Sentinel: generating GUI tests for sensor leaks in Android and Android wear apps

Haowei Wu, Hailong Zhang, Yan Wang & Atanas Rountev

Software Quality Journal

ISSN 0963-9314

Software Qual J
DOI 10.1007/s11219-019-09484-z

ONLINE FIRST

Springer

Software Quality Journal
https://doi.org/10.1007/s11219-019-09484-z

SENTINEL: generating GUI tests for sensor leaks in Android and Android wear apps

Haowei Wu[1] · Hailong Zhang[2] · Yan Wang[1] · Atanas Rountev[2]

Published online: 12 December 2019

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. Edges in the model are labeled with symbols representing the acquiring/releasing of sensors and the opening/closing of UI windows. The model is traversed to identify paths that are likely to exhibit sensor leaks during run-time execution based on two context-free languages over the symbol alphabet. 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. This approach has been applied to both open-sourced and closed-sourced regular Android applications as well as watch faces for Android Wear smartwatches. 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.

Keywords Android · GUI · Android Wear · Smartwatch · Energy · Sensor · Static analysis · Testing

The two lead authors Haowei Wu and Hailong Zhang contributed equally to this work.

✉ Hailong Zhang
zhang.4858@osu.edu

Haowei Wu
haowei@google.com

Yan Wang
wysnow7@gmail.com

Atanas Rountev
rountev@cse.ohio-state.edu

[1] Google Inc., Mountain View, CA 94043, USA

[2] Ohio State University, Columbus, OH 43210, USA

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. Studies have shown that such improvements are important for both developers (e.g., to achieve market success) and app markets (e.g., to maintain market quality and prestige) (Corral and Fronza 2015; Corral et al. 205; d'Heureuse et al. 2012). Complex event-driven behavior and limited device resources present challenges for developers. One such challenge comes from various "leaking" behaviors that can lead to energy-related inefficiencies (Pathak et al. 2012; Liu et al. 2013, 2014, 2016; Banerjee et al. 2014, 2016; Wu et al. 2016a; Banerjee and Roychoudhury 2016).

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 *sen*sor *t*est*in*g to d*e*tect *l*eaks. The tool takes as input an Android Application Package (APK),[1] 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 will be referred to as the *sensor effects control-flow graph (SG)* in the rest of the paper. Edges in the graph are labeled with symbols representing the opening/closing of UI windows and the acquiring/releasing of sensors. We then define a *context-free-language reachability* (CFL-R) problem over the graph. This problem is based on two context-free languages over the symbol alphabet. A graph path whose edge labels define a string from these languages is suspicious and is likely to exhibit sensor leaks during run-time execution.

SENTINEL considers two categories of likely 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. SG paths that match these two patterns are used to generate test cases. The execution of each test case tracks the run-time behavior of sensors and reports observed sensor leaks.

In addition to sensor leaks in regular Android applications, SENTINEL was also applied to uncover sensor leaks in wearable devices—specifically, Android Wear smartwatches. Android Wear (AW) is Google's platform for developing apps for wearable devices (Google 2018b). Many features and behaviors of AW apps are different from those of regular Android apps. One major category of AW apps is *watch faces*. Such apps are prevalent in AW app markets (Zhang et al. 2018). A watch face displays the current time together with rich information such as weather, daily agenda, and the user's exercise statistics. Our study of watch faces, described elsewhere (Zhang et al. 2018), identified energy inefficiency as a key consideration for these apps, and highlighted that sensor leaks are one of the major sources of such inefficiency. Based on this study, we developed a variant of SENTINEL

[1] The file format used by Android for distribution and installation of apps.

which constructs the SG model for a given watch face and then identifies paths that match the two categories of sensor leaks described earlier. As for regular Android apps, we use the problematic SG paths to generate test cases which then are executed to confirm the presence of leaks at run time.

Our experimental results indicate that SENTINEL effectively detects sensor leaks in Android apps and Android Wear watch faces, 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. The implementation SENTINEL and all benchmarks used in its evaluation are available at https://presto-osu.github.io/Sentinel.

An earlier version of the work on sensor leaks testing for Android apps appeared at the 13th IEEE/ACM International Workshop on Automation of Software Test (Wu et al. 2018). Some of the material on sensor leaks testing for watch faces is based on work that was published at the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Zhang et al. 2018). These two approaches were developed independently. This journal paper proposes unified analysis formalism and algorithms that capture both approaches. Compared to our prior work on sensor leak testing for regular Android apps (Wu et al. 2018), we propose more general leak patterns (Section 3.2) and corresponding algorithms (Section 3.3), as well as additional implementation details (Section 3.6). Updated evaluation is presented in Section 6.1 in accordance with the new definitions.

Our prior work on leak testing for watch faces (Zhang et al. 2018) does not define any leak pattern languages at all. There is no context-free grammar of any kind, nor is there a definition of a "leaking string." By defining the generalized languages in this journal paper, we show that the sensor leaks for watch faces from that prior work are intrinsically similar to the ones we considered earlier for regular Android apps (Wu et al. 2018). This similarity exists even thought there are completely different APIs, component lifecycles, etc. between regular Android apps and Android Wear watch faces. This means that essentially the same test generation machinery can be used in both cases. The unification of the leak testing techniques for regular apps and for watch faces indicates that the underlying formalisms, code analyses, and test generators have certain generality and could also potentially be useful in other scenarios.

In this journal paper, we propose a modified control-flow model for watch faces, different from the one in our prior work (Zhang et al. 2018) and better suited for the leak pattern analysis. We also propose an SG model for watch faces (Section 4.3; not present in our prior work), a new SG-traversal static analysis (Section 4.4), and a new implementation of test generation (different from the previous implementation, although equivalent in terms of final result). The experimental results (Section 6.2) have been revised to use the new formulation and implementation.

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, callback `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 behavior for certain events, such as pressing the BACK button. As a result, *lifecycle*

```
 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 }
```

Fig. 1 Example derived from Calculator Vault

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 subscript 1 denotes `SettingActivity` and subscript 2 dentoes `UnlockActivity`.

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, `Sensor.TYPE_ACCELEROMETER` corresponds to all accelerometer sensors. 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 Fig. 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 opening `UnlockActivity`, 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 activity (e.g., by pressing the BACK button) 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. After additional GUI events, the user could quit the app and return to the main screen of the device. However, the sensor will still be alive after this, 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 for sensor leaks in Android apps

3.1 Static models

Window transition graph The starting point of test generation is a static app control-flow model referred to as the window transition graph (WTG) (Yang et al. 2015b). Formally, the

WTG is defined as $G = (N, E, \epsilon, \delta, \sigma)$. Graph nodes in N represent windows such as activities, dialogs, and menus; activity fragments are not represented. An edge $e = (w_i, w_j) \in E \subseteq N \times N$ 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. We will use $w_i \rightarrow w_j$ to denote an edge from w_i to w_j. It is possible that $i = j$, in which case the current window does not change.[2]

Let V be the set of GUI events defined by Android semantics (e.g., "click" events). Labels $\epsilon : E \rightarrow V$ indicate that the window transition represented by an edge could be triggered due to a particular event. As discussed earlier, callback methods are executed during a transition from w_i to w_j and windows may be opened/closed. The callback methods are defined in the app code but they override methods from the Android platform code. These platform methods often come from interfaces; a typical example is interface `Android.view.View.OnClickListener` which defines a signature for a method `onClick` to handle "click" events. In addition to such widget event handlers, a transition may trigger callback methods to manage the lifecycle of windows. Standard examples are methods `onCreate` and `onDestroy` for activities. Let C denote the set of callback methods. Labels $\sigma : E \rightarrow ((W \cup N) \times C)^*$ shows the sequence of callbacks for the transition. Here W is the set of GUI widgets on which events are triggered. For an edge $e \in E$, $\sigma(e)$ is a sequence of pairs (w, c) where callback c was invoked to handle an event on widget w, or (n, c) where c is a lifecycle callback for window n. In addition to execution of callback methods, a transition from w_i to w_j may open/close windows. Labels $\delta : E \rightarrow (\{\mathsf{open}, \mathsf{close}\} \times N)^*$ annotate each edge with a sequence of open and close operations. The meaning of an edge $e = w_i \rightarrow w_j$ is as follows. Suppose that the currently active window is w_i. If event $\epsilon(e)$ is issued by the user, the processing of this event triggers the open/close operations described by $\delta(e)$, resulting in w_j being the newly active window. During these changes, the callback sequence $\sigma(e)$ is observed.

Sensor effects control-flow graph Next, we define the sensor effects control-flow graph (SG), a static model derived from the WTG and via further analysis of callback methods along WTG edges. Each path in this graph corresponds to a sequence of GUI events. Such event sequences are used to build the test cases produced by the test generation described later in Section 3.3.

The graph is $SG = (N, E, \epsilon, \sigma, \psi)$ where N and E are the node set and edge set from the WTG, ϵ and σ are the edge labels in the WTG, and $\psi : E \rightarrow \Sigma^*$ defines a label $\psi(e)$ for each $e \in E$. The label is a sequence of symbols from set $\Sigma = (\{\mathsf{open}, \mathsf{close}\}) \times N) \cup (\{\mathsf{acquire}, \mathsf{release}\} \times (L \times S))$ where L denotes the set of sensor event listeners and S denotes the set of sensor objects, illustrated in Section 2.2 and defined formally in Section 3.5. In the rest of the paper we denote elements of Σ by $\mathsf{open}(\cdot)$, $\mathsf{close}(\cdot)$, $\mathsf{acquire}(\cdot)$, and $\mathsf{release}(\cdot)$.

