E-Spector: Online Energy Inspection for Android Applications

Chengke Wang, Yao Guo, Peng Shen, Xiangqun Chen
Key Laboratory of High-Confidence Software Technologies (Ministry of Education)
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
Email: {wangchk11, yaoguo, shenpeng13,cherry}@pku.edu.cn

***Abstract*—Energy consumption is one of the most important aspects of mobile apps. During energy testing, it is important for developers to understand not only the energy consumption rate of an app, but also why energy is consumed. However, existing energy testing tools are more concerned about the accuracy of energy estimation, while typically not providing explanations on why and how exactly energy has been consumed.**

This paper presents E-Spector, an online energy inspection method for Android apps, which can not only visualize the energy consumption of an app in an instant online manner, but also can tell what happened behind each energy hotspot on the energy curve. E-Spector relies on static analysis and app instrumentation to collect the activities from an app execution in real-time. Then it presents the activities on an instant energy curve, such that the user can easily tell what happened behind each energy spike. Experimental result shows that the energy estimation error of E-Spector is less than 10% and its overhead on energy consumption is about 4%. We also show case studies to demonstrate the applicability and effectiveness of E-Spector in energy monitoring, analysis and bug inspection.

***Keywords*-Energy testing, mobile applications, static analysis, Android.**

I. Introduction

Mobile applications (apps for short) have become more and more complicated since the emerging of iPhone and Android-based smartphones. More functionalities and extensive use of sensors have increased the energy consumption rate for most popular apps. As a result, mobile app developers are very concerned about the energy consumption of their apps.

Energy visualization and debugging tools [15], [5], [10], [13] can help app developers understand the overall energy consumption of an app. However, many existing tools are either based on offline models or only provide offline debugging (estimating and debugging the energy issues of an app after execution) capabilities.

In this paper, we present E-Spector, an online energy inspection method that not only provides instant online visualization of energy consumption patterns, but also can tell why and how energy is spent at each time point.

E-Spector relies on an online energy model [14] that can be used to calculate energy consumption based on system utilization data. We obtain power-related data including CPU utilization, Wifi and 3G/4G data traffic, and screen on/off (and brightness information), and use these information to calculate instant energy consumption for each app at runtime. We can break down the system energy for each app including both apps running in the foreground and background services.

In order to reveal the reasons behind the energy consumption, we also instrument the app under test (AUT) to log all its API calls and other related events to reveal the activities performed by the AUT. These events will be synchronized to the energy visualization interface, and displayed to reveal what happened behind each power spike.

In summary, E-Spector achieves the following goals:

- It does not require hardware meters, instead using an online software-based power model to calculate instant power numbers for each app at runtime.
- It provides detailed energy breakdowns (including components such as screen, CPU and network) for each process running on the device, including both apps running in the foreground, and background services as well.
- E-Spector is able to tell at each time point, especially at energy spikes, what really happened in the app by mapping the events and API calls to the energy spikes in almost real-time.

We implement E-Spector for Android smartphones and use a web-based user interface to provide online instant power consumption curves, as well as provide user interaction capability to inspect the reasons of energy consumption at each time point. E-Spector can be used to visualize energy consumption patterns during app execution, without the need of hardware meters, while offering the capability of debugging the energy hotspots instantly.

We evaluate the accuracy and overhead of E-Specter. The results show that E-Spector can provide energy estimation within an error less than 10%, while the energy overhead is less than 4%. We also demonstrate its capability in finding energy hotspots and bugs with a couple of case studies.

II. Methodology

In this section, we introduce our methodology and the design of E-Spector.

