SENTINEL: Generating GUI Tests for Android Sensor Leaks

Haowei Wu
Ohio State University
wuhaow@cse.ohio-state.edu

Yan Wang
Ohio State University
wang10@cse.ohio-state.edu

Atanas Rountev
Ohio State University
rountev@cse.ohio-state.edu

ABSTRACT

Due to the widespread use of Android devices and apps, it is important to develop tools and techniques to improve app quality and performance. Our work focuses on a problem related to hardware sensors on Android devices: the failure to disable unneeded sensors, which leads to *sensor leaks* and thus battery drain. We propose the SENTINEL testing tool to uncover such leaks. The tool performs static analysis of app code and produces a model which maps GUI events to callback methods that affect sensor behavior. The model is traversed to identify paths that are likely to exhibit sensor leaks during run-time execution. The reported paths are then used to generate test cases. The execution of each test case tracks the run-time behavior of sensors and reports observed leaks. Our experimental results indicate that SENTINEL effectively detects sensor leaks, while focusing the testing efforts on a very small subset of possible GUI event sequences.

1 INTRODUCTION

There are more than 2 billion active Android devices and 3.5 million Android apps in the Google Play store, with many other app stores also becoming popular (e.g., in China). It is important to develop techniques and tools to improve app quality and performance. Complex event-driven behavior and limited device resources present challenges for developers. One challenge are various "leaking" behaviors that can lead to energy-related inefficiencies [3–5, 13–15, 22, 30].

The focus of our work is one instance of this problem: the leaking of *hardware sensors*. Sensors in Android devices can track changes in acceleration, rotation, proximity to screen, light, temperature, pressure, humidity, etc. However, the use of sensors creates opportunities for energy inefficiencies. As a general Android developer guideline, the app should always disable sensors that are not needed. Failing to disable unneeded sensors—that is, *sensor leaks*—can drain the battery. If possible, sensor leaks should be detected and eliminated before an app is released in an app store.

We propose a testing approach targeting sensor leaks. The approach was implemented in the SENTINEL tool for sensor testing to detect leaks. The tool takes as input an Android APK, performs static analysis of app code, and identifies sensor-related objects and API calls. The static analysis produces a model which maps GUI events to callback methods that affect sensor behavior. This model, referred to as the *sensor effects control-flow graph*, is traversed to identify paths that are likely to exhibit sensor leaks during run-time execution. The reported paths are then used to generate test cases. The execution of each test case tracks the run-time behavior of sensors and reports observed sensor leaks.

SENTINEL uncovers two categories of violations of Android guidelines for sensor management. The first category identifies leaking components that, during their lifetime, acquire a sensor but do not release it. The second category identifies components that acquire a sensor but do not release it when suspended for a long period of time. Our experimental results indicate that SENTINEL effectively detects sensor leaks, while focusing the testing efforts on a very small subset of possible GUI event sequences, as determined by our targeted sensor-aware static analysis of app code.

AST'18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5743-2/18/05…$15.00
https://doi.org/10.1145/3194733.3194734

2 ANDROID UIS AND SENSORS

In Android, the user interface (UI) thread is the main app thread. Various *windows* are displayed in the UI and *widgets* inside these windows can be the targets of UI events (e.g., "click"). These events could have several effects, including UI changes such as opening a new window. The main category of windows is *activities*, which are the core components of Android apps. Two other categories are *menus* and *dialogs*. We will discuss only activities and will use "window" and "activity" interchangeably; however, menus and dialogs are also handled by SENTINEL.

2.1 Running Example

Figure 1 shows a simplified example derived from a sensor leak uncovered by SENTINEL. Calculator Vault is a vault app used to hide photos and other documents. The app has over a million downloads in Google Play. The example shows two of the app's activities: `SettingActivity` and `UnlockActivity`. The first activity has a button widget (`btn` at line 4); the second one has a switch widget (`sc` at line 17) which is a toggle to select between two options.

An app user can trigger events on widgets; as a result, *event handling callback methods* are executed. For example, if `btn`'s button is touched, `onClick` (lines 7–13) is invoked by the Android platform code. In this example, using `startActivity` at line 12, the event handler opens a new window corresponding to `UnlockActivity`. The new window is pushed on top of a window stack, immediately above the window for `SettingActivity`. When eventually this new window is closed, it is popped from the stack and `SettingActivity` is redisplayed. As another example, when the state of switch `sc` changes, `onCheckedChanged` (lines 24–27) is invoked. As discussed later, this event handler registers a listener for the accelerometer sensor.

