Mobile Malware Detection Based on Energy Fingerprints — A Dead End?

Johannes Hoffmann, Stephan Neumann, Thorsten Holz

Horst Görtz Institute (HGI), Ruhr-University Bochum, Germany
{firstname.lastname}@rub.de

Abstract. With the ever rising amount and quality of malicious software for mobile phones, multiple ways to detect such threats are desirable. Next to classical approaches such as dynamic and static analysis, the idea of detecting malicious activities based on the energy consumption introduced by them was recently proposed by several researchers. The key idea behind this kind of detection is the fact that each activity performed on a battery powered device drains a certain amount of energy from it. This implies that measuring the energy consumption may reveal unwanted and possibly malicious software running next to genuine applications on such a device: if the normal energy consumption is known for a device, additional used up energy should be detectable.
In this paper, we evaluate whether such an approach is indeed feasible for modern smartphones and argue that results presented in prior work are not applicable to such devices. By studying the typical energy consumption of different aspects of common Android phones, we show that it varies quite a lot in practice. Furthermore, empirical tests with both artificial and real-world malware indicate that the additional power consumed by such apps is too small to be detectable with the mean error rates of state-of-the art measurement tools.

1 Introduction

In the last years, smartphone sales began to rise significantly [3] and also the number of malicious software for these devices grew [4,21]. As a result, several techniques to analyze smartphone applications emerged with the goal to detect and warn users of unwanted software. Most solutions are based on classic techniques known from the PC area, such as dynamic and static analyses (*e. g.*, [8,7,10,22]). Based on the fact that mobile phones are powered by a battery and the insight that every performed action drains a specific amount of energy from that battery, the idea came up to measure the consumed energy and to deduce from that data whether any unwanted (malicious) activities occurred, possibly hidden from the user [12,13]. The developed tools use the system API or additional external devices to obtain information about the battery status, running applications, actions performed by the user (if any), and calculate the normal amount of energy a clean device should consume under such circumstances. This model is then used in the detection phase to compare live measurement data against it in order to detect additional activities. Such a method could—at least in theory—detect software that was loaded onto the device or applications that suddenly behave in a different way.

The proposed prototypes [12,13] were implemented and tested on feature phones with a limited amount of additional installable (third party) applications compared to

the "application markets" of today's smartphones. Furthermore, the devices themselves were equipped with considerably less features and sensors, such as an accelerometer, GPS, WiFi, large touchscreens, or a full-blown browser. Compared to a modern smartphone, these feature phones offer less possibilities to a user.

Throughout this paper, we attempt to verify or disprove the possibility to detect malware on modern smartphones based on their energy consumption. We use a specialized tool named *PowerTutor* [20] to measure the energy consumption of several potentially malicious activities in both short and long time test scenarios. We evaluate the energy consumption for each action and the energy consumption for the complete device based on the reports provided by *PowerTutor*. Our short time tests aim to get an idea of the measurement possibilities for a short time duration (5 minutes) and the long time tests (1 hour) evaluate what is possible in scenarios that can be found on smartphones used every day. We measure the impact of classic malicious activities such as stealing personal data or abusing the short message service (SMS) next to artificial ones like draining the battery as fast as possible in order to commit some kind of denial-of-service attack. We implement our own proof-of-concept malware that accomplishes our malicious tasks and we validate our findings with two real-world malware samples.

Our main contribution is the evaluation of a method to detect malicious software that was conducted in the first place on "old" feature phones rather than on modern smartphones. We argue that the proposed methods do not hold in practice anymore and study in detail how a modern Android phone consumes power. We show that the energy needed to perform relevant malicious activities, such as stealing private data, is too small to be detectable with the mean error rates of state-of-the art measurement tools.

2 Related Work

Since we want to (dis)prove that malware detection is possible on a modern smartphone by measuring its power consumption, we first discuss related work in this field.

Kim *et al.* introduced the idea of detecting malicious software based on its power consumption [12]. They built a prototype for phones running Windows Mobile 5.0 that works with power signatures. These signatures are based on the power consumption of a program rather than its code or exhibited behavior. In order to be useful to the enduser, a signature database has to be available. This circumstance does not allow the detection of new and unknown malware, as no signature is available.

Another tool for Symbian based phones was proposed by Liu *et al.* [13]. Their tool, called *VirusMeter*, works without any signatures but on heuristics. In a first step, the user's behavior and the corresponding power consumption on a clean system is profiled. Then, in a second step, the actual used energy is compared against the learned profile and if a certain threshold is reached, the systems alerts the user that additional (maybe malicious) activities have been performed on the phone. Throughout this paper, we perform similar tests not on features phones but on modern Android smartphones and evaluate to what extend malicious activities can be detected (if any).

Work by Dixon *et al.* shows that the location has a huge impact on the user's activities [5]. Leveraging this information, the average power consumption for different locations can be computed that could then be used to detect anomalies in the power

signature for these locations if, *e. g.*, malware performs additional operations next to the expected power consumption introduced by a user. A study performed by Balasubramanian *et al.* [2] analyzed the tail energy overhead introduced by transfers over the wireless connections offered by smartphones. Although they measured the used energy for different connection types, they focused on the amount of energy that can be saved if a special protocol is used by applications that make use of wireless connections.

Dong *et al.* propose *Sesame*, a tool that is able to generate a power model for smartphones and notebooks and the underlying hardware, battery, usage etc. by itself without external tools [6]. They argue that factory-built models are unlikely to provide accurate values for different scenarios, such as different hardware or usage patterns.

Since all such tools need to measure the used energy in one or another way, work related to this task is also relevant for us. The first tool, called *PowerTutor* [20], was designed to provide a precise report of energy spent on a smartphone. This report includes the power consumption of sole devices such as the NIC or the display. In order to provide a very detailed power model for an analyzed application, a power model for the used mobile device has to be calculated in the first place. This model was generated with the help of specialized hardware that precisely measured the power consumption of the device under certain circumstances. Since these models are bound to the device, accurate results with a claimed long-term error rate of less than 2.5% for an application's lifespan can only be provided if *PowerTutor* runs on such a "calibrated" device. *PowerTutor* runs on Android, requires no changes to the operating system and the Android framework, and its source code is freely available.

