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

Tuan Dao*, Indrajeet Singh*, Harsha V. Madhyastha⋆,
Srikanth V. Krishnamurthy*, Guohong Cao†, Prasant Mohapatra‡
*UC Riverside ⋆University of Michigan †The Penn State University ‡UC Davis

***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 (a) identify apps that are energy hungry, and (b) understand why an app is consuming energy, on her phone. Towards 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 testbed 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 $< 6\%$.**

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 apps for smartphones, this trend has left users distressed about how long their phone's battery lasts even after a full recharge. A report in 2012 [1] 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." While there exist tools that try to quantify the energy consumption of smartphone apps, they primarily target software developers who want to check for power inefficiencies in their products before release. These tools either require the use of external equipment (e.g., a power meter), or need modifications to the smartphone's operating system (OS). In addition, these tools need to be run continuously to track an app's operations, and hence consume significant energy themselves.

It is desirable to have an efficient tool, capable of reporting which apps on a specific user's phone dominate battery consumption. This tool should not simply focus on detecting apps with energy bugs [2] or ignore *user-specific factors* that influence battery drain (e.g., as in [3]); for each user, it should identify apps that consume a disproportionate amount of energy *on that user's phone*. When run on a specific user's phone, one could envision this tool as roughly categorizing apps as energy-hungry or energy-thrifty based on how an app is used by the user and the environment in which it is used. Once energy hungry apps are identified, a user can reduce or cease to use such apps when needed.

Challenges: Unfortunately, developing such a user-centric tool is hard. Since normal users will be reluctant to install modifications to the smartphone OS (this voids the phone's warranty), the tool must only use information exported by the OS to the application layer. This information is however insufficient for 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 drain is affected by many factors, including the device features, 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 videos are 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 conduct an extensive measurement study on our 22 Android phone testbed, which shows how differing network conditions, device features, and usage patterns affect the energy consumed by apps. Our study also highlights the challenges in building a user-centric tool as discussed above viz., the need to (a) efficiently sample the information exported by the OS, and (b) filter noisy data due to the co-existence of multiple active apps on a smartphone. Next, we design, implement, and evaluate TIDE (Tool for Identifying Dominant Energy apps), 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. Our evaluation of TIDE, based on a detailed emulation of traces of usage patterns from 17 volunteer users, shows that it correctly estimates the level of energy consumption for 225 out of 238 apps. Further, TIDE delivers this level of accuracy while imposing only 0.5% of overhead on the average consumption of the phone's battery per hour.

II. Related work

Android provides a battery manager tool [4], Fuel Gauge, which estimates the percentage of battery consumed by each app. However, in Section III, we show that the tool does not

account for several user-specific factors that can significantly influence the energy consumption of an app (e.g., the network link quality when data is transferred). Other prior efforts on estimating application-specific energy/power consumption can be broadly classified into three major classes.

Tools: Current tools that try to characterize the power consumed by apps use offline tests and/or fail to account for one or more factors affecting the battery drain due to an app. PowerTutor [5] estimates an app's power consumption due to its interactions with hardware components (e.g., LCD, 3G interface) 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 requires per-app resource consumption information, which is not readily available in newer versions of Android, and c) it needs offline calibration for each device type. Carat [3] 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 analyses locally on any particular device (there is no need for either offline calibration or server-side aggregation). Falaki et al. [6] also suggest that diversity across users in terms of their app interactions can affect battery drainage rates. However, they do not develop a tool such as TIDE for user-specific estimation of app energy consumptions.

Identifying energy bugs: There exist tools for detecting energy bugs in apps (e.g., [2]). However, they require an external power meter for energy measurements and/or the modification of the underlying OS. eDoctor [7] identifies abnormal drain issues on phones by comparing app behaviors with well known good versions; however, user-centric factors are not accounted for.

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. [8] build a model to estimate the power consumption in different hardware components, based on a set of apps. However, the model does not work for new apps outside this set. WattsOn [9] is an 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 this 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 [10] argues that such models must be generated based on measurements using individual smartphones, rather than offline in a lab. Carroll et al. [11] perform offline measurements of the power consumed by Android components while running various benchmarks. The energy consumed on different wireless networks is studied in [12]. None of these efforts develop a user-centric tool for identifying energy hungry apps.

