Improving Power Efficiency Using Sensor Hubs Without Re-Coding Mobile Apps

Haichen Shen, Aruna Balasubramanian, Eric Yuan, Anthony LaMarca, and David Wetherall*
*University of Washington, *Intel*

Abstract

Emerging sensor hubs for smartphones allow long-lived sensing applications to run efficiently, but they require applications to be developed to new APIs. We develop *MobileHub* on Android to show how unmodified mobility applications can be translated with bytecode rewriting to use sensor hubs. The key to our approach is to use data and control information flow tracking to learn how applications use sensor values, and to map this usage to a simple API we developed to use the sensor hub. In experiments on three applications downloaded from the Android marketplace, we achieved power gains of up to 80%.

1 Introduction

Smartphone applications that continuously sense the ambient environment are becoming popular for a diverse range of uses, from providing context-aware services [8], to monitoring the health of Parkinson patients, to encouraging users to be more social and active (BeWell [17]). However, continuous sensing applications quickly drain the battery of existing smartphones [20] because they prevent the main processor from sleeping: to obtain each sensor reading, not only the sensor, but most of the phone circuitry and the main processor need to be powered up.

To efficiently support continuous sensing applications in the future, dedicated low-power micro-controllers, often called *sensor hubs*, are being integrated into smartphones, e.g., Texas Instruments's Tiva [9], Intel's Merrifield [5], and Apple's A7 [2]. With this hardware support, the low-power co-processor can collect, filter, and process sensor data, while the main processor stays in the sleep state.

While this architecture offers the potential for low-power, continuous sensing, it complicates programming by changing what was formerly a homogenous computing platform into a heterogeneous one. Some systems expose this heterogeneity directly to the application developer. TI's Syslink [7], for example, provides a facility to program the micro-controller as well as a message passing API between the host and sensor hub. While this approach provides full functionality, it means that applications must be developed specifically for this model (often using multiple languages). Research systems can ease this task somewhat: Reflex [18] uses a distributed shared memory abstraction so that developers do not need to manage communication between the main processor and sensor hub, but they must still specially code their applications.

A second approach is to sequester the sensor hub behind system libraries that expose a set of commonly-used functions that are optimized to run on the sensor hub. Apple's A7 processor uses this to approach to expose IMU functionality via its M7 "motion coprocessor". This approach offers the benefits of portability and ease of application development, but it still requires applications to be coded (or recoded) to use the system libraries, and limits its power improvements to the functionality anticipated by the library designers.

In this paper, we argue for a third approach: translating existing applications to leverage the sensor hub for power efficiency. The insight that makes this approach possible is that mobility apps tend to use continuous sensing in structured ways that are not deeply intertwined with application processing. Typically, an "inner loop" of sensing with a minimal computation gathers data at a high-rate until a pre-defined condition is met, at which point substantial "outer loop" application processing on the data may occur. By identifying and leveraging this structure, we can extract the majority of the power gains enabled by the sensor hub without re-coding applications.

To this end, we design *MobileHub*, a system that uses binary rewriting of the application bytecode to offload sensing and computation to the sensor hub. The translation is transparent to the developer and does not require source code. The challenge, of course, is to determine how to offload sensing tasks without delaying or altering the application behavior. For example, some applications may gather fifty readings before processing them, allowing us to buffer all fifty readings at the sensor hub. Other applications may rely on the near-real time delivery of sensor events, to detect motion for example, and any extra delay would alter the user experience.

We use information flow tracking [16, 12] to learn how an application uses sensor data. As we developed our approach, we learned that it is essential to track control flow as well as data flow, even though control flow is seldom tracked in prior work because of the problem of taint explosion. This is because sensing applications often determine whether to perform further computation based on the value of the sensor reading. Since the sensor value itself is often not retained, tracking data flow alone is insufficient.

We build *MobileHub* on Android, and implement information flow tracking by extending the TaintDroid [16] data flow system to also track control flow. We then used our system to study the sensor usage patterns of seven mobility tracking applications that we download from the Android Marketplace. The different apps use sensor data in different ways. Nonethless, a simple sensor hub API that allows applications to specify the sensing rate and buffering policy is sufficient to capture the inner loop behavior for offloading.

MobileHub learns the application's sensor usage and rewrites the application Dalvik [15] bytecode to use our sensor hub API. We implement this API in the Android OS framework, and prototype the sensor hub using an 8-bit Atmel AVR micro-controller, to which we attach several sensors. We used our complete system to experiment with three applications. Our evaluation shows that the power consumption of these applications in the lab is reduced by up 80% for stationary scenarios and by over 50% for mobile scenarios for 2 of the applications. We then estimated power savings using traces of real user behavior, and find that *MobileHub* can decrease power consumption by up to 80%

Our contributions are as follows:

- the design and implementation of *MobileHub*, a system that enables an application to use a sensor hub without any re-coding by the developer

- an information flow tool that tracks data and control flows to identify how applications use sensor data; we present a technique that propagates taint to portions of the application without suffering from taint explosion in our structured setting.

- an evaluation that shows off-the-shelf sensor applications can be translated to use the sensor hub and realize up to 80% power gains.

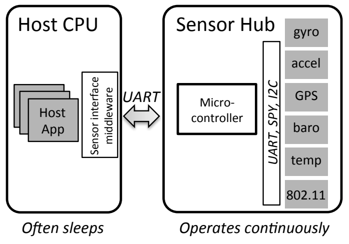


Figure 1: Example sensor hub architecture.

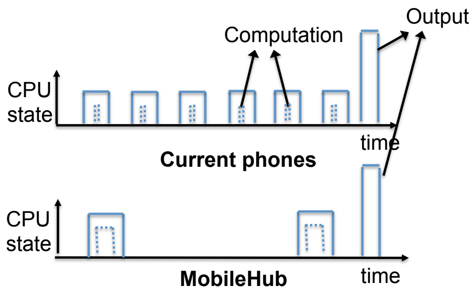


Figure 2: CPU activity timing diagram for a hypothetical application, with and without *MobileHub*. Figure 11 shows the power profile of a real application with and without *MobileHub*, and the figure matches well with our hypothetical example.

2 *MobileHub* Approach

The goal of *MobileHub* is to allow sensor applications to seamlessly leverage low power sensor hub hardware. Below, we describe the sensor hub architecture, provide an example to illustrate how leveraging the sensor hub can reduce power consumption. We then describe our approach.