Next to *PowerTutor*, a tool called *eprof* was introduced to measure the power consumption of a given app on Windows Mobile and Android smartphones [16]. It is also able to provide a breakdown of the power consumption of sole methods inside applications. This is possible because *eprof* works on the system call level: all I/O operations that consume energy from internal devices are realized through system calls performed by the application, *e. g.*, sending a packet over the mobile Internet connection through the GPS modem. This enables a precise measurement of the energy spent for an application in question. This measurement method is different compared to the utilization-based one performed by *PowerTutor*. The authors of *eprof* claim an error rate of under 6% for all tested applications in contrast to an error rate of 3–50% for utilization-based methods. Furthermore, *eprof* can be used to measure which application components use what amount of energy [15]. The tool is not available and the authors describe changes to the OS kernel, the OS/Android framework, and the analyzed application itself.

Yoon *et al.* recently proposed another tool named *AppScope* [19] to measure the energy consumption on Android smartphones. Their monitoring software is able to estimate the energy consumption on a per app basis in a similar way as *PowerTutor* by making use of a kernel module that hooks and reports certain events on the syscall level and by using a linear power model. The error rate ranges from 0.9–7.5% depending on the tested software as long as no GPU intense tasks are performed. For games like Angry Birds it raises up to 14.7%.

All three tools can interfere the current power consumption of an app at whole or access to some component in detail from some previously generated power model. The subsystems itself, *e. g.*, the WiFi device or its driver, do not provide such information.

3 Measurement Setup

To measure accurate power consumption traces for several use cases on a modern smartphone, we first have to chose a stable setup under which all studies are performed. Furthermore, we need some way to actually generate accurate power measurements and we need a tool that performs defined actions that consume power.

Our tool of choice to measure the power consumption is *PowerTutor* [20], which was already introduced in the last section. Having access to *PowerTutor*'s sources, we modified it slightly such that it generates verbose log files which we used for our calculations throughout this paper. Since we want to verify if a software-based detection mechanism is capable of detecting additionally installed malware on a smartphone, we cannot make use of any hardware-assisted measurement mechanisms. Such additional devices (note that the phone itself is not capable of doing this with the exception of reporting an approximate battery charge level and voltage) would severely reduce the user acceptance to perform such measurements at all. Since end users are the target of such a software as they shall be protected from malicious software, it should be a purely software based solution as one would expect from traditional AV software products. We chose *PowerTutor* over *eprof* because we have access to the tool, the mean error rate is comparable, and we are able to generate good measurement results despite using a utilization-based measurement method since we have control over the test system (*i. e.*, we can control how much parallel interaction occur, see Section 4 for more details).

We now describe our software which we used for our test cases and explain the choice of our used smartphones.

3.1 Android Application

We now describe how we perform the power consumption measurements of different smartphone features. Since the main contribution of this paper is to (dis)prove the possibility to detect malicious software due to it's power consumption, we wrote a software that is able to run artificial tests of relevant functions that actual Android malware exhibits. While our test malware performs these actions, the power consumption is measured by *PowerTutor*.

Our proof-of-concept malware is able to perform the following functions in order to evaluate what features or combinations of features are detectable. It can send and receive SMS; make use of the location API; access content providers, *e. g.*, contacts and SMS database; send arbitrary (encrypted) data over the network; access serial numbers, *e. g.*, the IMEI; record audio; set Android wake locks to control the power states of the CPU and the screen; and run in an endless loop to put a heavy burden on the CPU.

These features are typically (more or less) used by malicious applications once they are installed, with the exception of the last one. Nevertheless, a malware that aims to disrupt operational time of the smartphone is easily imaginable. The measurement results for these functions or a combination thereof are later evaluated in order to see whether such activities are detectable by the amount of consumed power, similar to the malware tests conducted by *VirusMeter* [13].

Our software is written in Java and is installed like any other Android application. To be able to perform the described actions, all required Android permissions are requested

in the application's Manifest file. It basically consists of a control program that initiates an action over the network and a service which performs it. Actions can be run once, repeated in an interval, delayed and so on. This scheduling is performed with the help of the Android AlarmManager. All actions are performed by the service and are therefore performed without any GUI elements. This is crucial for the measurement step, as *PowerTutor* accounts the power consumption of GUI elements to the appropriate app. They influence the displays power consumption for OLED displays and, additionally, foreground processes have a higher priority than background processes within Android. The power consumption of this test malware will be referred as "MW" in all tables.

3.2 Test Devices

We performed most tests with a HTC Nexus One smartphone. The reason for this is that this phone was explicitly tested and used by the *PowerTutor* developers, saving us from calculating our own power model for the smartphone. They used three different phones, but the Nexus One is the newest one and is upgradeable to a recent Android version (Android 2.3.6). Having a rooted phone also enables *PowerTutor* to calculate a more precise power consumption for the built-in OLED display which depends on the visible pixel colors. By using this phone we believe we get the most accurate measurements out of *PowerTutor*. All tests are performed by this phone unless stated otherwise.

We additionally performed some tests with a Samsung Galaxy Nexus phone in order to validate our results. This is the latest Android developer phone by the time of writing and runs Android version 4.0. The phone is also equipped with an OLED display, albeit with a newer version being called "HD Super AMOLED", next to some additional sensors and it is used for validation purposes (although *PowerTutor* measurements might be less accurate due to a missing calibration). The phone's remaining battery capacity and its runtime can still be used to compare the results with those of the Nexus One.

Both phones have been equipped with new and formerly unused batteries in order to ensure maximum battery lifetimes. Note that our setup suffers from the same problems all such systems have, *e. g.*, the reported battery capacity and voltage may change a lot due to different parameters [14].

4 Short Time Tests

In order to determine whether malicious software is detectable on a phone with the help of power signatures, we first need to know the power requirements of several soft- and hardware components. To obtain an overview, we first conducted short time tests to measure which features consume what amount of battery capacity for later comparisons. First, all tests were run with the same basic settings. The hardware GPS module is activated, but not used. The display brightness is set to a fixed value of 130/255 and it switches off after 30 seconds of inactivity. The standard live wallpaper is active on the home screen but no synchronization, background data, mail fetching, or widgets are active. Internet connectivity is either provided by WiFi or by 3G, depending on the test. Additionally, the OS is freshly installed and only a few additional applications are installed: *PowerTutor* to be able to perform our measurements; *MyPhoneExplorer* to

easily access logged data from a PC; *K-9 Mail* for email fetching; and our own proof-of-concept malware for our evaluations. All tests are repeated six times in a row for 5 minutes from which the arithmetic median of the consumed energy is calculated. During this time, no additional interaction with the phone occurs. Note that such measurements do not represent a valid usage pattern in any case, but they enable us to determine the power consumption of basic phone features.

For all following tests, the same usage pattern is used. When the phone is fully charged and set to an initial state, *PowerTutor* is started and directly put in the background such that the home screen with the live wallpaper and the launcher is visible. No further input will occur in the next 5 minutes which causes the screen to be turned off after 30 seconds. As long as nothing is noted, a test does not deviate from this pattern.

