Exploring Sensor Usage Behaviors of Android Applications Based on Data Flow Analysis

Xing Liu, Jiqiang Liu, Wei Wang*
School of Computer and Information Technology, Beijing Jiaotong University
Beijing, China, 100044
{xingliu, jqliu, wangwei1}@bjtu.edu.cn

***Abstract*—Today's Android-powered smartphones are equipped with various embedded sensors, such as the motion sensors, the environmental sensors and the position sensors. Many functions in the third-party applications (apps) need to use these sensors. However, embedded sensors may lead to security issues, as the third-party apps can access data from these sensors without claiming any permissions. It has been proven that embedded sensors can be exploited by well designed malicious apps, resulting in leaking users' privacy. In this work, we are motivated to provide an up-to-date overview of sensor usage patterns in current apps by investigating what, why and how embedded sensors are used in all the apps collected from a complete app market. To fulfill this goal, We develop a tool called "SDFDroid" to identify the sensors' types and to generate the sensor data propagation paths in each app. We then cluster the apps to find out their sensor usage patterns based on their sensor data propagation paths. We apply our method on *AppChina*, a widely used Chinese Android app market. Extensive experiments are conducted and the experimental results show that most apps implement their sensor related functions by using the third-party libraries. We further study the sensor usage in the third-party libraries. Our results show the accelerometer sensor is the most frequently used sensor. Though many third-party libraries use no more than four types of sensors, there are some third-party libraries register all the types of sensors recklessly. These results show the need for better regulating the sensor usage in Android apps.**

***Index Terms*—Android system; Sensor usage; Data-flow analysis; Clustering**

I. Introduction

In recent years, Android-powered smartphones have become very popular for both personal and business use. According to a report from the International Data Corporation (IDC) [1], Android-powered smartphones dominate the market with a 78.0% share in the first quarter of 2015. At the same time, the smartphone's hardware is more and more advanced. Various sensors have been embedded in smartphones, such as the motion sensor (*e.g.,* accelerometers, gyroscopes), the environmental sensor (*e.g.* temperature, illumination) and the position sensor (*e.g.,* orientation sensors and magnetometers) [2]. These sensors are used by the third-party applications (apps) to support their novel features, such as the Spirit Level in some Camera-related apps.

When Google designed Android, it did not consider the security issues that may be led by the embedded sensors. Therefore, the third-party apps can use embedded sensors without claiming any permissions on Android platform. However, embedded sensors can be exploited by well designed malicious apps (malapps), resulting in leaking users' privacy. For example, tapping different positions on the touchscreen will cause different motion changes of the smartphone. If the correlations between the tap events and the data collected by the motion sensors are learned, one can successfully guess users' input through the data collected by the motion sensors. Related work [3][4][5] has already proven that smartphone's embedded sensors can expose users' privacy data. In particular, Xu *et al.* [6] designed and implemented "TapLogger", a trojan app that uses obtained sensor data to log user inputs on touchscreen stealthily. This app only needs two permissions, INTERNET and READ_PHONE_ STATE, and disguise itself as a game. Due to its high threat to users' confidential information, this work was reported by various media [7][8]. Besides, other researchers [9][10] proposed some user identity recognition mechanisms based on embedded sensor data. These work indicates that embedded sensors' data can not only be used to identify whether a user is the smartphone's owner, but can also leak users' identity information.

Although these studies still remain in the experimental stage, they attract our attentions to the sensor usage patterns in the current Android apps. According to a statistical result from AppBrain [11], the number of apps in Google Play has reached 1.5 million, while according to the Android API Guides [2], Android system supports more than ten types of sensors. Since embedded sensors can expose confidential information on smartphones, curiosities are aroused on understanding what, why and how the embedded sensors are used in apps collected from a whole app market. As a first step in the direction of answering these questions, we explore sensor usage behaviors of Android apps based on data flow analysis. To fulfill this goal, we design and implement a tool called SDFDroid (Sensor data flow droid). SDFDroid first disassembles the apk files to smali code files, and then performs two kinds of data flow analysis. One is backward tracking analysis, which starts from the API that registers a sensor listener in the system to find the used sensors' type. The other is forward tracking analysis, which starts from the API that read sensor data from the system to generate sensor data propagation path graph. We calculate the similarity between the sensor data propagation path graphs which are generated from many apps, and then cluster the graphs with DBSCAN, a well known clustering

* Corresponding author

978-1-4673-8590-9/15/$31.00 ©2015 IEEE

algorithm. Sensor usage patterns are thus constructed from the cluster results. We analyze all the apps in *AppChina*, one of the main Android app markets in China, and find that almost all the apps implement sensor related functions with the help of the third-party libraries. We also find that accelerometer sensor is the most-used sensor and that the used sensor type has a clear association with the app's function. We make the following contributions in this paper:

- We design and implement SDFDroid that fast and accurately analyzes Android apps' sensor data propagation pathes.
- We employ SDFDroid to analyze all the apps in *AppChina* and reveal sensor usage patterns in apps on large-scale.
- To the best of our knowledge, our work is the first research on the sensor usage analysis on the view of an Android whole app market.