2.1 What is a sensor hub?

A sensor hub is dedicated hardware that serves as an intermediary between the main platform and the sensors. Sensor hubs are ideally suited for reducing wakeup overhead be- cause they allow sensing without waking the host CPU.

Figure 1 shows an example sensor hub architecture. Sensor hub designs typically use of a micro-controller in lieu of a microprocessor, because of its low cost and power efficiency. For example, the AVR micro-controller that we use in our prototype running at a speed of 2MHZ, is designed to draw less than 1mA of current [4], an order of magnitude less than smartphones. The micro-controller has several standard buses (12C, SPY, UART) to connect with sensors as well as connect with the phone. These buses also draw very little power.

2.2 How we leverage the sensor hub

Figure 2 illustrates how *MobileHub* uses a sensor hub to improve the power consumption, using a hypothetical example. The figure shows the CPU state of a stereotypical sensor application. This application periodically wakes up, receives a sensor sample, performs a small computational task over the sample, and goes back to sleep.

More importantly, the application in this example produces user-perceivable output much less often than the sensor collection frequency. This output could be an update to the screen, data over the network, or writing data to disk; i.e., any activity that has effect on the user. Related work [20, 18] show that several sensor applications follow this pattern.

Given this application profile, the goal of *MobileHub* is to leverage one specific feature of sensor hubs: its ability to buffer sensor data and allow the main processor to stay idle. *MobileHub* modifies the rate at which sensor samples are collected. As a result, the CPU is idle for long periods of time, as shown in Figure 2), marked *MobileHub*.

Of course, the sensor hub should be able to preempt the application when it detects that the application needs to process sensor data. The goal of *MobileHub* is to buffer sensor samples, but not modify the timing of user-perceivable outputs.

By allowing the CPU to be idle for longer, *MobileHub* can significantly improve the power consumption of applications. Since the data processing on typical sensor applications is small, processing a large number of sensor samples can be performed without affecting application performance. Further, buffering sensor data does not affect the application's expectation. Sensor events are delivered asynchronously and include timestamp with the sensor data. Our approach is to adjust the delivery of these asynchronous events to arrive in power-efficient bursts.

2.3 Approach

While *MobileHub* can potentially reduce the power consumption, it is challenging to implement this. *MobileHub* seeks to rewrite application binary without requiring developer support. This means, the application source code is typically unavailable. However, *MobileHub* requires knowledge of how application uses the sensor data. The question is: how can *MobileHub* learn the application usage patterns without source code?

MobileHub uses *information flow tracking* to track how an application uses sensor data. Information flow tracking (or taint tracking [16] or dynamic taint analysis [12]) allows us to systematically track the information flow through a program at runtime and determine when an input produces user-perceivable outputs.

For a given application, *MobileHub* tracks sensor usage under various usage scenarios. By understanding how sensor data is used by the application under different conditions, *MobileHub* infers an appropriate buffering policy for the application. Next, *MobileHub* rewrites the application binary based on the buffering policy inferred through information flow tracking. Once re-written, the modified application starts using the sensor hub to significantly reduce its power consumption.

We designed and implemented *MobileHub* for Android OS. Android is open source software is one of the most popular OSes with over 70% market share. However, *MobileHub* is built on top of a sensor API that is similar in design to several popular mobile operating systems. Therefore, our design itself is not specific to Android.

To rewrite the binary, *MobileHub* converts the Android bytecode to an intermediate language using Soot. Soot [25] is a Java optimization framework which provides a translation bidirectionally between the Android Dalvik bytecode and the Jimple intermediate language [26].

3 Information Flow Tracking

Information flow tracking has been primarily used for security and privacy [16, 21, 27, 19].

In an information flow tracking system, there are two kinds of flows that can be tracked: explicit and implicit. Information which is explicitly passed from the right-hand to left-hand is an *explicit flow*. A simple example is *var = sensordata* which assigns the value in *sensordata* to a variable *var*. If *sensordata* is tainted (i.e., contains the information we are trying to track), then the assignment operation passes this information to *var*, so *var* also needs to be tainted. If *var* is now assigned to another variable, the taint continues to propagate. This type of tracking alone is sufficient for many usages – for example, to detect if any confidential emails are sent over unencrypted network connections.

More subtle, implicit tracking allows tainted information to be tracked through control flow. For example, *if (sensordata* $>$ *X) then*{step++;} is an implicit flow. if *sensordata* is tainted, then the variable *step* should be tainted, since the value of *step* is affected by data with a taint tag.

The *MobileHub* information flow tracking tools implements both explicit and implicit flow tracking. For explicit flow tracking, we build on TaintDroid [16], a tool designed to detect privacy leaks in mobile phones. Unfortunately, TaintDroid does not capture implicit flows; i.e., information flow through a control loop. Tainting control flow is crucial for sensing applications, because these applications often perform computation by conditioning on a sensor value. For example, for all of the sensing applications we study in our evaluation (Section 5.1), we find that explicit flow tracks only up to 13% of the information flow, while remaining information flow is passed through implicit flows.

We first describe how we adapt TaintDroid for the sensing system. We then describe how we enable tracking through control flow.

3.1 Explicit tracking using TaintDroid

TaintDroid is a privacy monitoring system designed to track private information leaks over a network. It provides variable-level tracking by associating a 32-bit taint tag with each data unit. TaintDroid instruments the Dalvik VM [1] interpreter to propagate the taint tag at each Dalvik instruction [15]. It also provides message-level tracking for interprocess process communication (IPC), which appears in the form of messages between applications.

The tracking defines one or more taint sources and sinks; the information is tainted at the source, tracked through the program, until it gets to the sink. Since our flow tracking goals are different from that of TaintDroid, we make the following changes

- The original taint sources of TaintDroid are SMS, IMEI code, Wifi status etc. Instead, we set the taint sources as the different sensor sources in the phone including accelerometer, gyroscope, magnetometer, etc. Each taint source holds a bit in the 32-bit taint tag.
- To track timeliness of data, *MobileHub* assigns a unique tag for each sensor data; for example, the 7th accelerometer reading has a different tag thanthe 8th accelerometer reading.
- As our sinks we use any user-perceivable update including: sending tainted data over the network, saving tainted data to the disk, and printing tainted data on the screen.

