Detection, Classification and Characterization of Android Malware Using API Data Dependency

Yongfeng Li(✉), Tong Shen, Xin Sun, Xuerui Pan, and Bing Mao

State Key Laboratory for Novel Software Technology,
Department of Computer Science and Technology, Nanjing University,
Nanjing, China
{jsliyongfeng,shentongnju,sunxin508,xueruipan}@gmail.com,
maobing@nju.edu.cn

Abstract. With the popularity of Android devices, more and more Android malware are manufactured every year. How to filter out malicious app is a serious problem for app markets. In this paper, we propose DroidADDMiner, an efficient and precise system to detect, classify and characterize Android malware. DroidADDMiner is a machine learning based system that extracts features based on data dependency between sensitive APIs. It extracts API data dependence paths embedded in app to construct feature vectors for machine learning. While DroidSIFT [13] also attempts automated detection of Android applications according to data flow analysis, DroidADDMiner can not only reduce the run time but also characterize malware's behaviors automatically. We implement DroidADDMiner based on FlowDroid [14] and evaluate it using 5648 malware samples and 14280 benign apps. Experiments show that, for malware detection, DroidADDMiner achieves a 98% detection rate, with a 0.3% false positive rate. For malware classification, the accuracy of classifying malicious apps under their proper family labels is 96%. Although performing data flow analysis, most of the experimental samples can be examined in 60 seconds.

Keywords: Android malware · Machine learning · Data flow · Flowdroid

1 Introduction

Smartphone is performing a more and more important role in people's daily life. According to a recent study [1], in United States and Great Britain, Android has reached over 50% market share. Meanwhile, in China, the market share has exceeded 70%. There's no doubt that Android has become the most popular platform for smart phone today. This trend has attracted attention of attackers, more and more malicious applications emerged in the official and alternative Android marketplaces. As described in [2], over 150,000 malicious applications and 253 new malware families have been discovered in 2013 alone. In order to

B. Thuraisingham et al. (Eds.): SecureComm 2015, LNICST 164, pp. 23–40, 2015.
DOI: 10.1007/978-3-319-28865-9_2

maintain a healthy ecosystem for Android, robust malware detection techniques need to be designed.

Previously, many machine learning based approaches have been proposed to detect malware. Before utilizing machine learning algorithms, they use feature vectors to model the app's behaviors. Their main difference lies in how to extract feature vectors. Rather than in-depth understanding program semantics, Drebin [10] and DroidAPIMiner [20] extract features from application syntax like permissions listed in manifest file and API parameters used by application code. Malware and benign apps may use the same APIs and permissions, because some benign apps also need to access sensitive resources. So these approaches are not robust enough to model malware's behaviors. DroidMiner [11] focuses on the control flow of Android application, API sequences extracted from control flow graph are used to construct feature vectors. But it may miss important data flow information that can help build better behavior models which have effects on the detection rate.

For Android application, APIs can be invoked under two contexts: user interface and background callback. Malware always exploit background callbacks to launch malicious behaviors. Constant values like network address can also reveal a malware's intention when they are used as parameters of some APIs. Hence, DroidSIFT [13] adopts data flow analysis to construct weighted contextual API dependency graphs which contain data dependency, context and constant information. Their feature vectors are extracted based on similarity between weighted contextual API dependency graphs. Although DroidSIFT represents program semantics well, it cannot automatically generate malicious behavior characterization of malware. Moreover, DroidSIFT is time-consuming when analyzing large-scaled apps because it calculates all objects' point-to information during data flow analysis.

We present DroidADDMiner to automate the process of Android malware detection, classification and characterization. DroidADDMiner is a machine learning based system which extracts features on the basis of API data dependency and also considers context and constant information just like DroidSIFT. We define API data dependence path with context and constant information as *modality*. A modality repository is built by collecting all modalities extracted from malware samples. Feature vector is then generated according to whether the app's modalities are contained in modality repository. Finally, feature vectors are feeded to machine learning techniques for detecting, classifying and characterizing malware.

Data flow analysis is the most important part of DroidADDMiner. Flowdroid [14][15] and Amandroid [17] are two state-of-the-art data flow analysis tools for Android. Like DroidSIFT [13], during data flow analysis, Amandriod calculates all objects' point-to information. Analyzing the same app, Flowdroid is quicker than Amandroid since it only focuses on objectes related to some specified sources and sinks. Using machine learning techniques needs to analyze abundant apps, so we choose to extend Flowdroid to build DroidADDMiner. We evaluate our system using 5648 malware samples and 14280 benign apps.

Experiments show that DroidADDMiner can achieve 98% accuracy in malware detection with 0.3% false positive rate, and it can label 96% malware instances to their right family. Although performing data flow analysis, for most of the experimental samples, DroidADDMiner can accomplish analysis in 60 seconds which leads us to believe that DroidADDMiner can handle large-scale applications.

To summarize, this paper makes the following contributions:

- We propose a semantic-based malware detection, classification and characterization approach. The program semantics of malware are modeled by API data dependence paths with context and constant information.
- We make an extension on Flowdroid [14]. Using the extended tool, we can perform API data dependence path construction, API context and constant analysis.
- We make an in-depth evaluation of DroidADDMiner. Experiments include run-time performance and efficacy in malware detection, family classification, and behavior characterization.

