TIDE: A User-Centric Tool for Identifying Energy Hungry Applications on Smartphones

Tuan A. Dao, Indrajeet Singh, Harsha V. Madhyastha, Srikanth V. Krishnamurthy, Fellow, IEEE, Guohong Cao, Fellow, IEEE, and Prasant Mohapatra

Abstract—Today, many smartphone users are unaware of what applications (apps) they should stop using to prevent their battery from running out quickly. The problem is identifying such apps is hard due to the fact that there exist hundreds of thousands of apps and their impact on the battery is not well understood. We show via extensive measurement studies that the impact of an app on battery consumption depends on both environmental (wireless) factors and usage patterns. Based on this, we argue that there exists a critical need for a tool that allows a user to: 1) identify apps that are energy hungry and 2) understand why an app is consuming energy, on her phone. Toward addressing this need, we present TIDE, a tool to detect high energy apps on any particular smartphone. TIDE’s key characteristic is that it accounts for usage-centric information while identifying energy hungry apps from among a multitude of apps that run simultaneously on a user’s phone. Our evaluation of TIDE on a test bed of Android-based smartphones, using week-long smartphone usage traces from 17 real users, shows that TIDE correctly identifies over 94% of energy-hungry apps and has a false positive rate of < 6%.

Index Terms—Energy management, user-centric, smartphones.

I. INTRODUCTION

While smartphones are evolving with richer capabilities and more powerful hardware, their batteries are not keeping up. Coupled with the explosion in the number of applications1 for smartphones, this trend has left users distressed at keeping up. Coupled with the explosion in the number of applications, this has left users distressed about how long their phone’s battery lasts even after a full recharge. A report in 2012 [3] says that “Despite activities such as web browsing, watching videos, and using downloadable apps have become (sic) an everyday part of smartphone use, their impact on battery performance is largely excluded from the data published by manufacturers.”

Need for a user-centric app profiling tool: While there exist tools that try to quantify the energy consumption of smartphone apps, they are not user-centric. The target for these tools is software developers who want to check for power inefficiencies in their products before release. These tools either require the instrumentation of the smartphone with specialized external equipment (e.g., a power meter), or require modifications to the smartphone’s operating system (OS). A typical user cannot perform either. In addition, these tools need to be run continuously to track an app’s operations, and hence consume significant energy themselves.

Instead, it is desirable to have a tool that is capable of reporting which apps on a user’s phone dominate battery consumption. This tool should not simply focus on detecting apps that have energy bugs [19] or ignore user-specific factors that influence battery drainage (e.g., as in [18]); for each user, it should identify apps that consume a disproportionate amount of energy on that user’s phone. When run on a particular user’s phone, one could envision this tool as roughly categorizing every app as energy-hungry, energy-thrifty, or energy-moderate, based on how the app is used by the user and the environment in which it is used. Once energy hungry apps are identified, a user can reduce her use of or cease to use such apps when needed.

Challenges: Unfortunately, developing such a user-centric tool to detect high energy apps on smartphones is a hard problem. Since normal users will be reluctant to install modifications to the smartphone OS (this voids the phone’s warranty), the identification of energy-hungry apps must be based on information exported by the OS to the application layer. This information is however insufficient for directly measuring the precise amount of resources, and hence energy, consumed by any specific app. First, the OS only reports aggregate resource usage metrics to the application layer. Second, at the application layer, one can only measure the durations between instances when the residual battery life decreases by 1%. During any one such interval, there are typically several apps running simultaneously on the phone.

On the other hand, offline calibration of an app’s energy consumption is insufficient, since the determination as to whether a specific app is energy hungry critically depends on how and in what setting the app is used. First, battery drainage is affected by a variety of factors, including the features of the device, the processing invoked by each app, and network conditions. Thus, the power consumed by the same app can significantly vary across different settings. In addition, different users may interact with an app in different ways (e.g., the energy consumed by a video sharing app can differ based on whether the user views videos of high or low quality).
Therefore, an app that is energy hungry on one user’s phone may not be so on another’s.

Due to all of the above factors, it is a significant challenge to tease out the apps that are the real culprits with respect to energy drainage on a particular user’s phone.

**Our Contributions:** In this paper, we first undertake an extensive measurement study on a testbed of 22 Android phones. Our study demonstrates how differing network conditions, device features, and usage patterns influence the energy consumed by apps. Our study also highlights the challenges that need to be addressed in building a user-centric tool as described above. These challenges include the need to (a) sample the information exported by the OS in an effective way, and (b) filter noisy data due to the typical coexistence of multiple active apps on a smartphone. Finally, as our main contribution, we design, implement, and evaluate TIDE, a user-centric tool that can be readily installed and used by real users for identifying the energy hungry apps specific to their usage profiles. TIDE is itself implemented as a smartphone app, which continually performs lightweight monitoring of a user’s usage of apps and the resources that these apps consume. This information is then fed to a classifier which efficiently categorizes apps as high, moderate, or low consumers of the phone’s battery. In our evaluation of TIDE, based on a detailed emulation of traces of usage patterns from 17 volunteer users, we find that it correctly estimates the level of energy consumption for 225 out of 238 apps. Furthermore, TIDE delivers this level of accuracy while imposing only 0.5% of overhead on the average consumption of the phone’s battery per hour.

A preliminary version of this work appeared in [10].

**II. RELATED WORK**

Android provides a battery manager tool [1] which estimates the percentage of battery consumed by each app. It considers the resource consumption of an app with respect to the number of CPU ticks, the number of bytes transferred over the network, the time for which the display was active, etc. It uses a model-based estimate of how much energy is consumed due to the use of a unit of each specific resource (e.g., per CPU tick, per TCP byte transferred) and multiplies this value by the number of units of that resource used by an app. The tool however does not account for several user-specific factors that influence energy consumption per-unit resource, e.g., link quality influences the energy consumed per byte transferred on the network. In Section III, we show via measurements that these factors can have a significant impact on an app’s energy consumption.

Prior efforts on estimating application-specific energy/power consumption can be broadly classified into three major classes.

**User-Centric Tools:** Current tools that try to characterize the power consumed by apps either use offline tests and/or fail to account for one or more factors that affect the battery drainage due to an app. PowerTutor [23] estimates an app’s power consumption due to its interactions with different hardware components (e.g., LCD, GPS, WiFi, and 3G interfaces) based on a regression model. Unlike TIDE, a) PowerTutor itself consumes high power since it queries the OS at a high sampling rate, b) it depends on per-app resource consumption information, which is not readily available in newer versions of Android, and c) it requires offline calibration for every device type.

Carat [18] uses crowdsourcing to estimate the energy impact of an app; it compares battery drainage statistics with and without the app. This approach however fails to account for both user-specific app usage and user-specific network conditions, which can affect battery behavior, as we show later. Further, unlike Carat, TIDE only runs on user’s devices and performs all analysis locally on any particular device, i.e., there is no need for either offline calibration or server-side aggregation. Falaki et al. [12] also highlight the impact of user-specific factors on battery consumption; they suggest that ‘diversity’ across users in terms of their app interactions can influence battery drainage rates. However, they did not focus on the development of a tool such as TIDE for user-specific estimation of app energy consumption.

**Determining Energy Bugs:** Another body of work tries to detect energy bugs in apps. Yoon et al. [22] use Kprobes, a Linux kernel module in Android, to track native system calls for detecting anomalous behaviors. Pathak et al. [19] design a framework that needs access to system calls and applications’ native code, in order to detect energy bugs. However, such tools require an external power meter for energy measurements and/or the modification of the underlying OS. eDoctor [15] identifies abnormal drain issues on phones by comparing app behaviors with well known good versions. Their goal is different from ours; we seek to identify apps consuming energy on individual users’ phones, regardless of whether the high energy consumption is due to a bug.