Table 1 shows how the taint gets propagated in *MobileHub*. Notice that we only store a single tag for a whole array and we treat the array as a primitive variable. If the array is tainted, all elements in this array are considered tainted. When the Android application contains native functions, if any input to the function is tainted, we taint the output.

3.2 *MobileHub* Implicit flow tracking

Enabling flow tracking through control flow blocks is not as straightforward as tracking via assignment and can create a taint explosion. As a result, dynamic taint-tracking systems do not support control-flow tracking, including the two taint tracking systems designed for Android [16, 13]. Our approach is to use explicit flow tracking to help with implicit flows. We identify the control flow blocks that are likely to be tainted, that we call *TaintBlock*. A TaintBlock is a control block, where the control statement is conditioned on a tainted variable. For example, a control block,

if (sensordata $>$ X){ step++; }

is a TaintBlock. *MobileHub* instruments this TaintBlock by adding a statement for each assignment operation. In this example, *MobileHub* adds a statement that associates the variable *step* with the variable *sensordata*. During runtime, explicit flow control will automatically propagate the taint from *sensordata* to *step*. If *sensordata* is not tainted, then the additional instrumentation acts as a noop.

```
 1  tag1 = σ(sensorValue1);
 2  if (sensorValue1 > T1) {
 3    avg = (sensorValue2+sensorValue3)/2;
 4    σ(avg) = σ(avg) ⊕ tag1;
 5    tag2 = tag1 ⊕ σ(avg);
 6    if (avg > T2) {
 7      stepCounter++;
 8      σ(stepCounter) = σ(stepCounter) ⊕ tag2;
 9    }
10  }
```

Figure 3: An example instrumentation of a TaintBlock.

Instrumenting TaintBlock Figure 3 shows an example of how we instrument a TaintBlock for a nested condition. The lines marked in red are instrumentation code, and the lines in black are the original code. (Note that while we are showing high level Java code for ease of understanding, our system actually operates on a lower-level intermediate language.)

The instrumentation is completely automatic. When *MobileHub* identifies that a control block is a TaintBlock, it adds the block tag before the start of the control block. In Figure 3, Lines 1 and 5 are block tags. The block tags contain the taint values of the two control variables (represented by σ). For each assignment within the TaintBlock, *MobileHub* ors the value of the variable with the block tag. Notice that if any one of the variables *sensorValue1* and *avg* are tainted, then the taint will be propagated to the variable *StepCounter* through implicit flow tracking.

Method calls within a TaintBlock are more complicated to instrument. For each method call inside the TaintBlock, *MobileHub* pushes the block tag into a global program stack before the method call. At the beginning of the callee method, *MobileHub* retrieves the tag from the global stack. Each assignment within the method is instrumented using the retrieved block tag. As before, if the block tag is empty, there will be be no effect to the assignments within the method call.

[1]Dalvik is the virtual machine that runs Android applications.

Instruction	Taint Propagation	Description
$x \leftarrow C$	$\sigma(x) \leftarrow$ Untainted	Set taint status of x to Untainted
$x \leftarrow y$	$\sigma(x) \leftarrow \sigma(y)$	Set taint status of x to status of y
$x \leftarrow y \otimes z$	$\sigma(x) \leftarrow \sigma(y) \oplus \sigma(z)$	Set taint status of x to the union status of y and z
$o.f \leftarrow x$	$\sigma(A, f) \leftarrow \sigma(x)$	Set taint status of field f to status of x
$x \leftarrow o.f$	$\sigma(x) \leftarrow \sigma(A, f)$	Set taint status of x to status of field f
$x[t] \leftarrow y$	$\sigma(x) \leftarrow \sigma(x) \oplus \sigma(y)$	Update taint status of array x with status of y
$x \leftarrow y[t]$	$\sigma(x) \leftarrow \sigma(y) \oplus \sigma(t)$	Set taint status of x to the union status of array y and index t

Table 1: Taint Propagation Logic. x, y, t refer to primitive variables. A refers to a Java class, f is a primitive field of class A, and o is an instance of class A. C refers to constant. σ represents the function that maps variables and fields to the taint tag.

3.3 Efficient instrumentation

To this point, we have described our system as dynamically tracking taints through all control blocks. This is unnecessary in many cases and inefficient since it causes increased code size and slows down execution. Fortunately many control block can be eliminated as TaintBlocks with static analysis. Since taints in our system flow from a fixed number of known sensor callbacks, we can perform a static program analysis to determine that many control blocks can never be tainted. In this way, we are able reduce the program growth of over 90% in the case of complete block tainting to around 1% using static analysis for certain applications (Section 5.1).

4 *MobileHub* design and implementation

There are two parts to the *MobileHub* design and implementation: 1. Designing APIs and policies that lets the application communicate with the sensor hub, and 2. Re-writing mobile binaries and re-architecting the OS to benefit from the sensor hub. Finally we describe the implementation of the sensor hub prototype.

4.1 Protocol, API, and Policies

Popular smartphone operating systems, including Android, iOS, and Windows Mobile all use an event-driven model to interface with the sensors. The application registers for a sensor event, and when the event occurs the application is notified using a callback mechanism. The goal of *MobileHub* is to modify the application's sensor interface so that the callbacks occur in bursts with length delays in between.

The problem is that the callback API provided by the OS is rather limited. Android provides the following API to register a callback: *registerListener(SensorEventListener listener, Sensor sensor, int rate)*, where listener is the callback function, sensor is the type of sensor, and rate is the sampling frequency. Using this API, the listener function is notified whenever any sensor data becomes available, and the sensor is sampled at a certain rate. However, the API does not provide support for buffering sensor values or using the callback to interrupt only under certain sensor events.

Application API *MobileHub* replaces the existing API with: *registerListener(SensorEventListener listener, Sensor sensor, int rate, int bufferSize, Conditions cond)*. Using this API, the application can request the sensor hub to trigger callback under two conditions: when a bufferSize number of sensor readings have been sampled, under different conditions.

Policies *MobileHub* makes a decision on an application-specific set of API parameters; i.e., how many samples to buffer and what conditions to use to initiate callback. The goal is to allow the sensor hub to buffer data as long as possible without creating a user-perceivable delay in output.

