Enhancing Mobile Apps To Use Sensor Hubs Without Programmer Effort

Haichen Shen, Aruna Balasubramanian[†], Anthony LaMarca*, David Wetherall
University of Washington, [†]Stony Brook University, *Intel

ABSTRACT
Always-on continuous sensing apps drain the battery quickly because they prevent the main processor from sleeping. Instead, sensor hub hardware, available in many smartphones today, can run continuous sensing at lower power while keeping the main processor idle. However, developers have to divide functionality between the main processor and the sensor hub. We implement MobileHub, a system that automatically rewrites applications to leverage the sensor hub *without additional programming effort*. MobileHub uses a combination of dynamic taint tracking and machine learning to learn when it is safe to leverage the sensor hub without affecting application semantics. We implement MobileHub in Android and prototype a sensor hub on a 8-bit AVR micro-controller. We experiment with 20 applications from Google Play. Our evaluation shows that MobileHub significantly reduces power consumption for continuous sensing apps.

Author Keywords
Mobile sensing; energy-efficiency; sensor hub; dynamic taint tracking; machine learning

ACM Classification Keywords
C.5.0 Computer System Implementation: General

INTRODUCTION
Today's smartphones provide a rich sensing platform that developers leverage to enable tens of thousands of mobile applications. Many of these applications require continuous sensing and monitoring for tasks ranging from simple step counting to more complex fall detection, sleep apnea diagnoses, dangerous driver monitoring and others.

Unfortunately, continuous sensing applications are power hungry. Interestingly, it is neither the sensors nor the computation that make these applications battery drainers. Instead, the main processor needs to be powered on frequently to collect sensor samples, in-turn increasing the power consumption [33, 28, 34].

Hardware manufacturers recognize that supporting *low-power* continuous sensing is crucial. To this end, companies such as Texas Instruments (TI) [31], Intel [29], and Apple [2], are embedding a low power micro-controller called a *sensor hub* in their smartphones. The sensor hub continuously collects sensor data keeping the higher power main processor idle.

In practice, however, sensor hubs fail to deliver on their power efficiency promise. The problem is in the difficulty in programming them. For example, to leverage the sensor hub for a fall detection app, the developer not only needs to write the main application, but also needs to program the sensor hub to sense and notify the main application when a fall is detected. Two approaches have been used to make it easier for developers to program the sensor hub: APIs and hardware SDK.

In the APIs approach [5, 1], a set of important sensor inference functions are exported via high level APIs to the app developers. The problem is that the APIs only support a set of pre-defined events or activities such as step counting. Today, a fall detection application cannot use any of the existing APIs to leverage the sensor hub. It is possible that sensor hub APIs will stabilize, but this is unlikely to happen for many years. Consider how much location APIs have evolved since the Java Location API (JSR 179) was introduced in 2003. Sensor hubs themselves have regularly been part of phones since 2011, but it is only in 2014 that a small set of sensor APIs are aligning around common functionality. In the meanwhile, ambitious sensing applications such as BeWell [3] cannot leverage the sensor hub for power efficiency.

In the hardware SDK approach, the developer is provided with specialized tools to leverage the sensor hub. For example, TI provides a proprietary TivaWare Sensor Library [10] to allow developers access to functionality not exposed by software APIs. Similarly, Intel provides a proprietary Context Sensing SDK [4]. However, applications developed in one vendor's environment will not work in another's and vice versa. In other words, not only does the developer need to learn a new software/SDK, but she needs to do it for potentially each hardware platform.

We explore a third approach. We present MobileHub, a system that rewrites applications to leverage the sensor hub without any programming effort. Our goal is to keep the main processor in a sleep state while the sensor hub reads and buffers sensor readings. By then delivering the buffered readings, we allow the host to process an entire batch of sensor data for little more than the cost of a single reading.

The core of our system is about learning when it is safe to buffer sensor readings without altering the application semantics. Our key insight is that typical mobile sensing apps have an "inner loop" that reads sensor data at a high-rate, until a specific event occurs. For example, a fall detection application reads accelerometer data at a high rate, until a user falls down. At that point, an "outer loop" takes over, resulting in an

UbiComp '15, September 7-11, 2015, Osaka, Japan.

http://dx.doi.org/10.1145/2750858.2804260

application notification, such as raising an alarm that the user has fallen down. MobileHub learns, to the first approximation, the statistical characteristics of sensor values that cause the application notification.

To determine when a sensor reading triggers a notification, we use dynamic taint tracking [20, 13]. This technique tracks the flow of sensor data during runtime to determine precisely when the sensor data leads to a particular application behavior. MobileHub then learns the statistical properties of the sensor data that does not cause any application notification. Using this learning model, MobileHub buffers sensor values at the sensor hub as long as the classifier predicts that the sensor data will not trigger any notification; else, MobileHub wakes up the application and passes the buffered sensor data to it. Eventually, all sensor values reach the application; we are simply adjusting the delivery schedule to optimize power.

We implement MobileHub on Android and prototype a sensor hub using an 8-bit Atmel micro-controller. We evaluate the performance of MobileHub using 20 applications that we download from Google Play. The applications span 4 different sensors—accelerometer, magnetometer, gyroscope, and orientation, and span diverse tasks—step counting, fall detection, sleep monitoring, driver monitoring, earthquake detection, motion detection, and metal detection. For each application, MobileHub analyzes the application using taint tracking, learns a classifier, and rewrites the binary. MobileHub also implements the classifier in the sensor hub firmware. We conduct one end-to-end lab experiment as a proof of concept, and conduct emulation experiments driven by user-traces for a more extensive evaluation.

In our end-to-end experiment, MobileHub improved power consumption of a Pedometer app by 72%. In the emulation experiment, MobileHub improved the power consumption for 17 of the 20 applications. Three of the applications could not benefit from MobileHub, or for that matter cannot benefit from sensor hubs at all, without changing their semantics. For the remaining applications, except the metal detection apps, MobileHub reduced power consumption by an average of 70%. MobileHub reduced power consumption for the metal detection apps by an average of 33%.

MobileHub was able to get the power benefits with practically no change to the application semantics. By semantics, we mean that the application made the same sounds, notifications, and vibrations as the original and on a timeline that was *nearly* identical. MobileHub delayed the timing of the application notification a small number of times: 1.5% of the time across all applications on an average. Even when MobileHub delayed application notification, the maximum delay induced by MobileHub was less than 1 second for 80% of the applications.

BACKGROUND

Sensor Hubs

Hardware manufacturers are increasingly embedding a sensor hub in their phones [31, 29, 2] to reduce power consumption of continuous sensing tasks. Figure 1 shows an example sensor hub architecture. Sensor hubs can perform continuous sensing while drawing a fraction of the power compared to

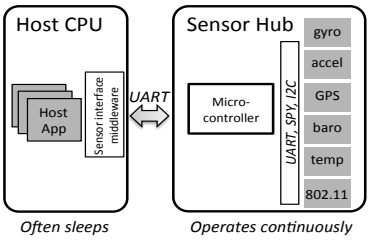

Figure 1. A typical sensor hub architecture.

