RepassDroid: Automatic Detection of Android Malware Based on Essential Permissions and Semantic Features of Sensitive APIs

Niannian Xie*, Fanping Zeng+, Xiaoxia Qin*, Yu Zhang+, Mingsong Zhou* and Chengcheng Lv*
School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China
Anhui Province Key Lab of Software in Computing and Communication, Hefei, Anhui, PR China
Email: *{xnn,qinxx,mingsong,lvcc}@mail.ustc.edu.cn, +{billzeng,yuzhang}@ustc.edu.cn

***Abstract*—Most current literature on Android malware pays particular attention to the features of applications. Much of them focus on permissions or APIs, neglecting the behavioral semantics of applications, and the literature considering behavioral semantics is often expensive and weak in extendibility. In this paper, we introduce RepassDroid – a relatively coarse-grained but faster tool for automatic Android malware detection. We define Generalized-sensitive API and emphasize on considering if the trigger points of generalized-sensitive APIs are UI-related or not. It analyzes the application by abstracting the generalized sensitive API with its trigger point as the semantic feature, with the addition of Really-essential Permission as the syntax feature. Then it utilizes machine learning to automatically determine whether an application is benign or malicious. We evaluate RepassDroid on 24288 samples in total, 20000 for training and 4288 for test. With the comparative experiments, we find that Random Forest is the optimal classification technique for our feature set, achieving 97.7% accuracy and 0.99 AUC, along with a malware classification precision as high as 99.3%. Our evaluation results confirm that our approach and the feature set are logical and effective for Android malware detection.**

Keywords- Android malware detection; Static analysis; Semantic features; Machine learning

I. INTRODUCTION

According to the global mobile device report, Android device accounts for 87.7% of the market. The huge Android market attracts lots of malware developers. With the security consciousness, Android permission mechanism enforces that each application declares permissions in AndroidManifest.xml to indicate whether it has the ability to access relative sensitive resources. Whereas applications may request excessive permissions and malware may also request useless normal permissions to hide their own maliciousness. For this reason, judging the function of applications simply through the requested permissions is not completely reliable.

Over the past decade, research into Android malware detection can be summarized into three categories. (1) Dynamic analysis [1–4] needs to run applications, costing long time and plenty of resources, and provides limited code coverage. (2) Without running the application, static analysis [5–8] is faster and easier, with higher code coverage. Nevertheless, it cannot easily analyze the programs when it meets dynamic code, code confusion and so on. (3) Machine learning based detecting technology [9–12] can construct a learning-based classification model through a big dataset. The key of this technology is to seek out appropriate feature set, like permissions, APIs and behavioral semantic features. Notably, the main difference between benign and malicious applications is that the latter invoke sensitive APIs in an unexpected context, not under the control of users. AppContext [9] extracted sensitive behaviors of applications to construct the training set by abstracting their context semantics, activation events and environmental conditions, to analyze whether the data flow is malicious. There is no doubt that it is fine-grained but complicated, and the overhead is too high. Due to the rapid evolution of the malware, learning-based detecting technology must regularly retrain the classification model through new datasets. However, the high computational cost causes that the extendibility of AppContext is not strong. Drebin [11] only considered the syntax features like permissions and APIs, neglecting the semantic features of applications. Nevertheless, permissions and APIs are the basis for performing malicious operations. Inspired by this, we propose a relatively coarse-grained but much cheaper and effective approach to detect Android malware.

This paper proposes ***RepassDroid*** – a tool combining ***R***eally-***E***ssential Permissions and sensitive APIs with trigger points to construct ***S***yntax and ***S***emantic features by static analysis, then leveraging machine learning to find the optimal classifier for malware detection. Specifically, we define Generalized-sensitive API, including the permission-protecting API and two kinds of Approximately-sensitive API we defined. We regard the generalized-sensitive API with their trigger points as the semantic feature. In order to analyze applications more holistically, we also analyze APIs in the program to find out the Really-essential Permission used by applications as the syntax feature.

In summary, our contributions are as follows:

- We propose a new representation of the semantic feature to help detect Android malware: the Generalized-sensitive API as well as its trigger point, highlighting if the trigger point is UI-related.
- We extract the Really-essential permission to represent the syntax feature, including the permission of sensitive APIs and that of approximately-sensitive APIs.
- Combining syntax and semantic features, we implement RepassDroid, a tool for automatic Android malware detection. The experiments verify that our approach has high efficiency and low cost, with the classifier achieving 97.7% accuracy and 0.99 AUC.

