permissions the app actually used through the API-permission mappings provided by PScout [38] as another feature set.

FS6. *Code Patterns*: Android system does not provide valid authentication and protection mechanism for external loaded resources. Libraries can be malicious for modification and masquerading. Attackers may thus try to hide parts of their apps' malicious functionalities in libraries. We check whether an app executes shell commands, or whether it dynamically loads external files in this feature set. Besides, we check whether an app uses Java reflection techniques or invokes cryptographic functions.

We extract the above 6 types of static features as string features to detect malapps. In addition, we add function call graph features as the structural features. Both types of features are described in table 1.

2) Structural Features

Testing whether two graphs are isomorphic is not easy in polynomial time. We simplify this process by measuring the similarity between two graphs by counting the same number of subgraphs in an app's function call graphs. In this work, we generate isometric encodings for an app's function call nodes based on the Dalvik instructions [39]. We used the method proposed in [30] to identify the benign and malicious apps by calculating the similarity between two apps with the number of the same encodings of two apps. It includes the following three steps.

TABLE 1. Descriptions of string features and structural features

Feature type	Feature set	# of features
String features	FS1. requested permissions	93
	FS2. hardware features	41
	FS3. filtered intents	132
	FS4. restricted API calls	34188
	FS5. used permissions	93
	FS6. code patterns	5
Structural features	FS7. function call graph	/

Step 1. An app is disassembled by apktool [40] and its function call graphs are extracted with androguard [41]. The nodes of the function call graph are then labeled in 15-bit sequence [30].

Formally, the graph is formed as a 4-tuple $G = (N, E, L, l)$ [30], where N is the set of nodes and each node $n \in N$ is associated with one of the app's functions. $E \subseteq N \times N$ denotes the set of directed edges, where an edge from a node n_1 to a node n_2 indicates a call from the function represented by n_1 to the function represented by n_2. L is the multiset of labels in the graph and $l: N \rightarrow L$ is a labeling function, which assigns a label to each node by considering instruction types of the function it contains [30].

As shown in Table 2 [30], we adopt 15 distinct categories of instructions based on their functionality by reviewing the Dalvik specification. Each node can thus be labeled with a

TABLE 2. Instruction categories and their corresponding bit in the node label

Category	Bit	Category	Bit
nop	1	branch	9
move	2	arrayop	10
return	3	instanceop	11
monitor	4	staticop	12
test	5	invoke	13
new	6	unop	14
throw	7	binop	15
jump	8		

15-bit field, where each bit is associated with one of the categories. Formally, the function label l can be defined as follows. The set of instruction categories is represented as $C = \{c_1, c_2, \ldots, c_m\}$ and the bit is set 1 if this type of instruction appears in the function, i.e., $l(n) = [b_1(n), b_2(n), \ldots, b_m(n)]$ where

$$b_c(n) = \begin{cases} 1 & if\ n\ contains\ an\ instruction\ from\ category\ c \\ 0 & otherwise \end{cases}$$

Therefore, the function of each app can be represented by multiple 15-bit encodings [30] and we thus have the initial encoding of the app.

Step 2. In general, there are caller and callee among function nodes. Based on this relationship, for each node, we compute a neighborhood hash over all of its direct neighbors in the function call graph, as suggested in [30]. This computation method is based on the neighborhood hash graph kernel (NHGK) [30], [42].

The computation of the hash for a given node n and its set of adjacent nodes N_n is defined by [30]

$$h(n) = r(l(n)) \oplus \left(\bigoplus_{z \in N_n} l(z)\right) \tag{1}$$

where $\oplus$ represents a bit-wise XOR on the binary labels and r represents a single-bit rotation to the left. It is worthy to note that the computation is conducted in constant time for each node.

Based on this computation, we update the initial encoding and compress the relevant structural information of a function node into a 15-bit encoding [30].

