Diagnosing Energy Efficiency and Performance for Mobile Internetware Applications

Yepang Liu, The Hong Kong University of Science and Technology

Chang Xu, Nanjing University

Shing-Chi Cheung, The Hong Kong University of Science and Technology

// *Many smartphone application bugs compromise energy efficiency and performance, yet we lack powerful tools to address them. Examining their characteristics and diagnostic challenges offers a key step forward.* //

MOBILE INTERNETWARE applications seamlessly connect physical and cyber environments and provide smart services to users. However, unlike their desktop counterparts, smartphone applications run on resource-constrained platforms, so even small implementation inefficiencies can lead to a bad user experience. Ensuring a satisfactory user experience is a nontrivial task for developers, especially because many smartphone applications have these requirements:

- *Seamless communication.* Smartphone applications (such as email clients or shopping applications) often use the latest cloud data. Inefficient communication with Internet cloud services can easily waste valuable battery power and network bandwidth.
- *Frequent sensing.* Smartphone applications often use various sensors to detect users' physical and cyber environments and thereby provide context-aware services (such as navigation). However, sensing operations consume considerable energy, and problematic use of sensors can easily waste energy.
- *Intensive computation.* Most smartphones have fully fledged operating systems, enabling computationally intensive applications (such as games). However, such computation places a big burden on smartphone GPUs and CPUs, and inefficient computation easily hurts application performance.

The smartphone application market is extremely competitive. Developers rarely have sufficient time or resources to carefully optimize their

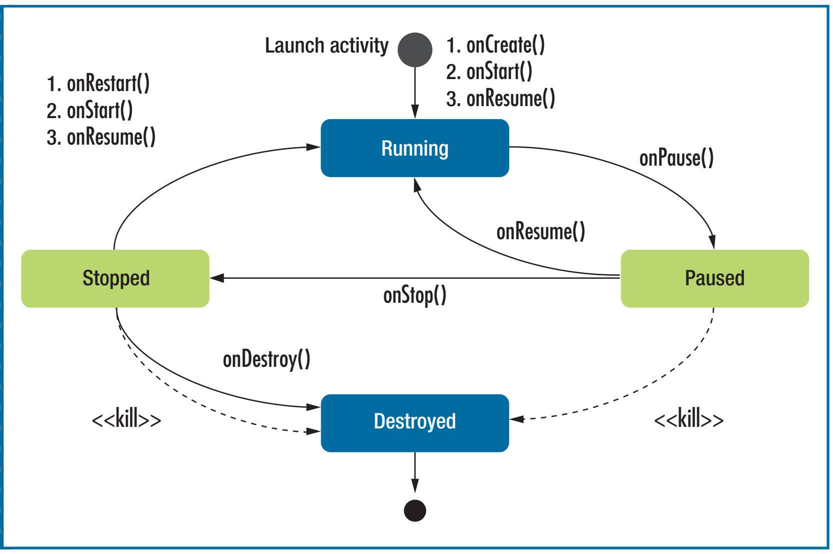


FIGURE 1. The life cycle of an Android application's activity component. Application GUIs are defined in activities. The activity's life cycle starts when the **onCreate()** handler is called and ends after the **onDestroy()** handler is called.

applications' energy efficiency and performance before pushing them to market. So, many applications suffer from energy and performance bugs.[1] The former can seriously waste battery power; the latter can significantly reduce smartphones' responsiveness and available computational resources (such as memory and network bandwidth). These bugs severely impact the user experience and cause significant user frustration.[2,3] Despite such bugs' pervasiveness, both the research and industry communities haven't sufficiently understood them. The result is a lack of mature tools to effectively help developers locate energy black holes and performance threats in their smartphone applications.

Here, we discuss the challenges in diagnosing energy and performance bugs in real-life Android applications. We hope to inspire further efforts to adequately address these challenges. We also review state-of-the-art diagnostic techniques and tools. In particular, we offer results from our case study, which applied a representative tool to popular commercial Android applications and the Samsung Mobile Software Development Kit (SDK). Our study results demonstrate that such tools are useful to and necessary for smartphone application developers; their feedback further suggests our research's practical value.

