ADEL: An Automatic Detector of Energy Leaks for Smartphone Applications

Lide Zhang† Mark S. Gordon† Robert P. Dick†
Z. Morley Mao† Peter Dinda⋆ Lei Yang*

†EECS Department,
University of Michigan
Ann Arbor, MI, USA
{lide,msgsss,dickrp,zmao}
@umich.edu

⋆ EECS Department,
Northwestern University
Evanston, IL, USA
pdinda@northwestern.edu

* Google Inc.
Mountain View,
CA, USA
leiyang@google.com

ABSTRACT

Energy leaks occur when applications use energy to perform useless tasks, a surprisingly common occurrence. They are particularly important for mobile applications running on smartphones due to their energy constraints. Energy leaks are difficult to detect and isolate because their negative consequences are often far removed from their causes. Few tools are available for addressing this problem. We have therefore developed ADEL (Automatic Detector of Energy Leaks). ADEL consists of taint-tracking enhancements to the Android platform. It detects and isolates energy leaks resulting from unnecessary network communication by tracing the direct and indirect use of received data to determine whether they ever affect the user. We profiled 15 applications using ADEL. In six of them, energy leaks detected by ADEL and verified by us account for approximately 57% of the energy consumed in communication. We identified four common causes of energy leaks in these applications: misinterpretation of callback API semantics, poorly designed downloading schemes, repetitive downloads, and aggressive prefetching.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design

Keywords

Energy leaks, energy bugs, mobile applications

1. INTRODUCTION

Energy is a scarce resource for smartphone users. Available energy is constrained by limits on battery size and weight, and improvements in battery technology have historically been slow. Moreover, energy demands often increase with the addition of new hardware and software features. To make matters worse, operating systems and applications frequently consume energy to perform tasks that are ultimately useless, a phenomenon we call *energy leaks*. Smartphone users commonly complain about the effects of energy leaks. For example, many iPhone users reported a sudden drop in battery life from 100 hours standby to 6 hours standby due to new energy leaks in Apple's iOS 5 [1]. Android users also experienced bad battery life on various type of devices [2].

Energy leaks are in general difficult to detect and isolate [3]. There are two main reasons for this difficulty. First, it is much more difficult to determine how much energy an application leaks than to determine how much energy it uses because some applications necessarily consume a lot of energy while wasting little, e.g., Google Maps. Second, there can be a wide range of causes for unintended energy use, e.g., incorrect API use or poor application design. As we will demonstrate in Section 7, even mature and carefully developed operating systems or applications can contain energy leaks.

An energy leak is the use of energy on activities that never directly or indirectly influence the user-observable output of the smartphone. More specifically, if removing a task from the application never directly or indirectly changes any output presented to users, the task is unnecessary and the energy consumed by it is wasted. We consider energy leaks with two sources. The first source is unambiguous programming errors, e.g., frequently accessing sensor data without using it. The second source is reliance on predictions that have the potential to be too inaccurate for their intended purposes, as can happen with applications that use aggressive prefetching. In addition to identifying unambiguous energy bugs, our work can also determine the percent of energy wasted as a result of prefetching misprediction and identify the particular prefetched data elements that are not ultimately used, thereby supporting improvements to predictors or adaptation of prefetching aggressiveness.

We now describe the design, implementation, and evaluation of ADEL (Automatic Detector of Energy Leaks). ADEL is an extension of the Android platform that tracks the information flow of network traffic through applications. We focus on detecting and isolating energy leaks in network communication for mobile applications. We choose to focus on network communication because network communication, including both Wi-Fi and 3G interface, accounts for 39% of system-wide energy consumption. More importantly, developers have a hard time optimizing for network energy efficiency, resulting in substantial energy waste [4]. By tracking all the computations that depend on each network packet, ADEL

This work was supported in part by the NSF under awards CNS-1059372 and CNS-0720691 and in part by Google.

CODES+ISSS'12, October 7–12, 2012, Tampere, Finland.

is able to determine whether or not a packet's transmission constitutes an energy leak. ADEL helps developers isolate the causes of energy leaks by showing them the arrival times and contents of unused packets. ADEL uses dynamic taint-tracking analysis to detect energy leaks. It automatically labels each data object with a tag when it is first downloaded, then follows and propagates the tag when new data objects are derived from the tainted object. Thus, system outputs can be associated with the downloaded data that influenced them. ADEL further provides developers with insight on the use of each packet, associated with the packet arrival time and its content, allowing design flaws resulting in energy waste to be more easily detected and isolated.

We evaluated the effectiveness of ADEL by using it to profile five open source applications and three closed source applications. For the open source applications, we confirmed the wasted network communication by manually checking the source code to determine whether ADEL correctly labeled each packet. For the closed source applications, we manually correlated the network packet labeled by ADEL with the displayed content.

Our ADEL-supported analysis of 15 applications revealed four categories of energy leaks.

1. Misinterpretation of callback APIs that results in wasteful downloads. For example, an application containing a separate downloading thread might not kill the thread properly, allowing the download to continue after the application exits the foreground.

2. Inefficient data refreshing behavior that ignores the application and device status. For example, a widget might be updated whether or not the display is currently on.

3. Repetitive downloads. For example, an application might download the same content every time it updates because it fails to cache downloaded content.

4. Aggressive prefetching. For example, the prefetching scheme of an application can be so aggressive that significant energy is wasted without improving the user's experience.

Eliminating energy leaks of the first three kinds reduces the network energy consumption by approximately 62.1% while reducing energy leaks in prefetching can potentially reduce the network energy consumption by 52.7%. In summary, ADEL helps identify and isolate communication energy leaks.

This work makes the following contributions.

- We provide a definition of *energy leaks*: the energy consumed by actions that never influence any output produced by a system.

- We describe ADEL, a dynamic taint-tracking infrastructure that identifies network energy leaks. ADEL tracks data originating at the network interface through applications to their eventual use or deletion. Automatically determining whether particular network data are ever used can help applications developers and users identify, isolate, and fix energy leaks. ADEL is the first system that uses taint-tracking for energy efficiency analysis.

- We used ADEL to profile 15 applications. Six of them have energy leaks that account for more than 57% of their communication energy use. By further examining these leaky applications, we identified four common causes. Our analysis may be useful for both application and platform developers.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 formally defines *energy leak* and describes the problem ADEL seeks to solve. Section 4 gives an illustrative example of ADEL's use and explains its design challenges. Section 5 describes the implementation of ADEL in detail. Section 6 explains our efforts to validate ADEL and points out its limitations. Section 7 describes our analysis of energy leaks in 15 applications and indicates major classes of design characteristics resulting in energy leaks. Section 8 concludes the paper.