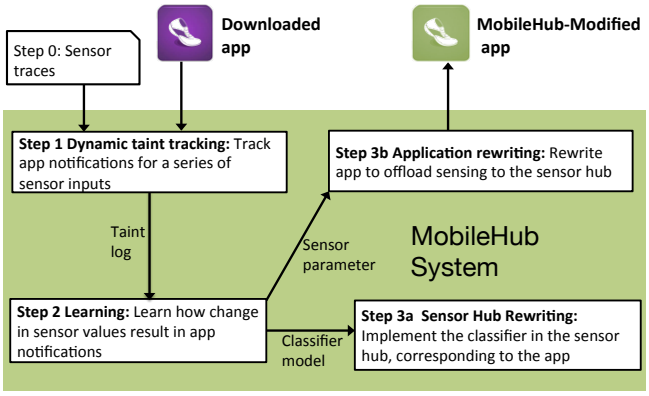

Figure 2. The *MobileHub* architecture

the main processor. For example, the AVR micro-controller in our prototype runs at 2MHz and draws less than 1mA of current [18], two orders of magnitude less compared to the main processor. The sensor values are then written to the host processor via a UART connection.

MobileHub design decisions

The problem with sensor hubs is in the difficulty in programming them. We design MobileHub to leverage the sensor hub without additional programming effort. At a high level, MobileHub analyzes how sensor values affect application behavior and then rewrites the application based on the analysis. This can be implemented either by modifying the application source code or the binary. We choose to directly modify the binary that lets us evaluate existing apps even when the source code is not available.

MobileHub is primarily designed for application developers who would use the system as a last pass to optimize their applications. To do this, the developer provides a representative sensor trace to drive their application; in return, they get a MobileHub-instrumented app. When developers release new versions of their app, we expect that they will rerun the MobileHub optimization. Since MobileHub does not require source code, it can also be used by a third-party such as an app store, to ensure that their offerings have been power-optimized.

ARCHITECTURE

Figure 2 shows the MobileHub architecture, comprised of a four-step pipeline.

Step 0: Sensor Traces. The input to the MobileHub system is a sensor trace under representative conditions. Since the

purpose of the app is known, we can reasonably expect that such a trace can be collected. For instance, for a pedometer app, the sensor trace is a series of accelerometer readings under conditions of walking, running, standing, and sitting.

There are two obvious approaches to picking a proper sensor trace: a) the developer can upload a sensor trace that capture activities important to the app, b) alternatively we can run the application using a canonical "day in the life" trace during which a user performs a wide variety of typical tasks.

Step 1: Dynamic Taint Tracking (§Taint). In MobileHub, we use dynamic taint tracking [19] to track the flow of sensor data from when it is collected, to when it results in an application notification. MobileHub's taint tracker takes as input the app and a representative sensor trace. MobileHub instruments the app binary for taint tracking and runs the app against the sensor trace. The result of tracking is a *taint log* with time-stamped entries of each sensor reading and the application notifications. This log is the input to the next step.

For many applications, the instrumented version of the app can be executed without any manual intervention. But other apps may require user input. Automating application execution is an orthogonal research area where a lot of advances have been made [22, 35]. In the future, MobileHub will leverage these advances to autonomously run the instrumented application.

Step 2: Learning (§Learning). In the second step, MobileHub takes an input the taint log and learns the application behavior. It first labels and pre-preprocesses the taint log, and splits the data into training and testing dataset. MobileHub then trains the classifier with a 10-fold cross validation. The learning stage produces a simple classifier that, given a stream of sensor data, predicts if the sensor value can be buffered without affecting application semantics.

Step 3: Rewriting the Sensor hub and the Application (§Implementation). In the final step, MobileHub implements the learned classifier in the sensor hub firmware, and associates the classifier with the application.

Finally, MobileHub rewrites the original application, so that the application now receives either single or buffered sensor data directly from the sensor hub. Note that the sensor interface in most mobile platforms is asynchronous; i.e., the application is built to receive multiple sensor values at the same time. This makes our rewriting easier since the application's sensor interface does not need to change.

DYNAMIC TAINT TRACKING

Let's use a canonical fall detection application as an example. A fall detection app collects accelerometer readings, runs a fall detection algorithm, and sends out alarms when a fall is detected. We call the alarm an application notification. Our goal now is to track when a sensor value leads to the application notification. Notice that observing the application alone is insufficient. We might observe the alarm, but without knowing semantics, we cannot be certain that the alarm was raised in response to the accelerometer value. Static analysis is also not enough because it does not let us observe the application behavior during runtime under different sensing conditions. Instead we turn to information flow tracking [19].

```
void onSensorChange
(sensorEvent) {
[T]X = sensorEvent.val;[T]

[T]Y = function(X);

  printToScreen(Y);[T]
}
```

(a) Explicit Information Flow

```
void onSensorChange
(sensorEvent) {
[T]X = sensorEvent.val;[T]
  if (X > 10) {
  [T] Y += 1;
  }
  printToScreen(Y);[T]
}
```

(b) Implicit Information flow

Figure 3. Examples of explicit and implicit information flows from the sensor source at *sensorEvent.val* to the sink *printToScreen*. The taint tag is marked as $[T]$

Background: Implicit and Explicit Flow Tracking

Information flow tracking, or dynamic taint tracking, allows us to accurately track if a program input at the source affects the program output at the sink. In the fall detection example, if a flow of information is detected from the accelerometer source to the alarm notification sink, we can be sure that the alarm was raised in response to the accelerometer reading.

There are two kinds of information flow: *explicit* and *implicit*. Figure 3(a) shows an example of an explicit information flow from a source to the sink. In this example, the sensor value is the source and is associated with a *taint tag T*. The taint tag propagates during an assignment, from right to left. Due to the assignment at X, the taint tag propagates from the source to X. The taint tag then propagates through the program as X is assigned to other variables (in this example, to the variable Y). At the sink, we detect an information flow from the source.

Implicit information flow, on the other hand, is the flow of information through control flows, without *any* assignment operation. Figure 3(b) shows an example of an implicit information flow. Explicit information flow tracking will not capture the flow of information from X to Y because there is no direct assignment.

Both explicit and implicit flow tracking is required to completely capture the flow of information. However, existing taint tracking systems for smartphones [20, 36] do not support implicit information flow tracking because it can cause a taint explosion [20]. The problem is that, for sensing applications, sensor data is often propagated through implicit flows.

MobileHub's Taint Tracking

We borrow ideas presented in related work [16] to implement our own implicit information flow tracking system over TaintDroid [20], a tool that tracks explicit information flow.