**Characterizing Energy Consumption by Individual Components:** Finally, there are efforts that try to assess the power consumed by smartphone components (as opposed to apps). Shye et al. [21] build a model which estimates the breakdown of power consumption in different hardware components, based on a set of apps. However, their estimation does not work for new apps not present in this set. WattsOn [16] is an energy emulator that uses power models developed offline for individual smartphone components. However, to emulate an app’s usage pattern on WattsOn, we would need to capture a user’s interactions with the apps on her phone, and collecting this information would require rooting the phone; most users are unlikely to permit this. Most smartphones use battery models to provide the user with coarse-grained battery usage statistics; Sesame [11] argues that such models must be generated based on measurements using individual smartphones, rather than offline in a lab setting. Carroll and Heiser [9] instrument the components of an Android device offline, and measure the power consumed by each while running various benchmarks. Balasubramanian et al. [8] focus specifically on the energy consumed by the network using different technologies. eCalc [14] estimates the energy consumption of the CPU when an app is executed by profiling the app’s binary. None of these efforts look into developing a user-centric tool for identifying energy hungry apps.
III. SHOWCASING USER-CENTRIC APP BEHAVIORS

In this section, we present an extensive measurement study to demonstrate that user behaviors, network conditions, and even phone features impact the energy consumption of apps. These demonstrate that crowdsourcing (e.g., Carat [18]) cannot accurately account for user-specific app behaviors. We also showcase the limitations of the Android system tool in capturing energy consumption behaviors of apps.

A. Impact of Network Conditions

First, we show that the network types and link qualities significantly affect the energy consumed by an app. We experiment with four HTC Touch 4G phones, each of which uses a different network with different qualities. All the phones use the same email account and we write a script to send emails to the logged in accounts. Emails are sent at high (every 30 seconds), moderate (every 5 minutes rate) or low (every 10 minutes) rates. We turn off the display and all background activities to make sure that the network I/O is the only contributor to battery drain. The phones are notified of new emails via push notification messages. These messages wake up the phones if they are in the sleep state. A pair of phones use 3G connections, while another pair uses WiFi. For the pair of phones on the same network, we put one phone at a location with good signal strength (between -69 and -55 dBm) and the other at a location with poor signal strength (between -103 and -97 dBm). We fully charge the phones before the experiment and measure the energy consumed after 1 hour.

Results: Fig. 1 shows the battery drainage with each phone in different network conditions. In poor signal conditions, as one might expect, (i) the amount of energy used to transfer packets is higher [16], and (ii) the amount of corrupted packets is significantly higher [17], which causes many packet retransmissions. Thus, the energy consumption is much higher; for example, with a high volume of data, in 1 hour, the phone with poor 3G signal consumes more than 8% of the battery, while the phone with good 3G signal consumes only around 5%.

Thus, these experiments show that the energy consumption of an app not only depends on the amount of network traffic that it sends and receives, but also on the type and quality of the network connection that the user experiences. We repeated the experiments in this section with different pairs of phone models and different network providers, and we still observed qualitatively similar results. We do not report the other results here due to space limitations.

B. Impact of User Behaviors and Phone Features

Beyond variance in network conditions, different users can potentially use the same application quite differently, which can in turn affect that app’s energy consumption.

An Example With Youtube: To demonstrate the impact of user-specific workloads on energy consumption, we perform experiments with YouTube. We play different videos on a smartphone (Dev 1). Videos 1 and 2 are full screen; however, video 1 is of high quality (480p) whereas video 2 is of default (360p) quality. Videos 3 and 4 cover 3/4th of the screen when playing; again, the former is of high (480p) quality and the latter is of normal (360p) quality. We play these videos on Dev1 when the video files are (a) stored locally on the smartphone’s memory card, (b) downloaded over WiFi, or (c) downloaded over 3G. Finally, we repeat case (a) with a different smartphone (Dev2). Dev1 is a Samsung Galaxy SII and Dev2 is a HTC MyTouch 4G phone.

Results: The results of our experiments are shown in Fig. 2. On one hand, with Dev1, we observe that streaming over 3G always consumes the most energy; streaming over WiFi consumes slightly more energy than when playing local files. This reaffirms our previous finding that, depending on the network coverage (3G versus WiFi) enjoyed by the user, the energy consumption of an app can differ.

On the other hand, we also observe significant differences in the energy consumed when playing different videos (all playing on the same device); between the two videos, we see a difference of as much as 20% in terms of the time taken to deplete the battery by 1%. Thus, depending on the video itself (rate of motion, black and white versus color, etc.), its resolution (high quality versus low quality), and the display size, the YouTube app’s energy consumption may vary. As the choice of video, resolution, etc. depend on user preferences and choices, the user’s behavior strongly influences the energy consumption of this application.

Finally, we also observe differences in the energy consumptions across devices when playing the same video file (from local memory). In fact, the difference is as high as 49%; this is primarily due to the differences in the hardware on the two phones. Dev1 uses a Super AMOLED Plus display, which does not require a backlight and is thus, more energy-thrifty as compared to the LCD display on Dev2.

Other Examples: While the above example was with respect to YouTube, other apps also exhibit such multi-modal energy consumption patterns based on their usage.

Music Folder Player: The MusicFolderPlayer app allows a user to either keep the screen on or off when playing music. Depending on which option a user chooses to use, the energy
consumed by this app can vary. Fig. 3 shows the energy consumed by this app in 5 minutes in three different modes. As one might expect, if the screen is on, this app is a high energy app; else, it behaves as a low energy app.

Angry Birds: We next consider a game app and observe varied energy consumption depending on the expertise of the user playing the game. Specifically, we have two users play the Angry Birds game for 10 minutes each. One user, who is well-versed with the game, plays the game constantly and moves to higher levels of play. The other novice user progresses through the game at a slower pace as he takes time figuring out how to play at each level. On a Galaxy SII phone, we observe that the novice user’s usage of the game consumes 0.72 kJ of energy as compared to the 0.91 kJ consumed by the expert user. This amounts to a difference of 26.39% (= 4.8% in terms of the battery percentage consumed) per hour of play.

The Android System Tool Does Not Account for User-Centric Factors: As discussed in Section II, the Android system tool attributes energy consumption to an app based on its usage of specific resources. For each app, the tool records the number of units of each hardware component used by the app. This number is multiplied with the average energy consumption of the corresponding component to estimate the energy consumed by the app due to the use of that component. The sum of these values across all components is the energy consumed by the app. In an Android device, the average power consumption values of the various components (in mAh) are stored in the power_profile.xml file provided by the manufacturer; a shortened version of the file is shown in Table I. Note that the contents of the file are fixed and not updated (the energy information is not re-calibrated) when the environment changes. We see that the average energy used by the WiFi interface in one time unit is shown on line 4. Similarly, line 6 shows the average power used by the cellular interface. It is evident that the network link quality is not accounted for by the Android tool.

Further, from the source code of the tool [1], one can see that while computing the energy consumption due to an app’s network activities, the tool does not differentiate between the app’s use of WiFi and cellular networks. If the total amount of data sent and received by all apps over the cellular and WiFi interfaces are mobileData and wifiData, respectively, then the Android OS computes the average power consumed per byte as \( (3\text{GEnergyPerByte} + \text{wifiEnergyPerByte} \times \text{wifiData})/(3\text{GEnergyPerByte} + \text{wifiData}) \), where \( 3\text{GEnergyPerByte} \) and \( \text{wifiEnergyPerByte} \) are obtained from the power model (Table I). For each app, the OS then computes the energy consumed due to network activities by simply multiplying the average energy per byte computed above with the total amount of data transferred by the app over all interfaces.

Since network conditions are not taken into account, the tool may not always yield accurate outputs. To validate this hypothesis, we conduct an experiment wherein three different applications read the same file in the memory card and send the content to our server. The apps are run on the same device and use exactly the same source code but send data in different network settings. We turn off the WiFi connection on the device and run App1, thus causing it to send data over the 3G network. Subsequently, with WiFi turned on, we run App2 at a location near an access point such that the device enjoys good signal strength. Finally, App3 is run at a location with weak WiFi signal strength. The Android system tool shows App1, App2 and App3 consume 2%, 3% and 3% of the phone’s battery, respectively. These numbers are far from what we get from direct measurement with a power meter; the measurements show that the three apps consume 6%, 1% and 2.5% of the battery, respectively.

These experiments show that results from the Android System tool do not capture changes in the energy due to specifics of the usage environment (the actual conditions) in which the user applications are executed; in other words, the tool is not user-centric.