2 Motivation and System Goals

2.1 Motivation

We explain the motivation of our system design by introducing the inner working of a real-world malicious Android application. This malware sample (MD5: ecbbce17053d6eaf9bf9cb7c71d0af8d) belongs to the family of zitmo. The code of this malware is listed in Fig. 1. From the code snippet we can know that once a SMS is arrived, life cycle call `onReceive()` is invoked by Android system. Then `abortBroadcast()` is issued to abort current broadcast. In order to steal SMS message, an intent carries SMS message information is created to launch a background service (named "MainService"). Once the service is triggered, SMS message extracted from intent is stored in an object array named "pdus". Next, for extracting originating address (sender) and message body from this object array, `getOriginatingAddress()` and `getMessageBody()` are called. Now the address and message body are stored in String value "str1" and "str2" respectively. Meanwhile, after invoking `getDeviceId()`, the device id is stored in "str3". While malware gets all sensitive information it needs, these information are encoded into an `UrlEncodedFormEntity` object. Before sending these information through network, `HttpPost` object is created with a constant string "http://softthrifty.com/security.jsp" and then `setEntity()` is called to encode these information into a form that can be sent through network. Finally, `DefaultHttpClient.execute()` is issued to post data to remote server.

From the above description, we find an important design premise that when malware authors design malicious apps to achieve malicious behaviors, they always have to use some sensitive API calls like the APIs marked with red font in Fig. 1. DroidMiner [11] and DroidSIFT [13] is two state-of-the-art malware detection tools base on machine learning techniques. DroidMiner extracts API sequences according to control flow. For malware sample

```
1:public class SmsReceiver extends BroadcastReceiver{
2:  public void onReceive(Context pcontex,Intent pintent){
3:   Bundle localBundle = pintent.getExtras();
4:   if((localBundle != null)&&(localBundle. containsKey("pdus"))){
5:      abortBroadcast();
6:      Intent targetService = new Intent(pcontex, MainService.class);
7:      targetService.putExtra("pdus", localBundle);
8:      pcontex.startService(ts);
9:    }
10:  }
11:}
12:public class MainService extends Service{
13:  public int onStartCommand(Intent pintent,int pintent1,int pintent2){
14:    Bundle localBundle = pintent.getBundleExtra("pdus");
15:    Object[] pdus = (Object[])localBundle.get("pdus");
16:    ArrayList localAL = new ArrayList();
17:    SmsMessage localSMS = SmsMessage.createFromPdu(pdus);
18:    TelephonyManager localTM =  MainService.this.getSystemService("phone");
19:    String str1 = localSMS.getOriginatingAddress();
20:    String str2 = localSMS.getMessageBody();
21:    String str3 = localTM.getDeviceId();
22:    localAL.add(str1);localAL.add(str2);localAL.add(str3);
23:    postRequest(new UrlEncodedFormEntity(localAL));
24:  }
25:  public void postRequest(UrlEncodedFormEntity UEFE){
26:    String addr = "http://softthrifty.com/security.jsp";
27:    HttpPost localHP = new HttpPost(addr);
28:    localHP.setEntity(UEFE);
29:    BasicResponseHandler BRH = new BasicResponseHandler();
30:    DefaultHttpClient().execute(localHP, BRH));
31:  }
32:}
```

Fig. 1. Example Malware

depicted in Fig. 1, it will extract a control flow sequence [`BroadcastReceiver`, `abortBroadcast()`, `setEntity()`, `execute()`] and sensitive resources "Vres" {`getOriginatingAddress()`, `getMessageBody()`, `getDeviceId()`}. DroidMinerthey does not analyze the data flow of sensitive data, they simply consider that there is an edge from the root "Vroot" (one component, in our example is BroadcastReciever "SmsReceiver") to the resources "Vres". Actually, this is not precise. For example, in "SmsReceiver", the app invokes getOriginatingAddress(), getMessageBody() and getDeviceId() to obtain sensitive information, but if we change the sensitive information what we put into ArrayList "localAL" by Line 22, the malware's behaviors will be different. That's why analyzing data flow will get more precise behavior models which will affect the accuracy of identification process.

DroidSIFT [13] is another malware detection tool adopts machine learning techniques. During data flow analysis, it calculates all objects' point-to information, this is time consuming. Moreover, when DroidSIFT analyzes the demonstrated malware sample, it will construct a data dependence graph which is composed of red font marked API nodes list in Fig. 1. Because of utilizing the data dependence graph as an integrity to compute similarity related to base graphs in DroidSIFT's database, it loses the ability of digging out the relationships between APIs and malicious behaviors. So it cannot characterize a

malware's behaviors automatically. In order to address this deficiency, we extract API data dependence paths embedded in known malware samples. Then mine out the relationships between API data dependence paths and malicious behaviors according to the malicious behaviors malware contain. We use these relationships to characterize a unknown malware's behaviors.

2.2 Goals and Assumption

DroidADDMiner is aimed to detect whether an app is malicious, label malware to correct family, and more specially, give a concise description of a malware's malicious behaviors. For example, given the app demonstrated in section 2.1, DroidADDMiner can know it is a malware, classify it to zitmo family, and find out that it can get SMS message, block SMS message, and send sensitive information to remote server. DroidADDMiner is built based on Flowdroid [14], so its data flow analysis has the same limits as Flowdroid.

3 System Design

We demonstrate DroidADDMiner's work flow in Fig. 2. As depicted in this figure, DroidADDMiner contains two phases: program analysis phase and machine learning phase.