Upon a UI event, a new window could be opened and pushed on the window stack, or currently-alive window(s) could be closed and popped from the stack. These open/close effects could be due to (1) code inside callback methods, or (2) platform-defined default

```
1  class SettingActivity
2      extends Activity implements OnClickListener {
3    onCreate(...) {
4      Button btn = findViewbyId(R.id.rl_unlockSetting);
5      btn.setOnClickListener(this); ...
6    }
7    onClick(View v) {
8      switch(v.getId()) {
9        ...
10       case R.id.rl_unlockSetting:
11         Intent i = new Intent(UnlockActivity.class);
12         startActivity(i); break;}
13   }
14 }
15 class UnlockActivity extends Activity {
16   onCreate(...) {
17     SwitchCompat sc = ...;
18     SensorManager sm = ...;
19     Sensor accel = sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
20     SensorEventListener shakeListener = new SensorEventListener {
21       onSensorChanged(...) {
22         if (...) { sm.unregisterListener(this); }}};
23     sc.setOnCheckedChangeListener(new OnCheckedChangeListener{
24       public onCheckedChanged(View v) {
25         ...
26         sm.registerListener(shakeListener, accel);}});
27   }
28   onDestroy() { ... }
29 }
```

Figure 1: Example derived from Calculator Vault.

behavior for certain events. As a result, *lifecycle callback methods* could be invoked. Figure 1 shows lifecycle callbacks `onCreate` (in both activities) and `onDestroy` (in the second activity). There are additional lifecycle callbacks not shown in the figure. For example, when line 12 is executed, in the general case the sequence of invoked callbacks would be $\texttt{onPause}_1$, $\texttt{onCreate}_2$, $\texttt{onStart}_2$, $\texttt{onResume}_2$, $\texttt{onStop}_1$; here the subscript denotes the activity.

2.2 Sensors in Android Apps

There are multiple categories of sensors on an Android device. Each category is represented by an integer constant defined in class `android.hardware.Sensor`. For example, all accelerometer sensors are represented by `Sensor.TYPE_ACCELEROMETER`. A hardware sensor is represented by a sensor object, instantiated from `android.hardware.Sensor`. These sensor objects are created by the Android framework and will not be replaced or destroyed unless the app process is killed. From our case studies, we observed that developers rarely use more than one sensor from a sensor category: typically, only the default sensor is obtained, by calling `getDefaultSensor`. At line 19 in Figure 1, `accel` refers to the default accelerometer sensor object, which is used by the application to detect when the user shakes the device in order to unlock it.

To obtain sensor data, the programmer registers a *sensor event listener*. Such a listener is an instance of `SensorEventListener` (line 20). Callback `onSensorChanged` is invoked on this listener whenever new sensor data is available. Line 26 shows how a listener is registered with a sensor object. The sensor hardware will be enabled when there exists any listener registered to listen to the sensor's changes. The hardware will be turned off when all listeners are removed via `unregisterListener` (illustrated at line 22).

The sensor leak in the running example occurs as follows. After `UnlockActivity` is opened, the user may toggle `sc`'s switch in the UI, which will invoke `onCheckedChanged` and as a result will (1) register `shakeListener`'s listener object with `accel`'s sensor object, and (2) wait for a shake gesture from the user to unlock the vault. Whenever the device is moved, `onSensorChanged` is invoked with information about the physical movement. If this movement is above some threshold (checked at line 22), it is considered to be "shake to unlock" which releases the listener via `unregisterListener` and unlocks the vault. However, if the user does not shake the device, the listener will continue to listen for updates. If the user quits this app and makes the phone stationary, `UnlockActivity` will be closed. At that time, lifecycle callback `onDestroy` (line 28) does not release the sensor either. Thus, the window that acquired the sensor does not release it, which keeps the sensor alive and drains the battery. This sensor will still be alive after the user quits the app, as the application process remains active upon quitting. Using Sentinel, we generated a test case that triggers this behavior on an Android device.

3 GENERATION OF TEST CASES

Window transition graph. The starting point of test generation is a static app model referred to as the window transition graph (WTG) [31]. Graph nodes represent windows such as activities, dialogs, and menus; activity fragments are not represented. An edge $e = w_i \rightarrow w_j$ indicates that when window w_i is interacting with the user, some GUI event can cause window w_j to be displayed and to begin interacting with the user. It is possible that $i = j$, in which case the current window does not change. As discussed earlier, callback methods are executed during a transition from w_i to w_j. The WTG contains information about (1) the GUI event/widget that triggered the transition, (2) the callback methods executed during the transition, and (3) the corresponding window open/close effects.

Sensor effects control-flow graph. Next, we define the sensor effects control-flow graph (*SG*), a static model derived from the WTG and further analysis of callback methods along WTG edges. Paths in this graph correspond to GUI test cases. The graph is $SG = (N, E, L)$ where N and E are the node set and edge set from the WTG and $L : E \rightarrow \Sigma^*$ defines a label $l(e)$ for each $e \in E$. The label is a sequence of symbols from set $\{\text{open}(w_i), \text{close}(w_i), \text{acquire}(s_k), \text{release}(s_k)\}$.

Symbols $\text{open}(w_i)$ and $\text{close}(w_i)$ denote the opening/closing of a window $w_i \in N$. Symbols $\text{acquire}(s_k)$ and $\text{release}(s_k)$ denote the acquiring and release of a sensor s_k. The set of sensor abstractions s_k will be described later in this section. To determine label $l(e)$ for an edge e, we analyze the bodies of the callback methods executed during the transition represented by e (Section 3.3). Note that some WTG self-edges may not have any effects that correspond to such symbols; these edges are not included in *SG*.

Example. Figure 2 shows *SG* for the running example. Node w_1 corresponds to activity `Main`, which is not shown in the code from Figure 1. A widget event handler in w_1 opens `SettingActivity`. The self-edge for w_3 corresponds to a change in the state of switch widget `sc`; the invoked callback `onCheckedChanged` acquires the accelerometer sensor, denoted by s in the figure. Note that this example is rather simple. However, we have seen many apps where a single edge contains several symbols (e.g., it represents the opening/closing of several windows, or the acquiring of several sensors).