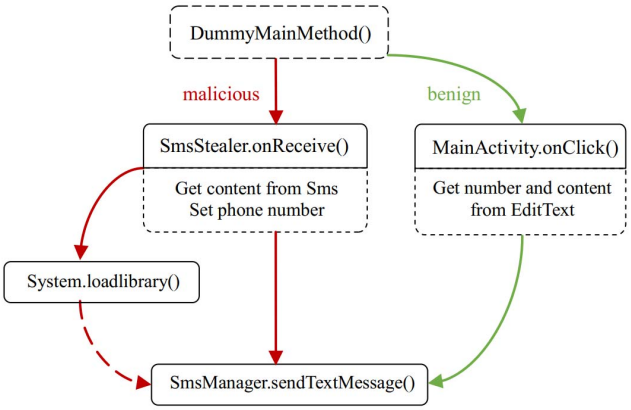
978-1-5386-7305-8/18/$31.00 ©2018 IEEE
DOI 10.1109/TASE.2018.00015

The remainder of this paper is organized as follows. Section II introduces the motivating example. Section III presents RepassDroid in detail. Section IV talks about our experiments to evaluate RepassDroid and the feature set. Section V analyzes the limitations of our work and the future work while Section VI introduces the related work in three aspects. The last section makes a conclusion for this paper.

II. MOTIVATION EXAMPLE

```
1  <application
2    <activity android:name=".MainActivity">
3      <intent-filter>
4        <action android:name="android.intent.action.MAIN" />
5        <category android:name="android.intent.category.LAUNCHER" />
6      </intent-filter>
7    </activity>
8    <receiver android:name=".SmsStealer">
9      <intent-filter>
10       <action android:name="android.intent.action.DATE_CHANGED"/>
11     </intent-filter>
12   </receiver>
13 </application>
14 <uses-permission android:name="android.permission.READ_SMS"/>
15 <uses-permission android:name="android.permission.SEND_SMS"/>
```

(a) Code snippet of the malware's AndroidManifest.xml



(b) Comparison of simplified CG of the malicious and the benign

Figure 1. A simplified example of a SMS Trojan

Figure 1 gives an example of the SMS-stealing malware. As is shown in lines 8-12 in Figure 1(a), this malware registers a broadcast event in the component SmsStealer. Once the date changes, the DATE_CHANGED broadcast will trigger the method SmsStealer.onReceive(). It will resolve users' text messages and send them to the phone number set beforehand by the developer so that the malware can steal SMS.

API SmsManager.sendTextMessage() is sensitive, so it needs to request the permission android.permission.SEND_SMS in AndroidManifest.xml. Obviously, the SMS-sending behavior is triggered by method onReceive(), as shown in the red straight route in Figure 1(b). Whereas under benign circumstance, as shown in the green route in Figure 1(b), a SMS-sending event should be triggered by the user clicking the SMS-sending button, and both the content and recipient of message should be obtained from the edit box on the user interface as well. Generally, the benign trigger point should be onClick() or some other UI-related callback method while the malicious case may be triggered by the UI-unrelated. That's the reason why we underline to consider if the trigger point is UI-related or not. Besides, malware often invokes malicious code in the form of dynamic code. As shown in the red curved route in Figure 1(b), it utilizes System.loadlibrary() to load a third-party dynamic library to achieve the SMS-sending operation, where the sensitive API SmsManager.sendTextMessage() cannot be analyzed by static analysis. Therefore, we treat dynamic code related APIs and some other methods involving sensitive data as the sensitive API as well.

III. REPASSDROID

A. Overview

The overall architecture of RepassDroid is shown in Figure 2, which is comprised of two modules: Feature Extraction Module and Training Learning-based Classifier Module. First, we generate the call graph (CG) of applications with the help of FlowDroid [5]. After that, we extract features of applications from CG to form the feature vectors. Then we apply the training dataset to train the machine learning-based classification model by Weka [14], aiming to find the optimal classifier. For the unknown application, we extract their features, and input them into the trained classification model to determine whether they are benign or malicious. The following is demonstrated based on the two modules.

B. Feature Extraction Module

Because the malicious sensitive behavior is often operated without users' consciousness while the benign sensitive behavior is under the control of users, we abstract sensitive APIs with corresponding trigger points to represent behaviors. With the consideration of comprehensiveness, we also need to find out the really-essential permission used by applications through analyzing APIs in the program. In a word, we take two types of features into account, that is, semantic features and syntax features.

*1) **Semantic Features**:*

We define that the semantic feature $S^{\{T \rightarrow G\}}$ consists of two parts: *Generalized-sensitive API* and the corresponding trigger point. Now, we will describe them in detail.

- *Generalized-sensitive API G:*

Generally, the sensitive API refers to permission-protecting APIs. To analyze applications more accurately and smooth over the limitation of static analysis, we define the generalized-sensitive API to contain three types:

(1) *APIs Protected by Sensitive Permissions.*

PScout [13] summarizes the match between permissions in Android and their protecting APIs. We have gathered the results of PScout under five versions of Android system, contributing for a more complete match between permissions and APIs. Based on the aggregated result of PScout, finally we extract the

match between 59 sensitive permissions and corresponding APIs.

(2) *Dynamic Code Related Methods.*

Many malware seem not to contain malicious intentions apparently, but actually they may load malicious code from external to avoid detection. Static analysis cannot accurately handle such reflection and dynamic code-loading methods while they often contain sensitive APIs. Therefore, we treat them as sensitive APIs as well. They mainly involve methods in class *DexFile*, *ClassLoader*, *Class*, *Constructor*, *Method* and *Field* [15–17]. We also consider other APIs that are often invoked by malware as sensitive APIs, such as methods in class *Crypto.Cipher* for code obfuscation, the method for loading libraries *System.loadLibrary()*, the method for shell command *Runtime.exec()* and so on.