In the following, we calculate the amount of used energy in mW and its *coefficient of variation* (CV) for several power consumers or the whole system, respectively. First, the CV is calculated for an idling phone (see next paragraph) and this defines the average percentage of deviating consumed energy during a given time interval. In other words, the CV for an idling phone describes the average amount of *noise* that is introduced by all components. If any action consumes less energy than the noise rate (*i. e.*., amount of energy described by the CV for an idling phone), it is not measurable with a single measurement. We could of course measure the power demands of such consumers if we would perform many measurements of the same consumer and would calculate the noise out of the results. A detection engine that works with power signatures does not have this kind of luxury, as it has to pinpoint malicious behavior as soon as possible. If many measurements must occur in the first place, a malicious software could already have easily performed its payload undetected. If the additionally consumed power of some activity is given in later tests in a table (referred as "Rise" in the corresponding column), it will be shown in bold letters if its value is above the CV of an idling phone (WiFi or 3G), meaning the measured action has a higher energy consumption than the average noise ratio of an idling phone. Such a component could be detected by a power signature.

Tables with measurement results will also often contain a column labeled "Total Cons." that depicts the total consumed energy during the test as reported by *PowerTutor*. Unexpected Framework and OS activities triggered during a test might introduce additional noise, which can be seen in this column. The impact is of course higher for the conducted short time test. If this value is higher than the total consumption of the initial tests (see next paragraph) plus the noise ratio (CV value), it will also be written in bold letters. This value does not related to the "Rise" column but describes unexpected introduced noise in addition to any used energy throughout the test. Note that if the value is written in bold letters, it does not imply that it can be detected in a reliable way. It's value must be significant higher than the CV value, which describes the average noise. False positives are possible here, one must carefully check the size of the value. Higher differences to the initial total consumption mean potentially less false positives.

Since *PowerTutor* is unable to measure the power consumption of the GSM modem, we cannot provide any measurement about it's usage. Still, we performed a test that includes the sending of short messages in Section 5.5. In order to overcome the drawbacks of the utilization-based measurement method of *PowerTutor*, we strictly control all ad-

ditionally running applications (next to the running OS applications) and their access to any device. Doing this mitigates the problem of accounting the used energy to programs running in parallel.

4.1 Initial Tests

We start our evaluation with tests in which we measure the power consumption of several components such as the display as well as the influence of running software. These initial tests define a basis for later tests which are compared with the initial ones. Knowing the minimum amount of energy a smartphone requires in certain circumstances is crucial for the detection of additional malicious activities.

Data Connectivity. This test evaluates the differences between a WiFi and a 3G connection on an otherwise idling phone. Table 1 shows how much power their usage consumes if the connection is only established, but no data is actually transferred.

Table 1. Short time initial tests for a 5 minute period. Average power consumption for wireless connections.

Connection	Consumption	CV
WiFi (always on)	51.17 mW	0.87%
WiFi (if screen is on)	51.26 mW	1.14%
3G	68.47 mW	9.49%

The WiFi connection can automatically be turned off if the smartphone's screen blanks in order to safe energy. Using this feature saves no energy in this short time span compared to being always on. On average, the smartphone consumes $51mW$ with an enabled WiFi connection with a low CV. Remarkable among these numbers is that the smartphone consumes 34% less energy using WiFi instead of 3G. Additionally, the CV is much higher for the 3G connection, with measured absolute numbers from 47.77 to $75.85mW$. This is likely caused by different bitrates and link qualities (GPRS, EDGE, UMTS, HSDPA) depending on the coverage area and the signal strength at the time the test was conducted. It may even change for the same location at different times. For the rest of this section, we compare the results of the other tests against the values from this test where WiFi is always on and from the 3G case.

Background Processes. To get an idea of the energy consumption of the applications running on a smartphone, we used *PowerTutor* to measure the energy usage of the automatically started preinstalled applications after each restart. The results can be found in Table 2. What can be seen in this table is the fact that the foreground application—which is the Launcher—consumes the largest amount of power. As it manages the live wallpaper, *PowerTutor* will add the power consumption used by the OLED display to show the wallpaper to the Launcher instead of to the Wallpaper application. However, the same is not true for its CPU consumption. *PowerTutor* itself consumes about 3.0% compared to the overall consumption, but this value is calculated out in all further tests. All other values are left alone, as they present characteristics of the base system, such as *Android Services* (by this term we mean several Android OS processes). Again, no synchronization or other activities occurred during the short time tests, they will be evaluated in Section 5.

Brightness. The brightness of the display scales between 0 and 255, while higher numbers represent a brighter display. The lowest user selectable value is 20. The value can be set manually or by the system itself, which can determine the brightness of the phone's surroundings with a light sensor.

We measured the power consumption for different values and the results can be found in Table 3. During this test, the display was never turned off which will prevent the phone from entering the sleep state. Additionally, the WiFi connection was enabled. With these settings, the battery lasts for about 10 hours with a difference of 2 hours between the darkest and the brightest setting.

Table 2. Exemplary power consumption of different apps after system start for a 5 minute interval. Values in mW (missing energy was consumed in unlisted components).

Application	OLED	CPU	WiFi	Total
Desktopclock	0.00	0.03	0.00	0.03
MyPhoneExplorer	0.00	0.00	0.00	0.05
Gallery3D	0.00	0.07	0.00	0.07
Android Services	0.00	0.12	0.11	0.23
Maps	0.00	0.00	0.34	0.92
PowerTutor	0.00	1.87	0.00	1.87
Wallpaper	0.00	4.31	0.00	4.31
Launcher	39.79	0.01	0.00	39.80

What can be seen is that the brighter the display is, the smaller the CV gets. This is caused by the relative high amount of power which is consumed by the display, even for dark settings. All other energy consumers such as background processes quickly loose their significance in contrast to this huge energy consumer, compared to the numbers from Table 2. These results show that the display's energy demand plays a big role for the smartphone's runtime.

Table 3. Average power consumption for different brightness levels.

Setting	Consumption	VC
Dark (20)	445.89mW	2.50%
Auto (standard)	462.20mW	1.73%
Medium (130)	494.61mW	1.13%
Bright (255)	550.70mW	1.01%

4.2 Energy Greedy Functions

This section deals with software which aims to draw as much power as possible by various means. Such activities can be seen as a kind of DOS attack against the smartphone, as it is unable to operate with a depleted battery.