For SG we replace the label function δ from the WTG with ψ, which takes into account sensor effects. Symbols $\mathsf{open}(w)$ and $\mathsf{close}(w)$ denote the opening/closing of a window $w \in N$. Symbols $\mathsf{acquire}(s_k)$ and $\mathsf{release}(s_k)$ denote the acquiring and release of a sensor $s_k \in (L \times S)$. The set of sensor abstractions s_k is described in detail in Section 3.5. To determine label $\psi(e)$ for an edge e, we analyze the bodies of the callback methods executed during the transition, i.e., the callback sequence $\sigma(e)$ from the WTG. Note that some WTG self-edges may not have any effects that correspond to such symbols; these edges are not included in SG.

[2] In some cases (e.g., when the device is rotated) the current window is destroyed and then recreated with a different layout. Such cases are also represented as $w_i \rightarrow w_i$ transitions.

Example Figure 2 shows *SG* for the running example. Labels ϵ and σ are omitted for simplicity. Node w_1 corresponds to activity `Main`, which is not shown in the code from Fig. 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).

3.2 Sensor leak patterns

Using *SG*, we define two sensor leak patterns. These patterns are similar to GPS leaks observed in prior work (Wu et al. 2016a). In Android, the 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 (Reps 1998), 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, for example, one of the two languages defined below, 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 (Liu et al. 2014; Banerjee et al. 2016; Banerjee and Roychoudhury 2016; Wu et al. 2016a).

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. For any window w_i, $L_1(w_i)$ is defined by the following context-free grammar:

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

where λ represents an empty string, w_j denotes any window and s_k is a sensor abstraction $s_k \in (L \times S)$, as defined in Section 3.5. Here *Bal* describes balanced sequences of matching open and close symbols that could be interleaved with acquire/release symbols. A string of $L_1(w_i)$ 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.

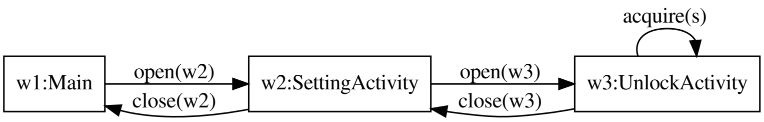
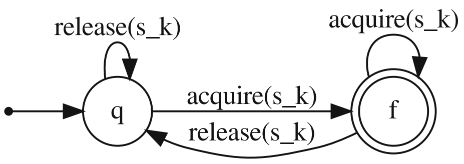


Fig. 2 *SG* graph for the running example

Fig. 3 Finite automaton $\mathcal{F}_k$



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 $\mathcal{F}_k = (s_k, Q, \Sigma, \delta, q, f)$. 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 Fig. 3. The figure shows only transitions for symbols acquire and release for the sensor of interest s_k. For the rest of Σ—that is, open(w)/close(w), as well as acquire/release for other sensors—there are self-transitions in both states.

If a string belongs to the language defined by $\mathcal{F}_k$, 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 $\mathcal{F}_k$.

Consider the context-free 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 Fig. 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.

```
class InCallActivity extends Activity {
  CallProximityManager proximityManager = ...;
  onCreate(...) { proximityManager.startTracking(); ... }
  onResume() { ... }
  onPause() { ... }
  onDestroy() { proximityManager.stopTracking(); ... }
}

class CallProximityManager implements SensorEventListener {
  SensorManager sm = ...;
  Sensor proximitySensor = sm.getDefaultSensor(Sensor.SENSOR_PROXIMITY);
  onSensorChanged(...) { ... }
  startTracking() {
    sm.registerListener(this, proximitySensor); ... }
  stopTracking() {
    sm.unregisterListener(this); ... }
}
```

Fig. 4 Example derived from CSipSimple

`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, for each window w_i, we add a symbol $\mathsf{suspend}(w_i)$. Graph *SG* 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 $\mathsf{suspend}(w_i)$. Figure 5 shows part of the augmented SG for `InCallActivity` from the CSipSimple app in Fig. 4.

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

$$\begin{aligned} S_2 &\rightarrow \mathsf{open}(w_i)\ \mathit{Val}\ \ \mathsf{suspend}(w_m) \\ \mathit{Val} &\rightarrow \mathsf{open}(w_j)\ \mathit{Val} \mid \mathit{Bal}\ \mathit{Val} \mid \lambda \end{aligned}$$

where *Bal* and λ are defined earlier and w_j and w_m denote any windows. Here *Val* represents a valid sequence of symbols which could have not-yet-matched open symbols. Language $P_2(w_i, s_k) = L_2(w_i) \cap R(s_k)$ captures the scenario where execution is suspended without releasing sensor s_k. This is the second sensor leak pattern we consider. Note that this pattern definition generalizes the one from the earlier version of this work (Wu et al. 2018), in that it uses the non-terminal *Val* to allow the final $\mathsf{suspend}$ symbol on a different window other than the first opened window w_i and multiple unmatched open symbols between them.

3.3 Generation of test cases

For any w_i, 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. When an open symbol is encountered along a path, it is pushed on top of the stack. For a $\mathsf{close}(w_j)$ symbol, the top of the stack is checked for a matching $\mathsf{open}(w_j)$; if there is a match, $\mathsf{open}(w_j)$ is popped from the stack and the path traversal continues. When the stack is empty, the traversed path matches language $L_1(w_i)$. For language $L_2(w_i)$, the path matches if the stack is not empty and any $\mathsf{suspend}$ is encountered.

There are two ways in which the sensor effects of traversed paths could be accounted for, in order to identify strings from languages $P_1(w_i, s_k)$ and $P_2(w_i, s_k)$. First, during path traversal, the state of finite automaton $R(s_k)$ can be updated based on symbols $\mathsf{acquire}$ and $\mathsf{release}$. Since typically there would be several possible sensors s_k, several finite automata for the corresponding $R(s_k)$ would be maintained. Alternatively, the entire set of paths for $L_1(w_i)$ and $L_2(w_i)$ could be generated first, and then later each path could be checked for each sensor s_k. Our implementation follows the second approach as it is easier to implement and has more potential for parallelization. Since the number of L_1 and L_2 paths is

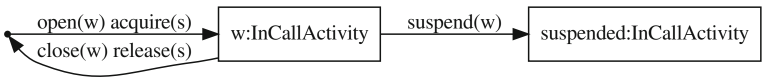


Fig. 5 Augmented *SG* for the example in Fig. 4

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 ℓ. Our implementation uses $\ell = 4$. We have tested larger values for ℓ in the experiment but found no more leaks by the test generation and execution, which will be discussed in Section 5. 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 a generated *SG* path, it can be mapped to a sequence of GUI events using labels ϵ. For example, for the graph in Fig. 2, the path with edge labels open(w_3), acquire(s), close(w_3) will be mapped to the test case shown in Fig. 6. The test case uses a Python wrapper for Google's UI Automator testing framework (He 2018). Statements at lines 1, 5 and 6 call APIs of the wrapper for turning on device screen, selecting certain widgets and pressing the BACK button, respectively. Helper functions `killApp` at line 2 and `startActivity` at line 4 use Android Debug Bridge (ADB) (Google 2018a) to stop and start an application activity given its package name and `Activity` name. Helper function `readAssociatedSensors` at line 3 and 7 sends `dumpsys` (Google 2018d) commands to retrieve information about active sensors from Android's sensor service. By comparing the sensor information before and after executing the test case, developers can confirm the existence of leaks along the path. Section 5 describes the details about the execution of tests.

3.4 Test case filtering

We employ three 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, after leaking paths are generated, we select only one path from each $P_1(w_i, s_k)$ set for test generation—specifically, any minimal-length path in that set. Similar filtering is applied to any $P_2(w_i, s_k)$. Formally, given a set P of paths, the filter returns some element of set $\text{argmin}_{p \in P}\, \textit{length}(p)$, where $\textit{length}(p)$ gives the length of a path.

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 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.

```
1 d.screen.on()  # turn on device screen
2 killApp("com.calculator.vault") # kill app to clean up acquired resources
3 oldsensors = readAssociatedSensors() # gather currently-acquired sensors
4 startActivity("com.calculator.vault", "com.calculator.vault.UnlockActivity")
5 d(resourceId="com.calculator.vault:id/shake_btn").click() # click the widget
6 d.press.back() # press the back button
7 newsensors = readAssociatedSensors() # get acquired sensors after execution
8 # report differences between newsensors and oldsensors
```

Fig. 6 Example of generated test case

To achieve this filtering, we consider a modified version of language $L_1(w_i)$, specialized to a particular s_k. Let $L_1(w_i, s_k)$ denote this language. Its definition uses a modified version of the first production for $L_1(w_i)$:

$$S_1 \rightarrow \mathsf{open}(w_i)\ Bal\ \mathsf{acquire}(s_k)\ Bal\ \mathsf{close}(w_i)$$

The remaining productions are the same. The effect of this change is the following. At the time when $\mathsf{acquire}(s_k)$ is encountered, the top of the open/close stack is $\mathsf{open}(w_i)$, because the *Bal* balanced subpath has no effect on the stack state. This means that w_i is responsible for acquiring s_k. Thus, if $\mathsf{acquire}(s_k)$ leaks, we should blame w_i. We then redefine the leak pattern language as $P_1(w_i, s_k) = L_1(w_i, s_k) \cap R(s_k)$.

To achieve this filtering, during the traversal of an $L_1(w_i)$ path we ignore $\mathsf{acquire}(s_k)$ if w_i was not responsible for acquiring s_k. To make this decision, we consider the state of the open/close stack at the time when $\mathsf{acquire}(s_k)$ was encountered. If the top of the stack is not $\mathsf{open}(w_i)$, the acquire operation is ignored.[3] This guarantees that any leaking s_k reported due to $P_1(w_i, s_k)$ can be blamed on w_i.