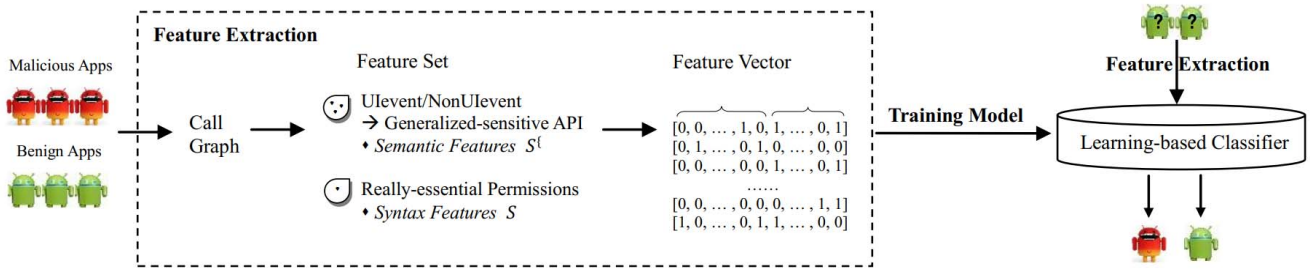


Figure 2. Overall Architecture of RepassDroid

(3) *Source and Sink Methods.*

As we all know, source and sink methods involve sensitive resources as well, but they are not entirely equal to the permission-protecting APIs. Most methods in class *Intent* are sources or sinks. For instance, *Intent.getStringExtra()* is a source to retrieve extended data from the intent, but it does not require any permission to be invoked. Apart from this, we also take the other source or sink methods into account, involving class *URL*, *Activity*, *IO*, etc. SUSI [18] has counted source and sink methods in Android system. With the help of the result of SUSI, we extract source and sink methods other than permission-protecting APIs.

We denote the permission-protecting APIs as A while the latter two types of methods are named *Approximately-sensitive API*, denoted as $\tilde{A}$, so $G = A \cup \tilde{A}$. In consequence, we collect 10815 generalized-sensitive APIs in all: the number of the first type is 10476, the second type is 290 and the third is 49.

- *Trigger Point T:*

We emphasize whether the sensitive API is triggered by users to consider the security of behaviors, so there is no need to focus on the method signature of the trigger point. It can also reduce feature redundancy. Instead, we abstract it into two values: *UIevent* and *NonUIevent*, representing the UI-related and the UI-unrelated callback method respectively, that is T ={*UIevent, NonUIevent*}. For example, *onClick()* and *onReceive()* in Figure 1 are respectively represented as *UIevent* and *NonUIevent* in the semantic feature.

When we click UI, the method *onClick()* within class *android.view* will be invoked. Hence, in order to find the UI-related callback, we need to analyze whether its class is *android.view* or the subclass of it. Finally, we calculate out 88 UI-related trigger points in all.

Algorithm 1 represents the detailed process of extracting the semantic feature.

In line 1, we parse the APK by FlowDroid to generate the call graph $cg = (N, E)$, so that we can get the trigger point and their reachable sensitive APIs from CG. All the trigger points of sensitive APIs must be a subset of the entry point set $\tilde{T}$. Therefore, we find out all the entry point and then check if their reachable nodes are sensitive APIs. The following procedures can be divided into two parts.

In lines 2-13, it finds all the entry points and their reachable nodes from CG. Specifically, in lines 2-5, for each $n_{src} \rightarrow n_{dest} \in E$ in CG, if the source node n_{src} is neither a dummy main method nor a destination of any edge, it is put into the entry point set $\tilde{T}$. The destination node n_{dest} joins into the reachable node set of n_{src} namely $n_{src}.DestNode$. Then in lines 6-10, for each entry point $n_{src} \in \tilde{T}$, we search for all its reachable nodes in CG, no matter how many steps passed. If there exists $n_{src} \rightarrow n_{dest} \rightarrow n_{dest'} \in E$, we add $n_{dest'}$ into $n_{src}.DestNode$, too. The loop terminates until we traverse the CG to the leaf node.

In lines 14-22, it finds all the trigger points with their reachable sensitive API nodes and then formalizes them in the form of the semantic feature. To be specific, in lines 14-15, for each entry point $n_{src} \in \tilde{T}$, we determine whether its reachable API is generalized-sensitive. Then in lines 16-20, we determine whether each trigger point n_{src} is UI-related or UI-unrelated, and respectively represent the match between n_{src} and its reachable sensitive API $g \in G$ as $UIevent \rightarrow g$ or $NonUIevent \rightarrow g$.

In line 23, after the above traversal process, we get the semantic feature set $S^{\{T \rightarrow G\}}$ of the APK. Each sensitive API $g \in G$ is denoted in the form of specific method signature.

For example, the three cases in Figure 1 abstracted into the semantic feature are (short for the API signature): *UIevent→sendTextMessage()*, *NonUIevent → sendTextMessage()* and *NonUIevent → loadLibrary()*.