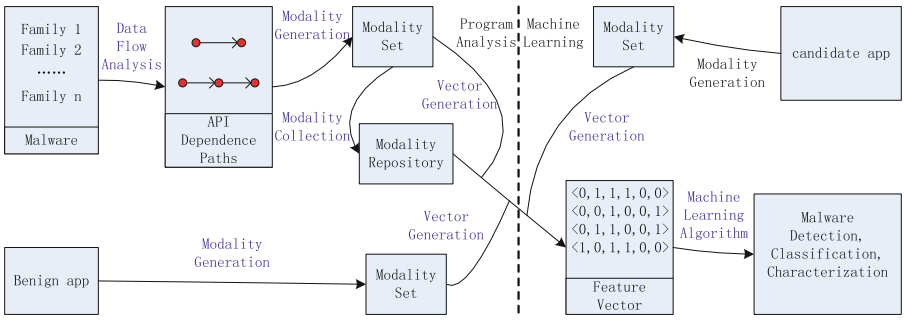

Fig. 2. System Architecture

The most important component of DroidADDMiner is program analysis. As described in section 2.1, we choose to use API data dependency, contex and constant information to represent the program semantics of malware. When performing data flow analysis, analyzing too much APIs will be very expensive. It is necessary to choose a set of APIs which can achieve computational efficiency and security analysis in the same time. So we leverage the API-permission mapping from Pscout [21] to conduct our data flow analysis.

We also need to know whether an API is activated from background callbacks, this is called context analysis. For context analysis, we select some background

callbacks like `BroadcastReceiver$onReceive` and `GpsStatus$Listener`. Constant information of parameters of some sensitive APIs, like `exec()` are significant signature to identify malware. These parameters can decide an app's behavior significantly. Due to space limitations, we don't list all these callbacks and APIs in this paper. To extract API data dependence path and extract context and constant information, we extend Flowdroid [14], a detail description will be given in section 4.1.

After analyzing an app, DroidADDMiner will obtain some API data dependence paths with context and constant information. We define API data dependence path with context and constant information as *modality*. In this paper, we use following formula to represent *modality*:

$$S_1[constant; context] \rightarrow \cdots \rightarrow S_k[constant; context] \rightarrow \cdots$$

In this formula, S_k represents sensitive API. `'constant'` represents the constant information of sensitive API, for APIs whose constant information we don't analyze, `'constant'` value will be `'none'`. On the other hand, if we analyze an API's constant information, the value will be `'ture'` or `'false'` depends on whether the API's parameter contains constant value. `'context'` represents the context information, if the API is invoked under a background callback, `'context'` value will be this callback, otherwise the value will be `'none'`. For example, for the malware shown in section 2.1, one of its modalities is:

$$setEntity()[false; onReceive] \rightarrow execute()[true; onReceive]$$

The modality is made up of at least one node, each node is a sensitive API with its context and constant information. We show how to extract modalities from app in section 4.2. After analyzing all malware samples, we collect all modalities DroidADDMiner obtains, then build a modality repository. For the sake of performing machine learning techniques, we need to generate feature vector for every app. Those feature vectors can be calculated based on modality repository. The detail of how to generate feature vectors is shown in section 4.3. At machine learning phase, we use the classical algorithm to detect whether an app is malicious. If it is a malware, we can label it to correct family. Finally we use "Association Rule Mining" technique to characterize a malware's behaviors. This will be described in section 4.4.

4 Implementation

4.1 Extension of FlowDroid

In this section, we introduce how we extend FlowDorid [14] to extract API data dependence path with constant and context information.

In order to extract API data dependence path, for adopting FolwDroid, we need to solve two problems: First, in FlowDroid, all data dependencies are starting from source and ending at sink, but we need to extract API data dependence

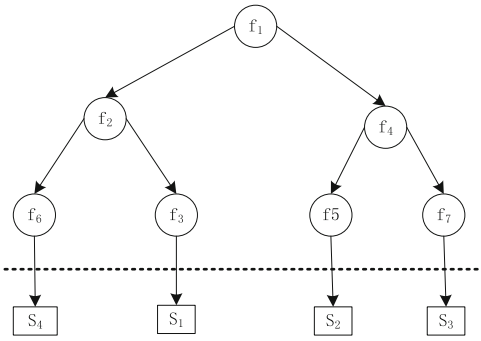

Fig. 3. An Example of Call Graph. Each circle vertex stands for a function, each rectangle vertex stands for a sensitive API

path, data dependency can start from or end at any sensitive API; Second, FlowDroid can just output data dependency between every two API(source and sink), but the API data dependence path may contains more than two nodes. To solve the first problem, we modify FlowDroid to make it treat the sensitive APIs we specified as both source and sink. The data flow analysis start from a sensitive API, during taint propagation, if a tainted factor encounters a sensitive API, we will record it and stop propagate this factor. Because for every sensitive API, we'll also treat it as source and start taint propagation from it. In this way, we can get propagation path between every two sensitive APIs which have data dependency relationship. For the second problem, as we get data flow propagation path between every two sensitive APIs. To construct long API data dependence path, we use these propagation paths. For simplicity and time efficiency, when using these propagation paths, we only focus on their call context(function call sequence), so we modify FlowDroid to output these call context.