3.1 Sensor Leak Patterns

Using *SG*, we define two sensor leak patterns. These patterns are similar to GPS leaks observed in prior work [30]. In Android, the

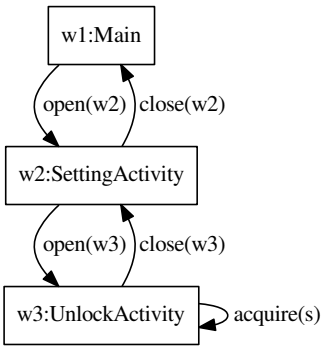

Figure 2: *SG* graph for the running example.

GPS is considered different from sensors and is managed via completely different APIs. That prior work did not consider sensors and did not perform test generation or execution. Our formulation is inspired by context-free language reachability [23], a well-known approach to define a set of paths in a labeled graph using a context-free language. Each *SG* path contains a sequence of edges; the concatenation of their labels forms a *path string*. If this string belongs to a pre-defined language, the path is "suspicious" and will be used to generate a test case. The focus on these particular suspicious paths is motivated by prior work that identifies them as potential sources of leaks [4, 5, 15, 30].

Leaks beyond window lifetime. We start by defining a language $L_1(w_i)$ describing *SG* paths that represent the lifetime of a window w_i. By intersecting this language with several regular languages over sensor acquire/release effects, we will capture one common pattern of sensor leaks. The subscript indicates that this is the first pattern being considered. A second pattern, described later, will be based on another language $L_2(w_i)$.

$L_1(w_i)$ is similar to classic balanced-parentheses languages:

$$\begin{array}{rcl} S_1 & \rightarrow & \mathsf{open}(w_i)\ Bal\ \mathsf{close}(w_i) \\ Bal & \rightarrow & \mathsf{open}(w_j)\ Bal\ \mathsf{close}(w_j) \mid Bal\ Bal \mid Sen \mid \epsilon \\ Sen & \rightarrow & \mathsf{acquire}(s_k) \mid \mathsf{release}(s_k) \end{array}$$

A string corresponds to a run-time execution scenario in which window w_i is opened, a number of other windows are opened and closed, and at the end w_i itself is closed. During the execution described by an $L_1(w_i)$ string, w_i is pushed on top of the window stack, additional push/pop operations are performed on top of w_i, and at the end w_i is popped from the stack. Any string from the language describes a possible lifetime for w_i.

To define the correct behavior for sensor effects, we define a regular language $R(s_k)$ for each sensor s_k, using a deterministic finite automaton $\mathcal{F} = (Q, \Sigma, \delta, q, f)$. Here $Q = \{q, f\}$ is the set of states, with q being the initial state and f being the final state. The input alphabet Σ is the set of symbols defined earlier. The transition function $\delta : Q \times \Sigma \rightarrow Q$ is shown in Figure 3. The figure shows only transitions for symbols acquire and release for the sensor of interest s_k. For the rest of Σ (open/close, as well as acquire/release for other sensors), there are self-transitions in both states.

If a string belongs to the language defined by $\mathcal{F}$, it represents a leak of sensor s_k. Note that in Android it is possible to perform successive acquire operations on the same sensor without in-between release operations; the second, third, etc. acquire have no effect. Similarly, it is possible to have successive release operations without in-between acquire; all but the first release are no-ops. Finally, it is also possible to execute release operations on a sensor that was never acquired. All these scenarios are captured by $\mathcal{F}$.

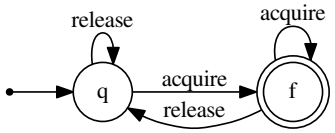

Figure 3: Finite automaton $\mathcal{F}$.

Consider language $P_1(w_i, s_k) = L_1(w_i) \cap R(s_k)$. If there exists an *SG* path whose string is in $P_1(w_i, s_k)$, the lifetime of window w_i acquires sensor s_k without releasing it, and thus matches our first pattern of sensor leaks. Any such path is a *static candidate* for a run-time sensor leak. Of course, due to the conservative nature of static analysis, it is possible that a static candidate does not actually trigger a sensor leak during execution. Thus, in SENTINEL a static candidate path is used to generate a test case whose execution is observed for an actual run-time leak.

Leaks in suspended state. The second pattern of sensor leaks will be illustrated using the example in Figure 4. CSipSimple is an open-source VoIP app that has been used by several commercial VoIP app which have more than a million downloads on Google Play Store. `InCallActivity` will register a listener for the proximity sensor in `onCreate` and will release this listener in `onDestroy`. This example does not exhibit the leak pattern described earlier: by the time the activity is destroyed, the sensor is released. However, another possible scenario is when the activity is suspended for a long period of time (e.g., hours). For example, if the user presses the HOME button, the app is put in the background but the sensor is still active.