Solutions Such as Carat [18] Cannot be Easily Extended to Account for User-Centric Behaviors: By its very nature, crowdsourcing (the basis for Carat [18]) ignores user-specific characteristics of apps. We downloaded and tested Carat on our own Android phones for a week. Carat classified two of our appsâ€”Google Maps and Skypeâ€”as energy hogs. However, we had only used Google Maps for a very short time during the study and it barely consumed any energy. Further, we used Skype with audio only and over WiFi, because of which it consumed little energy; Carat classified it as a energy hog since most users used it with video. Other users of Carat have experienced similar issues [6]. One can think of extending Carat to check if an app is an energy hog on a particular userâ€™s phone by comparing energy consumption on that phone across periods when the app was active/inactive. We did examine this approach with a rudimentary implementation but found that it mis-classified low energy apps as high energy ones. This was primarily because such apps often executed simultaneously with other high energy apps, and it was difficult to isolate their behaviors in terms of energy consumption. Further, the approach did not account for multimodal behaviors of apps (described later in section IV-C).

We address these challenges in TIDE.
Summary: Our experiments show that the energy consumption of an app depends on several factors: (i) network conditions experienced by the user, (ii) her usage patterns, and (iii) her device’s characteristics. This highlights the need for user-centric classification of apps, i.e., it must account for the user’s typical profile in terms of the above factors.

IV. CHALLENGES IN DESIGNING TIDE

Having motivated the need for a user-centric tool for identifying high energy apps, we now highlight the challenges in building such a tool on the Android platform. Based on our preliminary studies, we believe that iOS has similar limitations and poses similar challenges.

A. Lack of OS Support

Developing TIDE would be easy if smartphone OSes monitored all the activities or resource usage of every app and exported this information to all other apps. However, as one would expect, smartphone OSes either do not record the necessary details for energy efficiency or hide this information because of security concerns. As a result, smartphone OSes complicate the development of TIDE in several ways.

Lack of Precise Energy Usage Information: In prior work, researchers have either instrumented smartphones with devices such as the Monsoon meter [7], or plugged special sense resistors into hardware components on the phone to measure the energy consumed [9]. Such setups were then used to either measure the energy consumption of a single app in isolation or to build power models of individual hardware components. In contrast, for our goal of developing the TIDE app, smartphone OSes do not provide such precise measurements of energy consumption. The only energy-related information exported by the OS is the battery level, which is reported with a 1% granularity.

Thus, TIDE’s estimation of energy consumption by apps has to be based on its observation of when the phone’s battery level changes, i.e., drops by 1%. Hereafter, we refer to each time period in which the battery drains by 1% as simply an interval. In Section V-B, we elaborate on how this information can be captured on the Android platform.

Lack of app-Specific Resource Usage Information: A potential approach to side-step the limitation of the lack of precise energy information is as follows. For each type of phone, one can construct an accurate power model for every hardware component (e.g., LCD display, network interfaces, and CPU) in every environment (e.g., LCD power consumption as a function of brightness and 3G power consumption as a function of signal strength). Discounting the fact that gathering such a power model will be cumbersome, TIDE can then estimate the energy consumption of any particular app by 1) monitoring the environment in which the phone is used and the app’s usage of each of the phone’s components, 2) for every component, multiplying the app’s usage of that component with the power coefficient value of the component, and 3) summing up this value across all components.

Unfortunately, such an approach would be hard to implement on today’s smartphone OSes since, for many of the phone’s hardware components (e.g., display, GPS), the OS only provides aggregate resource usage for the whole phone and not for each individual app. For example, to track LCD usage, Android permits an app to register for the events corresponding to the screen being turned on or off. While this would enable TIDE to determine the time for which the phone’s LCD was on, it cannot determine how much of this usage can be attributed to each app on the phone. Thus, when many apps are running simultaneously, though the OS lets an app query for the list of all other apps active on the phone, it would be difficult for TIDE to partition the aggregate resource consumption across these apps.

While the OS does track and export per-app usage of some resources, there are complications involved even in their use. For example, Android maintains two files—/proc/uid\_stat/[uid]/tcp\_snd and /proc/uid\_stat/[uid]/tcp\_rcv—which list the amount of TCP traffic sent and received over the network (both 3G and WiFi) by an app; here uid is the unique identifier of the app on the device. However, this feature is optional and is disabled in some phone models (e.g., Galaxy Nexus and Sony Ericsson Xperia X10 Mini Pro); thus, on such phones, a user will have to root the phone and install a new kernel for the OS to be able to track TCP traffic. Moreover, power consumption of the network interfaces also depends on packet arrival rates, which determine the energy drainage during transmission tail periods [20].

The only resource whose usage TIDE can track on a per-app basis is the CPU. On Android, every running app has a unique process ID (pid) and its CPU usage is provided in the file /proc/[pid]/stat. The CPU usage time is measured in ‘system ticks’. In Android, the number of ticks per second is usually set to 100 [2].

Overhead of Querying Information: One way to cope with the availability of only aggregate resource usage information would be to have TIDE query the OS frequently (e.g., every second). TIDE can then attribute all the resource consumption in the last second to the app that was actively used in that period. On the Android OS, TIDE can discover the app currently being used by querying the OS for the foreground app. However, frequently querying the OS for both the foreground app and the usage of all resources can itself consume high energy. Fig. 4 shows the power consumed over an hour when querying Android on a Galaxy Nexus phone at different rates; we perform this measurement on a phone where only our querying application was active and all other apps were disabled. If we query every second, TIDE would itself consume 3.2% of the battery in an hour on the tested phone.
Consuming over 3% of the phone’s battery every hour would make TIDE prohibitive for use. On the other hand, if we query every 30 seconds, the querying application only consumes 0.5% of the battery in an hour; however, this leads to the challenges discussed next.

B. Challenges in Associating Energy Consumption to Specific Apps

It is difficult to tease out app-specific energy consumption from the inherently noisy data that the OS provides when queried less frequently (e.g., once every 30 seconds). To show this, we not only perform select experiments on our smartphones, but also rely on measurements from the smartphones of real users. Specifically, we distributed an Android app to 17 volunteer users with IRB approval (details later in Section VI-A).

Co-Existence of Multiple Active Applications: A major obstacle in attributing the energy consumed to a specific (say target) app is that there are many co-existing active apps when the target app is running; in our measurements, almost all intervals contain multiple concurrently active apps. There are several reasons for this. First, there are background processes (including system processes) that continuously run on a phone. Second, users often switch between multiple apps; for example, a user may switch between checking email, posting on Facebook, and listening to music within a short time. Finally, to reduce load times for recently used apps, Android keeps an app in memory even after use; it kills the app only when the phone’s memory has to be devoted for other apps. Thus, many recently used apps are included in the list of active apps reported by the Android OS.

To determine the apps in the active list that actually contribute to energy consumption, we need to estimate their activity levels. One way to estimate an app’s activity level is based on the app’s CPU usage (the OS can be queried for this information); note that an app consumes a non-trivial number of CPU ticks even when it sends/receives data over the network. Simply eliminating all apps that have consumed zero CPU ticks in the interval is insufficient because some apps may use a little CPU only to periodically poll for updates; these apps are unlikely to contribute much to battery drainage in that interval. Hence, we need to use a threshold to filter out apps that were largely dormant. However, determining a good threshold for CPU ticks is challenging; this threshold will depend on the smartphone architecture and on an application’s implementation.

In Fig. 5, we plot the CDF of the number of simultaneous apps (from the dataset for one user from our study) with different thresholds for CPU ticks. We see that if a low threshold is used, we cannot filter out apps that run for short periods. For example, with a threshold of 20 ticks, 60% of the intervals have more than 5 simultaneously active apps. However, if the threshold is too high, a majority of apps are filtered out, some of which may be energy hungry. Note that this profile (how many simultaneous apps are active in an interval) is user-specific.

Work Delegation Between apps: Another major hurdle in attributing energy consumption to specific apps is work delegation, which is possible on Android devices. Specifically, the functions of one app are delegated to another app. One example of an app that receives many such delegated functions is the Mediaserver app. Every media app delegates data retrieval operations to Mediaserver; once Mediaserver has received data over the network, the data is exported to the Mediaserver app. A naive energy monitoring tool would hold Mediaserver responsible for the energy consumed due to network transfers. Based on this information, since Mediaserver is a system application that cannot be completely disabled, the user may continue to use YouTube as normal and drain her phone’s battery. To be accurate, TIDE must identify YouTube as the main culprit for energy drainage in this case.

C. Multi-Modality of Apps

Finally, the determination of energy hungry apps is complicated by the various modes in which a single app can function. There are several apps that consume high energy only when they use a high amount of a specific resource(s). As we show later in Section VI-C.2, YouTube and Pandora are two examples of multi-modal apps. YouTube’s classification as an energy hungry app depends on the network quality, whereas the Pandora app consumes high energy only while the display is on. Therefore, TIDE must have the capability to classify apps under different usage scenarios.