A. Reasons of Energy Consumption

Energy consumption is an important aspect to evaluate an app. The energy consumed while apps are running could be influenced by a lot of factors such as hardware components, network condition, OS version, user interaction, etc. From

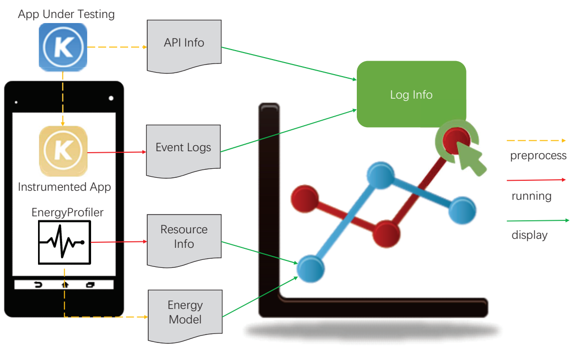


Fig. 1. Overview of the E-Spector working process.

different angles, people are concerned with different factors of energy.

For example, app users mainly focus on the relation of user interaction and energy consumption, while system developers and hardware providers care more about the resource usage. When developing an app, app developers should consider not only the user interaction and hardware power features, but also how to use the hardware resources to deal with user requests.

Thus, when performing energy testing or profiling, app developers need to know which code is executed, why the code is executed and what resources are used. This paper uses these information to explain the reason of energy consumption.

Many prior research works and tools are focused on the user interaction and system resources[11], [15], but few works have analyzed the behaviors of apps. Thus, we propose an energy inspection system, E-Spector, to display the energy consumption of running apps, and also reveal the reasons of the energy consumption in an instant manner.

B. Overview of E-Spector

The main goal of E-Spector is to display instant energy consumption statistics of all apps running on a smartphone, while being able to tell why and how the energy is consumed. In order to be able to visualize energy consumption without affecting the power dissipation on the device, we choose to connect the device to a PC server that will provide visualization and inspection services to the developer.

The architecture of E-Spector is shown in Figure 1. We first perform static analysis on the AUT and instrument it such that it can print log information while running on the testing device. We connect the smartphone to a desktop with ADB and run the instrumented AUT on the device. The logs contain both device utilization information, which will be used to calculate energy numbers, and app events and API calls, which will be synchronized with power traces to explain power surges or spikes at each point on the power curve.

C. Energy Estimation

We do not want to use a hardware power meter such as Monsoon [1] to measure the instant power numbers because it is obviously not widely applicable. Thus we use a software-based energy estimation model, which relies on device utilization number collected by an energy profiler that runs as a service on the smartphone.

TABLE I
INFORMATION COLLECTED BY THE ENERGY PROFILER.

Info	Acquiring Method
CPU	/proc/[pid]/stat
Network	WifiManager & TelephonyManager
Traffic	TrafficStats
Screen	Settings.System.SCREEN_BRIGHTNESS
Apps	ActivityManager
Battery	Intent.ACTION_BATTERY_CHANGED
Screen	Intent.ACTION_SCREEN_ON/OFF
Network	android.net.conn.CONNECTIVITY_CHANGE

TABLE II
ENERGY MODEL FOR THE NEXUS S SMARTPHONE.

Hardware	Parameter	Energy Model
CPU	Utilization(u)	$161.6 * u^2 + 97.5 * u + 6.0$
WiFi	Traffic Speed(t)	$0.5 * t$
Cellular	Traffic Speed(t)	$0.8 * t$
Screen	Brightness(b)	$0.6 * b + 112.0$

Table I shows the list of information it collects, which includes CPU utilization, network traffic data and screen brightness, and also the voltage of battery, etc. The information in the first five rows is collected periodically. In order to obtain balance between accuracy and overhead, we select a frequency of 2Hz as the sampling frequency. The rest of the information collection is event-driven. We record relevant data with time stamps only when events occur.

E-Spector is then able to calculate energy numbers based on a device energy model we built based on the V-Edge power model [14]. We use V-Edge to calculate the power dissipation of the device based on battery voltage, and use a regression method to build the energy estimation model. Table II shows the resulting model for the Nexus S smartphone, which was used in our experiments.

D. Collecting App Events

We then instrument each app to record all the events in order to reveal what happens during app execution. We build an app analysis tool to analyze each app and instrument it to record all events related to energy consumption.