2. RELATED WORK

We use dynamic taint analysis to identify energy leaks due to network communication. This section summarizes prior work on the following related topics: energy debugging, network energy reduction, and taint analysis.

Energy debugging is an emerging research area, with few prior publications. The work by Pathak et al. [3, 5] is closest to ours. They first provide a taxonomy of energy bugs based on posts to mobile user forums and operating system bug repositories. They recognize four classes of energy bugs: hardware-related, software-related, external-condition-triggered, and those of unknown cause [3]. They also provide an automatic method to detect no-sleep bugs, which arise from mishandling power control APIs by applications or the operating system, resulting in significant and unexpected energy drainage [5]. Our work differs by detecting and isolating energy leaks, a different class of energy bugs.

There has been prior research on reducing energy consumption due to network communication. This work can be divided into two major categories: (1) adapting power management schemes to network traffic [6, 7, 8] and (2) adapting network traffic to power management schemes [9, 10, 11, 4]. In the first category, Krashinsky and Balakrishnan [6] propose to adapt network interface sleep durations depending on past application network activity. This allows the network interface to sleep for longer periods of time when there is no activity, thereby reducing the energy consumed receiving beacons sent from the access point. In the second category, Armstrong et al. [9] propose to shift polling responsibility from mobile clients to a proxy server so that network traffic can be batched. This approach prevents the network card from frequently transitioning among power states, allowing it to sleep longer. However, most prior work assumes that all transmitted data are useful. We question this assumption and detect network energy waste due to transmission of useless data. Existing network energy consumption optimization techniques can be applied together with our work: they are orthogonal or synergistic.

Our energy leak detections and isolation framework uses dynamic taint analysis. In particular, we build upon TaintDroid [12], an extended Android platform that supports system-wide taint tracking through the Dalvik Virtual Machine and persistent storage, e.g., in files. TaintDroid detects security flaws in mobile applications by associating tags with sensitive information, e.g., location information or phone contacts. Information leaks are detected when sensitive information leaves the mobile devices, e.g., via the network interface. Our work differs by solving the problem of identifying communication energy leaks. This new problem definition requires changes in the representation of tags and method of tracking information flow (from network interfaces to eventual display or deletion). To the best of our knowledge, this is the first time taint tracking analysis has been used to identify energy leaks.

There are other methods of dynamic taint analysis in virtual machine and interpreter environments [13, 14, 15]. Haldar et al. [13] instrumented the Java String class with taint tracking. Xu et al. [14] instrumented the PHP interpreter source code for dynamic information tracking. Both of these papers have the goal of preventing SQL injection attacks. Chandra and Franz [15] proposed to instrument Java byte-code to aid control flow analysis in additional to fine-grained information flow tracking within the Java virtual machine. These papers all solve security problems. They do not focus on improving energy efficiency, but their implementations are related because all use taint tracking.

3. PROBLEM DEFINITION

We define an *energy leak* as *energy consumed that never influences the outputs of a computer system, e.g., through display interface, audio interface, or network interface*. In other words, any energy consumed by actions that never change system outputs, directly or indirectly, is wasted. Eliminating such waste cannot cause observable changes in application behavior. For example, the widget of a news application might frequently update the latest news regardless of whether it is currently visible to the users, e.g., whether the display is even on or off. From the user's perspective, these downloads are useless and waste energy. Eliminating these extra downloads (e.g., by only downloading when the widget is visible to users) will not degrade the user's experience. Prefetching is more subtle, and we will address it in Section 7.4.

Automatic Detector of Energy Leaks (ADEL) system detects smartphone application energy leaks due to network downloads by tracking network data in order to determine which downloads ever have user-observable effects. We focus on network downloads for two main reasons. First, network devices, including 802.11 card and cellular devices, are power-hungry; the two devices together consume 39% of smartphone. Among the network transmissions, 70% of energy is due to downloads. This is estimated by aggregating 137 users' over a month of traces. The traces were generated by PowerTutor [16], an Android-based power estimation tool that estimates power consumption every 1 second. Second, application developers have an especially difficult time optimizing applications for network energy efficiency due to the complicated power management characteristics of the network interface [4]. As a result, a significant amount of energy is wasted in network communication [4].

The definition of *energy leak* is general. In practice, it is difficult to track accurately, as we explain in Section 6.1. However, we believe it is an appropriate goal to proceed toward.

4. ILLUSTRATIVE EXAMPLE AND DESIGN CHALLENGES

We designed and implemented a system supporting on-line analysis of transitive use of downloaded data objects. We anticipate that this work will be most useful to developers attempting to improve the energy efficiency of their applications. In this section, we will first use a simple example to explain the use of our infrastructure. We will then discuss design challenges.

4.1 Illustrative example

Consider a game that downloads crossword puzzles and displays them to users. As shown in Figure 1, the game has two major threads: one downloading puzzles from the Internet and one managing the user interface (UI), including showing the puzzle and taking user inputs. The downloading thread receives network packets that contain the grid layout for each game, the hints to each word, and the solution to each game. It then parses the contents of different packets and passes them as messages to the UI thread. The UI thread first displays the grid based on the messages from the downloading thread. Then it displays hints when the user clicks on the corresponding grid element. When the user finishes and submits the game, it shows the solution.

To track the use of the downloaded data objects in the crosswords game, the system first identifies them in the network interface where a *tag* is associated with each object at the packet level. These network interfaces are *taint sources* and the downloaded objects are *tainted*. As shown in Figure 1, each type of data object is associated with a unique *tag*. During the propagation of the *tainted* object, the system passes on a *tag* when one data object is derived from a *tainted* one. The shaded area represents the functions the *tainted* objects pass through. Finally when tainted objects reach the *taint sink* (e.g., the display) they are detected and the *tags* representing the original downloaded data objects are recorded.

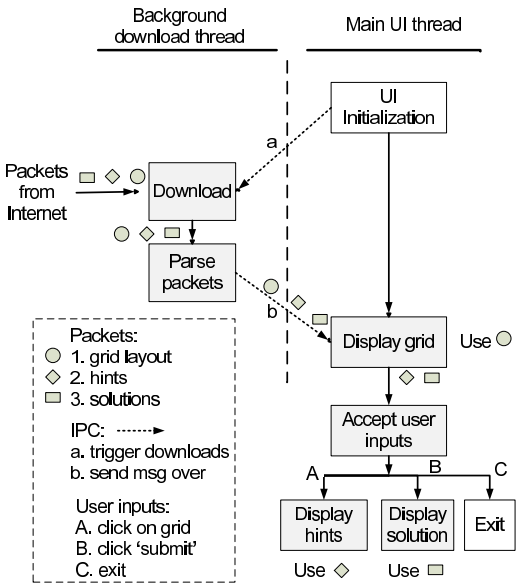

Figure 1: Crosswords game has two major threads: downloading and user interface. Tags of packets are propagated to derived data objects. Eventual use of packets depends on user inputs.

