Mobile Offloading in the Wild: Findings and Lessons Learned Through a Real-life Experiment with a New Cloud-aware System

Marco V. Barbera, Sokol Kosta, Alessandro Mei, Vasile C. Perta, and Julinda Stefa
Department of Computer Science, Sapienza University of Rome, Italy
Email: {barbera, kosta, mei, perta, stefa}@di.uniroma1.it

***Abstract*—Mobile-cloud offloading mechanisms delegate heavy mobile computation to the cloud. In real life use, the energy tradeoff of computing the task locally or sending the input data and the code of the task to the cloud is often negative, especially with popular communication intensive jobs like social-networking, gaming, and emailing.**

We design and build a working implementation of CDroid, a system that tightly couples the device OS to its cloud counterpart. The cloud-side handles data traffic through the device efficiently and, at the same time, caches code and data optimally for possible future offloading. In our system, when offloading decision takes place, input and code are likely to be already on the cloud. CDroid makes mobile cloud offloading more practical enabling offloading of lightweight jobs and communication intensive apps. Our experiments with real users in everyday life show excellent results in terms of energy savings and user experience.

I. Motivation, vision, and goals

In our thinking, a smartphone represents the "whole package", the all-in-one gadget that has made our life much easier, almost substituting our computers. Almost. Not because of the computation or storage limits—nowadays smartphones are more powerful than the desktop computers of a few years ago. Not because of the networking limits—smartphones come with built-in 3G/WiFi technology. But because of the limited energy. The battery cannot even cope with the multitude of mobile purpose-built apps that we like to use everyday, that make us exploit our smartphones to the fullest.

To overcome the hard limits imposed by the battery duration of smartphones researchers have focused on building frameworks that enable offloading of mobile computation to the cloud [1], [2], [3]. On a high level, offloading techniques operate as follows: Whenever a piece of code is to be offloaded they trade-off between the energy saved on the device by not computing the code locally, and the energy consumed by the network interfaces to send to the cloud the code and the input needed for the remote execution. Put shortly, they are useful only as far as computation-intensive applications that involve little data transfer to offload are concerned. They often take decisions to execute locally, even though the task is heavy, just because the networking costs to send the input necessary to run it on the cloud would induce huge overheads to the device. In addition, when it comes to apps that involve heavy communication, rather than computation, like Facebook mobile, WhatsApp, Email, Twitter, online gaming and so on, current offloading techniques cannot do much. Unluckily, these apps are the ones that drain most of our devices' battery [4], [5]. For these reasons, researchers are becoming skeptic on the actual benefits of offloading. This feeling is strengthened by the fact that no offloading framework built so far is publicly available, nor has it been tested with real users (only in labs).

At this point, the following questions come naturally: "Can users in real life profit from cloud offloading?", and "Is it possible to design a cloud-aware system that makes computation offloading much more efficient and allows for offloading of communication-intensive tasks?". Our goal in this work is to give an answer to the above questions. For this reason we build CDroid (Cloud-anDroid), an offloading-aware system transparent to the user, and distribute it to real-users to assess its performance. We design CDroid so that it enables offloading of communication-intensive apps, and increases the gain of offloading of computation-intensive apps. In CDroid every device comes with its own cloud counterpart, seen as just another resource of the real device, only a 3G/LTE Advanced connection away from it. So, it brings the device and the cloud closer, towards fully integrating them.

The cloud-side of CDroid is made of different components. One of them is the *communication handler*, which handles all the mobile data traffic. Together with the *caching and prefetching* and *traffic compression* components they optimize the usage of the network interface of the device. Similarly to [6], [7], but residing on the cloud-side of the device. Most importantly, by having all the data traffic handled by the cloud counterpart, CDroid allows us to boost the gain from offloading: The code of applications installed and most of the data reside on the cloud, except for user-generated data (e.g. notes, pictures and so on) that are efficiently synchronized with the cloud-side through the CDroid *syncronization module*. This lowers the traffic the device needs to send for remote execution, by thus allowing for more battery savings.

These are not the only benefits of a CDroid-like system: It allows us to envision and build new security and privacy mechanisms that go far beyond code offloading. In the prototype we built we included components that provide anti-phishing and spam notification filtering, and pre-screening of new installed apps for malware prior downloading them to the real device.

We deployed CDroid on 5 real devices (primary smartphones of 5 participants in our experiment) for two weeks, and compared its performance with that of plain Android OS. The testbed results show that, with CDroid, we achieve: (1) improvement of the device's usability and responsiveness; (2) a boost in the security and privacy of both the device and our

This work has been performed in the framework of the FP7 project TROPIC IST-318784 STP, which is funded by the European Community. The Authors would like to acknowledge the contributions of their colleagues from TROPIC Consortium (http://www.ict-tropic.eu).

data; (3) a significant reduction on the battery consumption; (4) remarkable savings in cellular data traffic that directly traduces in monetary savings for the users.

II. Related work

Here we summarize relevant contributions to mobile cloud offloading and works dealing with mobile spamming, that is becoming one of the major user concerns.

A. Cloud-aided mobile applications

Cloudlets [8], is one of the first works that suggests remote execution of mobile code. Its authors make use of resource-rich nearby computers to which smartphones connect over wireless LAN to instantiate customized service software. MAUI [1] provides method level code offloading for Microsoft .NET applications. The framework decides during runtime whether to offload a running method. Conversely, CloneCloud [2] uses an offline static analyzer and a dynamic profiler to partition the bytecode of an application. After the partitioning, an optimizer solver determines the portions to offload so to minimize the expected cost of the app as a whole. The optimizer solver is fed with a set of running conditions, and an offloading profile is generated for each of them, and pre-loaded on the device. But, the bounded number of offloading profiles generated this way limits the framework's usability. ThinkAir [3] provides on-demand resource allocation and parallel execution on the cloud operating at method level. In [9] the authors study, through a testbed of 11 Android phones and relative software clones running on Amazon EC2, the offloading costs in terms of bandwidth and energy consumption.