When analyzing an app, FlowDroid constructs an extended call graph. Any control flow transformation like lifecircle or callback method is modeled in this graph and this call graph has only one entry point. It means if one sensitive API data depends on the other sensitive API, the call graph must contains a fuction can reach both these two sensitive APIs. For example, in fig 3, if S_2 data depends on S_1, f_1 is the function which is able to reach both S_1 and S_2. More generally, if an app has an API data dependence path, there must exist a function in the call graph which can reach all sensitive APIs this API data dependence path contains. The API data dependence path with call context is defined as:

$$F_t\{\cdots \rightarrow (\ldots, C_{k_m}, \ldots, C_{S_j})S_i \rightarrow \cdots\}$$

F_t represents the function this API data dependence path happens. C_{k_m} represents call statement, k_m is a function. Statement sequence in parentheses is the call context of propagation path, the last call statement must call sensitive API. S_i represents sensitive API, and API data dependence path contains at

least two nodes. Right arrow represents data dependence. The formula shows F_t can reach S_i through propagation path $(\ldots, C_{k_n}, \ldots, C_{S_j})$.

Short API Data Dependence Path with Call Context. To demonstrate the API data dependency path construction process, we assume that, for call graph in fig 3, we get following short paths

$$f_1\{(c[f_2], c[f_3], c[S_1])S_1 \rightarrow (c[f_4], c[f_5], c[S_2])S_2\} \quad (1)$$

$$f_4\{(c[f_5], c[S_2])S_2 \rightarrow (c[f_7], c[S_3])S_3\} \quad (2)$$

c[F] denotes a call statement which invokes function or API.

Long API Data Dependence Path Construction. Before constructing long paths, we need to define what kind of paths can be assembled. Every path has at least two nodes, we call the first node start node and the last node end node. If two path can be assembled to construct a long path, this means the first path's end node is "equal" to the second path's start node. In this case, two nodes are "equal" does not mean they are identical. Every node in the API data dependence path has call context. During our path construction process, end node is "equal" to start node means their call context are identical or one's call context is the subsequence of the other one's. For example, $(c[f_5], c[S_2])S_2$ is subsequence of $(c[f_4], c[f_5], c[S_2])S_2$, so path (1) and path (2) can be assembled to a long path:

$$f_1\{(c[f_2], c[f_3], c[S_1])S_1 \rightarrow (c[f_4], c[f_5], c[S_2])S_2 \rightarrow (c[f_4], c[f_7], c[S_3])S_3\} \quad (3)$$

Context and Constant Analysis. After data flow analysis, we get all API data dependence paths embedded in an app. In this section, we demonstrate how to add context and constant information to API nodes in these API data dependence paths.

For constant analysis, APIs (such as `Runtime.exec()`) whose parameter have special meaning are selected. To perform constant analysis, starting from statements invoke these APIs, we backward search the control flow graph. Call context will be stored during this process. Hence, we can obtain sensitive APIs' constant information with call context. Just like path construction, using the call context, we can add constant information to nodes in API data dependence path. For example, if we get following constant information:

$$f_2\{(c[f_3], c[S_1])S_1[true; none]\}$$

$f_2\{(c[f_3], c[S_1])S_1$ is the subpath of $f_1\{(c[f_2], c[f_3], c[S_1])S_1$, so we can add this information to path (3), we'll get a new path:

$$\begin{aligned} f_1\{(c[f_2], c[f_3], c[S_1])S_1[true; none] &\rightarrow (c[f_4], c[f_5], c[S_2])S_2 \\ &\rightarrow (c[f_4], c[f_7], c[S_3])S_3\} \end{aligned} \quad (4)$$

For context analysis, we need to know whether a function is triggered in background. Among the code of an app, background callback is overridden to do some operations. In Flowdroid [14], all callback methods are modeled in a dummy method. This means if we perform a backward search on control flow graph, we can reach a single entry point. We know that every API data dependence path is happened in a function. For example, the path (4) is contained in function f_1. Starting from nodes in control flow graph which invoke f_1, we perform backward search. If we encounter a background callback method, record it. After the backward search, we can decide the context of f_1 based on the callback methods we record. But we can't directly apply the f_1's context to all nodes in path (4), because the nodes in this path also have call context. If a node's call context contains a background callback method, we specify this background callback method as the node's context. Otherwise, the node's context is decided by f_1's context. Using this approach, we can obtain context of all nodes in API data dependence path. For example, if our backward analysis find that f_1 is invoked under `onRecieve`, and $f_1, f_2, f_3, f_4, f_5, f_7$ are not background callback, we can get a new path with constant and context information:

$$f_1\{(c[f_2], c[f_3], c[S_1])S_1[true; onReceive] \rightarrow (c[f_4], c[f_5], c[S_2])S_2[none; onReceive] \rightarrow (c[f_4], c[f_7], c[S_3])S_3[none; onReceive]\} \quad (5)$$

Finally, we remove the call context information, and can get a API dependence path:

$$S_1[true; onReceive] \rightarrow S_2[none; onReceive] \rightarrow S_3[none; onReceive]\} \quad (6)$$

4.2 Modality Generation

In section 3, we define *modality*. And in section 4.1, we demonstrate how to extract API data dependence path with constant and context information. For an API dependence path, we extract its subpaths, because these subpaths are both modalities. The length of these subpaths are not less than one. For path (6) obtained from section 4.1, we can get following subpaths:

$$S_1[true; onReceive] \quad (7)$$

$$S_2[none; onReceive] \quad (8)$$

$$S_3[none; onReceive] \quad (9)$$

$$S_1[true; onReceive] \rightarrow S_2[none; onReceive] \quad (10)$$

$$S_2[none; onReceive] \rightarrow S_3[none; onReceive] \quad (11)$$

So, for Fig. 3, path (6)(7)(8)(9)(10)(11) are modalities. Through the approach, we collect modalities embeded in all malware samples to build a modality repository.