4.2 Design challenges

When designing a system to track the use of data objects, we were faced with the following challenges.

- *Downloaded data objects are not necessarily used immediately.* In the above example, the hints and solution of each game may be stored in memory for a long time before the user requires them. This delay between downloading and use prevents some alternative approaches. For example, tracking data use by first monitoring both the network and display interface simultaneously, then comparing downloaded content and displayed content is infeasible due to the time lag. As a result, the monitoring system must be able to trace data objects through memory until they are used or deleted.

- *Data objects can be parsed or processed before they are displayed.* For example, the grid color can be specified using strings in an XML file. Although the strings are not displayed, they will influence the color of the output grid. Transitive use of downloaded data must be considered.

- *The use of data objects depends on the dynamic environment of the application.* For example, a valid code path that leads to the display of the downloaded data objects might rarely or never be executed during a particular run of the application. As a result, static analysis of application source code is insufficient to determine the application's run-time use of objects. Therefore, we focus on dynamic analysis.

- *The use of data objects depends on the dynamic environment of the application, e.g., user behavior.* As shown in Figure 1, the use of hints and solutions to the game depend on user behavior. As a result, static analysis of application source code is insufficient to fully understand the application's real-world data use.

- *It is necessary to identify the original data object when a derivative is displayed.* Our goal is to help developers to (1) understand the use of each downloaded data object and (2) identify

potential bugs in their code that produce unnecessary data transmission. Therefore, it is necessary to trace downloaded data objects and their derivatives for their entire lifespans so that if they are ultimately displayed, the original downloaded data objects can be identified. To achieve this, the system needs to associate a unique ID with each data object and propagate this ID to all its transitive derivatives.

- *Mobile devices are resource constrained.* Limits on smartphone CPU speed, memory size, and energy capacity makes high-overhead techniques such as instruction-level taint tracking [17] less practical than lower-overhead techniques.

5. SYSTEM ARCHITECTURE OF ADEL

The challenges described in the previous section motivate us to use dynamic taint tracking analysis of downloaded data objects. The closest existing implementation we found is TaintDroid [12], an extended Android platform for improving security by identifying the flow of private information. This section first presents background information on Android that is necessary to understand ADEL, our taint tracking infrastructure. It then provides an overview of the ADEL system architecture. Finally, we explain each component of ADEL in detail.

5.1 Background: Android

Android is a Linux-based operating system for mobile devices such as smartphone and tablets developed by Google. It differs from other Linux distributions due to an additional layer of abstraction between user applications and Linux system library, called the application framework layer. The four major differences between Android and other Linux distributions follow.

Dalvik virtual machine, Java Native Interface (JNI), and native interface: Android is composed of a modified Linux kernel written and compiled to native machine code, with middleware, libraries, and APIs written in both Java and native code. Although development of applications using native machine code is allowed, most Android applications are written in Java.

Each Java application is compiled to bytecode and runs in the Dalvik virtual machine (VM) and each Dalvik VM instance contains only one application. The Dalvik VM has its own instruction set. An interpreter converts Java bytecode into Dalvik code at run time. ADEL consists of taint tracking code inserted into the interpreter.

There are two types of native methods that can be accessed by Android applications: (1) native methods in the Android framework or system library such as WebKit or SQLite and (2) native methods stored in shared objects in the application package. All native methods accessed from a Java-based application must be called through JNI methods. Application packages containing native methods account for 5% of applications according to a survey of the Android market [12] and handling taint tracking through native code would substantially complicate the taint tracking infrastructure. Therefore, ADEL does not cover application-specific native methods in the taint tracking flow. Note, however, that it can do information flow tracking in the Java portions of applications that contain JNI methods as explained in Section 5.3.

Binder: Binder is a specially customized kernel mechanism for Android to do Inter Process Communication (IPC). In the core of binder, messages are passed between processes using *parcel*, a container of serialized data objects.

Shared memory: A shared memory region in the Android framework allows sharing between different processes. Note that reference counting in the shared memory region to prevent deletion of live objects.

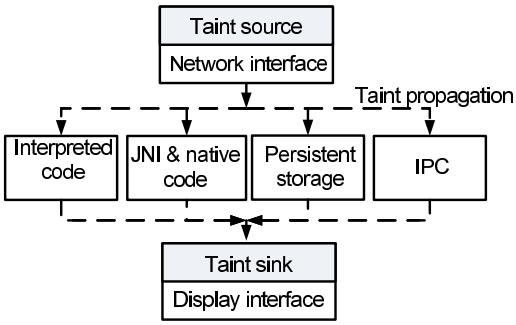

Figure 2: ADEL architecture.

Zygote process: Zygote is the first application process loaded by the Dalvik VM when the system boots. It maintains memory regions that are shared by multiple applications and spawns all other Android applications via fork().

5.2 Architecture overview

We now provide an overview of the ADEL system architecture. ADEL provides on-line monitoring of the use of downloaded objects by applications. It uses dynamic taint tracking to follow the information flow during the entire lifespan of the downloaded object, from download to use or deletion. As shown in Figure 2, the taint flow can be decomposed into three phases: *taint source*, *taint propagation*, and *taint sink*.

Taint source: The network interface is a *taint source*. We detect all downloaded data objects and associate a unique *tag* with each at packet-level. Each *tag* is a 32-bit integer that indirectly indicates (indexes) the size of the packet. Note that packet size is needed to determine the number of bytes of network transmission associated with energy leaks. A hash table is used to map tags to packet sizes. The table is kept in shared memory region and maintained by Zygote. To store tags, we instrument the Dalvik VM to change the memory allocation and Java class data structures in order to store an indexing tag adjacent to its corresponding data object.

Taint propagation: During *taint propagation*, tags are propagated to new variables when they are derived from tainted variables. Taint propagation happens at variable-level, method-level, file-level, and message-level. These levels correspond to four sources for which tags are propagated: interpreted code, JNI and native code, persistent storage, and IPC. For interpreted code, an alternative design is instruction-level taint tracking [17]. However, this fine-grained approach imposes up to 20× performance overhead for workstations: it is contradicted for resource-constrained mobile platforms. For JNI and native code, the system retrieves the tags of the inputs to invoked methods and propagates them to the return value. This heuristic is used to prevent the overhead of monitoring native code. For persistent storage, each file is associated with a tag. Any write of the tainted data will taint the tag of the file and any data read from the file will be tainted with the same tag. Finally for IPC propagation, we associate a tag with the message passed between processes. File-level and message-level propagation permits false positive. By propagating the tag at multiple levels, we are able to achieve system-wide taint tracking.

