Detection of Energy Inefficiencies in Android Wear Watch Faces

Hailong Zhang
Ohio State University
Columbus, Ohio, USA
zhang.4858@osu.edu

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

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

ABSTRACT

This work considers *watch faces* for Android Wear devices such as smartwatches. Watch faces are a popular category of apps that display current time and relevant contextual information. Our study of watch faces in an app market indicates that energy efficiency is a key concern for users and developers.

The first contribution of this work is the definition of several energy-inefficiency patterns of watch face behavior, focusing on two energy-intensive resources: sensors and displays. Based on these patterns, we propose a control-flow model and static analysis algorithms to identify instances of these patterns. The algorithms use interprocedural control-flow analysis of callback methods and the invocation sequences of these methods. Potential energy inefficiencies are then used for automated test generation and execution, where the static analysis reports are validated via run-time execution. Our experimental results and case studies demonstrate that the analysis achieves high precision and low cost, and provide insights into potential pitfalls faced by developers of watch faces.

CCS CONCEPTS

• **Theory of computation** → *Program analysis*; • **Software and its engineering** → *Software testing and debugging*;

KEYWORDS

Android Wear, smartwatch, energy, sensor, static analysis, testing

ACM Reference Format:
Hailong Zhang, Haowei Wu, and Atanas Rountev. 2018. Detection of Energy Inefficiencies in Android Wear Watch Faces. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236073

1 INTRODUCTION

Wearable devices are becoming increasingly popular. Annual sales of smartwatches are expected to double by 2021, reaching over 80 million devices [15]. Other wearables such as head-mounted displays, body cameras, and wrist bands exhibit similar trends. This popularity is driven by the enhanced mobility and range of activities supported by such devices, and by their ability to sense external conditions (e.g., temperature and GPS location) and user's physiological state (e.g., heart rate, perspiration, and range of movement).

The focus of our work is the *Android Wear* (AW) platform. While AW shares some components with "traditional" Android, its core features are different from those in the well-studied Android platform. For example, an AW app running on a wearable device has a lifecycle different from the standard Android lifecycle [17] and uses entirely different platform APIs. Existing techniques for analysis and testing of traditional Android apps cannot be applied directly to AW apps. Some prior work [69] has considered the analysis and testing of push notifications, where an Android app running on a handheld device (e.g., a smartphone) displays notifications on a wearable. However, we are not aware of any work focusing on *standalone* AW apps, in which the wearable operates independently from any handheld. Such apps presents the next generation of software for AW devices and their importance is being emphasized in the latest AW releases. The increasing use of standalone AW apps requires new research advances and tools to improve developer productivity and software quality, performance, and security.

An important category of standalone AW apps are *watch faces*. Our studies, discussed in the next section, indicate that watch faces are some of the most widespread examples of standalone AW apps: among the AW apps we examined, around 59% were watch faces. A watch face uses a digital canvas to display the current time and other contextual information. It has access to sensors, GPS, Bluetooth, networks, and data providers such as the user's agenda. Watch faces can be designed to be interactive, reacting to user's touch and performing tasks according to user-provided feedback. With the introduction of AW 2.0, the functionality of watch faces has become much richer than simply displaying time. In our case studies we observed watch faces that monitor and display information related to weather, light intensity, humidity, and air pressure, as well as user-specific information about steps taken, heart rate, exercise statistics, and daily agenda. We have performed an initial study of watch faces in Google Play, which helped us highlight trends in this area as well as the concerns of watch face users.

Based on this study, we have identified energy efficiency as one of the key considerations for AW watch faces. While building watch faces, developers have to follow various guidelines to achieve energy efficiency. Our case studies of watch face code, as well as our examination of user comments, indicate that these guidelines are sometimes violated, leading to watch faces that drain the battery. The first contribution of our work is the definition of several *energy-inefficiency patterns of watch face behavior*. These patterns focus on two energy-intensive resources: sensors and displays.

Based on these patterns, we propose an approach to identify instances of energy-inefficient behaviors in watch faces. The approach is based on three contributions. First, we propose a *static control-flow model* for watch faces, in order to capture possible

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236073

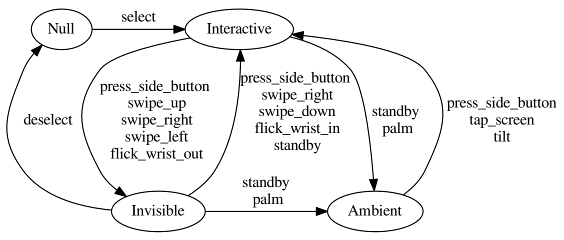

Figure 1: States, events, and transitions.

run-time sequences of events and the corresponding sequences of callbacks in the app code. While here we use this model for analysis of energy inefficiencies, the model itself is more general and could also be used for other categories of static analyses. Next, based on the model, we define *static analysis algorithms* based on interprocedural control-flow analysis of callback methods and the invocation sequences of these methods. Control-flow sequences that (statically) exhibit the inefficiency patterns are then used for *automated test generation and execution*, where the static analysis reports are validated via run-time execution.

The last contribution of this work is an *experimental study* of applying the proposed approach to a variety of watch faces. We demonstrate that the analysis achieves high precision and low cost. Our companion case studies explain the underlying causes of these inefficiencies and provide insights into potential pitfalls faced by developers of AW watch faces.

2 BACKGROUND AND APP MARKET STUDY

Several watch faces could be installed on an AW device, and only one of them is active at a time. The user selects the currently-active watch face using the "watch face picker" AW component.

2.1 Watch Face Lifecycle

The lifecycle of a watch face is defined with respect to four states, shown in Figure 1. 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.

While the AW documentation describes these states informally, it does not specify a detailed model for possible transitions between states. As a first step toward constructing such a model, we enumerated all possible user-triggered events e_j. For each of the four states s_i, we investigated the possible run-time transitions from s_i when e_j occurs. Figure 1 shows all possible transitions. Edges are labeled with events. A detailed discussion of possible events will

```
class BReelWatchFaceService extends CanvasWatchFaceService {
  Engine onCreateEngine() { return new Engine(); }
  class Engine extends CanvasWatchFaceService.Engine {
    WatchfaceController mWatchfaceController;
    void onCreate() { mWatchfaceController = new WatchfaceController(); }
    void onDestroy() { mWatchfaceController.destroy(); }
    void onAmbientModeChanged(boolean inAmbientMode) {
      mWatchfaceController.setAmbientMode(isInAmbientMode()); }
    void onVisibilityChanged(boolean visible) {
      mWatchfaceController.setVisibility(isVisible()); }
    void onDraw(...) {...} } }

class WatchfaceController {
  boolean mAmbientMode;
  boolean mVisibility;
  OrientationController mOrientationController;
  WatchfaceController() {
    mOrientationController = new OrientationController(); }
  void setAmbientMode(boolean ambientMode) {
    mAmbientMode = ambientMode;
    if (mAmbientMode) mOrientationController.stop();
    else              mOrientationController.start(); }
  void setVisibility(boolean visibility) {
    mVisibility = visibility;
    if (mVisibility) mOrientationController.start();
    else             mOrientationController.stop(); }
  void destroy() { mOrientationController.stop(); } }

class OrientationController {
  Sensor mSensor;
  SensorManager mSensorService;
 SensorEventListener mSensorEventListener = new SensorEventListener() { ... };
  OrientationController() {
    mSensorService = ... ;
    mSensor = mSensorService.getDefaultSensor(Sensor.TYPE_ACCELEROMETER); }
  void start() {
    mSensorService.registerListener(mSensorEventListener, mSensor); }
  void stop() {
    mSensorService.unregisterListener(mSensorEventListener); } }
```

Figure 2: Decompiled code from The Hundreds watch face.

be provided shortly; here we only mention a few examples. Event "select" refers to the selection of the watch face using the watch face picker; as a result, the watch face becomes active and visible. Event "deselect" is triggered when the user chooses another watch face using the picker; since the picker is displayed on top of the current watch face, the source of this transition is state *Invisible*. The default transition from *Interactive* to *Ambient* due to user inactivity is denoted by an artificial "standby" event.