4.3 Feature Vector Construction

Before applying machine learning techniques, we need translating extracted information to mathematical form. For every app, we generate a feature vector. All app's feature vectors will be added to a data set used by machine learning algorithms. In section 4.2, we get a modality repository. For an app, we can extract the modalities embeded in it. The app's feature vector is constructed as a boolean vector $(B_1, B_2, ..., B_n)$: $B_i = 1$, if app??s modality set contains modality M_i in the modality repository. Otherwise, $B_i = 0$. Through this vector, all API data dependencies can be represented.

4.4 Malware Detection, Classification and Characterization

In this section, we introduce how to use app's feature vectors to achieve malware detection, classification and characterization:

Malware Detection. One application scene is to determine whether or not an Android app is malicious. This is not straightforward. Some benign apps also use sensitive APIs to accomplish some actions like sending SMS message and getting location information. So their feature vectors may contain some modalities mined from malware. However, usually, malicious behaviors are not launched by just a single modality. Multiple modalities are needed to achieve a malicious behavior. This observation makes us treat an app as malware only when its modalities exceed a threshold. In this paper, we use machine learning technique to automatically find the relationships between modalities and malware. Machine-learning classifier mines the relationships based on feature vectors extracted from known malware samples and benign apps, then unknown apps can be detected by this classifier.

Malware Classification. Another application scene is to label malware to a malware family which it actually belongs to. Generally, malware belong to the same family always share similar malicious behaviors. This leads to their modalities are similar. For us, we can use the similarity between malware's feature vectors to classify malware. Using the malware samples from known malware family, we can build a machine-learning classifier to classify unknown malware samples.

Malware Characterization. The last application scene is to automatically characterize the malicious behaviors a malware contains. In fact, to achieve a specified malicious behavior, malware always needs to invoke some sensitive APIs. Such as sending a SMS message needs `sendTextMessage()`, getting location information needs `getLastKnownLocation()`. It means there exist relationships between modalities and malicious behaviors. Our work is to dig out which modalities result a specific malicious behavior. This goal can be achieved by using

a well-known machine learning technique called "Association Rule Mining". Malware from the same malware family share the similar malicious behavior, we can list malicious behaviors of a malware family from many sources [3][9]. Malware from a malware family may contains several malicious behaviors, like blocking SMS message, sending out phone id. We can use a boolean vector to represent a malware's behaviors according to whether it contains a specified malicious behavior. Then adding this vector to the end of malware's feature vector to construct a new vector. Feeding this vector to "Association Rule Mining" algorithm can mine out the relationships between modalities and malicious behaviors.

5 Evaluation

5.1 Dataset and Experiment Setup

We collect 6400 malware samples from the Android Malware Genome Project (AMGP) [9][22] and VirusShare project [5]. Then we submit these malware samples to VirusTotal [4]. For each malware, we get a VirusTotal report which lists the scan results of 57 different antivirus (AV) products. If a malware is labeled as malicious by more than 4 AVs, we add this malware to our malware dataset. Finally, we get a malware dataset contains 5648 malware samples. For malware classification, we need to know which malware family a malware belongs to. After we examine the scan results of AV products, we find Ad-Aware's [6] classification results are more approximate to the classification results of AMGP. So we chose the classification results of Ad-Aware to classify the malware. In order to construct a benign dataset, we crawls apps from two alternative Android markets(xiaomi [7] and anzhi [8]). We also upload crawled apps to VirusTotal. If an app passed all AVs, we add it to our benign dataset. In the end, we get 14280 benign apps. Finally, our dataset contains 5648 malware samples and 14280 benign apps.

We conduct experiments on a computer equipped with Intel(R) Core(TM) i7-4770k CPU(3.5GHz) and 16GB of physical memory. The operation system is Windows 7 and we utilize weka [25] as machine learning tool.

5.2 Summary of Modality Generation

The summary of Modality Generation is shown in Figure 4 and Figure 5. Among them, Figure 4 demonstrates the number of the modalities generated from both benign and malicious apps. As shown in this figure, for 94.3% of benign apps and 90.4% of malware samples, less than 20 modalities are extracted from an individual app. This is because the majority of apps don't invoke too many different sensitive APIs.

After analyzing 5648 malware samples, we obtain 4317 modalities. The length of modality is defined as the number of sensitive APIs it contains. Figure 5 illustrates the distribution of the length of modality. As shown in this figure, the longest modality is 7 and 87% of modalities carry less than 6 APIs.