*2) **Syntax Features**:*

We define the syntax feature $S^{\{P\}}$ represented by *Really-essential Permission*, including the following two types:

(1) *Permissions of Sensitive APIs A.*

We extract the really-essential permission used in programs to form the syntax feature because permissions requested in Androidmanifest.xml are not all used by applications, which may confuse the detecting process. First, we summarize the results of PScout under five versions of Android, which are the match between each permission (including normal permissions) and APIs, with 24147 matching results aggregately. Then we can collect all the really-essential permissions of all the reachable sensitive APIs in applications. Taking *SmsManager.sendTextMessage()* in Figure 1 as an example, its permission is *SEND_SMS*.

Algorithm 1: Extracting Semantic Feature Set

Input: APK α
Generalized-sensitive API Set $G = A \cup \tilde{A}$
UI-related Tigger Point Set U
Output: Entry Point Set $\tilde{T}$
Semantic Feature Set $S^{\{T \to G\}}$

```
CG cg = (N, E) ← FlowDroid parse α;
foreach n_src → n_dest ∈ E, n_src, n_dest ∈ N do
    if n_src.srcNode=null & n_src∉DummyMain then
        T̃.addNode(n_src);
        n_src.DestNode.addNode(n_dest);
        while n_dest ∉ LeafNode do
            foreach n_dest→n_dest' ∈E, n_dest, n_dest' ∈N do
                n_src.DestNode.addNode(n_dest');
            end
            n_dest=n_dest';
        end
    end
end
foreach n_src ∈ T̃ do
    foreach g ∈ n_src.DestNode & g ∈ G do
        if n_src ∈ U then
            S^{T→G}.addFeature(UIevent → g);
        else
            S^{T→G}.addFeature(NonUIevent → g);
        end
    end
end
return S^{T→G};
```

(2) *Permissions of Approximately-sensitive APIs* $\tilde{A}$.

Since we have defined two types of approximately¬ sensitive APIs but there are no corresponding permissions in Android system, we need to define *Approximate Permission* for them to keep the comprehensiveness of analysis. We did not use categories specified in SUSI due to its incomplete results. Otherwise, we count and define the approximate permission manually based on the functions and class names of approximately-sensitive APIs. Taking *System.loadlibrary()* in Figure 1 as an example, *LoadLibrary* is its approximate permission.

C. Training Learning-based Classifier Module

For the sake of classifying Android applications as the malicious and the benign, we formulate the malware detection as a classification problem. With the feature set generated from the feature extraction module, we resort to tool Weka [14] and try a number of classical machine learning techniques to train the optimal classification model, including C4.5 Decision Tree, Random Forest, Support Vector Machine, K-Nearest Neighbor and Naive Bayes.

IV. Experimental Evaluations

We conduct our experiments on a machine with Intel Core i5-4460 3.20GHz CPU, 16.0GB RAM and Windows 10 operating system. Based on FlowDroid, PScout and SUSI, we utilize Java with JDK 8.0 and Eclipse to extract features. We use 1 to represent the feature exists and use 0 to represent the opposite. After obtaining the feature matrix, we put it into Weka to train the optimal classification model. We utilize 10-fold cross validation to train RepassDroid and evaluate effectiveness of it and the feature set.

We have conducted three groups of experiments and will discuss the following research questions:

RQ1: How effective is RepassDroid in classifying Android applications and detecting malware? Which machine learning technique is the best for our feature set?

RQ2: How do the generalized-sensitive API and feature set contribute to the effectiveness of malware detection? What about the representation of the feature set?

RQ3: Compared with the Android malware detecting tool Drebin, how is RepassDroid? How does it compare with the detecting tools on the website VirusTotal?

A. Dataset and Evaluative Criteria

Our dataset contains 24288 applications (the samples whose parsing time is more than 10 minutes have been removed), 20000 for training and 4288 for test. Benign apps and malicious apps are each half of the training set. We analyzed 12086 Google Play applications to constitute the benign sample from the website AndroZoo [19] and 12202 malicious applications to construct the malicious sample from Android Malgenome Project [20], VirusShare [21] and Drebin[11]. The average time for analyzing an application is about 60 seconds, and the specific analysis time is decided by the size of application.

In summary, the feature set *S* collected from sample APKs has totally 871 features, including 811 semantic features and 60 syntax features (including 46 sensitive permissions and 14 approximate permissions).

$$\overbrace{\underbrace{S^{\{T \to G\}}}_{811} \bigcup \underbrace{S^{\{P\}}}_{60}}^{S}$$

In the experiments, the evaluative criteria we employed are as follows:

Accuracy = (*TP* + *TN*)/(*TP* + *TN* + *FP* + *FN*),
FPR = *FP*/(*FP* + *TP*), *FNR* = *FN*/(*FN* + *TP*),
TNR = *TN*/(*TN* + *FP*),
TPR = *Recall* = *TP*/(*TP* + *FN*),
Precision = *TP*/(*TP* + *FP*),
F1 = 2·*P*·*R*/(*P* + *R*), *AUC* = *Area*(*ROC*).

