Characterizing Sensor Leaks in Android Apps

Xiaoyu Sun*, Xiao Chen*, Kui Liu†, Sheng Wen‡, Li Li*, John Grundy*,
*Monash University, Melbourne, Australia
{Xiaoyu.sun, xiao.chen, Li.Li, john.grundy}@monash.edu
†Nanjing University of Aeronautics and Astronautics, Nanjing, China
kui.liu@nuaa.edu.cn
‡Swinburne University of Technology, Melbourne, Australia
swen@swin.edu.au

2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE) | 978-1-6654-2587-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISSRE52982.2021.00058

***Abstract*—While extremely valuable to achieve advanced functions, mobile phone sensors can be abused by attackers to implement malicious activities in Android apps, as experimentally demonstrated by many state-of-the-art studies. There is hence a strong need to regulate the usage of mobile sensors so as to keep them from being exploited by malicious attackers. However, despite the fact that various efforts have been put in achieving this, i.e., detecting privacy leaks in Android apps, we have not yet found approaches to automatically detect sensor leaks in Android apps. To fill the gap, we designed and implemented a novel prototype tool, SEEKER, that extends the famous FlowDroid tool to detect sensor-based data leaks in Android apps. SEEKER conducts sensor-focused static taint analyses directly on the Android apps' bytecode and reports not only sensor-triggered privacy leaks but also the sensor types involved in the leaks. Experimental results using over 40,000 real-world Android apps show that SEEKER is effective in detecting sensor leaks in Android apps, and malicious apps are more interested in leaking sensor data than benign apps.**

I. INTRODUCTION

As of 1st January 2021, there are nearly three million Android apps available on the official Google Play app store. The majority of them (over 95%) are made freely accessible to Android users and cover every aspect of users' daily life, such as supporting social networking, online shopping, banking, etc. Many of these functionalities are supported by application interfaces provided by the Android framework, essentially fulfilled by a set of hardware-based sensors [1]. For example, Android apps often leverage accelerometer sensors to detect the orientation of a given smartphone and user movement, and the temperature sensor to detect the device's temperature.

Despite being needed to support the implementation of many diverse Android apps, mobile phone sensors can also be abused to achieve malicious behaviors. There have been many reports of apps that exploit sensors in Android devices to conduct malicious activities. For example, Adam et al. [2] have experimentally shown that the accelerometer sensor could be leveraged as a side-channel to infer mobile users' tap and gesture-based input. Xu et al. [3] have also demonstrated the possibility of this attack by presenting to the community a Trojan application named *TapLogger* to silently infer user's tap inputs based on the device's embedded motion sensors. Similarly, Schlegel et al. [4] have provided another Trojan application called *Soundcomber* that leverages the smartphone's audio sensor to steal users' private information.

These studies have experimentally shown that the leaks of Android sensor data can cause severe app security issues. We argue that there is thus a strong need to invent automated approaches to detect such sensor leaks in Android apps before publishing them onto app markets. To the best of our knowledge, existing works focus on detecting certain types of sensor usage and its corresponding suspicious behaviors. None of them are designed as a generic approach for systematically revealing data leaks in all types of Android sensors. Also, these works mainly concentrate on discovering and understanding the usage patterns of Android embedded sensors, which do not involve completed data flow analysis to pinpoint sensitive data leaks caused by sensors.

Although many generic approaches to detect privacy leaks in Android apps have been proposed, none can be directly applied to achieve our purpose, i.e., detecting generic sensor leaks in Android apps. Indeed, the famous FlowDroid tool has been demonstrated to be effective in detecting method-based privacy leaks in Android apps. It performs static taint analysis on Android apps' bytecode and attempts to locate data-flow paths connecting two methods, i.e., from a *source* to a *sink* method. Here, *source* refers to such methods that obtain and return sensitive information from the Android framework (e.g., get device id), while *sink* refers to such methods that perform dangerous operations such as sending data to remote servers. FlowDroid has been designed as a generic approach. It has provided a means for users to pre-define the needed *source* and *sink* methods. Unfortunately,FlowDroid does not allow users to configure fields as *sources* so as to support the detection of privacy leaks flowing from *fields* to sensitive operations (i.e., *sink*). Since sensor data in Android is mostly provided via fields, FlowDroid cannot be directly applied to detect sensor leaks in Android apps.

To address this research gap, we designed and implemented a prototype tool, SEEKER, to automatically detect sensor data leaks in Android apps. We extend the open-source tool FlowDroid to support field-triggered sensitive data-flow analyses. Our new SEEKER further performs a detailed static code analysis to infer the sensor types involved in the sensitive data-flows as the leaked sensor data is not directly associated with

Li Li is the corresponding author.

2332-6549/21/$31.00 ©2021 IEEE
DOI 10.1109/ISSRE52982.2021.00058

the sensor type. (we detail this challenge in Section III-C). We then apply SEEKER to detect and characterize sensor leaks in real-world Android apps. Based on 40,000 randomly selected Android apps, including 20,000 benign apps and 20,000 malicious apps, our experimental results show that SEEKER is effective in detecting sensor leaks in Android apps. We also find that malware is more interested in obtaining and leaking sensor data than benign apps, and Accelerometer and Magnetic are among the most targeted sensors by those malicious apps.

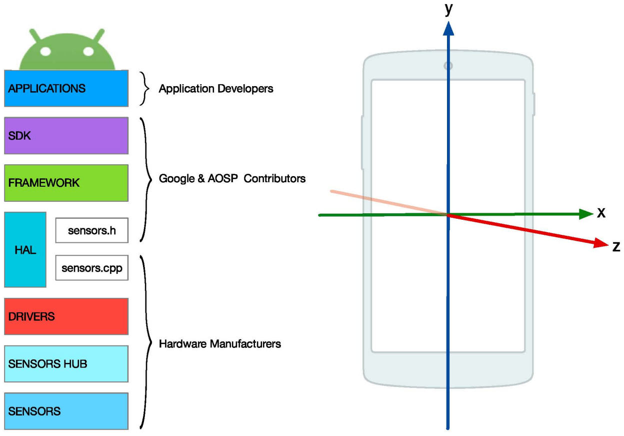
We make the following main contributions in this work:

- We have designed and implemented a prototype tool, SEEKER (Sensor leak finder), that leverages static analysis to automatically detect privacy leaks originated from Android sensors.
- We apply SEEKER to analyze both malware and benign apps at a large scale. Our results show many sensor leaks that are overlooked by the state-of-the-art static analysis tool.
- We have demonstrated the effectiveness of our tool by evaluating the sensor leaks it highlights.

II. BACKGROUND AND MOTIVATION

A. How sensors work in Android platforms

Figure 1(a) depicts the Android sensor stack. Sensors are Microelectromechanical systems (MEMS) chips that detect events or changes in surrounding environment. After the sensors capture the events, data is optionally passed on to the Sensors Hub. This Sensors Hub performs low-level computation as a support to the sensors, such as step counting and sensor fusion. Then the Drivers and Hardware Abstraction Layer (HAL) handles the interaction between the hardware and the Android framework. Finally, the Android apps access the sensor data through APIs provided by the Android Software Development Kit (SDK).



(a) (b)

Fig. 1. Layers and Coordinate system of the Android sensor stack. Source: https://developer.android.com/guide/topics/sensors/sensors_overview

In general, Android platform provides three broad categories of sensors for measuring motion, orientation, and various environmental conditions of the device:

- **Motion sensors**: These are used to monitor device movement, such as tilt, shake, rotation, or swing. The movement usually reflects direct user input or the physical environment around the device. Motion sensors include the accelerometer, the gyroscope, the step counter, etc.
- **Position sensors**: These determine the physical position of a device in the world's frame of reference or the orientation of a device. Position sensors include the geomagnetic field sensor, the proximity sensor, etc.
- **Environmental sensors**: These monitor various environmental properties, such as relative ambient humidity, illuminance, ambient pressure, and ambient temperature near the device. Examples of environmental sensors include the light sensor, the pressure sensor, etc.

Android uses a standard 3-axis coordinate system to represent data values, as shown in Figure 1(b). The X-axis is defined relative as horizontal, the Y-axis is vertical, and the Z-axis points towards the outside of the screen face. This coordinate system is unalterable when the device's screen orientation changes, which means the sensor's coordinate system remains the same even if the device is on the move.