Unlike traditional computation tasks, mobile apps are mostly interactive. Figure 2 shows the event-driven model for Android apps. The event stream consists of a list of events, such as user click event, timer event, network message event, etc. When an event occurs, the framework will notify the app if it is listening to the event. Then the app starts to run from the callback function, which be can be regarded as an "event entrance". Table III shows the types of different "event entrances".

While the "entrance" explains "What happens", the code executed by the "entrance" tells "What does the app execute". As analyzing all the code could become difficult for complex apps, we focus on only the APIs related to energy.

As we know, energy consumption is related to hardware resource usage. Besides the CPU and screen usage, which are

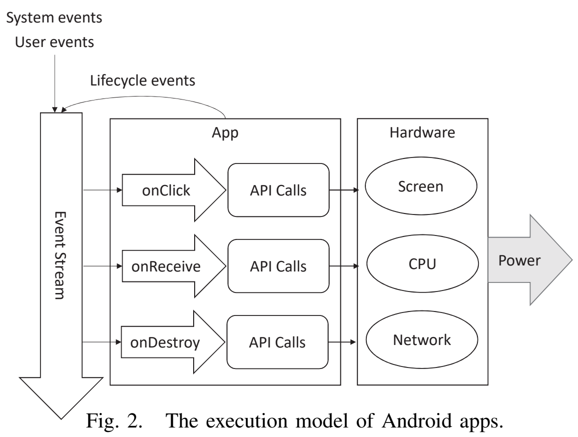


Fig. 2. The execution model of Android apps.

TABLE III
TYPES OF "ENTRANCES".

Type	Example
User Event	onClick, onScoll
System Event	onReceive
Lifecycle Event	onResume, onDestory

hard to infer from the source code, we are more concerned about the resource and related APIs in Table IV. We call these APIs resource-related APIs. Through the information of resource-related APIs, developers could analyze which resource is used and where the energy is consumed.

When analyzing an app, we first identify all the "event entrances", and perform instrumentation at the beginning of each "entrance". We then search for the API calls along with the Call Graph and Control Flow Graph to instrument and record all the events as listed in Table IV.

For example, the app of Kugou Music has 3,178 "event entrances" and 3,622 resource-related API calls among all the 35,991 API call instructions. The number of data-base API is 3,129, which accounts for the most of resource-related APIs. While the second ranked is network APIs, with a number of 381.

E. Data Synchronization and Visualization

In the testing phase, we run the instrumented app on the testing device. While the app is running, the instrumented code will print logs on "entrances" and API calls, while the energy profiler will print logs on resource usage of the app.

These data will be synchronized based on timestamps, and transferred to the E-Spector service for power calculation and display. According to the power model of the testing device, we could calculate instant energy consumption of the app. Then we display the energy curve of the AUT with events and API information synchronized to the energy curve. The user can then click on any point on the power curve to learn the events corresponding to the energy pattern.

III. IMPLEMENTATION

In this section, we introduce the implementation details of E-Spector.

TABLE IV
RESOURCE-RELATED APIS.

Type	APIs
PowerManager	android.os.PowerManager.*
Sensor	android.hardware.*
Network	android.net.* org.apache.http.* java.net.URL
Bluetooth	android.bluetooth.*
Location	android.bluetooth.*
Media	android.media.*
Database	android.content.SQLite.* android.database.sqlite.*
I/O	java.io.*

A. App Instrumentation

We first perform static analysis on each AUT in a preprocessing step before running it. The preprocessing step consists of 2 parts: instrumenting all the "entrances" with the code of logging and listing resource-related APIs following each "entrance". The instrumented app will execute on the testing device. The API list is deployed on the PC server, which will be displayed to the user (i.e., app developers) along with the energy estimation results.

We build our static analysis tool based on the open-source project AndroGuard[2]. We use the basic analysis tools in AndroGurad, such as Control Flow Graph, Call Graph, Basic Blocks, etc, to perform our analysis. Our instrumentation does not need the source code of AUT. All we need is the apk file of each app.