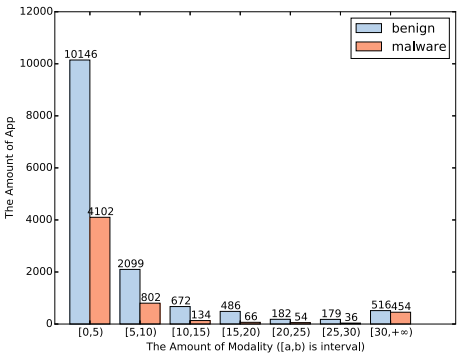

Fig. 4. Distribution of The Amount of Modality Extracted from Each App

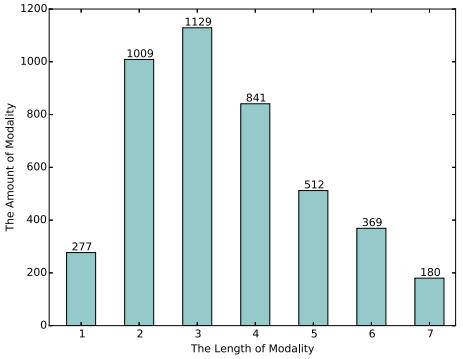

Fig. 5. Distribution of The Length Modality

5.3 Malware Detection Result

As introduced in section 4.4, we use machine learning techniques to detect malware. In our experiment, we adopt NaiveBayes, SVM and Random Forest to conduct malware detection, and we use 10-fold cross validation to evaluate these machine learning approaches. The malicious apps and benign apps are both randomly split into 10 groups. In each time of 10 rounds, we select combination of one group of benign apps and malicious apps as testing dataset. The reminding groups are treated as training dataset. When using NaiveBayes, we can correctly identify 91.5% of experimental apps with a 0.8% false positive rate. This process can be completed in 30 seconds. For SVM algorithm, there are four kinds of kernel function in weka [25]: linear, polynomial, radial basis function and sigmid. After testing all these kernel functions, we find linear kernel can achieve 97.3% accuracy rate with a 1.6% false positive, the training and testing procedure can be finished in 3 minutes. We also evaluate the efficiency of using Random Forest, the experiment completes in 20 minutes and 98.5% of apps are correctly identified with a 0.3% false positive rate. For DroidMiner, it achieves 82.2%, 86.7% and 95.3% accuracy rate when using NaiveBayes, SVM and Random Forest respectively. This verify feature vectors extracted based on data flow is more efficient than control flow on modeling the program semantics of malware. The comparison is shown in Table 1.

5.4 Malware Classification Result

In this section, we evaluate the ability of DroidADDMiner [20] to label malware to its correct family. We select 1168 malware samples from 16 malware families. The number of samples selected from each family is listed in Table 2. For malware of each family, we divide them into training set and testing set. Training set contains 66% of malware samples and testing set contains 34% of

Table 1. Effectiveness of Malware Detection(DR denotes Detection Rate, FP denotes False Positive Rate)

Classifer	**NaiveBayes**		**SVM**		**Random Forest**	
Tool	DR	FR	DR	FR	DR	FR
DroidADDMiner	91.5%	0.8%	97.3%	1.6%	98.5%	0.4%
DroidMiner	82.2%	4.4%	86.7%	1.1%	95.3%	0.3%

malware samples. Then we use Random Forest as classifier for training and prediction. The experiments show the classifier can correctly label 96% of malware samples. We further examine 4% of the samples that are mislabeled. 7 samples from DroidDeluxe and GingerMaster are labeled as one another, DroidDeluxe and GingerMaster both root the phone and share some similar malicious behaviors, thus these mislabels are understandable. DroidKungFu4 is the variant of DroidKungFu2, so 4 samples belong to them are mislabeled as one another.

Table 2. Malware Samples Used for Classification

Ind	**Family**	**Num**	**Ind**	**Family**	**Num**
1	GingerMaster	42	9	DroidKungFu2	26
2	DroidDeluxe	22	10	DroidKungFu3	305
3	ADRD	27	11	DroidKungFu4	71
4	BaseBridge	114	12	Geinimi	67
5	AnserverBot	183	13	GoldDream	42
6	DroidDreamLight	46	14	KMin	71
7	DroidDream	21	15	Pjapps	44
8	DroidKungFu1	28	16	SmsSpy	59

5.5 Malware Characterization Result

As described in section 4.4, in order to characterize a malware's behaviors, we need to construct a boolean vector for each malware family to model its malicious behaviors. We use the malicious behavior characterization of malware family collected by DroidMiner [11], and also focus on following behaviors: stealing phone information (GetPho), Sending SMS (SdSMS), blocking SMS (BkSMS), communicating with a C&C (C&C), escalating root privilege (Root) and accessing geographical information (GetGeo). Then malicious behavior boolean vectors are generated for each malware family. Adding corresponding malicious behavior boolean vector to the end of a malware's feature vector, we can get new vector for Association Rule Mining.

We utilize Apriori algorithm [12] to mine the relationships between malicious behaviors and modalities. After mining, DroidADDMiner obtained 492 behavior

Table 3. Behaviors of 5 Test Malware Samples

MD5	Family	Behavior
3ae5c5ee6c118a3cdbf2c55132f55948	SmsSpy	BkSMS,C&C,SdSMS
156fdce65eb6e4287aed687a1c9c2589	GGTracker	BkSMS,C&C,GetPho,SdSMS
60ce9b29a6b9c7ee22604ed5e08e8d8a	Endofday	BkSMS,GetPho,SdSMS
e98791dffcc0a8579ae875149e3c8e5e	zitmo	BkSMS,SdSMS
de04914d84239fbd40aa470ad86e388c	DroidKungFuUpdate	Root,GetPho,C&C