Table I summarises the main embedded sensors supported by Android with their categories, types, and descriptions. The Android sensor framework provides both hardware-based and software-based sensors. Hardware-based sensors are accessed by reading the data directly from physical components built in the device, such as acceleration, geomagnetic field strength, or angular change. Software-based sensors derive their data from one or more of the hardware-based sensors. Examples of software-based sensors includes the linear acceleration sensor and the gravity sensor.

```
public class SensorActivity extends Activity implements
     SensorEventListener {
 private SensorManager sensorManager;
 private Sensor pressure;
 private List<Sensor> deviceSensors;
 @Override
 public final void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
  // Get an instance of the sensor service, and use
     that to get an instance of a particular sensor.
  sensorManager = (SensorManager)
     getSystemService(Context.SENSOR_SERVICE);
  deviceSensors =
     sensorManager.getSensorList(Sensor.TYPE_ALL);
  pressure =
     sensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE);
 }
 @Override
 public final void onAccuracyChanged(Sensor sensor, int
     accuracy) {
  // Do something here if sensor accuracy changes.
 }
 @Override
 public final void onSensorChanged(SensorEvent event) {
  float millibarsOfPressure = event.values[0];
  // Do something with this sensor data.
 }
 @Override
 protected void onResume() {
 //Register a listener for the sensor.
 super.onResume();
 sensorManager.registerListener(this, pressure,
     SensorManager.SENSOR_DELAY_NORMAL);
 }
 @Override
 protected void onPause() {
```

TABLE I
SENSOR TYPES SUPPORTED BY THE ANDROID PLATFORM.

Sensor Type	Sensor Category	Description
Gravity	Motion sensor	Provides a three dimensional vector indicating the direction and magnitude of gravity
Linear acceleration	Motion sensor	Provides a three-dimensional vector representing acceleration along each device axis
Rotation vector	Motion sensor	Provides the orientation of the device
Significant motion	Motion sensor	Triggers an event each time significant motion is detected and then it disables itself
Step counter	Motion sensor	Provides the number of steps taken by the user since the last reboot
Step detector	Motion sensor	Triggers an event each time the user takes a step
Accelerometer	Motion sensor	Measures the acceleration applied to the device, including the force of gravity
Gyroscope	Motion sensor	measures the rate of rotation in rad/s around a device's x, y, and z axis
Game rotation	Position sensor	Identical to the Rotation vector sensor, except it does not use the geomagnetic field
Geomagnetic rotation	Position sensor	Similar to the rotation vector sensor, but it doesn't use the gyroscope
Geomagnetic field	Position sensor	Monitor changes in the earth's magnetic field
Uncalibrated magnetometer	Position sensor	Similar to the geomagnetic field sensor, except that no hard iron calibration is applied
Proximity sensor	Position sensor	Determine how far away an object is from a device
Light	Environment sensor	Provides Illuminance
Pressure	Environment sensor	Provides ambient air pressure
Temperature	Environment sensor	Provides device temperature
Ambient temperature	Environment sensor	Provides ambient air temperature
Humidity	Environment sensor	Provides ambient relative humidity

```
31      //Unregister the sensor when the activity pauses.
32      super.onPause();
33      sensorManager.unregisterListener(this);}}
```

Listing 1. Example of demonstrating how to access the device's sensors.

The Android sensor framework provides several APIs for developers to access its sensors and acquire raw data. We present an example in Listing 1 to elaborate on how one identifies and determines sensor capabilities. First, to identify the sensors on a device, developers need to obtain the sensor service by calling the `getSystemService()` method and then passing the constant ”Context.SENSOR_SERVICE” as an argument (line 10). After that, developers can get a list of all sensors on a device through invoking `getSensorList(int type)`(line 11). To access a specific sensor, method `getDefaultSensor(int type)` can be called with a specific type constant (line 12). To monitor sensor events, the developer should implement two callback methods that are exposed through `SensorEventListener` interface, which are `onAccuracyChanged()` and `onSensorChanged()` (lines 15-17 and 19-22, respectively). Whenever a sensor detects a change, the Android system will call these two methods to report the following details to users:

Sensor accuracy changes When the sensor's accuracy changes, `onAccuracyChanged()` will provide users with a reference of the `Sensor` object and the new accuracy status of this sensor.

Sensor value changes When a sensor obtains a new value, `onSensorChanged()` will provide users with a `SensorEvent` object, which contains the accuracy of the data, the sensor object, the timestamp when the data was generated, and the new data that the sensor recorded.

Last, the `onResume()` (lines 24-28) and `onPause()` (lines 30-34) callback methods are used to register and unregister the listener for the sensor. When an activity is paused, the related sensors should be disabled to avoid battery draining.

B. Motivation

TABLE II
EXAMPLES OF SENSOR-BASED CYBERSECURITY ATTACKS.

Sensor Category	Sensor Type	Attack Description
Motion sensor	Accelerometer	sniffing smartwatch passwords [5]
	Accelerometer, Gyroscope	Text Inference [6]
	Accelerometer, Gyroscope	Motion-based keystroke inference [7]
	Accelerometer, Gyroscope	Keystroke inference on Android [8]
	Accelerometer	Accelerometer side channel attack [2]
	Accelerometer	Touchscreen area identification [9]
	Accelerometer	Decoding vibrations from nearby keyboards [10]
	Gyroscope	Single-stroke language-agnostic keylogging [11]
	Accelerometer, Gyroscope	Inferring Keystrokes on Touch Screen [12]
	Accelerometer, Gyroscope	Inferring user inputs on smartphone touchscreens [3]
	Accelerometer, Gyroscope	Keystroke Inference [13]
	Accelerometer	keystrokes Inference in a virtual environment. [14]
	Accelerometer, Gyroscope	Risk Assessment of motion sensor [15]
	Accelerometer, Gyroscope	Infer tapped and traced user input [16]
	Accelerometer, Gyroscope	Motion-based side-channel attack [17]
	Accelerometer	Keystroke inference with smartwatch [18]
	Accelerometer	Motion leaks through smartwatch sensors [19]
	Accelerometer	Side-channel inference attacks [20] [21]
	Accelerometer	Smartphone PINs prediction [22]
	Gyroscope	Inferring Mechanical Lock Combinations [23]
	Accelerometer, Gyroscope	Inference of private information [23]
	Accelerometer, Gyroscope	Typing privacy leaks via side-Channel from smart watch [24]
	Accelerometer, Magnetometer	Input extraction via motion sensor [25]
	Gyroscope	Recognizing speech [26]
Position sensor	Magnetic	Compromising electromagnetic emanations [27]
	Magnetic	My Smartphone Knows What You Print [28]
	Magnetic	Location detection [29]
Environment sensor	Light Sensor	Optical eavesdropping on displays [30]

Sensors have been widely adopted for launching side-channel attacks against smart devices [31]. Table II summarizes a diverse set of sensor-based attacks targeting smartphones and smartwatches. Since accessing sensitive sensor data does not require any security checks (e.g., permission check), attackers can easily trigger malicious behaviors by making use of such data. As revealed in the table, generally, sensor leakage are performed with the aim of (1) keystroke inference, (2) task inference (refers to a type of attack which reveals the information of an on-going task or an application in a smart device), (3) location inference, and (4) eavesdropping. For example, motion and position sensors can be exploited for keystroke inference, leading to severe privacy leaks such as passwords, credit card information, etc. Light sensor is found to eavesdrop acoustic signals in the vicinity of the device, causing private information leak. Magnetic sensors can be exploited to compromise electromagnetic emanations, which would affect the confidentiality of the devices.

As a concrete example, Lu et al. [5] revealed that sensitive intercepting password could be accessed through motion data on the smartwatch's onboard sensors. They proposed *Snoopy*, a password extraction and inference approach via sensor data for PIN attack, which could affect smartwatch users in a non-invasive way. *Snoopy* extracts the segments of motion data when users entered passwords and then applies deep learning techniques to infer the actual passwords. Figure 3 gives two examples of the differences of the motion sensor data changes when the user swipes or taps a password on a smartwatch. *Snoopy* demonstrates the feasibility of sensor data leaks by intercepting password information entered on smartwatches. Such real-world sensor-enabled attacks motivated us to provide automatic tools for characterizing universal sensor leaks in Android Apps that have been long overlooked.