Similar filtering is used for $P_2(w_i, s_k)$. In this case, a specialized language $L_2(w_i, s_k)$ is derived from the general language $L_2(w_i)$ by replacing the first production with

$$S_2 \rightarrow \mathsf{open}(w_i)\ Bal\ \mathsf{acquire}(s_k)\ Val\ \mathsf{suspend}(w_m)$$

Intuitively, when $\mathsf{acquire}(s_k)$ occurs, the top of the open/close stack is $\mathsf{open}(w_i)$ and the leaking path should be blamed on w_i.

The last filter we apply is for $L_2(w_i, s_k)$ paths. While the definition of this language allows strings starting with $\mathsf{open}(w_i)$ and ending with $\mathsf{suspend}(w_m)$, we only report paths for which w_m is either the same as w_i, or a menu/dialog acting on behalf of w_i (menus and dialogs will be described shortly). It is easy to show that if there exists any leaking $L_2(w_i, s_k)$ path, there also exists a leaking $L_2(w_i, s_k)$ path that satisfies this constraint. The changes in the production for $L_2(w_i, s_k)$ are straightforward: $\mathsf{suspend}(w_m)$ at the end of the path is restricted to be one of the allowed w_m as opposed to any w_m.

3.5 Static sensor-related abstractions

A sensor $s_k \in (L \times S)$ described earlier is defined as a pair $\langle l, o\rangle$ of a sensor listener object $l \in L$ and a sensor object $o \in S$, where L denotes the set of all sensor listener objects and S denotes the set of all sensor objects. Recall from Section 2.2 that Android defines several sensor types, encoded by integer constants in fields `Sensor.TYPE_*`—for example, `Sensor.TYPE_ACCELEROMETER`. In general, there could be several `Sensor` objects (created internally by the Android framework) for each sensor type. In practice, we have observed that apps used only one `Sensor` object per sensor type, usually obtained via the `SensorManager.getDefaultSensor` API. Thus, we treat each `Sensor.TYPE_*` constant field as a representative of a unique `Sensor` objects. S is the set of all such unique objects. Set L contains `new` expressions that instantiate sensor listener classes.

In Fig. 1 the sensor being analyzed is a pair of the `SensorEventListener` object l referenced by variable `shakeListener` and the `Sensor` object o referenced by `accel`. To determine these s_k, our analysis first creates static abstractions of `Sensor` objects; these abstractions form set S. One static object $o \in S$ per sensor type `Sensor.TYPE_*` (e.g., accelerometer, proximity) is created. Next, constant propagation from constant fields

[3] More generally, the top of the stack could be $\mathsf{open}(w_j)$ for some menu or dialog w_j working on behalf of activity w_i. This generalization is discussed in Section 3.6.

`Sensor.TYPE_*` to calls to `getDefaultSensor` is performed. The sensor objects o returned by such calls, based on the incoming sensor type integer constant, are then propagated to calls to `registerListener`.

Why this representation? We use this abstraction of sensors since a sensor object only takes effect when it is associated with some sensor listener. Another reason to use this notion is that there could be several listeners for the same sensor object, or several sensors that one listener listens to. We have observed such cases during our experimental evaluation and case studies. Using this abstraction allows us to uniquely identify the listener object and sensor object at each acquire/release operation, as well as the set of listeners for each sensor object.

Run-time listener objects are created by instantiating classes that implement interface `SensorEventListener`. Each such `new` expression corresponds to a static listener object $l \in L$. These objects are also propagated to `registerListener` calls. 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 Figs. 1 and 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 Fig. 2) is derived from the window transition graph (Yang et al. 2015b). WTG edges are labeled with invoked callbacks—e.g., lifecycle callbacks `onCreate/onDestroy` and event handler `onCheckedChanged` in Fig. 1. Each callback is analyzed to determine whether it contributes any $\mathsf{acquire}(s_k)$ or $\mathsf{release}(s_k)$ symbols to the corresponding SG edge. In addition, each WTG edge describes the window open and close effects, from which symbols $\mathsf{open}(w_i)$ and $close(w_i)$ can be directly derived.

The analysis of the sensor effects of a callback method m considers m and its transitive callees in the app code. If any of these 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 $\mathsf{acquire}(s_k)$. Callbacks `onCheckedChanged` in Fig. 1 and `onCreate` in Fig. 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 $\mathsf{release}(s_k)$. Callback `onDestroy` in Fig. 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 $\mathsf{release}(s_k)$, $\mathsf{acquire}(s_k)$.

We also need to consider callback `onSensorChanged` (illustrated at lines 21–22 in Fig. 1). Whenever a listener object 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 $\mathsf{acquire}(s_k)$ 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 $\mathsf{acquire}(s_k)$ symbol in SG. For the example in Fig. 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 Fig. 2 would be labeled with acquire(s), release(s).

3.6 Implementation

Test generation was implemented in the Soot framework (Soot 2017). The starting point of the implementation is the publicly available GATOR analysis toolkit for Android (GATOR 2017) which contains an implementation of the WTG representation (Yang et al. 2015b). 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 3.5. The propagation of integer constants and object references, which is needed to determine the set of s_k abstractions as well as the program statements that acquire/release them, is performed in a flow/context-insensitive manner, based on an internal representation similar to the pointer assignment graph used in Soot (Lhoták 2002). To compute acquire/release effects, a method's callees are determined using class hierarchy analysis (Dean et al. 1995) and Soot's control-flow-graph representation for each callee is used during the interprocedural traversal outlined in the section above.

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 UI Automator API calls (He 2018). Widgets are referenced using their ids defined in XML layout files or in `setId` calls in the code, as determined by GATOR (Rountev and Yan 2014; GATOR 2017). 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.

Activities are the primary windows in Android apps. However, the UI also allows for menus and dialogs, which are windows used to provide helper functionality for an activity. For example, in Fig. 1, the developer could have chosen to add a dialog window that is opened when `btn`'s button is pressed, in order to ask for confirmation that the vault should be locked. Upon user confirmation, the dialog's event handler would have started the sensor event listener. In general, menus and dialogs have their own widgets, UI event handlers, and lifecycle callbacks. Our implementation handles all these features; for example, these callbacks are analyzed to determine acquire/release symbols as described earlier. Languages P_1 and P_2 are only considered for activities because menus and dialogs are short-lived and a sensor they have acquired on behalf of some activity may be still active past their lifetime.

In the presence of menus and dialogs, the filtering of traversed paths (Section 3.3) needs to be generalized. Recall that when considering $L_1(w_i)$ and $L_2(w_i)$ paths, acquire(s_k) is ignored if w_i is not responsible for acquiring s_k. It is possible that a menu or a dialog was the actual active window that acquired s_k on behalf of activity w_i. In this case w_i is the last still-opened activity at the time when the menu/dialog was opened. Thus, when acquire(s_k) is encountered, our implementation will consider the top of the stack and will ignore the sequence of open(w_j) for menus and dialogs at the top of the stack. If the first activity-open symbol below this sequence is open(w_i), activity w_i is the one to be blamed for acquiring s_k, and thus acquire(s_k) is accounted for in the analysis of the corresponding $L_1(w_i)$ or $L_2(w_i)$ path. Otherwise, acquire(s_k) is ignored.

4 Generation of test cases for sensor leaks in Android Wear watch faces

In initial releases of the Android Wear (AW) platform, the emphasis was on a wearable device (e.g., smartwatch) that is paired with a companion handheld device (e.g., smartphone). As AW evolved, there was increased emphasis on standalone AW apps, for which there is no expectation for a companion handheld. Watch faces are one of the most popular categories of standalone AW apps (Zhang et al. 2018). There could be several watch faces installed on a smartwatch. The currently active watch face is selected using a standard watch face picker facility in the AW platform.

4.1 Watch face lifecycle

The lifecycle of a watch face is defined with respect to four states. State *Null* indicates that the watch face is not selected by the user to be displayed (that is, another watch face is currently active). In *Interactive* state, the watch face is selected to be active, is visible on the screen, and is responsive to user interactions. After a certain period of inactivity (5 seconds by default), or in response to certain user actions, the watch face can transition to state *Ambient*. In AW devices, there is a special mode to save battery life: *ambient mode*, in which users are not interacting with the device and the watch face only shows limited information in a battery-friendly manner, e.g., with lower resolution and fewer colors. Finally, state *Invisible* indicates that a watch face is active, but is not visible on the screen and thus is inaccessible for users, e.g., because it is covered by some launched AW app, because the watch face picker is invoked, or because a push notification is displayed.

4.2 Running example

State changes trigger various callbacks from the AW platform to the watch face code. To illustrate these callbacks, we use the example in Fig. 7. The example is extracted from The Hundreds watch face, which is available in the Google Play app store and has 50K–100K installs. (The Hundreds is an apparel and media brand.) The figure shows the decompiled code; non-essential details are elided.

The code defines a subclass of `CanvasWatchFaceService`. This superclass, defined in `Android.support.wearable.watchface`, provides a canvas on which the code can draw using Android painting APIs. Nested class `Engine` contains the implementation of the watch face: e.g., drawing hands on the screen, setting timers, fetching sensor data. The watch face lifecycle start and end are defined by callbacks `onCreate` and `onDestroy` declared in `Engine`. When the watch face enters ambient mode, callback `onAmbientModeChanged` is invoked by the AW platform with formal parameter `inAmbientMode` equal to `true`. Upon exiting ambient mode, the same method is called with a `false` parameter value. Similarly, callback `onVisibilityChanged` is invoked when the watch face becomes invisible (with parameter `visible` equal to `false`) and again when it becomes visible (with `true` parameter).