Sleep Mode. We first determine how much energy gets consumed by the CPU if it is not allowed to reach its energy saving sleep modes. It is easy to do this in Android, as one only has to set a partial wake lock. This will cause the screen to be turned off after the normal timeout but the CPU keeps running. This feature is normally used for tasks which run periodically in the background and that shall not be interrupted when the phone would otherwise enter its sleep mode.

Such a setting will consume $81.50mW$ in total and causes a raise of 59.27% in terms of used battery power. Although *PowerTutor* does not detect that our software sets the wake lock, the Android system does and marks it correspondingly in the "battery settings". Note that this can be easily detected by the user. However, setting a wake lock is not a feature that has to be used to hide malicious activities in the background—at least not to such an extend. Such a setting, whether used by mistake or on purpose, can easily be detected by any program monitoring the power consumption.

CPU Burn. The last test revealed a high rise in energy consumption if the CPU keeps running all the time. This test will determine how big the impact is when the CPU will not only run all the time, but also has to crunch some numbers. Table 4 shows the results of the following two tests. In the first one, the CPU is allowed to sleep when the screen turns off. This way, the CPU will only have a maximum load when the phone is active. In the second test the CPU is disallowed to enter it's sleep state when the screen turns off. During both tests, the program calls `Math.sqrt()` in a loop.

Table 4. Average power consumption for diff. power states.

Function	MW Cons.	Total Cons.	Rise
Sleepmode allowed	54.02mW	110.29mW	**105.57**%
Sleepmode disallowed	518.84mW	602.92mW	**1,013.95**%

Both tests put a heavy burden on the phone's runtime. While the first test "only" consumes about double the energy than it would normally do, the second test clearly shows that a malicious program can totally disrupt the battery lifetime. With a raise of over 1,000% in energy consumption, the battery would only last for about 8 hours even though the screen turns off. But again, the Android system detects that our application wastes so much energy and the user can take countermeasures. Additionally, the phone gets quite hot under such load. Some AV program could also easily detect such (mis)use and alert the user.

4.3 Location API

Next, we evaluate how much energy is consumed while the location API is used. We cover the case where the last known position is reused and when an accurate GPS position is requested. Since location data represents a very sensitive piece of information, we measure the energy required to steal it from the phone.

Last Known Position. In this first test, our software will only use the last known position (LKP) which is returned by the Android API. Because no new position is determined, the energy consumption is expected to be low. To mimic actual malware, the returned coordinates are wrapped in an XML structure and sent over the network through the WiFi or 3G connection. Table 5 shows the results; the position is retrieved only once during the test. As expected, the power consumption is really low if the data is only retrieved and not forwarded at all (WiFi is enabled, though). If it is sent over the WiFi connection, the consumed energy raises a bit, but is still very low with a rise of 0.25% over the normal consumption. This is basically only the amount of energy needed to use the WiFi interface, which is evaluated in more detail in Section 4.4.

If position data (LKP) is sent over the 3G connection, 2.36% more energy is consumed in contrast to the CV for an idling phone with an established 3G connection, cf. Section 4.1. In Section 5, we evaluate whether the added consumption in the 3G case is still measurable in real life scenarios and would therefore be detectable.

Determine GPS Location. This test makes use of the current GPS position which has to be determined by the hardware GPS module. It is said that it consumes a lot of power; we will see if this accusation is correct or not. The position is again retrieved only once

by our software and sent over the network encapsulated in XML format. The results are also presented in Table 5.

What can be seen is that our software consumes more than $7mW$ additional power when the GPS module gets active. We have to note that *PowerTutor* measures the GPS module's power consumption separately, but we added it to our malware consumption as it is the sole program using it. It does not matter whether the data is sent over the network or not in order to introduce a huge gain in consumed energy. If sent over the 3G connection, a rise of 17.54% is measured, which is clearly above the noise ratio even for the 3G connection.

Table 5. Average power consumption for accessing the location API. LKP = Last known position.

Connection	Function	MW Cons.	Total Cons.	Rise
WiFi	LKP	0.017mW	**52.25mW**	0.03%
	LKP (sent)	0.126mW	51.39mW	0.25%
	GPS	7.91mW	**61.18mW**	**15.46**%
	GPS (sent)	7.97mW	**63.28mW**	**15.58**%
3G	LKP (sent)	8.111mW	**87.25mW**	**11.85**%
	GPS (sent)	12.01mW	**107.79mW**	**17.54**%

4.4 Data Heist

This section examines whether the acquisition and forwarding of (private) information raises the energy consumption to an extent that it is detectable. This is a common feature of current mobile malware [21].

Data Size. We first measure the impact of the file size of the data which is sent over the Internet connection. To get an idea of how much data is transferred, our malware sends data equivalent to the size of 1, 200 and 2,000 short messages encapsulated in a XML structure over TCP/IP. Table 6 lists the power consumption for both Internet connection types. As one can clearly see, more sent data consumes more energy. The higher consumption whilst using WiFi is more visible than for 3G, as this connection type implies less noise. Sending small quantities of data quickly puts the energy consumption over our threshold for this short duration with both connection types. In Section 5, we evaluate if this is still true for real world scenarios.

Table 6. Average power consumption for data transmission.

Connection	Function	MW Cons.	Total Cons.	Rise
WiFi	349 Bytes (1 SMS)	0.112mW	51.55mW	0.22%
	37.6kB (200 SMS)	1.182mW	**53.39mW**	**2.31**%
	365kB (2,000 SMS)	1.949mW	**54.73mW**	**3.81**%
3G	349 Bytes (1 SMS)	8.114mW	**99.15mW**	**11.85**%
	37.6kB (200 SMS)	8.161mW	**103.41mW**	**11.92**%
	365kB (2,000 SMS)	13.724mW	**86.95mW**	**20.39**%

We also tested whether the data source has some impact on the energy consumption. The results show that it does not matter if our data originates from some content provider, the SD card and so on. Only the amount of data matters.

Encryption. Some sophisticated malware might encrypt the sent data to hide its intention. As encryption of course uses CPU cycles, we are interested if this overhead is measurable. We performed the same measurements as above, but the data was additionally encrypted with AES in Counter Mode with PKCS5Padding and a random key. We

have measured that our malware consumes $1.19mW$ of energy to encrypt 37.6kB of data which is sent over the WiFi connection. Compared to our last test with data of the same size, almost the same amount of energy is consumed: a rise of 2.33% instead of 2.31% is measured, which lets us conclude that the encryption only consumes 0.02% more energy. Rather than using the 3G interface, we only performed the test with the WiFi interface as the results are more clean due to lower noise. Additional encryption is therefore not measurable as it is indistinguishable from noise, at least with a cipher such as AES.