Table 4. Representative Rules for Malicious Behavior Characterization

Index	Behavior	Rule
1	GetGeo	LocationManager.getBestProvider()[false;none] → Location.getLastKnownLocation()[false;none]
2	GetGeo	LocationManager.requestLocationUpdates()[true;none]
3	Root	Runtime.getRuntime()[false;none] → Runtime.exec()[true;none]
4	Root	Process.killProcess()[false;none]
5	C&C	ConnectivityManager.getActiveNetworkInfo()[false;none] → WifiManager.setWifiEnabled()[false;none]
6	C&C	URLConnection.openConnection()[false;none] → HttpURLConnection.connect()[false;none]
7	SdSMS	gsm.SmsManager.getDefault()[false;none] → gsm.SmsManager.sendTextMessage()[true;none]
8	SdSMS	SmsManager.getDefault()[false;none] → SmsManager.sendTextMessage()[true;none]
9	GetPho	TelephonyManager.getLine1Number()[false;none] → ConnectivityManager.getActiveNetworkInfo()[false;none]
10	GetPho	TelephonyManager.getDeviceId()[false;none] → HttpEntityEnclosingRequestBase.setEntity()[false;none]
11	BkSMS	ContentResolver.delete()[false;BroadcastReceiver$onReceive]
12	BkSMS	abortBroadcast()[false;BroadcastReceiver$onReceive]

association rules. Some representative rules are listed in Table 3. Then we use these mined rules to test malware samples which are not used in mining phase. The results show we can correctly characterize the malware's behaviors. We list the results in Table 4.

5.6 Runtime Performance

DroidADDMiner needs three steps to identify an app: modality generation, feature vector construction and machine learning. Compared with modality generation, the time of feature vector construction and machine learning are negligible. So we just focus on the time of modality generation. Fig. 6 illustrates the runtime performance of modality generation for benign apps and malware samples. As shown in this figure, because most of malware samples are very small, majority (91%) of malware samples are completed in 10 seconds. For 89% of benign apps and 95% malware samples, the process of modality generation can be completed in 1 minute. The average runtime of modality generation is 10 seconds. DroidSIFT [13] also performs data flow analysis on Android app, as shown in Table 5, although their hardware is better than ours, its average runtime is 175.8 seconds. It's no doubt that when analyzing large-scaled apps, DroidADDMiner can vastly reduce the running time.

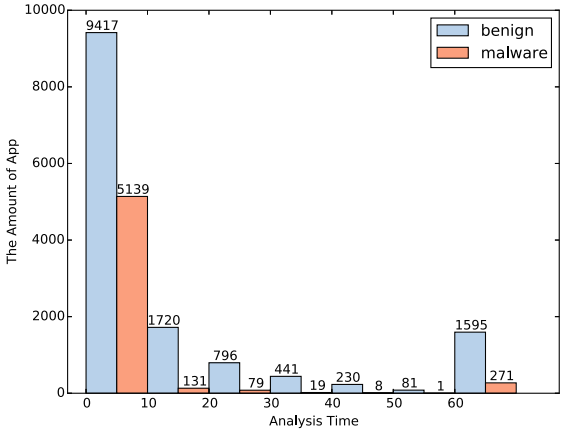

Fig. 6. Distribution of Modality Generation Time

Table 5. Runtime Performance of Malware Detection Tools

Tool	Average Performance	CPU	physical memory
DroidADDMiner	10s	Core(TM) i7-4770k	16G
DroidSIFT	175s	Xeon(R) E5-2650	128GB

6 Discussion

There is competition between defender and attacker, Android malware always evolutes itself to evade detection. DroidChameleon [23] and Adam [24] have demonstrated common malware transformation techniques like repackaging, changing field names could evade many existing commercial anti-malware tools. But for DroidADDMiner, it does not rely on external symptoms like package name, field name. So it's resilient to these common transformation attacks. Other transformation techniques like call indirections, code reordering and junk code insertion also can not evade DroidADDMiner. Because DroidADDMiner focuses on data flow between sensitive APIs, these transformation techniques do not change the data flow of sensitive APIs. To demonstrate it, we use DroidChameleon and Adam to obfuscate 100 malware samples selected from DroidKungFu3 family. As expected, DroidADDMiner can label all these obfuscated samples to DroidKungFu3 family. But DroidADDminer also has some limitations. It does not take native code into consideration right now, so a malware can put malicious behaviors in native code to bypass detection of DroidADDMiner. And DroidADDMiner just performs a simple constant analysis, if malware author splits an string like "content://sms" into two parts, we can not get the original semantics of some APIs. These limitations are left for future work.

Table 6. Comparison of Different Tools

Tool	Modeling of App Behavior	Explanation of App Behavior
Dredin	permission, API, manifest file	support
DroidAPIMiner	API, parameters of API	-
DroidMiner	control flow of API	support
DroidSIFT	data flow of API context and constant information	-
DroidADDMiner	data flow of API context and constant information	support

7 Related Work

Static analysis techniques are widely adopted to extract features for using machine learning algorithm to detect and classify Android malware. We summarize the difference of exist tools in Table 6. We don't list the detection rate and time efficiency in this table, because these tools use different machine learning algorithms and hardwares. Drebin [10] proposes to detect Android malware by extracting feature vectors from application manifest file and app code. DroidAPIMiner [20] extracts features at API level, and they take some APIs' parameters into consideration. Despite the effectiveness, the extracted feature vectors of these approaches are related to application syntax instead of program semantics. The feature vectors they extract are not robust enough to reflect app's behaviors. DroidMiner [11] focuses on control flow, they select some sensitive APIs and specific resources as the nodes to construct control flow graph, node sequences are extracted from this graph to generate feature vectors. Missing of data flow information could affect its detection rate. DroidSIFT [13] performs data flow analysis on Android apps. For every app, it generats a weighted contextual API data dependence graph. Then similarities between graphs are calculated to construct feature vectors. Compared with DroidADDMiner, it not only lacks of the ability to automatically characterize the behaviors of malware but also needs more time to analyze an app.