In this example, helper class `WatchfaceController` maintains two fields. Field `mAmbientMode` records the parameter value for the last call to `Engine.onAmbientModeChanged`. Note that the call at line 8 uses the return value of `isInAmbientMode()` instead of parameter `inAmbientMode`. Helper method `isInAmbientMode` is defined in a superclass of `Engine` and returns a value which is the same as the last `inAmbientMode` value. Field `mVisibility` records the parameter

```
 1 class BReelWatchFaceService extends CanvasWatchFaceService {
 2   Engine onCreateEngine() { return new Engine(); }
 3   class Engine extends CanvasWatchFaceService.Engine {
 4     WatchfaceController mWatchfaceController;
 5     void onCreate() { mWatchfaceController = new WatchfaceController(); }
 6     void onDestroy() { mWatchfaceController.destroy(); }
 7     void onAmbientModeChanged(boolean inAmbientMode) {
 8       mWatchfaceController.setAmbientMode(isInAmbientMode()); }
 9     void onVisibilityChanged(boolean visible) {
10       mWatchfaceController.setVisibility(isVisible()); }
11     void onDraw(...) {...} } }

12 class WatchfaceController {
13   boolean mAmbientMode;
14   boolean mVisibility;
15   OrientationController mOrientationController;
16   WatchfaceController() {
17     mOrientationController = new OrientationController(); }
18   void setAmbientMode(boolean ambientMode) {
19     mAmbientMode = ambientMode;
20     if (mAmbientMode) mOrientationController.stop();
21     else              mOrientationController.start(); }
22   void setVisibility(boolean visibility) {
23     mVisibility = visibility;
24     if (mVisibility) mOrientationController.start();
25     else             mOrientationController.stop(); }
26   void destroy() { mOrientationController.stop(); } }

27 class OrientationController {
28   Sensor mSensor;
29   SensorManager mSensorService;
30   SensorEventListener mSensorEventListener = new SensorEventListener() { ... };
31   OrientationController() {
32     mSensorService = ... ;
33     mSensor = mSensorService.getDefaultSensor(Sensor.TYPE_ACCELEROMETER); }
34   void start() {
35     mSensorService.registerListener(mSensorEventListener, mSensor); }
36   void stop() {
37     mSensorService.unregisterListener(mSensorEventListener); } }
```

Fig. 7 Decompiled code from The Hundreds watch face

of the last call to `onVisibilityChanged`. The call at line 10 does not use directly the parameter value of `visible`, but rather an equivalent return value from `isVisible`.

Class `OrientationController` is used to manage the watch face's use of the device's accelerometer sensor. At initialization, an instance of this class obtains the sensor. Upon state changes, the listener is registered and unregistered. The unregistration (line 37) is needed for energy efficiency reasons: the AW developer guidelines recommend that whenever the watch face enters ambient mode, sensors are turned off to allow the device to enter low-power mode and avoid battery drain. The code aims to identify transitions to state *Ambient* (i.e., `inAmbientMode` at line 7 is `true`) and stop the sensor. Similarly, transitions to state *Invisible* (i.e., `visible` at line 9 is `false`) stop the sensor.

Despite these efforts to follow the guidelines, the code contains a logical error. One possible run-time behavior is a transition from *Interactive* to *Invisible* and from there to *Ambient*. This could happen, for example, when the user opens a push notification (which transitions from *Interactive* to *Invisible*) but does not do anything for 5 seconds (which automatically transitions to *Ambient*). The sequence of callbacks is `onVisibilityChanged(false)`, `onAmbientModeChanged(true)`, `onVisibilityChanged(true)`. The last callback occurs because in AW ambient mode is considered to be a visible (low-power) state. For this callback sequence, the sensor is deactivated but then re-activated and remains active in ambient mode, in clear violation of AW guidelines. The underlying problem is the condition at line 24: the correct condition is `mVisibility && !mAmbientMode`.

4.3 Sensor effects control-flow graph for watch faces

Recall that our test generation for regular Android apps employs the SG static model. In this section we propose an equivalent model for Android Wear watch faces. Given this definition, the remainder of our approach for regular Android apps can be directly applied to AW watch faces: the definition of leak patterns is based on the same context-free languages described earlier, and the test generation is again structured as path traversals of this model. While inspired by our earlier work on leak testing for watch faces (Zhang et al. 2018), the proposed SG-based formulation is completely different from the one in that prior work, which was essentially a gen-kill formalism in the style of dataflow analysis. This re-formulation captures the inherent similarities between our test generation for regular Android apps and AW watch faces, and allows the same path traversal machinery for test generation to be employed in both scenarios.

We first present the components of the SG model that include only open and close symbols. This part of the model is common for all watch faces. We use *wf* to denote any watch face in the rest of the paper. The sensor effects, encoded by acquire and release symbols, depend on the code inside callbacks such as `onVisibilityChanged` and `onAmbientModeChanged` and are specific to each individual watch face.

Figure 8 shows the proposed SG model. Each node in the model corresponds to one of the watch face lifecycle states described earlier. Event open(*wf*) corresponds to the selection of the watch face via the watch face picker. The corresponding event close(*wf*) indicates that the user selected another watch face to be displayed, and the current watch face was closed.

Once the watch face is opened, it transitions to state *Interactive*. There are two ways to then transition to *Invisible*: either the list of apps on the watch is opened to select an app for

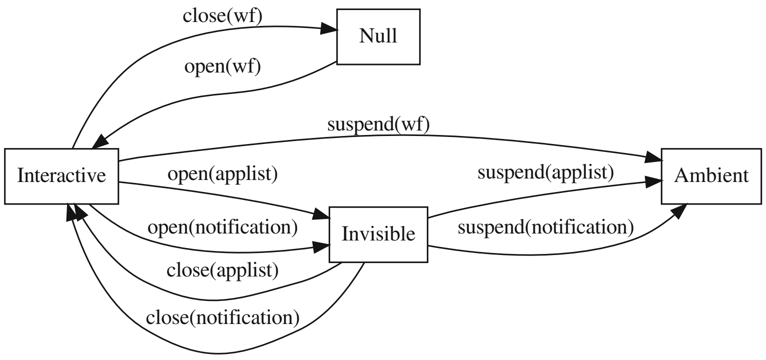


Fig. 8 *SG* model for watch faces

execution, represented by symbol open(*applist*), or a push notification is opened to examine its content, represented by symbol open(*notification*). In either case, a transition back to state *Interactive* could occur, as denoted by symbols close(*applist*) and close(*notification*).

Transitions to state *Ambient* are denoted by suspend symbols. Whenever there is user inaction for more than 5 seconds, the watch face automatically transitions to the low-power ambient state. In general, the majority of time is spent in this state, as the interactions of the user with the watch are supposed to be short and relatively infrequent. In the absence of such interactions, the default state is *Ambient*.

Each transition in the model from Fig. 8 is triggered by certain events and triggers various callbacks. Table 1 shows this information. For simplicity, we do not show additional events that are equivalent to the shown ones and do not contribute any new information to the model.

Static analysis of the callback sequences shown in the table can be used to determine symbols acquire and release for the transitions in the model. The static analysis used for regular Android apps, as described in Section 3.5, can be used here as well. However, a refinement is needed in the analysis of callbacks. Recall that `onVisibilityChanged` and `onAmbientModeChanged` take as input boolean parameters. The values of these parameters define four possible invocation contexts, one for each combination of "invisible on/off" and "ambient on/off". Here "on/off" refers to the return (boolean) values of internal APIs `isInAmbientMode` and `isVisible`, illustrated at lines 8 and 10 in Figure 2. Whenever `onAmbientModeChanged` is invoked with `true`, subsequent calls to `isInAmbientMode` return `true`, until `onAmbientModeChanged` is called again with a `false` parameter. There is a similar relationship between `onVisibilityChanged` and `isVisible`. This allows the analysis to determine possible effects of code that queries the watch face state (e.g., as done in the running example at lines 8 and 10). Specifically,

Table 1 Events and callbacks for watch face state transitions

State transition	GUI event	Callbacks
open(*wf*)	Select watch face	`wfsOnCreate()`, `wfsOnCreateEngine()`, `onCreate()`, `onVisibilityChanged(true)`
close(*wf*)	Deselect watch face	`onVisibilityChanged(false)`, `onDestroy()`, `wfsOnDestroy()`
open(*applist*)	Press side button	`onVisibilityChanged(false)`
close(*applist*)	Swipe right or press side button	`onVisibilityChanged(true)`
open(*notification*)	Swipe up	`onVisibilityChanged(false)`
close(*notification*)	Swipe down or press side button	`onVisibilityChanged(true)`
suspend(*wf*)	Standby	`onAmbientModeChanged(true)`
suspend(*applist*)	Standby	`onAmbientModeChanged(true)`, `onVisibilityChanged(true)`
suspend(*notification*)	Standby	`onAmbientModeChanged(true)`, `onVisibilityChanged(true)`

the context information is used to resolve branches that depend on the state, as illustrated by the checks at lines 20 and 24 in Fig. 8.

For the running example, we have acquire for `onVisibilityChanged(true)` and `onAmbientModeChanged(false)` and release for `onVisibilityChanged(false)`, `onAmbientModeChanged(true)`, and `onDestroy()`. As a result, the final SG model is extended as shown in Fig. 9. For a path from *Interactive* to *Invisible* and from there to *Ambient*, the sensor is first unregistered in the transition to *Invisible*, unregistered again in `onAmbientModeChanged(true)`, and then re-activated in `onVisibilityChanged(true)`. As a result, the sensor remains active in ambient mode.

4.4 Test generation for watch faces

We define two patterns for sensor leaks similar to the ones described in Section 3.2. Specifically, consider the language $L_1(wf)$ that defines a string for the lifetime of a watch face. The string starts with open(wf) and contains an equal number of matching open and close symbols. For any sensor s_k, language $P_1(wf, s_k) = L_1(wf) \cap R(s_k)$ defines suspicious SG paths. Similarly, consider $L_2(wf)$ which starts with open(wf), ends with some suspend symbol, and contains a valid sequence of open and close symbols. The corresponding language $P_2(wf, s_k) = L_2(wf) \cap R(s_k)$ defines suspicious SG paths. For the running example, all leaky paths belong to language P_2. The shortest such paths are open(wf), acquire(acc), open$(applist)$, release(acc), suspend$(applist)$, release(acc), acquire(acc) and a similar path for *notification*.