III. SHOWCASING USER-CENTRIC APP BEHAVIORS

In this section, we present a measurement study to demonstrate that user behaviors, network conditions, and even phone features impact the energy consumption of apps.

Impact of network type and signal strength: 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 drain with each phone in our experiment. We see that transferring data over 3G costs more than a transfer via WiFi, by a factor of 1.5 to 2.5. With the same network type, in poor signal conditions, as one might expect, there are many packet retransmissions, and thus, the energy consumption is much higher; for example, with a high volume of data, in 1 hour, the phone with the poor 3G signal consumes more than 8% of the battery, while the phone with the good 3G signal consumes only around 5%. In summary, 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 with different phone models and different network providers and still observed qualitatively similar results. These results are deferred to [13] due to space limitations.

Impact of user behaviors and phone features: Different users can potentially use the same application quite differently and this affects that app's energy consumption.

An example with YouTube: To demonstrate the impact of user-specific workloads on energy consumption, we experiment with YouTube. We play different videos on a smartphone Samsung Galaxy SII (referred as *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 $\frac{3}{4}^{th}$ 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 HTC MyTouch 4G smartphone (referred as *Dev2*).

Results: Our results, shown in Fig. 2, first re-affirm the differences in the consumed energy when the user is on different networks. Second, we see significant differences in the energy consumed when playing different videos (on the same device); between the two videos, we see a difference of as much as 20% in 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 versus low quality), and the display size, the YouTube app's energy consumption may vary. These choices depend on user preferences, i.e., the user's behavior influences the energy consumed with YouTube.

We also observe differences in the energy consumptions (as much as 49 %) across devices when playing the same video file (from local memory). This is primarily due to the differences in

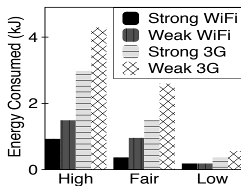

Fig. 1: Network impact on energy

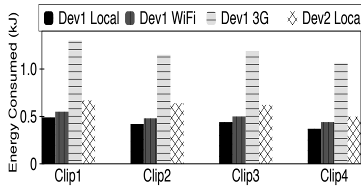

Fig. 2: YouTube's energy consumption when playing different videos

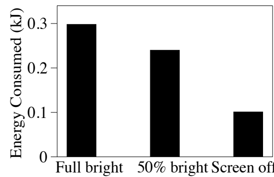

Fig. 3: MusicFolderPlayer's energy consumption

```
1: <device name="Android">
2:     <item name="screen.full">211.6</item>
3:     <item name="WiFi.on">1.38</item>
4:     <item name="WiFi.active">62.09</item>
5:     <item name="WiFi.scan">52.1</item>
6:     <item name="radio.active">185.6</item>
(... file content is shortened for clarity...)
```

TABLE I: Android power_profile.xml

the hardware on the two phones and shows that device features affect battery consumption. *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: Next, we show that a few other apps also exhibit such *multi-modal* energy consumption patterns.

MusicFolderPlayer: 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 player. Specifically, we have two users play the Angry Birds game for 10 minutes each. One user, 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 % ($\approx$ 4.8 % in terms of the battery percentage consumed) *per hour* of play.

The Android system tool does not account for user-centric factors: Fuel Gauge, the Android system tool, attributes energy 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 from using that component. The sum of these values across all components is the energy attributed to 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, a partial view of which is shown in Table I. We see the average energy used by the WiFi and cellular interfaces in one time unit on lines 4 and 6. It is evident that network quality is not accounted for by the tool.

Further, from the source code of the tool [4], we see that when computing the energy consumed from network activities, the tool only considers the amount of data transfers due to the app. It does not differentiate between the app's use of WiFi and cellular networks. If the (total) data transferred by all apps over 3G and WiFi are $3GData$ and $wifiData$ respectively, the Android OS computes the average energy consumed per byte as $(3GEnergyPerByte * 3GData + wifiEnergyPerByte * wifiData)/(3GData + wifiData)$, where $3GEnergyPerByte$ and $wifiEnergyPerByte$ are obtained from the power model (Table I). For each app, the OS computes the energy consumed from network activities by multiplying the average energy per byte 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; in fact, as shown in the earlier experiments in this section, the energy consumed due to network activities depend on both the network type and the link qualities experienced. We have validated via experiments that Fuel Guage in many cases, does not provide accurate battery drain values (by comparing the results with measurements using a power meter). The differences in the results are large and could lead to a mis-classification of apps. We do not provide more details here due to space limitations.