Taint sink: At the *taint sink*, the output of tainted data is identified. Every time a tainted object is detected, the original downloaded object(s) that influenced the tainted object is logged, based on its tag. Mobile computers may have multiple output device, e.g., video and audio displays. Currently ADEL only supports the video display interface taint sink, as it was used for displaying the most network-tainted data in the applications we are aware of. However, the approach could be easily extended to handle other output

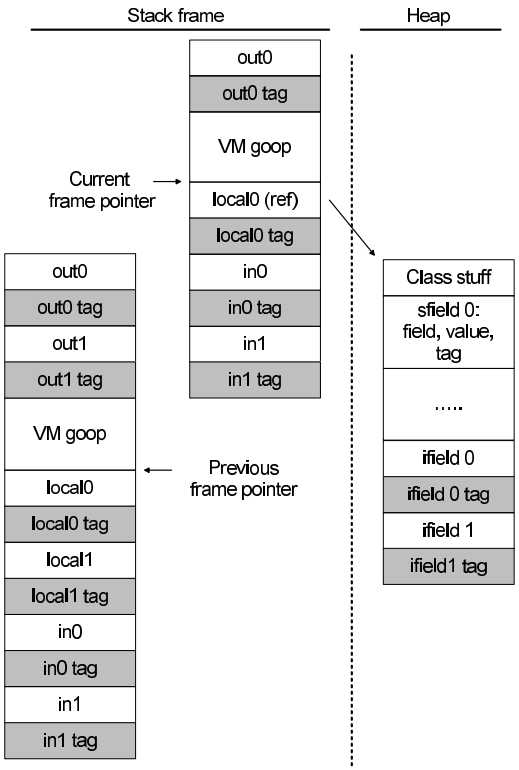

Figure 3: Taint tags are stored adjacent to their data objects in both stack and heap.

devices. For example, the audio interface can be added by instrumenting the audio API in the Java framework.

5.3 Implementation

We implemented ADEL on top of TaintDroid [12] with 2,414 lines of code modifications. The adaptation of this infrastructure for use in energy leak detection required changes to taint tag representation, taint propagation rules, taint source, and taint sink.

Tag storage

Figure 3 shows how ADEL stores taint tags in the Dalvik VM. Tags are associated with five types of data: local variables in method, method arguments, method return value, class static fields, and class instance fields. Whenever a new method is invoked, a new stack frame containing local variables, method arguments, and return values is constructed. To store their tags, the size of the stack frame is doubled. This allows us to store an additional 32-bit tag for each data object. Tags must be retrieved whenever the corresponding data object is operated upon. Therefore, they are stored adjacent to their corresponding data objects. As shown in Figure 3, the shaded area represents the added memory for tags. For class static fields and instance fields stored in the heap, additional memory is used. A taint tag field is added for static fields while each instance field is extended by adding a 32-bit tag space adjacent to it. In the case of arrays and strings, only one tag is associated with the whole class object. That is to say, any value coming out of a tainted array or string is tainted. This decision was made to avoid the overhead of having multiple tags, each associated with one data element.

Tag representation and propagation logic

In order to determine the presence and causes of energy leaks, we need to track the flow of each network packet and record the contribution of tainted variables to each newly derived variable. In this case, ADEL can work backwards to determine whether each packet ultimately influences displayed content. One common solution in security research is to use each bit to represent a unique type of tainted variable and use a bitwise OR operation when merging variables. Unfortunately, this solution is infeasible for our application because it requires one bit for each new tainted variables. Recall that each packet represents a new tainted variable. In ADEL, each tag represents a set of packets that have been used to derive the corresponding variable as shown in Figure 4. Each packet is represented by a unique ID, packet_tag. Every time a new variable is derived from multiple tainted variables, the union of all sets is taken to derive the new tag.

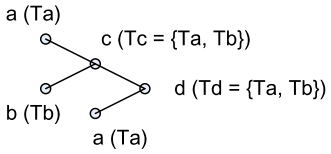

Figure 4: Example of correct taint tag propagation.

Algorithm 1 Copy taint tag

Require: *SrcTag*
1: Increase the ref_count of *SrcTag*
2: *DestTag* ⇐ *SrcTag*
3: **return** *DestTag*

We use two hash tables in ADEL. One maps packet_tag, the packet ID, to packet size. The other maps tag to the set of packet_tag as shown below.

```
map<int packet_tag, int size> packet_map;
map<int tag, set<int packet_tag>> tag_map;
```

A new tag is generated if any of the following conditions holds: (1) a new data object is downloaded or (2) a new variable is derived from more than one tainted variable. The allocation of a new tag is controlled by a unique sequence number. Note that these two maps are global data structures containing all the tags for any application. Consequently, we place them in shared memory and modify Zygote to insert new tags and propagate existing tags.

Taint tags are propagated using the following rules.

- Copy taint tag: When only one tainted object is used to derive a new variable, e.g., move and single-input instructions, we copy the taint tag to the newly derived variable so that both variables have identical sets of packet_tags. Meanwhile, each tag is associated with a reference counter to record the number of variables associated with that tag. That reference counting is needed for copy-on-write when merging taint tags. This algorithm is shown in Algorithm 1.

- Merge taint tags: When more than one tainted object is used to derive a new variable, we must take the union of the sets. This is required for any two-input instructions, e.g., add. As shown in Algorithm 2, copy-on-write is used to reduce memory overhead. A new tag and set are only constructed if both of the source variable's tag have multiple references.

Tag sources and sinks

To determine the percentage of energy leaked, ADEL needs to track the network packet from downloaded until display. Therefore, we identify the network interface as the taint source and video display interface as the taint sink. Because taint tags are stored only in the interpreted code as explained in Figure 5.3, we instrument the Java side of the library instead of the native side of the library.

We instrument the receive function in the network Java library to implement network taint sources. The granularity of tag association determines the trade-off between tracking accuracy and memory overhead. Packet-level tracking is chosen over byte-level tracking

Algorithm 2 Merge taint tag

Require: *Src1Tag*, *Src2Tag*, *tag_map*
{Update an old tag or merge to create a new one}
if any of *Src1Tag* or *Src2Tag* has only one reference count **then**
  DestTag ⇐ the one in *Src1Tag* and *Src2Tag* that has one tag
  update the set of *DestTag* by inserting the set of the other source tag
  increase the ref_count of $DestTag$
else
  DestTag ⇐ a newly created tag
  new_set ⇐ union of sets of both *Src1Tag* and *Src2Tag*
  insert (*DestTag*, *new_set*) into *tag_map*
  initialize ref_count of *DestTag*
end if
return *DestTag*

to control overhead. Fortunately, despite of the inaccuracy induced by this granularity, ADEL still identifies numerous energy leaks in real applications as explained in Section 7.