Taint Sources and Sinks. We modified TaintDroid to track the flow of sensor data, different from the original TaintDroid that tracked privacy-related data. The sinks are where we capture application notifications. In our current implementation, we define an *application notification* to be any changes in application behavior that causes a user-perceivable output. Accordingly, our sinks are the network, the disk, the display, any vibrations, and sound effects, and any taint data sent to these sinks will be captured. Until tainted data reaches these sinks, the taint states of variables are held in memory. In the

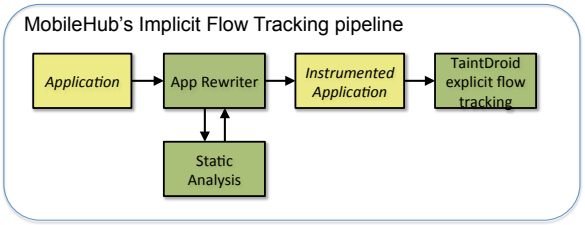

Figure 4. The steps taken for implicit flow tracking in MobileHub.

```
1  tag = σ(sensordata);
2  if (sensordata > X) {
3    stepCounter += 1;
4    σ(stepCounter)=σ(stepCounter)⊕tag;
5  }
```

Figure 5. An example MobileHub instrumentation to track implicit flows. The code in red is the instrumentation added by MobileHub. The function σ returns the taint tag of a variable.

future, the sinks can be extended to other events such as the application waking up a higher power sensor.

Implicit Information Flow Tracking. Our next task is to implement implicit information flow tracking over TaintDroid. Our approach is to use explicit information flow tracking to help implement implicit information flow tracking. Figure 4 shows an overview of MobileHub's taint tracking system.

Instrumentation. Given an application binary file, we first add some instrumentation. We start by identifying control branches that are likely to be tainted, that we call *TaintBlocks*. If we propagate the taint tag of the condition variables to each variable inside the TaintBlock explicitly, we can capture the implicit information flow.

For example, Figure 5 is a potential TaintBlock. The condition variable is *sensordata*. MobileHub adds instrumentation for each variable inside the TaintBlock (shown in red). The instrumentation in Line 1 retrieves the taint tag of the condition variable *sensordata*; σ returns a variable's taint tag. The second instrumentation is done in Line 4, where the variable inside the TaintBlock is associated with the taint tag of the control variable. At runtime, if *sensordata* is tainted, then *stepcounter* also becomes tainted using explicit information flow; if not, the instrumentation acts as a NOOP.

Instrumentation gets more complicated for nested conditions and method calls. MobileHub carefully tracks the tags at each level of the nested conditions in the first case, and maintains a global program stack in the second case. We omit details of the more complicated instrumentation in the interest of space.

Static Analysis for Efficient Instrumentation. We have described how our system tracks implicit information flows by instrumenting TaintBlocks. However, it would be highly inefficient to instrument all control flows because that would increase code size and slow down execution. MobileHub uses static analysis to determine when a condition variable of a control flow is unlikely to be tainted; MobileHub then avoids instrumenting this control flow.

In our evaluation across 20 applications (Table 1), MobileHub added less than 5% of code to most applications by leveraging

Figure 6. An example of of how MobileHub converts the taint log into training data.

the static analysis. Blindly instrumenting every control loop caused the program size to balloon, in some cases more than doubling the original size. Similarly, MobileHub's implicit taint tracking tracked over 80% of the sensor information flow from the source to the sink. Explicit taint tracking only tracked 20% of the flows. We omit showing results from the taint tracking evaluation in the interest of space.

The instrumentation and the static analysis is completely automatic. We note that although we use TaintDroid in our implementation, MobileHub can be built over any explicit information flow tracking tool that is designed for smartphones.

LEARNING THE BUFFERING POLICY

Again, let's look at a fall detection application example. The application uses a complex logic to determine when a series of accelerometer readings indicates a fall. MobileHub does not know this complex logic. It simply *learns* the statistical properties of the accelerometer readings that does not trigger an alarm notification.

If MobileHub's learning indicates that the sensor data may trigger an alarm, clearly it is *unsafe* to buffer the sensor values. This is because, if buffered, the application will not process the sensor data in time, and may not raise an alarm when the user falls down. If MobileHub's learning indicates that the sensor data will not trigger an alarm, then it is safe to buffer the sensor data. This is the core of MobileHub's buffering policy. We describe this is more detail below.

Labeling and Pre-processing

The input to our learning step is the taint log generated by the taint tracking tool. The log contains information about when a sensor value was collected and when an application notification occurred. We convert this log into our training set.

As a first step, we mark each sensor value in the log with a *trigger* or *non-trigger* label. A sensor value is marked with a *trigger* label if an application notification occurs immediately after the sensor value was collected. Figure 6(a) shows an example of how the sensor values in the taint log are labeled.

The challenge is that most sensing applications are idle most of the time as they wait for an interesting sensor event to occur.

As a result, the dataset is skewed heavily towards *non-trigger* labels, making learning difficult.

To improve learning under these challenges, we expand the labels by clustering the log into *active* and *inactive* periods. An active period contains at least one sensor value with a *trigger* label, indicating that it is not safe to buffer the sensor data. An inactive period is one where there are no sensor values with a *trigger* labels indicating that the sensor values can be buffered.

The *active* and *inactive* periods are determined using a *window size*. If any sensor value has a *trigger* label at any time in a sliding window, we label the entire window as *active*; otherwise, we label the window as *inactive*. Figure 6(b) shows an example labeling for a window size of 2. The samples in the active period are then labeled *active*, and samples in the inactive period are labeled *inactive* as shown in Figure 6(c).

Intuitively, *window size* depends on the sampling interval and the max delay in notification that the application can tolerate. We set the window size to be:

$$\frac{\textit{max tolerable delay}}{\textit{sensor sampling interval}}$$

In our implementation, we set the maximum tolerable delay to be 500ms for all apps.

There is an inherent trade-off between the application delay tolerance and power savings. If we buffer aggressively, we get higher power savings, but may delay the notification from the application. If we buffer conservatively, we will not delay application notification, but we reduce power savings. The *window size* parameter can be tuned to choose different trade-off points.

Classification Task and Buffering Policy

The task of our classifier now is to predict, given a series of sensor values, if the application is in the active or inactive period. Our buffering policy is as follows: If the classifier predicts that the application is in an inactive period, MobileHub buffers the sensor readings; if not, then MobileHub returns the sensor readings to the application.

MobileHub learns the classifier using the last several sensor values as input. This mirrors application logic, as applications often compute functions over several sensor values before making notification decisions. For instance, many applications use simple smoothing and thresholding techniques over a series of sensor values. Some applications implement more sophisticated algorithms such as clustering or frequency analysis. In this case, MobileHub may not be able to learn the logic even approximately, but it will be able to learn enough to buffer sensor data conservatively.