Activities

Our article focuses on the Android OS, which is Linux-based. Android applications are written primarily in Java, although, for performance reasons, developers might write critical parts using native languages (such as C).

Android applications typically comprise four types of components: *activity*, *service*, *broadcast receiver*, and *content provider*.[1] Each component follows a prescribed life cycle defining how it's created, used, and destroyed. We describe the activity component here to aid the understanding of our later discussion. (For more on application components, see the official Android developer website, http://developer.android.com.)

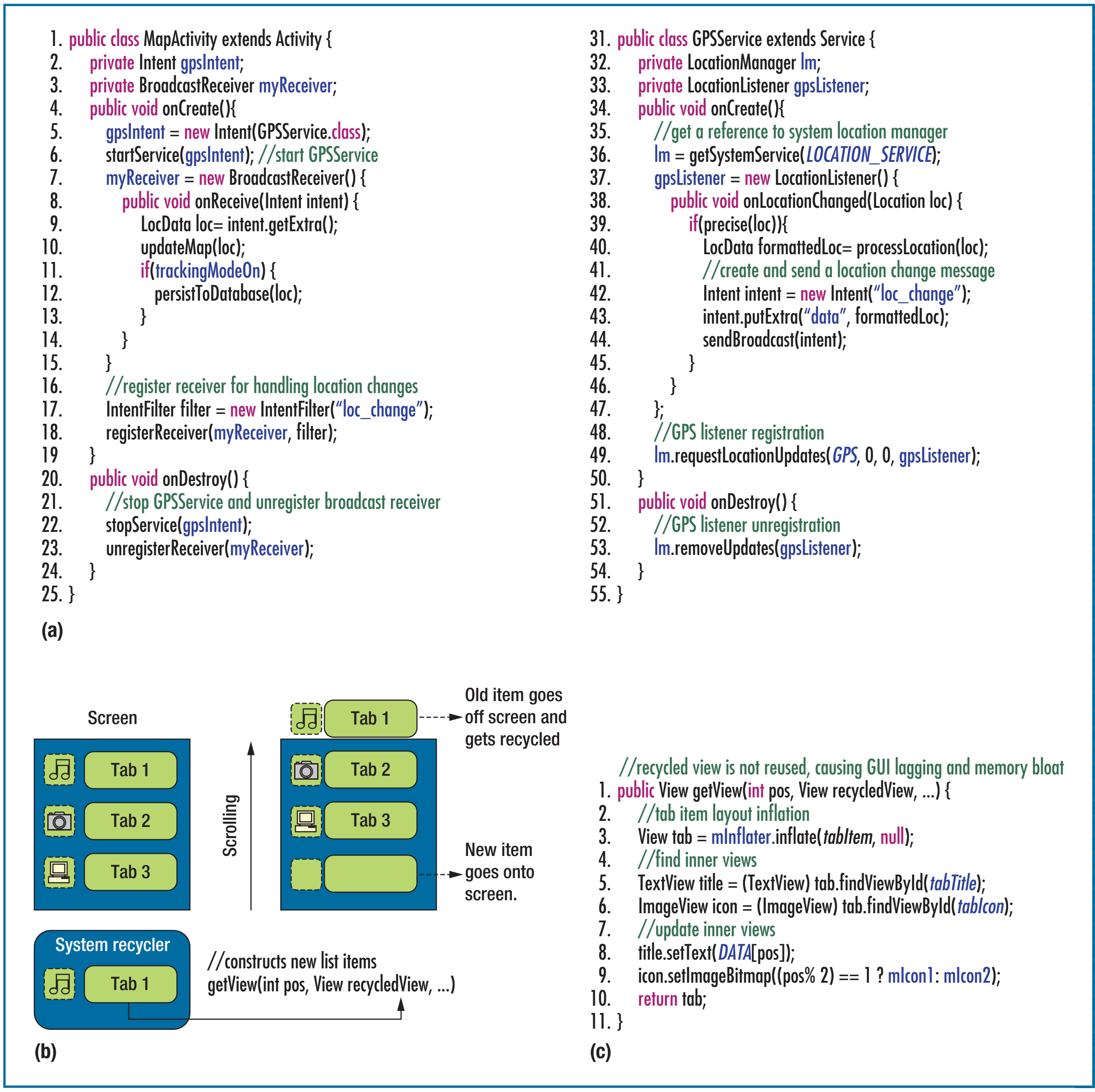
Application GUIs are defined in activities. Figure 1 shows an activity's life cycle. It starts after the **onCreate()** handler is called and ends after the **onDestroy()** handler is called. An activity's foreground lifetime (its "running" state) starts after the **onResume()** handler is called and lasts until the **onPause()** handler is called. At that point, the activity goes to the background (its "stopped" state) and becomes invisible. An activity can interact with users only when it's in the foreground. When it goes to the background, its **onStop()** handler is called. When users navigate back to a paused or stopped activity, the activity's **onResume()** or **onRestart()** handler is called, and it returns to the foreground. In exceptional cases, a paused or stopped activity might be killed to release memory to higher-priority applications.