It is important to note that this model is imprecise. While it represents all and only possible transitions from each state s_i upon each event e_j, not all *paths* in this model correspond to feasible run-time behaviors. We revisit this issue in the next section.

2.2 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 Figure 2. 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, etc. The watch face lifecycle start/end is defined by callback methods `onCreate` and `onDestroy` declared in `Engine`. When the watch face enters ambient mode, callback method `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 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` manages the watch face's use of the 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. 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 run-time sequence of callbacks in this scenario 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 reactivated 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`.

This example illustrates some of the challenges in developing watch faces. First, possible AW behaviors are not defined by precise models and could be misunderstood by app developers. One contribution of our work is defining a control-flow model to capture possible event sequences and the corresponding callbacks. Second, the management of energy consumption is an important consideration for AW devices and apps. As our studies suggest, mismanagement of energy-intensive resources (e.g., sensors, screen) does occur in real-world watch faces.

2.3 App Market Study

To understand how AW watch faces fit in the larger AW ecosystem, we performed an app market study. We used two collections of AW apps: (1) Android Wear Center (AWC) [62] and (2) Goko Store [28]. Each collection provides a reference to a Google Play AW app, together with an app classification. For our purposes, this classification can be used to distinguish watch faces from other AW apps.[1] For each AW app in these collections (including watch faces), we used Google Play to determine the date of the last update. As of February 2018, the total number of AW apps in these collections was 5198, of which 3070 were watch faces.

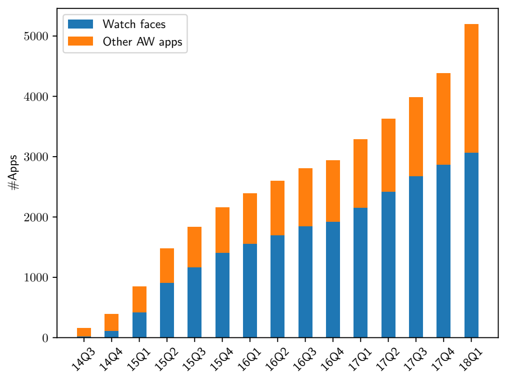

Figure 3: Number of watch faces and other AW apps.

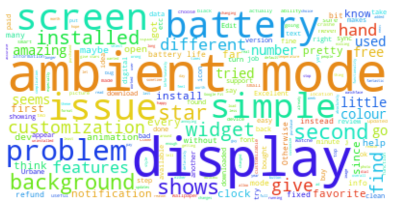

Figure 4: Word cloud of reviews for watch faces.

Figure 3 shows a cumulative distribution of the number of AW apps, based on the date of the last app update. Each x-axis point corresponds to a quarter (e.g., Q4 of 2017). The corresponding y-axis point is the number of apps whose last update is in this quarter or in any previous quarter. Watch faces present a sizeable fraction of all AW apps; this fraction is 59% for the last point on the x-axis. Further, many of the watch faces in these collections have been updated relatively recently. From the set of 3070 watch faces, some are paid and some are free. We obtained 1490 watch faces that were free and available in an unrestricted Play mirror [1]. Those watch faces were used in our experimental evaluation described later.

We used a crawler to obtain the most helpful reviews for each watch face. Figure 4 shows a word cloud for the reviews. We then performed text analysis on reviews containing the top frequent words. We saw many reviews complaining about the battery usage of watch faces: for example, *"although this watchface is very informative, it uses up too much battery life"*, *"it's great but drains my battery crazy fast"*, and *"unnecessary battery drain for the LG G Watch in ambient mode"*. These observations highlight the necessity for watch faces to effectively manage energy usage.

3 MODELING OF CONTROL FLOW AND ENERGY INEFFICIENCIES

Motivated by the prevalence of watch faces among AW apps, we developed a control-flow model that can be used as the basis for static analysis of watch faces as well as for test generation. To

[1]Google Play does not directly provide a way to identify watch faces, and searching for relevant keywords produces a limited number of results, many not related to AW.

the best of our knowledge, this is the first attempt to model the possible control-flow behaviors of watch faces. Based on this model, we present patterns of energy inefficiencies—specifically, patterns related to (1) the use of sensors and (2) the displays in ambient mode. The next section presents static analyses and testing techniques for finding instances of these inefficiency patterns.

3.1 Refined Control-Flow Model

The model presented earlier in Figure 1 does not capture faithfully the full complexity of watch face behavior. As described shortly, we created a tool to explore dynamically all possible sequences of events and states. Based on the observed paths, we developed a refined model with a set of states **State** defined as {*Interactive*, *Ambient*, *Null*, *InvisibleAppList*, *InvisibleNotification*, *InvisiblePicker*}. Here we represent the circumstances under which the watch face becomes invisible: (1) the user opens the list of apps to select an app to run, (2) a push notification is received and displayed to the user, and (3) the user opens the watch face picker to select a new watch face. The resulting model is shown in Figure 5.

The model is based on several categories of events. The effects of *swiping* depend on the direction of the swipe: for example, swiping from bottom to top opens the stream of push notifications. *Putting a palm* over the screen causes the watch face to enter the low-power ambient mode. Pressing the *side button* has a variety of effects, depending on the current state. *Wrist gestures* can be used for convenience: e.g., flicking the wrist in/out corresponds to swiping down/up, and shaking the wrist acts like pressing the side button. *Tapping* on the screen wakes up the watch from ambient mode; *tilting* the screen also wakes up the watch. As discussed earlier, *select* and *deselect* are artificial events representing the activation/deactivation of the watch face via the picker, and *standby* denotes an automatic transition after a period of user inactivity.

There is considerable overlap in the behaviors of many of these events. For simplicity, we elide events that have the same behavior as other events, but are difficult to trigger automatically: specifically, palm, tilt, and wrist gestures. Thus, we define the set of events as **Event** = {select, deselect, press_side_button, tap_screen, standby, swipe_right, swipe_left, swipe_down, swipe_up}.

Each $e \in$ **Event** triggers a sequence of callbacks. The previous section discussed lifecycle callbacks defined in subclasses of `Engine`: (1) `onCreate`, called during initialization; (2) `onDestroy`, invoked when a watch face is deselected; (3) `onAmbientModeChanged`, used when entering/leaving ambient mode; (4) `onVisibilityChanged`, called when the watch face becomes visible or hidden. The last two callbacks are invoked with a boolean parameter indicating the state change. Callbacks `onCreateEngine`, `onCreate` and `onDestroy` in `WatchFaceService` (the class in which an `Engine` is nested) are also of interest since they are invoked when a watch face is created and destroyed. We define the set **Callback** of lifecycle callbacks to contain `onVisibilityChanged(true)`, `onVisibilityChanged(false)`, `onAmbientModeChanged(true)`, `onAmbientModeChanged(false)`, `onCreate()`,`onDestroy()`, `wfsOnCreateEngine()`, `wfsOnCreate()`, and `wfsOnDestroy()`.

In the model in Figure 5, each state transition is $(s, s', e, c) \in$ **State** $\times$ **State** $\times$ **Event** $\times$ **Callback**$^+$. Here the transition is from s to s' upon event e. The transition triggers the sequence of callbacks c.