V. TIDE: ARCHITECTURE AND IMPLEMENTATION

We next describe the architecture of TIDE and provide the details of our implementation. Since TIDE seeks to capture user-centric attributes, it runs on every user’s own smartphone and identifies energy hungry apps based on the user’s profile. Specifically, it inspects the correlation of apps’ occurrences and high energy/resource usage periods on the phone. TIDE seeks to identify the energy-hungry apps by long term profiling; thus, the more the user invokes an app, the higher the accuracy of TIDE’s classification of the app.

We wish to point out here that when we classify apps with TIDE, we focus on the energy consumption due to the CPU, the network interfaces, and the display. However, the framework that we use in TIDE is extensible to account for other resources. For example, one resource whose use is known to lead to high energy consumption is the GPS. Similar to techniques that we describe in this section, TIDE can identify an app’s energy consumption due to use of the GPS by correlating periods when the GPS is turned on with
intervals in which the app either has significant CPU activity or is in the foreground.

A. System Architecture

Fig. 6 depicts the architecture of TIDE; it consists of two main components: Process Monitor and App Classifier.

1) Process Monitor: TIDE’s first component profiles app behaviors on the user’s phone. Recall that the smartphone OS does not provide fine-grained information with regards to energy consumption; the only information that the OS exports are the durations between instances when the battery level drops by 1% (intervals). The Process Monitor runs in the background and keeps track of these intervals. At the end of each interval, it queries the OS for the resource usage information in that interval. Specifically, it obtains information relating to (i) the duration for which the screen was on during the interval, and (ii) the aggregate network usage in that interval (in bytes). Within each interval, the Process Monitor also queries the OS periodically (once every $\tau$ seconds) for a list of the running apps and the CPU usage of each app in the preceding $\tau$ seconds. The information collected is stored in the phone’s SD card and is later processed by the App Classifier.

Adaptive Sampling: In TIDE’s querying of the OS once every $\tau$ seconds for a list of active apps, there is an inherent trade-off in choosing a value for $\tau$. On one hand, the larger the value of $\tau$, the more coarse grained the information obtained from the OS. As a result, the query returns co-existing apps more often than not. Further, it cannot accurately map resource usage to apps; this makes it especially difficult to capture multi-modal behaviors. On the other hand, Process Monitor can query the OS more often (e.g., $\tau = 1$ second), but this increases the energy overhead imposed by TIDE.2

To address this trade-off in TIDE, we use an adaptive sampling approach. Specifically, Process Monitor queries the OS more often when the battery drainage is heavy (i.e., when it observes short intervals) and less often when battery drainage is minimal (long intervals). The basis for this is that, in order to identify energy hungry apps, fine grained information is required only during those periods when the rate of energy consumption is high. In more detail, after a high-drainage interval is seen, the Process Monitor switches to fine-grained sampling, and $\tau$ is set to 1 second. Typically, during high usage periods, short intervals appear in bursts (we observe this in our experiments) and thus, the next interval is also likely to be a short one. On the other hand, after $k$ long

2Note that the number of co-existing apps with $\tau = 1$ sec is drastically lower than when $\tau = 30$ secs, but apps may still co-exist.

Algorithm 1 TIDE’s algorithm for app classification

1: //Phase 1
2: for all app $x$ do
3: $s :=$ Fraction of intervals containing only $x$ that are short
4: $l :=$ Fraction of intervals with $x$ that are long
5: if $s \geq f_H$ then
6: Mark $x$ as $HIGH$
7: else if $l \geq f_L$ then
8: Mark $x$ as $LOW$
9: end if
10: end for
11: //Phase 2
12: \forall unclassified app $x$, calculate $conf(x)$
13: while $\exists$ unclassified app $x$ with $conf(x) \geq \gamma$ do
14: Find app $x$ that has the highest confidence
15: Mark $x$ as $HIGH$
16: Remove all short intervals that contain app $x$
17: Recalculate confidence values of unclassified apps
18: end while
19: //Phase 3
20: Multi-mode candidates = apps classified in phase 1 $\cup$ all unclassified apps
21: for all multi-mode candidate app $x$ do
22: Calculate $conf(x,r)$ for app $x$ and resource $r$
23: end for
24: while $\exists$ tuple $(x,r)$ with $conf(x,r) \geq \gamma$ do
25: Find tuple $(x,r)$ that has the highest confidence
26: Mark app $x$ as $HIGH$ when it intensively uses resource $r$
27: Remove short intervals with app $x$ and high utilization of $r$
28: Recalculate confidence values of remaining tuples
29: end while
30: Mark all unclassified apps as $MODERATE$
the intervals in which this app is seen to be active (details in section V-B). Among these intervals, if the fraction of intervals that are short and have no other concurrent app with $X$ is greater than a threshold $f_{fr}$, then we mark $X$ as an energy hungry application. Similarly, among the intervals in which an app $Y$ occurs, if the fraction of long intervals is greater than a second threshold $f_{fl}$, then we consider $Y$ to be a battery-thrifty application.

However, the above procedure by itself is insufficient to classify all apps. This is because, as discussed earlier in Section IV, many intervals include several concurrently active apps. Hence, if a short interval includes many active apps, we cannot attribute the high energy consumption in that interval to any one app with certainty.

**Phase 2: A Greedy Algorithm to Handle Co-Existing apps:**
To account for multiple active apps in short intervals, we use a greedy algorithm in the second phase of the App Classifier’s execution. In a nutshell, the larger the fraction of short intervals among the intervals in which an app is active, we can have greater confidence in declaring the app as energy hungry. The algorithm identifies energy hungry apps in the decreasing order of associated confidence. Once a particular app is marked as energy hungry, we greedily attribute all the energy consumption on the phone to this app in all the short intervals in which the app is active.

In more detail, let us define the confidence value for an application $X$ being energy hungry, $conf(X)$, to be the probability that an interval which contains $X$ is also a high battery drainage interval. The App Classifier deems an application $X$ as energy hungry if $conf(X)$ is more than a threshold (say $\gamma$). Once app $X$ is marked as energy hungry, the classifier discards all high battery drainage intervals that contain the app from future consideration; this essentially attributes the high battery consumption in these intervals to app $X$. The classifier thereafter repeats the procedure of identifying the app with the next highest confidence value ($\geq \gamma$) based on the intervals that have not yet been discarded. We repeat this process until no apps with a confidence value $\geq \gamma$ remain.

In the above algorithm, one can envision cases where a high energy app $Y$ gets filtered out simply because it also appears with another high energy app $X$. However, first we argue that these cases are rare in practice (as also seen in our experiments). When a high energy app is being executed, the phone drains energy very quickly (in less than 2 minutes in our setting). In such a short interval, the likelihood that the user uses and switches between several high energy apps (such as games, video streaming apps, etc.) is really low; such apps usually require user involvement. This decreases the likelihood that such cases happen to begin with.

Second, TIDE fails to identify $Y$ from being a high energy app only if $Y$ is not frequently used by the user. In such cases, $Y$ may not be executed in isolation by the user in the near future; if the user uses app $Y$ frequently, in the long run (say, 1 week), there will be intervals in which $Y$ does not co-occur with other high energy apps (e.g., $X$) and will thus be correctly classified. We show this later in section 6.3.3.

Finally, one might expect the user to stop the usage of app $X$ because of TIDE’s classification. This then precludes the simultaneous execution of $X$ and $Y$ and thus, the high energy usage of $Y$ will be discovered by TIDE much more quickly and efficiently.

To improve the effectiveness of TIDE in such cases, viz., when app $X$ and app $Y$ are almost always executed together, they can be considered as a tuple $\{X, Y\}$ that causes high drainage on the phone. We defer such optimizations to future work.

**Phase 3: Dealing With Multi-Modal apps:** Multi-modal apps that exhibit different energy consumption rates in different execution modes may however have a low confidence value, since intervals containing an app $X$ combine data from all of $X$’s modes. To handle such cases, in App Classifier’s final phase, we also define the confidence value for a tuple of application $X$ and resource $R$, $conf(X,R)$, to be the probability that an interval which contains $X$ and has high utilization of $R$ is a high battery drainage interval. Using $conf(X,R)$, TIDE is able to detect apps that are energy hungry only in execution modes where a specific resource (e.g., network, screen) is intensively used. This information will allow a user to decide how to (or rather how not to) use certain apps, e.g., the user may decide against uploading videos to Facebook if TIDE determines that Facebook’s high energy consumption is correlated with heavy network usage.