Energy and Performance Bugs

Our recent studies identified that the following three common bug types seriously affect smartphone applications' energy efficiency and performance:[1,4]

- *Energy leak bugs* can quickly exhaust batteries.
- *GUI-lagging bugs* can significantly reduce application responsiveness.
- *Memory bloat bugs* can consume too much memory.

Here, we discuss representative examples of these bugs from real-world Android applications. For a more comprehensive discussion, along with additional bug patterns and examples, see our empirical studies.[1,4]



```
1.  public class MapActivity extends Activity {
2.    private Intent gpsIntent;
3.    private BroadcastReceiver myReceiver;
4.    public void onCreate(){
5.      gpsIntent = new Intent(GPSService.class);
6.      startService(gpsIntent); //start GPSService
7.      myReceiver = new BroadcastReceiver() {
8.        public void onReceive(Intent intent) {
9.          LocData loc= intent.getExtra();
10.         updateMap(loc);
11.         if(trackingModeOn) {
12.           persistToDatabase(loc);
13.         }
14.       }
15.     }
16.     //register receiver for handling location changes
17.     IntentFilter filter = new IntentFilter("loc_change");
18.     registerReceiver(myReceiver, filter);
19    }
20.   public void onDestroy() {
21.     //stop GPSService and unregister broadcast receiver
22.     stopService(gpsIntent);
23.     unregisterReceiver(myReceiver);
24.   }
25. }
```

(a)

```
31. public class GPSService extends Service {
32.   private LocationManager lm;
33.   private LocationListener gpsListener;
34.   public void onCreate(){
35.     //get a reference to system location manager
36.     lm = getSystemService(LOCATION_SERVICE);
37.     gpsListener = new LocationListener() {
38.       public void onLocationChanged(Location loc) {
39.         if(precise(loc)){
40.           LocData formattedLoc= processLocation(loc);
41.           //create and send a location change message
42.           Intent intent = new Intent("loc_change");
43.           intent.putExtra("data", formattedLoc);
44.           sendBroadcast(intent);
45.         }
46.       }
47.     };
48.     //GPS listener registration
49.     lm.requestLocationUpdates(GPS, 0, 0, gpsListener);
50.   }
51.   public void onDestroy() {
52.     //GPS listener unregistration
53.     lm.removeUpdates(gpsListener);
54.   }
55. }
```

Screen
Tab 1
Tab 2
Tab 3
Scrolling
Tab 1
Tab 2
Tab 3
Old item goes off screen and gets recycled
New item goes onto screen.
System recycler
Tab 1
//constructs new list items
getView(int pos, View recycledView, ...)