Our mobile devices are used not only to store data that are important to us (pictures, media files, credit card codes, and so on), but also for economic transactions (internet shopping, online banking, payphone systems, etc.). Thus, mechanisms that protect our data and our wallet from the imminent security threats [10] have become more than valuable in mobile systems. Unfortunately, excellent techniques like TaintDroid [11] can be unfeasible for our battery limited smartphones. This is also confirmed by [12], that pursues a mobile security versus energy tradeoff. The authors conclude that protecting against code-driven attack can be affordable, while protecting against all data-driven attacks is prohibitively expensive. The first cloud-based approaches [13], [14], [15], [16] assume a single entity to which the information from all the mobiles in the system conveys to, and that takes decisions regarding the presence of viruses or malwares in the single devices. But, these approaches have scalability problems. When the number of smartphones in the system becomes very large, the load induced to the security server increases enormously. In Paranoid Android [17] and [18] the approach is different: The systems use replicas of smartphones on the cloud to provide security services for mobile devices. A recording/replaying technique is used in [17] to synchronize the replica with the phone. The overhead introduced by the recorder on the phone accounts for 30% of extra energy spent. Lastly, in [19], [20], [21], [22], the authors show how to exploit virtual peer-to-peer network of smartphones on the cloud so to increase the security of our devices. In addition, SociableSense [23] shows how also social related applications can benefit from cloud offloading and [24], [25] exploit sociality among smartphone clones to secure mobile devices. Similarly, also systems like [26], [27], [28], [29], [30] might benefit from offloading computation to the cloud.

Recently, Google considered introducing a data compression proxy in Chrome browser for Android[1]. They notice that, for an average page, over 60% of the transferred bytes are images. Works like [31], [32] show the benefits of local or cloud-base proxies to mobile traffic optimization. In [5] the authors observe that, in terms of energy consumption, rendering of JPEG encoding is the most efficient.

B. Mobile Notification Spamming

The notification status bar, coupled with the Android notification service notifies the user about system-related issues (e.g. through the Google Cloud messaging[2], the Android push service), and about the app related events (e.g. a new email and so on). Its utility is unquestionable. This might be the reason why it has become the favorite target exploited to launch vicious phishing or spamming attacks [33], [34]. The work in [35] shows the smap side of ads: The way developers trick the users to click on them is very intrusive and annoying. The authors in [36] present an Android app profiler. They study the behavior of 27 famous applications and observe that many of them communicate with way more servers than expected (as many as 13). They show that the free versions of the apps could end up costing more than the paid counterparts, due to an order of magnitude increase in network traffic.

That said, it is important to notice that, in all of the current offloading or cloud-based security mechanisms, the device and the cloud are two different entities, limiting the possibilities offered by the mobile cloud paradigm. In addition, they are designed to be app-based, rather than system based. Thus, if put together as they are on a single device, it is very likely that the energy-efficiency benefits would be shadowed by the enormous overload they would incur to the system.

Our approach here is inspired by [6], [7]. These works aim at using the wireless network links more efficiently by respectively exploiting local proxies [6] or integrating efficient prefetching systems on the device [7]. But, our goal is different: To build a user-friendly cloud-aware system that optimizes networking of the real device through the cloud, makes offloading of computation-intensive applications more effective, achieves offloading for communication-intensive applications, and increases the security of our smartphones. All this with minimum energy costs (just for the sync between the device and its cloud-counterpart) of 2.5% of daily battery, that bring enormous benefits: 62% of cellular traffic reduction, 4–times less energy spent for web-surfing, 10–times less energy for offloading of computation-intensive apps, and 2.5–times less energy waisted by a more parsimonious use of the 3G radio.

III. Architecture

The CDroid system (depicted in Figure 1) consists of two parts: The device-side, residing on the smartphone, and the cloud-side, residing on the cloud and coupled with the device. The cloud-side is an instance of Android-x86 OS[3], the porting of the Android OS to the x86 platform. During the set-up phase the user sets an account password that allows her to access the features of CDroid. Here we describe the components of the system that we implemented.

[1] https://developers.google.com/chrome/mobile/docs/data-compression

[2] http://developer.android.com/guide/google/gcm/index.html

[3] www.android-x86.org

1) CDroid-Device: Is the device-side of the system. It executes on the phone as a background service. It collects informations about user activities and behavior like phone calls, SMS sender and receiver information, GPS coordinates, emails, Bluetooth devices and WiFi hotspots discovered or connected to, and 3G networks. This information is recurrently saved in a log file on the device. The logs are sent in batches to the cloud-side of the system as a piggyback to user traffic.

2) CDroid-Server: Is the first cloud-side component of the system. It collects the information sent by the CDroid-Device and passes it to the other components. In addition, it handles the various user connections by invoking the proper handlers (described later in this work) that allow user commands to enable or disable specific system features.

3) Connection handler: This component is responsible for the http traffic tunneling of the real device through the CDroid-Server. To make this possible, we implemented it as a http proxy. The Connection handler is the first component to receive the connections from/towards the phone. It analyzes separately the header and the data of the connection, and, depending on them, delegates the connection to the responsible component.

4) Secure connection handler: Our CDroid system also features a handler for secure connections, that takes care of https connections. The choice of handling http and https connections separately by using two different components has two advantages: (1) The user can decide to tunnel the http connections only (e.g. the user might not want the cloud-resource to handle its online banking). (2) The user can enable the CDroid-Server to decrypt the secure connections and analyze the header and the data for other reasons, e.g., compressing https traffic to reduce cellular bandwidth through the Content Compression handler, described later in this section.

5) App handler: At the beginning of 2012 Google introduced Bouncer[4], a system that scans all apps on Google Play store for malware. Unfortunately, third party markets, becoming more and more popular, are not using any system to protect users from malware code. Recently, after acquiring VirusTotal company, Google announced that will introduce the anti malware feature in all the new Android phones[5]. Besides from suggesting that the risk of malware apps is still present and cannot be stopped only by pre-scanning apps uploaded on the Google Play store, this solution will certainly induce energy overhead to our devices.

The App handler component of our CDroid system serves as a security screen of new apps downloaded, prior the installation on the real device. When the CDroid-Server detects a request for a new app download, the App handler installs the *apk* in a sandbox on the cloud-side, and puts it in quarantine state. Then, one or more anti-malware softwares are run to scan the app for malicious behavior. In addition, the App handler blocks and logs all Internet connections that start from the quarantined app. If the app passes successfully the quarantine phase, the *apk* is sent to the user device for installation. Otherwise, the app is flagged as potentially malicious and the apk is not sent to the device unless the user decides to install the flagged app regardless. The quarantine time can be set and changed by the user in any moment.