Features

As a first step, we used 97 training features to learn the classifier; the features including the magnitude of the last n sensor values; the maximum, minimum and mean value over different window sizes; the absolute difference in sensor data in each axis, etc. Before we implement the trained classifier on the sensor hub, we run a feature selection algorithm to select the top 10 most effective features using a leave-one-out cross validation [40]. This reduced amount of computation required for the sensor hub. The feature selection did not reduce the learning accuracy appreciably.

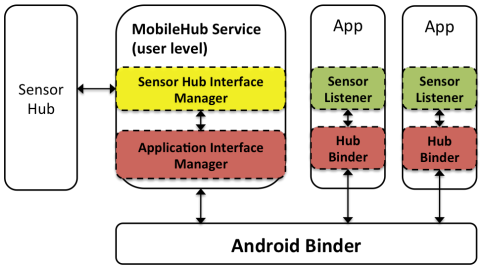

Figure 7. MobileHub implements a sensor hub service at the user level to interface the sensor hub with the application.

Implementing the classifier

We first train the classifier with our labeled sensor trace. After we have a trained classifier, we smooth the classification results to predict the active and inactive periods. We used the Weka platform [21] for learning the classifier.

We experimented with a standard set of learning algorithms ranging from linear algorithms—Naive Bayes, Logistic Regression, LibLinear—to non-linear algorithms— J48 Decision trees and Random Forest. We evaluated the classifier accuracy for all the 20 applications we use in our evaluation (see Table 1) against all the learning algorithms (Methodology in §Evaluation). Our evaluation showed that Naive Bayes and Random Forest performed consistently well in reducing both false positives and false negatives. Implementing the Random Forest model in a micro-controller is challenging because of memory constraints. Instead, we use Naive Bayes in our implementation and all our experiments. We omit showing the comparison results across the different learning algorithms in the interest of space.

Certain factors can lead to learning an inaccurate classifier. We discuss the implications of an inaccurate classifier in §Discussion.

IMPLEMENTATION

Sensor hub prototype

Commercial sensor hubs are already integrated into the phone [2, 31], but are closed systems and cannot be modified. Instead, for evaluation purposes, we prototype our own sensor hub using an 8-bit Atmel AVR micro-controller, XMega-A3BU Xplained [8]. Figure 8 shows our prototype setup. To the micro-controller, we add a Spark IMU sensor (combined 3-axis accelerometer, gyroscope, and magnetometer sensor) through the I^2C interconnect. For convenience, the AVR micro-controller communicates with the USB host device by tunneling serial communication through USB [9].

For a given application, we install the corresponding classifier function in the sensor hub firmware. This app-specific classifier decides, given a sensor input, if the sensor value should be buffered or sent to the application.

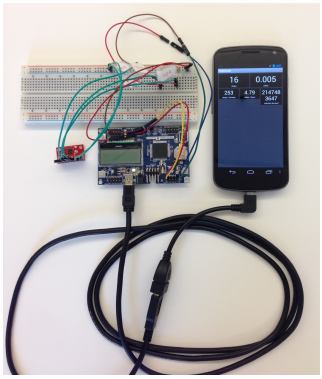

Figure 8. Sensor hub prototype.

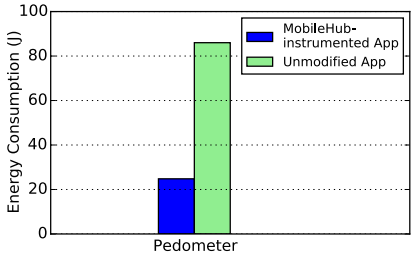

Figure 9. End-to-end MobileHub Experiment.

Android Implementation

MobileHub is implemented as a user level service as shown in Figure 7. The MobileHub service acts as a proxy between the external sensor hub and the application: it interfaces with the sensor hub to read sensor data, and with the applications to supply the sensor data. We implement the communication between the sensor hub and the MobileHub service using the Physicaloid Library [6]. We implement the communication between the MobileHub service and the application using the Android Binder, an IPC mechanism. Because the service is at the user level, MobileHub does not require a rooted phone.

To rewrite an app, we simply replaces the sensor calls in the application with calls to the MobileHub service. We call this rewritten app a *MobileHub-instrumented app*. We rewrite using Soot [41], a tool that provides translation bidirectionally between the application binary (in Android Dalvik byte code) and the Jimple intermediate language [42]. The rewriting is done completely automatically. Currently, we sign the rewritten app using our own credentials; in the future, the application developer could re-sign the app to preserve integrity.

End-to-end experiment

As a proof of concept, we perform an end-to-end experiment using the set up in Figure 8 on a Galaxy Nexus phone. We instrument *pedometer*, one of the 20 applications in our larger scale evaluation (see Table 1). To emulate mobility we let the phone be stationary for 30 seconds and manually moving it for 30 seconds, for a total of 5 minutes.

We use the Monsoon power monitor to measure power consumption of the entire unit: the phone and the sensor hub. The power monitor samples the current drawn from the battery at a fine-grained frequency of 5000 Hz. By design, the power measurements include the power consumed at the sensor hub for sensing, making buffering decisions, writing sense data to the phone, as well the power consumed by the phone.

Power results: Figure 9 shows that the power consumption of the MobileHub-instrumented app was only 22J compared to 84J when using the default app. MobileHub is able to keep the CPU idle for long periods of time, leading to the reduction in power consumption.

Sensor Hub micro-benchmarks: During the experiment, the sensor hub power consumption was 0.55J, which is only 2.5% of the total power consumption of the MobileHub-instrumented app. We also found that the time take for the sensor hub to perform the classification was barely significant. The sensor hub was able to run the *Pedometer* specific classifier for 100 runs in under 350 microseconds.

TRACE-DRIVEN EVALUATION

The end-to-end experiment shows that MobileHub is practical and can be implemented in smartphones. However, the set up is bulky and cannot be moved from the lab. Further, sensor applications are largely driven by the user's environment which is difficult to replicate in the lab.

To perform controlled and reproducible experiments, we perform phone emulation using real sensor traces. The goal of our evaluation is to show that, for a diverse range of applications, the MobileHub-instrumented app: (a) reduces power consumption significantly, and (b) only minimally changes the application semantics.

All experiments were performed on two Galaxy Nexus and two Nexus S phones that run Android version 4.2.2.

Applications

We download 20 applications from Google Play for evaluation (Table 1). The table shows the name of the app, the purpose of the app, and the sampling frequency used by the app. Accelerometer is by far the most popular sensor; accordingly, 75% of the applications in our evaluation use the accelerometer sensor. The other applications use gyroscope, magnetometer, and orientation sensors. The orientation sensor is a software sensor that combines magnetometer and accelerometer.