In TIDE, the environmental factors and user behaviors are fully captured when classifying apps. Specifically, it detects high energy apps by capturing the correlation between app activities and the energy drainage rate on the phone. The drainage rate implicitly accounts for how the user interacts with the apps, as well as how much and in what conditions resources are consumed. If the same “amount of” resource is consumed in “favorable” conditions (e.g., good network, low quality video), the drainage rate would be lower, and vice versa. Thus, even though we only provide coarse grained classification information, the results are fully user-centric and accurately capture energy consumption of the apps on the specific user’s phone.

### B. Implementation Details

Next, we describe our Java-based implementation of TIDE for Android phones.

**Process Monitor:** TIDE captures a phone’s battery usage by monitoring what are called “Intent” messages on the Android platform. The Android OS broadcasts notifications about important system events to apps (with the right permissions) through Intents. TIDE registers for the ACTION_BATTERY_CHANGED event, and by means of the associated Intent message that it receives, determines when the residual battery level drops by a percent. TIDE also registers for the ACTION_POWER_CONNECTED and ACTION_POWER_DISCONNECTED events; with these, it is notified when the phone is plugged in or unplugged from the power outlet. Lastly, TIDE registers to be notified of the ACTION_SCREEN_ON and ACTION_SCREEN_OFF

---

3We are working towards releasing TIDE on the Google Play store; a preliminary version can be found at http://bit.ly/1lnp51f.
works well in practice. We also observe that minor variations in the CPU ticks threshold do not affect TIDE’s accuracy.

**Detecting app LCD Usage:** Detecting active apps by just using a CPU threshold however is not enough, because an app can keep the screen on without using the CPU. Hence, we consider active apps in an interval to be the ones which either consume CPU or run in the foreground. By using adaptive sampling, in high energy intervals, we sample for the foreground app every second and thereby capture the LCD usage of apps. In other intervals, TIDE can only capture the foreground app once every \( \tau = 30 \) secs; thus, we can miss out on the apps that use the display at other times in between. However, this is not of consequence since, regardless of whether or not the app uses the display, it consumes low energy in such long intervals.

Once the active apps are determined as above, the App Classifier executes the classification algorithm described in Section V-A.2. Here, we need to choose appropriate thresholds for 1) the long and short intervals in which an app has to appear, in order to be classified as a low or high consumer of energy (referred to as \( f_L \) and \( f_H \) in Section V-A.2), and 2) the \( \text{conf}(X) \) or \( \text{conf}(X,R) \) values associated with any app \( X \). We experiment with different values for these thresholds with different user workloads and on different types of phones. To keep the false positive rate low, we find that \( f_L = f_H = \frac{3}{4} \) and \( \gamma = 0.66 \) works well. With lower thresholds, false positive rates are high; higher thresholds do not significantly reduce the false positive rate further, without also increasing the false negative rate.

**Accounting for Work Delegation:** Finally, whenever an app \( X \) (e.g., YouTube) appears in the same interval as another app \( Y \) (e.g., MediaServer) to which \( X \) delegates work, we simply attribute all of \( Y \)’s resource usage in that interval to \( X \). If two apps that delegate work to \( Y \) simultaneously appear in an interval, we attribute each app with half of \( Y \)’s resource usage. A similar approach can be applied to cases with more than two apps. However, in our user traces, we never observed any interval wherein more than two different apps delegated work to the same app within an interval.

**Defining High and Low Drainage Intervals:** TIDE enables a user to choose the thresholds that define HIGH and LOW drainage intervals based on the user’s preferences and expectations. However, for evaluating TIDE’s performance, we define intervals in which 1% of the battery is drained in less than 2 minutes as HIGH and intervals in which 1% of battery is drained in more than 6 minutes as LOW. This is based on running known high energy (e.g., Skype) and low energy apps (e.g., MusicFolderPlayer) on our phones and noting how long they take to consume 1% of the battery; for example, Skype takes 1.8 minutes whereas MusicFolderPlayer takes around 6.5 to 9 minutes.

**When is Resource Usage High?** When multi-modalities of apps are considered, we need to construct tuples of the form \([X, R]\) to represent the presence of an app \( X \) in a high battery drainage interval in which resource \( R \) is also heavily utilized. Thus, a question that needs to be answered is: “when should the usage of resource \( R \) be considered high?” To answer this question, we perform
measurements using known resource hungry applications with respect to each resource. Specifically, for network usage, we measure the traffic generated by YouTube while watching 20 random video clips of HD quality, and by Skype during a video conference. We choose these specific apps as they are known to result in high network usage. We measure the volume of traffic while the apps are executed on 4 different devices and in different network conditions. In all our measurements, the apps generate $\geq 5.5$ MB of traffic per minute, and hence, we set this to be the threshold for high network usage. Similarly, we consider 5 different 3D games (known to be CPU intensive) to set the benchmark for high CPU activity. We find that all of these games consumed more than 1000 CPU ticks per minute. Thus, we set this to be the threshold for high CPU activity. Like with the CPU ticks threshold we use to identify active apps in an interval, here too we linearly scale this threshold for high CPU usage based on the CPU frequency of the phone. As discussed earlier, with adaptive sampling we can capture LCD usage of apps in high energy intervals.

VI. EVALUATION

Next, we present a detailed evaluation of TIDE based on experiments conducted on a testbed of Android phones. Our experiments are driven by traces gathered from the phones of several users. We use a Monsoon power meter for all energy measurements on our testbed.

A. Collection of Real User Workloads

To capture user-centric behaviors, we collect data from 17 volunteer users. Our study has been IRB approved by our institution. Since a phone has to be rooted in order to gather the data that we need (note that using TIDE itself does not need the rooting of phones), we handed out rooted smartphones to our volunteers after swapping the phones’ SIM cards with the SIM cards from the users’ own phones; this obviates the need for volunteers to root their own phones. To ensure consistency, we matched the model of the phone handed out to a user to the user’s own phone. The volunteers used our phones for their daily use for a week. The collected user traces are used to generate realistic workloads on our Android testbed for establishing the ground truth (as discussed later in Section VI-B). Furthermore, we run TIDE on these phones to get its output assessments.

1) Capturing User Interactions: On every phone handed out to our volunteers, we installed a background process that captures all of the user’s interactions with her phone. Capturing these interactions in a manner that allows for accurate replay is however a significant challenge. For example, a user’s interaction with a web page may be hard to replay since the web page’s content may vary over time. Moreover, some apps (e.g., Facebook) may require the user to be logged in, which we cannot emulate during trace replay. To capture interactions in a manner that enables high fidelity trace replay, we adapt the technique proposed by Gomez et al. [13] to capture user input events with low overhead. To do so, we poll the smartphone’s system files for events generated by the user’s interactions.

Apart from storing user input events, we also need to associate these events to apps. Unfortunately, system files that log user input events do not provide information about the app with which the user is interacting. Therefore, for every interaction, we also capture the foreground app on the phone by querying the ActivityManager class. Since the number of user input events is large (e.g., a simple swipe event on the phone can generate more than 10 records in the /dev/input/event2 file), in order to minimize overhead, we query the OS for the foreground app only on “key released” records; these records are generated when the user releases her fingers from the screen or from a button. Note that, in order to gather the above information, root privilege on the phone is necessary. Hence, collection of such information is possible only for our purpose of gathering user traces and not as part of TIDE’s operation.

We store all of this information in a file so that we can later replay on our testbed all of a user’s interactions with every app used by the user. By emulating different network conditions, we can build the ground truth information with regards to the “user-centric” energy consumed by every app.

2) Capturing User-Centric Resource Usage Patterns: For privacy reasons, many users were wary of their interactions being captured; in fact only two of our volunteers allowed us to log these interactions. Thus, we seek a different way to estimate the app-specific energy consumption on such users’ phones. For this, we capture the resource usage on the phone when an app is running and mimic these utilizations on the same phone to represent the app’s execution.