The remainder of this paper is organized as follows. Section 2 describes the background of Android sensor system and smali code files. Section 3 gives our system design. Section 4 presents experimental results. Limitation and related work are given in Section 5 and 6, respectively. Section 7 concludes our work.

II. Background

A. Android Sensor System

Today's Android-powered smartphones are equipped with various embedded sensors. These sensors are used to monitor the device movement, position or other surrounding environmental conditions. Android system supports many types of sensors (See Table I). Some sensors are hardware types, which means that the sensor data are directly read from physical components built in the smartphone. Other sensors are software types, which means that sensor data are read from one or more of the hardware sensors [2]. The embedded sensors are widely used in the third-party apps. For example, a navigation app may use the magnetic field sensor to determine the compass bearing.

The embedded sensors are managed by Android Sensor Framework. Different from the Camera, GPS and Bluetooth which are protected by Android permission mechanism, the embedded sensors can be directly used by the third-party apps without any requirement of permissions. With the help of Android Sensor Framework, a third-party app can read a sensor's data by the following steps (as shown in Figure 1).

- First, instantiate the object of *SensorManager* class (Line 12). In this step, an app creates an instance of the sensor service. This class provides many methods for interaction with sensors, such as access specified sensors and list all available sensors, register and unregister sensor event listeners.
- Second, instantiate the object of *Sensor* class by calling the *getDefaultSensor()* (Line 14). In this step, an app gets an object of *Sensor* with specific type. The type of sensors can be specified by the method's parameter. As shown in this example, *Sensor.TYPE_ACCELEROMETER* is a constant describing an accelerometer sensor type. If light sensor is required, one can set the method's parameter as *Sensor.TYPE_LIGHT*. All available constants can be found in the Android developer website [12].
- Third, instantiate the object of *SensorEventListener* interface (Line 15) and override two callback methods which are *onAccuracyChanged()* (Line 17) and *onSensorChanged()* (Line 20). These two methods are used to receive sensor events when sensor accuracy changes or when sensor values change. Android system will invoke *onSensorChanged()* when sensor data changes automatically, and a sensor event object is put in the methods' parameter. The sensor event object is created by the system. It contains the following information: the raw sensor data, the sensor's type, the accuracy of the data and the timestamp of this event. Codes that use sensor values can be written in this method (Line 22).

TABLE I
SENSOR TYPES SUPPORTED BY THE ANDROID PLATFORM

Sensor	Type	Description
Accelerometer	Hardware	Measures the acceleration force
Ambient_Temperature	Hardware	Measures the ambient room temperature
Gravity	Software or Hardware	Measure the force of gravity
Gyroscope	Hardware	Measures a device's rate of rotation
Light	Hardware	Measures the ambient light level
Linear_Accelerometer	Software or Hardware	Measures the acceleration force, excluding the force of gravity
Magnetic_Field	Hardware	Measures the ambient geomagnetic field
Orientation	Software	Measures degrees of rotation that a device makes around all three physical axes
Pressure	Hardware	Measures the ambient air pressure
Proximity	Hardware	Measures the proximity of an object relative to the view screen of a device
Relative_Humidity	Hardware	Measures the relative ambient humidity
Rotation_Vector	Software or Hardware	Measures the orientation of a device
Temperature	Hardware	Measures the temperature of the device

```
1  //SensorActivity.java
2  public class SensorActivity extends Activity
3  {
4   float accx;
5   private SensorManager sensorManager;
6   private Sensor sensora;
7
8   public void onCreate(Bundle paramBundle)
9   {
10   super.onCreate(paramBundle);
11   setContentView(R.mainview);
12   sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
13   Log.d("sensor", "Get sensorManager Success");
14   sensora = sensorManager. getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
15   SensorEventListener accelerometerListener = new SensorEventListener()
16   {
17    public void onAccuracyChanged(Sensor arg0, int arg1)
18    {
19    }
20    public void onSensorChanged(SensorEvent arg0)
21    {
22     accx = arg0.values[0];
23     ......
24     // Do something with this sensor value
25    }
26   };
27   this.sensorManager.registerListener( accelerometerListener, sensora,
        SENSOR_DELAY_NORMAL);
28  }
29  ......
30 }
```