MobileHub uses information flow tracking to learn the sensor usage of applications. In this work, we evaluate *MobileHub* specifically to improve performance of mobility-tracking applications. To this end, *MobileHub* tracks application usage under two different scenarios that affect mobility tracking applications: user mobility and user's application usage. User mobility is self evident. We also track the user's application usage because mobile applications often change behavior according to whether the application is in the background or foreground; i.e., whether or not the user has currently opened the application.

The simplest policy will be to let the sensor hub buffer the smallest number of samples that triggers a user-perceivable event during information flow tracking. However, when we can correlate external events with sensor usage patterns, we can do even better. In the case of mobility-tracking applications, we know that applications use two modalities–stationary and mobile. Most sensor hubs already contain libraries that track if the phone is stationary or mobile [2]. Therefore, as an optimization, we can use two different buffering policies for when the phone is stationary and when the phone is mobile. This, of course, may not be feasible for all applications. We register different listeners and buffer sizes depending on the condition.

Similarly, a policy could depend on whether the phone is the foreground or background. However, this policy is superfluous because there is no need for a buffering when the phone is the foreground; the CPU is already active and buffering will provide no energy benefit. In Section 4.2 we describe how we design *MobileHub* to not buffer data when the CPU is active.

More generally, the set of usage scenarios that are needed to learn how an application uses sensor data depends on the type of application, the type of sensors used, and the kinds of user interactions that are possible in the application. For example, an application that only uses the accelerometer sensor is very likely to be affected by mobility, but less likely to be affected by location. Similarly, applications that use the temperature sensor is affected by varying heat conditions, but not the phone orientation, etc.

In this work, we infer simple policies for the mobility-tracking application, and show how these policies can be used to rewrite application binaries to be more power efficient. Other applications that use sensors such as GPS likely require a more complex set of policies and a larger set of usage scenarios to learn from. This is a limitation and we discuss how we can address this in Section 7.

Sensor hub - OS protocol The sensor hub protocol translates the application API into a set of actions for the sensor hub to perform. We note that our goal is not to innovate on a sensor hub design. Several sensor hub implementations already exist in the market [5, 2, 9] in the market. Since we cannot directly integrate these existing sensor hubs in *MobileHub*, we design our own sensor hub that replicate existing sensor hubs. We defer discussion of the sensor hub implementation later in this section, but here we describe the protocol between the sensor hub and the phone.

Figure 4 is part of a protocol that the OS (Figure 5) uses to translate the applications requirement. The jobID uniquely identifies each request to the sensor hub; an application can request the sensor hub to perform multiple jobs. The taskID specifies what task the sensor hub should perform: for example, buffer sensor readings, cancel the job, or to return the collected samples to the OS. When the OS requests for the samples, the sensor hub returns the samples according to the protocol marked *sensorhub_to_os*. We also design protocols to specify buffering under different conditions such as mobility.

```
struct os_to_sensorhub_{
        jobID::4;
        taskID::4; //buffer, return, cancel etc
        sensorID::4; // accelerometer, gyro, etc

        //If taskID is buffer
        persistent::1; //is sampling continuous?

        rate::6; //sampling frequency
        bufferSize::4; //num of samples to buffer
}
struct sensorhub_to_os{
        jobID::4;

        sensorID::4;
        numSamples::4
        sample1
        sample2
        ...
}
```

Figure 4: Structures used to communicate between the host and the sensor hub.

4.2 Rewriting applications

MobileHub re-architects the OS to support the sensor hub protocol, and then rewrites the Android application to use the modified API explained above.

Re-architecting the OS Android (and several popular smartphone OSes including iOS and Windows Mobile) bundle sensing tasks into a *Sensor Manager*. The Sensor Manager communicates directly with the hardware sensors[2]. Each application registers with a Sensor Manager instance, creates an event listener with the Sensor Manager instance, and listens for events. When the Sensor Manager receives sensor data from the hardware sensors, it notifies all the instances that are registered for the event. Each of these instances provide the sensor data to the application using the asynchronous callback mechanism.

Figure 5 shows how we modify the OS to leverage sensor hubs (the figure is a modified version of the Android OS stack [1]). *MobileHub* replaces the Sensor Manager with a Sensor Hub Manager. The Sensor Hub Manager communicates with the sensor hub using an appropriate communication driver.

The Power Manager provides input to the sensor hub regarding the state of the CPU and the screen; the Sensor Hub Manager implements buffering only if the screen and the CPU are inactive.

Rewriting application binary *MobileHub* rewrites the application binaries to use the Sensor Hub Manager. Rather than registering with the Sensor Manager, the rewritten application registers with the Sensor Hub Manager. The application replaces the existing API to register

[2]iOS uses a separate Sensor Manager for each sensor activity; for example, a Motion Manager, a Location Manager, etc

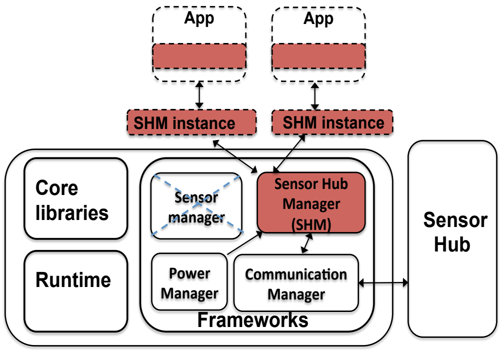


Figure 5: Android OS with sensor hub.

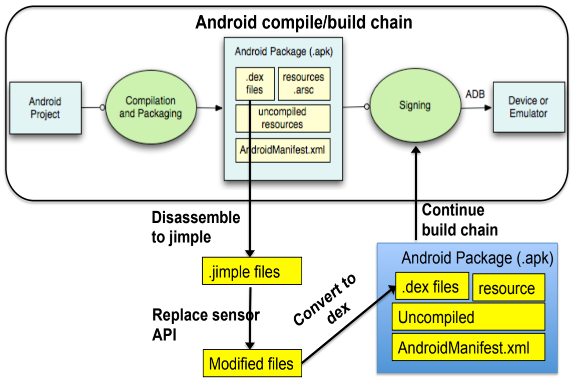


Figure 6: Bytecode translation in Android.

with the sensor hub with the API provided in Section 4.1. The modified APIs are invoked using the buffering parameters based on the inferred policies.

MobileHub rewrites the Dalvik byte code by first converting the binary into the Jimple intermediate language. *MobileHub* identifies calls to the Sensor Manager and replaces the call with the appropriate call the Sensor Hub Manager.