III. Approach

This work aims to automatically detect information leaks of onboard sensors in Android apps. To this end, we design and implement a prototype tool called SEEKER for achieving this purpose. Figure 2 describes the overall working process of SEEKER, which is mainly made up of three modules, namely Sensitive Sensor Source Identification, Sensor-triggered Static Taint Analysis and Sensor Type Inference.

A. Sensitive Sensor Source Identification

The first module, *Sensitive Sensor Source Identification*, aims to identify sensor-related sources that access and obtain sensitive information related to the device's sensors. As reported by Liu et al. [32], Android sensor data can be obtained through invoking sensor-related APIs or directly accessing local fields in which the sensor data is stored. In this work, we take both of these types into consideration, aiming at pinpointing all the possible sensor-triggered privacy leaks.

To do this we need to identify all the sensor-related sources, including both Android methods and fields. For Android methods, we use the well known SUSI tool [33] to obtain sensor-related source methods. SUSI is a novel machine-learning guided approach that scans Android API's source code to predict *source* and *sink* methods, based on a training set of hand-annotated sources and sinks. In this work, we launch SUSI on the latest Android Open Source Project (i.e., AOSP version 11.0) and manually filter out non-sensor related source methods.

To identify sensor-related fields (as sources), there is no existing approach to achieve such a purpose. We resort to a manual process of going through the Android Developers' Documentation to identify source fields storing sensitive sensor information. The identified fields are then discussed and confirmed by the authors by measuring whether leaking such information would potentially expand the attack surface to users' privacy. Finally, we identified 79 fields and 20 methods as the sources. Table III lists the selected sources that indeed introduce leaks in our experimental dataset. A full list of field and method sources can be found in the *SourcesAndSinks.txt* file of our open-source project[1].

B. Sensor-triggered Static Taint Analysis

The ultimate goal of SEEKER is to detect sensor-related data leaks. To this end, we implement the *Sensor-triggered Static Taint Analysis* module that extends state-of-the-art tool FlowDroid [34] to facilitate sensor-related data leak detection. FlowDroid detects data leaks by computing data flows between sources and sinks. FlowDroid defined a sensitive data flow happens when a suspicious "tainted" information passes from a source API (e.g., `getDeviceId`) to a sink API (e.g., `sendTextMessage`).

TABLE III
The list of sensitive sensor sources.

Sensor-related Source	Source Type
SensorEvent#values	Field
SensorEvent#timestamp	Field
Sensor#getName()	Method
Sensor#getVendor()	Method
Sensor#getVersion()	Method
SensorManager#getDefaultSensor(int)	Method
Sensor#getMaximumRange()	Method
SensorManager#getSensorList(int)	Method
Sensor#getType()	Method
Sensor#getResolution()	Method
Sensor#getPower()	Method

FlowDroid is a state-of-the-art tool and it provides a highly precise static taint-analysis model, especially for Android applications. However, FlowDroid only takes API statements as sources or sinks, leading to false negatives because of the lack of field-triggered sources. Thus, in this work, we extend FlowDroid by supporting field statement as sources, so as to pinpoint data leaks originated from specific field sources of interest.

Our preliminary study discovered that certain sensor-related data leaks are sourced from data stored in class fields (e.g., android.hardware.SensorEvent#values). We therefore implemented our own class that implements the `ISourceSinkDefinitionProvider` interface in FlowDroid for supporting the declaration of fields as sources. Also, based on the feature of class fields, we defined a new model names AndroidField extends from `SootFieldAndMethod`. After loading a specific field statement from *source&sink.txt* file, we apply a field pattern regular expression to convert it to the AndroidField model.

FlowDroid has the ability to compute data flow connections between all possible statements. In the implementation of FlowDroid, `ISourceSinkManager` interface marks all statements as possible sources and then records all taint abstractions that are passed into `getSourceInfo()`. To that end, we pass the constructed field model as a source statement to the following taint analysis process. In this way, sensitive data flow can be detected starting at given field source statements.

C. Sensor Type Inference

The primary goal of SEEKER is to detect data leaks from Android platform sensors. With the help of FlowDroid's taint analysis, SEEKER's second module can detect sensor-triggered sensitive data flows. Unfortunately for the field-triggered ones, the identified data-flows only show that there is sensor data leaked but do not tell from which sensor the data is collected. The sensor type information is important for helping security analysts understand the sensor leaks. Therefore, in our last

[1] https://github.com/MobileSE/SEEKER

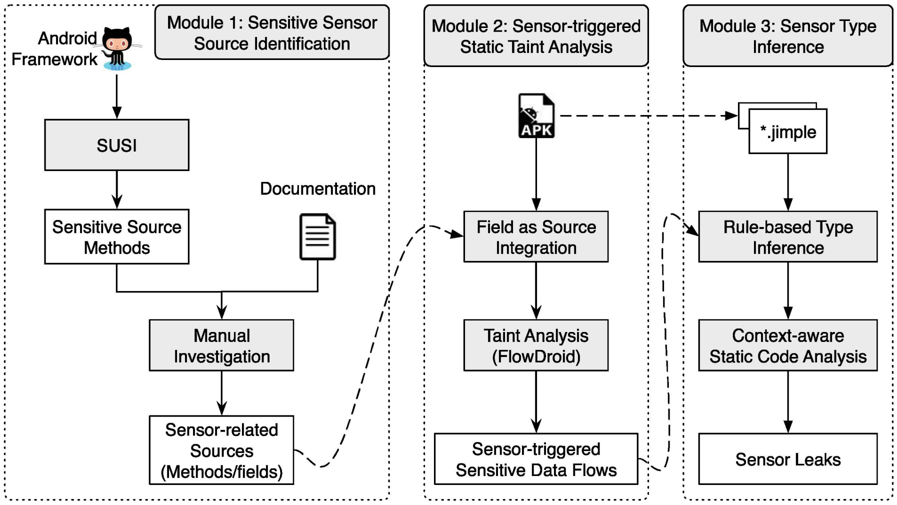

Android Framework
Module 1: Sensitive Sensor Source Identification
SUSI
Sensitive Source Methods
Documentation
Manual Investigation
Sensor-related Sources (Methods/fields)
Module 2: Sensor-triggered Static Taint Analysis
APK
Field as Source Integration
Taint Analysis (FlowDroid)
Sensor-triggered Sensitive Data Flows
Module 3: Sensor Type Inference
*.jimple
Rule-based Type Inference
Context-aware Static Code Analysis
Sensor Leaks

Fig. 2. The working process of our approach.

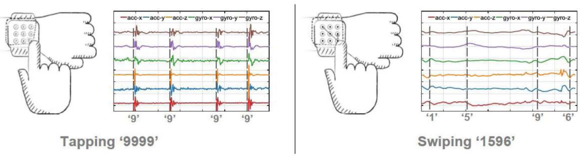


Fig. 3. The snoopy example of sniffing smartwatch passwords via censoring motion sensor data [5].

module, we identify the types of sensors that are leaking information.

To identify which sensors exist on a specific Android device, we first get a reference to the sensor service by creating an instance of the `SensorManager` class via calling the `getSystemService()` method with *SENSOR_SERVICE* argument. After that, we can determine available sensors on the device by calling the `getSensorList()` method. The `getSensorList()` method returns a list of all available sensors on the device by specifying constant *TYPE_ALL* as the parameter. A list of all sensors from a given type can also be retrieved by replacing the parameter as the constants defined for corresponding sensor types, such as *TYPE_GYROSCOPE, TYPE_LINEAR_ACCELERATION*, etc. We can also determine whether a specific type of sensor exists by calling the `getDefaultSensor()` method with the target type constant (the same as the ones passed in to `getSensorList()` method). If a device has that type of sensor, it will return an object of that sensor. Otherwise, null will be returned.