The display interface is instrumented as the taint sink, i.e., tainted objects are logged when they are rendered to the display. We determine which network packets are useful by tracking the tainted objects backwards through packet_tags. It is challenging to identify all tainted object accurately because Android provides application developers multiple ways to render the display. By performing a survey of the network-intensive applications in the Android Market, we determined that the Canvas class in graphic interface and LoadListener in Webkit are the two most frequent APIs for eventual rendering use by other graphics display APIs. We therefore use these two APIs as taint sinks in ADEL. Note that WebKit is a complicated rendering engine in which data objects are passed between native methods and Java methods before display. Our Dalvik-based taint tracking framework cannot trace through native methods. We instead use the following heuristic to approximate taint tracking through native graphics display routines: data objects being passed to the native WebKit rendering engine will be displayed. We have not found contradictory examples in the applications we examined; please note that perfect accuracy is not necessary to achieve our goal of identifying and isolating energy leaks.

Tag propagation

Taint tags are propagated through four paths: interpreted code, JNI and native code, persistent storage and IPC.

To track tags through interpreted code, we modify the native code interpreter in the Dalvik VM to realize the propagation logic. The native code interpreter is one of two interpreters in the VM. We modified the native code interpreter because it can be used on any Android system supporting native code. An alternative would have been to modify the assembly code interpreter. This interpreter has the advantage of higher performance but the greater disadvantage of only being available on a few phone models.

JNI and native code taint tag propagation is handled by manually instrumenting methods that are necessary for APIs in taint sink and source or selected benchmark applications. This is mainly because all taint tags are stored in the interpreted code stack and therefore once a native method is invoked, the tags cannot be accessed any more. To handle taint propagation through native code, we instrument code to guarantee one postcondition: if the arguments accessed are tainted, the return value of native methods is associated with the new tag that follows the propagation logic. 1098 out of 3024 native methods in Android source code, which do not use object references, are covered by this heuristic. Other native methods are instrumented as needed.

Persistent storage taint tag propagation is handled by associating each file with one taint tag. This is achieved by using the extended attribute of the Android file system. That is to say, any byte read from a tainted file will be tainted with the same tag. This results in false positives, especially if the file is big, e.g., database. Ideally we would like byte-level filesystem taint propagation logic. However, this would greatly increase implementation complexity or require modification of database implementation [18]. This is an area for future work. Despite the false positives resulting from course-grained file taint tracking, the impact on accuracy was negligible for the applications we studied: only 0.26% of network traffic ended up in files such as database in these applications.

Message-level taint tag propagation is implemented for IPC in ADEL. This is achieved by instrumenting the *parcel* in binder as explained in Section 5.1.

6. VALIDATION

We next examine the accuracy of ADEL and the overhead incurred by ADEL. The experiment is done on a Nexus One phone running Android 2.1 and ADEL.

6.1 Accuracy and limitations

We examine the accuracy of ADEL by evaluating 8 applications, 5 of which are open-source and three of which are closed-source. Table 1 lists all the application names and corresponding functionalities. Note that we include three closed-source applications for validation because we found a limited number of open-source Android applications. By analyzing each application with ADEL, we produced profiles of all network packets. This profile contained each packet's content, its arrival time, and whether the packet ultimately influenced outputs to the user. Packets (transitively) reaching the taint sink were flagged as used while the rest were considered unused. We read the source code of the open-source applications, then manually examined all the network packets and used our understanding of the code to determine whether ADEL correctly labeled each. Analysis of closed-source applications required less direct confirmation. We compared the profile generated by ADEL with the content observed on the display. For example, we manually compared the content of each network packet with the news stories displayed on the user interface for ABC news application. We identified the following types of the packet classification errors.

- **False negatives** are useful network transmissions that are incorrectly identified as useless. ADEL tracks data flow, not control flow. This is mainly because control flow tracking either requires static analysis [19] of application source code, which would undermine on-line use, or incurs heavy overhead [17]. One example of a false positive due to neglecting control flow follows. Many applications contain some empty HTTP responses that contain only headers and codes, e.g., "200 OK". These responses are indeed useful, e.g., the status code determines which branch to execute. However, because the influence on application behavior is a result of control flow, ADEL neglects it. Luckily, the size of such packets is usually very small, resulting in only minor errors in network energy leaks. In the applications we study in Section 7, these empty HTTP responses account for only 0.52% of the total traffic.

- **False positives** are useless network transmissions that are falsely identified as used. Another limitation of ADEL is the false positives induced during tag propagation, including packet-level, array-level, message-level, and file-level tag propagation. One example we found in our experiment results from packet-level association. For applications in which the data unit is large, e.g., a whole news article for news application, using packet-level association does not influence accuracy. However, in applications like "Busmap", each data unit contains the text information for one bus

Table 1: Applications for Validation

Application			Functionality
Open source	Real-world apps	Crossword	A game application that downloads crossword from multiple online sources
		Read for speed	An application that downloads book and display it word by word for speed reading
		BusMap	A map-based application that shows the incoming buses for all bus stops
		Photostream	A sample application that downloads and displays picture from Flikr
	Syn. app	Taint test	A self-written application that downloads and displays specific webpages
Closed source	ABC, Stock quote, and CNN		Refer to the Android Market

Table 2: Performance Overhead

Benchmark	ADEL (ms)	Android (ms)	Slow down (×)
Download	1285	1055	1.21
Inter. code	43	4	10.75
IPC	15	4	3.25
File	233	139	1.67
Read	204	503	2.46
Xwords	442	4565	10
P stream	641	6091	9.5
BusMap	2083	8745	4.19

stop and 2 to 3 data units are packed in the same packet. As a result, information for unused bus stops is falsely identified as used if any bus stop in the packet was used. These false positives could potentially be reduced by fine-grained propagation at the cost of performance overhead.

We should comment at this point that unlike the use of taint tracking in security applications, misclassification of some network traffic does not necessary prevent ADEL from fulfilling its design objective: to help application developers identify and isolate programming and design flaws accounting for most energy leaks. ADEL allowed the identification and isolation of numerous large energy leaks in real-world energy leaks as described in Section 7.

6.2 Overhead analysis

The performance overhead of ADEL results from additional computation and storage necessary for tracking taint tags. We have two goals when designing the experiments. First, we want to understand the average slowdown when ADEL is used on real-world applications. Second, we want to examine all taint propagation paths to understand their overheads, allowing us to determine what sorts of slow-downs can be expected for applications with particular behaviors. To achieve these goals, we conduct two sets of experiments: one with real-world applications and one with synthetic applications that stress particular propagation paths.