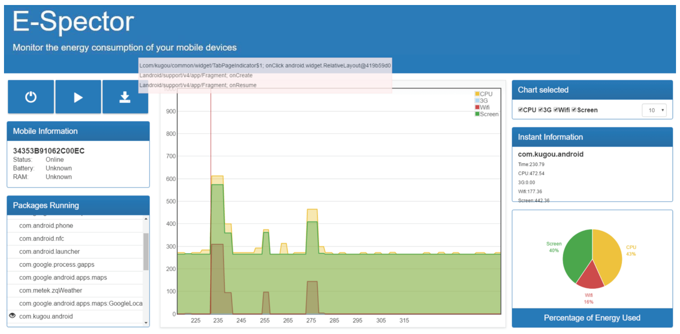
1) Entrance Instrumentation: We decompile the apk file of AUT into smali code with AndroGuard and then perform instrumentation on the smali code.

We first find all the "entrances" according to method names and Call Graphs. Specifically, an "entrance" method should not be called by other methods explicitly and have a specific format name, such as "onClick", "onDestory", as shown in Table III.

Then, we insert an "invoke" instruction at the beginning of each "entrance". This instruction invokes a static method for printing logs about the information of "entrances".

2) API information: A straightforward approach to acquire API information is to instrument a log printer before every API call in the smali code. However, the number of API calls is very large and the overhead may be intolerable. Thus, we introduce a method to estimate the API calls via static analysis.

For each "entrance", we walk through the Call Graph and find all "invoke" instructions that call resource-related APIs, as listed in Table IV. We save the API lists on the PC server. In the testing phase, if the user/developer clicks a sampling point on the power curve, E-Spector will search the "event entrance" around the point and display the corresponding resource-related API list, along with the power curve.


E-Spector

Monitor the energy consumption of your mobile devices

```
Lcom/kugou/common/widget/TabPageIndicator$1; onClick android.widget.RelativeLayout@419b59d0
Landroid/support/v4/app/Fragment; onCreate
Landroid/support/v4/app/Fragment; onResume
```

Mobile Information

34353B91062C00EC

Status:	Online
Battery:	Unknown
RAM:	Unknown

Packages Running

- com.android.phone
- com.android.nfc
- com.android.launcher
- com.google.process.gapps
- com.google.android.apps.maps
- com.metek.zqWeather
- com.google.android.apps.maps:GoogleLoca
- com.kugou.android

Chart selected

✓CPU ✓3G ✓Wifi ✓Screen 10

Instant Information

com.kugou.android

Time:230.79
CPU:472.54
3G:0.00
Wifi:177.36
Screen:442.36

Fig. 3. Screenshot of E-Spector, showing the power curve of Kugou Music.

B. Testing and Output

Before running an instrumented app, we connect the testing device to a PC server via a USB cable. The information of energy consumption, resource usage and events is logged on the device while testing.

On the PC server, we deploy a TomCat web application to obtain these data using the "adb logcat" command in a synchronized manner. The web application estimates the energy consumption and display it with events and API information on a web interface. The interface was refreshed every 0.5 seconds, which is the same frequency as we collect power-related statistics.

Since E-Spector uses an online power model based on resource usage, USB charging would not affect the energy estimating results for AUT. E-Spector can also avoid the interference of the log service and other apps on the testing device, because the energy of the whole device is broken down to each app and service running on the device.

C. User Interface

We build a web interface for E-Spector, which can be used in any popular web browser. Figure 3 shows a screenshot of E-Spector. The information displayed by E-Spector includes:

- The instant energy curve for an app, which is updated in real-time as the app executes on the device. The energy numbers can be broken down into different power consuming components including CPU, Network and Screen, which can be toggled on/off.
- Users can choose to inspect the energy consumption for any app/process from the list of all running processes, including background services running on the smartphones.
- For all instrumented apps, the user can click on the energy curve to see what happened behind each time point. If the user wants to know what causes an energy spike, moving the mouse to the point before the energy spike will reveal all the events and API calls.
- We also show other useful information and statistics, including phone status, device utilization and energy breakdown by components, etc.