(b)

```
//recycled view is not reused, causing GUI lagging and memory bloat
1.  public View getView(int pos, View recycledView, ...) {
2.    //tab item layout inflation
3.    View tab = mInflater.inflate(tabItem, null);
4.    //find inner views
5.    TextView title = (TextView) tab.findViewById(tabTitle);
6.    ImageView icon = (ImageView) tab.findViewById(tabIcon);
7.    //update inner views
8.    title.setText(DATA[pos]);
9.    icon.setImageBitmap((pos% 2) == 1 ? mIcon1: mIcon2);
10.   return tab;
11. }
```

(c)

FIGURE 2. Examples of energy and performance bugs. (a) An energy leak bug. The code implements a callback that uses sensory data in an energy-inefficient way. (b) A list view (Firefox's tab tray). (c) A GUI-lagging and memory bloat bug. The code implements an inefficient list view callback.

Energy Leak Bugs

Many applications that communicate with the physical or cyber environment through sensors or network interfaces suffer from subtle energy leak bugs. One common cause is cost-ineffective use of sensory or network data.[4,5]

Figure 2a shows a simplified version of a real-world bug.[4] The associated application uses GPS data for navigation and location tracking (which can be disabled). When users launch the application, MapActivity (lines 1–25) starts, creating a live map for user interactions. For navigation, the application maintains a long-running

GPSService in the background for location sensing (lines 31–55). On receiving new location data, GPSService checks whether the data satisfy certain precision criterion (line 39). If that's the case, it processes and broadcasts the data (lines 40–44) so that MapActivity can update its navigation map (line 10). MapActivity also stores the data on a database if location tracking is enabled (lines 11–13). Background location sensing is disabled only when MapActivity is destroyed (lines 20–24 and 51–54), which happens when users exit the application.

This design works well in many situations but can cause problems. For example, when users enter an area with weak GPS signals, the application might continue discarding noisy location data. This continual but useless location sensing can quickly drain the phone battery. Another problematic situation arises when users switch MapActivity to the background without enabling location tracking. In such cases, even if GPSService obtains precise location data, the data will be used only to render the navigation map, which is completely invisible. Again, this wastes battery energy.

To fix the bug, developers can tune down the application's location-sensing frequency or temporarily disable location sensing in problematic scenarios.

GUI-Lagging and Memory Bloat Bugs

This example, simplified from a Firefox bug,[1] relates to frequently invoked callbacks.

Android applications are event-driven programs; a set of callbacks defines their major functionalities. The list view, for example, is a widely used GUI widget for displaying scrollable data items (such as an email list). Figure 2b shows an abstraction of a list view (Firefox's tab tray). Each list item represents a browser tab and contains two elements: a webpage icon and a webpage title label.

Triggering bugs is a critical step before diagnosing and fixing them.

To render the list items, developers define a callback getView(). At run time, when users scroll the list view, getView() is continually invoked for constructing new items. The callback typically conducts two operations:

- In *item inflation*, the callback parses the list item's layout configuration files and constructs its GUI element tree.
- In *inner view update*, the callback traverses the list item's GUI element tree to retrieve specific elements for content updating.

However, file parsing and tree traversing can be time-consuming, especially when list items have hierarchical inner structures. Frequently conducting such operations can significantly reduce a list view's scrolling smoothness.

To improve performance, the Android OS recycles list items that go off the screen during user scrolling. Because list items often have an identical layout, Android applications can reuse the recycled items to render new ones, avoiding two heavy operations on each invocation of getView(). This approach is called the view holder pattern (see http://developer.android.com/training/improving-layouts/smooth-scrolling.html). Unfortunately, many real-world applications don't adopt this good practice.

For example, Figure 2c shows how Firefox developers implemented the tab tray's getView() callback. In this inefficient version, item inflation (line 3) and inner view update (lines 5–9) occur each time getView() is invoked. This hurts the tab tray's scrolling performance. This implementation also fails to reuse items and consumes much memory by continuously inflating list items. Owing to the resulting memory pressure, the garbage collector will run frequently, further degrading the whole system's performance.