It is important to note that the second pattern does not necessarily signify a problem with the app—in some scenarios, the app has to record sensor data even in ambient mode. However, in our studies we observed that typically this pattern *does* indicate unnecessary sensor usage, and the programmer should have released the sensor resource before entering ambient mode. The code in Fig. 7 exemplifies this problem: the programmer has indeed attempted to release the sensor, but did so incorrectly.

Test generation is performed similarly to the approach for regular Android apps described in Section 3.3. SG paths are traversed as before, including the restrictions that no edge is repeated along a path, and the number of open and close symbols is $\leq \ell$ (as before, we use

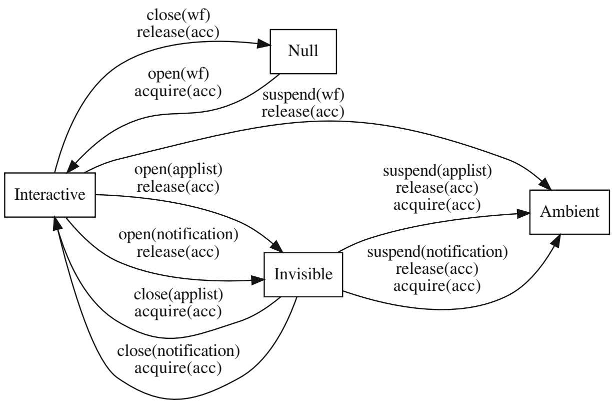


Fig. 9 Final *SG* model for the running example

$\ell = 4$). For each reported path, we generate a test case based on the events along the path. Event implementation is based on a wrapper for MonkeyRunner (Google 2018f) developed by us. Note that our earlier work on test generation for leaks in watch faces (Zhang et al. 2018) was based on a different implementation which did not employ an SG model. Our new implementation, which is now based on path traversals in the SG model, produces equivalent results.

5 Test execution

For regular Android apps, the generated test cases are executed using a Python wrapper for UI Automator (Google 2017b; He 2018). This framework allows testing to be controlled from a computer with direct access to the ADB (Google 2018a), which is necessary for run-time sensor 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 Fig. 6. 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. Out of the 18 Android apps used in our study, 5 required such manual steps, as detailed in the public release of our implementation and benchmarks.

Acquired sensors with information about listener's package names and sensor types can be queried using the `dumpsys` command (Google 2018d) in ADB. Each generated test case performs this measurement at the start and at the end of its execution (recall `readAssociatedSensors` from Fig. 6). If a sensor is not active 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 sensors on a Google Nexus 5X smartphone with Android 7.1.2.

For watch faces, test cases are generated and executed based on a wrapper for MonkeyRunner (Google 2018f) developed by us. The underlying mechanism is similar to the approach for regular Android apps. Each transition is mapped to an API call to the wrapper (e.g., `swipe_up` and `press_side_button`). The testing infrastructure enables control of wearables via ADB to retrieve information about sensors. The same helper function `readAssociatedSensors`, illustrated in Fig. 6, is used to record active sensors on the wearable device. In the experiments, test execution was conducted on an LG Watch Style smartwatch running Android Wear 2.9.

6 Evaluation

The goal of the experimental evaluation is to (1) investigate the effectiveness of SENTINEL for sensor leak detection in Android apps and Android Wear watch faces, and (2) quantify the accuracy and cost of the code analysis and test generation. Specifically, we define the following research questions:

- **RQ1**: Does SENTINEL discover true sensor leaks in real-word apps and watch faces?
- **RQ2**: What is the accuracy of the static analysis in SENTINEL and how well does it reduce the number of generated test cases?
- **RQ3**: Is the cost of test generation in SENTINEL suitable for practical use?

To address RQ1, we use the following metric: number of observed run-time leaks exposed by running the test cases generated by SENTINEL. The subjects for this experiment are Android apps and Android Wear watch faces collected from public repositories (e.g., top popular apps from Google Play). The selection process for these experimental subjects will be described shortly. The experimental procedure is to apply our implementation of SENTINEL to generate test cases for a subject, then execute these test cases, and finally determine whether sensor leaks are observed at run time. To answer RQ2, we use the same subjects and report the following metrics: (1) reduction in the number of generated test cases, and (2) number of false positives. For the first metric, we execute our implementation of the static code analysis and measure the number of relevant paths reported by it. For the second metric, we observe the run-time execution of a generated test case and determine whether it does not trigger a run-time leak. Finally, for RQ3 we measure the running time of the analyses in SENTINEL on the same subjects. The implementation and all subjects are available at https://presto-osu.github.io/Sentinel.

6.1 Test cases for sensor leaks in Android apps

Study subjects 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. We then determined which of these apps contained sensor listeners. The evaluation of SENTINEL was performed on the entire set of such apps (a total of 709 apps). The static analysis identified 18 apps for which the code exhibited the sensor leak patterns described earlier. Tables 2 and 3 show measurements for these apps. The first six apps are from F-Droid (open-sourced, available along with the

Table 2 Applications and their sizes

	App size				
Application	Class	Stmt	Node	Edge	Time (s)
`Mtpms`	37	3148	20	38	0.1
`Drismo`	325	24592	425	645	1.2
`Geopaparazzi`	1467	149469	283	534	2.6
`Itlogger`	296	30516	30	77	0.6
`AIMSICD`	921	79438	59	137	0.8
`Coregame`	44	1988	3	7	0.1
`NightVisionCamera`	408	44399	74	93	0.6
`Voxofon`	2637	184406	451	1084	7.4
`VRVideoPlayer`	334	24296	13	29	0.5
`MobinCube`	502	67186	975	1165	16.9
`CSipSimple`	1319	111659	57	223	9.4
`Calculator Vault`	2025	137364	220	694	137.7
`Comebacks`	160	21246	67	96	0.3
`Pushups`	956	87545	627	1401	2.9
`Dogwhistier`	2415	181626	125	302	48.1
`Hideitpro`	3315	227087	807	1610	10.5
`LikeThatGarden`	1678	115092	634	1748	72.2
`MyMercy`	907	74914	258	629	11.7

Table 3 Paths and tests

	L paths		$L \cap R$ paths		Tests		Leaks	
Application	L_1	L_2	P_1	P_2	P_1	P_2	P_1	P_2
`Mtpms`	7	8	6	6	1	1	1	1
`Drismo`	1740	719	320	512	1	1	0	1
`Geopaparazzi`	615	721	24	24	1	1	1	1
`Itlogger`	24	44	7	24	1	1	1	1
`AIMSICD`	73	72	4	2	1	1	1	1
`Coregame`	2	2	2	2	1	1	1	1
`NightVisionCamera`	5	25	0	16	0	1	0	–
`Voxofon`	4011	994	0	2	0	1	0	1
`VRVideoPlayer`	5	8	2	2	1	1	0	0
`MobinCube`	12735	83209	3155	3907	3	5	0	0
`CSipSimple`	750	402	0	100	0	1	0	1
`Calculator Vault`	49064	13837	1128	3108	2	2	2	2
`Comebacks`	46	50	0	16	0	1	0	–
`Pushups`	13731	2364	0	2	0	1	0	0
`Dogwhistier`	2257	1739	25	27	1	1	1	1
`Hideitpro`	5171	1460	1058	1347	2	2	2	2
`LikeThatGarden`	575773	201332	8165	21425	1	1	1	1
`MyMercy`	286	400	0	5	0	1	0	–

SENTINEL tool) and the rest are from Google Play (close-sourced, many are obfuscated). In Table 2, 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.

Test generation time, in seconds, is shown in column "Time" in Table 2. 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, which provides a positive answer to RQ3.

6.1.1 Test generation and run-time leaks

Columns two to five of Table 3 show measurements for the number of *SG* paths. Under "L paths" are included the number of paths from languages L_1 and L_2 (Section 3.2), with the restrictions described in Section 3.3: path length limited by parameter $\ell = 4$ and without duplicated edges along a path. 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 three filtering techniques (Section 3.3). 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 provide an answer to one component of RQ2 and 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. It is a common practice that an application has a template activity and some other activities are subclasses of this activity. If this parent activity has a defect, all of its subclasses will have the same defect. Therefore, when we report the results for columns "Tests" and "Leaks," we exclude test cases and leaks caused by the subclasses of the same defective parent activity class. 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 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.

For our research questions, these experiments provide the following insights. First, for RQ1, we demonstrate that sensor leaks are discovered in real-world Android apps. For RQ2, in 4 out of 18 apps there are false positives: statically determined leaks are not confirmed by the corresponding run-time test execution. We provide details on these false positives later in the paper. Given these results, we conclude that the analysis exhibits good but not perfect accuracy. Of course, there are various threats to these conclusions, as discussed at the end of this section.

6.1.2 Manual investigations of app code

Next, we briefly present additional details on several analyzed apps. These details were obtained via manual investigation of app code. For F-Droid apps, we considered the publicly available source code. For Google Play apps, we used the `jadx` decompiler (Skylot 2018) to study the app code. While these details do not directly help to answer our research questions, they provide additional context and insights about the detected leaks and the performance of SENTINEL.