6) Synchronization module: In CDroid he cellular traffic is tunneled through the cloud-side of the device. This allows the cloud-side to be up to date with the smartphone, as far as the uploaded/downloaded data are concerned (pictures uploaded to Facebook, files downloaded from the Internet, code of newly installed apps etc.). But, it is not so for user-generated data (notes, pictures taken with the camera, and so on). The synchronization module fills this sync gap: It makes sure that the data generated by the user on the phone is sent to the cloud-side. For efficiency reasons, the synchronization module exploits a mechanism similar to the rollback hashes technique used in [9] to send only newly generated files, or changes of existing files to the CDroid-Server. It handles all the preparation (on the device side) of the data to be sent, and takes care of copying the data on the CDroid-Server side within the same path the files were created on the CDroid-Device. This module enables the user to set different synchronization frequencies for her data and her apps backup, according to her needs. As we will see from the results achieved by our testbed, the sync module makes the device spend 2-times less energy with respect to [9] for the sync with the cloud-side for offloading purposes.

7) Remote Code Executor: We integrate the ThinkAir framework [3] to the CDroid architecture to enable remote code execution. When the user decides to include an app in those usable through the ThinkAir framework for better battery performance, the CDroid-Server is notified, and the app is installed also on the cloud resource. The Remote Code Executor is then, from that moment on, responsible for the execution of the offloaded methods and for handling the necessary data transfer from the device. The ThinkAir framework is modified so to communicate with the synchronization module: Only files that are not on the cloud-side yet are sent after an offloading decision is taken. Our experimental results show that this aspect, together with the lightweight sync module, increase the energy-gain from the offloading of heavy tasks up to 10-times.

8) Remote wiper: It can be used when the user loses the phone and wants to delete everything remotely. It operates as follows: The user, after deciding to do a remote wipe, connects and authenticates herself to the CDroid-Server from a third terminal. Then, the user asks for the remote wiper procedure to be started. This procedure triggers a command sent to the CDroid-Client that immediately executes a phone wipe and delete all user data and installed applications from the device.

9) Cookie handler: It protects the user from cookie theft and enables the unification of cookies for all applications. When enabled, it intercepts the cookies received from servers the user connects to, removes them from the response header prior sending the response to the user device, and stores them on the cloud-side of the system only. On consecutive connections the Cookie handler puts back the correct cookies on request headers coming from the phone.

Another feature of the Cookie handler component is to safeguard users' privacy and accounts enabled on the smartphone in case of theft: The user can trigger the component from a third terminal to block all cookies towards the device. So, the thief cannot access to any of user accounts unless he breaks the password and re-authenticates with the CDroid-Server.

10) Sensitive information blocker: It analyzes the traffic generated by the device to detect leakage of (unaware) users' sensitive data. If an anomaly is detected the Sensitive information blocker alerts the CDroid-Server which notifies the CDroid-Device to inform the user, and blocks the transmission

[4]http://googlemobile.blogspot.it/2012/02/android-and-security.html
[5]http://tinyurl.com/c7n6g5o

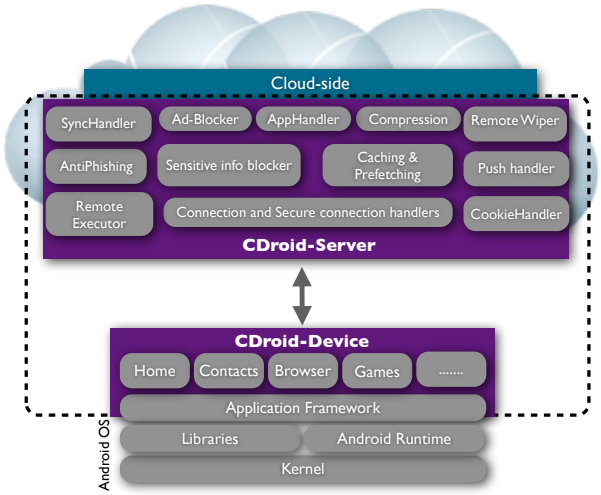


Fig. 1: The CDroid architecture.

if necessary. Sensitive informations are passwords, credit card or cellphone numbers, etc., and are selectable by the user. Finally, this component features an *Anti-phishing handler* that checks the urls in the user request headers to detect if there are links to phishing websites.

11) Mobile Advertisement blocker: App developers often display intentionally ads over areas where the user would normally tap during the app usage. This is to maximize the chances for the user to click on the ad, so to increase their revenue. Once accidentally clicked, the ad opens the browser by interrupting the user and sending her to a web-page related to the ad. While this results in a very annoying user experience, it is a very commonly used technique [35]. In addition, ads generate data traffic unwanted by the user. This traffic can severely impact on the costs of the mobile Internet plans for many user categories: The "tricky" ads are being introduced in every type of app—from mobile gaming, free messaging, horoscope apps (more popular to younger generations), to health and aging related apps (more popular among older generations). These unwanted costs can raise up to 10 euro per day for travelers: Roaming Mobile Internet with Italian carriers for example, costs about 5 euro daily for fixed amounts of traffic (typically of 50-100 MB per day). When the mobile traffic exceeds this limit, the cost raises up to 2 euro per MB.

What is worse, the traffic generated by these ads can be a threat to the privacy: Many ad companies send ads customized to the user profile, which suggests that sensitive user information is leaked when retrieving new ads.

The Mobile Advertisement blocker component frees the user from the unwanted ad traffic. It is user-enabled and on a per application bases. It operates as follows: When an app or a web-page sends a request for an ad, the CDroid-Server is alerted to not satisfy the request, and simply reply with a denied message. Ad requests are easily detectable—they are directed to servers publicly known as dedicated to ad distribution. Enabling this component results in less traffic generated, better user experience, and more privacy.

12) Content compression: This module aims at improving user experience when navigating or accessing client-server based platforms, like e.g., Facebook. It compresses text pages in gzip streams, and converts all images in JPEG, the most efficient encoding format supported by Android [5]. In addition, it enables the user to setup a target image quality, and reduces the image size accordingly. As we will see from our experimental results, the downloads result faster and the cellular traffic is dramatically reduced. Consequently, the battery consumption for networking is considerably lower (up to 4–times less).