Fig. 1. Code example to use sensor

```
.super Landroid/app/Activity;
.source "SensorActivity.java"

# instance fields
.field accx:F
.field private sensorManager:Landroid/hardware/SensorManager;
.field private sensora:Landroid/hardware/Sensor;

# virtual methods
.method public onCreate(Landroid/os/Bundle;)V
   .parameter "savedInstanceState"

   invoke-super {p0, p1}, Landroid/app/Activity;->
         onCreate(Landroid/os/Bundle;)V
   const/high16 v2, 0x7f03
   invoke-virtual {p0, v2}, Lcom/my/sensorgravity/SensorActivity;->
         setContentView(I)V
   const-string v2, "sensor"
   invoke-virtual {p0, v2}, Lcom/my/sensorgravity/SensorActivity;->
         getSystemService(Ljava/lang/String;) Ljava/lang/Object;
   move-result-object v2
   check-cast v2, Landroid/hardware/SensorManager;
   iput-object v2, p0, Lcom/my/sensorgravity/SensorActivity;->
         sensorManager:Landroid/hardware/SensorManager;
   const-string v2, "sensor"
   const-string v3, "Get sensorManager Success"
   invoke-static {v2, v3}, Landroid/util/Log;->
         d(Ljava/lang/String;Ljava/lang/String;)I
   iget-object v2, p0, Lcom/my/sensorgravity/SensorActivity;->
         sensorManager:Landroid/hardware/SensorManager;
   const/4 v3, 0x1
   invoke-virtual {v2, v3}, Landroid/hardware/SensorManager;->
         getDefaultSensor(I)Landroid/hardware/Sensor;
   move-result-object v2
   iput-object v2, p0, Lcom/my/sensorgravity/SensorActivity;->
         sensora:Landroid/hardware/Sensor;
   new-instance v0, Lcom/my/sensorgravity/SensorActivity$1;
   invoke-direct {v0, p0, v1}, Lcom/my/sensorgravity/SensorActivity$1;->
         <init>(Lcom/my/sensorgravity/SensorActivity;
         Landroid/widget/TextView;)V
   .local v0, acceleromererListener: Landroid/hardware/SensorEventListener;
   iget-object v2, p0, Lcom/my/sensorgravity/SensorActivity;->
         sensorManager:Landroid/hardware/SensorManager;
   iget-object v3, p0, Lcom/my/sensorgravity/SensorActivity;->
         sensora:Landroid/hardware/Sensor;
   const/4 v4, 0x3
   invoke-virtual {v2, v0, v3, v4}, Landroid/hardware/SensorManager;->
         registerListener( Landroid/hardware/SensorEventListener;
         Landroid/hardware/Sensor;I)Z
   return-void
.end method
```

Fig. 2. Smali code for SensorActivity

- Last, register the listener (Line 27). By invoking the method *registerListener()*, the app registers a *SensorEventListener* into the system. The sensor to be registered is specified by the second parameter of this method. When a sensor's value changes, the system will only notify the apps that have registered this sensor. The third parameter of this method is used to set the data delay. The data delay controls the interval at which sensor events are sent to the app. In this example, the default data delay (*SENSOR_DELAY_NORMAL*) let this app receive the raw sensor values every 0.2 second.

B. Smali Code

Smali code is a disassembled version of the DEX format used by Dalvik, Android's JVM implementation. A given app's smali codes can be retrieved with Apktool [13], a tool for reversing Android apps. The smali codes obtained by disassembling are matching to the app's source codes. Hence, analysis performed on the smali codes directly is reasonable. We provide the corresponding disassembled smali codes of Figure 1 in Figure 2 and Figure 3. Figure 3 is the smali codes associated with the interface class *SensorEventListener*. Apktool generates an individual smali file for the interface class and inner class.

```
# virtual methods
.method public onAccuracyChanged(Landroid/hardware/Sensor;I)V
   .parameter "arg0"
   .parameter "arg1"
   .prologue
   return-void
.end method

.method public onSensorChanged(Landroid/hardware/SensorEvent;)V
   .parameter "arg0"
   .prologue
   const/4 v3, 0x0

   iget-object v0, p0, Lcom/my/sensorgravity/SensorActivity$1;->
         this$0:Lcom/my/sensorgravity/SensorActivity;
   iget-object v1, p1, Landroid/hardware/SensorEvent;->values:[F
   aget v1, v1, v3
   iput v1, v0, Lcom/my/sensorgravity/SensorActivity;->accx:F
   return-void
.end method
```

Fig. 3. Smali code for SensorActivity$1

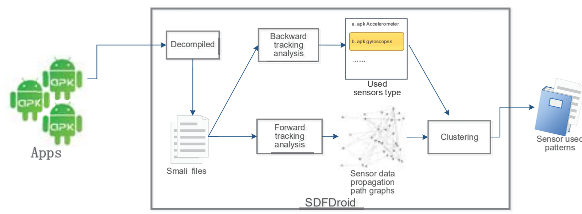

Fig. 4. Overview of SDFDroid

III. System Design

A. High-level Overview

We design and implement a tool called "SDFDroid" that performs static analysis of Android apps to identify the used sensor type and the sensor data propagation path in the apps. SDFDroid supports automated analysis. In addition, it is efficient to process a large number of apps. After we get all the apps' sensor data propagation path graphs, we calculate their similarities. We convert the similarities into distances between each graph before we cluster the graphs. In this work, DBSCAN [14] is employed to cluster the app samples based on the distances between each graph, as DBSCAN is a widely-used clustering algorithm based on density estimation that can describe the behaviors of various apps. The sensor usage patterns of each cluster are then constructed based on the clustering results. We illustrate the system's overview in Figure 4.

B. Static Analysis

In this Section, we describe the static analysis method used in this work. Static analysis methods try to cover all possible execution paths of the app. Complete codes are statically analyzed without the need of its execution. Therefore, static analysis can be done at app market level efficiently.