CSipSimple This VoIP app was illustrated in Fig. 4. When there is an incoming or an outgoing call, Activity `InCallActivity` will be started by an Android intent broadcast. 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 10 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

```
 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(); } }
```

Fig. 10 Example derived from Geopaparazzi

`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 `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 four apps, the test cases do not produce run-time leaks. In first 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. Unfortunately, this problem is pervasive in static analysis of Android code. For general object-oriented code there are many options (e.g., rapid type analysis (Bacon and Sweeney 1996) and 0-CFA (Grove and Chambers 2001; Shivers 1991)) for more precise call graph construction (Ryder 2003). The framework/callback-driven control-flow in Android apps present significant new challenges and existing approaches such as FlowDroid (Arzt et al. 2014) and GATOR do not present sound call graphs, as demonstrated elsewhere (Wang et al. 2016).

In the last case, the false positive in `Drismo` is caused by a limitation in the WTG construction of GATOR. In `Drismo`, the defective activity acquires a sensor resource in `onCreate` and releases it in the `onBackPressed` callback, which is the event handler callback for the hardware BACK button. However, the `onBackPressed` callback is not considered in GATOR. Therefore, SENTINEL cannot detect the release effects in this `onBackPressed` callback, causing a false positive. If this WTG limitation is eliminated, this false positive will be eliminated as well.

The results in Tables 2 and 3 differ slightly from the ones in an earlier published version of this work (Wu et al. 2018). Some differences are due to minor changes in the underlying GATOR tool: for example, revised handling of ⟨activity-alias⟩ tags in XML files, which allows for more comprehensive representation of app structure. As a result, for several apps the numbers of WTG nodes/edges and explored paths are different from the previously published results. For all but one app, these changes do not affect the number of generated test cases or confirmed leaks. In addition, one difference in the number of reported leaks is due to an inaccuracy in our manual investigation of run-time leaks.

6.2 Test cases for sensor leaks in Android Wear watch faces

Study subjects We collected a set of 1490 Android Wear watch faces that were free and available in an unrestricted Play mirror (APKPure 2018), and identified the ones that contained sensor listeners. Here we only consider unobfuscated watch faces and the ones in which the Android Wear support library is unobfuscated, since our implementation identifies relevant APIs based on their signatures. The resulting set of 58 watch faces was used for the evaluation. The analysis identified 31 watch faces with instances of the sensor leak patterns, excluding the ones that are reported but use sensors that are not available on the LG Watch Style smartwatch used in our experiments. Details about these apps are presented in Table 4.

6.2.1 Test generation and run-time leaks

Columns “Class” and “Stmt” contain the total number of classes and statements in Soot’s IR, respectively. Column “Time” shows the time in seconds for running the static analysis and test generation. Columns “$L \cap R$ paths” show the numbers of paths that match the two leak patterns from languages P_1 and P_2, with limit parameter $\ell = 4$ (Section 4.4). These paths are directly used to generate test cases without any filtering as the numbers are relatively small. Since the *SG* model for all watch faces is the same with respect to open and

Table 4 Watch faces, paths, and tests

Watch face	App size		$L \cap R$ paths		Leaks		Time (s)
	Class	Stmt	P_1	P_2	P_1	P_2	
`Analog Glow Lite`	46	3631	0	2	0	2	2.41
`Animated Earth`	71	6635	0	2	0	2	2.92
`Beautiful Rhinestone`	40	2701	0	2	0	2	1.59
`BLiS`	68	4523	1	0	1	0	2.45
`Bokeh`	127	5390	1	0	1	0	1.90
`BMW CSL`	96	6193	1	0	1	0	3.66
`BMW M1`	96	6098	1	0	1	0	3.94
`Ceres`	26	2781	0	2	0	0	1.88
`Christmas Counter`	26	1596	1	3	1	3	1.94
`Coubertin Rings`	45	4100	1	3	1	3	4.61
`CryptClock`	46	2055	0	3	0	3	1.60
`Date Stamp`	48	4240	0	2	0	2	2.68
`Diamond`	37	2939	0	2	0	2	1.99
`Diet`	65	10236	1	3	1	3	4.78
`Illusion`	27	2983	0	2	0	2	1.86
`KGB`	59	3777	1	3	1	3	4.86
`Many Icons`	40	3471	0	2	0	2	2.19
`Meo`	32	1277	0	3	0	3	1.18
`Paranormal`	39	2089	0	2	0	2	1.52
`Rambler`	560	70133	0	3	0	3	22.4
`Scuba`	280	15910	1	3	0	0	20.3
`Sensor Analog`	44	3043	0	3	0	3	1.82
`Snow Watch`	18	613	0	3	0	3	0.84
`Speeds`	214	14874	1	3	0	0	15.6
`The Hundreds`	50	3444	0	3	0	3	2.77
`Time Mesh`	48	10856	0	3	0	3	1.86
`TrombT1 Pearl`	21	1023	1	0	1	0	1.06
`Turbo Interactive`	173	30112	0	3	0	3	20.7
`Ultron Interactive`	173	37112	0	3	0	3	18.5
`YT1300 model 101.B`	41	2932	1	3	1	3	3.31
`YT1300 model 102`	42	4406	1	3	1	3	3.81

close symbols, the number of L_1 and L_2 paths is the same for any watch face and thus the table does not show columns "L paths" that are shown in the earlier table Table 2 for regular Android apps. Columns "Leaks" show the number of test cases that resulted in observed run-time leaks during execution.

In our experiments, a total of 13 watch faces are reported to have sensor unreleased when they are inactive and destroyed (P_1). This means that the developer forgot to unregister sensor listeners in `onDestroy`, `wfsOnDestroy`, or `onVisibilityChanged(false)`. For 11 out of the 13 reports, the test cases exposed unreleased sensors. The analysis reports 26 instances for P_2. Usually, this means that the watch face attempted to unregister the sensor listener in an incorrect way—e.g., some cases were missed for transitions to the power-saving ambient mode, as illustrated by the insufficient check at line 24 in the running example. This is likely caused by programmers' misunderstanding of the watch face lifecycle. Using test execution, we confirmed 23 of the 26 reports. With respect to our research questions, these results are consistent with the conclusions reached earlier for regular Android apps. For RQ1, we confirm that real sensor leaks are discovered in many of the analyzed watch faces. For RQ2, the conclusion is that the analysis exhibits a small number of false positives. For RQ3, the cost of test generation is confirmed to be low.

6.2.2 Manual investigations of watch face code

Bokeh This watch face contains an instance of P_1. It acquires a gravity sensor to guide the movement of the background image, similarly to a live wallpaper in regular Android. There is only one implementation of `SensorEventListener`. Registrations for the gravity sensor occur in `onAmbientModeChanged(false)` and `onCreate`. Every time the watch face enters ambient mode, the gravity sensor is released by an unregistration in `onAmbientModeChanged(true)`. No other places have calls to `(un)registerListner`. The developer intentionally did this to avoid unnecessary sensor acquisition as there is no animation in ambient mode. However, when the watch face is deselected, no release operation is performed during the transition from *InvisiblePicker* to *Null* and *Interactive* to *InvisiblePicker*. Thus, the sensor remains active and drains the battery.

Test cases without leaks We observed three apps where the test case did not produce a run-time leak. First, in the `Ceres` watch face (reported as an instance of P_2) a call to `isInAmbientMode` is performed inside `onDraw`, and sensor listener unregistration is performed in ambient mode. Our analysis of sensors does not consider this callback. However, according to AW guidelines, the system *"calls the Engine.onDraw() method every time it redraws your watch face, so you should only include operations that are strictly required to update the watch face inside this method"* (Google 2018g). Since `onDraw` is called much more frequently than the lifecycle callbacks, a better design is to move the release of sensors outside of `onDraw`. The other two examples are `Scuba` and `Speeds`. They both maintain an internal state machine, using custom `enum`s to represent the state of the watch face. This state is then used to correctly acquire and release the sensors. Our analysis does not model the effects of these internal states and state transitions.

6.3 Limitations of proposed approach

Our static analysis and testing is based on GUI events. Any leaks that are not GUI related cannot be detected by SENTINEL. Our implementation based on the GATOR tool suffers from the inherent imprecision of GATOR's static analysis and modeling. Another issue is that the tool does not support analysis of native code, making detection of leaks at the C/C++ level impossible. Although there are no existing related approaches specific for Android Wear, our experiments include no comparison with other approaches and tools for energy leak detection in Android apps, such as GreenDroid (Liu et al. 2014) (which uses dynamic rather than static analysis). The specific leak patterns we consider are similar to ones that appear in leaks of other resources (e.g., Pathak et al. 2012; Liu et al. 2013, 2014, 2016;

Banerjee et al. 2014, 2016; Wu et al. 2016a; Banerjee and Roychoudhury 2016), but they may not be representative of the most prevalent sensor leak defects in real apps and watch faces.

6.4 Threats to validity

The scale of the experiments poses a threat to the generalizability of the conclusions that SENTINEL can effectively discover true sensor leaks (RQ1) and exhibits high accuracy (RQ2) and low cost (RQ3). Factors such as programming style and application complexity may affect the presence of sensor leaks and the results of our analysis, but we have no evidence that we have a representative sample with respect to such factors. To address this threat, we have included the entire sets of apps and watch faces from two open repositories, as well as all top-100-per-category apps from Google Play. In addition, the watch faces used for evaluation are all unobfuscated, as discussed in Section 6.2. Thus, they may not be representative of the entire population, especially for commercial apps. This introduces a threat to the external validity of the study conclusions.

The implementation of the static analysis and test generation introduces a threat to internal validity: if the implementation is incorrect, the results of the study may be invalid. To ameliorate this threat, we performed extensive testing and compared analysis results against expected results we inferred manually from app code. The inability to execute some test cases (Section 6.1.2) presents a treat to internal validity, as leak detection cannot be quantified in such cases. Our manual investigation of decompiled code, for the purposes of obtaining additional insights, may also introduce threats to the validity of the derived conclusions since the logic of such decompiled code is difficult to understand.