To formalize this second pattern of sensor leaks, we add a symbol $\mathsf{suspend}(w_i)$ for each window w_i. Graph *SC* is augmented as follows: for each window w_i a new node $\bar{w}_i$ is added to represent the suspended state of w_i. An edge $w_i \rightarrow \bar{w}_i$ is labeled with symbols representing the sensor effects of lifecycle callbacks (e.g., `onPause`) executed before entering the suspended state. The last symbol on the edge is $\mathsf{suspend}(w_i)$. Another edge $\bar{w}_i \rightarrow w_i$ captures the sensor effects of resuming the app (e.g., sensors being reacquired in lifecycle callback `onResume`). Figure 4 illustrates these two callbacks at lines 4 and 5. In this app, both callbacks have no effect on sensors.

As before, we define a language to express how a window w_i reaches a suspended state. This language $L_2(w_i)$ is:

$$S_2 \rightarrow \mathsf{open}(w_i)\ Bal\ \mathsf{suspend}(w_i)$$

where *Bal* was defined earlier. Language $P_2(w_i, s_k) = L_2(w_i) \cap R(s_k)$ captures the scenario where w_i is suspended without releasing sensor s_k. This is the second sensor leak pattern we consider.

3.2 Generation of Test Cases

For any w_i, all path strings in language $P_1(w_i, s_k)$ can be determined by traversing *SG* paths starting at w_i and maintaining a stack corresponding to window open/close events. The stack elements are open and close symbols. A new open symbol is pushed on top of the stack. A new $\mathsf{close}(w_j)$ is allowed only if the current

```
1  class InCallActivity extends Activity {
2    CallProximityManager proximityManager = ...;
3    onCreate(...) { proximityManager.startTracking(); ... }
4    onResume() { ... }
5    onPause() { ... }
6    onDestroy() { proximityManager.stopTracking(); ... }
7  }
8  class CallProximityManager implements SensorEventListener {
9    SensorManager sm = ...;
10   Sensor proximitySensor = sm.getDefaultSensor(Sensor.SENSOR_PROXIMITY);
11   onSensorChanged(...) { ... }
12   startTracking() {
13     sm.registerListener(this, proximitySensor); ... }
14   stopTracking() {
15     sm.unregisterListener(this); ... }
16 }
```

Figure 4: Example derived from CSipSimple.

```
d.screen.on()  # turn on device screen
killApp("com.calculator.vault") # kill target app to clean up acquired resources
oldsensors = readAssociatedSensors() # gather currently-acquired sensors
startActivity("com.calculator.vault", "com.calculator.vault.UnlockSettingActivity")
d(resourceId="com.calculator.vault:id/shake_btn").click() # click the switch widget
d.press.back() # press the back button
newsensors = readAssociatedSensors() # gather acquired sensors after execution
# report differences between newsensors and oldsensors
```

Figure 5: Example of generated test case.

stack top is open(w_j) for the same window w_j; as a result, the top stack element is popped. During path traversal, the state of finite automaton $R(s_k)$ is updated based on symbols acquire and release. Since typically there would be several possible sensors s_k, several finite automata for the corresponding $R(s_k)$ would be maintained. The path strings for $P_2(w_i, s_k)$ can be generated similarly. Since the number of paths is typically infinite, we define a finite subset of paths using two criteria: (1) a path cannot contain the same edge more than once, and (2) the number of open and close symbols along a path cannot exceed a certain pre-defined limit k (our implementation uses $k = 4$). The second criterion captures the complexity of sequences of GUI control-flow events, regardless of how these events affect sensors. These two restrictions control the number and length of generated test cases.

Given an *SG* path generated as described above, it can be mapped to a sequence of GUI events using information available in the WTG. For example, for the graph in Figure 2, the path with edge labels open(w_3), acquire(s), close(w_3) will be mapped to the test case shown in Figure 5. The test case uses a Python wrapper for Google's UI Automator testing framework [28]. Several low-level details of the test case are omitted for brevity. Section 4 provides additional details on how such test cases are generated and executed.

Test case filtering. We employ two filtering techniques to reduce the number of generated test cases. First, note that all paths in a particular set $P_1(w_i, s_k)$ are in some sense equivalent: they exhibit the same pattern, for the same window w_i and sensor s_k. Thus, when generating test cases, we select only one path from each $P_1(w_i, s_k)$ set—specifically, any minimal-length path in that set. Similar filtering is applied to any $P_2(w_i, s_k)$.

Next, consider *SG* for the running example. The path with labels open(w_3), acquire(s), close(w_3) is in language $P_1(w_3, s)$. But the path with labels open(w_2), open(w_3), acquire(s), close(w_3), close(w_2) is in language $P_1(w_2, s)$. It is redundant to generate test cases for both paths: from the point of the view of a programmer, the "blame" should be assigned to activity w_3 because that activity was responsible for acquiring (but not releasing) the sensor. Thus, only a test case for the first path should be generated and executed.

To formalize this notion, consider again $P_1(w_i, s_k)$. During the traversal of a path with matching open and close symbols, we can observe when the finite automaton $\mathcal{F}$ for $R(s_k)$ transitions to its final state f. Such a transition corresponds to acquiring sensor s_k. When the transition occurs, we can record the current top symbol in the stack. This must be some open(w_j) symbol. If, later in the path, another transition to state f in $\mathcal{F}$ occurs, the old record of the stack top would be discarded and the new stack top would be recorded. When a path is determined to be in $P_1(w_i, s_k)$, it is reported only if the last recorded stack top is open(w_i)—that is, only if w_i was the currently-active window when s_k was last acquired. The same criterion is used for $P_2(w_i, s_k)$.