5 Long Time Tests

This section covers long time tests which evaluate if and to what extend the aforementioned features are measurable by means of their power consumption under two more realistic scenarios. The first scenario (A) covers a real world scenario where the smartphone is heavily used, while the other (B) covers a scenario with light usage. The details of the two scenarios can be found in Tables 7 and 8. Both scenarios are run for 1 hour and repeated three times, resulting in a total duration of three hours for each test run.

Table 7. Joblist scenario A.

Minute	Job	Duration
5	1x write SMS	1 minute (160 characters)
10	1x send SMS	1 minute (160 characters)
20	Use Browser	5 minutes (4 site accesses)
25	Music	10 minutes (display off)
35	Facebook App	2 minutes
50	Angry Birds	5 minutes
55	1x E-Mail	1 minute (120 characters)

In order to simulate an average Jon Doe's smartphone, several Widgets were visible on the home screen (*Facebook*, *Twitter*, a clock, and a weather forecast) in scenario A. These were absent in scenario B, but both additionally made use of background mail fetching (POP3 with K-9 Mail, every 15 minutes) and synchronized the data with Google. GPS was enabled all the time and everything else was left at it's default setting.

Table 8. Joblist for scenario B.

Minute	Job	Duration
5	1x write SMS	1 minute (160 characters)
20	Use Browser	2 minutes (1 site access)
30	Music	3 minutes (display off)
40	1x E-Mail	1 minute (120 characters)

5.1 Initial Tests

In order to detect malicious activities, we again first need to know how much energy is consumed in both scenarios without them. Table 9 shows the four CV values which again represent our threshold values. Any action which consumes less energy than these values is indistinguishable from noise in the corresponding scenario.

The battery charge value is eye-catching. Although in scenario B the total energy consumption differs by approximate 50%, the charge level is even higher for a more depleted battery. This is a strong indicator that the user cannot trust the battery charge value by any means and that it should only be considered as a very vague value.

As this test includes normal user behavior such as Web browsing, the power consumptions depends a lot on the actual user input. For example, when and how long the Web browser is used is defined and always the same in all tests, but the actual power consumptions is influenced a lot by the actual accessed Web sites. The OLED display might consume more or less energy on one website as it would displaying another one. The same is true for the browser process. How many and what scripts are executed, is the browser's geolocation API accessed? During the test the same websites were visited, but the content changed over time which at least might influenced the OLED display to a certain extent.

Table 9. Long time initial tests (3 hour period).

Scenario	Function	Charge	Total Cons.	CV
A (heavy)	WiFi	63%	299.67mW	2.08%
	3G	48%	419.09mW	2.67%
B (light)	WiFi	77%	97.28mW	2.79%
	3G	78%	145.14mW	3.16%

As one could already see in the short tests, the 3G Internet connection uses more power than the WiFi connection. The CV for the tests with 3G connections is much lower as in the short tests because there are a lot more actions performed than just keeping this connection up, which reduces the noise introduced by this consumer. The same is true in the opposing way for the WiFi connection, as the CV goes up for these scenarios.

Table 10. Exemplary power consumption of different apps (scenario A). Values in mW (missing energy was consumed by unlisted components).

Application	OLED	CPU	WiFi	Total
PowerTutor	0.25	3.45	0.00	3.70
K9-Mail	13.87	0.60	0.48	14.95
MMS Application	21.44	1.04	0.00	22.48
Music	0.00	0.34	0.00	26.28
Launcher	28.05	1.46	0.00	29.51
Facebook	26.98	12.47	8.40	47.85
Angry Birds	53.89	9.71	1.51	68.10
Browser	39.79	14.28	1.01	78.01

Table 10 provides an overview of the power consumption of several apps as we did in the last section. As one would expect, the game *Angry Birds* and the Web browser consume a lot of energy. The values for the OLED display, the CPU, and the WiFi module also look sane and correlate to the provided applications functionality except for the *Facebook* app. The CPU consumption seems a bit high. The reason for this is unclear, but the app felt unresponsive on the old phone which might be caused by not well written code. The missing values for the apps total energy consumption are used by the GPS module, the speaker and other devices.

5.2 Energy Greedy Functions

In this test, we again stress the CPU to its maximum in both scenarios. Since we want to know how big the impact of such energy-greedy software is in contrast to all other apps, we disabled the sleep mode, meaning that the CPU and all apps keep running even when the display blanks. Table 11 shows the results. As one would expect, our malware consumes a lot of energy in both scenarios but most in scenario B, as it gets more CPU cycles in total because there is less concurrent interaction opposed to scenario A. This is also caused by the fact that Android prioritizes foreground apps. The energy consumption compared to Table 4 is a bit lower, as other software runs next to our malware. The values are not higher as one might wrongly expect because W is defined as one joule per second.

Under these circumstances, the smartphone's battery will last for approximately 8 hours in scenario A and 6.7 hours in scenario B. If the user does not know how to check which apps consume what amount of energy, this will vastly degrade the user's smartphone experience. Additionally, if the CPU is not the fastest, the user might feel some unresponsivenesses in some apps. Nevertheless, this behavior can be detected by AV software in both scenarios.

Table 11. Average power consumption with disabled sleepmode (WiFi only).

Scenario	MW Cons.	Total Cons.	Rise
A (heavy)	419.26mW	**764.53mW**	**139.91**%
B (light)	505.55mW	**645.82mW**	**519.69**%

5.3 Location API

In this section we test how much energy a "tracker app" consumes under what circumstances. If an app retrieves the last known location from the API, almost no energy is consumed. We therefore limit our tests to the case where our malware retrieves the GPS location. We chose four different intervals for each scenario and the location is always encapsulated in an XML structure and sent out through the WiFi interface.

Table 12. Average power consumption for stealing GPS position (WiFi only).

Scenario	Function	MW Cons.	Total Cons.	Rise
A (heavy)	5 minutes	5.32mW	**315.61mW**	1.78%
	15 minutes	2.88mW	**328.49mW**	0.96%
	30 minutes	2.56mW	304.88mW	0.85%
	60 minutes	0.87mW	292.97mW	0.29%
B (light)	5 minutes	6.11mW	**105.42mW**	**6.28**%
	15 minutes	2.24mW	**100.84mW**	2.30%
	30 minutes	1.73mW	**104.12mW**	1.78%
	60 minutes	0.94mW	**101.08mW**	0.97%

Table 12 shows the consumed energy for each test case. The results show that retrieving the location during the long time tests is less obtrusive compared to the short time tests. In scenario A, the added power consumption is indistinguishable from noise and in scenario B carefully set parameters are also indistinguishable (interval ≥ 15). Our location listener was updated at the set interval, but an additional parameter which sets the minimum distance from the last location which must be reached in order to get notified was set to 0. This means that our malware woke up at all interval times, even if the location did not change. One could be much more energy friendly if a minimum distance is set and/or if a passive location listener is used which only gets notified if some other app is performing a regular location request.