Solutions such as Carat [3] cannot be easily extended to account for user-centric behaviors: By its very nature, crowdsourcing (the basis for Carat [3]) ignores user-specific characteristics of apps. We downloaded and tested Carat on our own Android phones for a week. Carat classified two of our apps—Google Maps and Skype—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 [14].

IV. CHALLENGES IN DESIGNING TIDE

Next, we describe the challenges in building a user-centric battery management tool on Android. Based on some preliminary studies, we believe that iOS poses similar challenges.

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, they either do not record these details for energy efficiency or hide this information due to security concerns.

Lack of precise energy usage information: In prior work, researchers have either used devices such as the Monsoon meter [15], or plugged special sense resistors into hardware components on the phone to measure the energy consumed [11] by a single app in isolation, or towards building 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 when the phone's battery level changes, i.e., drops by 1%. Hereafter, we refer to each period in which the battery drains by 1% as simply an *interval*. In Section V-B, we elaborate on how this information is captured on the Android platform.

Lack of app-specific resource usage information: To sidestep the above limitation one could think of the following approach. For each type of phone, one can construct an accurate power model for every hardware component (e.g., LCD

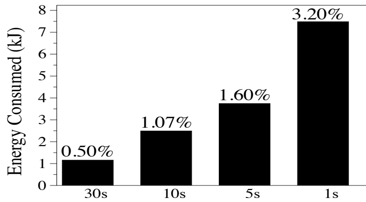

Fig. 4: TIDE's energy consumption vs. sampling rates

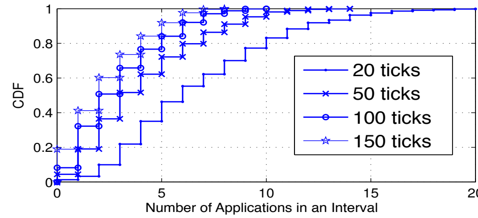

Fig. 5: Number of active apps

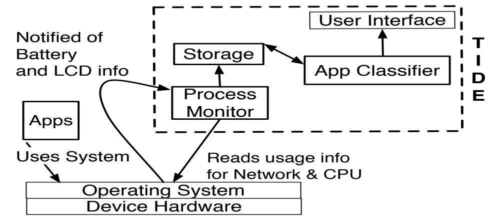

Fig. 6: TIDE architecture

display, CPU) in every environment (e.g., power consumed by LCD as a function of brightness or, 3G based on signal strength). Discounting the fact that building such models will be cumbersome, TIDE can then estimate the energy consumed by each app by 1) monitoring the environment of usage 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 is 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 across apps (and not for each individual app). For example, Android permits an app to be notified when the screen is turned on or off. While this would enable TIDE to know the time for which the phone's display was on, it cannot determine how long each individual app uses the screen when multiple apps are simultaneously run by the user.

We point out that while the OS tracks and exports 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).

The only resource whose usage TIDE can track on a per-app basis is the CPU. On Android, the CPU usage of an app with process ID pid, is provided in the file `/proc/[pid]/stat`. The CPU usage time is measured in 'system ticks'. The number of ticks per second is usually set to 100 [16].

Overhead of querying information: To cope with the availability of only aggregate resource usage information TIDE can query the OS frequently (e.g., every second). It can then attribute all the resource consumption in the last second to the app actively used (in the foreground) in that period. On the Android OS, TIDE can query 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. Upon querying every second, TIDE itself consumed 3.2% of the battery in an hour. Consuming over 3% of the phone's battery every hour makes 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.

Extracting app-specific energy consumption: 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).