3.3 Static Sensor-Related Abstractions

A sensor s_k described earlier is actually a pair $\langle l, o \rangle$ of a sensor listener l and a sensor object o. For example, in Figure 1 the sensor being analyzed is a pair of the `SensorEventListener` object l referenced by `shakeListener` and the `Sensor` object o referenced by `accel`. To determine these s_k, our analysis first creates static abstractions of `Sensor` objects. One static object o per sensor type (e.g., accelerometer, proximity) is created. Next, propagation for constants `Sensor.TYPE_*` is used to determine which sensor types reach calls to `getDefaultSensor`. The sensor objects o returned by such calls are then propagated to calls to `registerListener`.

The listener objects are created by instantiating classes that implement interface `SensorEventListener`. Each such `new` expression corresponds to a static listener object l. These objects are also propagated to calls to `registerListener`. For every l and o that reach some such call, the analysis creates a corresponding sensor abstraction $s_k = \langle l, o \rangle$. Each such call is considered an instance of an "acquire" operation for s_k. Similarly, calls to `unregisterListener` are instances of "release" operations. Note that in both Figure 1 and Figure 4, the call to `unregisterListener` takes as a parameter the listener but not the sensor object. This method has two versions: one that takes as parameters both l and o, and another that takes only l. In the latter case, the call is considered to be a release operation for any $s_k = \langle l, \ldots \rangle$.

Recall that graph *SG* (illustrated in Figure 2) is derived from the window transition graph [31]. WTG edges are labeled with callbacks invoked at run time—e.g., lifecycle callbacks `onCreate`/`onDestroy` and event handler `onCheckedChanged` in Figure 1. Each such callback is analyzed to determine whether it contributes any acquire(s_k) or release(s_k) symbols to the corresponding *SG* edge.

This analysis of a callback method m considers m and its transitive callees in the app code. If any one of those methods contains an acquire operation for some s_k, it is necessary to check whether there is an interprocedural path from that operation to the exit of m that is free of a corresponding release of s_k. If such a path exists, callback m contributes symbol acquire(s_k). Callbacks `onCheckedChanged` in Figure 1 and `onCreate` in Figure 4 are examples of this case. It is also necessary to check whether every interprocedural path from the entry to the exit of m contains a release operation for s_k. If this is the case, the execution of m is guaranteed to release s_k and the callback contributes symbol release(s_k). Callback `onDestroy` in

Figure 4 illustrates this case. Note that a callback m could contribute both a release operation and an acquire operation for the same s_k (e.g., if it releases the sensor and then re-acquires it). In this case the analysis of m results in the string release(s_k), acquire(s_k).

In addition to lifecycle callbacks and GUI event handler callbacks, we also need to consider callback `onSensorChanged` (illustrated at lines 21–22 in Figure 1). Whenever a listener l is registered with a sensor object o, the listener is (almost) immediately notified of the current value of the sensor data, via an invocation of `onSensorChanged` on l. It is possible that this invocation unconditionally releases the sensor, and we have seen such examples in real apps. To account for this possibility, for each acquire(s_k) where $s_k = \langle l, o \rangle$, we identify the corresponding callback `onSensorChanged` for l and analyze it using the callback analysis described earlier. The contributions of the callback are appended at the end of each acquire(s_k) symbol in *SG*. For the example in Figure 1, the callback contains a release operation but this operation does not occur along every path, due to the `if` statement. If, hypothetically, the callback did *not* contain this `if`, it would contribute release(s). In this case, the self-edge for w_3 in Figure 2 would be labeled with acquire(s), release(s).

4 TOOL IMPLEMENTATION

4.1 Test Generation

Test generation in SENTINEL was implemented in the Soot framework [27]. The starting point of the implementation is the publicly-available Gator analysis toolkit for Android [8] which contains an implementation of the WTG representation [31]. Rather than explicitly building the sensor effects control-flow graph *SG*, our implementation directly uses the WTG. The analysis works in two stages. First, acquire(s_k) and release(s_k) symbols are introduced along WTG edges using the analysis described in Section 3.3. Next, *SG* paths are traversed to decide whether they exhibit the targeted patterns of open/close and acquire/release. To reduce the number of test cases, the analysis identifies equivalence sets of edges: if two edges have the same source, target, and label, they are equivalent. Only one edge per equivalence class is considered during path traversals. Test generation maps an *SG* path to a sequence of calls to UIAutomator API calls. Widgets are referenced using their ids defined in XML layout files or in `setId` calls in the code, as determined by Gator [24]. If widgets do not have static ids (e.g., list items), a test case cannot be generated. For widgets that require user input (e.g., `EditText`), manual post-processing is needed; we have seen a very small number of cases in which this occurs.

4.2 Test Execution

The generated test cases are executed using a Python wrapper for UI Automator [28, 29]. This framework allows testing to be controlled from a computer with direct access to the Android Debug Bridge (`ADB`), which is necessary for run-time sensor leaks measurements. Given a path reported by the static analysis, SENTINEL generates code which sets up the test case, starts the first activity on the path using an Android intent, and triggers the necessary GUI events. A simplified example of such code was presented in Figure 5. Depending on the application, it may be necessary to perform additional steps by the tester to fully set up the test case: for example, for the calculator vault app, it is necessary to setup a password for unlocking before the rest of the app can be used.