To generate this model, we implemented and instrumented a sample watch face to track state changes. More specifically, we created a watch face that implements all relevant APIs and then instrumented every method to track state transitions. Further, we developed a tool to automatically trigger events in **Event** starting from state *Null*. This tool also serves as the test case execution engine which will be discussed later in the paper. We recorded the state change if there was one, and continuously triggered events in the new state. This process was repeated until the state returned back to *Null* or there was no state transition observed. This systematic exploration produced a set of traces, which we analyzed to create the state transition graph in Figure 5.

3.2 Potential Inefficiencies Due to Sensors

One of the critical issues for wearable devices is their low battery capacity. When examining reviews for AW apps, we often see users complaining about battery drain. Hardware sensors may be one of the reasons for such drain. Sensors have to be acquired before and released after use. An app binds a listener to a sensor, and unbind it when the data is no longer needed. Callbacks in the listener are invoked when new sensor data is obtained. This is illustrated by the `SensorEventListener` object created at line 30 in Figure 2 and the related calls to `registerListener` and `unregisterListener`.

There are multiple categories of sensors, e.g., acceleration, rotation, gravity, magnetic field, heart rate, etc. Each category is represented by an integer constant defined in class `Sensor`, as exemplified by `TYPE_ACCELEROMETER` at line 33 in the example. Typically there exists only one hardware sensor for each category. A sensor is represented by an instance of `Sensor`; in the example, `mSensor` refers to such an object. When a listener is registered with a sensor, a callback `onSensorChanged` in the listener is invoked when new data is available; for brevity, this callback is not shown in Figure 2.

The guidelines for apps using sensors warn developers to *"be sure to unregister a sensor's listener when you are done using the sensor or when the sensor activity pauses"* since *"if a sensor listener is registered and its activity is paused, the sensor will continue to acquire data and use battery resources unless you unregister the sensor"* [19].

In order to apply this guideline to watch faces, we define two patterns of potential misuse. In this section we define them with respect to run-time behavior, while the next section uses static abstractions of this behavior. First, we define a *sensor resource* as a pair $res = \langle lis, sen \rangle$ where *lis* is an instance of `SensorEventListener` and *sen* is an instance of `Sensor`. We use this notion because there could be several listeners for the same sensor object, or several sensors that one listener listens to. Let **Sensor** be the set of such pairs. An invocation of `registerListener` for particular *lis* and *sen* is an "acquire" operation for the corresponding resources; we will denote it by $acq(res)$. Similarly, calling `unregisterListener` is a "release" operation $rel(res)$.

Consider a sequence of invocations of callback methods $s = c_1, \ldots, c_m$ where $c_i \in$ **Callback**. Let acq_i contain a set of $acq(res)$ elements such that the execution of callback method c_i (including the effects of its callees) acquires sensor resource *res*—i.e., it calls `registerListener` for *res* and reaches the exit of c_i without invoking a corresponding `unregisterListener`. Similarly, rel_i contains a set of $rel(res)$ that occurred during the execution of c_i.

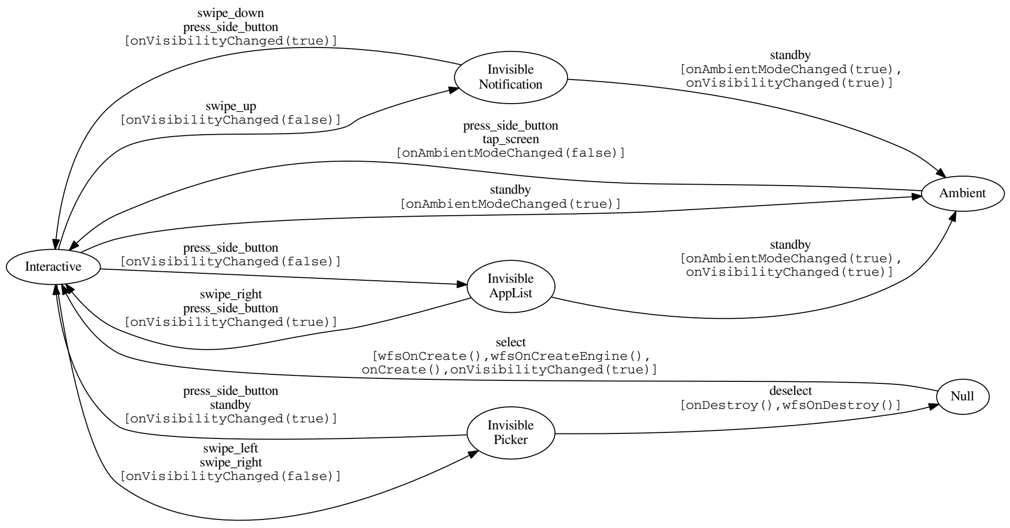
swipe_down
press_side_button
[onVisibilityChanged(true)]
Invisible Notification
standby
[onAmbientModeChanged(true), onVisibilityChanged(true)]
swipe_up
[onVisibilityChanged(false)]
press_side_button
tap_screen
[onAmbientModeChanged(false)]
standby
[onAmbientModeChanged(true)]
Ambient
Interactive
press_side_button
[onVisibilityChanged(false)]
Invisible AppList
standby
[onAmbientModeChanged(true), onVisibilityChanged(true)]
swipe_right
press_side_button
[onVisibilityChanged(true)]
select
[wfsOnCreate(),wfsOnCreateEngine(), onCreate(),onVisibilityChanged(true)]
press_side_button
standby
[onVisibilityChanged(true)]
deselect
[onDestroy(),wfsOnDestroy()]
Null
Invisible Picker
swipe_left
swipe_right
[onVisibilityChanged(false)]

Figure 5: State transition graph for watch face lifecycle.

For the running example in Figure 2, consider the sequence of states *Interactive*, *InvisibleNotification*, *Ambient*. As shown in Figure 5, the corresponding sequence of callback invocations is $c_1 =$ onVisibilityChanged(false), $c_2 =$ onAmbientModeChanged(true), $c_3 =$ onVisibilityChanged(true). There is one sensor resource *res* defined by the pair of objects obtained at lines 30 and 33 in Figure 2. The acquire/release sets are $acq_1 = \emptyset$, $rel_1 = \{rel(res)\}$, $acq_2 = \emptyset$, $rel_2 = \{rel(res)\}$, $acq_3 = \{acq(res)\}$, and $rel_3 = \emptyset$.

Similarly to traditional program analyses, the effects of some c_i can be expressed by a transfer function $f_i(S) = (S - Kill_i) \cup Gen_i$ where S is a set of sensors resources, $Kill_i = \{res \mid rel(res) \in rel_i\}$ and $Gen_i = \{res \mid acq(res) \in acq_i\}$. Given a path p in the control-flow model from Figure 5, let f_p be the composition of functions f_i for the callback sequence along the path.

Following the informal general guidelines for sensor management, we define two specific patterns of potential inefficiencies. First, consider a path p that starts from state *Null*, ends at that state, but does not contain it otherwise. If $f_p(\emptyset) \neq \emptyset$, this means that the watch face acquired some sensor resource but did not release it by the time the watch face was deselected. The second pattern occurs when a path p starts at state *Null* and ends at state *Ambient*, but does not contain either one as an intermediate state. If $f_p(\emptyset) \neq \emptyset$, in the low-power ambient mode (which could exist for a long period of time) there is active sensor that could drain the battery. Similar patterns have been studied in prior work for Android apps [41, 64, 65] but we are not aware of any similar work for AW apps, and in particular watch faces, which have their own lifecycle and control flow different from that of Android apps for handheld devices.

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 Figure 2 exemplifies this problem: the programmer has indeed attempted to release the sensor, but did so incorrectly.