Our goal when choosing the applications was to cover a diverse set of tasks: step counting, fall detection, metal detection, theft and motion monitoring, earthquake detection, sleep monitoring, and driver monitoring. We searched for each of these tasks and randomly picked top application(s) returned on Google Play. We rewrite each application to create a MobileHub-instrumented version as described in §Implementation.

Methodology

For emulation, we run each application on the phone. However, the applications receive sensor values from a trace file rather than from the on-board sensors. For instance, a step-counting application emulated using a walking trace will behave as if user were taking the same steps captured in the trace.

The trace file embeds meta information about the buffering policy. In the default case, there is no buffering, and the sensor values are returned at the rate requested by the application. For the MobileHub experiments, we run the classifier over the

Name	Google Play Store ID	Purpose	Sensor	Sample Interval (ms)
nwalk	pl.rork.nWalk	Step counting	Accelerometer	20
pedometer	bagi.levente.pedometer	Step counting	Accelerometer	20
stepcounter	Stepcounter.Step	Step counting	Accelerometer	60
appsone	net.appsone.android.pedometer	Step counting	Accelerometer	200
virtic	jp.virtic.apps.WidgetManpok	Step counting	Accelerometer	60
walking	cha.health.walking	Step counting	Accelerometer	20
lodecode	com.lodecode.metaldetector	Metal detector	Magnetometer	60
imkurt	com.imkurt.metaldetector	Metal detector	Magnetometer	20
tdt	com.tdt.magneticfielddetector	Metal detector	Magnetometer	20
multunus	com.multunus.falldetector	Fall detector	Accelerometer	20
iter	com.iter.falldetector	Fall detector	Accelerometer	40
t3lab	it.t3lab.fallDetector	Fall detector	Accelerometer	20
fall	com.fall	Fall detector	Accelerometer	20
jietusoft	com.jietusoft.earthquake	Earthquake detector	Accelerometer	20
vibration	ycl.vibrationsensor	Earthquake detector	Orientation	20
posvic	cz.posvic.fitnessbar.sleeptrack	Sleep monitoring	Gyroscope	20
myway	myway.project.sleepmanagement	Sleep monitoring	Accelerometer	40
driving	jp.co.noito. Accelerometer	Driver monitoring	Accelerometer	40
motion	com.app.accelerometer	Motion detector	Accelerometer	40
thefthead	com.thefthead.appfinalsettings	Theft detector	Accelerometer	200

Table 1. Applications used in our evaluation.

trace file to determine the buffering policy at each instance of the trace; i.e., given the sensor reading, should the reading be buffered or sent to the application. This policy is embedded in the trace file.

Trace Collection We built an Android app to collect user traces as participants perform representative tasks for each application type. We did not monitor the participant as they performed any of the tasks. We did not tell them the purpose of the trace collection. None of the authors participated in the trace collection. The institutional human subjects review board determined that the trace collection was not human subjects research.

For step detection, we asked the participant to simply carry the phone around as they go about their normal activities. For sleep monitoring and driver monitoring apps, we asked the participant to start the trace collection before they go to sleep and start to drive, respectively. For fall detection apps, earthquake detection and motion detection apps, we asked the participant to perform the representative task every once in a while. For example, for fall detection, we asked the user to drop the phone every now and then. We did not instruct them on exactly how or when they should perform the task.

We collected 10 traces each for sleeping, driving, and day-in-the-life activities, and 5 traces for the other activities. One participant could collect traces for multiple application types. In total, we collected traces from 21 participants. The traces were between 14 minutes and an hour long, each containing about 40k to 200k sensor samples.

Running the classifier on the trace: To train the classifier, we randomly pick 1-3 user traces. We use the remaining traces for testing. We perform a 10-fold cross validation to ensure that our results are sound and not influenced by the choice of training and testing sets. We omit the results evaluating the classifier accuracy in the interest of space. We use the learned classifier for each application in the rest of our experiments. Each trace was down-sampled according to the application's sampling rate if needed.

Power Accounting

We break down the power measurement in two parts: the power consumption of the application running on the phone and the power consumption of the sensor hub.

We directly measure the power consumption of the phone using the Monsoon power monitor. In the end-to-end experiments, we measure the power consumption of the sensor hub also. However, power measurements of the sensor hub is tricky during emulations because we are replacing the sensor hub functionality using the emulator. Instead, we build a model of the sensor hub power consumption using real experiments.

We use the MobileHub implementation set up (Figure 8) and repeat each representative sensor hub activity: reading sensor values, running the classifier, buffering, and writing the sensor values back to the phone. We repeat these activities for varying parameters: different sensor frequencies, different buffering rate, different classifiers, etc. As before, we connect the set up to the Monsoon Power Monitor to measure the power consumption. Using the measurement, we build a model for sensor hub power consumption. We verify the accuracy of the model using our end-to-end experiment.

The total power consumption is computed as the sum of the phone power consumption and the sensor hub power consumption, minus the power consumed by the emulation trace service. Finally, we measure the power consumption of the USB OTG cable and subtract this from the total power consumption.

Metrics

Power consumption is the most important metric against which we evaluate MobileHub. However, it is equally important that MobileHub does not change the application semantics. Recall

that MobileHub delays when the application receives sensor data, and this can potentially delay application notification. For example, a fall detection application may not call for help immediately after the fall occurs.

We measure the delay in notification using 3 metrics: *fraction of delayed notification*, the *95th percentile delay* and the *max notification delay*. We first run the original application over a user trace and record the application notifications. We then run the MobileHub-instrumented app using the same trace and compare the two sets of notifications. The notification is said to be delayed if the MobileHub-instrumented application does not generate a notification in the same active period (of 500ms length) as the original application.

Results

Power consumption: Figures 10–14 show the improvement in power consumption when using MobileHub for 17 of the 20 apps. We present the results for three user traces not part of the training set.

Taking the Metal detector apps out, the average power improvement across all the other apps, across both traces is 70%, with a median of 74%. The power improvement in some cases is as high as 81%.

The metal detector apps provided an average energy improvement of 33%. On further analysis, we find that the metal detection traces induced frequent app notifications. MobileHub only provides benefits when the application is idle for long periods of time, and the notification behavior markedly reduced the benefits of MobileHub. This notification behavior is likely because participants collecting the traces were often close to a metal source.

We found that 3 out of the 20 apps we evaluated—*multunus*, *walking*, and *myway*—were structured in a way that could not leverage MobileHub. For that matter, the apps cannot leverage any sensor hub optimization unless they are restructured. We discuss these later.

Effect on application semantics: Tables 2–5 show how often the notifications were delayed and by how much. The first column is a fraction of notification delays. The second and third column shows how much the delay was in terms of the *95th percentile delay*, and the *max notification delay* respectively.