CHEX [16], Flowdroid [14], AmanDroid [17] are three tools designed to deal with information leakage problem. CHEX [16] uses a ??spit?? based approach to perform data flow analysis, each program split includes code reachable from a single entry point. For every program split, a system dependence graph [18] will be generated. Sources and sinks connections are extracted from this graph. Amandroid [17] computes an inter-component data flow graph (IDFG) which contains all objects?? points-to information in a both flow and context-sensitive way. This IDFG can be used to solve security problems including information leakage problem. Flowdroid [14] is quite different from CHEX [16] and Amandroid [17], it models data flow analysis problem within the IFDS [19] framework for inter-procedural distributive subset problems. Flowdroid is faster than the other two tools, because when performing data flow analysis, it only focuses on the variables related to sources and sinks. DroidADDMiner is built based on Flowdroid, so it can benefit from Flowdroid.

8 Conclusion

In this paper, we propose a semantic-based approach which detects, classifies and characterizes Android malware via API data dependency. For each app, we extract API data dependence paths which we call modality embedded in the app. Feature vectors are constructed for every app according to these modalities. We present our prototype system, DroidADDMiner, extends FlowDroid [13]. We evaluate our system using 5648 malware samples and 14280 benign samples. Experiments show that DroidMiner can achieve 98% accuracy in malware detection, and it can label 96% malware instances to its right family. Although performing data flow analysis, for most of the experimental samples, DroidADDMiner can complete analysis in 60 seconds.

Acknowledgements. We would like to thank anonymous reviewers for their comments. This work was supported in part by grants from the Chinese National Natural Science Foundation (61272078, 61073027, 90818022, and 61321491), and the Chinese National 863 High-Tech Program (2011AA01A202).

References

1. iPhone market share shrinks as Android, Windows Phone grow. http://www.cnet.com/news/iphone-market-share-shrinks-as-android-windows-phone-grow/
2. Mobile threat report 2013 q3. F-Secure Response Labs (2013). https://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q3_2013.pdf
3. Symantec enterprise. http://www.symantec.com/security_response/landing/azlisting.jsp
4. Virustotal. https://www.virustotal.com/
5. Virusshare. http://virusshare.com/
6. Ad-Aware. http://www.lavasoft.com/
7. Xiaomi android market. http://app.mi.com/
8. Anzhi Android market. http://www.anzhi.com/
9. Android malware genome project. http://www.malgenomeproject.org/
10. Arp, D., Spreitzenbarth, M., Hbner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: effective and explainable detection of android malware in your pocket. In: Proceedings of NDSS, February 2014
11. Yang, C., Xu, Z., Gu, G., Yegneswaran, V., Porras, P.: DroidMiner: automated mining and characterization of fine-grained malicious behaviors in android applications. In: Kutyłowski, M., Vaidya, J. (eds.) ICAIS 2014, Part I. LNCS, vol. 8712, pp. 163–182. Springer, Heidelberg (2014)
12. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of the 20th International Conference on Very Large Data Bases, pp. 641–644. Morgan Kaufmann Publishers Inc. (1994)
13. Zhang, M., Duan, Y., Yin, H., Zhao, Z.: Semantics-aware android malware classification using weighted contextual API dependency graphs. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1105–1116. ACM, November 2014

14. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: 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, pp. 29. ACM, June 2014
15. Fritz, C., Arzt, S., Rasthofer, S., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Highly precise taint analysis for Android applications. EC SPRIDE, TU Darmstadt, Tech. Rep. (2013)
16. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 229–240. ACM, October 2012
17. Wei, F., Roy, S., Ou, X.: 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, pp. 1329–1341. ACM, November 2014
18. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems (TOPLAS) **12**(1), 26–60 (1990)
19. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 49–61. ACM, January 1995
20. Aafer, Y., Du, W., Yin, H.: DroidAPIMiner: mining API-level features for robust malware detection in android. In: Zia, T., Zomaya, A., Varadharajan, V., Mao, M. (eds.) SecureComm 2013. LNICST, vol. 127, pp. 86–103. Springer, Heidelberg (2013)
21. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: Pscout: analyzing the android permission specification. In: Proceedings of the 2012 ACM conference on Computer and communications security, pp. 217–228. ACM, October 2012
22. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 95–109. IEEE, May 2012
23. Rastogi, V., Chen, Y., Jiang, X.: Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, pp. 329–334. ACM, May 2013
24. Zheng, M., Lee, P.P.C., Lui, J.C.S.: ADAM: an automatic and extensible platform to stress test android anti-virus systems. In: Flegel, U., Markatos, E., Robertson, W. (eds.) DIMVA 2012. LNCS, vol. 7591, pp. 82–101. Springer, Heidelberg (2013)
25. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. ACM SIGKDD Explorations Newsletter **11**(1), 10–18 (2009)