Diagnosis Challenges

Diagnosing energy and performance bugs in smartphone applications is time-consuming and painful, and poses three main challenges.

Triggering Energy and Performance Bugs

Triggering bugs is a critical step before diagnosing and fixing them. Unfortunately, triggering energy or performance bugs isn't easy.[1]

First, the bugs often occur only in certain usage scenarios, and exposing them requires complex user interactions. Consider our first bug example. To expose the energy waste, we had to

- launch the application and switch on GPS,

- configure the application to disable location tracking, and
- run the application for a while and then switch it to the background.

Such interaction is nontrivial. In reality, users interact with applications in many ways, so it's hard to predict which user interactions might trigger energy inefficiency or performance degradation.

In addition, triggering these bugs could require external stimulus. In the energy bug example, exposing the energy waste might require simulating the physical environment, such as poor GPS signals, which is a nontrivial endeavor.

Finally, these bugs might be triggered only under a sufficient workload. For example, to trigger Firefox's GUI-lagging and memory bloat bug, we must open several browser tabs before scrolling the tab tray.

Judging Energy Inefficiency or Performance Degradation

Energy bugs might waste energy silently, and performance bugs might degrade performance gradually. Such bugs rarely cause immediate fail-stop consequences (such as a crash). This makes judging their existence and extent difficult.

Developers often adopt three judgment strategies:[1]

- Make the judgments manually by running an application and observing its energy consumption and performance.
- Compare an application with similar applications to check whether its energy efficiency and performance are comparatively satisfactory.
- Rely on engineering experience. For example, many developers assume an application suffers from performance bugs if it can't handle a user event within 200 milliseconds.[1,3]

However, these strategies either require nontrivial manual effort or haven't been clearly defined. This makes energy and performance diagnosis less systematic and difficult to automate.

Diagnosis Adequacy and Effort

Diagnosing energy and performance bugs often requires considerable effort.[1] For example, to understand the root cause of our example energy bug, developers must analyze how the application uses GPS data in many scenarios, including these:

1. The GPS data are precise, and the application is running in the background with location tracking enabled.
2. The GPS data are precise, and the application is running in the background with location tracking disabled.
3. The GPS data are continuously noisy, and the application is running in the background with location tracking enabled.
4. The GPS data are continuously noisy, and the application is running in the background with location tracking disabled.

Following such analyses, developers might realize that

- battery energy is completely wasted in scenarios 2 through 4 because all the collected GPS data are either discarded or used to render an invisible map; and
- in scenario 1, the application has slightly better data utilization because it stores data for future use, but battery energy is still wasted in rendering an invisible map.

Similarly, to diagnose Firefox's GUI-lagging and memory bloat bug, developers must test how quickly Firefox responds and how much memory it consumes under different workloads.

Our examples are simplified. Diagnosing real-world bugs might require analyzing many more scenarios. So, it's important to study how to improve diagnosis efficiency and effectiveness. For example, developers might want to analyze a minimal set of critical application usage scenarios to quickly understand the energy and performance bugs' root causes. This would definitely boost their productivity.

State-of-the-Art Diagnosis

Here we review some of the most important energy and performance diagnosis techniques.

Energy bugs might waste energy silently, and performance bugs might degrade performance gradually.

Measurement and Estimation Techniques

Researchers have designed many techniques to measure or estimate smartphone applications' energy consumption and performance. For example, vLens,[6] eProf,[7] and PowerTutor[8] estimate energy consumption of Android applications or system components (such as GPS). In particular, vLens can quickly calculate the energy consumption of fine-grained source code entities such as statements.

Such information can help developers effectively locate energy hot spots in their applications and optimize them accordingly. Although such techniques are applicable to general smartphone applications, Arvind Thiagarajan and his colleagues designed a framework specifically for smartphone browsers to measure the energy used for rendering webpages and their elements (such as Cascading Style Sheets).[9]

Researchers have expended great effort to identify common patterns of energy and performance bugs.