Step 3. With the updated encodings described in Step 2, we calculate the similarity between two apps by counting the number of the same encoding between the two apps. The same encoding indicates that both have the same function structure. In addition, as Android malicious families usually have similar malicious functions, we compute the similarity between the apps with this method and generate the feature matrix for the subsequent classifications.

B. CLASSIFICATION MODELS

In order to describe the apps' behaviors for further analysis, we embed all 6 types of features into a high dimensional feature vector, and embed each call graph in a feature space, respectively.

Based upon the two types of features, we construct three supervised learning models, namely, Support Vector Machine (SVM), k-Nearest Neighbor (kNN) and Random Forest (RF), as these three methods have been widely used for binary classification. Our feature sets are theoretically linearly separable. In practice, SVM performs well for high-dimensional linear separable classification problems. The matrix generated by the function call graph is used to measure the similarity between samples. To facilitate performance comparison, we also kNN and RF for classification.

Support Vector Machine (SVM): SVM is a binary classification model that attempts to find the best linear hyperplane decision boundary that maximizes the margin between two classes. In general, SVM embeds the original features to a higher dimensional feature space using kernel function. In this work, we use linear kernel function to classify the apps with string features staying in the original feature space, and adopt pre-computed kernel function to deal with the matrix generated by structural features.

k-Nearest Neighbor (kNN): Given a test sample, kNN classifier finds the k training samples closest to the test sample in the training set based on distance measures, and then uses the majority voting to predict which class the test sample belongs to.

Random Forest (RF): RF is a classifier that contains multiple decision trees where each tree is learned independently on a randomly selected subset of training data. A subset for training each decision tree is selected by randomly sampling from both features and samples. The final classification is based on the ensemble learning technique.

C. ENSEMBLE OF FEATURES

As mentioned, each string feature and structural feature have their own merits and demerits. Accordingly, a number of malapps can be correctly identified by string features, but not by structural features, and vice versa.

In order to address this problem, we propose DroidEnsemble that improves the detection performance with ensemble of both types of features. In this work, we mark the predict results based on string features as P_{Str}, and the results based on structural features as P_{Fcg}. We define the final detection results P_{Final} as

$$P_{Final} = P_{Str} \cdot W_{Str} + P_{Fcg} \cdot W_{Fcg} \tag{2}$$

where W_{Str} and W_{Fcg} represent the weights of string features and structural features, respectively. As string features are generally more effective and efficient than structural features, we assign a weight of 60% to the prediction result of string features and 40% to the result of structural features.

We judge the detection results through the values of P_{Final}. For benign samples, if the P_{Final} value is equal to -1, our method regards this app as malicious, which is so-called False Positive (FP). Likewise, for malicious samples, if the P_{Final} value is equal to +1, the app is considered as benign one, which is False Negative (FN). The rest values are correspondingly divided into True Positive (TP) and True Negative (TN).

IV. EVALUATION

A. DATA SET

In the experiments, we collect a number of real-world Android apps that include benign apps and malicious apps. We acquired benign apps from four app markets and malapps from various of sources.

Benign apps. We collected benign apps from Google Play and three third-party app markets, i.e., AnZhi [43], LenovoMM [44], Wandoujia [45]. First, we download a large number of apk files from these markets. Second, we scan these apks with an online service called VirusTotal [46] that provides detection service with over 50 antivirus (AV) scan engines. We label an apk as benign only if all AV scanners identified it as benign. Otherwise we consider it as malicious. Finally there are 1386 benign apps remained in our data set.

Malicious apps. In order to ensure the accuracy of our method, we construct a malicious data set similar to the size of the benign data set. We collect malapps from multiple malicious families, such as FakeInst, Opfake, FakeInstaller, DroidKungFu, GinMaster, Plankton. Their malicious behaviors cover stealing phone information, communicating with a C&C, escalating root privilege, repackaged, sending premium-rate SMS and other common ones. Finally, we form 1296 malapps as malicious data set.