Figure 6 shows how *MobileHub* rewrites the binary of an application. This rewriting is used both for instrumenting the Android app for tracking, and for rewriting the application binary to use the sensor hub. In Android, the application binary is packaged as an application package file, or an *apk*; for instance, the applications downloaded from the Android Marketplace use the apk format. The apk consists of *dex*, the Dalvik executable, and other supporting resources. *MobileHub* converts the Dalvik executable to the intermediate Jimple program, modifies the intermediate program, and converts it back to *dex*. *MobileHub* then combines all the supporting resources with the new *dex* and signs the *apk*.

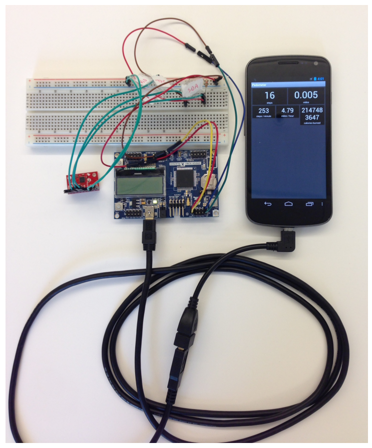


Figure 7: Sensor hub prototype.

4.3 Prototype

We prototype the sensor hub using an 8-bit Atmel AVR micro-controller, XMega-A3BU Xplained [10]. To the micro-controller, we add several sensors: a ITG3200/ADXL345 IMU (a combined tri-axis accelerometer and gyro sensor) through I^2C interconnect, an analog Electret microphone through ADC, and a 12 Channel Copernicus II GPS Receiver through UART. The micro-controller itself has in-built temperature sensor.

Commercial sensor hubs are integrated into the phone [2, 9] and communicate using internal buses. Since we are unable to integrate our sensor hub prototype into the phone, we use the common USB interface to interconnect between the sensor hub and the phone.

Figure 7 shows our prototype setup. The AVR micro-controller communicates with the USB host device by tunneling serial communication through USB [3]. (Like most modern smartphones, our Galaxy Nexus is capable of acting as a USB host when used with a special USB OTG (On The Go) cable).

5 Evaluation

Our goal is to evaluate *MobileHub* in terms of the energy benefits it can provide to mobile applications. To this end, we first study the sensor usage of seven off-the-shelf mobile-tracking applications using the *MobileHub* information flow tracking tool. We find that three of the seven applications can benefit from the sensor hub. We modify the application binaries for the three applications using *MobileHub*. Our laboratory experiments on the three modified applications show that *MobileHub* can improve power consumption by up to 83%, with no effort from the developer. Finally, we conduct a trace-driven evalua-

Name	Android playstore ID	Rate	What effects sensor usage
nWalk	levente.pedometer-1.apk	20ms	Mobility, App usage
Walking	cha.health.walking	35ms	Periodic
Pedometer	bagi.levente.pedometer	20ms	Mobility, App usage
Pedometer Pro	oodot.pedometer	20ms	Periodic
Universal	kr.still.universalpassometer	20ms	Periodic
StepCounter	Stepcounter.Step	20ms	Mobility, App usage
Simple Steps	cc.mannam.steps	20ms	App usage

Table 2: Application usage characteristics.

Name	Stationary	Mobile
nWalk	400	340
Walking	None	None
Pedometer	400	30
Pedometer Pro	5	5
Universal	20	20
StepCounter	400	350
Simple Steps	400	400

Table 3: Application policies. The table shows that number of sensor samples that can be buffered for each application under different settings.

tion to quantify the energy benefits of *MobileHub* on real users. We find under real user mobility and application usage, *MobileHub* improves energy consumption by 50% to 80% for all three applications.

We do all our energy evaluation on Galaxy Nexus phones, Android version 4.2.2. The information flow tracking experiments were done on Nexus S phone.

5.1 Tracking off-the-shelf applications

We instrument information flow tracking to track seven off-the-shelf mobility-tracking applications (shown in Table 4). All the applications are free for download from the Android Marketplace. The instrumentation is completely automatic and can be performed at scale.

We chose the mobility-tracking applications because this is the most popular class of sensor driven applications. Many commercial sensor hubs specifically target the mobility-tracking applications [2]. Further, this is the most commonly studied applications in mobile sensing power optimization studies [20, 18, 22].

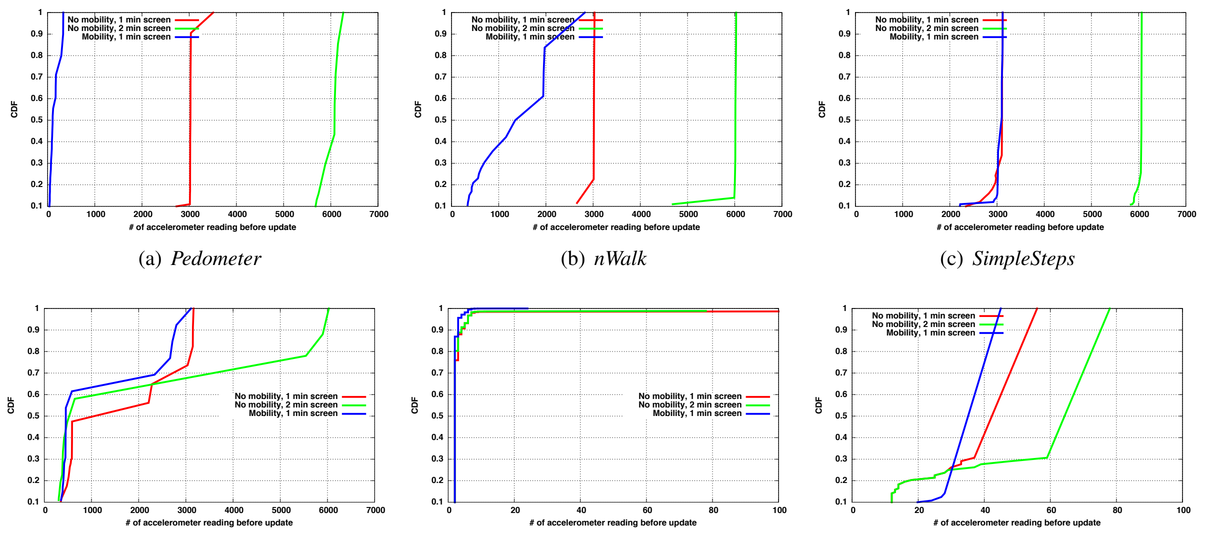
Information flow tracking meta evaluation Table 4 shows details from the instrumentation. The first column represents the total number of variables in each application. *MobileHub* marks less than 13% of the fields as tainted. Importantly, a larger fraction of the fields are tainted through the control flow tracking. The column marked *% Code added* shows the lines of code *MobileHub* adds to the intermediate language for instrumentation. The column marked *% code if inefficient* shows the lines of code we would have added if we blindly instrument every control flow loop. *MobileHub* uses static analysis to only instrument control flows that are likely to be tainted (Section 3.3). Blindly instrumenting every control flow loop can significantly blow up the program size, in some cases even more than double the original program.