IV. Evaluation

In this section, we evaluate the accuracy of our energy model and the overhead of E-Spector.

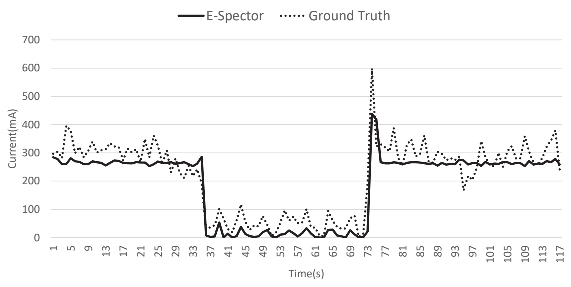


Fig. 4. Accuracy of E-Spector.

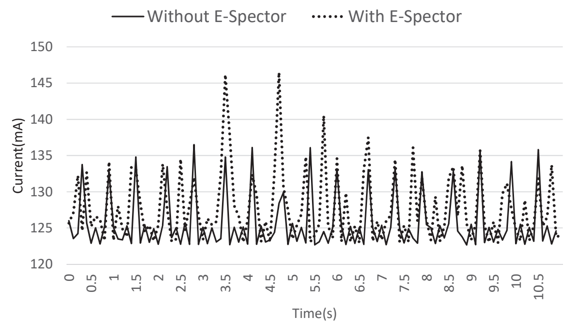


Fig. 5. Overhead of E-Spector.

A. Accuracy

To evaluate the accuracy of our power model, we compare the estimated energy of the whole device to the measured power value using the popular hardware power meter, the Monsoon Power Monitor [1].

We run our energy profiler on a Nexus S smartphone, which is connected to the Power Monitor. After running some test cases, we estimate the energy consumption of the smartphone through the data collected by energy profiler, and compare it with the ground truth measured by Power Monitor.

Figure 4 shows the comparison result of a test case in which the screen of the smartphone is turned off and then turned on again. The result indicates that our energy model represents the actual power consumption numbers closely, while the average error is within 10%.

B. Overhead

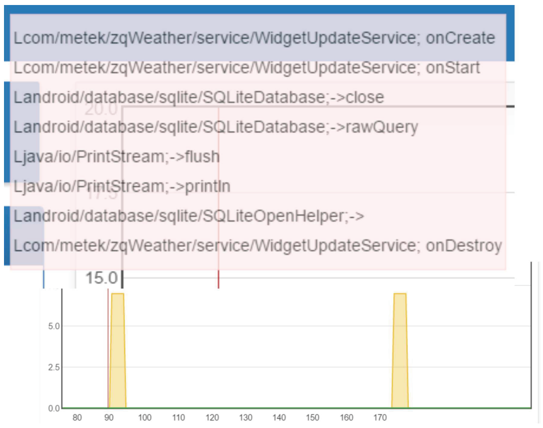
To evaluate the overhead of E-Spector during app execution, we compare the power consumption with and without running the energy profiler and app instrumentation.

Since the testing need to be conducted twice(or more) and it is difficult to reproduce a complex execution condition, we just choose the simplest condition, not running any other apps, to evaluate the overhead.

Figure 5 shows the comparison result, which suggests that the energy overhead of logging is less than 4%.

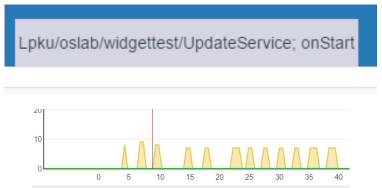
V. Case Studies

We present three different case studies to illustrate the capability and effectiveness of E-Spector in energy analysis and bug detection.



```
Lcom/metek/zqWeather/service/WidgetUpdateService; onCreate
Lcom/metek/zqWeather/service/WidgetUpdateService; onStart
Landroid/database/sqlite/SQLiteDatabase;->close
Landroid/database/sqlite/SQLiteDatabase;->rawQuery
Ljava/io/PrintStream;->flush
Ljava/io/PrintStream;->println
Landroid/database/sqlite/SQLiteOpenHelper;->
Lcom/metek/zqWeather/service/WidgetUpdateService; onDestroy
```

Fig. 6. Test case for a background app (Zhiqu Weather).



```
Lpku/oslab/widgettest/UpdateService; onStart
```

Fig. 7. Test case for an energy bug (Anki-Android).