We use a rule-based strategy to identify the sensor type of a leak in the case of only one sensor registered in the given app. To do this, SEEKER obtains the sensor type by looking into the type constant in the `getDefaultSensor()` statement. For instance, `getDefaultSensor(Sensor.TYPE_ACCELEROMETER)` indicates that the Accelerometer sensor is obtained. We can then reasonably assume that all sensor-related data leaks in the class are associated with the identified sensor (because only this sensor is registered).

```
 1 public class MainActivity extends AppCompatActivity
       implements SensorEventListener{
 2 @Override
 3 public void onSensorChanged(SensorEvent sensorEvent) {
 4  switch(sensorEvent.sensor.getType()) {
 5   case Sensor.TYPE_ACCELEROMETER:
 6    accX = sensorEvent.values[0];
 7    accY = sensorEvent.values[1];
 8    accZ = sensorEvent.values[2];
 9    ...
10   case Sensor.TYPE_GYROSCOPE:
11    gyroX = sensorEvent.values[0] * 5;
12    gyroY = sensorEvent.values[1] * 5;
13    gyroZ = sensorEvent.values[2] * 5;
14    ...
15   case Sensor.TYPE_ROTATION_VECTOR:
16    rvX = sensorEvent.values[0];
17    rvY = sensorEvent.values[1];
18    rvZ = sensorEvent.values[2];
19    ...
20 }}}
```

Listing 2. An example of sensor type usage with switch branch.

In the case of multiple sensors registered in the given app, we further leverage context-aware static code analysis to find the connection between sensor types and the leaked field data. Firstly, we locate the invocation statement of API `android.hardware.SensorManager#getDefaultSensor(int)` in the `onSensorChanged()` method. In the multiple sensors scenario, different sensor's behavior is handled in a conditional branch (e.g. if-then-else statement or switch statement). We then apply context-aware static code analysis to detect the code branch that contains the taint sensor source statement, based on which we then resolve the sensor type in the branch condition.

We further elaborate on the context-aware static code analysis with an example presented in Listing 2. The code snippet in the Listing shows an example of how multiple sensors are handled with `onSensorChanged(android.hardware.SensorEvent)` method. Android determines the activated sensor by matching the `sensorEvent.sensor.getType()` method (line 4). For example, if `getType()` returns `Sensor.TYPE_ACCELEROMETER` (line

5), the data obtained by `sensorEvent.values` is associated with the Accelerometer sensor (lines 6-8); if `getType()` returns `Sensor.TYPE_GYROSCOPE` (line 10), the data contained in `sensorEvent.values` is accordingly associated with the current activated sensor, i.e., Gyroscope (lines 11-13).

IV. Experimental Setup and Results

SEEKER is designed to expose the data leak issues of sensors in Android apps. We investigate the feasibility and effectiveness of detecting sensor leaks in Android apps with the following three research questions:

- **RQ1:** *Can* SEEKER *effectively detect sensor leaks in Android apps?* This research question aims to investigate the feasibility of detecting sensor leaks in Android apps with SEEKER.
- **RQ2:** *To what extent diverse sensor leaks can be identified by* SEEKER*?* With this research question, we explore the sensor types related to the identified sensitive data leaks, and investigate to what extent such sensor leaks are targeted by attackers.
- **RQ3:** *Is* SEEKER *efficient to detect the sensor leaks in Android apps?* In this study, we leverage the time costs of detecting sensor leaks to assess the efficiency of SEEKER.

A. Experimental Setup

To answer the aforementioned research questions, we build the experimental dataset with a *malware* set and a *benign* set. The *malware* set contains 20,000 Android apps including malware downloaded from VirusShare repository [35] that were collected between 2012 and 2020. The 20,000 Android apps in *benign* set are crawled from the official Google Play store. All of the 40,000 apps are submitted to VirusTotal [36], the online scan engines aggregating over 70 anti-virus scanners (including the famous Kaspersky, McAfee, Kingsoft anti-virus engines), to check whether they contains viruses or not. For the *malware* set, we select the malware Android apps that have been labeled by at least five anti-virus engines to ensure their maliciousness, while for the *benign* set, the Android apps that are not tagged by any anti-virus engines are selected. SEEKER is designed to detect sensor leaks, thus we filter out the Android apps without any onboard sensors by checking whether their code contains the string "`android.hardware.sensor`". The final experimental dataset used in this study consists of 6,724 malware apps and 12,939 benign apps (cf. the 3rd column of Table IV). Our experiment runs on a Linux server with Intel(R) Core(TM) i9-9920X CPU @ 3.50GHz and 128GB RAM. The timeout setting for analyzing each app with SEEKER is set 20 minutes.

B. RQ1 – Feasibility of Detecting Sensor Data Leaks

Our first research question evaluates the feasibility of SEEKER on detecting sensor leaks in Android apps, of which results are illustrated in Table IV. For the quantitative aspect, 9,905 potential sensor leaks are identified by SEEKER in 1,596 apps. On average, one Android app could be injected with six sensor leaks. It indicates that the sensor leaks could exist in Android apps which might have been overlooked by the security analysts of Android apps. From the malicious aspect, 14.4% (967 out of 6,724) malware apps are identified with sensor leaks, while 4.9% (629 out of 12,939) benign apps are identified with such leaks. Figure 4 further presents the number of sensor leaks detected in each Android app, which shows that each malware app could be identified with more sensor leaks than the benign one. It is significantly confirmed by the Mann-Whitney-Wilcoxon (MWW) test [37], of which the resulting *p-value* is less than $\alpha = 0.001$. All of these results imply that malware apps have a higher possibility of containing sensor leaks than benign apps.

TABLE IV
Experimental results of the detected sensor leaks.

Dataset	# apps	# selected apps	# apps identified with sensor leaks	# identified sensor leaks
Malware	20,000	6,724	967	6,103
Benign	20,000	12,939	629	3,802
Total	40,000	19,663	1,596	9,905

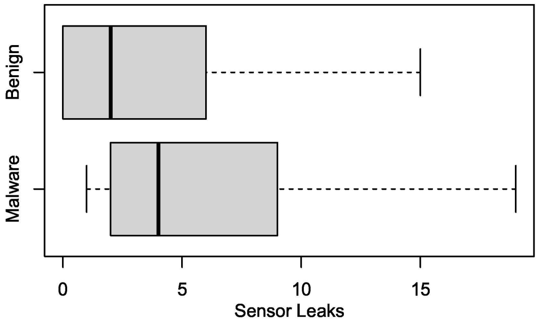


Fig. 4. Distribution of sensor leaks in each app.

Note that: there is lack of the ground-truth dataset about the sensor data leaks in Android apps. To address this limitation, we consider a sensor leak existing in an Android app if there is the data flow interaction between sensor-related sources (i.e., class fields or methods) and sinks. With this criterion, we manually checked the 229 sensor leaks detected by SEEKER in 20 randomly selected apps (10 malware apps and 10 benign apps). There are only 4 false-positive identified sensor leaks among the 229 identified sensor leaks in the 20 Android apps, which are caused by inaccurate data-flow analysis results of FlowDroid (we detail this limitation cased by FlowDroid in Section V-B). Such results show that SEEKER is capable of identifying the sensor leaks in Android apps. Simultaneously, it raises a major alarm for security analysts to pay attention to sensor leaks in Android apps that are not protected by the Android permission mechanism.

RQ1 ☞ Feasibility and Effectiveness

SEEKER is capable of automatically detecting sensor leaks in Android apps. Malware apps present higher possibility of committing sensor leaks than benign apps, and the sensor leaks might be ignored by security analysts.

C. RQ2 – Characterization of Sensor Leaks

Sources Triggering Sensor Leaks: The data leaks in sensor of Android apps are mainly triggered by two kinds of source: field and method (cf. Section III-A). As presented in Table V, ∼80% (= 7941/9905) of identified sensor leaks are triggered by the method sources. For the benign Android apps, ∼85.8% sensor data leaks are sourced from methods, while in the malware Android apps, ∼76.6% leaks are originated from methods. Table VI lists the top-10 most frequent sources triggering sensor leaks identified by SEEKER, which are 8 `getter` methods and 2 public fields from `Sensor`-related classes. We observe that the most frequent leaking source is the method `android.hardware.SensorManager#getDefaultSensor(int)` that is used to get the specific sensor of a given type, which is followed by the field `values` of class `SensorEvent`. The leaking source `android.hardware.SensorManager#getDefaultSensor(int)` occupies ∼89.1% (7074 out of 7941) of the method-triggered sensor leaks, and the field `SensorEvent#values` occupies ∼95.6% (1877 out of 1964) of the field-triggered sensor leaks. The sensor leaks triggered by the two sources occupy ∼90% of all identified sensor leaks.

TABLE V
NUMBER OF IDENTIFIED METHOD/FIELD-TRIGGERED SENSOR LEAKS.