Android apps are packaged as APK files with compiled bytecode, additional metadata and resources in them. SDFDroid disassembles the APK files into smali code files with the help of Apktool. Then, SDFDroid parses the smali files and creates corresponding java objects of its contents.

Based on the previous step, SDFDroid performs two kinds of static analysis. One is backward tracking analysis that specifically focuses on the analysis of *registerListener()*'s second parameter to find which sensors the app use. The other is forward tracking analysis that focuses on finding the propagation path of the sensor data. We describe the detail of these two kinds of static analysis as follows (we use the codes in Figure 2 and Figure 3 for examples):

- Backward tracking analysis. In the backward tracking analysis, the analysis begins at the method *registerListener()*, whose second parameter specifies the used sensor. SDFDroid first finds out the corresponding smali code of *registerListener()* in the smali code files, then begins to track the register $v3$ which stores the value of this method's second parameter (see line 35, Figure 2). SDFDroid searches the smali codes in reverse order to find the value of $v3$. Then, we can find it gets an object value assigned at line 33. This object is a class field defined in Line 7. In the next step, SDFDroid searches the corresponding setters of this field. One setter can be found in Line 28. The value in register $v2$ is assigned to this field. After this, SDFDroid backtracks register $v2$, and finds $v2$ stores the result of invoking method *getDefaultSensor* (see Line 26-27). Hence, the registers corresponding to this method's parameters are backtracked (See register $v3$ in Line 26). If a register obtains a constant value, the tracking of this register will be terminated, and the result will be reported. As shown in this example, register $v3$ gets a constant value in Line 25. This result is reported, and we realize the sensor type is 1. Hence, the sensor used by this app is an accelerometer sensor. The relationship between the number and the real sensor type can be found in Android Developer Website [12].
- Forward tracking analysis. The forward tracking analysis is similar with the backward tracking analysis. SDFDroid first finds out the register which stores the value of the data need to be tracked in the smali codes. Then, it determines whether the register assigns its value to another object, or it is overwritten by another value. In forward tracking analysis, if a tracked register assigns its value to another object, the target object is added to the track queue. Otherwise, if the register is overwritten, the tracking of this register is stopped. For example, in Figure 3, sensor data is obtained from the system through the object: *Landroid/hardware/SensorEvent;->values:[F* and assigned to the register $v1$ (See Line 15). SDFDroid searches the smali files in order in forward tracking analysis. Finally, SFDroid finds register $v1$ assigns its value to a class field *accx* in Line 17, and finds this research process reaches the end of this method. Hence, the track of register $v1$ is stopped, and SDFDroid will search at where the class field *accx* is used in all other smali code files of the same app in the next step.

Both backward tracking analysis and forward tracking analysis have a queue to store objects that need to be tracked. Once the tracking process finds an object that needs to track, it puts this object in this queue. When the track of the current object is finished, SFDroid gets an object from the top of the queue and starts to track. The static analysis of an app is complete when the queue is empty. The used sensor type is reported in a *.xml* file. The sensor data propagation path is reported in a *.gexf* file. GEXF(Graph Exchange XML Format) is a language for describing complex network structures. Gephi can open the *.gexf* file and show the structure in a visible graph. Nodes in the graph are smali code lines, while edges represent data propagation direction.

C. Clustering Analysis

Once the used sensor type and sensor data propagation path graph of each app are generated, we perform clustering analysis on these graphs to construct their sensor usage patterns.

1) Distance calculation: The first step to perform clustering analysis is to calculate the similarity between each graph. One effective way to calculate the similarity between two graphs is calculating the edit distance between the two graphs. Graph edition distance measures the minimum number of graph edit operations (including insert or delete a node, insert or delete an edge or change the label of a node or edge) to transform one graph to another. However, calculating the edit distance between two graphs is NP-hard [15]. It is only applicable to graphs that have a small number of nodes. Although many researchers modified the basic algorithm of graph edit distance to improve the efficiency, it cannot apply to the sensor data propagation path graph that may have hundreds of nodes. Besides, an Android app market may have more than ten thousands apps to be analyzed. As we need a fast algorithm to calculate the similarity between each graph for apps on large-scale, in this work, we compute a *graph hash* for each graph and calculate similarity between each graph based on their hash values, a procedure inspired by Neighborhood Hash Graph Kernel (NHGK) originally proposed by Hido and Kashima [16].

NHGK is a kernel operation over labeled graphs. It has high expressiveness of the graph structure, and fast execution speed on graphs with large number of nodes. Hence, it is applicable to process graphs with hundreds of nodes such as the sensor data propagation path graphs. The main idea of NHGK is to integrate the information of a node and its neighbor nodes into a hash value. The calculation of a node's hash value is defined as follows:

$$H(v) = R(l(v)) \oplus (l(v_1^{NE})\oplus, \cdots, \oplus(l(v_n^{NE})))$$

where $l(v)$ represents the binary label that is transformed from the node's real label. $\oplus$ represents a bit-wise XOR. R means a one-bit rotation to the left. v^{NE} is the neighbor nodes of the node v. This algorithm can also be used iteratively to integrate information across neighbors up to a path with length k. The hash of a graph G is obtained by calculating hashes for each node. Then the similarity between two graphs is computed based on the number of nodes that has the same hash in two graphs.