Methodology As described in Section 4.1, we vary two factors that affect the behavior of mobility-tracking applications: the user's mobility and user's application usage. The user's application usage refers to how often the application is brought to the foreground by the user.

To learn sensor usage and infer buffering policies, we track the application in controlled settings. We use the following settings: *(i)* No mobility, 1 min screen: The phone is placed on a flat surface with no mobility, and the application is bought to foreground once every 1 minute for a 10 second duration. *(ii)* No mobility, 2 min screen: Similar to the previous setting, the application is bought to foreground once every 2 minutes for 20 second duration. *(iii)* Mobility, 1 min screen: The phone position varies constantly, the application is bought to foreground once every 2 minutes for 20 second duration.

These settings let us figure out if the screen time and mobility makes a difference in when sensor applications generate output. We vary the speed at which the phone moves, but do not find difference in sensor usage. However, changing the rate at which the application is bought to the foreground did change the sensor usage. We capture this in our controlled setting.

We run each of the 7 applications under the three settings for 5 runs. Each experiment lasted 15 minutes. For all the experiments, the application remain in the foreground for 10% of the total experiment time. Recall that when the application is in the foreground, or more generally, when the CPU is active, *MobileHub* does not buffer packets. As shown in Figure 5, the PowerManager lets the SensorHubManager know that the phone is in the active mode. We only infer the buffering policies for cases



(a) *Pedometer* (b) *nWalk* (c) *SimpleSteps*

(d) *StepCounter* (e) *Pedometer Pro* (f) *Universal*

Figure 8: Sensor usage across the apps.

when the CPU is not active. To facilitate repeatability, we use a *helper* application that brings a given app to the foreground at a specific time and for specific length of time.

Metric For each application, we measure the number of sensor samples that are collected until the application generates a tainted output. As before, generating an output refers to updating the screen, sending data over the network, or writing to disk. We verify that the seven applications only update the screen; they do not use the network or the disk to record outputs.

App characteristics Table 2 shows the application characteristics. The sensor sampling rate is 20ms for all applications except *Walking*. The *Walking* application uses a lower frequency of 35ms.

The last column refers to the factors that determine sensor usage. Specifically, if the application generates output based on user mobility or if the application is open.

For the apps, *nWalk*, *Pedometer*, and *StepCounter*, the sensor usage depends on both mobility and application use. On the other hand, the *Simple Steps* app only updates the screen when the application is bought to the foreground. In other words, even if the user is moving, *Simple Steps* only updates the number of steps taken by the user, when the user opens the application.

The apps *Pedometer Pro*, *Walking*, and *Universal*, are completely user agnostic. These apps generate an output after a fixed number of sensor samples, irrespective of what the user does. For example, the *Walking* app updates the screen after collecting each sensor sample. Clearly, *MobileHub* cannot benefit applications such as *Walking*.

Policies To infer the sensor hub policies, we plot the CDF of sensor usage of 6 apps over all the runs. We ignore the *Walking* app since the usage frequency is 1. Figure 8 shows how often the sampled sensor data is used by the application. The x axis is the number of sensor samples that are collected until the application generates a user-perceivable output. We ignore the sensor usage when the phone is in the foreground and only plot the data for the remaining 90% of the time.

Lets look at a single application. Figure 8(a) shows that, when the application is bought to the foreground once every minute without mobility, *Pedometer* generates an output after collecting 3000 samples. It takes 1 minute to sample 3000 sensor readings at a 20ms frequency. For *No mobility, 2 min screen* setting, *Pedometer* generates an output after around 6000 samples; i.e., after 2 minutes. In other words, when phone is not moving, *Pedometer* does not generate output unless the application is in the foreground.

On the other hand, when the user is moving, *Pedometer* generates an output much more often. Figure 9 is a magnified version of Figure 8, showing all of the applications under the mobility setting. It shows that, under mobility, *Pedometer* uses sensor data after 30 samples.

We infer the policies for *MobileHub* sensor hub from Figure 8, under idle and mobility condition. If the application only generates output when the application is bought to the foreground, then we buffer a default of 400

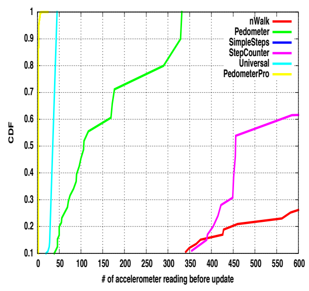


Figure 9: Sensor usage across applications under mobility.

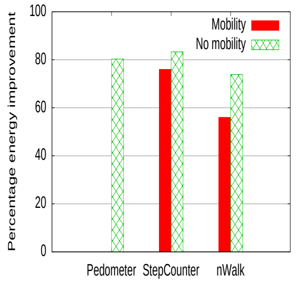


Figure 10: Energy consumption of applications.

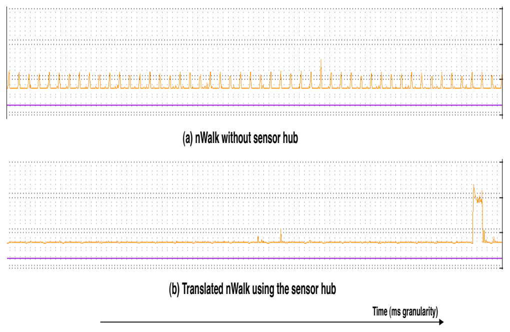


Figure 11: Snapshot of the nWalk application's energy consumption captured on the Monsoon power monitor.