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 this is based on an 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 an 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 drain in that interval. Hence, we need a threshold to filter out apps that were largely dormant. However, determining a good threshold for CPU ticks is hard; it will depend on the smartphone architecture and on an app's implementation. In Fig. 5, we plot the CDF of the number of simultaneous apps (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 (e.g., with a threshold of 20 ticks, 60% of the intervals have $>$ 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 (the number of simultaneous active apps in an interval) is user-specific.

Work delegation between apps: Another major hurdle in attributing energy consumption to apps is that the functions of one app are sometimes delegated to another app on Android devices. 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 appropriate app. For example, when a user is viewing a video with YouTube, the video streaming is delegated 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.

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, 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. 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. Note that we focus on the energy consumption due to the CPU, the network interfaces, and the display. However, TIDE is extensible to account for energy consumed while using other resources. For example, TIDE can identify an app's energy consumption due to the use of 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: The Process Monitor runs in the background and keeps track of *intervals* (durations between instances when the battery level drops by 1 %). After each interval, it queries the OS for the resource usage information in that interval. Specifically, it obtains (i) the time for which the screen was on and, (ii) the aggregate network usage (in bytes), during that interval. Within each interval, the Process Monitor also queries the OS once every τ seconds, for a list of the running apps and the CPU usage of each app in those preceding τ seconds. This information is stored in the phone's SD card and is later processed by the App Classifier.

Adaptive sampling: There is an inherent trade-off in choosing a value for τ. On one hand, the larger the value of τ, 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., τ = 1 second), but this increases the energy overhead imposed by TIDE. Note that the number of co-existing apps with $\tau = 1$ sec $\ll$ when $\tau = 30$ secs, but apps may still co-exist. To address this trade-off in TIDE, we use

Algorithm 1 TIDE's algorithm for app classification

//Phase 1
for all app x **do**
  s := Fraction of intervals containing only x that are short
  l := Fraction of intervals with x that are long
  if $s \geq f_H$ **then**
    Mark x as $HIGH$
  else if $l \geq f_L$ **then**
    Mark x as LOW
  end if
end for
//Phase 2
$\forall$ unclassified app x, calculate $conf(x)$
while $\exists$ unclassified app x with $conf(x) \geq \gamma$ **do**
  Find app x that has the highest confidence
  Mark x as $HIGH$
  Remove all short intervals that contain app x
  Recalculate confidence values of unclassified apps
end while
//Phase 3
Multi-mode candidates = apps classified in phase 1 $\cup$ all unclassified apps
for all multi-mode candidate app x **do**
  Calculate $conf(x, r)$ for app x and resource r
end for
while $\exists$ tuple (x, r) with $conf(x, r) \geq \gamma$ **do**
  Find tuple (x, r) that has the highest confidence
  Mark app x as $HIGH$ when it intensively uses resource r
  Remove short intervals with app x and high utilization of r
  Recalculate confidence values of remaining tuples
end while
Mark all unclassified apps as $MODERATE$

an adaptive sampling approach. Specifically, Process Monitor queries the OS more often when the battery drainage is heavy (i.e., in 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 when the rate of energy consumption is high. After a high-drainage interval is seen, the Process Monitor switches to fine-grained sampling, and τ 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 (low drainage) intervals, the Process Monitor returns to coarse-grained sampling; in our implementation (described later), we find that $k = 1$ works well and we set τ to 30 seconds for coarse-grained sampling. We evaluate the overhead and efficiency with adaptive sampling in Section VI.

2) App Classifier: The output of the Process Monitor contains the set of co-existing apps detected with each query, as well as the resource usage (e.g., screen,network) during an interval. The App Classifier takes this as an input and tries to identify the high-energy apps from this noisy data. It performs this classification in 3 phases (summarized in Algorithm 1).

Phase 1: Using interval lengths to categorize apps: First, since long intervals correspond to low battery drain, *all* of the active apps in that interval must have consumed low energy during the interval. If *any* of the active apps in a specific interval consumed high energy, then that interval is short. Second, if a single app was active in a short interval, then that app was definitely the cause for the high battery drain in that interval. Based on these, our first phase of app classification works as follows. For any app X, we consider all the intervals in which this app is 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_H, then we mark X as an energy hungry app. Similarly, among the intervals in which an app Y occurs, if

the fraction of long intervals is greater than a second threshold f_L, then we consider Y to be a battery-thrifty app. However, the above procedure by itself cannot classify all apps. This is because, as discussed in Section IV, many intervals may include multiple active apps; 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: Handling co-existing apps: To account for multiple active apps in short intervals, we use a greedy algorithm. In short, the larger the fraction of short intervals among the intervals in which an app is active, we have greater confidence in declaring the app as energy hungry. The algorithm identifies energy hungry apps in the decreasing order of associated confidence. Once an app is marked as energy hungry, we *greedily* attribute all the energy consumed in all the short intervals in which the app is active, to this app.