6.5 Lessons learned

Even though Google's guidelines suggest to *"be sure to unregister a sensor's listener when you are done using the sensor or when the sensor activity pauses"* (Google 2018c), we find in our experiments that many developers forget to release sensors. We also find cases that they unregister sensors incorrectly due to the misunderstanding of the lifecycle of Android and Android Wear GUIs. The results also show that, for the analyzed apps and watch faces, effective test cases for sensor leaks can be generated based on static control-flow analysis. With the help of test execution and run-time monitoring, it is (almost) effortless to validate possible leaks. The fact that the proposed approach can be applied to both regular Android apps and Android Wear watch faces shows the generality of SENTINEL. We have submitted the findings for one watch face which we could identify as open-sourced;[4] however, it appears that this project is currently inactive and our results have not been confirmed by the developers.

7 Related work

Analysis and testing for Android and wear There are various techniques for automated testing for mobile apps (Choudhary et al. 2015; Li et al. 2017a; Sadeghi et al. 2017; Linares-Vásquez et al. 2017). The Monkey testing tool (Google 2017a) generates UI events

[4]https://github.com/fathominfo/fathom-watchfaces/issues/53

randomly. Many other techniques aim at more systematic testing, based on some model of the app. Android GUI Ripper (Amalfitano et al. 2012) and MobiGUITAR (Amalfitano et al. 2015) generate test cases based on dynamically built GUI models. Yang et al. (2013) explore the application dynamically, but use static analysis to determine relevant UI elements. The A^{3}E GUI exploration tool (Azim and Neamtiu 2013) employs both dynamic depth-first exploration, similarly to Android GUI Ripper, as well as targeted exploration based on a model derived from static analysis. Work by Zhang et al. (2016) 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. (2013) generate event sequences using a UI model and event handler summaries derived from static analysis. CrashScope (Moran et al. 2016) uses a model-based approach for detecting and reporting run-time crashes.

A variety of other testing approaches have been studied for Android. Li et al. (2017b) consider the evolution of GUI test scripts for mobile apps. Fazzini et al. (2017) proposed an approach for platform-independent test scripts for Android apps. Sapienz (Mao et al. 2016; Alshahwan et al. 2018) uses search-based testing to explore test sequences. Garcia et al. (2017) leverage symbolic execution to generate inter-component communication exploits. ACTEve (Anand et al. 2012) performs concolic testing by symbolically tracking UI events from their origin to their handler. There are also examples of using machine learning techniques to improve automated test generations. SwiftHand (Choi et al. 2013) achieves code coverage by learning and exploring an abstraction of the app's GUI. Zhang and Rountev (2017) propose static analysis for generation of user inputs to test push notifications in Android Wear. Grano et al. (2018) designed an automated test generation tool based on machine learning from users' reviews. Google recently announced their automated testing service (Google 2018e). Examples of other relevant tools are Axiz (Mao et al. 2017), Dynodroid (Machiry et al. 2013), EvoDroid (Mahmood et al. 2014), FlowDroid (Arzt et al. 2014), GATOR (Yang et al. 2015b; Yang et al. 2018; Yang et al. 2015a), PATDroid (Sadeghi et al. 2017), PUMA (Hao et al. 2014), and TrimDroid(Mirzaei et al. 2016).

Leak detection and energy analysis There is a large body of work on leak detection and energy analysis for Android. Some approaches use run-time analysis. GreenDroid (Liu et al. 2014) detects energy-related resource underutilization and leaks using Java PathFinder, based on a hybrid UI model. A similar approach (Banerjee et al. 2014) uses a modified version of Dynodroid (Machiry et al. 2013) to perform dynamic GUI ripping, followed by analysis of energy-related leaks either dynamically (Banerjee et al. 2014; Banerjee et al. 2016) or using a hybrid dynamic/static approach (Banerjee and Roychoudhury 2016). Ma et al. (2017) developed a dynamic leak detector based on UI traversal and memory profiling.

Static analysis has also been employed for leak detection. Pathak et al. (2012) developed a static approach for detection of energy bugs based on a simplified control-flow model. Static analysis of missing-deactivation code patterns has been used to uncover leaks of GPS listeners (Wu et al. 2016a). Our approach employs similar patterns and callback analyses for sensor listeners, but uses a novel sensor effects control-flow model. SAAD (Jiang et al. 2017) generates callback sequences of Android application in order to identify unreleased resources. Wu et al. (2016b) developed a tool which can perform interprocedural analysis on Android applications to statically identify potential leaks. Unlike these techniques, our approach uses a static analysis for the purposes of focusing the testing efforts on a small subset of possible run-time behaviors.

Energy-related behaviors for Android have also been considered in other contexts. Jabbarvand et al. (2016) developed an approach to minimize the number of tests needed to uncover energy bugs. Follow-up work on μDroid (Jabbarvand and Malek 2017) defines a mutation testing approach to evaluate the ability of a test suite to reveal energy inefficiencies. Cruz and Abreu (2017) studied the effects of performance-based guidelines and practices on Android energy consumption, and highlighted the need for energy-aware techniques.

There are several studies of energy use in wearable devices. Min et al. (2015) present an exploratory investigation of users' expectations, interactions, and charging behaviors when using smartwatches. Poyraz and Memik (2016) collect activities of 32 smartwatch users in 70 days. They propose a power model to analyze the characteristics of user behaviors, power consumption, and network activities. Liu et al. (2017) investigate the usage of push notifications, apps, and network traffic for a comprehensive power model. Their findings highlight the power consumption in ambient/dozing mode because of its long duration. While these studies are general AW characterizations, our work focuses on the detection of sensor-related energy inefficiencies of watch faces by static analysis and testing.

8 Conclusions and future work

This work demonstrates that is is possible to automatically generate effective tests for sensor leaks in Android apps and Android Wear watch faces. While there are many possible GUI event sequences for an app, sensor-aware static analysis can reduce dramatically the number of event sequences executed during testing. Our evaluation confirms the utility of the SENTINEL approach by exposing a large number of sensor leaks in realistic apps and watch faces.

The patterns of control-flow for apps are not specific to sensor leaks. Other defects can be defined based on some notion of "acquire" and "release" operations inside callbacks along paths of static control-flow models. For example, one could consider resources that should be tracked across sequences of callbacks, such as GPS, native memory, and `Bitmap` objects. The control-flow analysis in SENTINEL could be extended to consider other GUI components such as `Fragment`, as well as APIs for asynchronous jobs, e.g., `AsyncTask`. These generalizations could be used for more comprehensive analysis for detection of both senor leaks and other categories of problematic patterns of behaviors.

Acknowledgments We thank the SQJ, AST and ESEC/FSE reviewers for their valuable feedback.

Funding information 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

Alshahwan, N., Gao, X., Harman, M., Jia, Y., Mao, K., Mols, A., Tei, T., Zorin, I. (2018). Deploying search based software engineering with Sapienz at Facebook. In *SBSE* (pp. 3–45).

Amalfitano, D., Fasolino, A., Tramontana, P., De Carmine, S., Memon, A. (2012). Using GUI ripping for automated testing of Android applications. In *ASE* (pp. 258–261).

Amalfitano, D., Fasolino, A., Tramontana, P., Ta, B., Memon, A. (2015). MobiGUITAR: automated model-based testing of mobile apps. *IEEE Software*, *32*(5), 53–59.

Anand, S., Naik, M., Harrold, M., Yang, H. (2012). Automated concolic testing of smartphone apps. In *FSE* (pp. 1–11).

APKPure (2018). Apkpure: Free APKs online. https://apkpure.com.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P. (2014). FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI* (pp. 259–269).

Azim, T., & Neamtiu, I. (2013). Targeted and depth-first exploration for systematic testing of Android apps. In *OOPSLA* (pp. 641–660).

Bacon, D., & Sweeney, P. (1996). Fast static analysis of C++ virtual function calls. In *OOPSLA* (pp. 324–341).

Banerjee, A., & Roychoudhury, A. (2016). Automated re-factoring of Android apps to enhance energy-efficiency. In *MOBILESoft* (pp. 139–150).

Banerjee, A., Chong, L., Chattopadhyay, S., Roychoudhury, A. (2014). Detecting energy bugs and hotspots in mobile apps. In *FSE* (pp. 588–598).

Banerjee, A., Guo, H., Roychoudhury, A. (2016). Debugging energy-efficiency related field failures in mobile apps. In *MOBILESoft* (pp. 127–138).

Choi, W., Necula, G., Sen, K. (2013). Guided GUI testing of Android apps with minimal restart and approximate learning. In *OOPSLA* (pp. 623–640).

Choudhary, S., Gorla, A., Orso, A. (2015). Automated test input generation for Android: are we there yet? In *ASE* (pp. 429–440).

Corral, L., & Fronza, I. (2015). Better code for better apps: a study on source code quality and market success of Android applications. In *MOBILESoft* (pp. 22–32).

Corral, L., Sillitti, A., Succi, G. (205). Defining relevant software quality characteristics from publishing policies of mobile app stores. In *International Conference on Mobile Web and Information Systems*.

Cruz, L., & Abreu, R. (2017). Performance-based guidelines for energy efficient mobile applications. In *MOBILESoft* (pp. 46–57).

Dean, J., Grove, D., Chambers, C. (1995). Optimizations of object-oriented programs using static class hierarchy analysis. In *ECOOP* (pp. 77–101).

d'Heureuse, N., Huici, F., Arumaithurai, M., Ahmed, M., Papagiannaki, K., Niccolini, S. (2012). What's app?: a wide-scale measurement study of smart phone markets. *ACM SIGMOBILE Mobile Computing and Communications Review*, *16*(2), 16–27.

Fazzini, M., Freitas, E., Choudhary, S.R., Orso, A. (2017). Barista: a technique for recording, encoding, and running platform independent Android tests. In *ICST* (pp. 149–160).