B. RESULTS ANALYSIS

We use 6 types of string features and function call graph as the feature set to carry out the experiments. In order to ensure the ensemble of the final prediction results, we do not use the method of n-fold cross-validation. We randomly select 70% of samples to train the SVM, kNN, RF models, and the rest for prediction. We employ pre-computed kernel of SVM for training the structural features and Linear SVM for string features with Python. For kNN and RF, we use tool scikit-learn [47].

The experiments are run on a Lenovo T468 G7 Server with four quad-core 3.10 GHz Xeon processors. In the following Sections, we firstly evaluate the effectiveness of two types of features based on different classifiers, i.e., string features and structural features. Secondly we discuss the performance of these three classifiers. Finally, we analyze and discuss the performance with the ensemble of these two types of features.

1) Features comparison

In this work, accuracy as well as F-score are employed to compare the performance with these two types of features based on different classifiers.

F-score is defined as the harmonic mean of precision and recall:

$$F-score = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \quad (3)$$

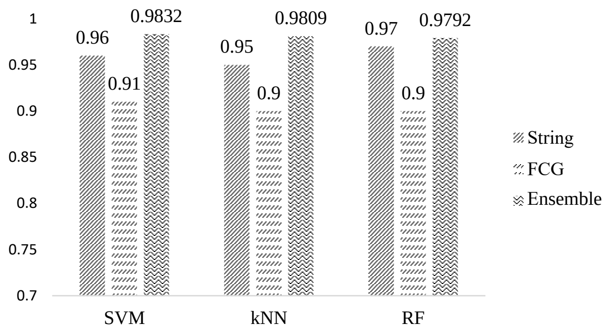

FIGURE 2. Detection performance comparison (F-score) with string features(String), structural features(FCG) and their ensemble.

TABLE 3. Detection accuracy with different types of features

Feature set	SVM	kNN	RF
String features	0.9578	0.9528	0.9702
Structural features	0.9068	0.8994	0.9031
Ensemble of features	0.9839	0.9814	0.9801

where Precision is the proportion of True Positive (TP) to all the positive results, and Recall is also called True Positive Rate (TPR) defined as the proportion of TP in all the positive instances. The F-score of an ideal classifier is close to 1, indicating that the Precision and Recall are both close to 1. Accuracy is the proportion of true results (both True Positive and True Negative) to all the instances.

Fig. 2 presents the F-score values of three types of feature sets with three classifiers. From Fig. 2, it is observed that (1) string features are more effective than structural features for malapp detection, as expected; (2) string features achieve highest F-score based on Random Forest classifier, while structural features perform well using the pre-computed kernel of SVM; (3) the ensemble of string features and structural features achieves the best classification performance.

In Table 3, we compare the detection accuracy with different classifiers based on the same samples. It is seen from the Table that compared to single type of features, the detection accuracy with ensemble of features improves, reaching over 98%.

The above observations show that string features are more effective than structural features for detecting malapps. Meanwhile, structural features make up for the deficiency of string features. The ensemble of two types of features achieves both highest F-score and accuracy based on SVM classifier, indicating that it is more reasonable and effective. Moreover, Android malapps have become increasingly sophisticated and we need to find more effective features to characterize their behaviors from different angles and levels. It's clear that our proposed DroidEnsemble performs well in characterizing the static behaviors of apps and detects Android malapps effectively.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2835654, IEEE Access