One refinement that is needed in these definitions is the following: in addition to the callbacks in the control-flow model, the effects of callback onSensorChanged should be accounted for. Upon listener registration, the current value of the sensor is almost immediately provided to the listener—specifically, the AW framework invokes onSensorChanged on the listener object. This callback could release the sensor. Thus, after each callback c_i with an $acq_i \neq \emptyset$, the effects of invoking onSensorChanged should be "appended" by composing transfer function f_i with the transfer function of onSensorChanged.

3.3 Potential Inefficiencies Due to Displays

Displaying graphics in bright and vibrant colors consumes more energy than in plain dark colors [36], and may damage the screen if managed incorrectly for OLED displays. Dark themes may save as much as seven times the power of all-white displays [16]. In particular, for watch faces in ambient mode, AW guidelines suggest that developers to keep the graphics simple: *"draw outlines of shapes using a limited set of colors"*, set background to *"completely black or grey with no image"*, and *"aim to have 95% of pixels black"* [18, 20].

Unlike regular Android apps, where developers typically utilize predefined screen components (e.g. Button), watch faces only provide a low-level canvas to draw on. Developers are given an instance of Canvas and the bounds in which the watch face should be drawn. Class Paint is used for colors and styles of the drawing. We are interested in colors, and in particular the following APIs: (1) Paint.setColor(int), which takes as input a color value and configures the paint to draw the corresponding color, and (2) Canvas.drawColor(int), which takes a color value as a parameter and fills the entire canvas with the specified color.

```
 1 class RadialWatchFaceService extends CanvasWatchFaceService {
 2   class Engine extends CanvasWatchFaceService.Engine {
 3     DrawableWatchFace faceDrawer;
 4     void onCreate() { faceDrawer = new DrawableWatchFace(); }
 5     void onAmbientModeChanged(boolean inAmbientMode) {
 6       faceDrawer.setAmbient(inAmbientMode); }
 7     void onDraw(Canvas canvas,Rect bounds) {
 8       faceDrawer.draw(canvas,bounds); } } }

 9 class DrawableWatchFace {
10   boolean mActive = true;
11   Paint mArcPaint = new Paint();
12   void setAmbient(boolean state) { mActive = !state; }
13   void void draw(Canvas canvas, Rect bounds) {
14     mArcPaint.setColor(0xff03a9f4); // blue
15     if (mActive) canvas.drawPath(secondsPath,mArcPaint);
16     mArcPaint.setColor(0xff8bc34a); // green
17     canvas.drawPath(minutesPath,mArcPaint);
18     mArcPaint.setColor(0xffe51c23); // red
17     canvas.drawPath(hoursPath,mArcPaint); } }
```

Figure 6: Decompiled code from Radial watch face.

Each color is represented by an integer value such as `0xFF000000` (black) and `0xFFFFFFFF` (white). There are also APIs to obtain color values from RGB and string representations. For example, `Color.parseColor` transforms a string to its integer representation. To simplify the discussion, we elide such cases and only discuss integer constants in the code, but our implementation does handle these other cases.

The drawing on the canvas is performed by callback `onDraw`, which is invoked (frequently) in state *Interactive* and (less frequently) in state *Ambient*. Figure 6 illustrates this behavior; some details have been omitted for brevity. Helper class `DrawableWatchFace` contains the code for updating the canvas when `onDraw` is invoked. In interactive mode, the hours, minutes, and seconds hands are drawn. In ambient mode, the seconds hand is not drawn because the screen updates (i.e., the calls to onDraw) by default happen every 60 seconds in order to conserve energy. Although this class records the current state (interactive vs. ambient, in field `mActive`), the code logic inside `draw` uses the same set of colors for both cases.

We define two patterns for "suspicious" behavior that may indicate display inefficiencies in ambient mode. Both are defined for a path p in the control-flow model of the form *Interactive*⟶*Ambient*⟶*Interactive*. We consider the callbacks along p, with invocations of `onDraw` interleaved: c_1 = `onAmbientModeChanged(true)`, c_2 = `onDraw`, c_3 = `onAmbientModeChanged(false)`, c_4 = `onDraw`. Let col_i be the set of colors (appearing as parameters to `setColor` and `drawColor`) during the invocation of c_i and its transitive callees.

The first pattern we consider is when the set of colors does not change when the state change occurs: that is, $col_1 \cup col_2 = col_3 \cup col_4$. The example in Figure 6 matches this pattern because $col_1 = col_3 = \emptyset$ and $col_2 = col_4 = \{blue, green, red\}$. The second pattern is when the colors do change, but in ambient mode there are colors that are not black or dark shades of grey. In the example, suppose hypothetically that the call at line 14 (which uses the blue color) were inside the if-statement at line 15. In that case we would have had $col_2 = \{green, red\}$ and $col_4 = \{blue, green, red\}$, which does not exhibit the first pattern but does exhibit the second one.

Both patterns indicate that display management in ambient mode may violate AW guidelines. However, we do not consider this to be enough evidence to report a problem. After our static analysis (described in the next section) identifies these patterns, we generate and execute a test case to trigger the corresponding behavior at run time. A snapshot of the watch face in ambient mode is taken and analyzed with respect to the following AW guideline: at least x% of pixels in ambient mode should be black [18]. The recommended threshold is 95%; in our experiments we use a more relaxed value of 90%, and report only watchfaces that are below this threshold. The entire process of test generation, execution, and snapshot analysis is automated.

4 STATIC ANALYSIS AND INEFFICIENCY TESTING

Given the patterns defined so far, we developed static analyses to (1) construct the control-flow model in Figure 5, (2) analyze callbacks from the model, as well as related callbacks such as `onSensorChanged` and `onDraw`, and (3) identify instances of the potential inefficiencies. These instances are then used to automatically create test cases, whose run-time execution is used to decide whether to report the behavior to the programmer.

The use of static analyses has several benefits, compared to purely dynamic approaches. First, it provides detailed information about the specific code paths along which the inefficiencies occur, which is useful for code analysis and optimizations. Static analysis results can be used to create a small number of targeted test cases to cover the suspicious behaviors, which reduces the cost of subsequent testing. Further, static analyses can be employed to discover potential problems early in the development process, by executing then as part of the suite of static checkers used in development environments.

4.1 Sensor-Related Inefficiencies

4.1.1 Sensor resources. The analysis first determines the acquire and release operations for sensor resources. Recall from Section 3.2 that a run-time sensor resource $res = \langle lis, sen \rangle$ is a pair of a listener object lis and a sensor object sen. Statically, lis is a new expression for a `SensorEventListener`. The sensor object is an instance of `Sensor` and always obtained by calls to `getDefaultSensor` with an integer constant such as `Sensor.TYPE_ACCELEROMETER`. We perform propagation of integer constants to such calls. For each sensor type, the analysis creates an artificial `Sensor` object. Those objects, together with the listener objects, are then propagated to calls that register and unregister listeners. Those calls correspond to $acq(res)$ and $rel(res)$ elements in the analysis. The propagation is done via flow-insensitive, context-insensitive, field-based value-flow analysis [29], similar in spirit to points-to analysis.

4.1.2 Acquires and releases. For each method $c_i \in$ **Callback**, we need to compute its acquire set acq_i and release set rel_i. Callback methods `onAmbientModeChanged` and `onVisibilityChanged` are analyzed under four different contexts, for all possible combinations 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`.

We consider the effects of `onAmbientModeChanged` under the four possible contexts of `isInAmbientMode()` ∈ { `true`, `false` } and `isVisible()` $\in$ { `true`, `false` }. This is needed to capture cases where these status methods are used by the code to query the watch face state (e.g., as done in the running example). Similarly, we consider the effects of `onVisibilityChanged` under the same four contexts. We have seen examples where both status methods are used in the internal logic of a callback method, which means that both should be included in the context.