To determine the CPU usage of an app, we read the file /proc/[pid]/stat (pid is the process ID of the app). To capture network traffic, we run tcpdump on the phone to captures all packets going through all network interfaces. Periodically, we run a modified version of netstat (provided by the Busybox tool set [5]) to record all the ports used by each app. We then correlate tcpdump’s output with the app to port mapping in order to map every packet to the corresponding app. To measure the time for which an app uses the screen, we access the system logcat information on the phone to estimate how long an app stays in the foreground. Again, note that these methods for capturing app-specific usage of the network/display is possible only with root privileges, and hence, such information is not available to TIDE.

B. Building the Ground Truth

To evaluate the accuracy of app classification with TIDE, we first need ground truth information. Specifically, for every app used by a particular user, we need to determine whether or not the app is indeed energy hungry from that user’s perspective. Generating this ground truth is non-trivial in itself. In real user workloads, apps do not run in isolation. Furthermore, user-centric factors such as the signal strength of the 3G network experienced at different times are not known. Therefore, to generate the ground truth, for every app used by one of our users, we run the app in isolation as per that user’s usage pattern of that app (other apps are turned off), and emulate different network conditions.
Similar to the drainage intervals, we assign one of three labels—HIGH, MODERATE, or LOW—to each app depending on how long it takes the app to consume 1% of the battery. While thresholds for determining these labels can be defined by user preferences in practice, we consider what we believe are reasonable thresholds in this study. For the reasons discussed in Section V-B, when replaying apps on a specific phone, we label any app that consumes 1% of the battery in < 2 minutes as HIGH; if this consumption takes > 6 minutes, the app is labeled LOW. We consider apps which consume 1% of the battery in a duration that is in between 2 minutes and 6 minutes as MODERATE. In what follows, for simplicity, we combine both MODERATE and LOW apps and label them as MODERATE, since from a user’s perspective it is not vital to distinguish between them.

1) Replaying User Traces: Replaying User Interactions: As discussed, only two volunteer users let us collect their fine-grained interactions with their phones. We replay these interactions with each app in isolation to quantify the real energy consumed by that app. Replaying app Behaviors Based on Resource Usage: For all volunteer users in our study, we replay the resource usage of each app in isolation, to estimate its energy consumption. Currently, we do not consider replaying multiple applications simultaneously, even though there might be mutual influences between them in terms of power consumption. This is because (i) if multiple apps consume a specific amount of energy together, it is not easy to break down the energy consumption due to individual apps, as each app might consume different resources to different extents, and (ii) when there are multiple apps requesting resource access, it is extremely challenging to replay the resource usage exactly in a dependent manner without modifying the Android OS itself. Thus, we defer this to our future work.

For replaying the network usage of an app, we run a server which generates the same network traffic as identified by tcpdump in the user trace. We emulate varying network conditions to generate the ground truth in different scenarios. As network activities also consume CPU, we record the number of CPU ticks associated with these activities. When replaying CPU usage of an app, we subtract this number of CPU ticks to preclude network activities.

For replaying display usage, we keep the screen on for the same amount of time and with the same brightness level as from the user-trace. One problem with capturing the display’s usage is that, though we periodically query for the screen’s brightness level when an application is running, we do not know the exact content on the screen at specific times. Therefore, in our experiments, we use a static background while replaying an app (the brightness is as per the user’s behavior). We try two extreme settings: (i) a dark and (ii) a relatively white background. Note that this limitation with respect to accounting for the impact of the displayed content on energy consumption is inherent in most of the energy models derived based on resource usage (e.g., [11], [22], [23]).

2) Can Replaying Resource Usage Patterns Capture App Energy Consumption?: Capturing fine-grained user interactions provides high fidelity in the user-centric classification of apps. However, since we have this detailed information only for two users, we assess how trace replay based on resource usage patterns compares to that based on user interactions. For the two users for whom we could capture their interactions with their phones, we estimate the energy consumed by each app (i) first, by replaying user interactions, and (ii) again, separately, by replaying the associated resource usage from our traces. For clarity, we only show the results for 4 apps in Fig. 7; we see similar results with the other apps. We observe that simply using the resource usage provides an estimate of energy consumption that is almost equal to that in the case where we capture user interactions. The use of a relatively white background provides the best estimate; the dark background underestimates the power consumption to some extent. This is to be expected since most apps have bright colored or relatively less dark backgrounds.

C. Evaluating TIDE

We next evaluate TIDE’s accuracy in classifying apps, and we thereafter assess its overhead.

1) App Classification Accuracy: We determine the accuracy of TIDE’s App Classifier first based on ground truth obtained by replaying fine-grained user interactions, and second, based on resource usage information. Note that, on each of our volunteers’ phones, TIDE was concurrently running while we were capturing logs that we later used for trace replay (in order to determine the ground truth for energy consumption of every app).

Accuracy as Compared to Ground Truth Based on User Interactions: The two volunteers, for whom we could capture input events, were seen to use apps under different network conditions; both of these users used Galaxy SII phones. We separated the collected data into 3 sets for each user based on their interactions and network usage; each set contained information spanning at least six hours. Fig. 8 shows TIDE’s accuracy on these datasets, in comparison with the ground truth. Each bar shows the total number of active apps in the respective dataset. The top and bottom parts of each bar show the number of high energy apps and the number of low/moderate energy apps, which TIDE’s App Classifier was able to correctly classify. The middle parts of each bar depict
false positive results, wherein LOW or MODERATE apps are mis-labeled as HIGH, and false negative results, where HIGH apps are mis-labeled as LOW or MODERATE.

False positives typically occur when a low energy app coexists in many of its intervals with other high-energy apps. This can happen for apps that are not frequently used by the user. For example, the one false positive in Fig. 8 corresponds to the case where one of the users was using a music player app for 10 minutes while simultaneously surfing the web. In this case, we associate the music player with a high confidence value due to the web browser’s high energy consumption. If TIDE monitors this user’s phone over a longer period, there are likely to be intervals where the user uses the music player in isolation or only with other LOW apps. TIDE can then be expected to classify the app correctly.

Similarly, false negatives occur when a high-energy app X coexists only with other high-energy apps; when we discard intervals attributing them to these other high-energy apps (with higher confidence values), app X gets filtered out. TIDE then labels such an app as MODERATE. As users use applications for extended periods and increased numbers of times, the coexistence pattern of other apps will vary. As a consequence, the false positive and negative rates can be expected to be lowered over time. We show experimentally that this is the case in Section VI-C.3.

Accuracy With Respect to Ground Truth Based on Resource Usage: Next, we examine the larger dataset from our 17 volunteer users, which includes resource usage information based on their daily smartphone use for a week. Fig. 10 shows TIDE’s accuracy in those 17 datasets, with the results amortized over different network conditions. The representation of the results are in the same form as in the previous case; each bar represents results from a different user’s data. In 7 of the datasets, TIDE was able to classify all the apps correctly. In almost every other dataset, we only obtained either one false positive or one false negative.

Dataset 10 and 17 are the only exceptions where we had two and three false positives respectively; however, all the high energy apps were correctly labeled in this user’s dataset. In summary, TIDE was able to correctly identify 66 out of 70 HIGH energy apps, and incorrectly classified 9 MODERATE apps as HIGH, from among a total of 168 MODERATE and LOW energy apps.

In the above analysis, we find several cases wherein TIDE correctly identifies the same app as HIGH for one user and LOW/MODERATE for another user. For example, TIDE identifies the YouTube app as HIGH for a user who always uses the 3G network on his phone. For another user who typically uses WiFi, TIDE correctly identifies YouTube as a MODERATE app from that user’s perspective. Thus, TIDE is able to accurately account for user-centric factors that cause differences in an app’s energy consumption across users.

Capturing User-Centric app Behaviors: Next, we demonstrate TIDE’s ability to capture the user-centric attributes of apps. Specifically, here we consider apps that change their behaviors from HIGH to MODERATE or vice versa, depending on network conditions.

We conduct in house experiments with five popular apps—Skype, YouTube, the default Android web browser, Angry Birds, and Pandora—on a Galaxy SII smartphone. First, we use each app for at least 15 minutes and capture all of the user’s interactions. Thereafter, we replay all those apps jointly under 4 different network conditions: strong WiFi, weak WiFi, strong 3G/4G, and weak 3G/4G. The reported signal strength from the phone was between -105 and -97 dBm under weak signal conditions, and between -69 and -55 dBm under good signal conditions.