A. Foreground Apps

Figure 3 shows a test case of Kugou Music, a popular music app, running in the foreground. It shows that the power curve rises up at about 232th seconds on the x-axis.

When we click the sampling point at the beginning of the rise, it shows that the cause is a user click event at the UI component of "android.widget.RelativeLayout@419b69d0". After this click event, the app would create a new Fragment, which explains the reason of the energy surge.

B. Background Activities

Figure 6 shows part of the screenshot of a testing case using a weather app (Zhiqu Weather), which runs in the background. It shows that the power curve rises up for about every 90 seconds. Clicking the sampling point at the beginning of one rise, we see that the app starts a "WidgetUpdateService" and calls a series of APIs about database accesses and I/O operations.

Thus, we can conclude that the app accesses the database periodically and consumes a certain amount of energy in the background during each access.

C. Energy Bugs

We then show an example using Anki-Android, an open source app on GitHub[12]. In a previous version of the app, some users complained about the energy problem. Then the developers debugged the app and found a lot of unnecessary "UpdateService" invocations.

We run the Anki-Android app version with energy bugs using E-Spector. It can successfully detect the exact problem, as shown in Figure 7.

D. Summary

These case studies suggest that E-Spector can help app developers perform energy testing and debugging on their apps.

- E-Spector can estimate the power consumption of an app during execution. Beside total power consumption, E-Spector can also estimate the detailed power breakdown of each app on each hardware component.
- E-Spector can help developers to find out the reasons behind energy hotspots. E-Spector shows the API calls of the AUT at each hotspot. Through these API calls, developers can analyze the energy cause (user inputs, system events, etc.) and foreground/background behaviors of the AUT (accessing network, updating UI, IO operations, etc.).
- E-Spector can detect energy bugs and locate them in either bytecode or the source code of the AUT. The results can help developers to locate and fix the corresponding energy bugs and optimize the power consumption of the AUT.

VI. Discussions and Future Work

In this section, we discuss the limitations of our work and present possible future research directions.

A. Accuracy of the Energy Model

In our energy model, we only considered the top three hardware energy consumers: CPU, screen and network (both WiFi and cellular network). In most instances, the CPU, screen and network combined consume almost all of the energy of the whole device. According to the evaluation results, the error rate of our model is less than 10%, which is acceptable considering that we only need a rough estimation to find out the energy spikes.

However, in some special scenarios, some other components also consume significant amount of energy. For example, when running map and navigation apps, the device may consume most energy in the GPS sensor. When running a pedometer, the device may consume more energy in a series of sensors including accelerometer or gyroscope.

In order to further improve the accuracy, we want to refine the energy model by adding more components, such as GPS, GPU, microphone in the future.

B. Detailed API Information

E-Spector can show the resource-related API list around a sampling point. In consideration of the high overhead of dynamic logging, we chose to generate the API lists via static analysis. Compare to API logging, the static estimation results is not as accurate. It can only tell which APIs the app calls, but include no information of the frequency of API calls or the parameters passed to the APIs.

Although the API lists are enough to tell the behaviors of AUT, we hope to give more accurate and detailed information about the resource-related APIs. One possible way is to estimate the number of API calls via analyzing the branches, loops and call graph of the AUT.

C. Remote Testing Suport

Currently, E-Spector is implemented as a local service, with the testing device (smartphone) connected to the testing server (PC) with USB. However, we deliberately implemented the user interface in a web-based manner such that it can be easily extended to support remote testing.

In a remote testing scenario, we can implement E-Spector as a web service. Developers can upload the AUT for static analysis and download our energy modeling tool for building energy model, collecting data, etc. While testing with these tools, the testing data can be transmitted to the server and the results would be displayed in a web page.