The analysis uses the context information to identify control-flow branches that are feasible under this context. First, the values of the formal parameters of `onAmbientModeChanged` (line 7) and `onVisibilityChanged` (line 9) are recorded based on the context. In the running example, `onAmbientModeChanged` under context ⟨`true`,$*$⟩ will set parameter `inAmbientMode` to `true`; here $*$ refers to either value for `isVisible()`. Further, parameter `ambientMode` at line 18 and field `mAmbientMode` at line 19 will be set to `true` as well. As a result, the `if` statement at lines 20–21 will execute the call to `stop`, which will release the sensor resource. If the context were ⟨`false`,$*$⟩, the call to `start` at line 21 would be executed instead and the sensor resource would be acquired.

To account for this context, we first propagate the context information using a value-flow analysis, similar to the one used to analyze sensor types, objects, and listeners. We utilize the SSA form of Jimple (the IR of Soot [13]) for this propagation. Since we use the analysis results to resolve conditionals (e.g., lines 20–21 and 24–25 in the running example), whenever several values flow to the same variable/field, we set its value to $\perp$ (i.e., "any"). In essence, this is a form of copy constant propagation. At the end of this propagation, the resulting values are used to resolve conditionals, if possible. A $\perp$ value means that both branches are possible.

Once feasible branches of conditionals are determined, the analysis traverses the control-flow graphs of c_i and its callees to determine $acq(res)$ and $rel(res)$ operations. For each encountered $acq(res)$, we perform an additional traversal to determine whether it is post-dominated by any matching $rel(res)$; if so, it should not be included in acq_i. For each encountered $acq(res)$, we perform a traversal to filter out those that are not guaranteed to be executed along all possible control-flow paths.

4.1.3 Path exploration. The acquire and release sets for callbacks are computed on demand during path exploration of the model from Figure 5. The context information described above is based on the model path being explored. Starting from *Null*, we perform a depth-first traversal to construct paths that represent the lifetime of a watch face (for pattern 1, ending with *Null*) or transition to ambient mode (for pattern 2, ending with *Ambient*). Only paths whose length does not exceed a parameter k are considered (value 3 is set in our implementation). Each c_i that is invoked along the path is considered, and the current state is maintained to determine the current values of `isInAmbientMode()` and `isVisible()`, needed to decide what context to use for c_i. The traversal maintains a set of acquired but not yet released sensors. When c_i is processed, all *res* from rel_i are removed from the set, and then the ones from acq_i are added to it. Any remaining *res* after the traversal is considered to be an indicator for a potential inefficiency. We record all such paths for subsequent test generation and execution.

For each reported path, we generate a test case based on the events along the path. The test generation and execution is based on a wrapper of MonkeyRunner [21] developed by us. Consider a path $s_1 \rightarrow \ldots \rightarrow s_n$. For each transition (s, s', e, c), event e is mapped to an API call of the wrapper. During the execution, we use `dumpsys` in Android Debug Bridge (ADB) to fetch information about acquired sensors. At the start and end of the execution of a test case, the test invokes ADB to record all listeners package names and sensor types. If a sensor is inactive at the start but stays active at the end, we consider the static analysis report to be confirmed.

4.2 Display-Related Inefficiencies

The first step of the analysis is to determine a set of colors for each call site of set-color APIs. Recall from Section 3.3 that a color is represented as an integer constant. We perform a propagation of integers in the range of `0xFF000000` to `0xFFFFFFFF` to calls to `Paint.setColor` and `Canvas.drawColor`. The propagation is similar to what was done for sensor types.

Next, a set of colors *col* is computed for `onAmbientModeChanged`. The analysis of this callback is done under two different contexts: when `isInAmbientMode()` is `true` and again when it is `false`. As before, the context is used to determine which branches of conditionals are feasible under that context. During a control-flow traversal, whenever a call to `setColor` or `drawColor` is encountered, its set of colors is added to *col*. To account for the effects of `onDraw`, an artificial call to it is added immediately before the exit of `onAmbientModeChanged`. This allows for the effects of the context to propagate to `onDraw`. For the example in Figure 6, the boolean parameter to `onAmbientModeChanged` (which is the same as the context) affects field `mActive` (line 12) which in turn is used during drawing (line 15). Our analysis captures these kinds of dependencies. If, hypothetically, the call to `setColor` at line 14 were guarded by the conditional at line 15, our static analysis would have determined that the corresponding blue color is used only under context `false`.

The color sets of the callbacks are examined for the two patterns introduced in Section 3.3. If there is a match of either pattern, a test case is generated to select the watch face and put it in ambient mode. A screenshot is taken during execution when the watch enters ambient mode. The automated analysis of screenshots is conducted offline. We calculate the percentage of black pixels and report the watch face as containing a display inefficiency if the percentage is below 90%, as described at the end of Section 3.3.

5 EVALUATION

We implemented the static analysis based on the Soot analysis framework [13]. Analysis performance was evaluated on a machine with 3.40GHz processor, 16GB RAM, and Ubuntu 16.04. Test execution was conducted on an LG Watch Style running Android Wear 2.9. The implementation of the approach and all benchmarks are available at https://presto-osu.github.io/fse18.

5.1 Experimental Subjects

There are two ways to distribute AW apps: (1) as standalone apps; and (2) embedded inside a handheld app. Recall from Section 2.3 that we obtained APKs for 1490 watch faces from a Play mirror [1]. We first perform a check on all downloaded APKs for embedded AW APKs. If an APK has any internal APK, we collect the embedded

Table 1: Characteristics of experimental subjects

#Apps	#Classes	#Methods	#Stmts	Time (sec)
1490	93532	438762	6807236	1885.72

Table 2: Summary of sensor-related inefficiencies.

SenPat 1		SenPat 2	
#Reported	#Confirmed	#Reported	#Confirmed
13	11	26	23

APK as a study subject. Otherwise, we directly use the original APK for further analysis. Table 1 shows the characteristics of all experimental subjects. The total number of classes of the 1490 watch faces is shown in column "#Classes". This includes all classes except those from the `android` library and some well-known third-party libraries such as `com.google`, `org.joda`, and `org.mozilla`. Column "#Stmts" shows the number of statements in Soot's IR. Column "Time (sec)" shows the running time of static analyses and test generation. The average cost of the analysis is around 2.7 seconds per 10K Jimple statements.

5.2 Sensor-Related Inefficiencies

Table 2 shows a summary of the result of detection for energy inefficiencies caused by mismanagement of sensors. Columns "#Reported" show the number of watch faces with potential inefficiencies reported by our static analysis for the two patterns described in Section 3.2, denoted as "SenPat 1" and "SenPat 2" in the table. Columns "#Confirmed" show the number of watch faces with a run-time unreleased sensor during test execution. In our experiments, a total of 13 watch faces are reported to have sensor unreleased when they are inactive and destroyed (SenPat 1). This means that the developer forgot to unregister sensor listeners in `onDestroy`, `wfsOnDestroy` and `onVisibilityChanged(false)`. For 11 out of the 13 reports, the test cases exposed unreleased sensors. The analysis reports 26 instances for SenPat 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. Table 3 shows all watch faces that are confirmed to have unreleased sensors during test execution. The checkmarks in column "SenPat 1" and "SenPat 2" indicate in what category a watch face is reported. The next-to-last entry corresponds to the running example, which exhibits SenPat 2.

An example of SenPat 1 is Bokeh. The watch face 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 `onCreate` and `onAmbientModeChanged(false)`. 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

Table 3: Confirmed sensor-related inefficiencies.