5.4 Data Heist

The short time tests revealed that even small quantities of data sent through either the WiFi or the 3G interface are detectable. This section examines if this is also true for real world scenarios. In both scenarios, 369kB are read and sent through each interface. Two different intervals were tested during which the data was sent. Table 13 shows the results for each test. It is clearly visible, that data heist from a spyware is not that easily detectable in a real world scenario. A well written malicious software that steals data could send approximately 35MB of data in small chunks in 3 hours without being detectable by its energy consumption. This amount decreases vastly for the 3G interface.

Data theft can—to some extent—be detectable by means of additionally used energy. This means that it gets detectable if, *e. g.*, many pictures or music files are copied. In contrast to that, theft of SMS databases or serial numbers such as the IMEI are unrecognizable.

Table 13. Average power consumption for data transmission.

Scenario	Function		MW Cons.	Total Cons.	Rise
A (heavy)	WiFi	5 min. (13MB)	2.01mW	295.71mW	0.67%
		1 min. (65MB)	11.42mW	**322.82mW**	**3.81**%
	3G	5 min. (13MB)	8.02mW	**450.65mW**	1.91%
		1 min. (65MB)	51.72mW	**538.74mW**	**12.34**%
B (light)	WiFi	5 min. (13MB)	2.14mW	**100.84mW**	2.20%
		1 min. (65MB)	6.11mW	**105.42mW**	**6.28**%
	3G	5 min. (13MB)	7.50mW	148.82mW	**5.17**%
		1 min. (65MB)	39.78mW	**197.78mW**	**27.41**%

5.5 Galaxy Nexus

Next to our test with *PowerTutor* on a HTC Nexus One, we also performed some tests with a Samsung Galaxy Nexus. Since *PowerTutor* is not fine tuned to this phone, we only use the provided battery charge level and the reported battery voltage by the tool (similar to the approaches presented in the literature [12,13]). This way we can determine what is possible without a sophisticated tool.

We performed three tests on the Galaxy Nexus. The first two are identical to the last two from the previous test: data is sent over the WiFi interface in two different intervals for our two scenarios. In the third test, our malware sends a short message every 5 minutes resulting in 36 messages over 3 hours. Unfortunately, *PowerTutor* is unable to measure the power consumption of the GSM modem. Therefore, this test was not performed on the Nexus One and cannot be compared to any previous measurements.

In order to obtain any information about the phone's power consumption, we began our evaluation with a measurement of the phone's energy demands for the two scenarios without any additional actions. We again call them *initial tests* and they are performed in the same way as mentioned before (3 hours in total). The results can be found in Table 14 and clearly tell one story: Without any sophisticated measurement of the actual consumed power, no predictions of any additional running malware can be made (at least for our chosen scenarios and tests). Each test ended up with a battery charge rate which was higher than that for the initial test. This should of course not be the case, as additional actions were performed. The reported voltage also does not correlate to our expectation that more energy is used and it should therefore be lower (the battery voltage decreases if depleted). Therefore, a user cannot trust the values displayed on the phone and so cannot any monitoring software.

Table 14. Battery charge level for the Galaxy Nexus after sending data and short messages (WiFi only).

Scenario	Function	Charge	Voltage
A (heavy)	Initial test	74%	3,812mV
	5 minutes/13MB	79%	3,887mV
	1 minute/65MB	76%	3,900mV
	36x SMS (every 5 minutes)	76%	3,845mV
B (light)	Initial test	90%	4,060mV
	5 minutes/13MB	91%	4,072mV
	1 minute/65MB	91%	4,023mV
	36x SMS (every 5 minutes)	90%	4,022mV

6 Validation with Real-World Malware

This section covers the energy demands of two malicious software samples named *Gone in 60 seconds* (GI60S) and *Superclean/DroidCleaner* (SC) that were found in the Google Play Store in September 2011 and January 2013. We have tested whether they are detectable in our test scenarios from the last section and validate our measurements for the Nexus One.

We now briefly explain what both samples do. GI60S is a not a malware per se, but mostly classified as such. Once it is installed, it sends the following data to some server: contacts, short messages, call history, and browser history. When finished, it will display a code that can be entered on the GI60S homepage which will enable the user to see all stolen data (messages are behind a paywall). In a last step, the software removes itself from the smartphone. In our case, 251kB of data got transferred. The name is based on the fact that all this is done in less than 60 seconds. SC promises to speed up the smartphone by freeing up memory. Additionally, it aims to infect a connected PC by downloading and storing files on the SD card which are possibly run by a Windows system if the smartphone is used as an external storage device. It also offers some bot functionality and is able to gather and forward a bunch of information about the infected device to the author. The author can also forward and delete arbitrary files, send and receive SMS, phish for passwords, and control some device settings. More detailed analysis reports are available on the Internet [18]. We wrote a small server for SC and tricked it into connecting to this one and not the original one (which was already down). This way we were able to control the bot and send commands to it in order to measure the consumed power. We used the functionality to download several files (images, PDF and music), SMS, and contacts next to retrieving all information about the phone. 22.46MB of data were transferred over WiFi to our server. Table 15 shows the results of our measurements.

Table 15. Verification with malware in controlled scenarios (WiFi only).

Scenario	MW	MW Cons.	Total Cons.	Rise
A (heavy)	GI60S	1.45mW	**311.51mW**	0.48%
	SC	4.06mW	296.87mW	1.35%
B (light)	GI60S	1.54mW	**103.35mW**	1.58%
	SC	**5.60mW**	**113.65mW**	**5.45%**

It can be seen that the energy consumption is similar to our test malware with the corresponding feature set. Therefore, our malware has a reasonable power consumption and the results should be comparable to other software performing similar tasks. This also means that both samples are in 3 out of 4 cases not detectable by its power consumption as our measurements reveal—they go down in the noise. The total power consumption is even lower than the initial one for the SC case and is only slightly above the CV for the initial consumption for both GI60S cases. Only the SC test in scenario B is detectable which is not astonishing, as

Table 16. Power Consumption during the "all day long tests". The CV is calculated from 8 time slices during that period lasting for 1 hour each.