13) Caching and pre-fetching: The cache size of Android devices is quite small [32], and does not allow for much content to be cached for long time. This puts limits to the usability and responsiveness of the device. The Caching and pre-fetching component on the cloud resource aims at fixing this shortage: We set a bigger cache on the cloud resource, which enables the possibility for more information to be cached and for a longer period of time. The component features also a predicting mechanism for the navigation patterns of the user. The pre-fetching module makes the connections according to the prediction, and loads and caches the content. So, the device fetches the content readily from the CDroid-Server, without needing to wait to connect to the respective (and possibly slow or overloaded) remote server from which it was pre-fetched by the Caching and pre-fetching component.

14) Push notifications handler: Many apps that we use (Gmail, Facebook, Twitter, WhatsAapp, and so on) feature push notifications that require the device to keep persistent connections with the server of the specific application. There are several reasons for users to control and filter push notifications. Here we list some of them:

- *Energy consumption:* Keeping many active connections with multiple push servers impacts severely the already limited battery of the smartphone.
- *Privacy concerns:* Push notifications are achieved through two main techniques: Either the phone keeps a persistent TCP connection towards the server (e.g. Google Cloud Messaging (GCM)) or connects at regular intervals to the server and asks for new notifications (e.g. Airpush ad network). In both cases the push server will be able to estimate the phone's location, by thus threatening the user's privacy.
- *Avoid spam:* Many Mobile Ad network companies—Airpush[6] being the most famous—are exploiting app notifications to inject ads in Android phones. This way, victim users receive ads at any time, even when non using the apps. What's worse, the user cannot detect the app responsible for these spam ad notifications. The problem is becoming so severe that apps like Airpush detector have achieved more than 1 M downloads in Android play store. These apps can only help the user identify the app responsible for the ad spam, but cannot block it. Recently, in Android 4.1, Google introduced a feature that allows to disable push notifications on a per-app basis. This solution however, together with spam notification, prevent the user from getting the real push notifications. In addition, it is not available to users with older versions of Android OS.

The push notification handler manages the device's persistent connections on the cloud-side, and shrinks them to 1: That with the cloud resource. It is this latter that keeps the many connections with the respective remote servers on the devices' behalf. It features a mechanism similar to Airpush detector, that detects the apps generating the notifications. The handler features also a spam report mechanism, that collects notifications

[6]http://www.airpush.com

TABLE I: Specifics of the testbed mobile devices.

Number, type & OS	CPU	RAM (MB)
3×Samsung Galaxy S+ (Android 2.3)	1.4 GHz	512
2×Samsung Galaxy S (Android 2.3)	1 GHz	512
2×Nexus S (Android 4.0.1)	1 GHz	512

marked as spam by the user, and uses them as a filter for future notifications. In addition, the user can decide: (1) To disable notifications on an app-bases; (2) the frequency of receiving notifications of any kind; (3) to store notifications on the cloud-side and to retrieve them in a consecutive moment. This has many benefits. First, delegating all the persistent connections to the cloud resource helps reducing the energy consumption on the real device. Second, it is the cloud resource, not the device, that is interfacing with the push servers. So, this completely hides the real device from the push servers, and makes it impossible to localize the user. Third, the per-app bases notification disabler allows even old fashioned devices not running Android 4.1 to use this feature. Last, but not least, the anti-spam notification filter dramatically reduces the spam notification received by the user. What's more, its detection mechanism that resides on the cloud side eliminates the need for the user to install and run, on her battery-limited device, remedy apps like Airpush detector.

15) Handling connection failures: Another benefit of CDroid comes from the way it handles failures due to sudden connection interruption (e.g. the smartphone enters in areas with no signal). When this happens, the communication between CDroid-Device and CDroid-Server is frozen, and it restarts from where it was left before the interruption occurred. In the mean time, the components of the CDroid-Server keep possible connections with remote servers up and running, and store locally the data directed to the device: The caching and prefetching component takes care of the data comming from possible opened connections (e.g. the user was checking her Facebook profile when the interruption happened), whereas the push-notification handler stores locally the possible notifications arriving in the mean-time. As soon as the device reconnects again, these components are notified from the Connection handler component. The data downloaded is then passed to it and sent to the device following a FIFO politic based on the download time. This is very beneficial from the devices' point of view: Instead of reconnecting to the many servers it would have to connect to to pull possible notifications (e.g. Facebook, Tweeter, Gmail, and so on), the device only connects to its cloud-side.

IV. Evaluation

To evaluate our CDroid system we used the devices whose technical specifications are detailed in Table I. The CDroid-Servers resided on Amazon EC2 platform's High-CPU Medium Instances–1.7 GiB of memory 5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each). To measure the energy consumed by the devices in our experiments we exploited the Mobile Device Power Monitor[7], used by many works in the area [1], [3]. It samples the smartphone battery with high frequency (5000 Hz) to yield accurate results on the battery's power, current, and voltage.

Towards assessing the performance of CDroid we performed two types of experiments. For the first one we deployed the system among three university students, a professor, and a teenager, installing the CDroid-Device on their personal phones. The experiment lasted 2 weeks. During the first week the system did not enable CDroid features—It behaved like plain Android OS. During the second week it switched to CDroid automatically, without notifying the users. This allowed us to conduct an unbiased study with the two systems at the end of the experiment, whose findings we report later in this section. The second experiment was carried out in our lab. It aimed at measuring the energy spent and the time taken by a Samsung Galaxy S phone to load HTTP and HTTPS pages with Android and CDroid.

A. First experiment: CDroid and regular users

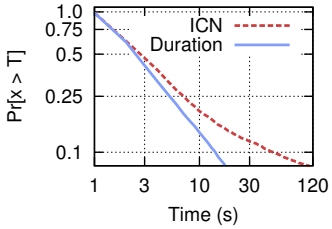
1) Setup and methodology: As we already mentioned, for the first experiment we installed the CDroid system to 5 participants (a teenager, three university students, and a professor) and let the system run for 2 weeks. The system was set-up so that it behaved like plain Android OS during the first week of the experiment, and switched to CDroid only during the second week. The users were not notified about this change, and the phone logged all user activity during both weeks (we used the logcat generated by the system). The sync-interval of CDroid was set to 2h, and the push-notification interval was set to 5 minutes.