Package Name	SenPat 1	SenPat 2
com.atektura.analogglowlitewatchface		✓
com.atektura.datestampwatchface		✓
com.blis.android.wearable.bliswatchface	✓	
com.codingforlove.wear.watchfaces		✓
com.deglise.sensorface		✓
com.face.watch.meo		✓
com.mogoolab.androidwear.christmascounter	✓	✓
com.newscope.bmwwatchface.row	✓	
com.newscope.bmwwatchface	✓	
com.osthoro.animatedearthwatchface		✓
com.osthoro.beautifulstuddedwatchface		✓
com.pandaeyes.cryptowatch		✓
com.smartartstudio.turbo.free.interactive.watchface		✓
com.smartartstudio.ultron.interactive.watchface		✓
com.trigonesoft.paranormal		✓
com.virtualgs.snowwatch		✓
com.zanyatocorp.illusionwatchface		✓
cz.dmn.bokehwatchface	✓	
eu.stettiner.diamondwatchface		✓
eu.stettiner.dietwatch	✓	✓
eu.stettiner.manyiconswatchface		✓
info.fathom.watchfaces.coubertin	✓	✓
net.yt1300.watchfacemodel101b	✓	✓
net.yt1300.watchfacemodel102	✓	✓
pl.nwg.dev.wear.rambler		✓
ru.slobodchikov.kgbwatchface	✓	✓
wearable.android.breel.com.thehundreds		✓
wear.trombettonj.trombt1pearlfree	✓	

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.

We observed false positives of the static analysis in the following three cases. First, in the Ceres watch face (reported as SenPat 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"* [22]. 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, reported as SenPat 1&2. 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.

Based on the results from Table 3, analysis precision is 11/13=85% for SenPat 1 and 23/26=88% for SenPat 2. We also determined analysis recall. First, we checked all 1490 watch faces and found that 58 of them register sensors. We extensively studied the code and the run-time behavior of all 58 watch faces, and manually identified 11 instances of SenPat 1 and 26 instances of SenPat 2. Thus, the recall is 11/11=100% for SenPat 1 and 23/26=88% for SenPat 2. In the three false negatives, the listener is the watch face service. It should be possible to generalize our analysis to handle this case.

For the 23 watch faces that did exhibit run-time violations of SenPat 2, we performed additional studies of the decompiled code to

Table 4: Summary of display-related inefficiencies.

DisPat 1&2		GEMMA (90% Black,10% White)
#Static	#Reported	#Reported
67	47	42

determine whether there was legitimate sensor use in ambient mode. An example of such use could be gathering heart rate statistics or user motion statistics for fitness apps. However, we found only one watch face in which the sensor use may be somewhat legitimate; in all other cases, the sensors should have been turned off in ambient mode. This one watch face writes the sensor information to the device log using `Log.d()` calls. These logs could conceivably be used by other apps on the device, but those apps would need a special `READ_LOGS` permission to be granted by the user. It is not clear that such logs would be of any use when the watch face is deployed on users' devices.

5.3 Display-Related Inefficiencies

Table 4 shows a summary of the static analysis reports, as well as a comparison with GEMMA [36], for the energy inefficiencies due to displays in ambient mode. Column "#Static" shows the number of watch faces identified by our static analysis as having the two display-related patterns described in Section 3.3. We denote these patterns as DisPat 1 and DisPat 2. Recall that DisPat 1 means the color set does not change when a watch face goes into ambient mode. DisPat 2 means that the set of colors changes but contains colors that are not black. We consider colors whose RGB values are below 10 as black. Any color with greater RGB values is considered not recommended in ambient mode. The number of reported potential inefficiencies is shown in column "#Reported" under "DisPat 1&2". As discussed earlier, these static reports should be followed by test execution and analysis of the actual display observed at run time on the device (in our case, on the LG Watch Style). In our experiment, 47 of 67 watch faces were observed to violate the guidelines because they contained too many (>10%) non-black pixels in ambient mode. Note that we only consider pixels inside the inscribed circle of a screenshot, since all screenshots are taken as square images while the smartwatch we use in the experiment has a round screen. Any pixel beyond the circle is ignored.

To further validate our results, we implemented estimates based on the GEMMA approach [36]. Higher power consumption implies higher energy use over a period of time; thus, we can use power as an indicator of the energy-intensiveness of each watch face in ambient mode. To obtain estimates of power consumption, we analyzed the ambient mode screenshots using our implementation of a GEMMA-based technique. GEMMA [36] calculates the theoretical power consumption of OLED displays for an app and provides suggestions for power efficient color palettes. Other researchers have used similar techniques [31, 63]. The power consumption for OLED displays can be modeled as a linear function of the RGB values for each pixel. GEMMA formulates the function as

$$TP = \sum_{x=0}^{X} \sum_{y=0}^{Y} \left(P_R(R_{x,y}) + P_G(G_{x,y}) + P_B(B_{x,y})\right)$$

Table 5: Reports of display-related inefficiencies.

Package Name	DisPat 1	DisPat 2	GEMMA
com.asus.facedesigner	✓		✓
com.dylanp.navballwatchface	✓		✓
com.epix.nicetimewatchface	✓		✓
com.gashfara.surfwatchface	✓		✓
com.goldenbrown.watches	✓		✓
com.milesoberstadt.radialwatchface	✓		
com.multidots.watchface	✓		✓
com.nickschwab.android.wear.simplefaces	✓		✓
com.qmzc.timagine.watchface.earth		✓	✓
com.qmzc.timagine.watchface.flat		✓	✓
com.qmzc.timagine.watchface.kiwi		✓	✓
com.qmzc.timagine.watchface.maze		✓	✓
com.raimund.bigsimple_ger	✓		✓
com.raimund.retrolcd	✓		✓
com.rocas.classicfree	✓		✓
com.runderbin.blackclassicwatchface	✓		
com.smartmadsoft.wear.face.everyday		✓	✓
com.stmp.counterface	✓		✓
com.syzygy.tarvos	✓		✓
com.watch.richface.delta		✓	✓
com.watch.richface.guard		✓	✓
com.watch.richface.infinity		✓	✓
com.watch.richface.smartdrive		✓	✓
com.watch.richface.throttle	✓		✓
com.watchwright.christmas	✓		✓
com.watchwright.cross	✓		✓
com.watchwright.us	✓		✓
com.wearclan.watchface.face	✓		✓
com.wearclan.watchface.lightsense	✓		✓
com.wearclan.watchface.technomachine	✓		✓
com.wearclan.watchface.vividthanksgiving	✓		✓
de.uschonha.tinylaser	✓		✓
eu.foxjunior.simpleandcleanwathfacefree	✓		✓
eu.stettiner.dietwatch		✓	✓
fi.fluid.watcherwear	✓		
org.beatonma.io16	✓		
re.hofer.watchface.binary	✓		
ru.devsp.apps.customwatch	✓		✓
watch.richface.androidwear.armada2	✓		✓
watch.richface.androidwear.digitalvision	✓		✓
watch.richface.androidwear.fury		✓	✓
watch.richface.androidwear.ntouch	✓		✓
watch.richface.androidwear.timegate	✓		✓
watch.richface.androidwear.valiant	✓		✓
watchface.lbriceno.com.binarynerd	✓		✓
wear.android.cricking.crickingandroidwear	✓		✓
wearable.android.ns.nl.wearabletest	✓		✓

Here TP is the total power for a given screenshot. X and Y are the total number of pixels in each dimension (in our case, restricted to the round watch face area), x and y are coordinates of a pixel, and $P_R(r)$, $P_G(g)$ and $P_B(b)$ are linear power consumption functions for RGB values in a pixel. As observed by others [36, 63], blue pixels consume nearly twice the power of red and green pixels, a pixel with white color consumes more power than one with any other color, and a black pixel has the lowest power consumption.