We use two Nexus one phones, one of which runs the stock Android 2.1 operating system, and one of which is equipped with ADEL. The portable interpreter was used on both phones (recall that ADEL is only implemented in the portable interpreter for the Dalvik VM).

Real-world applications: As shown in Table 2(down), ADEL incurs 6.14× slow down on average for real-world applications. We notice that for both application "Cross words" and "Photo stream", where tainted objects are intensively propagated between download and display, ADEL incurs roughly 10× overhead. For applications like "Read for speed" and "Busmap" where tainted objects are displayed immediately after being downloaded, ADEL incurs less than 4× slow down. Note that the past works on taint tracking generally reports significant overhead, e.g., 20× [17] slowdown. Despite the slowdown, ADEL adds at most 8 seconds to the time for real-world applications to load. We consider this to be a tolerable but annoying latency because ADEL is designed mainly for use by application developers during testing.

Synthetic applications: To cover all possible taint propagation paths, we construct synthetic benchmarks to intensively exercise specific paths in the taint flow. For example, intensive downloading exercises operations in the taint source. Intensive tag propagations through interpreted code, inter-process communication (IPC), and files are also exercised. Table 2(up) demonstrates the performance overhead of ADEL for these synthetic applications. Intensive tag propagation through interpreted code incurs the most overhead (10.75× slowdown). This benchmark approaches the worst-case situation: every instruction involves tag merging. This also explains the slowdown of "Cross words" and "Photo stream". There are two main causes of the overhead: context switches due to IPC and lock acquisition due to shared memory access. Both of these result from storing tags in a shared memory region and managing them with Zygote as explained in Section 5. The slowdown of all other benchmarks is less than 5×.

Table 3: Applications Studied

Category	Applications
News and Weather	CNN, ABC, World News, News and Weather
Transportation	Google Maps, BusMap
Game	Crossword, Read for Speed
Social	Facebook
Photography	Photostream
Sports	ESPN Score Center
Finance	Stock Quote
Widgets	ABC widget, Facebook widget
Other	Taint test

The overhead imposed by ADEL does not contradict its use in helping developers find energy inefficiencies at design time; its intended purpose. One major concern about overhead is that changes in performance might change the application behavior and hence prevent us from finding energy leaks. For example, an application that uses a timer to control downloads might potentially have different behavior if latencies were to change. This concern is legitimate. However, we have not encountered such cases in our study.

7. APPLICATION STUDY

This section describes ADEL-assisted study of network energy efficiency for 15 Android applications. Seven of these applications are suspected to contain substantial energy leaks and we were able to manually verify the presence of significant ADEL-detected leaks in six of these. For the leaky applications, approximately 57% of network energy consumption was the result of energy leaks. We explain four root causes for the detected energy leaks. Our findings illustrate the value of ADEL and reveal opportunities for improving application and Android framework API energy efficiency.

7.1 Experimental setup

We study the 15 applications summarized in Table 3. These applications are chosen based on two criteria: (1) there is intensive network activity and (2) there is limited database usage in the application. We use the second criteria because database use results in false positives as mentioned in Section 6. The closed-source applications are popular applications with more than 100,000 downloads in the Android Market.

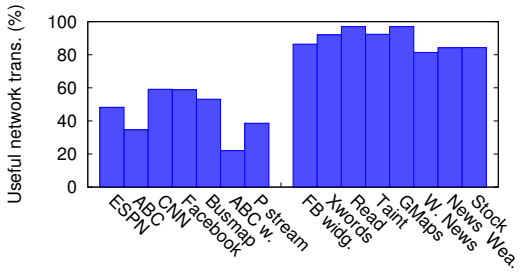

Figure 5: Percent of useful network transmissions for applications separated by efficiencies, for leaky (left) and efficient (right) applications.

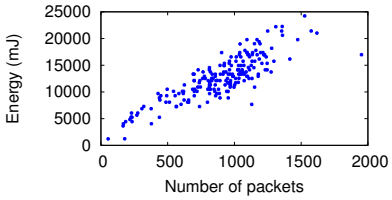

Figure 6: Correlation between network energy and transmission.

We used ADEL to identify (and isolate) network energy leaks in each application. We are aware of two possible sources of bias in our experiments.

- *Whether an application generates energy leaks is dependent on the application's active session length.* For example, an application may download a significant amount of content right after it is launched. These downloads are energy leaks if the user exits the application immediately after launch. However, they can be useful if application runs longer. To eliminate this bias, we emulate an average application session by spending more than 250 seconds in each application before exit. 250 seconds is reported to be the average application session duration [20]. During the application session, we manually exercise application functions. One exception for typical use cases is widget applications. Widgets typically stay active without exiting. We therefore attempt to approximate typical use of two widget applications by placing them on the home screen, reading updates from them every five to ten minutes, and monitoring their activities for 30 minutes for each application.

- *The propensity of an application to leak energy might depend on whether it is in its initialization or normal use phase.* Imagine a news application that downloads latest articles during the first time it is used during a day and stores them for later use. During a single session, a user might not read all the articles, making the unused articles appear useless. However, the user might ultimately read these articles in a later session. As a result, analysis of the first session alone might result in overestimation of energy leaks. To solve this problem, we run each application three times, with a five-hour intervals between runs.

To understand the root causes of the suspected energy leaks, first we examine the network packet usage profile generated by ADEL for each leaky application. The profile contains each network packet's taint tag, packet content, packet arrival time, and whether the packet ultimately influences displayed content. By doing this, we are able to identify suspected energy leak causes. We then manually examine application source code, if available, to confirm our suspicions.

7.2 Findings

Figure 5 shows the percentage of *useful network transmissions* for each application, i.e., packets that ultimately influence the content displayed. We derive this ratio for each application by averaging traces from three runs. A higher percent of useful transmissions suggests a more efficient design. As shown in Figure 5, the applications with lower bars on the left are suspected to have more serious energy leaks. Interestingly, there is a clear division between the suspected leaky applications and efficient applications at 60%.

Table 4: Root Causes for Leaks

Causes		Applications
Design flaws	Misinterp. API	Photo Stream
	Download scheme	BusMap, and ABC widget
	Repetitive download	ABC widget
Aggressive prefetching		ESPN, ABC, CNN, and Facebook

To determine the amount of energy saved by eliminating these energy leaks, we must understand the relationship between energy and total number of packets transmitted. Figure 6 shows the relationship between network energy and total number of packets for one phone user. The correlation coefficient between energy and number of packets is 0.79. We obtained this data by analyzing a month of network communication quantity and energy consumption logs for 51 Nexus One phone users [16]. Note that these energy use logs capture the real-world impact of time-dependent wireless interface power management state changes. Given this correlation, we claim that reducing the energy leaks detected by ADEL will save approximately 56.5% of the energy used in network communications for the detected leaky applications.