In our work, when we calculate the similarity between each graph, we simplify the label of the nodes in the graphs at first to ensure its robustness. The label of nodes is an entire smali code line in original graphs, and we relabel the nodes only with the operation code in the code line. For example, node label *mul-float v1,v2,v3* is relabeled as *mul-float*. For some special operation, such as invoking a method, we also reserve the name of the invoked method. Then we count the number of distinct labels in total, and relabel them with different binary labels. Finally, hash value is computed for each node. By doing this, we can calculate the similarity between each sensor data propagation path graphs with the algorithm above.

2) Clustering: After we have calculated the similarity between each graph, we cluster the graphs to find their common characteristics and thus to discover the sensor usage patterns of different apps. The algorithm we used is Density-based spatial clustering of applications with noise (DBSCAN) [14]. DBSCAN is one of the most widely-used clustering algorithms based on density. The main idea behind this algorithm is that, for a set of points in a given space, clusters points that are closely located together and make outliers for points that lie alone in low-density regions. In a DBSCAN clustering, the points are classified as core points, reachable points and noise. A core point p is a point that has at least m neighbor points within distance d of it, and these neighbor points are considered to be directly reachable from p. A point q is a reachable point of p if there is a path from p to q where each point on the path can be directly reached from the point before it. The points are neither core points nor reachable points are noise. After the clustering, each cluster consists of some core points and their reachable points. The points that are reachable but not core points form the cluster's edge [17].

DBSCAN only needs two parameters: m and d to confirm core points. It does not need to specify the number of clusters. Therefore, it is suitable on our app set, as we do not know how many clusters the apps are associated with.

In our work, we transform the similarity between each graph generated with SDFDroid to a distance between each graph. For two graphs, the more similar they are, the smaller distance they have. We measure the distance by:

$$d = \frac{1}{1+s}$$

where s is the similarity between two graphs and d is the distance.

IV. Evaluation

A. Dataset

The dataset in the experiments is collected from *AppChina*, one of the biggest Android app markets in China. The dataset contains 22010 apps available in the market in December 2014. The apps belong to 14 subcategories under "Soft" category and 14 subcategories under "Game". The number of apps in each subcategory is shown in Figure 5. For each app, we get it's apk file, size and description.

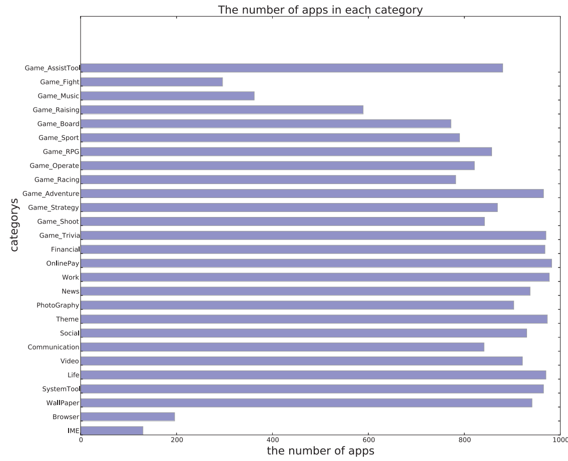

Fig. 5. The number of apps in each category

B. Analysis Results

Except some apps that cannot be disassembled by *Apktool*, we successfully analyzed 19914 apps. To our surprise, we find 10976 apps (55% of all) contain codes that can read sensor data. We present the detailed cluster and analysis results as follows.

1) Clustering results: In order to ensure the robustness and representative, in the process of clustering, we set the parameter d and m of DBSCAN to 0.1 and 10 respectively. After the clustering, we have 98 clusters with 8319 apps. We compare the samples in the same cluster and find the main reason why their sensor data propagation path are similar is the using of the same third-party libraries. Table II shows the third-party libraries used in the clusters that have more than 100 apps. Some third-party libraries have different sensor data propagation path in their different version, hence they are clustered into different clusters. The table shows *Tencent, Cocos2dx, Unity3d* is the most frequently used third-party development libraries which contain sensor related codes. *Com.tencent.mm.sdk* is a Jar package used to share the moments with one's friends in *WeChat* (The most widely-used IM tool in China). *Cocos2dx* is a framework for building 2D games and other graphical apps. *Unity3d* is a third-party game development toolkit that helps developers to build 3D games and real time 3D animations.

Figure 6 is a matrix describing clustering results by category (The number of Y-axis and category map can be found in Appendix), which shows us each category's sensor usage patterns. This figure illustrates that different category usually has different sensor usage patterns. Category IME (301), Browser (302) and Theme (309) have few apps that have sensor related codes. Cluster 14, which mainly denotes com.tencent.mm.sdk, only appears in category Wallpaper. This is because most of apps in category Wallpaper are developed by the same developer. Comparing with categories Tools, Games are more likely to use sensors, especially the games in category Adventure.