Garcia, J., Hammad, M., Ghorbani, N., Malek, S. (2017). Automatic generation of inter-component communication exploits for Android applications. In *FSE* (pp. 661–671).

GATOR (2017). GATOR: Program analysis toolkit for Android. http://web.cse.ohio-state.edu/presto/software/gator.

Google (2017a). Monkey: UI/Application exerciser for Android. http://developer.Android.com/tools/help/monkey.html.

Google (2017b). UI Automator testing framework. http://developer.Android.com/training/testing/ui-automator.html.

Google (2018a). Android Debug Bridge (adb). https://developer.Android.com/studio/command-line/adb.

Google (2018b). Android Wear. http://developer.Android.com/wear.

Google (2018c). Best practices for accessing and using sensors. https://developer.Android.com/guide/topics/sensors/sensors_overview.html#sensors-practices.

Google (2018d). dumpsys. https://developer.Android.com/studio/command-line/dumpsys.

Google (2018e). Firebase test lab Robo test. https://firebase.google.com/docs/test-lab/Android/robo-ux-test.

Google (2018f). MonkeyRunner. https://developer.Android.com/studio/test/monkeyrunner.

Google (2018g). Optimizing watch faces: move expensive operations outside the drawing method. https://developer.Android.com/training/wearables/watch-faces/performance.html#OutDrawing.

Grano, G., Ciurumelea, A., Panichella, S., Palomba, F., Gall, H.C. (2018). Exploring the integration of user feedback in automated testing of Android applications. In *SANER* (pp. 72–83).

Grove, D., & Chambers, C. (2001). A framework for call graph construction algorithms. *TOPLAS*, *23*(6), 685–746.

Hao, S., Liu, B., Nath, S., Halfond, W., Govindan, R. (2014). PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *MobiSys* (pp. 204–217).

He, X. (2018). Python wrapper of Android UI Automator test tool. http://github.com/xiaocong/uiautomator.

Jabbarvand, R., & Malek, S. (2017). μdroid: an energy-aware mutation testing framework for Android. In *FSE* (pp. 208–219).

Jabbarvand, R., Sadeghi, A., Bagheri, H., Malek, S. (2016). Energy-aware test-suite minimization for Android apps. In *ISSTA* (pp. 425–436).

Jensen, C.S., Prasad, M.R., Møller, A. (2013). Automated testing with targeted event sequence generation. In *ISSTA* (pp. 67–77).

Jiang, H., Yang, H., Qin, S., Su, Z., Zhang, J., Yan, J. (2017). Detecting energy bugs in Android apps using static analysis. In *ICFEM* (pp. 192–208).

Lhoták, O. (2002). Spark: a scalable points-to analysis framework for Java. Master?s thesis, McGill University.

Li, L., Bissyandé, T.F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., Traon, L. (2017a). Static analysis of Android apps: a systematic literature review. *IST*, *88*, 67–95.

Li, X., Chang, N., Wang, Y., Huang, H., Pei, Y., Wang, L., Li, X. (2017b). Atom: automatic maintenance of GUI test scripts for evolving mobile applications. In *ICST* (pp. 161–171).

Linares-Vásquez, M., Moran, K., Poshyvanyk, D. (2017). Continuous, evolutionary and large-scale: a new perspective for automated mobile app testing. In *ICSME* (pp. 399–410).

Liu, X., Chen, T., Qian, F., Guo, Z., Lin, F.X., Wang, X., Kai, C. (2017). Characterizing smartwatch usage in the wild. In *MobiSys* (pp. 385–398).

Liu, Y., Xu, C., Cheung, S.C. (2013). Where has my battery gone? Finding sensor related energy black holes in smartphone applications. In *PerCom* (pp. 2–10).

Liu, Y., Xu, C., Cheung, S.C., Lu, J. (2014). Greendroid: automated diagnosis of energy inefficiency for smartphone applications. *TSE*, *40*, 911–940.

Liu, Y., Xu, C., Cheung, S., Terragni, V. (2016). Understanding and detecting wake lock misuses for Android applications. In *FSE* (pp. 296–409).

Ma, J., Liu, S., Yue, S., Tao, X., Lu, J. (2017). LeakDAF: an automated tool for detecting leaked activities and fragments of Android applications. In *COMPSAC* (pp. 23–32).

Machiry, A., Tahiliani, R., Naik, M. (2013). Dynodroid: an input generation system for Android apps. In *FSE* (pp. 224–234).

Mahmood, R., Mirzaei, N., Malek, S. (2014). EvoDroid: segmented evolutionary testing of Android apps. In *FSE* (pp. 599–609).

Mao, K., Harman, M., Jia, Y. (2016). Sapienz: multi-objective automated testing for Android applications. In *ISSTA* (pp. 94–105).

Mao, K., Harman, M., Jia, Y. (2017). Robotic testing of mobile apps for truly black-box automation. *IEEE Software*, *34*(2), 11–16.

Min, C., Kang, S., Yoo, C., Cha, J., Choi, S., Oh, Y., Song, J. (2015). Exploring current practices for battery use and management of smartwatches. In *ISWC* (pp. 11–18).

Mirzaei, N., Garcia, J., Bagheri, H., Sadeghi, A., Malek, S. (2016). Reducing combinatorics in GUI testing of Android applications. In *ICSE* (pp. 559–570).

Moran, K., Linares-Vasquez, M., Bernal-Cardenas, C., Vendome, C., Poshyvanyk, D. (2016). Automatically discovering, reporting and reproducing Android application crashes. In *ICST* (pp. 33–44).

Pathak, A., Jindal, A., Hu, Y., Midkiff, S. (2012). What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys* (pp. 267–280).

Poyraz, E., & Memik, G. (2016). Analyzing power consumption and characterizing user activities on smartwatches. In *IISWC* (pp. 1–2).

Reps, T. (1998). Program analysis via graph reachability. *IST*, *40*(11-12), 701–726.

Rountev, A., & Yan, D. (2014). Static reference analysis for GUI objects in Android software. In *CGO* (pp. 143–153).

Ryder, B. (2003). Dimensions of precision in reference analysis of object-oriented programming languages. In *CC* (pp. 126–137).

Sadeghi, A., Jabbarvand, R., Malek, S. (2017). PATDroid: Permission-aware GUI testing of Android. In *FSE* (pp. 220–232).

Shivers, O. (1991). Control-flow analysis of higher-order languages. PhD thesis, Carnegie Mellon University.

Skylot (2018). jadx: Dex to Java decompiler. https://github.com/skylot/jadx.

Soot (2017).

Wang, Y., Zhang, H., Rountev, A. (2016). On the unsoundness of static analysis for Android GUIs.

Wu, H., Yang, S., Rountev, A. (2016a). Static detection of energy defect patterns in Android applications. In *CC* (pp. 185–195).

Wu, H., Wang, Y., Rountev, A. (2018). Sentinel: generating GUI tests for Android sensor leaks. In *AST* (pp. 27–33).

Wu, T., Liu, J., Deng, X., Yan, J., Zhang, J. (2016b). Relda2: an effective static analysis tool for resource leak detection in Android apps. In *ASE* (pp. 762–767).

Yang, S., Yan, D., Wu, H., Wang, Y., Rountev, A. (2015a). Static control-flow analysis of user-driven callbacks in Android applications. In *ICSE* (pp. 89–99).

Yang, S., Zhang, H., Wu, H., Wang, Y., Yan, D., Rountev, A. (2015b). Static window transition graphs for Android. In *ASE* (pp. 658–668).

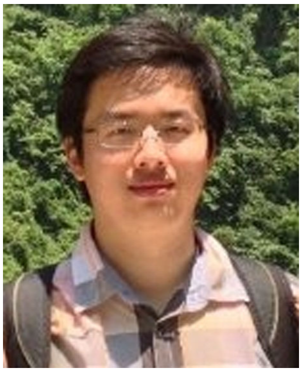
Yang, S., Wu, H., Zhang, H., Wang, Y., Swaminathan, C., Yan, D., Rountev, A. (2018). Static window transition graphs for Android. *JASE*, 1–41.

Yang, W., Prasad, M., Xie, T. (2013). A grey-box approach for automated GUI-model generation of mobile applications. In *FASE* (pp. 250–265).

Zhang, H., & Rountev, A. (2017). Analysis and testing of notifications in Android Wear applications. In *ICSE* (pp. 64–70).

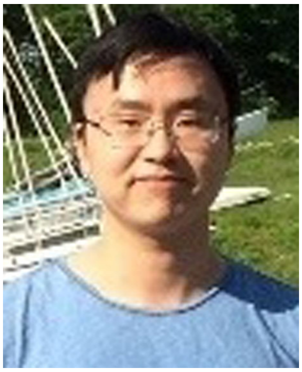
Zhang, H., Wu, H., Rountev, A. (2016). Automated test generation for detection of leaks in Android applications. In *AST* (pp. 64–70).

Zhang, H., Wu, H., Rountev, A. (2018). Detection of energy inefficiencies in Android Wear watch faces. In *FSE* (pp. 691–702).



Haowei Wu is an engineer at Google. He graduated from the Ohio State University in 2018. Before that, he earned his bachelor's degree in Information Security in Huazhong University of Science and Technology (HUST).

Hailong Zhang has been at the Department of Computer Science and Engineering of the Ohio State University since 2014. Before that, he studied at Beijing University of Posts and Telecommunications.



Yan Wang is an engineer at Google. He graduated from the Ohio State University in summer 2018. Prior to joining OSU, he earned his bachelor's degree in Software Engineering from Tongji University.

Atanas Rountev has been at Ohio State since 2002, becoming a tenured Associate Professor in 2008 and a Full Professor in 2015. Before that, he spent several years in New Jersey working on his Ph.D. at Rutgers University.