Table II shows the ground truth information and the results with TIDE. The ground truth labels are built by replaying the input events under the appropriate network conditions. Note that here we also experiment with two different thresholds to label an app as HIGH; an app is labeled HIGH if it consumes 1% of the battery (i) in less than 2 minutes in one case, and (ii) in less than 3 minutes in another case. The results demonstrate the low sensitivity of TIDE to the threshold.

In our experiments, Skype is always labeled HIGH, regardless of network conditions. Other apps, such as YouTube and the web browser, change their energy consumption profiles under different conditions. TIDE is able to capture these behaviors. In this experiment, we account for work delegation, and assign the resource usage by Mediaserver to YouTube (or Pandora) when they co-exist in the same interval. Without this, YouTube will always be labeled LOW.

2) Capturing Multi-Modal Apps: We next conduct an experiment to evaluate TIDE’s ability to classify multi-modal apps. Here, we first play Pandora for 1 hour using the 3G network while keeping the screen off. Subsequently, we set the screen at the highest brightness level and continue playing Pandora for the next 30 minutes. After this, we use YouTube for an hour using WiFi (Pandora is now off). Finally, we continue with YouTube but switch to 3G for the last 30 minutes. We keep the screen at the highest brightness level while using YouTube. During the entire experiment, we also have other apps (auxiliary apps) that run simultaneously with Pandora and YouTube. With Pandora, we run an app that executes in

<table>
<thead>
<tr>
<th>Application</th>
<th>Condition</th>
<th>2-minute threshold</th>
<th>3-minute threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skype</td>
<td>True</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Web browser</td>
<td>True</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Pandora</td>
<td>True</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>YouTube</td>
<td>True</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Angry Birds</td>
<td>True</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>M</td>
<td>M</td>
</tr>
</tbody>
</table>

Note: H = HIGH; M = MODERATE
the foreground and simply turns on the display while Pandora runs in the background; here our goal is to see if Pandora is correctly identified as a low energy app. With YouTube, we run an app that receives updates from a Twitter account; our goal is to see if TIDE can accurately capture YouTube’s high energy when the network usage is high. The auxiliary apps are turned on and off at random. When turned on they remain on for a uniformly chosen random period between 3 and 5 minutes; when turned off, they remain in that state for a uniformly chosen period between 7 and 10 minutes. Both of these auxiliary apps continue to run for 2 hours after the Pandora and YouTube apps are terminated. We find that TIDE accurately classifies all of the apps above. Specifically, it finds that: (i) Pandora consumes high energy only when the screen is turned on, (ii) YouTube consumes high energy only if 3G is used, and (iii) both our auxiliary apps consume low energy.

In more detail, the confidence value of Pandora in general, without considering its different usage patterns, is quite low (20% out of 16 intervals). Thus, TIDE classifies Pandora as a MODERATE application. However, when TIDE considers Pandora only in intervals in which the LCD is intensively used, the confidence value of the tuple (Pandora, LCD) is high (80%) and TIDE identifies Pandora as an energy hungry application. As for YouTube, the confidence value in general is low (33% out of 24 intervals). However, considered only when the 3G network is used, its confidence value is 100%; TIDE thus identifies YouTube as a high energy app under high 3G utilization.

3) Accuracy Versus Dataset Size: TIDE monitors user-specific factors (network, screen, CPU ticks) to create a profile of which apps consume high energy and how often, and what resource usage accompanies them. As a result, the longer the observation period, the better TIDE’s accuracy. Fig. 9 shows the impact of the number of observed intervals on the accuracy of TIDE with one of our datasets (results with other sets are similar). With the data collected for 12 hours, there was only one high energy app invoked by the user, and TIDE produced one false positive result. This is primarily because of the limited volume of data used to build the profile. With the data collected for a day, the user used more high energy apps and TIDE was able to detect all 4 of them. The earlier, wrongly classified app is now correctly labeled as MODERATE; however, a new (previously unseen) app is mis-labeled as HIGH. With the data collected for 3 days, no more new high energy apps were detected. Importantly, the mis-labeled app is now correctly labeled as MODERATE. To ensure that the periods are long, but are not influenced by stale behaviors, we set the monitoring period to one week by default. However, the user can choose the period over which TIDE should use data to classify apps (e.g., 1 day, 3 days, or a month).

Next, we compare the efficiency of TIDE in identifying high energy apps with that of the Android System Tool and the popular PowerTutor [23] tool; the latter has more than 500,000 downloads.

Experimental Setup: We run 6 popular Android applications separately on a Galaxy S4 phone. Each application, apart from Youtube, is executed for 20 minutes; for each application, either a WiFi connection or a 3G connection is used to transfer data during the entire time the app is executed. Youtube is the only exception, wherein we use a WiFi connection for the first 20 minutes and a 3G connection for another 20 minutes to emulate a multi-modal app. Subsequently, we capture and compare the results from TIDE and the other tools, as shown in Table III. With respect to the Android Tool and PowerTutor, we show the total amounts of energy reported to be consumed by each app by the tools. Specifically, for each app, (i) the Android Tool reports the percentage of energy consumed by the app with respect to the total energy consumption (by all the apps) on the phone, computed since the last time the battery was fully charged. By knowing how much energy the phone consumes in total, we convert these values into percentages of battery capacity and present them in Table III, (ii) PowerTutor reports the total energy consumption (in kilo Joules) of the app. For TIDE, we show the confidence values with respect to app classification and include the classification labels (as H or M). We also show the ground truth information captured by replaying these 6 apps with an external power meter (the real power consumed). The ground truth information is represented in terms of the energy consumption (in kilo Joules), corresponding battery percentage and the correct classification label for each app.

<table>
<thead>
<tr>
<th>App/Phone Component</th>
<th>Android Tool</th>
<th>PowerTutor</th>
<th>TIDE</th>
<th>Ground truth*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skype</td>
<td>6%</td>
<td>2.2 KJ</td>
<td>90% - H</td>
<td>4.8 KJ - 14% - H</td>
</tr>
<tr>
<td>Youtube (40 mins: WiFi+3G)</td>
<td>3%</td>
<td>5.2 KJ</td>
<td>47% - M</td>
<td>6.2 KJ - 17.3% - M</td>
</tr>
<tr>
<td>Youtube (the last 20 mins: 3G)</td>
<td>N/A</td>
<td>N/A</td>
<td>91% - H</td>
<td>3.9 KJ - 10.9% - H</td>
</tr>
<tr>
<td>Netflix</td>
<td>2%</td>
<td>1.9 KJ</td>
<td>20% - M</td>
<td>3.6 KJ - 8.5% - M</td>
</tr>
<tr>
<td>Pandora (3G)</td>
<td>1%</td>
<td>1.8 KJ</td>
<td>30% - M</td>
<td>3.4 KJ - 9.6% - M</td>
</tr>
<tr>
<td>AngryBird</td>
<td>3%</td>
<td>0.9 KJ</td>
<td>50% - M</td>
<td>3.3 KJ - 9.3% - M</td>
</tr>
<tr>
<td>Hill Climb</td>
<td>1%</td>
<td>1.6 KJ</td>
<td>75% - H</td>
<td>3.6 KJ - 10.1% - H</td>
</tr>
<tr>
<td>System</td>
<td>10%</td>
<td>4.9 KJ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MediaServer</td>
<td>5%</td>
<td>1.3 KJ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screen</td>
<td>3%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Ground truth (x KJ - y% - M/H; the app consumes x kilo Joules y% of the battery capacity, and is classified as a Medium or High energy app
Note: Apps use WiFi unless stated otherwise.
The Android System Tool: The tool only reports total energy consumption but does not capture the average consumption of the apps, which is more important in identifying energy hungry apps. An app should be identified as a high energy one only if it consumes a disproportionate amount of energy relative to its runtime, and not because it is continuously used for a long period of time. Further, the tool does not capture work delegation between media apps and the MediaServer process for media retrieval; thus, MediaServer is shown to consume a high amount of energy, but eventually, the media apps should be considered to be the main culprits for the drain. The System process, which takes care of network data transfers at the kernel level for all other apps (and thus, has a high CPU load), is another case for work delegation and identified as a high consuming app.

4) TIDE Versus Other Popular Approaches: More importantly, the tool does not breakdown the energy consumed by the screen to individual apps; in most cases, the screen consumes the highest amount of energy (30% of the battery). Thus, the energy consumption of all the apps shown by the tool is far from the ground truth results. For example, as the energy consumed by the screen and media retrieval is not contributed to the app, the tool shows that Pandora consumes about 1% of the battery. In reality, it consumes about 10% instead.