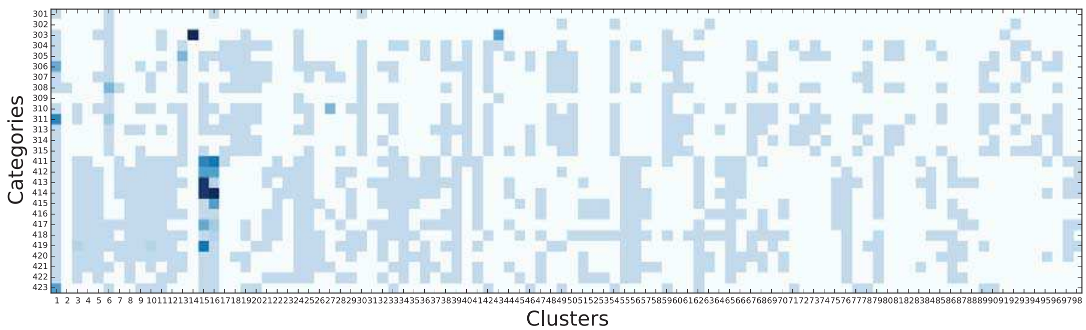

Fig. 6. Matrix for cluster result by category. Each grid corresponds to a cluster and category, and the depth of the grid's color represent the number of apps in this grid. The more apps in the corresponding cluster and category, the deeper the color.

TABLE II
THE THIRD-PARTY LIBRARIES IN EACH CLUSTER

ID	Cluster Number	The Number Of Apps	Third-Party Libraries
1	15	1097	org.cocos2dx.lib
2	16	938	com.unity3d.player
3	1	619	com.tencent.mm.sdk
4	14	398	com.tencent.mm.sdk
5	6	397	com.tencent.mm.sdk
6	3	313	org.cocos2dx.lib
7	25	214	com.adobe.air.Accelerometer
8	24	196	com.badlogic.gdx.backends.android
9	13	185	org.apache.cordova
10	43	177	com.mobi.view
11	10	166	com.tencent.mm.sdk org.cocos2dx.lib
12	34	165	com.unity3d.player com.tencent.mm.sdk com.millennialmedia.android.AccelerometerHelper com.mobclix.android.sdk
13	39	160	com.unity3d.player
14	62	137	org.andengine.engine.Engine
15	40	130	cn.sharesdk.onekeyshare.Shake2Share
16	33	112	com.adsmogo.mriad com.tapjoy.mraid org.ormma.controller
17	9	105	com.adobe.air
18	67	104	com.baidu.location cn.sharesdk.onekeyshare.Shake2Share com.umeng.socialize.sensor.UMShakeSensor
19	8	102	com.unity3d.player com.zhuohuamg.game.util.ShakeListener com.tencent.mm.sdk
20	27	101	com.trid.tridad.TriDContentsView

2) Used sensor type: According to backward tracking analysis results, accelerometer is the most frequently used sensor. More than 70 percent of apps use this sensor as indicated in Figure 7. This is mainly because the popular function, such as Shaking, needs this sensor. All the widely used third-party libraries, such as *Cocos2dx, Unity3D* and *OneKeyShare*, contain codes to use this sensor. Gyroscope sensor is often used to display the 3D model of an object. Proximity sensor almost only appears in *com.tencent.mm.sdk* while gravity sensor almost only appears in *Unity3D*. The third-party libraries that provide map function, such as *com.baidu.mapapi*, will use the Orientation sensor and Magnetic field sensor. The apps that can distinguish QR Code may use the light sensor. Motion sensors and position sensors are widely used, while environmental sensors are seldom used, except the light sensor. This is because many smartphones do not have the

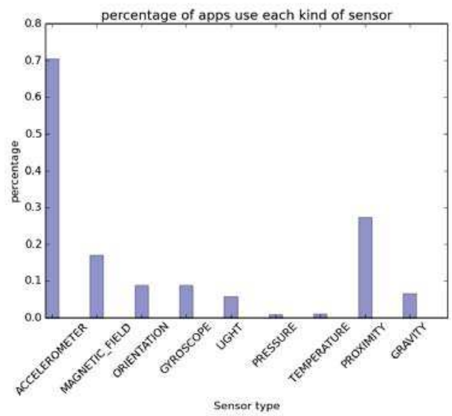

Fig. 7. Percentage of apps that use each type of sensor

hardware sensors to measure the temperature, pressure or humidity. More than 72 percent of apps only use one sensor, mostly accelerometer sensor. *Unity3D* can access four types of sensors. However, *madhouse*, an ad library, registers nine types of sensors for their ad display. Although attractive ads may need the data from sensors, we do not think nine types of sensors are necessary. We summarize the detail about the used sensor type in the most used third-party libraries in Table III.