Acquired sensors with information about listener's package names and sensor types can be queried using `dumpsys` command in `ADB`. Each generated test case performs this measurement at the start and at the end of its execution (`readAssociatedSensors` in Figure 5). If a sensor is not at the start but is active at the end of the test case, and if the listener's package name is the same as the target application, a leak report will be generated. In our experiments, we executed the test cases and observed the sensor on a Google Nexus 5X smartphone with Android 7.1.2.

5 EVALUATION AND CASE STUDIES

We considered the entire set of apps in the F-Droid repository, as well as the top 100 apps from each category of Google Play. The evaluation of SENTINEL was performed on the entire subset of apps that contained sensor listeners (a total of 709 apps). The static analysis identified 18 apps for which the code exhibited the sensor leak patterns described earlier. Table 1 shows measurements for these apps. The first six apps are from F-Droid (also available at http://web.cse.ohio-state.edu/presto) and the rest are from Google Play. Column "Class" shows the number of classes in the app. This number includes classes in libraries that are included in the app. Our analysis considers the code in all these classes and makes no distinction between app code and code in third-party libraries. Column "Stmt" contains the number of Soot IR statements for these classes. Columns "Node" and "Edge" show the total number of *SG* nodes and edges, respectively.

The next six columns show measurements for the number of *SG* paths. Under "L paths" are included the number of paths with matching open and close symbols—that is, paths from languages L_1 and L_2 (Section 3.1) limited by parameter $k = 4$ and without duplicated edges (Section 3.2). For many applications, the number of such paths is in the thousands. Executing test cases for each such path may be expensive. However, it is possible to reduce this number significantly by performing our static sensor analysis. The analysis identifies GUI event handlers and lifecycle callbacks that trigger acquire and release symbols; based on this, it determines L_1 or L_2 paths that exhibit the sensor leak patterns. Columns "$L \cap R$ paths" show the numbers of paths that match the leak patterns. Clearly, significant reduction in the number of paths can be achieved. For further reduction, we use two filtering techniques to select the "guilty" window and to choose minimal-length paths (Section 3.2). Columns "Tests" shows the actual number of test cases generated by SENTINEL after this filtering. Again, significant reduction is observed, ultimately producing only a few test cases per app. These measurements demonstrate that static analysis of app code can successfully identify only a small subset of possible GUI event sequences that need to be executed at run time.

Columns "Leaks" show the number of executed test cases that resulted in an observed run-time leak. Columns with "–" represent test cases that could not be executed, as described shortly. As can be expected, not every executed test case leads to leaking behavior, due to the conservative nature of static analysis. For 12 apps, the test cases exposed sensor leaks. Later we discuss examples of test cases that did not have leaks. It is worth noting that the apps listed in the table are not "toy" projects: in particular, the apps from

Table 1: Applications, paths, and tests.

	App size				L paths		$L \cap R$ paths		Tests		Leaks		Time
Application	Class	Stmt	Node	Edge	P_1	P_2	P_1	P_2	P_1	P_2	P_1	P_2	(sec)
Mtpms	37	3148	20	38	7	8	6	6	1	1	1	1	0.03
Drismo	325	24592	232	402	1739	718	320	512	1	1	1	1	1.1
Geopaparazzi	1467	149469	273	524	615	721	24	24	1	1	1	1	2.5
Itlogger	296	30516	30	77	24	44	7	24	1	2	1	2	0.7
AIMSICD	921	79438	59	136	67	69	4	2	1	1	1	1	0.8
Coregame	44	1988	3	7	2	2	2	2	1	1	1	1	0.07
NightVisionCamera	408	44399	74	93	5	25	0	16	0	1	0	–	0.5
Voxofon	2637	184406	451	1084	4011	994	0	2	0	1	0	1	9.1
VRVideoPlayer	334	24296	13	29	5	8	2	2	1	1	0	0	0.7
MobinCube	502	67186	959	1149	12735	83209	3155	3907	3	3	0	0	37.1
CSipSimple	1319	111659	57	223	750	402	0	100	0	1	0	1	10.7
Calculator Vault	2025	137364	220	694	49064	13837	1128	3108	2	2	2	2	179.9
Comebacks	160	21246	67	96	46	50	0	16	0	1	0	–	0.2
Pushups	956	87545	660	1434	14460	2607	0	2	0	1	0	0	2.2
Dogwhistier	2415	181626	125	302	2257	1739	25	27	1	1	1	1	49.2
Hideitpro	3315	227087	807	1602	5062	1434	1040	1321	0	1	0	1	12.5
LikeThatGarden	1678	115092	634	1748	575773	201332	8165	21425	1	1	1	1	70.8
MyMercy	907	74914	258	629	286	400	0	5	0	1	0	–	24.1

Google Play are among the most popular in their categories and have many thousands of downloads from users. These results show that even popular applications can contain sensor leaks and our test generation approach can expose these leaks successfully.

Test generation time, in seconds, is shown in column "Time". It includes callback analysis of acquire and release effects, path checking, and test case generation. These measurements indicate that the cost of code analysis and test generation is practical.

Next, we briefly present several case studies. For F-Droid apps, we considered the publicly-available source code. For Google Play apps, we used the `jadx` decompiler to study the app code.