Name	Total fields	Data flow taint	Control flow taint	% Code added	% code if inefficient
nWalk	506	3	28	15	69
Walking	497	7	8	0.9	95
Pedometer	304	3	27	28	81
Pedometer Pro	689	12	18	3	93.8
Universal	440	1	7	1.3	95.7
StepCounter	685	1	5	0.8	95.2
Simple Steps	125	7	11	8.6	114.8

Table 4: *MobileHub* information flow tracking meta evaluation.

samples. We choose 400 because it takes less than 200ms for the application to read the 400 buffered samples. The number of samples is a trade-off between energy savings and latency.

5.2 Quantifying *MobileHub* energy benefits

Next, we quantify the energy benefits of *MobileHub* using lab experiments. We rewrite 3 of the 7 applications we experiment with: *nWalk*, *Pedometer*, and *StepCounter*. We do not rewrite *Universal*, *Walking*, and *PedometerPro* because of the small buffering opportunity. We chose not to rewrite the *SimpleSteps* application due to its incorrect use of the system clock to determine the sensor data timestamps. (We discovered this using static analysis of the bytecodes.) This actually breaks the asynchrony assumption about sensor events, and inducing delays in reading deliveries would cause the application to infer different mobility results. We discuss this further in Section 7.

Methodology We measure the power drain with and without the *MobileHub* translation, using the Monsoon power monitor [6]. The power monitor provides accurate energy measurements by sampling the current drawn from the battery with a frequency of 5000 Hz. While a proper, integrated sensor-hub would communicate with the host using an on-die or on-chip bus or network, our prototype makes use on an external sensor-hub connected via USB using an OTG cable. Accordingly, we subtract the (non-trivial) power overhead the OTG cable adds from our energy measurements to more accurately model an integrated sensor hub.

In the lab experiments, we run each of the application in two settings: Mobile and Stationary. In the mobile setting, the phone in stationary for 30 seconds, moving for 30 seconds, and so on for a total of 2 seconds. In the stationary setting, the phone is stationary for 2 minutes. *MobileHub* uses the buffering policy from Table 3. We use a static buffering policy for the mobility and stationary scenarios.

Results Figure 10 shows the results of the lab experiments under mobility and stationary setting. Under stationary settings, the sensor reduces energy by over 80% for *Pedometer* and *StepCounter* and reduces energy consumption of the *nWalk* application by 74%. For *StepCounter*, *MobileHub* improves energy consumption by 76% and for *nWalk*, *MobileHub* improves energy consumption by 54%, because *MobileHub* buffers 350 and 340 samples respectively. The energy benefits under mobility is lower because the applications perform more computation under mobility. For example, we find that, for the *MobileHub* translated *nWalk*, computation increases linearly with buffer size. This suggests that there is a trade-off between the energy saved by buffering and the energy spent on computation; we plan to pursue this as part of future work.

On the other hand, under mobile settings, the *Pedome-*

ter application does not show reduced energy consumption. Our experiments show that buffering 30 sample or less does not provide energy benefits because of the overhead of the sensor hub. There is overhead is sending and receiving commands to and from the sensor hub that offsets the benefits of buffering, for small buffer sizes. We believe this overhead can be eliminated if the sensor hub were integrated into the phone using a faster, more efficient interconnect.

Finally, Figure 11 is a snapshot of the energy consumption of the nWalk application with and without *MobileHub*, captured by the Monsoon power monitor. The unmodified *nWalk* application continually draws power due to sensing. On the other hand, the *MobileHub* modified *nWalk* buffers packets for a long time before waking the CPU. Thus the CPU is idle for longer periods of time. We note that this figure is similar to the hypothetical example we presented in Figure 2 as motivation for this work.

5.3 Trace-driven study under real user behavior

Methodology Recall that the energy consumption of the 7 apps depends on how often users move and how often users open the application. Our goal is to quantify the energy benefits of *MobileHub* for a real mobility and application usage characteristics.

We provide phones to three users, and each phone runs the three applications, *Pedometer*, *nWalk*, and *StepCounter*. The users carry the phone around according to their daily routine.

This only captures user movement. We ask that the our users not operate the phones, but instead rely on our *helper* application to brings the application to the foreground periodically. To do this, we obtain data from the reality mining project [14] at MIT. The data set captures when users switch their screen on. We conservatively assume that the user opens the application every time she switches her phone on.

We randomly pick three users from this dataset, with high medium and low refresh rates. The mean number of screen refresh events from the data set was 23.7 times per day. We pick users with an average rate of 10, 20, and 33 screen refreshes per day. We run *MobileHub*'s information-tracking tool in the background to capture the sensor usage in the phones by running. We conduct this experiment for 3 days.

Finally, we build a model of energy consumption of the phone. We specifically model the energy consumption when each application is running and: the phone is stationary, the phone is idle, the screen is on. We build the model using the energy traces we collect using the Monsoon power monitor.

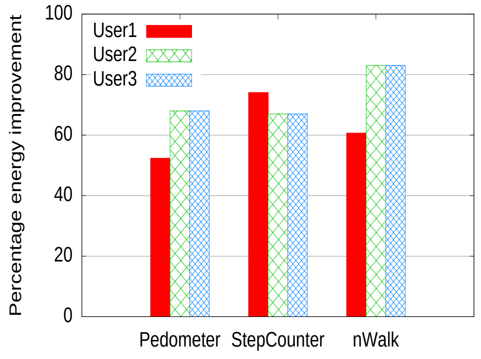


Figure 12: Trace driven simulation results.

Trace-driven study results Figure 12 shows that energy benefits for each user under the three applications. All three *MobileHub* translated applications reduce energy consumption by over 50% for all three users. Further *MobileHub* translated *StepCounter* and *nWalk* improve energy consumption by over 80% for certain users.

6 Related Work

Power efficiency is paramount for mobile devices and a variety of approaches have been applied to helping improve the efficiency of sensor-driven application.

Sensor Hubs Dedicated co-processors for sensors are a natural way to support always-on sensing efficiently and many commercial mobile processors now include sensor hubs. Priyantha *et. al* [20] 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* [22] analyze different design choices that the developer can use to partition a mobile application between the phone and the sensor hub. As we have previously discussed, making use of the sensor hub creates a trade-off in which the largest efficiency gains require the most effort by the application developer. At one extreme, the system is exposed to the developer as a heterogeneous multiprocessor platform that supports either a message passing [7], or shared memory [18] IPC model. At the other extreme, some new platforms reimplement existing system functions to make use of the sensor hub, offering limited, but application transparent performance optimizations [2]. Our system seeks the middle ground in which significant efficiency gains can be achieved without any application redesign by the developer.