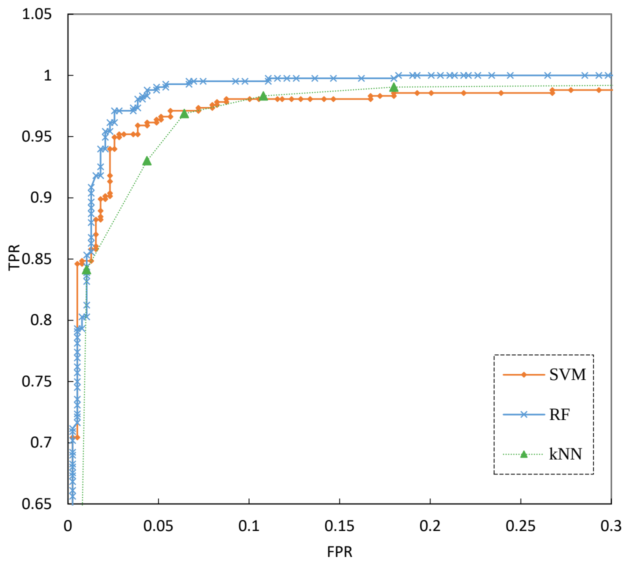

FIGURE 3. ROC curves based on SVM, RF and kNN with string features.

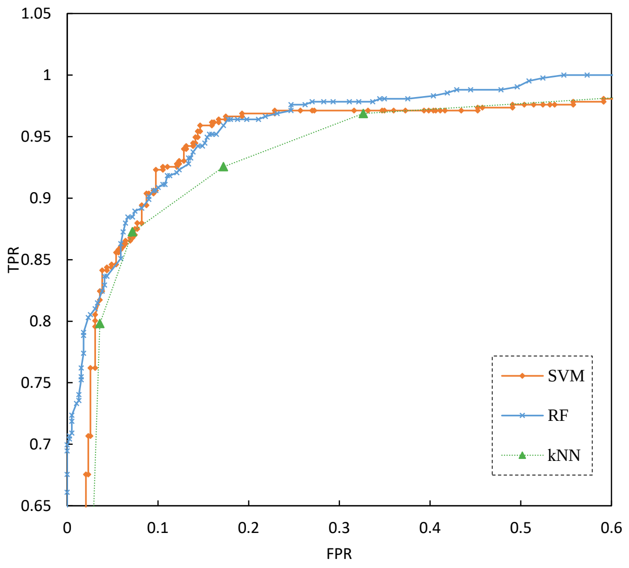

FIGURE 4. ROC curves based on SVM, RF and kNN with structural features.

2) Classifiers' Comparison

In this section, we discuss the performance of three classifiers. As shown in Fig. 3 and Fig. 4, the Receiver Operating Characteristic (ROC) curves for the three classifiers are clearly displayed. It is seen that these three classifiers perform comparably. Our feature space consists of millions of features. SVM performs more efficiently than the other classifiers with small FPR. By analyzing the prediction results, we also find that kNN performs well in distinguishing malapps with function call graph features, due to the great similarity of malapps from the same families. Moreover, RF outperforms the other classifiers.

SVM is more efficient to address high-dimensional separable classification problems. As shown in Table 3, it is seen that SVM achieves the best results with ensemble of two types of features compared to kNN and RF. Therefore, we analyze the detection results generated by the SVM classifier in the next section.

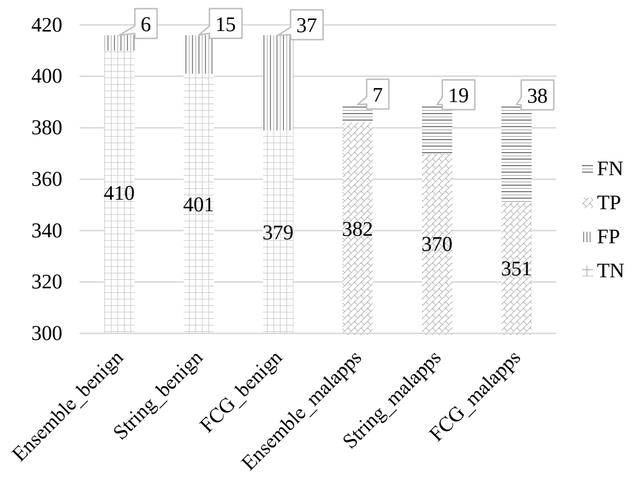

FIGURE 5. Classification results with different types of features. For instance, results for "Ensemble_benign" mean that the number of True Negatives (TNs) for the detection of benign samples with the ensemble of features is 410 and the number of False Positives (FPs) is 6.

