E-Android: A New Energy Profiling Tool for Smartphones

Xing Gao[1,2], Dachuan Liu[1,2], Daiping Liu[1], Haining Wang[1], Angelos Stavrou[3]
[1]University of Delaware, [2]College of William and Mary, [3]George Mason University
{xgao, dachuan, dpliu, hnw}@udel.edu, astavrou@gmu.edu

***Abstract*—As the limited battery lifetime remains a major factor restricting the applicability of a smartphone, significant research efforts have been devoted to understand the energy consumption in smartphones. Existing energy modeling methods can account energy drain in a fine-grained manner and provide well designed human-battery interfaces for users to characterize energy usage of every app in smartphones. However, in this paper, we demonstrate that there are still pitfalls in current Android energy modeling approaches, leaving collateral energy consumption unaccounted. The existence of collateral energy consumption becomes a serious energy bug. In particular, those energy bugs could be exploited to launch a new class of energy attacks, which deplete battery life and sidestep the supervision of current energy accounting. To unveil collateral energy bugs, we propose E-Android to accurately profile energy consumption of a smartphone in a comprehensive manner. E-Android monitors collateral energy related events and maintains energy consumption maps for relevant apps. We evaluate the effectiveness of E-Android under six different collateral energy attacks and two normal scenarios, and compare the results with those of Android. While Android fails to disclose collateral energy bugs, E-Android can accurately profile energy consumption and reveal the existence of energy bugs with minor overhead.**

I. Introduction

We have entered the era of smartphone. It is reported that, on average, there is one smartphone for every four people on earth [2]. A large number of apps developed for various purposes, including business and entertainment, emerge everyday. While smartphones have brought great convenience to our daily lives, the service availability of smartphones is still severely restricted by the limited battery lifetime. On one hand, the size of smartphone limits the battery capacity. On the other hand, various dazzling features make existing apps become energy hogs, forcing mobile users to charge their smartphones almost everyday. Moreover, one kind of malware targeting at depleting the battery of smartphones was reported before [18], [22]. Such energy malware exploits specific tricks like infinite loops and cache misses [22] or energy bugs [27] to deplete mobile devices' battery. Compared with other malware, energy malware usually exhibits normal activity features, and thus it cannot be captured by traditional dynamic or static malware analysis [18].

To provide a comprehensive view of the energy usage and detect abnormal energy consumption, previous works center on profiling energy consumption of each app. They track down the power consumption of each component and build energy models using either component utilization or different energy states. With the help of multiple sensors on the chips, current energy accounting could achieve high accuracy at high frequency. A well designed battery interface is further developed to visualize energy consumption to smartphone users. Through the battery interface, users can clearly understand where the energy is consumed, and take further actions such as terminating or even deleting those energy hog apps. As a result, it is not hard to detect existing energy malware under the combination of energy accounting and battery interfaces.

In this paper, we uncover the existence of collateral energy consumption and demonstrate that collateral energy consumption, which is overlooked and unaccounted by existing energy modeling, could cause biased energy profiling and user confusion. We define the existence of an unaccounted collateral energy consumption as a collateral energy bug. Even worse, by maliciously manipulating those mechanisms that trigger collateral energy bugs, adversaries are able to promote new energy attack techniques. These attacks can neatly sidestep current energy accounting and thus are undetectable to the battery interface, leading to stealthy energy drains in Android smartphones. The first attack vector is based on Inter-Process Communication (IPC). As each app runs in a separate sandbox, Android relies on IPC as the exclusive method for the communication between apps. However, we find that existing energy profiling approaches overlook IPC, and energy malware can trigger other innocent apps to drain battery through the IPC mechanism. The second attack vector is wakelock- and screen-based. Screen is a well-known energy intensive component in mobile devices, and wakelock is used to keep both screen and CPU awake. However, existing energy accounting does not track the screen energy consumption caused by background apps, and failing to release the wakelock can drain tremendous amount of energy. Thus, energy malware could simply modify screen configuration in the background or prevent other apps from releasing the wakelock, forcing the victim system running in a high power state.