Researchers have also tried to design similar techniques for performance estimation. For example, Mantis constructs precise performance models for Android applications and helps estimate their execution time on given inputs to pinpoint performance bottlenecks.[10]

Event Profilers

Developers have long used profilers to diagnose software energy and performance bugs;[11] researchers have tailored these techniques for smartphone platforms.

For example, ARO (Application Resource Optimizer) monitors cross-layer interactions—such as user events at the application layer and network packets at the system layer—to disclose inefficient radio resource usage, which commonly causes energy waste and performance degradation.[12] AppInsight helps instrument smartphone applications' binaries to identify long latency execution paths.[13] Panappticon identifies performance issues arising from inefficient platform code or problematic interactions among applications.[3] Most recently, SunCat logs the events in a test run and summarizes event repetition patterns to help developers understand and predict performance problems.[11] Such techniques suit in-house testing and can help developers reason about energy and performance problems' root causes.

Pattern-Based Analyzers

Researchers have expended great effort to identify common patterns of energy and performance bugs. On the basis of these patterns, they've designed dynamic and static code analyzers.

ADEL (Automated Detector of Energy Leaks)[5] and GreenDroid[4] locate Android application energy bugs caused by ineffective use of high energy-cost program data (such as network and sensory data). These dynamic analyzers execute an application and track program-data transformation, propagation, and consumption to locate problematic scenarios in which an application fails to effectively use that data (as in Figure 2a).

Static analyzers perform diagnosis without executing an application, thus requiring no test effort (generating reusable test cases for event-driven programs is nontrivial). State-of-the-art techniques often scan an application's source code or binary to locate energy and performance problems. For example, Abhinav Pathak and his colleagues' technique detects energy bugs caused by the forgotten release of wake locks (which keep smartphones awake).[14] Lint, a popular static analyzer in the Android Studio SDK, detects a range of energy and performance bugs. Our PerfChecker can detect eight patterns of energy and performance bugs and provide actionable diagnostic information.[1] (Android Studio developers integrated an enhanced version of our view holder violation checker into Lint; see https://developer.android.com/sdk for more information.)

End-User-Oriented Diagnosis

The techniques we've discussed so far are primarily for developers, but end-user-oriented techniques also exist. For example, eDoctor can correlate system and user events (such as configuration changes) to energy-heavy execution phases.[15] It can thus help end users troubleshoot abnormal battery drains and suggest repairs (such as a configuration rollback).

Carat shares the same goal as eDoctor but adopts a collaborative, big-data-driven approach.[2] It collects run-time data (for example, the active apps and device model) from a large community of smartphones to infer energy usage models. It thereby provides users with actionable advice on improving smartphone battery life. It can provide useful feedback without necessarily needing to profile much of a user's smartphone data.

Such user-oriented techniques can also give developers useful diagnostic information. Carat can tell

TABLE 1

The case study's commercial applications.

Name	Category	Version	Downloads* (in millions)	Reported warnings	True violations
Reddit is fun	News & Magazines	3.1.13	1 – 5	8	2
Wechat	Communication	5.1	100 – 500	18	12
BBC News	News & Magazines	2.5.2	5 – 10	3	0
Sina Weibo	Social	4.2.6	5 – 10	43	10
Flipboard	News & Magazines	2.2.7	100 – 500	18	10
Facebook	Social	6.0.0.28.28	500 – 1,000	1	0
LINE	Communication	4.0.1	100 – 500	5	2
Skype	Communication	4.6.0.42007	100 – 500	13	5
Dropbox	Productivity	2.3.12.10	100 – 500	5	1
Twitter	Social	5.2.2	100 – 500	16	4

*We counted downloads from the Google Play store only.

developers whether their applications would cause energy waste on certain smartphone models. eDoctor can tell them whether their applications would suffer from energy bugs under some configurations or whether new versions have energy and performance regression.

Discussion

These techniques have certain limitations.