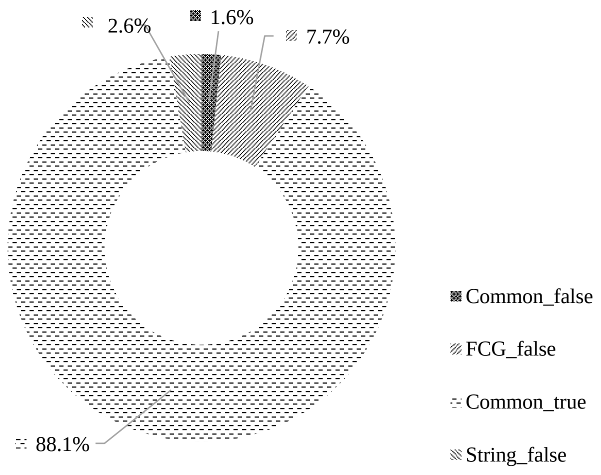

FIGURE 6. Detection results with both types of features.

3) Ensemble results

In this section, we demonstrate the effectiveness of DroidEnsemble. We comprehensively analyze the optimized detection results with SVM.

Fig. 5 shows the superiority of the ensemble of two types of features with SVM. Compared with the detection results using the individual type of features, DroidEnsemble effectively reduces the FPs and FNs. Fig. 6 shows the detection results with both string features and structural features. It is seen that 88.1% of the samples are correctly detected by both types of features and 1.6% are missed. The detection accuracy with string features is up to 96%. In contrast, structural features perform noticeably poorer. However, 2.6%

of samples that are not identified by string features can be identified by structural features. Therefore, we obtain the final optimal classification results by weighting the respective detection results with the two types of features.

TABLE 4. Detection results with ensemble of features

Predicted as →	Benign	Malicious
Benign	98.56%	1.44%
Malicious	1.80%	98.20%

The detection result P_{Final} with the ensemble of two types of features are shown in table 4. It is clear that DroidEnsemble achieves satisfied detection accuracy, reaching over 98%.

Among the false negatives (FNs), we find that the program function of these samples are relatively simple and some of their malicious behaviors are subtle, such as root exploits, stealing cookies, etc. On the one hand, the detection results with string features rely on the 6 types of features. The FNs may not involve many sensitive features. Therefore they are falsely considered as benign apps. On the other hand, in reality some benign apps' rich functions require many sensitive features, such as SEND_SMS, ACCESS_FINE_LOCATION, READ_CONTACTS and so on, which may lead to incorrect classification results. Unlike string features, structural features adopt the similarities between the apps. Malapps' behaviors are similar to their malicious families while benign apps' behaviors are diverse. Therefore, structural features can make up for the deficiency of string features in terms of malapp detection. For example, the malapp with sha1 value of 5a1eb830dd953a4cbc3c549ed9736d61ae5add54 is falsely identified as normal by string features. By examining the 6 types of features, we find that this app only uses a few common but sensitive permissions, such as ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE and RECEIVE_BOOT_COMPLETED, and no other obvious malicious features are found in its feature sets. Therefore, string features easily judge it as a benign app. In contrast, structural features can effectively identify it as malicious. The app belongs to the malicious family called FaceNiff, and its malicious behaviors are similar to the malicious family. Therefore it is correctly judged as a malapp with structural features. On the contrary, for 0c80bce773e2afda6802d6f0b2de8de6d1825713.apk, as it requests and uses all SMS-related permissions and the restricted API file also shows that it invokes the corresponding functions, the string features easily identify it as malicious. However, the malicious family "FakeInstaller" has many variants, which may result in a misclassification. As a consequence, we take advantages of both types of features and synthesize them for the detection, resulting in a better detection accuracy as over 98%, which demonstrates the effectiveness of DroidEnsemble.

V. LIMITATIONS

Although DroidEnsemble has demonstrated its ability to improve the detection performance with the ensemble of string and structural features, it still has inevitable limitations.