In more detail, let the confidence value for an app X being energy hungry, *conf(X)*, be the probability that an interval which contains X is short. An app X is deemed energy hungry if *conf(X)* is $>$ than a threshold (say γ). Once X is marked energy hungry, the classifier discards all high battery drainage (short) intervals that contain X from future consideration; this essentially attributes the high battery drain in these intervals to X. The classifier then repeats the process to find the app with the next highest confidence value ($\geq \gamma$), based on the intervals yet to be discarded. The process is repeated until no apps with a confidence value $\geq \gamma$ remain.

With the above algorithm, there may be cases where a high energy app Y is filtered out because it appears with another high energy app X in a some of the short intervals. To handle such cases, we could compare the lengths of the intervals when X is present by itself with the lengths of those intervals when X co-occurs with Y. If the length of intervals where Y and X simultaneously appear are much shorter than when X appears alone, this is potentially evidence of Y being a high-energy app. We however 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 end up with low confidence values, since intervals containing an app X combine data from all of X's modes. To handle such cases, we define the confidence value for a tuple of app 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 drain (short) interval. Using *conf(X,R)*, TIDE detects apps that are energy hungry only in execution modes where a specific resource (e.g., network, screen) is intensively used. This information allows 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).

B. Implementation details

Next, we describe our Java-based implementation of TIDE for Android phones[1].

Process Monitor: TIDE captures a phone's battery usage by monitoring system events broadcasted by the Android OS. It registers to receive the ACTION_BATTERY_CHANGED event, using which it track periods where 1% of the battery is depleted. The Android OS also notifies user-space apps (such as TIDE) when the display is turned on and off; using this TIDE learns the display usage in each interval. To capture other resource usage information in an interval, TIDE queries appropriate system files. For example, `/sys/class/net/wlan0/statistics/tx_bytes` and `/sys/class/net/wlan0/statistics/rx_bytes`, which are updated by the underlying Linux kernel, provide *aggregate* information about the number of bytes sent or received by all the apps via the WiFi interface. Due to space constraints, we defer specifics of the other system files that TIDE checks to [13].

App Classifier: The App Classifier first filters out inactive apps or apps that are not heavy contributors to the energy drain in each interval. It primarily considers an app to be active if it consumes more CPU ticks than a preset threshold. In some outlier cases, an app may use the LCD but not the CPU; to account for such cases, TIDE checks whether the app is a foreground app in high energy intervals. If so, the app's energy consumption due to the display is directly computed and thus, TIDE determines if it is energy hungry in this mode.

Choosing a CPU threshold: We classify an app as active if it uses more than a threshold number of CPU ticks; even if the app uses other resources (e.g. to render graphics on the screen, to stream data, etc.), it requires a significant number of CPU cycles. To establish the right threshold, we installed many popular apps from the Android market on a Galaxy SII phone and monitored their CPU usage with a real user's usage pattern (described in section VI-A). With this, we determined when the apps were actually being executed and when they were idle in memory. We considered two types of apps: one set which have high CPU usage (e.g., Skype, Angry Birds), and another set with low CPU usage (e.g., MusicFolderPlayer, Advanced Task Manager). We found that a threshold of 150 CPU ticks when $\tau = 30$, works well to ensure that we do not filter active periods of low CPU usage apps but do filter dormant periods of high CPU usage apps. For $\tau = 1$, a threshold of 5 CPU ticks accurately assigns the resource usage to an active app. We repeated the experiment with three other users' traces and obtained almost identical results. This leads us to believe that these thresholds on the Galaxy SII phone are appropriate for use in TIDE to identify active apps. When TIDE is used on other phone models, we apply a linear scaling between the CPU frequency of the new model and the reference model (Galaxy SII) to determine the CPU ticks threshold for the new model. We find that this approximation works well in practice. We also observe that minor variations in the CPU ticks threshold do not significantly affect TIDE's accuracy.