To improve energy profiling accuracy and tackle collateral energy bugs, we design E-Android that takes the collateral energy consumption into account. E-Android consists of three major components: (1) an extension of Android framework to record all events that potentially invoke collateral energy bugs, (2) an enhanced energy accounting module to calculate energy consumption with consideration of collateral effects, and (3) a revised battery interface to inform users all information related to energy consumption. When an app attempts to interact with other apps, perform wakelock related operations, or change screen configuration, E-Android collects apps' user IDs and the type of operations. E-Android also carefully monitors the activities of task stacks, especially the state of the current foreground activity, to accurately identify a collateral energy bug. E-Android maintains a collateral energy map for fine

grained collateral energy accounting. The collateral energy consumption will be appropriately charged to the initiated source. E-Android is able to handle sophisticated situations such as collateral energy attack chains. We modify two battery interfaces to work jointly with E-Android. A mobile user can clearly understand how battery drains in a smartphone. E-Android can not only be applied for the defense of a collateral energy attack, but also produce more accurate energy accounting of benign apps.

We conduct a series of experiments to evaluate E-Android. We implement six types of energy malware running in a Nexus 4 mobile phone. We use both original versions and our modified versions of Android's official Batterystats application and PowerTutor [36] to measure the energy consumption. Our experiments indicate that all collateral energy attacks can successfully deplete battery life without being noticed by the original versions of battery interfaces. They also indicate that E-Android is able to detect and expose all energy malware. Under normal scenarios, we show that E-Android can illustrate all collateral energy consumptions. Moreover, we use both microbenchmark and marcobenchmark to quantify the overhead of E-Android, in terms of performance and energy consumption. We also use AnTuTu benchmark [1] to measure the memory overhead. Our results show that E-Android incurs negligible overhead.

The remainder of the paper is organized as follows. Section 2 briefly introduces the existing energy modeling methods. Section 3 describes several attack vectors that can sidestep current energy modeling and then presents collateral energy attacks based on these attack vectors. Section 4 details our defense mechanism. Section 5 presents our experiments and evaluation results. Section 6 surveys related work. Finally, Section 7 concludes.

II. Energy Modeling Overview

Fine-grained energy modeling of smartphones provides a straightforward way to understand the energy consumption of mobile apps, and has been investigated for years. These works utilize either extra power meters or sensors [6], [17] to break down energy consumption and build power estimation models for apps. Including Android per se, a battery interface is further utilized to provide users with necessary and detailed energy information.

A mobile phone is composed of a wide variety of hardware components, with examples including CPU, memory, WiFi, GPS, camera, bluetooth, and LED screen. Different apps could differ in functionality and trigger different usage on different hardware components, and hence turn components in different utilization or states. Energy modeling works [31], [36] measure the corresponding energy consumption of each component under different utilization, e.g., CPU utilization from 0-100 percent. A mathematic model using linear regression is developed to estimate the energy usage of distinct applications with the utilization information collected from mobile operating systems. However, those utilization based approaches could have an error rate as high as about 20%. As many components (e.g., GPS and camera) do not have quantitative utilization, several further works [29], [34] build a power state machine for components triggered by system calls on the kernel level. Those works achieve better accuracy than the utilization-based since they take tail power into consideration.

As one of the most energy draining components, screen receives extra attention [9], [13]. While high modeling accuracy has been achieved with several determining factors such as brightness and materials considered, the policy to distribute screen energy remains a controversial issue. To illustrate the amount of energy spent on screen, a battery interface is normally equipped with two mechanisms. The first is always to allocate the energy of screen to the foreground app [34], [36], which is the center of interacting with users. Another policy is to treat screen as an independent part, where the energy consumed by screen is always displayed in total. Such a method is used by the Android official battery interface.

Those works, focusing on energy modeling on an independent app, have achieved impressive high accuracy. For example, eprof [29] specifically decomposes the energy consumption into the subroutine or thread level, enabling fine grained energy accounting on a single app. However, all existing approaches aiming to accurately model energy consumption ignore the IPC mechanism in the highly dynamic Android ecosystem. As the fact that each app runs in a separate sandbox with unique user identification, Android apps largely rely on IPC to communicate with each other. The negligence, as well as the unsound screen energy accounting methods, could not only result in possibly partial energy accounting, but be exploited by specific energy malware to deplete device's battery. We will detail this problem in the next section.

III. Insufficiencies in Existing Energy Modeling

In this section, we first present several mechanisms that cause collateral energy bugs (i.e., the existence of collateral energy consumption). Current Android energy modeling fails to produce appropriate interpretations for those collateral energy consumptions on related apps. These mechanisms, which largely exist in normal apps, can be easily exploited as attack vectors to drain the battery stealthily. We further introduce a new concept, namely collateral energy attack, to demonstrate the security risk of the collateral energy bugs. Exploiting those attack vectors, an adversary is able to deplete a smartphone's battery without being detected by energy accounting.