Recall(*R*) and *Precision*(*P*) are often contradictory, so we resort to *F*1 to do an integrated evaluation.

ROC curve represents the generalization performance of the classifier. *AUC* represents the area of ROC curve, the bigger the *AUC*, the better the classifier.

The positive is the malicious whilst the negative is the benign. *TP* is true positive, *FP* is false positive, *TN* is true negative, *FN* is false negative. In the following figures of evaluation results, *P_* represents the positive and *N_* represents the negative.

B. RQ1: Overall Effectiveness of RepassDroid

Now that there is no absolutely optimal machine learning classification technique, we employed several classification techniques in Weka to conduct comparative experiments to find the optimal scheme of our dataset, including K-Nearest Neighbour (KNN), Naive Bayes (NB), C4.5 Decision Tree (C4.5), Random Forest (RF) and Support Vector Machine (SVM). The comparative classification performance is shown in Table I.

TABLE I. COMPARATIVE PERFORMANCE OF DIFFERENT CLASSIFIERS

Indicatior	RF	SVM	C4.5	KNN	NB
Accuracy	97.7%	93.1%	95.8%	96.7%	87.5%
P_Recall	96.3%	89.8%	95.7%	96.9%	82.4%
N_Recall	99.2%	96.6%	96.0%	96.5%	92.9%
P_Precision	99.3%	96.6%	96.2%	96.6%	92.4%
N_Precision	96.2%	90.0%	95.5%	96.7%	83.3%
P_F1	97.8%	93.1%	95.9%	96.8%	87.1%
N_F1	97.7%	93.2%	95.7%	96.6%	87.8%
FPR	0.8%	3.4%	4.0%	3.5%	7.1%
FNR	3.7%	10.2%	4.3%	3.1%	17.6%
AUC	0.99	0.93	0.97	0.98	0.94

Results. Table I shows that RF is the optimal classification technique for our feature set, achieving 97.7% *accuracy* and 0.99 *AUC*, along with a *FPR* as low as 0.8%. The malware classification precision *P_precision* is as high as 99.3% and both the *P_F*1 and *N_F*1 of RF are the highest, too.

C. RQ2: Effectiveness of the Feature Set

In order to evaluate the effectiveness of the feature set, we conduct comparative experiments in three aspects.

1) Effectiveness of the generalized-sensitive API:

In this part, we explore whether the generalized-sensitive API we defined is logical or not. We denote the three kinds of generalized-sensitive API as A, $\tilde{A}_1$, $\tilde{A}_2$, respectively. Here $G = A \cup \tilde{A} = A \cup \tilde{A}_1 \cup \tilde{A}_2$. Then, we employed Random Forest to conduct three experiments with respective feature set $S^{\{T \to A\}} \cup S^{\{p\}}$, $S^{\{T \to A+\tilde{A}_1\}} \cup S^{\{p\}}$, $S^{\{T \to A+\tilde{A}_1+\tilde{A}_2\}} \cup S^{\{p\}}$. The result is shown in Figure 3, where the three feature sets are denoted as A, $A+\tilde{A}_1$, $A+\tilde{A}_1+\tilde{A}_2$.

Results. Figure 3 illustrates when the sensitive API G in $S^{\{T \to G\}}$ is $A+\tilde{A}_1+\tilde{A}_2$, the classification result is the best. That is to say the generalized-sensitive API we defined is quite reasonable, which can help to improve the result of classifying Android applications.

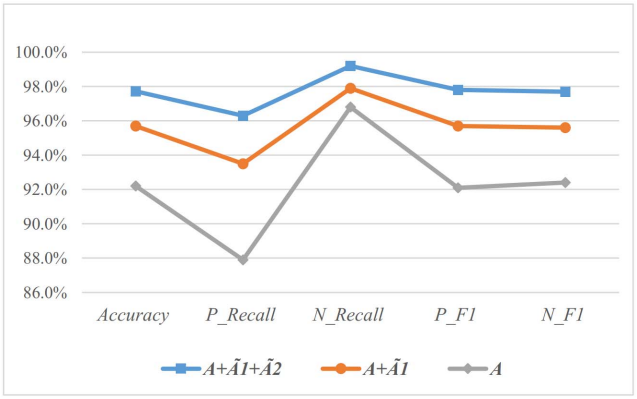


Figure 3. Comparative result of different sensitive APIs

2) Effectiveness of the feature set:

In this part, we explore the effectiveness of the feature set. Apart from the whole feature set $S = S^{\{T \to G\}} \cup S^{\{p\}}$ extracted in Section III-B, we extracted the requested permission *RP* in Androidmanifest.xml by *Android Asset Packaging Tool* (*AAPT*) and then elaborated a feature set that consists of the *RP* and the permission-protecting API *A* to do a comparative experiment. They are respectively denoted as *G* + *P*, *A* + *RP* in Table II. For a more clear result, we also illustrate them in Figure 4. Besides, in order to discuss the semantic and syntax feature separately, we did four more experiments with feature set *G*, *P*, *A*, *RP*. The result is shown in Table II.