The results show that MobileHub induces almost no delay in app notifications in most cases. For instance, the *nWalk* application had 3914 notifications, out of which MobileHub delayed only 1 notification. In many cases, MobileHub did not delay any notification.

Even when delayed, a notification for an application is delayed only 0.015 times across all the applications, on an average. In 80% of the cases the maximum delay is less than 1 seconds. For certain apps such as *appsone*, the max delay is as high as 59.6 seconds. This is because *appsone* samples the sensor value at low sampling rate of 200ms. When the classifier is wrong, the time it takes to recover from the wrong classification increases due to the low sampling rate.

App	nwalk	pedo-meter	step counter	appsone	virtic
Delay/Total	1/3914	0/1232	0/1132	4/2121	0/7483
95th delay(s)	0.46	0.46	0.52	0.80	0.52
Max delay(s)	1.86	0.48	0.52	59.60	0.52

Table 2. Notification delay metrics for step counting apps.

App	lodecode	imkurt	tdt	jietusoft	vibration
Delay/Total	0/39	2/142	0/3688	0/33	0/415
95th delay (s)	0.45	0.46	0.46	0.48	0.46
Max delay (s)	0.45	0.98	0.48	0.48	0.48

Table 3. Notification delay metrics for metal detection and earthquake detection apps.

App	iter	t3lab	fall
Delay/Total	0/7	3/26	0/11
95th delay (s)	0.40	1.86	0.38
Max delay (s)	0.40	15.28	0.38

Table 4. Notification delay metrics for fall detection apps.

App	posvic	driving	motion	thefthead
Delay/Total	1/36	0/2022	0/54	6/65
95th delay (s)	0.48	0.48	0.48	1.60
Max delay (s)	0.64	0.48	0.48	2.80

Table 5. Notification delay metrics for other apps.

As discussed earlier (§Labeling), the *window size* parameter can be tuned to trade-off between reducing power consumption and avoiding notification delays.

Static buffering policy: An alternative to MobileHub's buffering policy is to use static buffering. The problem with a static buffer is that a small buffer does not provide significant power benefits since it does not keep the main processor idle for long. However, a large buffer can significantly delay the application notification, changing the app semantics.

Figure 15 shows that *percentage notification delay* metric when using a static buffering policy with buffer size 10, 30 and 50 sensor samples, compared to MobileHub. The application's notification gets delayed up to 46% of the time when the buffer size is set to 50. Contrast this to MobileHub, which delays notification 0% of the time (see Table 2).

On further experimentation, we found that using a static buffer size of 10 samples reduced power consumption of the *pedometer* application by only 24.8%, compared to the 68% power reduction achieved by MobileHub (see Figure 11). MobileHub sets the buffering rate dynamically in response to the sensor readings. For example, in this experiment, MobileHub buffered between 0 to 18413 samples, with an average buffer size of 91 sensor samples.

Apps for which MobileHub does not help: For three apps we instrumented, MobileHub did not provide benefits. The *walking* app incorrectly uses the host's clock value to time the sensor data, rather than use the timestamp included in the sensor event. This is wrong programming practice when dealing with an asynchronous interface. Doing so results in MobileHub-instrumented app observing out-of-date and seemingly non-uniform streams of readings. Therefore, we do not optimize this app.The *MyWay* app constantly updates the screen. The *Multunus* app requires the screen to be switched-

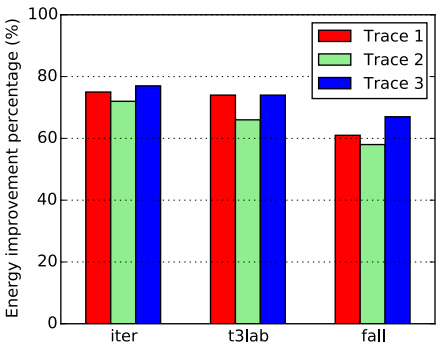

Figure 10. Fall detection apps.

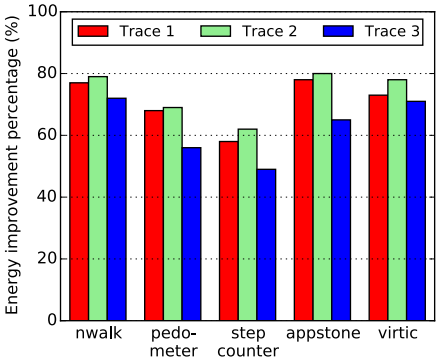

Figure 11. Step detection apps.

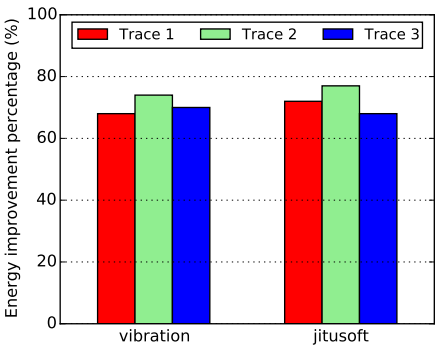

Figure 12. Earthquake detection apps.

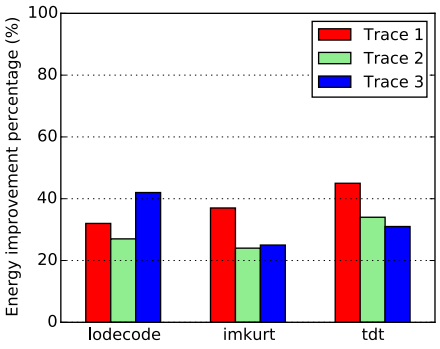

Figure 13. Metal detection apps.

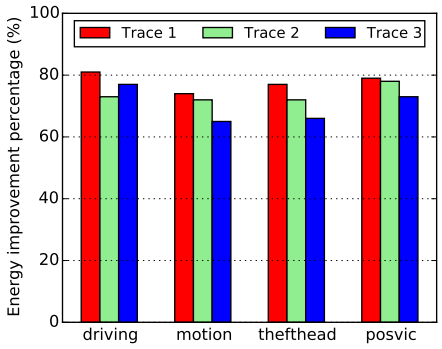

Figure 14. Misc. motion detection apps.

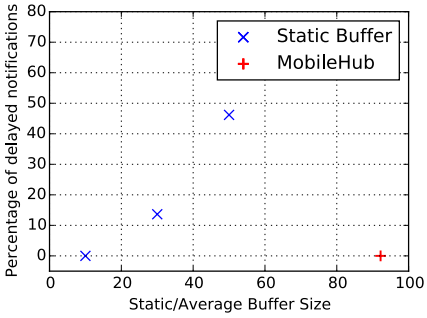

Figure 15. Percentage delayed notification when using a static buffer vs using MobileHub's dynamic buffering for the *pedometer* app.

on continuously. In both cases, the main processor cannot be idle, negating the benefits of MobileHub.