	# identified method leaks	# identified field leaks
Malware	4,677	1,426
Benign	3,264	538
Total	7,941	1,964

TABLE VI
TOP-10 FREQUENT LEAKING SOURCES.

Sensor Sources	Malware	Benign	Total
SensorManager#getDefaultSensor(int)	4,326	2,748	7,074
SensorEvent#values	1,358	519	1,877
SensorManager#getSensorList(int)	114	121	235
Sensor#getType()	123	6	129
Sensor#getName()	29	82	111
Sensor#getMaximumRange()	19	73	92
SensorEvent#timestamp	68	19	87
Sensor#getVendor()	12	74	86
Sensor#getVersion()	11	69	80
Sensor#getResolution()	13	64	77

Sensor Types of Field-triggered Sensor Leaks: The sensor type is essential for deepening the understanding of sensor data leaks, i.e., knowing from which sensor the data is originally collected, as by default, this information is not given in field-triggered sensor leaks (e.g., sourced from the field variable `values` in class `SensorEvent`). The last module of SEEKER is hence dedicated to infer the sensor types of such leaks. Overall, in the 1,964 identified field-triggered sensor leaks, SEEKER successfully infers the corresponding sensor types for 1,923 (97.9%) of them. After manually checking the unsuccessful cases, we find that the 41 failed cases are mainly caused by the mistaken usage of sensors which can cause the sensors unexpected functional behavior, such as lacking sensor register information. This high success rate demonstrates the effectiveness of SEEKER in pinpointing the sensor types associated with sensor data leaks.

TABLE VII
TOP-10 FREQUENT SENSOR TYPES OF FIELD-TRIGGERED SENSOR LEAKS.

Sensor Type	Malware	Goodware	Total
ACCELEROMETER	1,068	304	1,372
MAGNETIC_FIELD	131	50	181
ORIENTATION	92	84	176
PROXIMITY	12	32	44
LINEAR_ACCELERATION	40	4	44
STEP_COUNTER	14	9	23
TEMPERATURE	12	8	20
GYROSCOPE	7	8	15
PRESSURE	1	13	14
LIGHT	6	5	11

We further investigate the true-positive rate of the successfully inferred sensor types. Due to the lack of the ground-truth dataset of related sensor types for sensor leaks, we resort to a manual inspection on the source code of 20 randomly selected apps (10 malware apps and 10 benign apps), each of which is identified with at least one field-triggered leak (86 field-triggered sensor leaks in total). All leaks are confirmed with true-positive inferred sensor types, which implies that SEEKER is effective in inferring the sensor types of field-triggered leaks.

Table VII presents the top 10 leaking sensor types of the identified field-triggered sensor leaks. The type "Accelerometer" is the sensor type of 74.9% and 56.5% of identified field-triggered sensor leaks in the *malware* apps and the *benign* apps, respectively. Android apps widely use the Accelerometer to monitor device motion states by measuring the acceleration applied to a device on three physical axes (i.e., x, y, and z axes). The motion data captured by the Accelerometer can be further processed or analyzed. For example, *Smart-Its Friends* [38] pairs two devices by acquiring Accelerometer data in a shared wireless medium. Pirttikangas et al. [39] reported that the Accelerometer in smartphones can be used to track the accurate activity of users, such as brushing teeth and sitting while reading newspapers. Such information can also be utilized to steal the PIN of a device through side-channel attacks (such as [5] and [40]).

Apart from the Accelerometer, the other frequent sensor types of field-triggered sensor leaks include MAGNETIC_FIELD, ORIENTATION, PROXIMITY, LINEAR_ACCELERATION, STEP_COUNTER, TEMPERATURE, GYROSCOPE, PRESSURE and LIGHT. These sensors are also likely to be used to harm the user's privacy. Biedermann et al. [41] stated that the magnetic field sensor can be exploited to detect what type of operating system is booting up and what application is being started. The orientation sensor can wiretap the device's orientation without requesting any permission, which can be used by attackers to infer the user's PIN. The proximity sensor data can be a trigger to automatically start a phone call recording when users hold the smartphone against their face to make a call. The individual step details can be stored by collecting data from the step counter sensor when the app runs in the background. Temperature, pressure and light sensors are also widely used in IoT devices to monitor environmental conditions, while the gyroscope sensor is utilized to verify the user's identity [31].

Case Study: Here we show two real-world apps that leaks the sensor data, which could be leveraged by attacker to achieve malicious goals.

```
1 final class a.b.b implements SensorEventListener{
2  public void onSensorChanged(SensorEvent var1){
3      float var5 = var1.values[0];
4      float var6 = var1.values[1];
5      float var7 = var1.values[2];
6      Log.v("WindowOrientationListenerN3V", "Raw
        acceleration vector: x=" + var5 + ", y=" var6 + ",
        z=" + var7);
7 }}
```

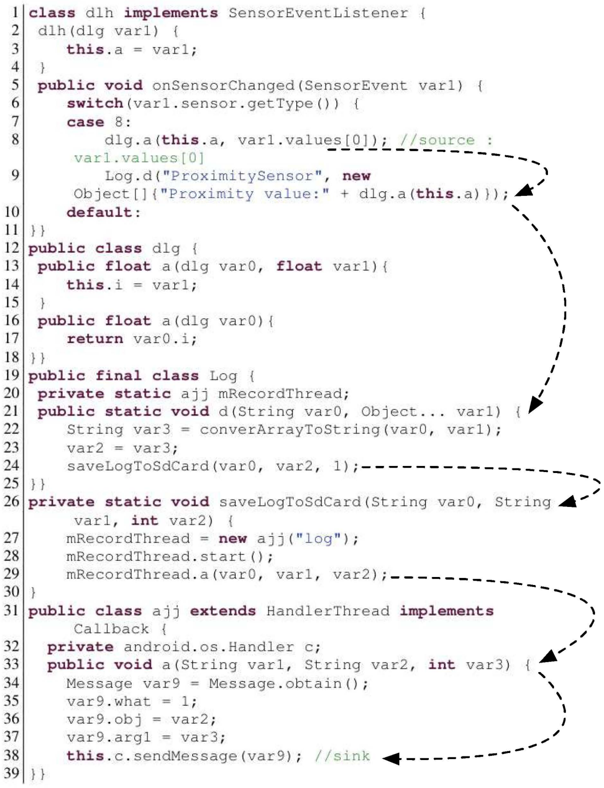
Listing 3. Example of a sensor leak in *com.n3vgames.android.driver*.

Listing 3 showcases a typical sensor leak case in real-world apps. The code snippet is excerpted from a malicious app *com.n3vgames.android.driver*. This app collects raw accelerometer data from the class field `SensorEven#values[]` (lines 3-5), and then leak them through invoking *android.util.Log* API (line 6). The app is flagged as a Trojan that downloads additional executable content from a remote server. While leaking such information may not direct link to its malicious behaviour, it expands the attack surface to the attackers. For example, the sensor information can be used to predict the device's motion state, which may lead to a stealthier attack (e.g., downloading malicious content when the device is not in use). It is worth noting that Zhang et al. [42] have demonstrated the possibility of using the sensor information to launch stealthy attack for taking control of an Android phone via Google's voice assistant.

Figure 5 shows another example derived from a phone book app *com.tencent.pb*. It collects the Proximity sensor data (line 8) and eventually send it out through `sendMessage` method (line 41) in a asynchronous thread. The Proximity sensor data was passed as a parameter of method `dlg.a(dlg, float)` (line 8), then the data was stored in the class field `i` of object `dlg`. The data flows through the method *Log.d(String, Object...)* (line 9), which obtains the field variable `i` of object `dlg` as the second parameter via `dlg.a(dlg)` (line 23). Finally, the tainted parameter passed on to the method `Log.saveLogToSdCard(String, String, int)`, which creates a new thread (line 29-33) and send the sensor data out (line 35-42).