Run	Application	Consumption	CV	Rise	Charge
1st day	Total	64.57mW	70.40%		40%
	GI60S	1.24mW		1.92%	
2nd day	Total	87.14mW	82.86%		56%
	GI60S	0.54mW		0.62%	

we copied a lot of data from the phone which raises the energy consumption a lot in the light usage scenario. Malware could act much less inconspicuous, but that was not our goal in this test.

Furthermore, we tested GI60S in an "all day long test" (i.e., the phone was "used normally" during an 8 hour period). During this time, GI60S was run once such that all data was stolen. This test was performed twice and the results can be found in Table 16. These show that the overall power consumption during an 8 hour period can greatly differ. The CV for the total consumption during a day (total runtime was divided into 8 slices lasting one hour each) is huge, with over 70%. This means, the power consumed during one hour of usage might be completely different from the last hour, depending on the actual usage pattern. Having such a high CV, it is almost impossible to detect anything based on a single power measurement. Even if very accurate and timely measurements with small intervals are available and the smartphone reports accurate battery levels, this would still be a tough job since the user has such a big influence and his actions are almost unpredictable resulting in a very high noise ratio. The solution proposed by Dixon [5] might lower the CV, but it seems unlikely that it will reach a usable value. We have not tested SC in this test since the results should be very similar.

7 Discussion

In this section, we evaluate our measurements and findings. We can boldly say that measuring power consumption on a smartphone in general is not an easy task. There are many parameters that influence the whole system and thus the energy demand and ultimately the smartphone's runtime. Let alone the fact that precise battery charge levels are very hard to measure and depend on a lot of different factors [1,17], it is even harder doing with software only. This fact is somehow mitigated as *PowerTutor* is a very specialized tool for this task and is adjusted for the used smartphone. We therefore deem its measurements as accurate enough for our purposes although it is not perfect.

We will now compare our results with the proposed solutions of *VirusMeter* [13]. The creation of power signatures would not be satisfactorily for us on a modern smartphone operating system: such a signature would contain the energy demands of the application in question under certain circumstances. If an app would suddenly act in a malicious way (*e. g.*, stealing private information) a monitor should detect these actions based on its power signature. In theory, this should work as all additional (malicious) actions will use additional energy which is measurable. In practice however, accurate measurements are hard to perform as discussed throughout this paper. This will yield to a certain error rate which we called "noise" in the previous sections. This noise describes the varying amount of energy which is consumed more or less for the same action(s) in the same amount of time. Even for a five minute interval, a noise ratio of 1% was measured. Despite the fact that we were able to control many settings on the smartphone during this time span, our measurements were not 100% accurate. Since we used a modern smartphone with a variety of features, this problems gets worse for larger intervals as more features kick in (*e. g.*, email fetching or synchronization). This leads to a noise ratio of up to 2.79% for long time tests. The fact that such a monitor should run on everyday smartphones, forces it to cope with such noise ratios.

Our measurements for the various test cases in Sections 4 and 5 show that such a power signature would not be accurate enough, as a lot of possible malicious activities can easily go by undetected compared to the measured amount of energy these actions cause. If such a signature would only work with the total consumed power of the smartphone, it will alert the user for a lot of these actions. But, if the total consumption is higher than the initial power consumption plus the CV value, this only means that the action required more energy than the average noise level. Many tests lead to values which are just a bit above this threshold which could lead to many false positives. Generating a good threshold is inherently hard, as the users' habits may change and even for the same user and for two consecutive days the CV is above 70% (see Table 16), which is completely unusable. Lowering the measurement interval could decrease the CV, but only to some extent as it heavily depends on actual user input in some cases, see Section 9 for an example. A detailed analysis of the smartphone usage of 250 users was conducted by Falaki *et al.* [9] and they also found out that even a single user can show very varying usage patterns. If the total consumption is not considered, an attacker could, *e. g.*, steal an amount as high as 35MB over 3 hours without being conspicuously. This is also true for a lot of other actions.

If one not only analyzes the energy consumption introduced by an application in total or even on a device basis (*e. g.*, WiFi), consumption patterns might occur. But these patterns still suffer from the introduced noise, as the power consumption is only interfered from a model that was previously generated (the phone does not provide power stats of sole devices). Having some kind of pattern which states that some app consumed x_1 *mW* during y_1 seconds in device z_1 and then x_2 *mW* during y_2 seconds in device z_2 and so on, one could use that as a signature. However, searching for that information in, *e. g.*, the syscall trace would also be enough because it was used to interfere these values in the first place.

Although such power signatures cannot detect the described activities, they still can detect some malicious ones. Amateurish written malware could be detected if too many features are used too aggressively, *e. g.*, determining the current position by GPS in a very short interval. What is easily detectable is energy-greedy malware which has the goal to disrupt the battery lifetime. But this clearly is not the normal behavior malware exhibits—most of them steal (private) data or send premium rate SMS.

This leads us back to *VirusMeter*: this approach makes use of predicted user behavior and their resulting energy demands. If the used energy (measured by the different battery charge levels) does not match the assumption, then something is wrong and an alert is generated. While the tools to measure events and power consumption clearly improved compared to the possibilities the authors of *VirusMeter* faced, we cannot verify their findings for a modern Android based smartphone. The noise ratio and the impact of interfering events is too big to get good and usable results (see, *e. g.*, Table 16). Even if all events and measurements are logged and some sophisticated heuristic performs the evaluation externally or on the smartphone itself if the battery is charging, malware can still hide below the noise level.

We believe the noise level is the biggest show stopper for such a detection approach. All other proposed tools such as *eprof* [15] and *AppScope* [19] have error rates, and therefore noise ratios, which are too high. Using some sophisticated power model will

not negate the small amount of additional energy (often below 2%, which is under the mean error rate for most tools and settings) that is needed to perform most malicious activities. We therefore opted to not generate our own model as it is unable to cope with such settings.

Even if malicious activities are detected by such means, most activities would already have finished and the damage would have been committed. Otherwise, no additional power would have been consumed in order to perform any detection. This assumption lets us further expect that such a system is not feasible in any satisfying manner as most of the relevant cases can only be detected when it is too late. Additionally, we believe that the false-positive and false-negative rate would be too high in practice, even if the system does not aim to prevent but only to detect malicious activities.

8 Limitations

In order to reach the goal of this paper—namely to evaluate whether the detection of malware running on a mobile phone is possible by measuring the power consumption of certain activities and devices—we need precise power measurements. We believe that *PowerTutor* is a good starting point on an adjusted device such as the Nexus One. Although the measurements are not perfect, we deem them accurate enough for our purposes. At least they are more accurate than the parameters used for *VirusMeter* [13]. Additionally, the mean error rate is comparable to other tools such as *Appscope* and *eprof*. One thing *PowerTutor* is unable to cope with is the power consumption of actions which make use of the GSM modem, such as the short message service. We were therefore unable to measure precise results for such activities. Another thing that is not reported in a good manner is the power consumption of the GPS device. *PowerTutor* can only report the consumption of the whole device, not the consumption of a specific "consumer". We therefore have to calculate an approximate value for its usage if more than one software is using it. *eprof* would be better suited for such a test case, as it is able to calculate the consumption for each app separately.