Detecting app LCD usage: To determine high energy apps that keep the screen on without using the CPU, we use adaptive sampling in high energy intervals to find the foreground app in each second. The LCD usage is attributed to that app. In longer low energy intervals, TIDE can only capture the foreground app once every τ=30 secs; thus, we miss the apps that use the display at times in between. However, this is not of consequence since, whether or not the app uses the display it consumes low energy in such intervals.

Once the active apps are determined, the App Classifier runs the classification algorithm in Section V-A2. 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 (recall f_L and f_H in Section V-A2), and 2) the *conf(X)* or *conf(X,R)* values associated with any app X. We experiment with different

[1]We are working towards releasing TIDE on the Google Play store; a rudimentary version can be found at http://bit.ly/1lEWgzD.

values for these thresholds with different user workloads and on different types of phones. To keep the false positive rate low, we find that $f_L = f_H = \frac{1}{4}$ 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 allows the user to choose the thresholds that define HIGH and LOW drainage intervals based on her preferences. For evaluation purposes, we define intervals in which 1% of the battery is drained in < 2 minutes as HIGH and intervals in which 1% of battery is drained in > 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 (e.g., Skype takes 1.8 minutes and MusicFolderPlayer takes around 6.5 to 9 minutes).

When is resource usage high? To account for multi-modal apps, we need to construct tuples of the form $\{$X, R$\}$ to represent the presence of an app X in a high battery drain interval in which resource R is also heavily utilized. Thus, we need to determine *"when should the usage of resource R be considered high?"* To answer this, we perform measurements using known resource hungry applications with respect to each resource. For network usage, we measure the traffic from YouTube while watching 20 random video clips of HD quality, and from Skype during a video conference. We choose these apps as they are known to manifest high network usage. We measure the traffic while the apps are executed on 4 different devices and in different network conditions. In all our experiments, the apps generate ≥ 2.5 MB of traffic per minute; 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, 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 evaluate TIDE based on experiments conducted on a testbed of Android phones, 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 from their daily phone usage for a week. Our study is IRB approved by UCR. 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. To ensure consistency, we matched the model of the phone handed out to a user to the user's own phone. The collected user traces are used to generate realistic workloads on our Android testbed for establishing the ground truth (discussed in Section VI-B). We also run TIDE on these phones to get its output assessments.

1) Capturing user interactions: On each phone handed out, we installed a background process that captures all of the user's interactions with her phone. Capturing these interactions such that accurate replay is possible is a significant challenge. For example, a user's interaction with a web page is hard to replay since the page's content varies over time. 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. [17] to capture user input events with low overhead by polling the phone's system files.

Apart from storing user input events, we also need to associate these events to apps. Unfortunately, the phone's system files do not provide this information. Thus, for each interaction, we also capture the foreground app 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 for the purposes of a later replay on our testbed. By emulating different network conditions during replay, we build the ground truth with regards to the "user-centric" energy consumed by each 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). We run `tcpdump` on the phone to capture all packets going through all network interfaces. Periodically, we run a modified version of `netstat` (provided by the Busybox tool set [18]) 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 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 thus they cannot be used in TIDE.

B. Building the ground truth

To evaluate the effectiveness of TIDE, for every app used by a specific user we need to know whether or not the app is indeed energy hungry "for that user." Generating this ground