CSipSimple. This VoIP app was illustrated in Figure 4. Activity `InCallActivity` will be started by an Android intent broadcast when there is an incoming or an outgoing call. When this activity is launched, lifecycle callback `onCreate` will be invoked and its callee method `startTracking` will acquire the proximity sensor. The activity does release the sensor in `stopTracking`, which is invoked by callback `onDestroy`. However, if a user presses the HOME button during the call and navigates to other applications, e.g., for browsing a web page or looking up a contact, the acquired sensor will still be held by `InCallActivity` even though it is not responding to user interactions.

Geopaparazzi. This F-Droid app, which is also available in Google Play, is used for engineering and geologic surveys. Figure 6 shows the simplified code for the leak. The app uses a wrapper class `SensorManagerL` to process all sensor-related operations. (We use this name for brevity; in reality, this is app class `eu.hydrologis.geopaparazzi.SensorManager`.) The class implements the singleton pattern. At line 12, method `startSensorListening` registers a listener for the accelerometer. This method is called during the singleton object creation (line 8). `GeoPaparazziActivity`'s callback `onCreate` calls `init`, which instantiates the singleton and acquires the sensor. The only method that releases the sensor is

```
1  class SensorManagerL implements SensorEventListener {
2    SensorManagerL sml;
3    SensorManager sm;
4    static SensorManagerL getInstance(...) {
5      if (sml == null) {
6        sml = new SensorManagerL();
7        sm = (SensorManager) getSystemService(SENSOR_SERVICE);
8        sml.startSensorListening();}
9      return sml; }
10   void startSensorListening() {
11     Sensor accel = sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
12     sm.registerListener(sml, accel); ... }
13   void stopSensorListening() { sm.unregisterListener(sml);}
14   void onSensorChanged(...) {...} }
15 class GeoPaparazziActivity extends Activity {
16   SensorManagerL sml;
17   void onCreate(...) { init(); ... }
18   void init() { sml = SensorManagerL.getInstance(); } }
```

Figure 6: Example derived from Geopaparazzi.

`stopSensorListening` (line 13). However, this method is not called by any app component. Once the activity turns on the accelerometer sensor, it can only be turned off by killing the app.

mTpms. This app from F-Droid is a motorcycle tire pressure monitor system reader. It uses the device's light sensor to detect changes of ambient light. It will change the background and text to dark colors when it detects that light level is below a certain threshold. In the `onCreate` method of the main activity, the app obtains the sensor object and registers a listener for it. However, there is no app code that unregisters this listener. Therefore, the light sensor will be turned on when this application is launched and will remain on unless this application is killed.

Non-executable test cases. Three generated test cases could not be executed ("–" table entries). Our tests use an explicit intent to open the first activity in a test case. This is a typical approach for unit testing for Android, but in those three cases the activity crashes when opened. We also attempted, unsuccessfully, to trigger these activities using GUI sequences that start from the main app activity. For `MyMercy`, such a sequence requires a pre-existing medical

account, which we are not able to obtain. For `NightVisionCamera` and `Comebacks`, the problematic activity is supposed to display a full-screen ad when the user clicks on an ad banner, but we were unable to trigger these ads on our device or in the emulator.

Test cases without leaks. For three apps, the test cases do not produce run-time leaks. In all three cases there are classes containing methods which override the same methods in their superclasses. The subclass methods acquire sensors and leak them. However, these subclasses are never instantiated at run time. Due to the use of class hierarchy analysis, when our analysis encounters an invocation of the superclass method, it incorrectly determines that the called method could be from the defective subclass. This imprecision causes the false positives. This is a well-known limitation of class hierarchy analysis. There are many options for more precise call graph construction [25], but they need to be adapted to the framework/callback-driven control flow in Android apps.

6 RELATED WORK

Test generation for Android. There are various techniques for automated testing for mobile apps [7, 11, 12, 26]. The Monkey testing tool [20] generates UI events randomly. Android GUI Ripper [1] and MobiGUITAR[2] generate test cases based on dynamically built GUI models. Yang et al. [32] explore the application dynamically, but use static analysis to determine relevant UI elements. CrashScope [21] uses a similar model-based approach for detecting and reporting run-time crashes. Work by Zhang et al. [33] generates UI tests based on a static model to expose resource leaks. It does not use static analysis to reduce duplicated tests and does not analyze acquire/release sequences for these resources. Jensen et al. [10] generate event sequences using a UI model and event handler summaries derived from static analysis. Examples of other relevant tools are EvoDroid [18], Dynodroid [17], SwiftHand [6], PUMA [9], and TrimDroid[19].

Leak detection. There is a body of work on leak detection for Android. Some approaches use run-time analysis. GreenDriod [15] detects energy related resource underuntilization and leaks using Java PathFinder, based on a hybrid UI model. A similar approach by Banerjee et al. [3] uses a modified version of Dynodroid [17] to perform dynamic GUI ripping, followed by analysis of energy-related leaks either dynamically [3, 4] or using a hybrid dynamic/static approach [5]. Ma et al. [16] developed a dynamic leak detector based on UI traversal and memory profiling. Unlike these techinques, our approach uses a purely static analysis that aims to focus the testing efforts on a small subset of possible run-time behaviors.