TABLE II. COMPARATIVE RESULT OF DIFFERENT FEATURE SETS

Indicatior	G+P	G	P	A+RP	A	RP
Accuracy	97.7%	97.6%	94.9%	91.3%	90.9%	66.0%
P_Recall	96.3%	96.1%	94.0%	88.9%	91.7%	35.2%
N_Recall	99.2%	99.1%	95.9%	93.3%	89.7%	98.4%
P_Precision	99.3%	99.2%	96.1%	93.4%	92.9%	95.8%
N_Precision	96.2%	96.0%	93.8%	88.9%	88.1%	59.0%
P_F1	97.8%	97.6%	95.0%	91.1%	92.3%	51.5%
N_F1	97.7%	97.6%	94.8%	91.0%	88.9%	73.8%
FPR	0.8%	0.9%	4.1%	6.7%	10.3%	1.6%
FNR	3.7%	3.9%	6.0%	11.1%	8.3%	64.8%
AUC	0.99	0.99	0.98	0.96	0.96	0.72

Results. As Table II and Figure 4 show, when the feature set *S* is *G* + *P*, namely $S^{\{T \to G\}} \cup S^{\{p\}}$, the classification result is obviously better than the result of *A* + *RP*. Besides, the columns *G* and *P* in Table II demonstrate that both the semantic feature set $S^{\{T \to G\}}$ and the syntax feature set $S^{\{p\}}$ can contribute to the classification. Comparing the columns *G* with *A* and the columns *P* with *RP*, the results verify that for classifying Android applications, the semantic feature

$S^{\{T\to G\}}$ is better than the sensitive API and the requested permission is worse than the really-essential permission $S^{\{P\}}$. In a word, all the results in Table II confirm the rationality of our feature set.

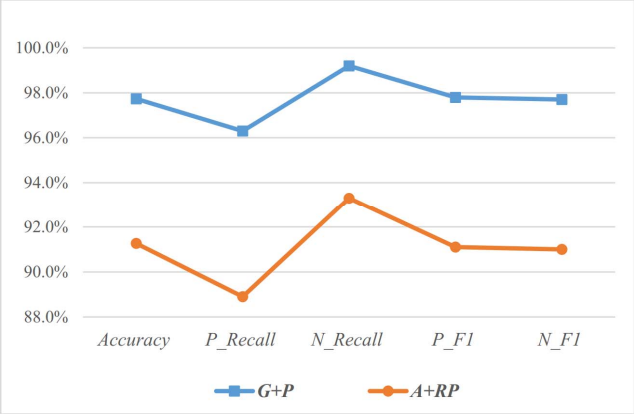


Figure 4. Comparative result of different feature sets

3) Effectiveness of the feature representation:

In this part, we explore the impact of different feature representations on experimental results. As described in Section III-B1, we collect plenty of sensitive APIs into the semantic feature, resulting in a big feature set. We would like to discuss if we reduce the quantity of the semantic feature, what will happen to the experimental result? Therefore, we make use of the coarse-grained category to represent the generalized¬ sensitive API. That is permission defined in Section III-B2, instead of the specific method signature. As Figure 5 shows, we use *S_Catego* to denote this form. In addition, we view that the coarse-grained generalized-sensitive API may result in loss of detailed information of features, so we also performed another experiment, in which the value of the semantic feature is represented by the frequency of its appearance in an application, with *S_CSum* to denote this form.

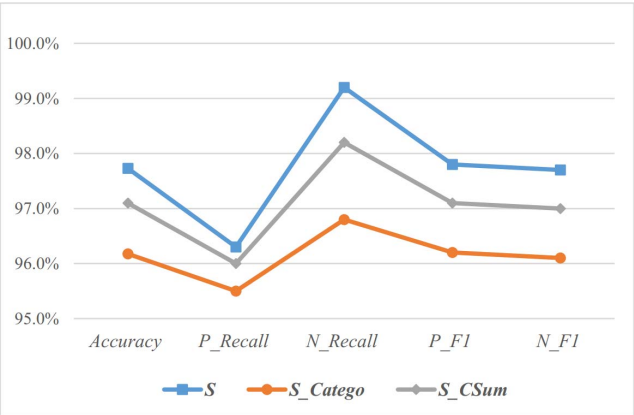


Figure 5. Comparative result of different feature representations

Results. Figure 5 illustrates that the coarse-grained representations of the generalized-sensitive API *S_Catego* and *S_CSum* are not as good as that in the original feature representation *S*, no matter the feature value is represented by binarity or frequency. As a result, the generalized-sensitive API in the feature set denoted as specific method signature is the best for the classification.

D. RQ3: Comparing with Malware Detecting Tools

1) Comparing with Drebin:

Drebin [11] is a lightweight Android malware detecting tool. It conducts static analysis to extract eight types of features of the application, mainly including permissions, components, APIs, etc. Then it trains SVM classification model to detect Android malware. To compare with Drebin, we analyzed the dataset of Drebin by our approach. In order to avoid the influence of different machine learning techniques on the classification result, we have leveraged RF and SVM to train two new classification models. Because Drebin only focuses on malware detection, we mainly compare RepassDroid with it according to the overall accuracy *Accuracy* and the malware detecting accuracy *P_Recall*, as shown in Figure 6.