2) Experimental results: This long experiment shed light on many insightful phenomena and aspects that allow CDroid-enabled smartphones to save battery and cellular data traffic. First, from our logs it resulted that during the first week of the experiment, when Android OS was enabled, there were many short background connections (not triggered by the user). This was true also for the 2 users with Android 4.x OS: Though Google introduced in Android 4 the possibility for the user to disable the background data for specific applications[8], none of them had actually disabled it. To investigate the characteristics of the background communications we plotted, for all the 3G connections, the distribution of their duration of the time-span between two successive 3G connections (inter-connecting time), and the cumulative distribution of data transferred. The results are shown in Figures 2(a) and 2(b). We note that more than 50% of all the 3G connections happen in the background, not being triggered by the user. These connections last less than 3 seconds (see Figure 2(a)) and the amount of data transferred is below 1KB per connection (See Figure 2(b)).

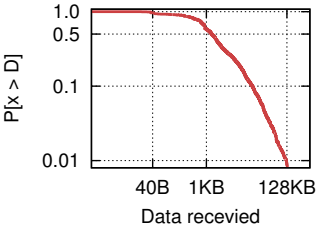
The fact that more than 50% of connections are short and in the background (active even when the user is not using the device) is not good from an energy standpoint. The reason is the following: After a transmission, the 3G radio passes from the DCH energy mode (full power, about 800 mW) to FACH mode (the half-power state, about 460 mW), that allows with less effort to go back to DCH mode in case of need [37]. If for a certain time-span (from 12 to up to 30 seconds for some devices) there is no connection request, it switches to the IDLE mode—the one that allows for lowest energy consumption. Therefore, short and very frequent background connections increase enormously the amount of time the 3G interface spends in the energy-costly half-power FACH mode. This directly impacts negatively the battery duration. To have

[7] http://www.msoon.com/LabEquipment/PowerMonitor/

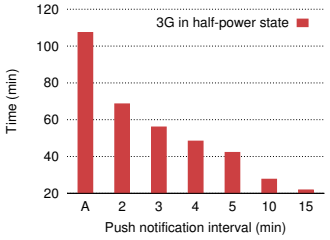
[8] http://support.google.com/android/bin/answer.py?hl=en&answer=1638168



(a) 3G connect duration and inter-connecting times (ICN).



(b) Data downloaded size.



(c) Average time per user per day of the half-power FACH state for the 3G radio in dependence of the push-notification sync interval.

Fig. 2: Cumulative distributions of (a) 3G connect duration and inter-connecting times between successive connections, (b) incoming connections data size, and (c) the time spent by the 3G radio in half-power state (A-is with Android OS (no sync). With CDroid we used 5-minutes long intervals. The other values are simulated according to the logs.).

an indication on how much, we measured the average 3G half-power time per user per day during the first week of the experiment. The result is surprising (see the first column in Figure 2(c)): more than 1h and 50 min per day the 3G radio power is in FACH mode. Speaking in terms of energy, this translates in 2962.6 J—about 13.52 % of the battery capacity. (The smartphones of the participants in the experiment were all powered by Lithium–Ion batteries (1.650 mAh, 3.7 V). These batteries, if fully charged, contain 21.9 KJ of energy.)

During the second week of the experiment, the one in which the users had CDroid enabled, we set the push-notification interval to be of 5 minutes (fifth column in Figure 2(c)). This drastically reduced the energy waisted due to the 3G radio to 1165.66 J (only 5.3% of the battery, 2.54 times less). For completion, we simulated what would have happened if the push-notification sync interval was longer (up to 15 min) or shorter (down to 2 min). The remaining columns of Figure 2(c) show the simulated results, according to our testbed logs. Recall that this interval is user-settable. The results are as expected: The longer the push-notification interval, the shorter the time in FACH mode of the 3G radio (which means less energy waisted). This suggests that the user should trade-off between freshness of notifications and battery duration.

Through our testbed, we are also able to investigate the amount of traffic handled per day with both Android and CDroid. The results are shown in Figure 3. Thanks to the Content compression component of the CDroid system, we gain more than 80% of cellular data traffic from ad blocking and image compressing, and more than 50% from text compression. Over all, the traffic saved is 62%. So, with CDroid, a user can save 62% of her data-plan per day. This becomes very valuable in roaming situations, when the cost of cellular traffic is already very high: As we already mentioned, Italian Network operators charge several euro per day (up to 5) for a few MB of traffic (around 50 per day). Not to mention the high charges when the user exceeds the limit.

Another aspect of CDroid that makes the user save battery is the caching and prefetching component and the optimized sync component. Recall that, thanks to the former, the files that are downloaded from the Internet are stored locally on the cloud-side, before sending them to the device. The same is valid for files that the user uploads on remote servers intentionally (e,g. pictures uploaded on Facebook). So, the optimized sync component, only takes care of sending to the cloud-side files that the user generates locally, and has not yet (before the sync interval) sent remotely. Thanks to a

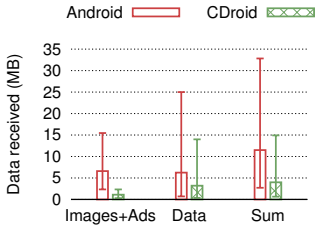
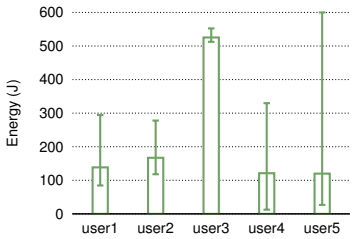
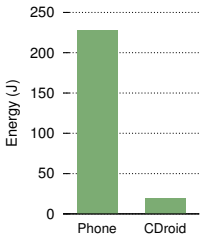


Fig. 3: Adds, images and text downloaded per day. The graphics include the average, minimum and maximum values per user.



(a) Energy gained with the optimized sync in CDroid.



(b) Energy saved by offlading with CDroid.

Fig. 4: Energy savings with CDroid: Optimized sync with the cloud-side and boost of computation-intensive apps offloading.

combination of these two components, we gain about 50% of the energy to sync daily the devices with the cloud, as studied in [9] (see Figure 4(a)). This is a very important result—CDroid reduces the energy costs to keep the cloud-side up-to-date for offloading purposes to 2.5% of the battery.