Recall that we report a watch face as exhibiting energy inefficiencies if the percentage of black pixels is below 90%. In terms of power, this means a watch face is reported if its power consumption in ambient mode is greater than the power consumption of an image with 90% black pixels. The upper bound of the power consumed by such an image occurs when the other 10% of pixels are all white, as white is the most power-intensive color. We use this as a baseline and calculate the power consumption estimate TP_{base}. Any watch face whose TP estimate is larger than TP_{base} can be regarded as a violation. Column "#Reported" under "GEMMA (90% Black,10% White)" shows the total number of such watch faces.

Table 5 shows the package names of all watch faces reported by the proposed analysis and by our implementation of GEMMA. Columns "DisPat 1" and "DisPat 2" show whether the watch face is reported by our approach as an instance of the pattern. A checkmark in column "GEMMA" indicates a report by our GEMMA implementation. Our analysis report covers all cases that are reported by GEMMA estimates. There are 5 reports by our analysis that theoretically consume less power than TP_{base} and thus are not reported by GEMMA estimates. They use colors other than black that are not very energy-consuming—for example, red and green. Note that it is easy to use the GEMMA-based criterion rather than the simpler number-of-black-pixels criterion when deciding whether to report pattern instances. In the public implementation of our approach, we include both filters as possible choices for the users of our analyses.

Summary. Our results can be summarized as follows: static analysis of potential energy inefficiencies in watch faces can be performed with low cost and high precision. The analysis output provides specific information about the underlying causes of inefficiencies (e.g., executions paths in the code) and can be used to generate test cases to exhibit the problem at run time. Using this analysis, combined with dynamic checks after test execution, we were able to identify 75 watch faces with energy-related inefficiencies.

6 RELATED WORK

Android Wear characterization and uses. Liu and Lin [38] examine hardware and OS level characteristics of AW devices to find execution inefficiencies and design flaws. Many researchers have focused on the security issues of AW devices and apps. Do et al. [11] present techniques to leak sensitive data from AW devices. Mujahid [50] has conducted a study of permission and feature mismatch of AW apps. Liu et al. [40] present side-channel attacks to infer user inputs by exploiting sensors on devices. There also exists several approaches from the HCI community, focusing on application scenarios and user interface design. Shen et al. [61] detect handshakes on AW devices to create secret keys for secure communication. SafeDrive [27] collects and analyzes behaviors of drivers for distraction detection. Arduser et al. [2] use motion data collected in AW smartwatches for text recognition. Reyes et al. [58] introduce novel gestures based on user's thumb movement. SHOW [34] captures sensor data to deduce handwriting.

Energy analysis for Android and Wear. There is a large body of work on energy issues for regular Android apps [3–7, 23, 25, 26, 30, 35, 41, 47, 51–55, 57, 65]. Several optimizations have been proposed for OLED displays [31, 36, 63]. GEMMA [36] generates color palettes using multi-objective optimization to produce energy-friendly colors. We use GEMMA-based estimates in our reports and experimental evaluation. For cases where display-related inefficiencies are reported by our hybrid static/dynamic approach, GEMMA could be used to provide suggestions for improvements.

Banerjee et al. [4] introduced a dynamic analysis for detection of energy hotspots and bugs in Android apps. Their follow-up work [3, 5, 6] proposed techniques for debugging and fixing energy inefficiencies, based on dynamically-generated GUI models. Dynamic analysis based on static GUI models has also been used for exploring run-time inefficiencies [41], including sensor-related ones. Static analysis has been employed to report missing-deactivation energy defects [65] and to generate test cases for sensor-related leaks [64]. The missing-deactivation patterns in our work have similar structure, but these existing approaches are specific to regular Android apps and cannot be applied directly to AW apps, including watch faces which have their own distinct lifecycle and control flow. Energy-related behaviors for Android have also been considered in other contexts. Jabbarvand et al. [26] developed an approach to minimize the number of tests needed to uncover energy bugs. Follow-up work on μDroid [25] defines a mutation testing approach to evaluate the ability of a test suite to reveal energy inefficiencies. Cruz and Abreu [10] 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. [46] present an exploratory investigation of users' expectations, interactions, and charging behaviors when using smartwatches. Poyraz and Memik [56] 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. [39] 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 specific energy inefficiencies of watch faces by static analysis and testing.

Testing and analysis of Android apps. Linares-Vásquez et al. [37] present a summary of the current state of frameworks, tools, and services for automated testing for Android. Choudhary et al. [9], Li et al. [32], and Sadeghi et al. [59] conduct similar studies. Fazzini et al. [12] propose a technique for generating platform-independent test scripts for Android apps. Li et al. [33] consider the evolution of GUI test scripts for mobile apps. Zhang et al. [70] generate tests using a static GUI model [66, 68]. Garcia et al. [14] leverage symbolic execution to generate inter-component communication exploits. Sapienz [44] uses search-based testing to explore test sequences. CrashScope [49] uses a model-based approach to detect and report crashes. Other representative tools include Axiz [45], Dynodroid [42], EvoDroid [43], PATDroid [60], GATOR [66–68], PUMA [24], SwiftHand [8], and TrimDroid [48].

7 CONCLUSIONS

With the increasing popularity of wearable devices, various challenges have emerged for both developers and software engineering researchers. Our work focuses on Android Wear watch faces, which are some of the most popular Wear apps. We propose a watch face control-flow model, define energy inefficiency patterns for sensors and displays, and implement static analysis and test generation to identify them. The evaluation shows that the proposed approach has low cost, high precision, and can successfully detect inefficiencies in a wide range of real-world watch faces.