3) Discussion: Our studies on an Android app market do not find any app that steals users' sensor data (The data read from the sensors which are not protected by Android permissions). Different from the methods of stealing users' location, contact or message, which often use Java reflection API to read data and send the data to the Internet, apps just use the sensor data in their local codes. However, since the embedded sensor can be directly used without requirement of permissions, some third-party libraries register all the sensor recklessly, no matter whether they really need. This not only lead to more power consumption, but also more easily be exploited by malicious apps. Though many apps contain the third-party libraries that can read sensor data, they do not

TABLE III
USED SENSOR TYPE IN EACH THIRD-PARTY LIBRARY

Libraries	Accelerometer	Magnetic Field	Orientation	Gyroscope	Light	Pressure	Temperature	Proximity	Gravity
org.cocos2dx	✔								
com.unity3d.player	✔	✔			✔				✔
com.tencent.mm.sdk								✔	
cn.sharesdk.onekeyshare	✔								
com.umeng.socialize.sensor.UMShakeSensor	✔								
com.adsmogo.mriad	✔	✔		✔	✔	✔	✔	✔	
com.adchina.android.ads.views	✔								
com.madhouse.android.ads	✔	✔	✔	✔	✔	✔	✔	✔	✔
com.millennialmedia.android	✔								
com.mobi.view	✔								
com.adcocoa	✔								
com.baidu.location	✔	✔			✔			✔	
com.baidu.navi	✔		✔						
com.amap.api.mapcore	✔	✔	✔						
com.google.zxing					✔				
com.badlogic.gdx		✔							
com.adobe.air	✔			✔					
com.mobclix.android.sdk	✔		✔	✔					
com.trid.tridad	✔								

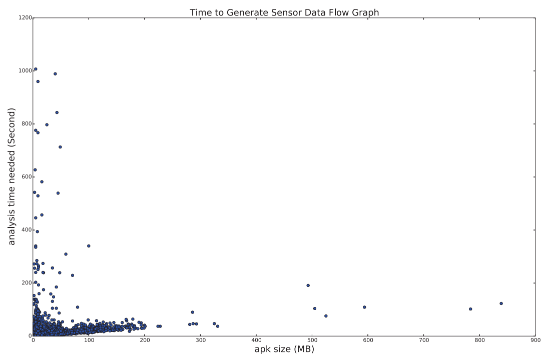

Fig. 8. Runtime for generating sensor data propagation path graph

functionally need it. Hence, we suggest the third-party libraries put the sensor related codes in a standalone package, and let the developers chose to use it or not by themselves.

4) Runtime Performance: We conducted the experiments on a quad-core machine with Inter Core i5 CPU (3.10 GHz $\times$ 4) and 8GB RAM, running Ubuntu 14.04. Figure 8 illustrates the runtime performance of SDFDroid. It shows the time for generating sensor data propagation path graph and backward tracking analysis for each app. Most of the apps can be processed within 3 minutes. The experimental results indicate that SDFDroid is efficient to explore sensor usage behaviors of Android applications.

V. Limitations

Our method may not perform very well on the Apps with smali code obfuscation. The *Reflection* API may also affect the analysis results. Another important factor affecting the analysis results is the use of Native methods. Some third-party development libraries write their codes that implement the sensor data process logic in the Native function with the help of Native Development Toolkit(NDK). The Native function is written by C++ and cannot be disassembled to smali codes, hence it cannot be analyzed by SDFDroid. However, the state-of-art analysis tools like Flowdroid also meet these shortcomings. SDFDroid uses the methods provided by *Apktool* to disassemble Android apps, hence we cannot analyze the apps that cannot be disassembled by *Apktool*.

VI. Related Work

The related work of this paper mainly falls into two aspects: static analysis on Android apps and graph-based program analysis of Android apps.

- *Static analysis on Android apps.* Many efforts have been made to perform static analysis on Android apps. Androgurad [18] is a toolset widely used in analyzing decompiled Android apps. It can be used to generate method invoke graph or detect malicious apps via signature matching. Androguard needs to be interactive with user to perform the analysis process, hence it cannot analyze a large number of apps automatically. Another well-known static analysis tool for Android apps is FlowDroid [19] which is based on Soot [20]. It conducts precise static taint analysis on Android apps and generates the propagation path for sensitive data, but the analysis result relies on the predefined source and sink API of sensitive data. William Kliebe *et al.* [21] propose a tool combined FlowDroid and Epicc [22] to track both inter-component and intra-component data flow in a set of Android apps. Wei *et al.* [23] build a general framework called Amandroid for security analysis of Android apps. However, this tool needs a very long time to analyze a single app. These tools need to generate a control flow graph for the whole app before they perform analysis, which cost much more time than SDFDroid that only concern sensor related codes. In other words, these tools are too heavyweight for the purposes of our study. The work which is most similar to ours is SAAF [24], a static Android analysis framework for Android apps. The difference between SDFDroid and SAAF is the ability to perform forward tracking analysis.
- *Graph-based program analysis of Android apps.* Gascon *et al.* [25] develop a method for Android malapps detection based on function call graphs. They employ an explicit mapping to map call graphs to a feature space, then train a support vector machine to distinguish malapps from benign apps. Zhang *et al.* [26] propose a

novel approach to classify Android malapps via weighted contextual API dependency graph. They assign weights to different API nodes in the graph when measuring similarity by graph edit distance algorithm. That means critical APIs, such as API requiring permission check, have greater weights and more easily to influence the classification results. The difference between our work and the related work is that we use data flow graphs to do clustering while they use API invoking graphs for classification.