The sync component of CDroid has another important benefit: It distributes the sync costs, even for user-generated files, overtime. Thus, the cost of sending the input do not impact negatively possible offloading decisions for computation-intensive apps. Actually, it makes so that energy gains are increased. To verify this we did the following: We individuated one particular user in our testbed (user 5—a grad student in our lab) that, during the experiment, was continuously using the camera to take pictures, and downloading images from Facebook. At the end of the experiment, we installed on her device a third party Face Detection app for Android[9]. It counts the number of faces in a picture and computes, for each detected face, simple metrics like distance between eyes. Then, we run the app over 100 images saved on user's phone (50

[9]http://www.anddev.org/quick_and_easy_facedetector_demo–t3856.html

generated by the user, and 50 downloaded from the Internet). First, we disabled CDroid, and installed on the device the ThinkAir offloading framework. The result is shown on the left column of Figure 4(b): ThinkAir decided to executed the task directly on the phone, due to the high predicted energy costs to send the pictures and the code remotely (more than double). Then, we enabled CDroid, and re-run the task again, on the same images. The result, also depicted in Figure 4(b), is very promising: Due to the optimized and lightweight sync mechanism of CDroid, which synchronizes the cloud-side continuously without impacting much on the battery (only about 2.5% of it daily), both the files and the code are already residing on the cloud-side. Therefore, the offloading costs drop down to 10% of the costs of running the task locally. This is again a very important result: CDroid not only increases the gains from computation-intensive apps, but it actually enables offloading in extreme cases, when the input necessary to run the task remotely is very large, and when other offloading frameworks decide (correctly in their case) that offloading is not going to reduce battery consumption. All this at the cost of 2.5% of the battery per day.

B. Second experiment: Loading HTTP and HTTPS pages with CDroid

1) Setup and methodology: The second experiment was carried out in our lab. It aimed at assessing the performance of CDroid with communication-intensive apps. We chose to target the Browser, one of the most used app in Android. To increase the accuracy of our results we modified and rebuilt the native Android browser so to log the internal activities. Then, we measured the following: (1) the number of bytes sent and received by the smartphone; (2) the time to load a given web-page; (3) the energy spent during the process. The pages involved in the experiment were the m.9gag.com page, and the m.facebook.com page. This choice was done on purpose: 9gag is a HTTP page that opens occasionally HTTPS connections for user comments to social sites; m.facebook.com is an HTTPS page. In addition, we chose m.facebook.com since Facebook is one of the mostly used apps on smartphones. The reason for not using the native apps is that they are not open source. Thus, they cannot be modified so to output log information as the native browser. The phone used for this experiment is the Samsung Galaxy S (see Table I).

We define the *startLoadingPage* as the moment the browser receives the command to start loading the webpage, and the *endLoadingPage* as the moment the browser finishes loading all the content for the current page (the loading bar has reached 100%). The time to load the page is given by the difference between these two moments, and the number of bytes RX/TX by the browser during this interval defines the quantity of data received/transmitted by the device to load the given page. The pages were loaded 10 times with a pause of 300s among two consecutive loads. The loading was carried out in automatic, exploiting Android input injection. Before reloading for the next measurement, the browser's cache was cleared in order to avoid biased results due to caching. The experiment was repeated with both Android and CDroid 3 times a day, during three different time-intervals: Morning (9am-1pm), Afternoon (1pm-5pm) and Evening (5pm-9pm).

2) Experimental results: Let us first comment on the results related to the bandwidth consumed (bytes received and sent), and the loading time, presented in Figures 5 and 6 for 9gag and Facebook, respectively. The first observation to make is that in both cases CDroid wins over Android in terms of both loading time and cellular data traffic generated. Especially in the 9gag case, the loading time with CDroid decreases by 5 times in the 9am-1pm time interval. The gap between the two systems shrinks as the time-interval approaches the evening. This is expected: We conducted this experiment in our lab, in a university campus. The campus is more frequented in the first half of the day due to students going to classes. As a result, more devices enter the campus area, and the cellular network starts suffering from the traffic generated. With CDroid, however, this effect is mitigated: By compressing both images and text, the CDroid system makes the device suffer less from the high-latency of the network bandwidth during the morning hours. Indeed, the data downloaded with CDroid is considerably less—down to 300% less on 9gag, and 36% less on Facebook according to the all-day average value (see Figures 5(b) and 6(b)). What is more, CDroid features the ad blocker component, which, by blocking the incoming ads, decreases also the out-going traffic: Ads often contain java-script code that opens connections towards other servers without user intervention. This is clearly reflected by Figures 5(c) and 6(c), where we show the amount of out-going data traffic with both systems. CDroid reduces the amount of bytes received to 33% less for 9gag, and to around 10% less for Facebook, according to the all-day average value. It is important to note that the gain is more evident with 9gag as it includes more images and more ads.

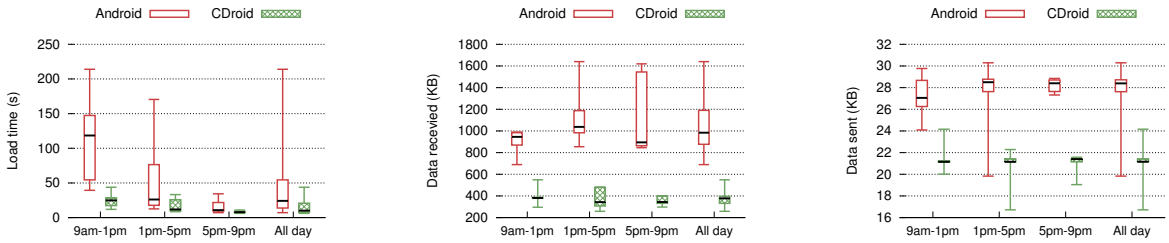
Finally, both the loading time and the usage of the network data traffic impact the energy spent on the device to load the pages. As expected, with CDroid it is much less—around 4 times less for 9gag, and 2.3 times less for Facebook according to the all-day average value (see Figure 7).

V. Findings, difficulties, and lessons learned

Though the current implementation of CDroid already achieves very good results, we acknowledge that there are aspects of the system that can be improved. Here we discuss some of them that we deem as important. In addition, we review some of the difficulties we faced while building the system and describe how we overcame them.