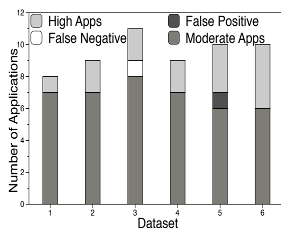

Fig. 7: TIDE's accuracy with user interaction based ground truth

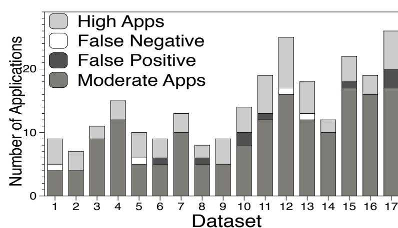

Fig. 8: TIDE's accuracy with resource usage based ground truth

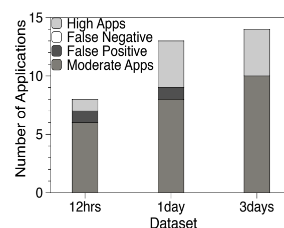

Fig. 9: TIDE's accuracy with different amounts of data

truth is non-trivial in itself. In reality, apps do not run in isolation. Further, user-centric factors such as the 3G signal strength at different times, are not known. Thus, to generate the ground truth, for each app used by each user, 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.

We assign the labels HIGH, MODERATE/LOW to each app based 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, we use what we believe are reasonable here. Based on our discussions 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; else, we mark the app as MODERATE/LOW. From conversations with our volunteer users, we believe that users are likely to treat MODERATE and LOW apps the same (they do not believe that it is vital to distinguish between them). Thus, we combine these into one category and label them as MODERATE.

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. 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. We also record the number of CPU ticks associated with these activities. When replaying the 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. Since we do not know the exact content on the screen at specific times, 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., [5], [10]).

C. Evaluating TIDE

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, with that based on resource usage information. Note that, on each of our volunteers' phones, TIDE was concurrently running while we were capturing logs later used for trace replay (towards determining the ground truth).

Accuracy as compared to ground truth based on user interactions: Both volunteers, for whom we could capture input events, 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. 7 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. 7 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.

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 apps (with higher confidence values), app X gets filtered out. TIDE then labels X as MODERATE. As users use apps 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.

Accuracy with respect to ground truth based on resource usage: Next, we use the datasets from our 17 volunteer users as the basis for ground truth. Fig. 8 shows TIDE's accuracy in those 17 datasets, with the results amortized over different network conditions. The results are presented in the same form as in the previous case; each bar represents results from a different user's data. For 7 users, TIDE was able to classify all the apps correctly. In almost every other case, we only saw either one false positive or one false negative. Users 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 these cases. 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.

		2-minute threshold		3-minute threshold	
Application	Condition	GT	Result	GT	Result
Skype	Strong WiFi	**H**	**H**	**H**	**H**
	Weak WiFi	**H**	**H**	**H**	**H**
	Strong 3G/4G	**H**	**H**	**H**	**H**
	Weak 3G/4G	**H**	**H**	**H**	**H**
Web browser	Strong WiFi	M	M	M	M
	Weak WiFi	M	M	**H**	**H**
	Strong 3G/4G	**H**	**H**	**H**	**H**
	Weak 3G/4G	**H**	**H**	**H**	**H**
Pandora	Strong WiFi	M	M	M	M
	Weak WiFi	M	M	M	M
	Strong 3G/4G	M	M	M	M
	Weak 3G/4G	M	M	**H**	**H**
YouTube	Strong WiFi	M	M	M	M
	Weak WiFi	M	M	M	M
	Strong 3G/4G	**H**	**H**	**H**	**H**
	Weak 3G/4G	**H**	**H**	**H**	**H**
Angry Birds	Strong WiFi	M	M	**H**	**H**
	Weak WiFi	M	M	**H**	**H**
	Strong 3G/4G	**H**	**H**	**H**	**H**
	Weak 3G/4G	**H**	**H**	**H**	**H**
Note: GT - Ground Truth; **H** - HIGH ; M - MODERATE					

TABLE II: Energy usage varies with network conditions

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 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 or, (ii) in less than 3 minutes. The stable results confirm 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.

Capturing multi-modal apps: We next evaluate TIDE's ability to classify multi-modal apps. 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 have other auxiliary apps that run simultaneously with Pandora and YouTube. With Pandora, we run an app that executes in 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 uniform 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 without considering its different usage patterns, is quite low (20% out of 16 intervals). Thus, TIDE classifies Pandora as a MODERATE app. However, when TIDE considers Pandora only in intervals where the LCD is intensively used, the confidence value of the tuple (Pandora, LCD) is high (80%) and TIDE classifies Pandora as energy hungry. 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.

Accuracy versus dataset size: The longer the monitoring period with TIDE, the better is its 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 four high energy apps and TIDE was able to detect all four 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.