The sensor hub architecture is designed to optimize power consumption of apps based on two assumptions: (a) the main processor can be idle for larger periods of time, and (b) the application will behave correctly even if it receives multiple sensor values at the same time. Since the three applications break these assumptions, they can benefit from the sensor hub architecture only if they are restructured.

RELATED WORK

MobileHub uses concepts from prior research on heterogeneous hardware, low-power sensing, and taint tracking. Below, we describe these related efforts and place MobileHub in context.

Power efficiency of sensing applications: There has been considerable work in reducing the power consumption of high power sensors such as GPS [45, 30, 15, 32, 26]. For example, the ACE [32] system uses user context to correlate low power sensor values with that of high power sensors. LAB abstraction [26] allows users to specify their latency, accuracy, and battery requirements, and the system automatically chooses sensor algorithms based on the requirements. These techniques are complimentary to MobileHub and can help further reduce the power consumption of sensing applications.

Heterogeneous architectures: Dedicated co-processors for sensors are a natural way to support always-on sensing efficiently and many commercial mobile processors now include sensor hubs. Bodhi *et al.* [33] have shown that careful coding of a pedometer application to make use of a sensor hub reduces power consumption by up to 95%. Going further, Ra *et al.* [34] analyze different design choices that the developer can use to partition a mobile application between the phone and the sensor hub.

Approaches such as Reflex [28] and K2 [27] provide support for a software distributed shared memory, so that the developers can communicate between the main processor and co-processor without worrying about cache coherency. These approaches all show the power efficiency can be had from a sensor hub. Our work hopes to get the power efficiency, but without requiring additional programming effort.

The sensor hub itself is not a new idea and the approach of using a tiered system for energy efficiency has been used in several settings before [11, 37, 38].

Alternate hardware support: The sensor hub design integrates all of the sensors into a single centralized resource that serves as the hosts single point of contact for sensor data. An alternative to a sensor hub design is to embed buffering and simple programmable logic into the sensors themselves. Such an approach is referred to as "smart sensing" and commercial versions exist (e.g., The Micronas Hall-effect sensors). This approach work well for special-purpose devices, but scales poorly and suffers from not being as programmable as a sensor hub. In the same vein, big.LITTLE ARM architecture [24] is a new architecture that allows switching between a higher power and a lower power processor with little latency. MobileHub can greatly benefit from such an architecture.

Taint tracking: Dynamic taint analysis in smartphones broadly used in the context of application security and privacy [20, 36]. Systems such as FlowDroid [12] use static taint analysis, also to reduce privacy leaks. Recently, taint tracking has been used to detect energy leaks in applications [44]. However, these existing taint tracking tools for smartphones only capture explicit information flows.

Capturing implicit flows is challenging and is a topic of continuing research efforts [17, 14, 25]. To avoid implicit flow from generating a large amount of false positive, both Bao *et al.* [14] and DTA++ [25] identify special types of implicit flow that specific control dependencies. The Dytan [17] system is a more general framework for implicit flow tracking, but targets x86. Vogt *et al.* [43] implement implicit flow tracking to prevent cross-site scripting attacks on web applications.

Given the importance of tracking implicit flows in smartphones, a recent work called Edgeminer [16] provides a technique to track implicit flows during static analysis. In MobileHub, we present a technique for implicit flow tracking during dynamic analysis for smartphones. MobileHub builds upon on TaintDroid [20], a dynamic flow tracking technique for smartphones that only tracks explicit flows. Our work borrows ideas from the existing work, especially from the work presented by Vogt *et al.* [43]. Similar to their work, we identify control flow loops that are likely to be tainted, and instrument these loops.

DISCUSSION

What applications cannot benefit from MobileHub? MobileHub is not a cure-all for inefficient mobile sensing applications. For example, a Pedometer application may update the user's step count twenty times per second. While it may not be necessary to update the user's step count that frequently, MobileHub does not know this. It will try to optimize based on the original application semantics and learn that the sensor values cannot be buffered for long periods of time.

MobileHub provides the biggest improvement for applications that make use of simple high-frequency sensors such as accelerometer, gyroscope and magnetometer. It provides little power improvement for GPS since the sensor itself is high power and wakes the processor less frequently.

What other applications can benefit from MobileHub? The applications we select in our evaluation have similar characteristics; they are based on detecting changes to common on-board sensors such as the accelerometer. This focus is in large part due to the availability and popularity of such apps. However, the core idea in MobileHub can be used in other scenarios. For example, if a microphone is connected to the sensor hub, the MobileHub system could directly be applied to apps making audio-based classification. Similarly, MobileHub can be extended to save power for networked sensors [23, 39]. This will of course require that the sensor hub have access to the network stack implementation [7] to allow it to buffer and delay the delivery of networked sensor packets.

What happens if the classifier is inaccurate? Certain factors can lead to learning an inaccurate classifier. First, we assume that the sensor value deterministically affects application behavior. However, applications may make decisions that combine the sensor values with other inputs. Our taint tracking approach cannot track this, and will not learn a classifier that predicts important sensor reading values. Similarly, we assume that the application notification occurs immediately after an important sensor event. However, application may produce a notification some time after the sensor event occurs, because of computation time. In both cases, the result is that the classifier accuracy is low. In MobileHub we measure the accuracy of the classifier using a test set, and return the application unoptimized if the accuracy is low.

Finally, MobileHub is designed to reduce the notification delay, but does not provide guarantees. Although our evaluations show that MobileHub does not delay notifications by more than 1 second in most cases, we cannot provide guarantees on the maximum notification delay. One possible way to reduce the severity of this problem is to use online loopback. We envision deploying MobileHub alongside an online loopback system that collects the user traces and notifications periodically. A cloud service analyzes these traces to replay the notification delays as seen by the application. Based on this analysis, MobileHub can either decide to be aggressive or conservative in buffering.

CONCLUSIONS

For sensor hubs to realize their potential, we must work out how application developers will use them. We have presented MobileHub, a system that rewrites applications to leverage the sensor hub for power efficiency but without additional programming effort. To do this, MobileHub analyzes applications to identify and learn when the application acts on the sensor data to generate a user-perceivable notification. Based on this learning, MobileHub simply buffers the sensor data when the application does not require it, and notifies the application with the buffered data when the application needs to act on it. By buffering the sensor data, MobileHub allows the application and the main processor to be idle, saving power. We implemented *MobileHub* over Android and prototyped a sensor hub comprised of an 8-bit AVR micro-controller for experimentation. We experiment with 20 sensor applications that we download from Google Play and use MobileHub to rewrite the applications. The applications perform diverse tasks—Step counting, Fall detection, Sleep monitoring, Driver monitoring, Earthquake detection, and Metal detection. Our evaluation demonstrates MobileHub's ability to significantly reduce power consumption with almost no change to the application semantics.