RQ2 ☞ Characterizing Sensor Leaks

SEEKER is capable of inferring sensor types and pinpointing the corresponding source sensors for data leaks. Our results show that the Accelerometer leaks the most sensor data, both in malware samples and benign apps. The most leaking sources are the method *SensorManager#getDefaultSensor(int)* and the field *SensorEvent#values*, the latter of which has been frequently leveraged in malicious behaviours such as inferring user's PIN.



```
1  class dlh implements SensorEventListener {
2   dlh(dlg var1) {
3      this.a = var1;
4   }
5   public void onSensorChanged(SensorEvent var1) {
6      switch(var1.sensor.getType()) {
7      case 8:
8          dlg.a(this.a, var1.values[0]); //source :
        var1.values[0]
9          Log.d("ProximitySensor", new
        Object[]{"Proximity value:" + dlg.a(this.a)});
10     default:
11 }}
12 public class dlg {
13  public float a(dlg var0, float var1){
14     this.i = var1;
15  }
16  public float a(dlg var0){
17     return var0.i;
18 }}
19 public final class Log {
20  private static ajj mRecordThread;
21  public static void d(String var0, Object... var1) {
22     String var3 = converArrayToString(var0, var1);
23     var2 = var3;
24     saveLogToSdCard(var0, var2, 1);
25 }}
26 private static void saveLogToSdCard(String var0, String
        var1, int var2) {
27     mRecordThread = new ajj("log");
28     mRecordThread.start();
29     mRecordThread.a(var0, var1, var2);
30 }
31 public class ajj extends HandlerThread implements
        Callback {
32   private android.os.Handler c;
33   public void a(String var1, String var2, int var3) {
34     Message var9 = Message.obtain();
35     var9.what = 1;
36     var9.obj = var2;
37     var9.arg1 = var3;
38     this.c.sendMessage(var9); //sink
39 }}
```

Fig. 5. Code snippet of sensor value leak excerpted from *com.tencent.pb*.

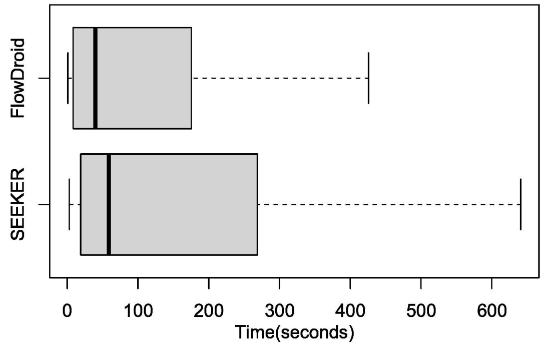


Fig. 6. Distribution of time Performance spent to analyze an app by FlowDroid and SEEKER, respectively.

D. RQ3 – Runtime Overhead

SEEKER extends FlowDroid to detect sensor-related data leaks and for inferring the sensor types involved in the leak. We evaluate the runtime overhead of SEEKER and compare it with the original FlowDroid. Figure 6 shows the time consumed by FlowDroid and SEEKER, respectively. On average, it takes 177.09 seconds for SEEKER to process an app in our dataset, which is comparable to that of the original FlowDroid (i.e., on average, 132.74 seconds to process an app). As experimentally demonstrated by Avdiienko et al. [43], by increasing the capacity of the execution server, the performance of FlowDroid could be further improved. This improvement should also be applicable to SEEKER, making it also possible to analyze real-world apps in practice. The fact that the time difference between SEEKER and FlowDroid is relatively small suggests that it is also capable of applying SEEKER to analyze (in parallel) large-scale Android apps, as what has been experimentally demonstrated to be true for FlowDroid.

RQ3 ☞ Efficiency

The time consumption of SEEKER is acceptable for real-time sensor leak detection, with on average 177.09 seconds for one app without a high increase when comparing with FlowDroid, which is suitable for real-time app analysis.

V. DISCUSSION

We now discuss the potential implications and limitations of this work.

A. Implications

Beyond smartphone apps. The motivating example presented in Section II-B is extracted from an attack targeting smartwatches, which also supply sensors to support client apps to implement advanced features. These sensors could be abused by smartwatch app developers, especially malicious attackers. We argue there is also a strong need to characterize sensor leaks in smartwatch apps, not just smart phones. Our preliminary experiment has shown that SEEKER can be directly applied to correctly pinpoint the sensor leaks in the Android-based smartwatch apps that sniff passwords [44].

Android has been used on more and more devices, such as TVs, home appliances, fitness machines and cars. The apps in these devices could also all be compromised to leak end-users' sensitive data and hence should also be carefully analyzed before releasing them to the public. SEEKER could also be useful to characterize data leaks for such Android devices and we will examine some of these in our future work.

Beyond sensor leaks. As argued by Zhang et al. [45], tainted values of string type could be organized as fields in objects, which cannot be detected by state-of-the-art static taint analysis tools such as FlowDroid. This is because FlowDroid only supports methods as sources. Thus, sensitive field sources are overlooked by FlowDroid, giving rise to many false negatives.

Our SEEKER extends FlowDroid to mitigate this research gap by introducing field-triggered static taint analysis. It is worth highlighting that SEEKER is capable of not only detecting sensor leaks but also pinpointing general privacy leaks, either triggered by source methods or fields. To help users experience this feature, we have committed a pull request to the original FlowDroid on GitHub so that users can easily access Field-triggered Static Taint Analysis by simply configuring their interested field sources in *SourcesAndSinks.txt* file.

Automated approaches for discovering sensitive source fields. In this work, the sensor-related sensitive source fields are identified through manual effort. These are well known to be time-intensive and error-prone. Hence, our current SEEKER approach is not directly applicable for detecting general field-triggered privacy leaks. To achieve this, we need to go through all the fields defined in the Android framework to identify sensitive ones. This is non-trivial as Android is now one of the largest community software projects and contains nearly 10K classes. There is a need to invent new automated approaches to discover sensitive source fields. One possible solution would be to extend the machine learning approach applied in the SUSI tool to support the prediction of sensitive source fields.

B. Limitations of SEEKER

Limitation of static analysis. One major limitation of our tool lies in the intrinsic vulnerability of static code analysis when encountering code obfuscation, reflection, native code, etc. These lead to the unsoundness of our approach. However, these challenges are regarded as well known and non-trivial issues to overcome in our research community. In our future work, we want to integrate other useful tools developed by our fellow researchers to overcome these shortcomings. For example, we plan to leverage DrodRA [46], [47] to reduce the impact of reflective calls on our static analysis approach.

As explained in Section III-C, our sensor type inference approach can not trace the sensor type in method-triggered leaks when multiple sensors are available on a device. This is because the actual calling object of a method can only be obtained at run-time. We plan to overcome this limitation in our future work by incorporating dynamic analysis approaches to obtain the required run-time values.

Limitations inherited from FlowDroid. Since our SEEKER approach directly extends FlowDroid to detect sensor-triggered privacy leaks, it also has all of the limitations of FlowDroid. For example, FlowDroid may yield unsound results because it may have overlooked certain callback methods involved in the Android lifecycle or incorrectly modelled native methods accessed by the app. FlowDroid is also oblivious to multi-threading and it assumes threads to execute in an arbitrary but sequential order, which may also lead to false results.

Limitations inherited from SUSI. The sensor-related sensitive source methods are collected based on the results of the state-of-the-art tool SUSI. This is also the tool leveraged by the FlowDroid to identify source and sink methods. However, the results of SUSI may not be completely correct – some of its identified sources may not be truly sensitive. However, this threat has no impact on our approach but only on our experimental results. This limitation could be mitigated if a better set of source and sink methods are configured.

Threats to Validity. Apart from these technical limitations, our work also involves some manual efforts. For example, the sensor-related sensitive source fields are summarized manually by reading the Android developers' documentation. Such manual processes may also introduce errors of their own. To mitigate this threat, the authors of this paper have cross-validated the results, and we release our tool[2] and dataset[3] for public access.

[2] https://github.com/MobileSE/SEEKER

[3] https://zenodo.org/record/4764311#.YJ91jJMzadZ

VI. RELATED WORK

Android sensor usage. Android sensor usage has long been analyzed in software security mechanisms. Related works [48], [49], [3], [50], [51], [2], [12], [9], [52] have indicated that embedded sensors can be intentionally misused by malicious apps for privacy compromise. Ba et al. [49] proposed a side-channel attack that adopts accelerometer data to eavesdrop on the speaker in smartphones. Xu et al. [3] have shown that it is feasible to infer user's tap inputs using its integrated motion sensors. Liang Cai et al.[12] revealed that confidential data could be leaked when motion sensors, such as accelerometers and gyroscopes, are used to infer keystrokes. Also, Lin et al.[53] demonstrated that the orientation sensor of the smartphone could be utilized to detect users' unique gesture to hold and operate their smartphones.