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

REFERENCES

[1] APKPure. 2018. APKPure: Free APKs online. https://apkpure.com.
[2] L. Arduser, P. Bissig, P. Brandes, and R. Wattenhofer. 2016. Recognizing text using motion data from a smartwatch. In *WristSense*. 1–6.
[3] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. 2017. EnergyPatch: Repairing resource leaks to improve energy-efficiency of Android apps. In *TSE*. 1–20.
[4] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting energy bugs and hotspots in mobile apps. In *FSE*. 588–598.
[5] Abhijeet Banerjee, Haifeng Guo, and Abhik Roychoudhury. 2016. Debugging energy-efficiency related field failures in mobile apps. In *MOBILESoft*. 127–138.
[6] Abhijeet Banerjee and Abhik Roychoudhury. 2016. Automated re-factoring of Android apps to enhance energy-efficiency. In *MOBILESoft*. 139–150.
[7] Xiaomeng Chen, Abhilash Jindal, Ning Ding, Yu Charlie Hu, Maruti Gupta, and Rath Vannithamby. 2015. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *MobiCom*. 40–52.
[8] W. Choi, G. Necula, and K. Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *OOPSLA*. 623–640.
[9] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for Android: Are we there yet?. In *ASE*. 429–440.
[10] Luis Cruz and Rui Abreu. 2017. Performance-based guidelines for energy efficient mobile applications. In *MOBILESoft*. 46–57.
[11] Quang Do, Ben Martini, and Kim-Kwang Raymond Choo. 2017. Is the data on your wearable device secure? An Android Wear smartwatch case study. *SP&E* 47, 3 (2017), 391–403.
[12] Mattia Fazzini, Eduardo Noronha De A Freitas, Shauvik Roy Choudhary, and Alessandro Orso. 2017. Barista: A technique for recording, encoding, and running platform independent Android tests. In *ICST*. 149–160.
[13] Soot Framework. 2018. Soot analysis framework. https://sable.github.io/soot.
[14] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. 2017. Automatic generation of inter-component communication exploits for Android applications. In *FSE*. 661–671.
[15] Gartner. 2017. Press release. https://www.gartner.com/newsroom/id/3790965.
[16] Google. 2017. Android Wear: What's new & best practices (Google I/O '17). https://www.youtube.com/watch?v=97U6W-5iF_o.
[17] Google. 2018. The activity lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle.html.
[18] Google. 2018. Always-on. https://designguidelines.withgoogle.com/android-wear/patterns/always-on.html#always-on-style.
[19] Google. 2018. Best practices for accessing and using sensors. https://developer.android.com/guide/topics/sensors/sensors_overview.html#sensors-practices.
[20] Google. 2018. Designing watch faces. https://developer.android.com/training/wearables/watch-faces/designing.html.
[21] Google. 2018. MonkeyRunner. https://developer.android.com/studio/test/monkeyrunner.
[22] Google. 2018. Optimizing watch faces: Move expensive operations outside the drawing method. https://developer.android.com/training/wearables/watch-faces/performance.html#OutDrawing.
[23] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. 2013. Estimating mobile application energy consumption using program analysis. In *ICSE*. 92–101.
[24] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *MobiSys*. 204–217.
[25] Reyhaneh Jabbarvand and Sam Malek. 2017. μDroid: An energy-aware mutation testing framework for Android. In *FSE*. 208–219.
[26] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2016. Energy-aware test-suite minimization for Android apps. In *ISSTA*. 425–436.
[27] Landu Jiang, Xinye Lin, Xue Liu, Chongguang Bi, and Guoliang Xing. 2018. SafeDrive: Detecting distracted driving behaviors using wrist-worn devices. *IMWUT* 1, 4 (Jan. 2018), 144:1–144:22.
[28] Jakob Körner, Lars Hitzges, and Dennis Gehrke. 2018. Goko store. https://goko.me.
[29] O. Lhoták and L. Hendren. 2003. Scaling Java points-to analysis using Spark. In *CC*. 153–169.
[30] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. 2013. Calculating source line level energy information for Android applications. In *ISSTA*. 78–89.
[31] Ding Li, Angelica Huyen Tran, and William G. J. Halfond. 2015. Nyx: A display energy optimizer for mobile web apps. In *FSE*. 958–961.
[32] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of Android apps: A systematic literature review. *IST* 88 (2017), 67–95.
[33] Xiao Li, Nana Chang, Yan Wang, Haohua Huang, Yu Pei, Linzhang Wang, and Xuandong Li. 2017. ATOM: Automatic maintenance of GUI test scripts for evolving mobile applications. In *ICST*. 161–171.
[34] Xinye Lin, Yixin Chen, Xiao-Wen Chang, Xue Liu, and Xiaodong Wang. 2018. SHOW: Smart handwriting on watches. *IMWUT* 1, 4 (Jan. 2018), 151:1–151:23.
[35] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining energy-greedy API usage patterns in Android apps: An empirical study. In *MSR*. 2–11.
[36] Mario Linares-Vásquez, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2015. Optimizing energy consumption of GUIs in Android apps: A multi-objective approach. In *FSE*. 143–154.
[37] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *ICSME*. 399–410.
[38] Renju Liu and Felix Xiaozhu Lin. 2016. Understanding the characteristics of Android Wear OS. In *MobiSys*. 151–164.
[39] Xing Liu, Tianyu Chen, Feng Qian, Zhixiu Guo, Felix Xiaozhu Lin, Xiaofeng Wang, and Kai Chen. 2017. Characterizing smartwatch usage in the wild. In *MobiSys*. 385–398.
[40] Xiangyu Liu, Zhe Zhou, Wenrui Diao, Zhou Li, and Kehuan Zhang. 2015. When good becomes evil: Keystroke inference with smartwatch. In *CCS*. 1273–1285.
[41] Yepang Liu, Chang Xu, S. C. Cheung, and Jian Lu. 2014. GreenDroid: Automated diagnosis of energy inefficiency for smartphone applications. *TSE* 40 (Sept. 2014), 911–940. Issue 9.
[42] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for Android apps. In *FSE*. 224–234.
[43] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented evolutionary testing of Android apps. In *FSE*. 599–609.
[44] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *ISSTA*. 94–105.
[45] K. Mao, M. Harman, and Y. Jia. 2017. Robotic testing of mobile apps for truly black-box automation. *IEEE Software* 34, 2 (2017), 11–16.
[46] Chulhong Min, Seungwoo Kang, Chungkuk Yoo, Jeehoon Cha, Sangwon Choi, Younghan Oh, and Junehwa Song. 2015. Exploring current practices for battery use and management of smartwatches. In *ISWC*. 11–18.
[47] Chulhong Min, Youngki Lee, Chungkuk Yoo, Seungwoo Kang, Sangwon Choi, Pillsoon Park, Inseok Hwang, Younghyun Ju, Seungpyo Choi, and Junehwa Song. 2015. PowerForecaster: Predicting smartphone power impact of continuous sensing applications at pre-installation time. In *SenSys*. 31–44.
[48] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of Android applications. In *ICSE*. 559–570.
[49] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk. 2016. Automatically discovering, reporting and reproducing Android application crashes. In *ICST*. 33–44.
[50] Suhaib Mujahid. 2018. *Determining and detecting permission issues of wearable apps*. Master's thesis. Concordia University.
[51] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2017. Software-based energy profiling of Android apps: Simple, efficient and reliable?. In *SANER*. 103–114.
[52] Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. 2013. Carat: Collaborative energy diagnosis for mobile devices. In *SenSys*. 1–14.
[53] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. 2012. Where is the energy spent inside my app?. In *EuroSys*. 29–42.
[54] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. 2011. Fine-grained power modeling for smartphones using system call tracing. In *EuroSys*. 153–168.
[55] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. 2012. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys*. 267–280.
[56] E. Poyraz and G. Memik. 2016. Analyzing power consumption and characterizing user activities on smartwatches. In *IISWC*. 1–2.
[57] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. 2011. Profiling resource usage for mobile applications: A cross-layer approach. In *MobiSys*. 321–334.
[58] Gabriel Reyes, Jason Wu, Nikita Juneja, Maxim Goldshtein, W. Keith Edwards, Gregory D. Abowd, and Thad Starner. 2018. SynchroWatch: One-handed synchronous smartwatch gestures using correlation and magnetic sensing. *IMWUT* 1, 4 (Jan. 2018), 158:1–158:26.
[59] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. 2017. A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software. *TSE* 43, 6 (2017), 492–530.
[60] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2017. PATDroid: Permission-aware GUI testing of Android. In *FSE*. 220–232.
[61] Yiran Shen, Fengyuan Yang, Bowen Du, Weitao Xu, Chengwen Luo, and Hongkai Wen. 2018. Shake-n-Shack: Enabling secure data exchange between smart wearables via handshakes. In *PerCom*. 1–10.
[62] Wearable software. 2018. Android Wear center. http://androidwearcenter.com.
[63] M. Wan, Y. Jin, D. Li, and W. G. J. Halfond. 2015. Detecting display energy hotspots in Android apps. In *ICST*. 1–10.

[64] Haowei Wu, Yan Wang, and Atanas Rountev. 2018. Sentinel: Generating GUI tests for Android sensor leaks. In *AST*. 27–33.
[65] Haowei Wu, Shengqian Yang, and Atanas Rountev. 2016. Static detection of energy defect patterns in Android applications. In *CC*. 185–195.
[66] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *JASE* (June 2018), 1–41.
[67] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static control-flow analysis of user-driven callbacks in Android. In *ICSE*. 89–99.
[68] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static window transition graphs for Android. In *ASE*. 658–668.
[69] Hailong Zhang and Atanas Rountev. 2017. Analysis and testing of notifications in Android Wear applications. In *ICSE*. 347–357.
[70] Hailong Zhang, Haowei Wu, and Atanas Rountev. 2016. Automated test generation for detection of leaks in Android applications. In *AST*. 64–70.