By exploring the network packet usage profile and source code, we identify four major reasons for energy leaks: (1) misinterpretation of callback API semantics, (2) poorly designed downloading schemes, (3) repetitive downloads, and (4) aggressive prefetching. We associate applications with their causes of energy leaks in Table 4. Note that a single application may have multiple types of energy leaks. We categorize the first three causes as design flaws and the last as aggressive prefetching.

7.3 Application design flaws

This subsection, expands on the three types of unambiguous design or programming errors that resulted in network energy leaks. Section 7.4 will deal with energy leaks caused by aggressive predictive prefetching.

Misinterpretation of callback API semantics: We find that the application "Photostream" incorrectly uses APIs, leading to substantial network energy leaks. "Photostream" is a sample application written by Google that intends to show beginner developers how to use specific APIs. The application allows users to retrieve other user's pictures from Flickr. As shown in Figure 5, only 38.6% of network transmissions ultimately influence the display.

Investigating the content and timing of unused network packets reveals that the "PhotoStream" application downloads many unused packets even when the application is put into the background. By exploring the source code, we confirm that the background downloading thread is stopped in *onDestroy()*, a callback method that is not guaranteed to be called after the application is put into background. As a result, the application continues downloading objects that cannot ever be displayed, resulting in network energy leaks.

This bug can influence many applications due to misunderstanding of the documentation. Android SDK clearly suggests that developers clean up threads in the *onDestroy()* callback method before the application is about to be killed. This suggestion is reasonable for handling threads in general because it reduces the cost of initializing a new thread when the application is resumed. However, in this particular case, it produces energy leaks because the thread continues downloading (useless) data. To make things worse, the source code of "PhotoStream" was provided as an example application for use by beginner Android developers, potentially prop-

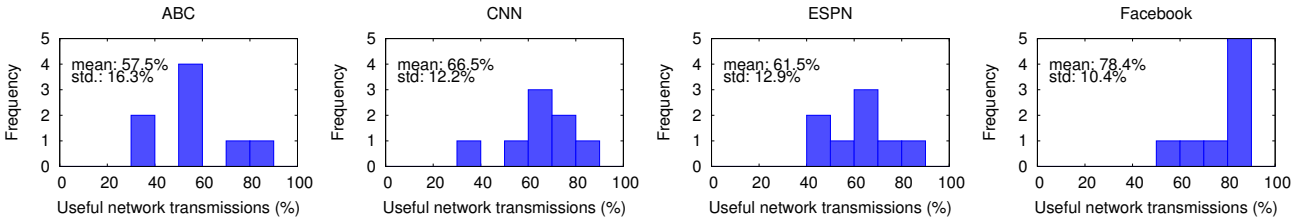

Figure 7: Histogram of percent of useful network transmission from user study for suspected leaky applications.

agating the design error into other applications. This bug could be fixed by stopping the thread in *onPause()*, which is guaranteed to be called after the application is placed in the background.

Poorly designed downloading scheme: In this case, the network device is frequently used to update data objects without considering either device state (e.g., whether the screen is on or off) or user input. We identify two applications with this problem: "ABC widget" and "BusMap". The first wastes 78.0% of network transmissions and the second wastes 46.9% of network transmissions.

"ABC widget" is a news widget application that can be placed on home screen. Surprisingly, by examining the timing of unused network packets, we find that the widget keeps updating news article every 5 minutes, regardless of whether the widget is being shown or even whether the display is on or off. This poorly designed downloading scheme is detrimental to smartphone battery life as it not only wastes communication energy and CPU when the phone is active, but also causes the phone to wake up every 5 minutes even when it is idle.

More interestingly, further exploration on the widget related APIs in Android leads us to believe that the Android framework fails to provide a good mechanism for widgets to stop updating when not visible. There are three scenarios when a widget is not visible: the display is off, the widget is placed on another home screen that is not displayed, or another application is running in the foreground. Android's API provides an indirect way to help the widget to identify the first scenario [21], but it is impossible for the latter two. Apparently, "ABC widget" even fails to identify the first one. This finding suggests improvements in the Android framework API, e.g., provide widget applications notifications when they are not visible or stop sending the update notifications when widgets are not visible.

"BusMap" contains another example of a poorly designed downloading scheme. It is a map-based application that informs users to the next three school buses to arrive at bus stops. A user selects a specific stop to view its bus schedule. The schedule for only a single stop can be examined at any particular time. Examination of the timing and content of unused network packets reveals a cause for energy leaks: regardless of which stop the user selects, the application downloads schedules for all stops, even those that are not visible on the current map. To eliminate the inefficiency, developers should design downloading schemes that take possible (and ideally common) user input into consideration.

Repetitive downloads "ABC widget" application is found to have repetitive downloads, e.g., packets containing identical content are downloaded but never used. By examining the content of its unused network packets, we discover that every time the application updates, it downloads all recent news articles, even when they have not been updated. Clearly, this observation suggests that the application naïvely fetches the latest news articles from the Internet without caching any of the downloads. In order to fix this bug, application developers could add an expiration time for each article or have the application server notify the client of new updates.

7.4 Aggressive prefetching

Prefetching has been used extensively in both desktop and mobile environments to improve user satisfaction by eliminating download delays for data that are expected to be required soon. It generally uses prediction, often of future user actions. Such predictions cannot be perfectly accurate. Therefore, even a well-developed predictive prefetching application will download some objects that are never used. In order to enable a rational understanding of the tradeoff between response latency and unused transmission, developers must first understand the percentage of unused transmissions by the target application. ADEL helps to provide this number and hence also assists developers to determine when it would improve application energy efficiency by adjusting prefetch aggressiveness and/or improving prefetching predictor accuracy. As shown in Table 4, four out of seven suspected leaky applications we studied have inappropriate prefetching settings.

Whether a significant portion of prefetching wastes energy cannot be determined by a single user's trace. This is mainly because an unnecessary packet for one user may be useful for another user due to varying use patterns. Hence, to capture the prefetching aggressiveness of an application, it is necessary for us to study the applications with different usage patterns.

We conducted a small-scale user study with 8 graduate students who are regular smartphone users running applications on top of ADEL. To avoid bias, none of the users were told the intention of the experiment until after it was completed. Each participant was asked to use the four leaky applications as they would in daily life; each application was run three times. There was no time constraint for any particular application. After gathering user traces for one application, the percent of *useful network transmission*s is derived from each trace. Note that this percentage is inversely related to prefetching aggressiveness. Figure 7 summarizes the distribution of the percent of *useful network transmission* for each application.