First, DroidEnsemble is based on static analysis and lacks the capabilities of run-time analysis, a.k.a dynamic analysis. Some malapps make use of anti-decompiling or obfuscation techniques to prevent feature extraction, or load code dynamically to hinder the static inspection. In order to reduce the impact of the absence of dynamic analysis and accurately characterize the behaviors of apps, we extract both string features and structural features from apps. In string features, we check whether an app dynamically loads external executable files or Linux native code, which may reflect some malicious behaviors, although we do not deeply analyze these codes. In addition, we utilize the structural features that encode the method's instruction sequence to resist the obfuscated code. However, structural features are not valid for junk code. Furthermore, although the structural features make up for deficiency of string features, the extraction of structural feature is very time-consuming.

Second, the accuracy of machine leaning algorithms relies on the data set used to train the model. The quality of the data set is thus important to determine whether the detection models are generally effective. In our experiments, we choose benign apps from multiple markets to make them as representative as possible, and malapps from different families to ensure the diversity. However, as mentioned, it is not trivial to choose suitable malicious and benign apps. We need to scan our samples with VirusTotal to guarantee the pureness of both malicious and benign apps. However, VirusTotal has its own limitations. Besides, there are some other factors that may influence the results, such as the size of apks, the apps from the same market or the same family, etc.

VI. CONCLUSION

Vetting and detecting malapps help to purify the app markets and to reduce app failures in Android systems. In this work, we propose DroidEnsemble that systematically and comprehensively characterizes the static behaviors of Android apps for the detection of malapps with ensemble of string and structural features. We employ three machine learning methods, namely, Support Vector Machine (SVM), including pre-computed kernel for structural features and linear kernel for string features, k-Nearest Neighbor (kNN) and Random Forest (RF) in the detection. We compare the detection performance with only string features or only structural features. The extensive experimental results demonstrate the effectiveness of DroidEnsemble, showing that (1) string features are more effective than structural features for detecting malapps; (2) structural features make up for deficiency of string features and thus can be used as complementary features in the detection; (3) the ensemble of both types of features outperforms any individual feature, yielding the best accuracy as 98%.

In our future work, we will investigate how the structural features and string features can be put into an uniform feature space so that they can be simultaneously processed in learning algorithms. As for some sophisticated malapps that use root, encryption, anti-disassembly, or kernel-level features to evade the detection, DroidEnsemble may not be able to detect them. In our future work, we are exploring more features, in particular the dynamic features, to better characterize the behaviors of apps from different angles and layers to improve the detection performance. We are also planing to collect more qualified app samples to test DroidEnsemble. Meanwhile, due to the fast increase of Android apps, we are developing semi-supervised learning methods for the detection.

ACKNOWLEDGEMENTS

The work reported in this paper is supported in part by National Key R&D Program of China, under grant 2017YFB0802805, in part by Shanghai Key Laboratory of Integrated Administration Technologies for Information Security, under Grant AGK2015002, in part by ZTE Corporation Foundation, under Grant K17L00190, in part by Funds of Science and Technology on Electronic Information Control Laboratory, under Grant K16GY00040, in part by the Fundamental Research funds for the central Universities of China, under grant K17JB00060 and K17JB00020, and in part by Natural Science Foundation of China, under Grant U1736114 and 61672092.

REFERENCES

[1] "Gartner data report," http://www.feng.com/iPhone/news/2017-05-24/IOS-and-Android-global-share-gap-is-more-bigger_679074.shtml.

[2] "360 security report," http://zt.360.cn/1101061855.php?dtid=1101061451&did=210412109.