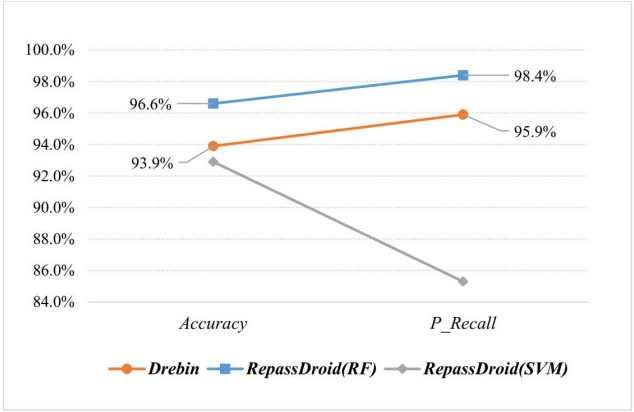


Figure 6. Comparative result with Drebin

Results. It can be seen from Figure 6 that RepassDroid trained with RF performs better than Drebin in malware detection. The result shows that the approach of extracting features in RepassDroid is effective.

2) Comparing with anti-virus tools on VirusTotal:

VirusTotal [22] is a free online malware detecting website, where a variety of anti-virus tools gathers. We use all the tools to detect the 4288 test applications. There are totally 57 tools while 52 tools perform worse than RepassDroid. We choose eight of them to show the comparative result in Table III.

TABLE III. COMPARATIVE RESULT WITH DIFFERENT DETECTING TOOLS

Detecting Tool	Accuracy	P_recall
RepassDroid	97.7%	96.3%
Sophos	97.2%	99.1%
BitDefender	96.7%	99.5%
Alibaba	87.6%	83.1%
AVG	87.4%	99.0%
Tencent	85.1%	84.1%
Microsoft	74.6%	99.7%
Baidu	63.2%	60.5%
Kingsoft	51.6%	53.5%

Results. Table III shows that there are three tools better than RepassDroid on malware detecting accuracy *P_recall*, but their overall detecting accuracy *Accuracy* is worse. Obviously, RepassDroid outperforms the remaining five tools. In summary, the performance of RepassDroid proves to be effective and encouraging.

V. Limitations and Future Work

Our analysis is based on FlowDroid and so inevitably inherits the shortcomings of static analysis, which cannot accurately handle the dynamic code. Thus the sensitive APIs we collected may be incomplete, which may cause false positives and false negatives. However, we have combined the results of PScout under multiple versions of Android, and the sensitive APIs we extracted are not limited to those permission-protected. In consequence, the generalized-sensitive API we defined is still relatively sound.

In future work, we may optimize our solutions by combining with dynamic analysis for detecting Android malware to improve the accuracy and precision. What's more, we will try to apply more classification techniques to find a better model.

VI. Related Work

The original research on Android malware detection is generally based on the permission requested in the manifest file [23–25]. However, applications often request redundant permissions and malware may even exploit the permissions of other applications to do corresponding malicious behaviors. Thus, the approaches just based on permissions may cause a large false positive rate.

As the research progressed, the researchers found the limitations of permissions, so they turned to combine APIs for the research on Android security. Cen et al. [26] found that the classification result of combining the API and the permission as features is better than that just using one of them. Aafer et al. [12] and Yerima et al. [27] used a similar approach, which chose feature sets based on the frequency of permissions and APIs. Arp et al. [11] extracted eight kinds of features and achieved a lightweight malware detecting tool called Drebin. None of the above studies put an eye to the behavioral semantics of applications.

In recent years, most researchers focus on the semantic features of applications. Both Apposcopy [7] and DroidSIFT [8] constructed relative graphs for applications to make signature matching with graphs in the database. So they cannot recognize novel malware. In the same way to choose the graph as the feature, Gascon et al. [28] constructed a feature vector and then detected malware by SVM. A large and growing body of literature has investigated Android applications by virtue of machine learning. Yang et al. [9] implemented AppContext to classify if the sensitive behavior is malicious or benign and have a good accuracy. However, the sample-constructing process is complicated and expensive so that it is poor for extendibility. Likewise, Miao et al. [10] used an analogical approach while their intention is to classify applications instead of sensitive behaviors. They also achieve a rather high accuracy, but their dataset is still small and the sensitive API is incomplete.

VII. Conclusion

We have presented a new feature set and implemented a tool for Android malware detection with low cost – RepassDroid, combining the semantic feature and the syntax feature and based on machine learning. The evaluation results indicate that Random Forest is the optimal classification technique for our feature set, achieving 97.7% accuracy and 0.99 AUC, with a malware detection precision as high as 99.3%. Besides, the results demonstrate that the abstraction scheme of the feature set is efficient.