A. Potential Attack Vectors

IPC-based Attack Vector. In Android, an activity occupies the screen and acts as the foreground user interface. Various activities collaborate together to form the functionalities of the app. Although activities are independent, they can switch and communicate to each other either inside the same app or between different apps. Here is a simple scenario. Bob receives a message from Alice while enjoying a music festival. To share the interesting moments, Bob uses the Message app to film a half minute video and send it to Alice, instead of quitting the Message and then starting the Camera app. A small camera window is embedded in the Message UI. From the perspective of Bob, all operations seem to occur in the Message app. Thus, it is reasonable to expect that the Message should be accounted for the corresponding energy consumption. To further understand the current energy profiling, we illustrate the energy consumption measured by Android official BatteryStats

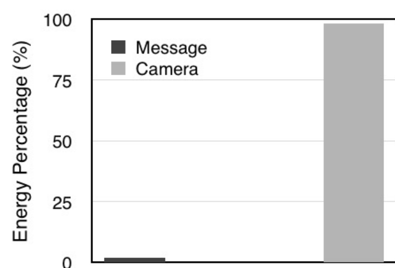


Fig. 1: Energy view when filming in the Message app.

in Figure 1. The figure shows the consumed energy percentages by the Message and the Camera. The result, however, indicates that the Message only consumes a quite small portion of energy. The fact is that the energy drained by video filming is assigned to the Camera, no matter what app opened the Camera or how it was opened.

The fact behind this observation is that, when Bob clicks "Record Video" in the Message, the Message sends an *Intent* to request the Camera app. It is the Camera app that actually records the video. After the recording, the video is returned to the Message app, and is available to be delivered to Alice. Interestingly, the camera is reported as the most energy draining app [4]. However, we observe that the energy consumption sometimes involves the communications between apps, and hence other apps should also be responsible for those indirect energy consumptions.

Such a scenario frequently and legitimately occurs in Android since IPC serves as the sole approach for enabling apps to communicate with one another. IPC, built upon the Binder in the Linux kernel, mainly relies on the *Intent* in the Android framework, by which an action from another component can be requested. The recipient of intent can be invoked by components from an external app once invoked. With the use of intent, apps can reuse existing components simply by starting an activity or a service from other apps. While the intent mechanism, along with other IPC mechanisms, plays a critical role in Android, existing energy accounting modules overlook such factors. The omission could beget unexpected consequences, including biased energy accounting on normal apps. Even worse, current energy modeling could be exploited by malicious apps for mounting a collateral energy attack, leading to even more serious damage.

Wakelock- & Screen-based Attack Vector. Android contains a sophisticated power management inherited from the Linux power system [19]. While Linux supports three global system power states: *on*, *off*, and *suspend*, Android suspends all peripheral devices by default and turns a device into deep sleep after some idle time to save energy. In the *suspend* state, devices are in a low power state with CPU disabled and processes halted. To override the aggressive power saving policy, Android introduces the wakelock, which is a special power management module to keep devices awake. Android developers are able to access to four types of wakelocks, including power consuming components like CPU and screen.

A wakelock must be released once acquired to avoid keeping device alive. Otherwise, battery will be drained up to 25% per hour [30]. In case apps fail to release the wakelock properly, Android does not release the wakelock until the process has been killed, assisted by the link-to-death mechanism of Android Binder. When "PowerManagerService" receives a request from an app to acquire the wakelock, it registers the wakelock and links a token to the death of the app's process. Only the death notification, which is dispatched by the kernel Binder driver, of the app's process will inform the device to release the wakelock.

Android attempts to inform developers about the usage of a wakelock. Unfortunately, Pathak et al. [30] observed that a large number of developers fail to understand how to properly use a wakelock. One improper usage is that, an app only releases the wakelock in the `onDestroy()` function, without releasing it in `onPause()` or `onStop()`. An activity will step into the `pause` state by invoking the `onPause()` function when it is covered by a transparent activity. The `onStop()` function is triggered when an activity enters background. Only when the process is destroyed, will `onDestroy()` be called. Such misinterpretation causes significant hazards to battery life. Normally, the app would be destroyed when the user quits the app, without causing any problem. However, in Android, a foreground activity could be easily interrupted by popup activities, e.g., the activity invoked by a notification, an incoming call or an alarm. The popup, either intentionally or unintentionally, makes the foreground app that fails to properly release the wakelock keep draining device energy.