[3] J. Song, C. Han, K. Wang, J. Zhao, R. Ranjan, and L. Wang, "An integrated static detection and analysis framework for android," Pervasive and Mobile Computing, vol. 32, pp. 15–25, 2016. [Online]. Available: https://doi.org/10.1016/j.pmcj.2016.03.003

[4] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," IEEE Trans. Information Forensics and Security, vol. 9, no. 11, pp. 1869–1882, 2014. [Online]. Available: https://doi.org/10.1109/TIFS.2014.2353996

[5] G. Dai, J. Ge, M. Cai, D. Xu, and W. Li, "Svm-based malware detection for android applications," in Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015, 2015, pp. 33:1–33:2. [Online]. Available: http://doi.acm.org/10.1145/2766498.2774991

[6] S. Sheen, R. Anitha, and V. Natarajan, "Android based malware detection using a multifeature collaborative decision fusion approach," Neurocomputing, vol. 151, pp. 905–912, 2015. [Online]. Available: https://doi.org/10.1016/j.neucom.2014.10.004

[7] K. Xu, Y. Li, and R. H. Deng, "Iccdetector: Icc-based malware detection on android," IEEE Trans. Information Forensics and Security, vol. 11, no. 6, pp. 1252–1264, 2016. [Online]. Available: https://doi.org/10.1109/TIFS.2016.2523912

[8] W. Wang, Y. Li, X. Wang, J. Liu, and X. Zhang, "Detecting android malicious apps and categorizing benign apps with ensemble of classifiers," Future Generation Comp. Syst., vol. 78, pp. 987–994, 2018. [Online]. Available: https://doi.org/10.1016/j.future.2017.01.019

[9] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, 2014, pp. 175–186. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568286

[10] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014, 2014, pp. 1329–1341. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660357

[11] Q. Lin, J. Li, Z. Huang, W. Chen, and J. Shen, "A short linearly homomorphic proxy signature scheme," IEEE Access, vol. PP, no. 99, pp. 1–1, 2018.

[12] Q. Lin, H. Yan, Z. Huang, W. Chen, J. Shen, and Y. Tang, "An id-based linearly homomorphic signature scheme and its application in blockchain," IEEE Access, vol. PP, no. 99, pp. 1–1, 2018.

[13] J. Shen, Z. Gui, S. Ji, J. Shen, H. Tan, and Y. Tang, "Cloud-aided lightweight certificateless authentication protocol with anonymity for wireless body area networks," Journal of Network and Computer Applications, vol. 106, pp. 117 – 123, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804518300031

[14] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye, "Significant permission identification for machine learning based android malware detection," IEEE Transactions on Industrial Informatics, pp. 1–1, 2018.

[15] T. Li, J. Li, Z. Liu, P. Li, and C. Jia, "Differentially private naive bayes learning over multiple data sources," Information Sciences, vol. 444, pp. 89 – 104, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025518301415

[16] Z. Huang, S. Liu, X. Mao, K. Chen, and J. Li, "Insight of the protection for data security under selective opening attacks," Information Sciences, vol. 412-413, pp. 223 – 241, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025517302177

[17] D. Xie, X. Lai, X. Lei, and L. Fan, "Cognitive multiuser energy harvesting decode-and-forward relaying system with direct links," IEEE Access, vol. PP, no. 99, pp. 1–1, 2017.

[18] J. Wu, M. Yang, and T. Luo, "Pacs: Pemission abuse checking system for android applictions based on review mining," in IEEE Conference on Dependable and Secure Computing, 2017, pp. 251–258.

[19] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell, "Androdialysis: Analysis of android intent effectiveness in malware detection," Computers & Security, vol. 65, no. C, pp. 121–134, 2016.

[20] F. Idrees, M. Rajarajan, M. Conti, T. M. Chen, and Y. Rahulamathavan, "Pindroid: A novel android malware detection system using ensemble learning methods," Computers & Security, vol. 68, pp. 36–46, 2017. [Online]. Available: https://doi.org/10.1016/j.cose.2017.03.011

[21] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. Mcdaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in ACM Sigplan Conference on Programming Language Design and Implementation, 2014, pp. 259–269.

[22] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications," in Computer Security - ESORICS 2014, M. Kutyłowski and J. Vaidya, Eds. Cham: Springer International Publishing, 2014, pp. 163–182.