VII. Conclusion

In this paper, we design and implement a tool called SDFDroid to analyze sensor usage patterns in current Android apps. SDFDroid performs forward tracking analysis to find sensor data propagation path and backward tracking analysis to find used sensor type. Through extensive experiments on a widely-used App market, AppChina, we find that the Android apps often preform sensor related functions with the help of third-party libraries. Accelerometer sensor is the most frequently used sensor. Though many apps only register one type of sensor, there are some apps registering nine types of sensors. This indicates the need for better regulating the sensor usage in Android apps. In the future work, we will extend the analysis to Native code, and generate complete sensor data propagation path to characterize the app's sensor related functions accurately.

VIII. Acknowledgment

The work reported in this paper is partially supported by the Fundamental Research funds for the central Universities of China (No. K15JB00190), Shanghai Key Laboratory of Integrated Administration Technologies for Information Security, the Ph.D. Programs Foundation of Ministry of Education of China (No. 20120009120010), the Scientific Research Foundation for the Returned Overseas Chinese Scholars, State Education Ministry (No. K14C300020), and in part by the 111 Project (B14005).

Appendix

The number and category map

301	IME	302	Browser	303	WallPaper	304	SystemTool
305	Life	306	Video	307	Communication	308	Social
309	Theme	310	PhotoGraphy	311	News	313	Work
314	OnlinePay	315	Financial	411	Game_Trivia	412	Game_Shoot
413	Game_Strategy	414	Game_Adventure	415	Game_Racing	416	Game_Operate
417	Game_RPG	418	Game_Sport	419	Game_Board	420	Game_Raising
421	Game_Music	422	Game_Fight	423	Game_AssistTool		

References

[1] I.D.Corporation, "Smartphone os market share, q1 2015," http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2015-05.

[2] Android, "Android sensor overview," http://developer.android.com/guide/topics/sensors/sensors_overview.html, 2015-03-28.

[3] L. Cai and H. Chen, "Touchlogger: inferring keystrokes on touch screen from smartphone motion," in *Proceedings of the 6th USENIX conference on Hot topics in security*. USENIX Association, 2011, pp. 9–9.

[4] A. J. Aviv, B. Sapp, M. Blaze, and J. M. Smith, "Practicality of accelerometer side channels on smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 41–50.

[5] R. Spreitzer, "Pin skimming: Exploiting the ambient-light sensor in mobile devices," in *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM, 2014, pp. 51–62.

[6] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2012, pp. 113–124.

[7] eSecurity Planet, "Penn state, ibm researchers develop taplogger android trojan," http://www.esecurityplanet.com/mobile-security/penn-state-ibm-researchers-develop-taplogger-android-trojan.html, 2012-04-23.

[8] T. H. News, "Taplogger android trojan can determine tapped keys," http://thehackernews.com/2012/04/taplogger-android-trojan-can-determine.html, 2012-04-21.

[9] J. Zhu, P. Wu, X. Wang, and J. Zhang, "Sensec: Mobile security through passive sensing," in *Computing, Networking and Communications (ICNC), 2013 International Conference on*. IEEE, 2013, pp. 1128–1133.

[10] W.-H. Lee and R. B. Lee, "Multi-sensor authentication to improve smartphone security," in *Conference on Information Systems Security and Privacy*, 2015.

[11] AppBrain, "Google play stats," http://www.appbrain.com/stats/, 2015-05-23.

[12] Android, "Android sensor type," http://developer.android.com/reference/android/hardware/Sensor.html, 2015-03-29.

[13] Apktool, "Apktool," http://ibotpeaches.github.io/Apktool/, 2015-05-20.

[14] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *Kdd*, vol. 96, 1996, pp. 226–231.

[15] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing Stars: On Approximating Graph Edit Distance," *Proceedings of The Vldb Endowment*, vol. 2, pp. 25–36, 2009.

[16] S. Hido and H. Kashima, "A Linear-Time Graph Kernel," *Ninth IEEE International Conference on Data Mining, 2009. ICDM '09.*, pp. 179–188, Dec. 2009.

[17] WIKI, "Dbscan," http://en.wikipedia.org/wiki/DBSCAN, 2015-04-05.

[18] A. Desnos, "Androguard: Reverse engineering, malware and goodware analysis of android applications... and more (ninja!)," http://code.google.com/p/androguard, 2013-03-26.

[19] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, 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*. ACM, 2014, p. 29.

[20] Soot, "Soot," http://sable.github.io/soot/, 2015-03-30.

[21] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. ACM, 2014, pp. 1–6.

[22] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *Proceedings of the 22nd USENIX Security Symposium*. Citeseer, 2013.

[23] F. Wei, S. Roy, X. Ou *et al.*, "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*. ACM, 2014, pp. 1329–1341.

[24] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing droids: program slicing for smali code," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 1844–1851.

[25] 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*. ACM, 2013, pp. 45–54.

[26] M. Zhang, Y. Duan, H. Yin, and Z. 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, pp. 1105–1116.