Finally, the tool does not capture multi-modal apps. Specifically, the tool does not differentiate Youtube when it uses (i) a WiFi and (ii) a 3G connections; thus, it only provides the energy consumption information of Youtube for the entire time the app is executed. Consequently, the tool does not identify Youtube as a high energy app when it uses the 3G connection for the last 20 minutes.

PowerTutor: PowerTutor is able to capture the total and the average energy consumption of apps by recording their runtimes. However, the accuracy of the PowerTutor is highly device dependent, since the tool estimates the energy consumption of an app by multiplying the amount of resource utilization with the corresponding average energy consumption for each of the resources. The average energy consumption information is calibrated for only a limited number of phone models; thus, when used on an unsupported phone, the results from PowerTutor might be significantly different from the ground truth information. For example, the tool reports that Youtube only consumes 3.2 KJ, whereas it actually consumes 6.2 KJ (when measured with the power meter). Further, the tool is not able to deal with work delegation or multi-modal apps (similar to the Android System tool).

TIDE: With TIDE, our main goal is to classify an app as HIGH or MODERATE/LOW, the tool relies on the rate of battery drainage reported by the phone and thus, does not require calibration for each specific phone model, as with PowerTutor. Further, with TIDE, when an app is the main culprit for high energy drainage, the correlation between its occurrences and short intervals is high. Therefore, the app is correctly identify as a high energy one instead of system processes which might be interacting with the app. Finally, TIDE is the only approach that is able to detect Youtube as a high consuming app, when the app uses 3G for downloading data. As shown in Table III, TIDE is able to correctly classifying all the 6 apps.

5) Overheads: We examine TIDE’s overhead along three dimensions: 1) energy consumed due to TIDE’s periodic querying of the OS, 2) the execution time of TIDE’s greedy algorithm, and 3) the storage space consumed by TIDE’s logs.

Energy Overhead: TIDE runs in the background and queries the OS periodically. We earlier showed in Fig. 4 that with a sampling rate of 30 seconds, TIDE consumes about 0.5% of the battery per hour. The power consumed by the App Classifier is negligible (especially if the processing is done when the phone is being charged). Even otherwise, to process a data file with 700 intervals, the execution of the App Classifier consumes roughly 192 Joules ($\approx$ 0.78% of the battery capacity on a Galaxy SII phone).

Overhead With Adaptive Sampling: To quantify the energy costs with adaptive sampling, we perform the following experiment. We use the gathered data from one of our volunteers with a Galaxy Nexus phone; this was the phone on which we previously measured the energy consumed due to the monitoring process (see Section IV) with different sampling intervals. In this dataset, for each day, we pick the period from 9 AM to 5 PM (this is the time when the user uses her phone the most). We consider energy-heavy intervals to be of duration 2 minutes or less; in such periods, we assume that we query the OS every second. For other intervals (considered low energy periods), we only sample once every 30 seconds. We measure the energy consumed with three different sampling schemes: (a) sampling periodically every second, (b) sampling periodically every 30 seconds, and (c) adaptive sampling as above. The mean values of the energy consumed by the three schemes (based on a 5 day user activity) are 3.20%, 0.50%, and 0.76% of the phone’s battery per hour, respectively. It is apparent that while adaptive sampling does increase TIDE’s energy overhead, the increase is not exorbitant and thus, the approach is viable.

One can claim adaptive sampling makes the phone consume more energy when the battery drain is already high, which will possibly affect user experience. To show otherwise, we do an experiment to measure additional overhead caused by adaptive sampling during high usage intervals. Specifically, we measure the energy overhead with adaptive sampling during a video conference using Skype. In those intervals, the phone consumes 1% of the battery on average in 108 seconds without having any sampling. With adaptive sampling enabled, the phone consumes 1% in 101 seconds. In other words, the penalty is $\approx 7\%$. This indicates that adaptive sampling is unlikely to significantly degrade user experience during high activity periods.

Processing Time of the Greedy Algorithm: Fig. 11 shows the execution times of the App Classifier with data collected over different numbers of intervals. We see that, even if the data in the input file spans 700 intervals ($\approx$ a week of data), the processing time is $\leq 7$ minutes. This processing can be done offline when the user is not using the phone (e.g., when it is plugged into a power outlet for charging at night).

Storage Space: Fig. 12 shows the average storage space used to store the input data collected by the Process Monitor, for different sampling rates. We see that, even when the collected input data spans 700 intervals, TIDE uses less than 6.5MB.
periodically (e.g., Facebook and Twitter) and find that these apps are taken into account by TIDE. Since the default sync intervals for such apps are typically set at around 30 minutes, network keepalive activity does not consume high energy (< 2.5% of the battery per day [4]) and TIDE is able to infer this. Further, if a short-lived app synchronizes its state more frequently, it consumes high CPU or network, and is treated as other normal apps by TIDE. In such cases, the app could be classified as a high energy app.

**Consistent Coexistent apps**: TIDE’s identification of energy-hungry apps depends on correlations between an app’s occurrences and periods of high energy/resource usage. Thus, if two or more apps are *always* used simultaneously, TIDE cannot identify which of the two apps is energy hungry. However, over long usage periods (e.g., a week), we observe that this situation rarely occurs. As soon as the user invokes the apps separately, the real culprit will be associated with a higher confidence value and will thus be correctly classified.

VIII. CONCLUSIONS

In this paper, we argue that there is a need for a user-centric tool to identify energy hungry apps on a user’s smartphone. We design and implement such a tool, TIDE. The key challenges addressed in TIDE are (a) it provides a lightweight way to determine active apps based on adaptive sampling and (b) it uses a novel greedy algorithm to filter out the real energy hungry apps from multiple simultaneously running apps on the users phone. It also effectively captures multi-modal energy behaviors. We show via both in house experiments and user-trace driven emulations that TIDE classifies apps as energy hungry (or not) with very high accuracy and low overhead.

REFERENCES


Tuan A. Dao received the B.S. and M.S. degrees in computer science from the University of Science, Ho Chi Minh City, Vietnam, in 2005 and 2009, respectively. He is currently pursuing the Ph.D. degree in computer science with the University of California at Riverside, Riverside, CA, USA. His research interests include mobile and distributed computing.

Indrajeet Singh received the B.Tech. degree from the Indian Institute of Technology (IIT) Guwahati, and the M.S. and Ph.D. degrees from the University of California at Riverside, all in computer science and engineering. He is currently working for Nerd-Wallet in the capacity of a Senior Software Engineer, prior to which he worked for Akamai Technologies. His research interests include understanding and improving privacy/security in emerging networks, mobile systems, systems research, steganography, and energy management on mobile devices.

Harsha V. Madhyastha received the B.Tech. degree from the Indian Institute of Technology (IIT) Madras, and the M.S. and Ph.D. degrees from the University of Washington, all in computer science and engineering. He is an Assistant Professor with EECS Department, University of Michigan. His research interests include distributed systems, networking, and security and privacy. His work has resulted in award papers at the USENIX NSDI, the ACM SIGCOMM IMC, and the IEEE CNS conferences..

Guohong Cao (S’98–A’99–M’03–SM’06–F’11) received the B.S. degree from Xi’an Jiaotong University, Xi’an, China, and the M.S. and Ph.D. degrees in computer science from The Ohio State University, in 1997 and 1999, respectively. Since then, he has been with the Department of Computer Science and Engineering, The Pennsylvania State University, where he is currently a Full Professor. His research interests are wireless networks and mobile computing. He has served on the Editorial Board of the IEEE TRANSACTIONS ON MOBILE COMPUTING and the IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS, and has served on the program committee of many conferences. He was a recipient of the U.S. National Science Foundation CAREER Award in 2001.

Prasant Mohapatra (S’88–M’93–SM’98–F’12) is a Professor with the Department of Computer Science, University of California at Davis, Davis, CA, USA, where he is currently serving as the Interim Vice-Provost and the CIO. He was on the Editorial Board of the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON MOBILE COMPUTING, the IEEE TRANSACTION ON PARALLEL AND DISTRIBUTED SYSTEMS, ACM Wireless Networks, and Ad Hoc Networks. His research interests are in the areas of wireless networks, sensor networks, Internet protocols, and QoS. He is a Fellow of AAAS.