[23] X. Chen and S. Zhu, "Droidjust: automated functionality-aware privacy leakage analysis for android applications," in Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015, 2015, pp. 5:1–5:12. [Online]. Available: http://doi.acm.org/10.1145/2766498.2766507

[24] S. Feng, "Android security via static program analysis," in The Workshop on MOBISYS 2017 Ph.d. Forum, 2017, pp. 19–20.

[25] S. Hou, A. Saas, L. Chen, Y. Ye, and T. Bourlai, "Deep neural networks for automatic android malware detection," in Ieee/acm International Conference, 2017, pp. 803–810.

[26] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "Droidsieve: Fast and accurate classification of obfuscated android malware," in ACM on Conference on Data and Application Security and Privacy, 2017, pp. 309–320.

[27] X. Wang, W. Wang, Y. He, J. Liu, Z. Han, and X. Zhang, "Characterizing android apps' behavior for effective detection of malapps at large scale," Future Generation Comp. Syst., vol. 75, pp. 30–45, 2017. [Online]. Available: https://doi.org/10.1016/j.future.2017.04.041

[28] M. Salehi, F. Daryabar, and M. H. Tadayon, "Welcome to binder: A kernel level attack model for the binder in android operating system," in International Symposium on Telecommunications, 2017, pp. 156–161.

[29] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in European Symposium on Research in Computer Security, 2012, pp. 37–54.
[30] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in Proceedings of the 2013 ACM workshop on Artificial intelligence and security, 2013, pp. 45–54.
[31] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "Droidnative : automating and optimizing detection of android native code malware variants," Computers & Security, vol. 65, pp. 230–246, 2016.
[32] J. Kim, T. G. Kim, and E. G. Im, "Structural information based malicious app similarity calculation and clustering," in Conference on Research in Adaptive and Convergent Systems, 2015, pp. 314–318.
[33] K. Dam and T. Touili, "Learning android malware," in Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017, 2017, pp. 59:1–59:9. [Online]. Available: http://doi.acm.org/10.1145/3098954.3105826
[34] M. Leeds, M. Keffeler, and T. Atkison, "A comparison of features for android malware detection," in Southeast Conference, 2017, pp. 63–68.
[35] N. Milosevic, A. Dehghantanha, and K. R. Choo, "Machine learning aided android malware classification," Computers & Electrical Engineering, vol. 61, pp. 266–274, 2017. [Online]. Available: https://doi.org/10.1016/j.compeleceng.2017.02.013
[36] H. A. Alatwi, T. Oh, E. Fokoue, and B. Stackpole, "Android malware detection using category-based machine learning classifiers," in Conference on Information Technology Education, 2016, pp. 54–59.
[37] D. Arp, M. Spreitzenbarth, M. HÍźbner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in Network and Distributed System Security Symposium, 2014.
[38] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in ACM Conference on Computer and Communications Security, 2012, pp. 217–228.
[39] "Dalvik bytecode," http://www.feng.com/iPhone/news/2017-05-24/IOS-and-Android-global-share-gap-is-more-bigger_679074.shtml.
[40] "Apktool," https://ibotpeaches.github.io/Apktool/.
[41] A. Desnos, "Androguard-reverse engineering, malware and goodware analysis of android applications," http://code.google.com/p/androguard/,2013.
[42] S. Hido and H. Kashima, "A linear-time graph kernel," in 2009 Ninth IEEE International Conference on Data Mining, Dec 2009, pp. 179–188.
[43] "Anzhi market," http://www.anzhi.com/.
[44] "Lenovomm market," http://www.lenovomm.com/.
[45] "Wandoujia market," http://www.wandoujia.com/.
[46] "Virustotal-free online virus, malware and url scanne," https://www.virus-total.com/.
[47] F. Pedregosa, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, and J. Vanderplas, "Scikit-learn: Machine learning in python," Journal of Machine Learning Research, vol. 12, no. 10, pp. 2825–2830, 2016.

• • •