Mis-releasing a wakelock also challenges the screen energy modeling policy. Existing energy profiling allocates energy to either the screen or the foreground app, leaving the bug app unnoticeable. Moreover, a wakelock could be acquired in background. If the consumed energy is only allocated to the foreground app rather than the initiator, the energy modeling would confuse and mislead users on the internal energy consumption.

Among four types of wakelocks, three of them can keep the screen on. A wakelock continues playing an irreplaceable role in the Android power management, although Android has deprecated parts of a wakelock since Android API 17. App developers still largely use a wakelock. Thus, existing energy profiling is insufficient for common real world cases. More seriously, the inadequate consideration might be abused by energy malware to drain energy without being disclosed by the battery interface.

B. *Collateral Energy Attack*

Threat Model. The collateral energy attacks are launched on normal unrooted devices to drain battery. An attacker has no need to exploit any vulnerabilities in operating systems or the Android framework. To avoid being exposed by the battery interface, the attacker exploits specific attack vectors, instead of using hands-on approach. For the IPC-based attack vector, usually the attacker does not need any permission to use an exported component of the attacked apps. Also, since the UI states of an app are standardized, the attacker can easily understand them by either installing the app or reverse engineering the app. For the screen- and wakelock-based attack vector, the attacker does need some permissions. Specifically, the attacker needs the WRITE_SETTINGS permission to modify the screen configuration and the WAKE_LOCK permission to acquire a wakelock.

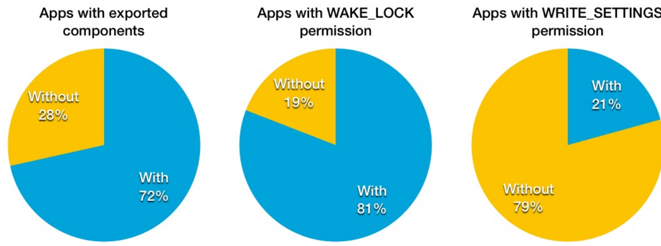


Fig. 2: Collected apps from Google Play.

To demonstrate those assumptions are common in the wild, we collect 1,124 popular apps from Google Play. The collected apps fall into 28 catagories, including game, business, and finance. We use APKTool [3] to extract the AndroidManifest.xml file of each app by reverse-engineering the app. We inspect those apps from three aspects: (1) does the app contain an exported component? (2) does the app require the WAKE_LOCK permission? and (3) does the app require WRITE_SETTINGS permission?. Figure 2 shows the result: 72% of apps contain exported components; 81% and 21% of apps require the WAKE_LOCK and WRITE_SETTINGS permissions, respectively. As a collateral energy attack can be launched by any apps, malware can camouflage as a game or useful tool to acquire those permissions above.

Attack #1. *Similar to the camera case, malware hijacks components belonging to other apps.* Existing energy modeling does not consider IPC communications. This mechanism enables malware to drain the target device's energy through a combination of legal operations, and hence bypass the energy monitoring. Compared with a traditional component hijacking problem, it is more difficult to prevent energy based hijacking, because there is no need for a data flow between components to transmit information. Thus, energy malware could choose the energy hog component to launch an attack.

Attack #2. *When malware is launched, malware can open other apps concurrently and make them run in background.* It has been long reported that a background app definitely drains battery. Android does not kill background apps immediately. Background activities enter the "pause" or "stop" state. The app will release all resources only when it enters the "destroy" state. Also, services run in the background handling extensive workload and drain battery. Even in the idle state, the battery life could be decreased by up to 77.5% by simply installing Google services comparing to pure AOSP Android [23]. Thus, triggering background apps is a very effective way to drain battery.