Overheads: We examine TIDE's overhead along three dimensions: 1) energy consumed due to TIDE's 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: 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, app classification using data over 700 intervals consumes $\approx$ 192 Joules ($\approx$ 0.78% of the

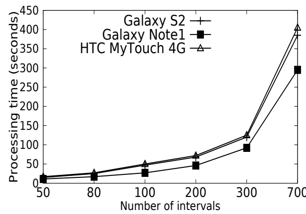

Fig. 10: App Classifier's processing times

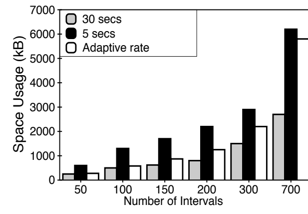

Fig. 11: Space used to store Process Monitor's logs

battery capacity on a Galaxy SII phone).

Overhead with adaptive sampling: Next, we quantify the energy costs with adaptive sampling. We use the data from one of our volunteers with a Galaxy Nexus phone. For each day, we pick the period from 9 AM to 5 PM (this is when the user uses her phone the most). We consider energy-heavy intervals to be 2 minutes or less; in such periods, we query the OS every second. For other intervals (considered low energy periods), we 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, and this will affect user experience. To show otherwise, we measure the energy overhead with adaptive sampling during a video conference using energy hungry Skype. The phone consumes 1% of the battery on average in 108 seconds without any sampling. With adaptive sampling enabled, the phone consumes 1% in 101 seconds. Thus, the penalty is $< 7\%$; this indicates that adaptive sampling is unlikely to significantly degrade user experience during high activity periods.

Processing time of the greedy algorithm: Fig. 10 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 ≤ 7 minutes. This processing can be done offline when the user is charging the phone at night.

Storage space: Fig. 11 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 input data spans 700 intervals, TIDE uses < 6.5MB. Old data is purged as new data is accumulated, and hence, TIDE's storage overhead does not continuously grow over time.

VII. Discussion

Determining usage thresholds: As discussed earlier, TIDE uses a few thresholds for classification purposes; these thresholds are determined by measurements from the usage traces of several users. We observed that minor variations in these thresholds do not affect TIDE's accuracy in classifying apps. While an alternative approach based on machine learning could be used to learn the appropriate values for these thresholds, the training process required by such an approach can potentially consume high energy. This requires more careful consideration in the future. In contrast, our simple approach offers high classification accuracy while being energy thrifty.

Coping with evolution of Android: As the information exported by Android evolves in the future, TIDE may have either less or more information that it can use. On one hand, Android may reduce the information exported to the user-level (e.g., due to security concerns). However, many inputs that TIDE relies on (e.g., when the battery level drops by 1% and the aggregate network consumption) are useful to users, and hence, Android is likely to continue to export this information. Moreover, TIDE's techniques can be folded into the OS itself, e.g., into Android's Fuel Gauge tool. On the other hand, if the information available to TIDE increases, TIDE's classification can potentially be made more efficient.

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 user's 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

[1] Battery life complaints causing operator headaches. http://bit.ly/PI5Fnj.

[2] 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*, 2012.

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

[4] Android battery tool source code. http://bit.ly/1i5cRKB.

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

[6] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *MobiSys*, 2010.

[7] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. Saul, and G. Voelker. eDoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *NSDI*, 2013.

[8] A. Shye, B. Scholbrock, and G. Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *MICRO*, 2009.

[9] R. Mittal, A Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *Mobicom*, 2012.

[10] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *MobiSys*, 2011.

[11] A. Carroll and G Heiser. An analysis of power consumption in a smartphone. In *USENIX ATC*, 2010.

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

[13] Tide system - technical report. https://www.dropbox.com/s/3d1ilxwy4nhivmb/main.pdf.

[14] Carat - Android version. http://bit.ly/1h5EYGS.

[15] Monsoon power monitor. http://bit.ly/p6qNfY.

[16] Android source code. http://bit.ly/1lOR2Ci.

[17] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: timing- and touch-sensitive record and replay for Android. In *ICSE*, 2013.

[18] Busybox tool set. http://www.busybox.net/.