Acknowledgment

We would like to thank AndroZoo, Malgenome and VirusShare for supporting us with the dataset. Thanks to X.C. Miao, Y. Aafer and L. Cen for providing helps. This work is mainly inspired by AppContext, Drebin and the work from X.C. Miao et al. This work is supported by the National Natural Science Foundation of China (NSFC) under grant 61772487 and the National Key R\&D Program of China 2017YFB1003000.

References

[1] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information flow tracking system for realtime privacy monitoring on smartphones. In Proceedings of the 9th USENIX Con ference on Operating Systems Design and Implementation, OSDI10, pages 1C6, Berkeley, CA, USA, 2010. USENIX Association.

[2] F. Wei, S. Roy and X. Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014:1329-1341.

[3] L. K. Yan and H. Yin. DroidScope: Seamlessly Recon structing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. USENIX security sympo sium. 2012: 569-584.

[4] Y. Zhou, Z. Wang, W. Zhou and X. Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. NDSS. 2012, 25(4): 50-52.

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. K lein, Y. Le Traon, D. Octeau, and P. McDaniel FlowDroid: Precise context, flow, field, object-sensitive and lifecycle aware taint analysis for Android apps. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 14, pages 259-269, New York, NY, USA, 2014. ACM.

[6] [6] L. Li, A. Bartel, T. F. Bissyand, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, 2015: 280-291.

[7] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: semantics-based detection of Android malware through static analysis. In Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014), 2014.

[8] M. Zhang, Y. Duan, H. Yin and Z. R. Zhao. Semantics Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014: 1105-1116.

[9] W. Yang, X. S. Xiao, B. Andow, S. H. Li, T. Xie and W. Enck. AppContext: Differentiating malicious and benign mobile app behaviors using context. Software engineering (ICSE), 2015 IEEE/ACM 37th IEEE international conference on. Vol. 1. IEEE, 2015.

[10] X. C. Miao, R. Wang, L. Xu, W. F. Zhang and B. W. Xu. Security Analysis for Android Applications Using Sensitive Path Identification. Journal of Software, 2017, 28(9) .

[11] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In 2014 Network and Distributed System Security Symposium, NDSS14, 2014.

[12] Y. Aafer, W. L. Du and H. Yin. DroidAPIMiner: Min ing API-Level Features for Robust Malware Detection in Android. International Conference on Security and Privacy in Communication Systems. Springer International Publishing, 2013:86-103.

[13] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: analyzing the Android permission specification. In Proceedings of the 19th Conference on Computer and Communications Security (CCS), pages 217-228, New York, NY, USA, 2012. ACM.

[14] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reute mann and L. H. Witten. The WEKA data mining software:an update. ACM SIGKDD explorations newsletter, 2009, 11(1): 10-18.

[15] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo and F. Massacci. StaDynA: Addressing the Problem of Dy namic Code Updates in the Security Analysis of Android Applications. Proceedings of the 5th ACM Conference on Data and Application Security and Privacy. ACM, 2015: 37-48.

[16] L. Li, D. Octeau, J. Klein. DroidRA: taming reflection to support whole-program analysis of Android apps. Pro ceedings of the 25th International Symposium on Software Testing and Analysis. ACM, 2016: 318-329.

[17] Z. Cheng, F. Zeng, X. Zhong, M. Zhou, C. Lv, S. Guo. Resolving reflection methods in Android applica tions. Intelligence and Security Informatics, 2017 IEEE International Conference on. IEEE, 2017: 143-145.

[18] S. Rasthofer, S. Arzt, and E. Bodden. A Machine learning Approach for Classifying and Categorizing An droid Sources and Sinks. In 2014 Network and Distribut ed System Security Symposium, NDSS14, 2014.

[19] AndroZoo. https://androzoo.uni.lu/.

[20] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In Proceedings of the IEEE Symposium on Security and Privacy (SP), pages 95-109, Washington, DC, USA, 2012. IEEE Computer Society.

[21] VirusShare. https://virusshare.com/.

[22] VirusTotal. https://www.virustotal.com/.

[23] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In Proceedings of the 16th ACM Conference on Computer and Commu nications Security, CCS 09, pages 235-245, New York, NY, USA, 2009. ACM.

[24] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R, Potharaju, C. N. Rotaru and I. Molloy. Using probabilistic generative models for ranking risks of android apps. Proceedings of the 2012 ACM conference on Computer and commu nications security. ACM, 2012:241-252.

[25] V. Moonsamy, J. Rong, S. Liu. Mining permission pat terns for contrasting clean and malicious android appli cations. Future Generation Computer Systems, 2014, 36: 122-132.

[26] L. Cen, C. S. Gates, L. Si and N. Li. A probabilistic discriminative model for android malware detection with decompiled source code. IEEE Transactions on Depend able and Secure Computing, 2015, 12(4): 400-412.

[27] S. Y. Yerima, S. Sezer, G. McWilliams and I. Muttik. A new android malware detection approach using Bayesian classification. Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Con ference on. IEEE, 2013: 121-128.

[28] H. Gascon, F. Yamaguchi, D. Arp and K. Rieck. Struc tural detection of android malware using embedded call graphs. ACM Workshop on Artificial Intelligence and Security. ACM, 2013:45-54.