Attack #3. *Bind to services without unbinding.* Malware could further launch attacks on background services, where heavy computational workload normally runs. A service could be started by calling `startService()` function or bound by `bindService()`, allowing IPC communications across processes. A started service will not be terminated even the started component is destroyed and must be stopped by `stopService()` or `stopSelf()` to avoid running indefinitely. Similarly, a bound service must be unbound. However, for services always running in the background, the life cycle differs from activities. Multiple components can bind to a single service simultaneously, making the service alive until all connections are unbound, even under the condition

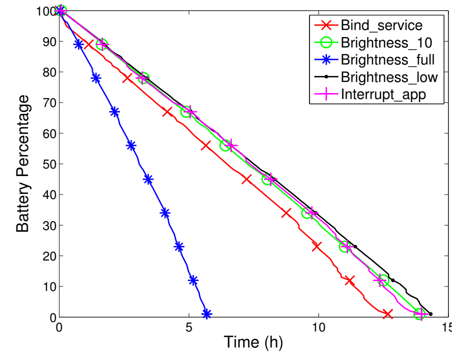


Fig. 3: Difference of time lapsed to drain the battery.

that `stopService()` has been triggered. Thus, an exported service bound by malware will keep alive infinitely and drain battery even after the victim attempts to stop the service.

Attack #4. *Interrupt attacked apps to background.* Malware could also forcibly switch the victim to background through normal operations, such as opening the launcher. Furthermore, sophisticated malware could utilize other complicated techniques to interrupt the foreground app without being noticed by users. For example, attackers can interrupt and switch the app to background when the user attempts to quit the app. The misinterpretation of wakelock could make the energy attack even more serious. Since the app enters the "stop" state, instead of being killed, it might fail to release the wakelock. Since the wakelock is un-released by the victim, energy accounting will tax the energy into the victim, without disclosing malware behind curtain.

Attack #5. *Drain energy through changing screen configuration.* The brightness acts as the determining factor for the screen energy consumption. Malware could change the screen setting in background. A large number of apps enhance the brightness when they are running in foreground. Therefore, users probably would not perceive the malicious adjustment of brightness by malware. In particular, to avoid being noticed, malware could secretly escalate the brightness with a few levels. Since Android provides 256 levels to adjust the brightness of screen, such a slight enhancement might not affect users, but cut the battery lifetime. The brightness could also be adjusted automatically. In the auto mode, Android chooses the brightness level based on surrounding environments and disables manual change on lighting. Complicated and advanced energy malware could camouflage as Android auto screen settings, by setting a higher value after obtaining current auto set brightness.

Attack #6. *Acquire a screen wakelock without releasing.* The wakelock could also be utilized to conduct a screen energy attack. Malware could easily keep screen on by intentionally acquiring but not releasing the wakelock. The wakelock could even be acquired by services. The consumed screen energy will be wrongly attributed to the foreground app or Android launcher, rather than malware.

Multi- & Hybrid Attack. Sophisticated malware could combine the above attacks together for more effective attacks. Attackers could also conduct multiple attacks on the same attacked app. For instance, malware could bind a victim's service and increase the brightness when the victim is running in foreground. Also, malware could spread the attack to a series

of victims. Malware could conduct an attack on one victim, which unintentionally involves another, leading energy attack chains.

Attack Analysis. Energy malware stealthily depletes battery by abusing attacked apps. Hence, the effectiveness of such an attack depends on the function of victims rather than malware per se. Different apps lead to different energy consumption. We utilize several simple cases to study the attack effectiveness. We measure the time duration of the above attacks for consuming the total battery. For each percentage of battery, we record the time until the battery is dead. Keeping screen on via the screen wakelock will significantly accelerate the battery consumption compared with using the auto lock that turns the screen off. For all experiments, we set the wakelock so that the screen will be forced on. We treated the lowest brightness case as the baseline case. Our simple cases include setting brightness with the maximum value, setting brightness with 10, binding a service of other app, and interrupting an app in the background. Figure 3 illustrates the results of our simple attack cases.

As shown in the figure, screen strongly affects battery drain. A small increase of brightness, which brings little visual effect, can increase battery drain. Also, running an app in background also divulges battery. Note that we simply use our demon apps that almost have no functionality as attacked apps in this experiment. Modern apps containing more complicated functions would consume more collateral energy. Also, the drain of battery will be even faster if energy malware employs multiple attack vectors simultaneously. Thus, a collateral energy attack is able to effectively drain a smartphones battery in a stealthily manner.

Attack Scenarios. Collateral energy attack could be viewed as a variation of denial of service attack. Malware can reduce the battery's lifetime and degrade a user's experience by indirectly consuming energy. Furthermore, collateral energy attacks could mislead a user's attention to an innocent app and cause unfair competition. For instance, an app that is competing with another app could intentionally mount collateral energy attacks on the rival so that the rival consumes much more energy unconsciously, resulting in energy disadvantage.