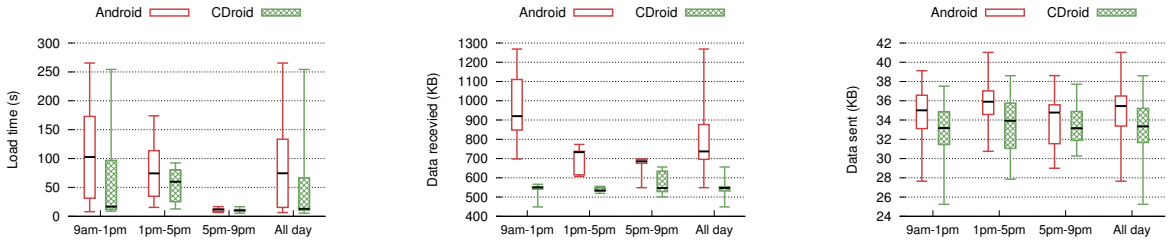
The various components of CDroid handle and modify both HTTP and HTTPS traffic, so to analyze their content and detect e.g. user information leakage, and to compress the content in order to improve rendering and reduce cellular network traffic. HTTPS traffic, however, has to be firstly decrypted before operating on it. In order to do so, in the current implementation of CDroid, the Secure connection component performs the well known man-in-the-middle attack: The Secure connection component generates on the fly a fake certificate of the targeted server. In addition, it acts as a Certificate Authority (CA) and signs this certificate with its own certificate, that we pre-load on the user device (we include it within the database of trusted CA provided by Android). It is worth noting that, though this technique works with most of the mobile apps, there exists specific apps (e.g. Twitter), that do not accept other certificates besides those signed by a trusted (to Twitter) CA. Even so, these apps are very limited.

That said, we applied the man-in-the-middle attack to empirically prove the benefits that can come from HTTPS mobile traffic handling. We do recognize though that this is certainly not the right way to handle HTTPS traffic on



(a) Page loading time. (b) Bytes received per page. (c) Bytes sent per page.

Fig. 5: Load time and cellular bandwidth vs the day-time interval: The 9gag case. The graphics include the average, minimum and maximum values, as well as the 25^{th}, and 75^{th} quartile.



(a) Page loading time. (b) Bytes received per page. (c) Bytes sent per page.

Fig. 6: Load time and cellular bandwidth vs the day-time interval: The Facebook case. The graphics include the average, minimum and maximum values, as well as the 25^{th}, and 75^{th} quartile.

real and large-scale deployments of a CDroid-like system. A legitimate way to deal with it could be by exploiting proxy re-encryption techniques [38], based on the user's passwords for each platform that generates the HTTPS traffic (e.g. Facebook or GMail). However, this would require that the CDroid system is equipped with a legitimate certificate and public/private key pairs (both belonging to the user), and recognized by the servers of the apps as a legitimate entity that can re-encrypt the traffic from/to the user device.

The current implementation of the sensitive information blocker component, analyzes only HTTP and HTTPS traffic in search of leakage of user information. We are conscious though, that the malware builders might encrypt the sensitive data with a symmetric key known only to the corresponding server, so to overcome our blocker. This would make it more difficult to detect the information leakage. A possible solution to this problem is to enhance the sensitive information blocker component by including complex tried and tested IT security mechanisms[10]. This component runs on the CDroid-Server, and not on the CDroid-Device. So, it does not matter how computation intensive these techniques are. The responsiveness of the device would not be impacted at all.

The Push notification handler component of the CDroid system in its current state, blocks spam notifications directed towards the device by applying IP-based filters according to a database of well-known mobile spamming server IPs. However, it is known that at times, it is the apps themselves that generate locally on the device the notifications. As soon as the user taps on them, they start connecting to spam servers. While the CDroid-Server will catch these connections and possibly block them (as they target IPs of spam servers), for now, these connections are not stopped on the device. As future work we plan to add a lightweight component on the device side, that blocks directly these types of connections. It would filter according to a small local database recurrently updated by the CDroid-Server.

Lastly, Android does not have a standard way for setting a global proxy. The issue opened in code.google.com[11] regarding this problem is starred by 4329 people, and Google has not provided a real solution yet. Approaches that rely on the hidden activity `com.android.settings/.ProxySelector` provided by Android, or on setting the proxy in the APN Mobile Network settings (for 3G connections only) do not work. In our implementation we solve the problem by including in the CDroid-Device component the redsocks transparent proxy[12]. We then make use of iptables to forward the cellular traffic to the redsocks proxy, that, in sequence, tunnels the connections towards CDroid-Server. However, we recognize that this solution, depending on user activity, might induce overhead on the device—for example bottlenecks due to many connections going through redsocks. Nonetheless, we did not observe this from our experimental results. In addition, we believe that by solving this problem by natively including a way to set a global proxy for Android OS, CDroid-like systems would benefit even more from the mobile-cloud paradigm.

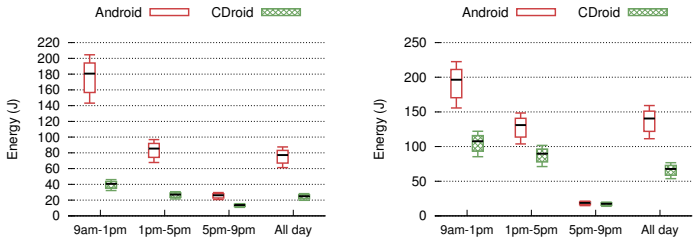
VI. Conclusions

In this work we design, build and implement CDroid, a system that aims at increasing the energy gains of offloading of computation-intensive apps, and making it possible to save energy for communication-intensive apps. CDroid tunnels all the mobile Internet data traffic through the cloud-side of the system so to handle it more parsimoniously and to provide new and innovative security features like mobile advertisement blocking, anti-phishing filtering, and privacy protection. We evaluate CDroid through a real testbed, lasted 2 weeks, of 5 Android phones (primary phones of 5 participants in the experiment) and relative Amazon EC2 cloud-side components, and

[10]http://www.infosecurity-magazine.com/view/21342/fortinet-expert-warns-on-mobile-malware-that-encrypts-its-data-streams/

[11]https://code.google.com/p/android/issues/detail?id=1273

[12]http://darkk.net.ru/redsocks/



(a) 9gag. (b) Facebook.

Fig. 7: Energy spent vs day-time interval: Average, minimum and maximum values, and 25^{th} and 75^{th} quartiles.

through experiments done, ad-hoc, in our lab. Our experiments show that, with CDroid, the phones consume less cellular data traffic (62% less), as well as less energy (2 times less to sync with the cloud-side for offloading purposes, and 2.54 times less energy waisted to keep the 3G radio in he FACH half-power mode). This enhances the usability of the device by making it load faster heavy pages and suffering less low-latency 3G network bandwidth due to peak-hours. Most importantly, the optimized sync component of CDroid enables, at the cost of 2.5% of the daily battery, offloading of computation-intensive apps when all other offloading frameworks fail to do so. Our experiment with a Face Detection app showed that the energy consumption can be as low as 10–times less.