Furthermore, we could also test the AUT on devices connected to the server (cloud testing), by providing a web-based UI for developers to interact with the AUT. This can release the burden of the developers to test their apps on different real devices as these devices can be provided by the testing service.

VII. Related Work

Many recent work are focused on energy consumption of mobile devices and mobile apps. Researchers have developed energy models for hardware components including network [3], screen [4], as well as mobile device energy models such as CABLI [17] and Devscope [7].

Based on the hardware power models, several research work are focused on the energy analysis of mobile apps. For example, Appscope [15] analyzes energy of apps based on DevScope via their hardware utility. Eprof [11] estimates the energy consumption of apps via system call traces. eLens [5] tests apps in a simulator and estimates their energy consumption via instruction models.

Some research work extend these energy models to analyze the energy features of mobile apps. For example, Maiti et al.[9] analyzed the efficiency of energy usage of mobile apps. Linares-Vasquez et al.[8] analyzed the relation between API behavior and energy consumption.

There are also many energy analysis tools available. PowerBooter [16] is a power monitor for android devices based on resource usage model. It only provides the power of the whole device. GreenMiner [6] is a power testing tool for Android apps. It measures the power of the whole device and breaks it down to each app. AppScopeViewer is an energy testing tool for mobile apps based on AppScope [15]. JouleUnit [13] is an energy testing tool for java applications based on JUnit. It can provide unit testing at method-level. Carat [10] is a crowdsourcing energy diagnosis tool for mobile apps. It collects energy information of mobile apps from a large number of users and compare the energy consumption of the same app among different users to find energy bugs.

These energy analysis tools could estimate the energy consumption of mobile apps, but they typically cannot help users/developers understand why their apps are consuming energy instantly in a visualized interface, which is the focus of E-Spector.

VIII. Concluding Remarks

In this paper, we have presented E-Spector, an online energy inspection method, which can not only display how much energy an app consumes instantly, but also reveal the reason of the energy consumption at each time point. We have evaluated E-Spector through experiments to demonstrate its accuracy and applicability. We also performed case studies to show that E-Spector is effective in energy monitoring, analysis and bug location.

Acknowledgement

This work was partially supported by National Key Research and Development Program No. 2016YFB1000105 and National Natural Science Foundation of China No. 61421091.

References

[1] Monsoon power monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.

[2] Androidguard. Androguard. https://github.com/androguard/androguard.

[3] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *IMC'09*. ACM, 2009.

[4] M. Dong, Y.-S. K. Choi, and L. Zhong. Power modeling of graphical user interfaces on oled displays. In *DAC'09*. ACM, 2009.

[5] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *ICSE'13*, 2013.

[6] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky. Greenminer: A hardware based software energy consumption framework. In *MSR'14*, 2014.

[7] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha. Devscope: a nonintrusive and online power analysis tool for smartphone hardware components. In *CODES+ISSS'12*, 2012.

[8] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, New York, NY, USA, 2014. ACM.

[9] A. Maiti and G. Challen. The missing numerator: Toward a value measure for smartphone apps. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile '15, New York, NY, USA, 2015. ACM.

[10] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *SenSys '13*, 2013.

[11] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *EuroSys '11*, 2011.

[12] Ramblurr. Anki-android. https://github.com/Ramblurr/Anki-Android.

[13] C. Wilke, S. Götz, and S. Richly. Jouleunit: A generic framework for software energy profiling and testing. In *GIBSE '13*, 2013.

[14] F. Xu, Y. Liu, Q. Li, and Y. Zhang. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *NSDI '13*, 2013.

[15] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. Appscope: Application energy metering framework for android smartphones using kernel activity monitoring. In *USENIX ATC'12*, 2012.

[16] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *CODES/ISSS'10*, 2010.

[17] X. Zhao, Y. Guo, Q. Feng, and X. Chen. A system context-aware approach for battery lifetime prediction in smart phones. In *SAC '11*, 2011.