The authors of *VirusMeter* build a profile for the user in order to detect anomalies which we did not do. We refrained from doing so, as our measured numbers are either too close at our thresholds (CV) or too far away. Without reasonable results for the long time tests generating such a model is futile in our opinion regarding a low false-positive count. The user's activities are just too random for modern smartphones [9].

Additionally, our tests were mainly performed with one smartphone, the Nexus One. A second phone, the Galaxy Nexus, was only used in two test cases to get a feeling of how a monitoring software performs which does not have access to accurate results such as provided from *PowerTutor*. More tested devices would of course be favorable, but the Nexus One is the only device which is supported by *PowerTutor* and is still modern enough to actually perform meaningful tests with it. In fact, *AppScope* also only supports this phone. Furthermore, the results are not encouraging at all.

We tried to be as precise as possible during our tests. But since these tests were all performed by hand, there are certainly slight variations for each result. Automatic testing was not possible, so all the performed tests took a lot of time and patience.

9 Conclusion

Our results indicate that software-based approaches to measure the power consumption of an Android smartphone and to interfere from these results whether additional malicious activities occurred, is not satisfactory in most cases. The approach mainly fails due to the noise introduced into the system by unpredictable user and environment interactions, such as the reception rate or the delivered content of accessed websites. While a more precise power model could mitigate effects such as varying reception rates, it cannot calculate out the effects of many user interactions, *e. g.*, browser usage. This is at least true for our long time test results, which do not have optimal but comparatively real world settings. The short time tests indicate that some activities can be detected by such a system, but under settings seldom found on a smartphone that is regularly used.

We even go one step further and think that such a system is not feasible at all on a modern smartphone—at least with available measurement methods and normal use cases. Let alone the fact that the hardware parts have to provide very accurate values of consumed energy, the system still needs a very precise model of what the user usually does and how much energy these actions typically consume. We assume that such an anomaly detection would generate a lot of false positives, as normal users change their behavior quite often, depending on the actually installed apps and so on. Even if a precise profile would exist and the user would not change his habits too often, apps can be very dynamic in a way that a power profile for these apps cannot be precise at all. Just imagine what the browser is capable of (*e. g.*, complete Office suites are offered as a web application) and try to generate a power signature for its behavior.

We conclude that well written malicious software running on a modern smartphone can hardly be detected by means of additionally consumed energy as the noise ratio is too high. Only DoS attacks against the battery runtime and so called “energy bugs” [11] as well as certain activities performed under strictly given scenarios can be detected, which is not enough to be of great use for normal smartphone usage patterns.

As a last point we note that modern smartphones with modern operating systems such as Android are more or less a general purpose computer with a very small form factor. If such proposed systems would be usable as a malware detector, they should also work on regular notebooks or PCs. To the best of our knowledge, no such system was ever used for this purpose. We therefore deem energy based approaches for malware detection as a dead end—at least for modern smartphones without extended capabilities to account for used energy.

Acknowledgments This work has been supported by the German Federal Ministry of Education and Research (BMBF grant 01BY1020 – MobWorm).

References

1. Battery Performance Characteristics. http://www.mpoweruk.com/performance.htm.
2. N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *Internet Measurement Conference (IMC)*, 2009.
3. Christy Pettey and Rob van der Meulen. Gartner Says Worldwide Sales of Mobile Phones Declined 3 Percent in Third Quarter of 2012; Smartphone Sales Increased 47 Percent. http://www.gartner.com/newsroom/id/2237315.

4. D. Maslennikov and Y. Namestnikov. Kaspersky Security Bulletin. The overall statistics for 2012. www.securelist.com/en/analysis/204792255/Kaspersky_Security_Bulletin_2012_The_overall_statistics_for_2012.
5. B. Dixon, Y. Jiang, A. Jaiantilal, and S. Mishra. Location based power analysis to detect malicious code in smartphones. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM, 2011.
6. M. Dong and L. Zhong. Self-Constructive High-Rate System Energy Modeling for Battery-Powered Mobile Systems. In *International Conference on Mobile Systems, Applications, and Services*, MobiSys, 2011.
7. M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Network and Distributed System Security Symposium (NDSS)*, 2011.
8. 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 *USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2010.
9. H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *International Conference on Mobile Systems, Applications and Services*, MobiSys, 2010.
10. M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *International Conference on Mobile Systems, Applications, and Services*, MobiSys, 2012.
11. A. Jindal, A. Pathak, Y. C. Hu, and S. P. Midkiff. Hypnos: Understanding and Treating Sleep Conflicts in Smartphones. In *EuroSys*, pages 253–266, 2013.
12. H. Kim, J. Smith, and K. G. Shin. Detecting Energy-Greedy Anomalies and Mobile Malware Variants. In *International Conference on Mobile Systems, Applications and Services*, MobiSys, 2008.
13. L. Liu, G. Yan, X. Zhang, and S. Chen. VirusMeter: Preventing Your Cellphone from Spies. In *International Symposium on Recent Advances in Intrusion Detection*, RAID, 2009.
14. S. Park, A. Savvides, and M. Srivastava. Battery Capacity Measurement And Analysis Using Lithium Coin Cell Battery. In *International Symposium on Low Power Electronics and Design*, ISLPED, 2001.
15. A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? Fine Grained Energy Accounting on Smartphones with Eprof. In *ACM European Conference on Computer Systems*, EuroSys, 2012.
16. A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-Grained Power Modeling for Smartphones Using System Call Tracing. In *ACM European Conference on Computer Systems*, EuroSys, 2011.
17. R. Rao, S. Vrudhula, and D. Rakhmatov. Battery modeling for energy aware system design. *Computer*, 36(12):77–87, Dec. 2003.
18. Victor Chebyshev. Mobile attacks! http://www.securelist.com/en/blog/805/Mobile_attacks.
19. C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. AppScope: Application Energy Metering Framework for Android Smartphones Using Kernel Activity Monitoring. In *USENIX Annual Technical Conference*, ATC, 2012.
20. L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Conference on Hardware/Software Codesign and System Synthesis*, 2010.
21. Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy*, 2012.
22. Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Network and Distributed System Security Symposium (NDSS)*, 2012.