REFERENCES

[1] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proc. MobiSys '10*.

[2] B. G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proc. EuroSys '11*.

[3] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading." in *Proc. IEEE INFOCOM 2012*.

[4] A. Saarinen, M. Siekkinen, Y. Xiao, J. Nurminen, M. Kemppainen, and P. Hui, "Can offloading save energy for popular apps?" in *Proc. ACM MobiArch '12*.

[5] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. Singh, "Who killed my battery?: analyzing mobile browser energy consumption," in *Proc. WWW '12*.

[6] T. Watson, "Application design for wireless computing," in *Proc. HotMobile '94*.

[7] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson, "Informed mobile prefetching," in *Proc. ACM MobiSys '12*.

[8] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE*, vol. 8, no. 4, 2009.

[9] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa, "To Offload or Not to Offload? The Bandwidth and Energy Costs of Mobile Cloud Computing," in *Proc. IEEE INFOCOM 2013*.

[10] R. Singh, P. Bhargava, and S. S. Kain, "Cell phone cloning: a perspective on gsm security," *Ubiquity*, vol. 2007, no. July, Jul. 2007.

[11] 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 *Proc. OSDI '10*.

[12] J. Bickford, H. A. L.-Cavilla, A. Varshavsky, V. Ganapathy, and L. Iftode, "Security versus energy tradeoffs in host-based mobile malware detection," in *Proc. ACM MobiSys '11*.

[13] J. Cheng, S. Wong, H. Yang, and S. Lu, "Smartsiren: virus detection and alert for smartphones," in *Proc. ACM MobiSys '07*.

[14] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian, "Virtualized in-cloud security services for mobile devices," in *Proc. MobiVirt '08*.

[15] C. Jarabek, D. Barrera, and J. Aycock, "Thinav: Truly lightweight mobile cloud-based anti-malware," in *Proc. ACSAC '12*.

[16] I. Burguera, U. Zurutuza, and S. N.-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proc. ACM SPSM '11*.

[17] G. Portokalidis, P. Homburg, K. Anagnostakis, and H.Bos, "Paranoid Android: versatile protection for smartphones," in *Proc. ACSAC '10*.

[18] L. Subramanian, G. Q. M. Jr, and P. Stephanow, "An architecture to provide cloud based security services for smartphones," in *WWRF '11*.

[19] M. V. Barbera, S. Kosta, A. Mei, V. C. Perta, and J. Stefa, "CDroid: Towards a Cloud-Integrated Mobile Operating System," in *Proc. IEEE INFOCOM 2013*.

[20] S. Kosta, V. C. Perta, J. Stefa, P. Hui, and A. Mei, "Clone2Clone (C2C): Peer-to-Peer Networking of Smartphones on the Cloud," in *Proc. USENIX HotCloud 2013*.

[21] A. Gaeta, S. Kosta, J. Stefa, and A. Mei, "StreamSmart: P2P Video Streaming for Smartphones Through The Cloud," in *Proc. IEEE SECON 2013*.

[22] S. Kosta, V. C. Perta, J. Stefa, P. Hui, and A. Mei, "CloneDoc: Exploiting the Cloud to Leverage Secure Group Collaboration Mechanisms for Smartphones," in *Proc. IEEE INFOCOM 2013*.

[23] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow, "Sociablesense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing," in *Proc. Mobicom '11*.

[24] M. V. Barbera, S. Kosta, J. Stefa, P. Hui, and A. Mei, "CloudShield: Efficient anti-malware smartphone patching with a P2P network on the cloud," in *Proc. IEEE P2P 2012*.

[25] C. A. Ardagna, M. Conti, M. Leone, and J. Stefa, "Preserving smartphone users' anonymity in cloudy days," in *Proc. of 22nd International Conference on Computer Communications and Networks (ICCCN)*, 2013.

[26] A. Mei and J. Stefa, "Give2Get: Forwarding in Social Mobile Wireless Networks of Selfish Individuals," in *Proc. IEEE ICDCS 2010*.

[27] A. Mei, G. Morabito, P. Santi, and J. Stefa, "Social-Aware Stateless Forwarding in Pocket Switched Networks," in *Proc. IEEE INFOCOM '11*.

[28] A. Mei and J. Stefa, "Give2Get: Forwarding in Social Mobile Wireless Networks of Selfish Individuals," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 4, pp. 569–582, 2012.

[29] M. Barbera, J. Stefa, A. Viana, M. D. Amorim, and M. Boc, "VIP Delegation: Enabling VIPs to Offload Data in Wireless Social Mobile Networks," in *Proc. IEEE DCOSS 2011*.

[30] A. Mei, G. Morabito, P. Santi, and J. Stefa, "Social-Aware Stateless Routing in Pocket Switched Networks," *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[31] B. Aggarwal, P. Chitnis, A. Dey, K. Jain, V. Navda, N. V. Padmanabhan, R. Ramjee, A. Schulman, and N. Spring, "Stratus: energy-efficient mobile communication using cloud support," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, 2011.

[32] I. Papapanagiotou, E. M. Nahum, and V. Pappas, "Smartphones vs. laptops: comparing web browsing behavior and the implications for caching," in *Proc. ACM SIGMETRICS/PERFORMANCE '12*.

[33] Z. Xu and S. Zhu, "Abusing notification services on smartphones for phishing and spamming," in *Proc. WOOT '12*.

[34] A. P. Felt and D. Wagner, "Phishing on mobile devices," in *Proc. W2SP '11*.

[35] V. Dave, S. Guha, and Y. Zhang, "Measuring and fingerprinting click-spam in ad networks," in *Proc. ACM SIGCOMM '12*.

[36] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Profiledroid: multi-layer profiling of android applications," in *Proc. ACM MobiCom '12*.

[37] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "Characterizing radio resource allocation for 3G networks," in *Proc. ACM IMC '10*.

[38] G. Ateniese, K. Fu, M. Green, and S. Hohenberger, "Improved proxy re-encryption schemes with applications to secure distributed storage," *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 1, Feb. 2006.