Acknowledgments

We gratefully acknowledge the anonymous reviewers. This work was supported by the Intel Science and Technology Center for Pervasive Computing and National Science Foundation (CNS-1217644, CNS-1318396 and CNS-1420703). Special thanks to Xin Xu, Eric Nhan, and Jeremy Teo for help with user data collection, power measurements, and automation. Thanks to Eric Yuan for helping us with an earlier version of the MobileHub system.

REFERENCES

1. Activity recognitionclient: http://developer.android.com/reference/com/google/android.
2. Apple M7. http://en.wikipedia.org/wiki/Apple_M7.
3. BeWell Mobile Application. https://play.google.com/store/apps/details?id=org.bewellapp&hl=en.
4. Intel Context Sensing SDK. https://software.intel.com/en-us/context-sensing-sdk.
5. ios core motion framework reference: https://developer.apple.com/library/ios.
6. Physicaloid : Physical computing with a smartphone. http://www.physicaloid.com/?lang=en.
7. RF, Wi-Fi and Other Wireless Microcontroller-Based Solutions. http://www.atmel.com/products/wireless/.
8. XMega-A3BU XPlained. http://www.atmel.com/tools/XMEGA-A3BUXPLAINED.aspx.
9. Atmel: USB Device CDC Application. http://www.atmel.com/Images/doc8447.pdf, 2011.
10. TivaWare Sensor Library User Guide. http://www.ti.com/lit/ug/spmu371/spmu371.pdf, 2015.
11. Agarwal, Y., Hodges, S., Chandra, R., Scott, J., Bahl, P., and Gupta, R. Somniloquy: augmenting network interfaces to reduce pc energy usage. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, USENIX Association (Berkeley, CA, USA, 2009), 365–380.
12. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, ACM (New York, NY, USA, 2014), 259–269.
13. Austin, T. H., and Flanagan, C. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, ACM (New York, NY, USA, 2010), 3:1–3:12.
14. Bao, T., Zheng, Y., Lin, Z., Zhang, X., and Xu, D. Strict control dependence and its effect on dynamic information flow analyses. In *Proceedings of the 19th international symposium on Software testing and analysis*, ACM (2010), 13–24.
15. Bo, C., Li, X.-Y., Jung, T., Mao, X., Tao, Y., and Yao, L. Smartloc: Push the limit of the inertial sensor based metropolitan localization using smartphone. In *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, MobiCom '13, ACM (New York, NY, USA, 2013), 195–198.
16. Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., and Chen, Y. Automatically detecting implicit control flow transitions through the android framework. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS15* (2015).
17. Clause, J., Li, W., and Orso, A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ACM (2007), 196–206.
18. DATASHEET, A. 8-bit avr® microcontroller with 4/8/16/32k bytes in-system programmable flash, 2010.
19. Denning, D. E., and Denning, P. J. Certification of programs for secure information flow. *Commun. ACM 20*, 7 (July 1977), 504–513.
20. Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010).
21. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter 11*, 1 (2009), 10–18.
22. Hao, S., Liu, B., Nath, S., Halfond, W. G., and Govindan, R. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, ACM (New York, NY, USA, 2014), 204–217.
23. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. System architecture directions for networked sensors. In *ACM SIGOPS operating systems review*, vol. 34, ACM (2000), 93–104.
24. Jeff, B. Advances in big. little technology for power and energy savings. *ARM White Paper* (2012).
25. Kang, M. G., McCamant, S., Poosankam, P., and Song, D. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS* (2011).
26. Kansal, A., Saponas, T. S., Brush, A. J. B., McKinley, K. S., Mytkowicz, T., and Ziola, R. The latency, accuracy, and battery (lab) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing. In *OOPSLA*, ACM (2013), 661–676.
27. Lin, F. X., Wang, Z., and Zhong, L. K2: A mobile operating system for heterogeneous coherence domains. In *ASPLOS* (2014).
28. Lin, X. F., Wang, Z., LiKamWa, R., and Zhong, L. Reflex: Using low-power processors in smartphones without knowing them. In *ASPLOS* (2012).
29. Lisa, E. Intel Unveils New Merrifield Smartphone Chip With Integrated Sensor Hub. http://blog.laptopmag.com/intel-merrifield-smartphone-chip.

30. Liu, J., Priyantha, B., Hart, T., Ramos, H. S., Loureiro, A. A. F., and Wang, Q. Energy efficient gps sensing with cloud offloading. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys '12, ACM (New York, NY, USA, 2012), 85–98.

31. Morales, M. An Introduction to the Tiva™ C Series Platform of Microcontrollers. Tech. rep., Texas Instruments, April 2013.

32. Nath, S. Ace: Exploiting correlation for energy-efficient and continuous context sensing. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, ACM (New York, NY, USA, 2012), 29–42.

33. Priyantha, B., Lymberopoulos, D., and Liu, J. Littlerock: Enabling energy-efficient continuous sensing on mobile phones. In *IEEE Pervasive Computing* (2011).

34. Ra, M.-R., Priyantha, B., Kansal, A., and Liu, J. Improving energy efficiency of personal sensing applications with heterogeneous multi-processors. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, ACM (New York, NY, USA, 2012), 1–10.

35. Ravindranath, L., Nath, S., Padhye, J., and Balakrishnan, H. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, ACM (New York, NY, USA, 2014), 190–203.

36. Rosen, S., Qian, Z., and Mao, Z. M. Appprofiler: A flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, ACM (New York, NY, USA, 2013), 221–232.

37. Shih, E., Bahl, P., and Sinclair, M. J. Wake on wireless: an event driven energy saving strategy for battery operated devices. In *Proceedings of the 8th annual international conference on Mobile computing and networking*, MobiCom '02, ACM (New York, NY, USA, 2002), 160–171.

38. Sorber, J., Banerjee, N., Corner, M. D., and Rollins, S. Turducken: hierarchical power management for mobile devices. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, MobiSys '05, ACM (New York, NY, USA, 2005), 261–274.

39. Stankovic, J. A., Wood, A. D., and He, T. Realistic applications for wireless sensor networks. In *Theoretical Aspects of Distributed Computing in Sensor Networks*. Springer, 2011, 835–863.

40. Stone, M. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society. Series B (Methodological)* (1974), 111–147.

41. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, IBM Corp. (2010), 214–224.

42. Vallee-Rai, R., and Hendren, L. J. Jimple: Simplifying java bytecode for analyses and transformations.

43. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS07* (2007).

44. Zhang, L., Gordon, M. S., Dick, R. P., Mao, Z. M., Dinda, P., and Yang, L. Adel: An automatic detector of energy leaks for smartphone applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, ACM (New York, NY, USA, 2012), 363–372.

45. Zhuang, Z., Kim, K.-H., and Singh, J. P. Improving energy efficiency of location sensing on smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, ACM (New York, NY, USA, 2010), 315–330.