Android Sensor misuse is one of the major causes of privacy leaks and security issues on the Android platform. Zhu et al. [48] collected sensor data from accelerometers, gyroscopes and magnetometers and constructs users' gesture based on these data. Their work indicates that it is feasible to get access to sensory data for personalized usage. Liu et al. [51] demonstrated the most frequently used sensors in Android devices and revealed their usage patterns through backward tracking analysis. They further investigate sensor data propagation path for accurately characterizing the sensor usage [32]. Their findings suggest that the accelerometer is the most frequently used sensor and the sensor data are always used in local codes.

Software side-channels attacks. Many previous studies [54], [55], [56], [57], [58], [4] explored password inference through specific sensors on smartphones. Owusu et al. [9] showed that accelerometer values could be used as a powerful side channel to figure out the password on a touchscreen keyboard. Cai et al.[12] provided insights of how motion sensors, such as accelerometers and gyroscopes, can be used to infer keystrokes. Cai et al. [59] found that mobile phone sensors are inadequately protected by permission system so that it can raise serious privacy concerns. Enck et al. [60] developed TaintDroid that takes sensor information (i.e., location and accelerometer) as sources to detect privacy leaks. Mehrnezhad et al. [61] show that orientation sensor can be stealthily listened to without requesting any permission, contributing for attackers to infer the user's PIN. However, these works emphasize the challenges facing the detection of sensor-sniffing apps or only provided specific attacks by using sensor data. None of them can systematically characterize data leaks in all kinds of sensors.

Static analysis on Android apps. Android users have long been suffered from privacy leaks [62], [63], [64], [65]. Several solutions have been proposed for detecting such data leaks through static taint analysis [66], [67], [68]. For example, Arzt et al. [34] developed FlowDroid, a context, flow, field and object-sensitive static analysis tool for detecting potential data leaks in Android Apps. Based on Soot [69], FlowDroid relies on pre-defined knowledge to pinpoint taint flows between source and sink APIs. Zhang et al. [45] developed ConDySTA, a dynamic taint analysis approach, as a supplement to static taint analysis by introducing inaccessible code and sources that help reduce false negatives. Further, Li et al. [70] presented IccTA, which can precisely perform data-flow analysis across multiple components for Android apps. Klieber et al. [71] augment the FlowDroid and Epicc[72] analyses by tracking both inter-component and intra-component data flow in Android apps. However, none of these tools concerns the leaks that originated from sensors. Apart from that, our tool only takes the sensor-related code into account, which cost less time by pruning the control flow graph.

The most similar work to ours is SDFDroid[32], which provides the sensor usage patterns through data flow analysis. As a static approach, however, it focuses on different research object compared to SEEKER. For example, SDFDroid reveals sensor usage patterns while our work explores how and where the sensor data are leaked. On the other hand, SDFDroid applies a static approach to extract sensor data propagation path to construct sensor usage patterns through clustering analysis. In contrast to SDFDroid, SEEKER provide detailed privacy leaks caused by misuse of sensor data, which haven't been found by SDFDroid.

VII. CONCLUSION

We have presented a novel tool, SEEKER, for characterizing sensor leaks in Android apps. Our experimental results on a large scale of real-world Android apps indicate that our tool is effective in identifying all types of potential sensor leaks in Android apps. Our tool is not only capable of detecting sensor leaks, but also pinpointing general privacy leaks that are triggered by class fields. Although there are related works on sensor usage analysis, to the best of our knowledge, there is no other work that thoroughly analyses Android sensor leakage. Unlike previous works, our tool is the first one to characterize all kinds of sensor leaks in Android apps. We extend FlowDroid for supporting field sources detection (i.e., merged to FlowDroid via pull #385 on Github[73]), which we believe could be adapted to analyze other sensitive field-triggered leaks. To benefit our fellow researchers and practitioners towards achieving this, we have made our approach open source at the following Github site.

https://github.com/MobileSE/SEEKER

ACKNOWLEDGEMENTS

This work was supported by the Australian Research Council (ARC) under a Laureate Fellowship project FL190100035, a Discovery Early Career Researcher Award (DECRA) project DE200100016, and a Discovery project DP200100020. This research was also supported by the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2020A06) and the Open Project Program of the Key Laboratory of Safety-Critical Software (NUAA), Ministry of Industry and Information Technology (No. XCA20026).

References

[1] Google, "Sensors Overview," https://developer.android.com/guide/topics/sensors/sensors_overview.html, online; accessed 12 March 2021.

[2] 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*, 2012, pp. 41–50.

[3] 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*, 2012, pp. 113–124.

[4] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones." in *NDSS*, vol. 11, 2011, pp. 17–33.

[5] C. X. Lu, B. Du, H. Wen, S. Wang, A. Markham, I. Martinovic, Y. Shen, and N. Trigoni, "Snoopy: Sniffing your smartwatch passwords via deep sequence learning," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 4, pp. 1–29, 2018.

[6] D. Hodges and O. Buckley, "Reconstructing what you said: Text inference using smartphone motion," *IEEE Transactions on Mobile Computing*, vol. 18, no. 4, pp. 947–959, 2018.

[7] L. Cai and H. Chen, "On the practicality of motion based keystroke inference attack," in *International Conference on Trust and Trustworthy Computing*. Springer, 2012, pp. 273–290.

[8] A. Al-Haiqi, M. Ismail, and R. Nordin, "Keystrokes inference attack on android: A comparative evaluation of sensors and their fusion." *Journal of ICT Research & Applications*, vol. 7, no. 2, 2013.

[9] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang, "Accessory: password inference using accelerometers on smartphones," in *proceedings of the twelfth workshop on mobile computing systems & applications*, 2012, pp. 1–6.

[10] P. Marquardt, A. Verma, H. Carter, and P. Traynor, "(sp) iphone: Decoding vibrations from nearby keyboards using mobile phone accelerometers," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 551–562.

[11] S. Narain, A. Sanatinia, and G. Noubir, "Single-stroke language-agnostic keylogging using stereo-microphones and domain specific machine learning," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, 2014, pp. 201–212.

[12] L. Cai and H. Chen, "Touchlogger: Inferring keystrokes on touch screen from smartphone motion." *HotSec*, vol. 11, no. 2011, p. 9, 2011.

[13] L. Bo, L. Fengjun, W. Guanghui *et al.*, "I know what you type on your phone: Keystroke inference on android device using deep learning," Ph.D. dissertation, University of Kansas, 2019.

[14] Z. Ling, Z. Li, C. Chen, J. Luo, W. Yu, and X. Fu, "I know what you enter on gear vr," in *2019 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2019, pp. 241–249.

[15] Y. Huang, X. Guan, H. Chen, Y. Liang, S. Yuan, and T. Ohtsuki, "Risk assessment of private information inference for motion sensor embedded iot devices," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 4, no. 3, pp. 265–275, 2019.

[16] T. Nguyen, "Using unrestricted mobile sensors to infer tapped and traced user inputs," in *2015 12th International Conference on Information Technology-New Generations*. IEEE, 2015, pp. 151–156.

[17] J. Lin and J. Seibel, "Motion-based side-channel attack on mobile keystrokes," 2019.

[18] X. Liu, Z. Zhou, W. Diao, Z. Li, and K. Zhang, "When good becomes evil: Keystroke inference with smartwatch," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1273–1285.

[19] H. Wang, T. T.-T. Lai, and R. Roy Choudhury, "Mole: Motion leaks through smartwatch sensors," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, 2015, pp. 155–166.

[20] A. Maiti, M. Jadliwala, J. He, and I. Bilogrevic, "Side-channel inference attacks on mobile keypads using smartwatches," *IEEE Transactions on Mobile Computing*, vol. 17, no. 9, pp. 2180–2194, 2018.

[21] ——, "(smart) watch your taps: Side-channel keystroke inference attacks using smartwatches," in *Proceedings of the 2015 ACM International Symposium on Wearable Computers*, 2015, pp. 27–30.

[22] A. Sarkisyan, R. Debbiny, and A. Nahapetian, "Wristsnoop: Smartphone pins prediction using smartwatch motion sensors," in *2015 IEEE international workshop on information forensics and security (WIFS)*. IEEE, 2015, pp. 1–6.