We can draw two conclusions from Figure 7.

- The percent of *useful network transmissions* for one application varies significantly across users due to variation in user behavior. As shown in Figure 7, the percent of *useful network transmission* spans a range of more than 40%. Note that each sample represents the average *useful network transmission* of three runs. The average variation of a particular user across runs is 4.2%. This substantial variation across users suggests that the optimal prefetching aggressiveness depends on the user; it suggests that a static prefetching scheme is unlikely to be optimal for all users.

- "Facebook" is more efficient than the other three suspected leaky applications. For example, more than 80% of the network transmissions are useful for 5 of the 8 users. What's more, the mean of the distribution is 78.4%, indicating that 78.4% the downloaded content turns out to be useful for an average user. As a result of this, we find that "Facebook" is not leaky, despite of our previous suspicion.

Based on our observations, the applications the energy efficiency of "ABC", "CNN", and "ESPN" can be improved by reducing

prefetching aggressiveness, at some possible cost to user-observed performance.

ADEL's ability to determine unused prefetched content enables three additional use scenarios in addition to detecting energy bugs. First, developers can use ADEL to learn the effectiveness of the application's prefetching scheme and adjust it during application design. Second, users can occasionally use ADEL to determine whether an application is prefetching with appropriate aggressiveness and adjust this parameter within the application or operating system. Third, with performance optimizations, ADEL could potentially be used within the operating system to provide applications with a real-time feedback on prefetching energy waste for one particular user. Applications could then adjust their prefetching aggressiveness to customize the user's current behavior.

7.5 Discussion

The above findings indicate that ADEL can help application developers to find the root causes of energy leaks in their applications. One may argue that a simpler approach can also detect such bugs. For example, repetitive downloads can be detected by simply checking the timing of the download pattern. However, without ADEL, the application developers do not even know the specific problems that should be tested for. For example, the developers may not know that the timing of download pattern is the most promising place to look for problems. ADEL makes it really easy for application developers to find where energy leaks occur.

Eliminating energy leaks detected by ADEL can improve energy efficiency. However, some leaks can be difficult to eliminate. For example, it is quite easy to fix the first three types of leaks in Section 7.2: the incorrectly used API can be fixed by changing API use; the widget that downloads data ignoring device state can be fixed by checking device state before download; and the repetitive downloads can be eliminated by caching. In contrast, it may not always be possible to develop a prefetching scheme that is both energy efficient and maintains adequate user performance because the very concept of prefetching relies on prediction, which is by nature imperfect. Therefore, it should be up to the developers to determine whether the potential energy saving suggested by ADEL justify the extra implementation effort.

8. CONCLUSION

In this work, we provided a definition of *energy leaks*: common but hard to detect energy waste resulting from useless activities in smartphone applications. We have described the design and implementation of ADEL, an Automatic Detector for Energy Leaks. ADEL identifies unnecessary network communication and its root causes via dynamic taint tracking. ADEL incurs 21.8% performance overhead on average.

We studied 15 real-world applications using ADEL and found that 6 of these have significant energy leaks. Eliminating these energy leaks results in an average 56.5% reduction in energy consumption due to network communication. Our study revealed four common causes for energy leaks: misinterpretation of callback API semantics, poorly designed downloading scheme, repetitive downloads, and aggressive prefetching. Both ADEL and our study of network energy leaks in Android applications have the potential to help application developers improve energy efficiency.

9. REFERENCES

[1] "iPhone battery life flaw confirmed by Apple," http://www.huffingtonpost.com/.

[2] "Android battery life: terrible, or just plain bad?" http://reviews.cnet.com/8301-19736_7-20058834-251.html.

[3] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices," in *Proc. Wkshp. Hot Topics in Networks*, Nov. 2011.

[4] F. Qian, et al., "Profiling resource usage for mobile applications: a cross-layer approach," in *Proc. Int. Conf. on Mobile Systems, Applications, and Services*, June 2011.

[5] A. Pathak, et al., "What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proc. Int. Conf. Mobile Systems, Applications And Services*, June 2012, pp. 267–280.

[6] R. Krashinsky and H. Balakrishnan, "Minimizing energy for wireless web access with bounded slowdown," in *Proc. Int. Conf. Mobile Computing and Networking*, Sept. 2002, pp. 135–148.

[7] M. Anand, E. Nightingale, and J. Flinn, "Self-tuning wireless network power management," in *Proc. Int. Conf. Mobile Computing and Networking*, Sept. 2003, pp. 176–189.

[8] D. Qiao and K. Shin, "Smart power-saving mode for IEEE 802.11 wirelss LANs," in *Proc. Int. Conf. Computer Communications*, Mar. 2005, pp. 1573–1583.

[9] T. Armstrong, et al., "Efficient and transparent dynamic content updates for mobile clients," in *Proc. Int. Conf. Mobile Systems, Applications And Services*, June 2006, pp. 56–68.

[10] B. Housel and D. Lindquist, "WebExpress: a system for optimizing web browsing in a wireless environment," in *Proc. Int. Conf. Mobile Computing and Networking*, June 1996, pp. 108–116.

[11] M. C. Rosu, et al., "PAWP: A power aware web proxy for wireless lan clients," in *Workshop on Mobile Computing Systems and Applications*, Dec. 2004, pp. 206–215.

[12] W. Enck, et al., "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. Int. Symp. Operating Systems Design and Implementation*, Oct. 2010.

[13] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for Java," in *Proc. Annual Computer Security Applications Conference*, 2005, pp. 303–311.

[14] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enformacement: a practical apporach to defeat a wide range of attacks," in *Proc. USENIX Conf.*, 2006, pp. 121–136.

[15] D. Chandra and M. Franz, "Fine-grained information flow analysis and enforcement in a Java Virtual Machine," in *Proc. Annual Computer Security Applications Conference*, Dec. 2007, pp. 463–475.

[16] PowerTutor, "PowerTutor," 2009, http://powertutor.org.

[17] H. Yin, et al., "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proc. Conf. on Computer and Communications Security*, Oct. 2007, pp. 116–127.

[18] B. Davis and H. Chen, "DBTaint: cross-application information flow tracking via databases," in *Proc. USENIX Conf.*, June 2010, pp. 12–12.

[19] A. C. Myers, "JFlow: practical mostly-static information flow control," in *Proc. of the ACM Symp. on principles of Programming Languages*, Jan. 1999, pp. 228–241.

[20] H. Falaki, et al., "Diversity in smartphone usage," in *Proc. Int. Conf. Mobile Systems, Applications And Services*, June 2010, pp. 179–194.

[21] "Android SDK reference," http://developer.android.com/reference/packages.html.