Heterogeneous architectures 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. For increasing the idle time of expensive NICs, Somniliquy [11] relies on a secondary embedded controller. Similarly, Wake on wireless [23] uses a lower energy Bluetooth communication link to wake up the main network interface. Sorber *et. al* [24] use a tiered system, with a sensor node embedded in a PDA embedded in a laptop [24].

Non-centralized 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.

7 Discussion

Correct use of timestamps: Our technique optimizes applications that use an asynchronous event-based API. This asynchrony gives us the flexibility to adjust sensor sample deliveries and achieve power gains.One way in which an application might be incorrectly written would be to use the host's clock value at the time of sensor data arrival rather than the timestamp included in the sensor event. Doing so would result in the application observing out-of-date and seemingly non-uniform streams of readings. *MobileHub* is not able to modify applications that use the host's clock. We in fact saw this behavior in one of our seven motion monitoring applications.

A buffering API as a better solution?: Our system optimizes applications that use a simple sensor API in which the rate at which the sensor should be read is the only parameter. An obvious extension would be to extend the system API to include an explicit buffering parameter. This would allow applications to realize the performance gains of batch-processing readings without any information flow tracking or binary rewriting. This would, however, add to the developer's burden in that they would have to understand the nature of their application's use of sensor data. So, even in the presence of an API supporting buffering, the information flow tracking tool can be used to make automatic buffering decisions or can even be useful for developers to infer the largest safe buffer size.

Information flow tracking as a developer's tool Tracking of sensor data usage itself can be a valuable developer tool. Much as existing tools allow a developer to see how memory or processor resources are used by their application, our tool would allow developers to see how much sensor data (and of what age) is actually being used. This has the potential to reveal sensors that are being polled too often or whose data is never used in a certain application context.

Determining when and how long to monitor: Our approach relies on tracking sensor data through an application for a period of time to understand how timely its use of sensor data is. One challenge is knowing when and for how long to do the sensor flow tracking to build the application's buffering policy. In our evaluation, we knew we were executing motion-tracking applications, so we performed our information flow tracking while a test user was walking. Determining when to track sensor usage for an unknown application is hard. One possible solution would be to perform continuous, periodic observations of an application's behavior over a long period of time with the hope of observing the user in all of the meaningful usage modes. Another would be to try to explicitly measure this by looking at code coverage or path execution. None of this, however, would allow our profiles to perfectly capture the sensor usage of applications that watch for rare events.

We can largely mitigate this issue of the rare event by putting a sensible cap on the amount of sensor data we would buffer. Say perhaps, five seconds worth. This still does not work, however, for the application watching for rare and latency-critical events. In such cases, our system would induce a user-observable latency based on the buffer size.

8 Conclusions

For sensor hubs to realize their potential to efficiently run long-lived sensing tasks on smartphones, we must work out how application developers will use them. We have presented a translation-based approach that offers both convenience and power savings: *MobileHub* rewrites the bytecode of Android applications to use a simple sensor hub API, with no developer effort and without access to source code. The key to our approach is an information flow analysis of applications to learn how they use sensor values. We show how to implement the control flow tracking needed for this use case, as well as data flow tracking, by extending TaintDroid. We prototyped *MobileHub* with a sensor hub comprised of an 8-bit AVR micro-controller attached to sensors, and by extending the Android OS to use this sensor hub. Lab experiments show power savings of up to 80% in our test scenarios, and a trace-driven evaluation using real user data shows power savings of up to 60%. These results are good enough that we consider our approach to be promising, and plan to extend the sensing and computation tasks that it can recognize and automatically offload.

References

[1] Android os stack: http://www.eventmoderna.com/?p=208.

[2] Apple m7: http://en.wikipedia.org/wiki/Apple_M7.

[3] Atmel: Usb device cdc application: http://www.atmel.com/Images/doc8447.pdf.

[4] Avr datasheet: http://www.atmel.com/Images/doc8161.pdf.

[5] Intel merrifield: http://blog.laptopmag.com/intel-merrifield-smartphone-chip.

[6] Monsoon power monitor.: http://www.msoon.com/.

[7] Syslink overview: http://processors.wiki.ti.com/index.php/SysLink_Overview.

[8] Tasker for android : http://tasker.dinglisch.net/.

[9] Ti tiva: http://www.ti.com/lit/sg/spmt285a/spmt285a.pdf.

[10] Xmega-a3bu xplained: http://www.atmel.com/tools/XMEGA-A3BUXPLAINED.aspx.

[11] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: augmenting network interfaces to reduce pc energy usage. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 365–380, Berkeley, CA, USA, 2009. USENIX Association.

[12] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 3:1–3:12, New York, NY, USA, 2010. ACM.

[13] S. A. Christian Fritz, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDanie. Highly precise taint analysis for android applications. In *EC SPRIDE Technical Report TUD-CS-2013-0113*, 2013.

[14] N. Eagle and A. (Sandy) Pentland. Reality mining: sensing complex social systems. *Personal Ubiquitous Comput.*, 10(4):255–268, Mar. 2006.

[15] D. Ehringer. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 2010.

[16] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[17] N. D. Lane, T. Choudhury, A. Campbell, M. Mohammod, M. Lin, X. Yang, A. Doryab, H. Lu, S. Ali, and E. Berke. BeWell: A Smartphone Application to Monitor, Model and Promote Wellbeing. In *Pervasive Health*, May 2011.

[18] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: using low-power processors in smartphones without knowing them. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 13–24, New York, NY, USA, 2012. ACM.

[19] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[20] B. Priyantha, D. Lymberopoulos, and J. Liu. Littlerock: Enabling energy-efficient continuous sensing on mobile phones. In *IEEE Pervasive*, 2011.

[21] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, pages 135–148. IEEE Computer Society, 2006.

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

[23] E. Shih, P. Bahl, and M. J. Sinclair. 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, pages 160–171, New York, NY, USA, 2002. ACM.

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

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

[26] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[27] Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Privacy scope: A precise information flow tracking system for finding application leaks. Technical Report UCB/EECS-2009-145, EECS Department, University of California, Berkeley, Oct 2009.