Developers often use estimation and measurement techniques to identify energy and performance hot spots. However, simply knowing an application's energy cost or response time might be inadequate for effective optimization. The key diagnostic information developers need is whether the consumed energy or performed computation is necessary. For example, one energy measurement technique might identify an application component using GPS for navigation as an energy hot spot, even though it's efficiently using the consumed energy to provide a smart service. Further research might study how to analyze such cost–benefit relations.

In addition, profilers can generate large profiles[1,11] that contain considerable redundant and useless information. Effective profile aggregation, simplification, and visualization techniques are highly desirable to improve developer productivity.[1] Besides, another open question is which information is critical to collect during profiling to effectively diagnose energy and performance bugs.

Finally, although code-pattern-based analyzers support a range of bug patterns, the root causes of many complex real-world energy and performance bugs are unclear.[1] In addition, analyzers such as ADEL require test cases, but we don't yet know how to effectively and efficiently construct test cases to manifest energy or performance bugs.

A Case Study

We applied the view holder checker in PerfChecker to 10 popular commercial Android applications to see whether PerfChecker can provide useful diagnostic information to developers. We chose our own tool because it's easier for us to preprocess reported issues before communicating with developers. (Static analyzers inevitably generate false warnings; pruning them helps ensure that developers aren't overwhelmed with useless information.)

Table 1 shows basic application information. These applications frequently fetch the latest cloud data to interact with users. We obtained their installation files (.apk) from the Google Play store and transformed them to Java bytecode for analysis. (PerfChecker doesn't require an application's source code for analysis, but if the source code is available, it can highlight code that might cause energy inefficiency or performance degradation.)

PerfChecker analyzed each application in a few seconds and reported a set of warnings. We manually validated the warnings by checking the corresponding source code, decompiled from Java bytecode. Owing to obfuscation, this

ABOUT THE AUTHORS

YEPANG LIU is a PhD student in the Hong Kong University of Science and Technology's Department of Computer Science and Engineering. His research interests include software engineering, software testing and analysis, and mobile computing. Liu received a BSc in computer science and technology from Nanjing University. Contact him at andrewust@cse.ust.hk.

CHANG XU is an associate professor in Nanjing University's State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology. His research interests include software engineering, software testing and analysis, and pervasive computing. Xu received a PhD in computer science and engineering from the Hong Kong University of Science and Technology. Contact him at changxu@nju.edu.cn.

SHING-CHI CHEUNG is a professor of computer science and engineering at the Hong Kong University of Science and Technology and a director at the Hong Kong R&D Centre for Logistics and Supply Chain Management Enabling Technologies. His research interests include program analysis, testing and debugging, big-data software, cloud computing, the Internet of Things, and mining software repositories. Cheung received a PhD in computing from Imperial College London. Contact him at scc@cse.ust.hk.

Chang Xu and Shing-Chi Cheung are the contact authors for this article.

obtained source code was of low quality. So, we conservatively considered a reported warning as a true violation only when we were highly confident that we understood the source code and that the violation was indeed problematic.

The last column in Table 1 shows the number of true violations after inspection. PerfChecker found real performance threats for eight applications. We reported these violations to the corresponding developer teams; we received enthusiastic confirmation and acknowledgment from the Reddit is fun, Flipboard, and LINE teams. Their comments included these:

We really appreciate your valuable comments and suggestions on improving LINE. We would like to pass your comments to the relevant departments, where they may be used for future versions of LINE. —LINE Customer Support

It feels great to be tested by your tool, and it is awesome to connect with bright developers this way. We want to explore more interesting stuff around Android development with you. —Flipboard Customer Support

When analyzing Twitter, we found violations of the view holder pattern in the latest version of the Samsung Mobile SDK (1.5 Beta1), which Twitter uses. Our manual examination later verified these violations. We reported our findings to Samsung developers. They were quite interested and generally agreed that improving this SDK's performance would benefit both application developers and end users (see http://developer.samsung.com/forum/board/thread/view.do?boardName=SDK&messageId=256618 for details):

With such powerful hardware, developers got lazy and started employing bad habits. … If we have better SDK [for example, improving their performance], we have happier developers and happier end users. —Samsung developer

These findings confirm that diagnosis tools such as PerfChecker can help developers find energy or performance optimization opportunities.