7 CONCLUSIONS

This work demonstrates that is is possible to automatically generate effective tests for sensor leaks in Android apps. While there are many possible GUI event sequences, sensor-aware static analysis can reduce dramatically the number of event sequences executed during testing. Our evaluation confirms the utility of the proposed SENTINEL testing tool in exposing sensor leaks in realistic apps.

Acknoledgements. We thank the AST reviewers for their valuable feedback. This material is based upon work supported by the U.S. National Science Foundation under CCF-1319695 and CCF-1526459, and by a Google Faculty Research Award.

REFERENCES

[1] D. Amalfitano, A.R. Fasolino, P. Tramontana, S. De Carmine, and A.M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *ASE*. 258–261.
[2] D. Amalfitano, A.R. Fasolino, P. Tramontana, B.D. Ta, and A.M. Memon. 2015. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software* (2015), 53–59.
[3] A. Banerjee, L.K. Chong, S. Chattopadhyay, and A. Roychoudhury. 2014. Detecting energy bugs and hotspots in mobile apps. In *FSE*. 588–598.
[4] A. Banerjee, H. Guo, and A. Roychoudhury. 2016. Debugging energy-efficiency related field failures in mobile apps. In *MOBILESoft*. 127–138.
[5] A. Banerjee and A. Roychoudhury. 2016. Automated re-factoring of Android apps to enhance energy-efficiency. In *MOBILESoft*. 139–150.
[6] W. Choi, G. Necula, and K. Sen. 2013. Guided GUI testing of Android apps with minimal restart and Approximate Learning. In *OOPSLA*. 623–640.
[7] S.R. Choudhary, A. Gorla, and A. Orso. 2015. Automated test input generation for Android: Are we there yet?. In *ASE*. 429–440.
[8] Gator 2017. *Gator: Program Analysis Toolkit For Android.* http://web.cse.ohio-state.edu/presto/software/gator.
[9] S. Hao, B. Liu, S. Nath, W.G.J. Halfond, and R. Govindan. 2014. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *MobiSys*. 204–217.
[10] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *ISSTA*. 67–77.
[11] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon. 2017. Static analysis of Android apps: A systematic literature review. *IST* 88 (2017), 67–95.
[12] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *ICSME*. 399–410.
[13] Y. Liu, C. Xu, S.C. Cheung, and V. Terragni. 2016. Understanding and detecting wake lock misuses for Android applications. In *FSE*. 296–409.
[14] Y. Liu, C. Xu, and S. C. Cheung. 2013. Where has my battery gone? Finding sensor related energy black holes in smartphone applications. In *PerCom*. 2–10.
[15] Y. Liu, C. Xu, S. C. Cheung, and J. Lu. 2014. GreenDroid: Automated diagnosis of energy inefficiency for smartphone applications. *TSE* 40 (Sept. 2014), 911–940.
[16] J. Ma, S. Liu, S. Yue, X. Tao, and J. Lu. 2017. LeakDAF: An automated tool for detecting leaked activities and fragments of Android applications. In *COMPSAC*. 23–32.
[17] A. Machiry, R. Tahiliani, and M. Naik. 2013. Dynodroid: An input generation system for Android apps. In *FSE*. 224–234.
[18] R. Mahmood, N. Mirzaei, and S. Malek. 2014. EvoDroid: Segmented evolutionary testing of Android apps. In *FSE*. 599–609.
[19] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. 2016. Reducing combinatorics in GUI testing of Android applications. In *ICSE*. 559–570.
[20] Monkey 2017. Monkey: UI/Application Exerciser for Android. (2017). http://developer.android.com/tools/help/monkey.html.
[21] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk. 2016. Automatically discovering, reporting and reproducing Android application crashes. In *ICST*. 33–44.
[22] A. Pathak, A. Jindal, Y.C. Hu, and S.P. Midkiff. 2012. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys*. 267–280.
[23] Thomas Reps. 1998. Program analysis via graph reachability. *Information and Software Technology* 40, 11-12 (1998), 701–726.
[24] Atanas Rountev and Dacong Yan. 2014. Static reference analysis for GUI objects in Android software. In *CGO*. 143–153.
[25] B.G. Ryder. 2003. Dimensions of precision in reference analysis of object-oriented programming languages. In *CC*. 126–137.
[26] A. Sadeghi, R. Jabbarvand, and S. Malek. 2017. PATDroid: Permission-aware GUI testing of Android. In *FSE*. 220–232.
[27] Soot 2017. *Soot Analysis Framework.* http://www.sable.mcgill.ca/soot.
[28] UI Automator 2017. *UI Automator Testing Framework.* http://developer.android.com/training/testing/ui-automator.html.
[29] UI Automator Wrapper 2017. Python wrapper of Android UI Automator test tool. (2017). http://github.com/xiaocong/uiautomator.
[30] H. Wu, S. Yang, and A. Rountev. 2016. Static detection of energy defect patterns in Android applications. In *CC*. 185–195.
[31] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. 2015. Static window transition graphs for Android. In *ASE*. 658–668.
[32] W. Yang, M. Prasad, and T. Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *FASE*. 250–265.
[33] H. Zhang, H. Wu, and A. Rountev. 2016. Automated test generation for detection of leaks in Android applications. In *AST*. 64–70.