[23] A. Maiti, R. Heard, M. Sabra, and M. Jadliwala, "Towards inferring mechanical lock combinations using wrist-wearables as a side-channel," in *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2018, pp. 111–122.

[24] Y. Liu and Z. Li, "aleak: Context-free side-channel from your smart watch leaks your typing privacy," *IEEE Transactions on Mobile Computing*, vol. 19, no. 8, pp. 1775–1788, 2019.

[25] C. Shen, S. Pei, Z. Yang, and X. Guan, "Input extraction via motion-sensor behavior analysis on smartphones," *Computers & Security*, vol. 53, pp. 143–155, 2015.

[26] Y. Michalevsky, D. Boneh, and G. Nakibly, "Gyrophone: Recognizing speech from gyroscope signals," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 1053–1067.

[27] M. Vuagnoux and S. Pasini, "Compromising electromagnetic emanations of wired and wireless keyboards." in *USENIX security symposium*, 2009, pp. 1–16.

[28] C. Song, F. Lin, Z. Ba, K. Ren, C. Zhou, and W. Xu, "My smartphone knows what you print: Exploring smartphone-based side-channel attacks against 3d printers," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 895–907.

[29] K. Block and G. Noubir, "My magnetometer is telling you where i've been? a mobile device permissionless location attack," in *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2018, pp. 260–270.

[30] S. Chakraborty, W. Ouyang, and M. Srivastava, "Lightspy: Optical eavesdropping on displays using light sensors on mobile devices," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 2980–2989.

[31] A. K. Sikder, G. Petracca, H. Aksu, T. Jaeger, and A. S. Uluagac, "A survey on sensor-based threats and attacks to smart devices and applications," *IEEE Communications Surveys & Tutorials*, 2021.

[32] X. Liu, J. Liu, W. Wang, Y. He, and X. Zhang, "Discovering and understanding android sensor usage behaviors with data flow analysis," *World Wide Web*, vol. 21, no. 1, pp. 105–126, 2018.

[33] S. Arzt, S. Rasthofer, and E. Bodden, "Susi: A tool for the fully automated classification and categorization of android sources and sinks," *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013.

[34] 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," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[35] *Virusshare*, 2021. [Online]. Available: http://virusshare.com/

[36] *VirusTotal*, 2021. [Online]. Available: https://www.virustotal.com/

[37] M. P. Fay and M. A. Proschan, "Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules," *Statistics surveys*, vol. 4, p. 1, 2010.

[38] L. E. Holmquist, F. Mattern, B. Schiele, P. Alahuhta, M. Beigl, and H.-W. Gellersen, "Smart-its friends: A technique for users to easily establish connections between smart artefacts," in *international conference on Ubiquitous Computing*. Springer, 2001, pp. 116–122.

[39] S. Pirttikangas, K. Fujinami, and T. Nakajima, "Feature selection and activity recognition from wearable sensors," in *International symposium on ubiquitious computing systems*. Springer, 2006, pp. 516–527.

[40] T. Giallanza, T. Siems, E. Smith, E. Gabrielsen, I. Johnson, M. A. Thornton, and E. C. Larson, "Keyboard snooping from mobile phone arrays with mixed convolutional and recurrent neural networks," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 3, no. 2, pp. 1–22, 2019.

[41] S. Biedermann, S. Katzenbeisser, and J. Szefer, "Hard drive side-channel attacks using smartphone magnetic field sensors," in *International Conference on Financial Cryptography and Data Security*. Springer, 2015, pp. 489–496.

[42] R. Zhang, X. Chen, S. Wen, X. Zheng, and Y. Ding, "Using ai to attack va: a stealthy spyware against voice assistances in smart phones," *IEEE Access*, vol. 7, pp. 153 542–153 554, 2019.

[43] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 426–436.

[44] X. Chen, W. Chen, K. Liu, C. Chen, and L. Li, "A comparative study of smartphone and smartwatch apps," in *The 2021 ACM/SIGAPP Symposium on Applied Computing (SAC 2021)*, 2021.

[45] X. Zhang, X. Wang, R. Slavin, and J. Niu, "Condysta: Context-aware dynamic supplement to static taint analysis."

[46] X. Sun, L. Li, T. F. Bissyandé, J. Klein, D. Octeau, and J. Grundy, "Taming reflection: An essential step toward whole-program analysis of android apps," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–36, 2021.

[47] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 318–329.

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

[49] Z. Ba, T. Zheng, X. Zhang, Z. Qin, B. Li, X. Liu, and K. Ren, "Learning-based practical smartphone eavesdropping with built-in accelerometer," in *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium*, 2020, pp. 23–26.

[50] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury, "Tapprints: your finger taps have fingerprints," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, 2012, pp. 323–336.

[51] X. Liu, J. Liu, and W. Wang, "Exploring sensor usage behaviors of android applications based on data flow analysis," in *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2015, pp. 1–8.

[52] W.-H. Lee and R. B. Lee, "Multi-sensor authentication to improve smartphone security," in *2015 International conference on information systems security and privacy (ICISSP)*. IEEE, 2015, pp. 1–11.

[53] C.-C. Lin, C.-C. Chang, D. Liang, and C.-H. Yang, "A new non-intrusive authentication method based on the orientation sensor for smartphone users," in *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 2012, pp. 245–252.

[54] K.-h. Chang, J. Hightower, and B. Kveton, "Inferring identity using accelerometers in television remote controls," in *International Conference on Pervasive Computing*. Springer, 2009, pp. 151–167.

[55] J. Lester, B. Hannaford, and G. Borriello, ""are you with me?"–using accelerometers to determine if two devices are carried by the same person," in *International Conference on Pervasive Computing*. Springer, 2004, pp. 33–50.

[56] J. Liu, L. Zhong, J. Wickramasuriya, and V. Vasudevan, "uwave: Accelerometer-based personalized gesture recognition and its applications," *Pervasive and Mobile Computing*, vol. 5, no. 6, pp. 657–675, 2009.

[57] N. Ravi, N. Dandekar, P. Mysore, and M. L. Littman, "Activity recognition from accelerometer data," in *Aaai*, vol. 5, no. 2005. Pittsburgh, PA, 2005, pp. 1541–1546.

[58] F. R. Allen, E. Ambikairajah, N. H. Lovell, and B. G. Celler, "Classification of a known sequence of motions and postures from accelerometry data using adapted gaussian mixture models," *Physiological measurement*, vol. 27, no. 10, p. 935, 2006.

[59] L. Cai, S. Machiraju, and H. Chen, "Defending against sensor-sniffing attacks on mobile phones," in *Proceedings of the 1st ACM workshop on Networking, systems, and applications for mobile handhelds*, 2009, pp. 31–36.

[60] W. Enck, P. Gilbert, S. Han, V. Tendulkar, 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," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 1–29, 2014.

[61] M. Mehrnezhad, E. Toreini, S. F. Shahandashti, and F. Hao, "Stealing pins via mobile sensors: actual risk versus user perception," *International Journal of Information Security*, vol. 17, no. 3, pp. 291–313, 2018.

[62] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. Le Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, 2017.

[63] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, 2018.

[64] J. Samhi, A. Bartel, T. F. Bissyandé, and J. Klein, "Raicc: Revealing atypical inter-component communication in android apps," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1398–1409.

[65] D. Octeau, S. Jha, M. Dering, P. Mcdaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon, "Combining static analysis with probabilistic models to enable market-scale android inter-component analysis," in *Proceedings of the 43th Symposium on Principles of Programming Languages (POPL 2016)*, 2016.

[66] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein, "Borrowing your enemy's arrows: the case of code reuse in android via direct inter-app code invocation," in *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.

[67] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. Le Traon, "ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis," in *Proceedings of the 30th IFIP International Conference on ICT Systems Security and Privacy Protection (SEC 2015)*, 2015.

[68] X. Yang, D. Lo, L. Li, X. Xia, T. F. Bissyandé, and J. Klein, "Characterizing malicious android apps by mining topic-specific data flow signatures," *Information and Software Technology*, 2017.

[69] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[70] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 280–291.

[71] 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*, 2014, pp. 1–6.

[72] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 543–558.

[73] Xiaoyu Sun, "Improve FlowDroid to support Field Sources(develop branch) ," https://github.com/secure-software-engineering/FlowDroid/pull/385, online; accessed 12 September 2021.