We hope that, in the future, research communities and industries will design useful techniques to help developers combat energy and performance bugs in their smartphone applications. Energy and performance diagnosis will surely become increasingly important as mobile Internetware continues to integrate itself into people's daily lives.

Acknowledgments
This research was funded by the Research Grants Council (General Research Fund 611813) of Hong Kong and the National Basic Research 973 Program (grant 2015CB352202) and National Natural Science Foundation (grants 61472174, 91318301, 61321491, and 61361120097) of China.

References

1. Y. Liu, C. Xu, and S.C. Cheung, "Characterizing and Detecting Performance Bugs for Smartphone Applications," *Proc. 2014 Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 1013–1024.
2. A.J. Oliner et al., "Carat: Collaborative Energy Diagnosis for Mobile Devices," *Proc. 2013 ACM Conf. Embedded Networked Sensor Systems* (SenSys 13), 2013, article 10.
3. L. Zhang et al., "Panappticon: Event-Based Tracing to Measure Mobile Application and Platform Performance," *Proc. 2013 Int'l Conf. Hardware/Software Codesign and System Synthesis* (CODES+ISSS 13), 2013, pp. 1–10.
4. Y. Liu, C. Xu, and S.C. Cheung, "GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications," *IEEE Trans. Software Eng.*, vol. 40, no. 9, 2014, pp. 911–940.
5. L. Zhang et al., "ADEL: An Automated Detector of Energy Leaks for Smartphone Applications," *Proc. 2012 Int'l Conf. Hardware/Software Codesign and System Synthesis* (CODES+ISSS 12), 2012, pp. 363–372.
6. D. Li et al., "Calculating Source Line Level Energy Information for Android Applications," *Proc. 2013 Int'l Symp. Software Testing and Analysis* (ISSTA 13), 2013, pp. 78–89.
7. A. Pathak, Y.C. Hu, and M. Zhang, "Where Is the Energy Spent inside My App? Fine Grained Energy Accounting on Smartphones with eProf," *Proc. 7th ACM Euro. Conf. Computer Systems* (EuroSys 12), 2012, pp. 29–42.
8. L. Zhang et al., "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," *Proc. 2010 Int'l Conf. Hardware/Software Codesign and System Synthesis* (CODES+ISSS 10), 2010, pp. 105–114.
9. N. Thiagarajan et al., "Who Killed My Battery: Analyzing Mobile Browser Energy Consumption," *Proc. 21st Int'l Conf. World Wide Web* (WWW 12), 2012, pp. 41–50.
10. Y. Kwon et al., "Mantis: Automatic Performance Prediction for Smartphone Applications," *Proc. 2013 Usenix Ann. Tech. Conf.* (ATC 13), 2013, pp. 297–308.
11. A. Nistor and L. Ravindranath, "SunCat: Helping Developers Understand and Predict Performance Problems in Smartphone Applications," *Proc. 2014 Int'l Symp. Software Testing and Analysis* (ISSTA 14), 2014, pp. 282–292.
12. F. Qian et al., "Profiling Resource Usage for Mobile Applications: A Cross-Layer Approach," *Proc. 10th Int'l Conf. Mobile Systems, Applications, and Services* (MobiSys 11), 2011, pp. 321–334.
13. L. Ravindranath et al., "AppInsight: Mobile App Performance Monitoring in the Wild," *Proc. 10th Usenix Conf. Operating Systems Design and Implementation* (OSDI 12), 2012, pp. 107–120.
14. A. Pathak et al., "What Is Keeping My Phone Awake? Characterizing and Detecting No-Sleep Bugs in Smartphone Apps," *Proc. 10th Int'l Conf. Mobile Systems, Applications, and Services* (MobiSys 12), 2012, pp. 267–280.
15. X. Ma et al., "eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones," *Proc. 10th Usenix Conf. Network System Design and Implementation* (NSDI 13), 2013, pp. 57–70.