Unlike traditional attacks targeting at leaking personal private information or controlling system resources, collateral energy attacks aim to significantly reduce the battery's lifetime, as the battery is the most scarce resource of a mobile device. Also, collateral energy attacks could be conducted accompanying with traditional attacks. For example, the privilege escalation attacks utilize IPC calls to escalate privileges inside the sandbox. At the same time, the IPC calls could be abused by malware to launch collateral energy attacks.

IV. E-ANDROID: A NEW ENERGY PROFILING TOOL

Traditional malware analysis including both static and dynamic techniques could be applied to detect collateral energy bugs. However, as we mentioned above, normal apps could also induce a large amount of collateral energy consumption. As a result, energy malware might exhibit normal activity features and can evade malware analysis. Furthermore, compared with other attacks utilizing IPC channels, explicit data flows between different components may not exist in collateral energy attacks. Thus, it might be difficult to discern collateral energy attacks

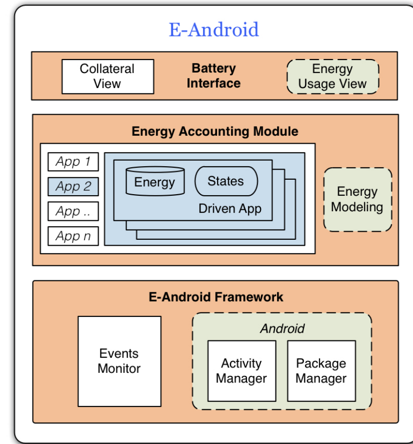


Fig. 4: E-Android Architecture

from normal operations. Also, it is entirely possible that an app consuming much collateral energy is still welcomed by mobile users. From the perspective of energy profiling, the key is to accurately and comprehensively profile the energy consumption so that users can understand where energy goes and make their own decisions on apps management. Therefore, we introduce E-Android to assist the battery interface to reveal collateral energy consumption and defend against collateral energy attacks.

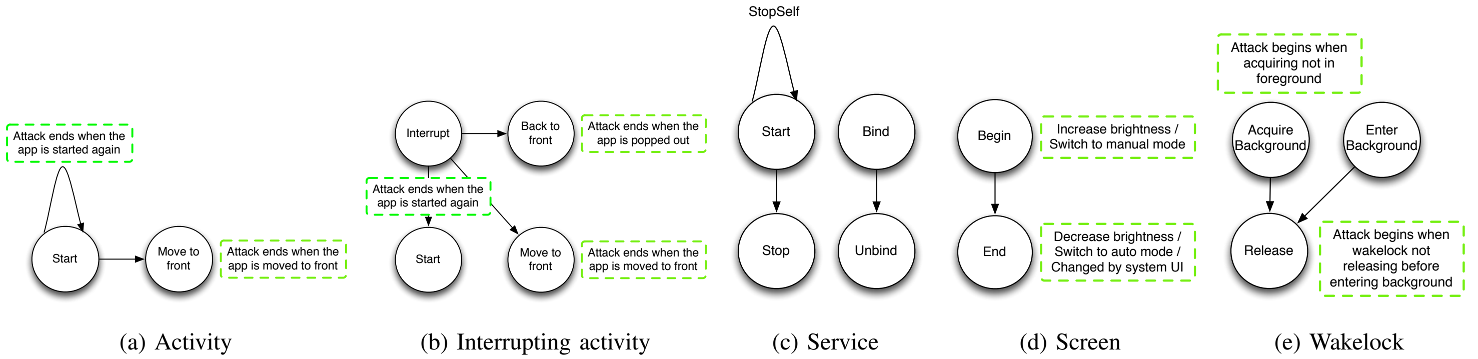
E-Android is composed of three major components: an extension of Android to log all potential energy operations, an enhanced energy accounting module to calculate energy consumption with consideration of collateral effects, and a revised battery interface to inform users of all the energy consumption related information. E-Android can not only be applied to defend against energy malware, but also provide users with a more accurate profile of the internal energy consumption in mobile devices. E-Android leverages dynamic methods, instead of static methods, to measure collateral energy consumption, because dynamic methods provide a better understanding of the runtime context, such as the timing of triggering collateral energy bugs, which are crucial for measuring collateral energy consumption.

We illustrate the architecture of E-Android in Figure 4. Basically, E-Android monitors a series of events that potentially lead to a collateral energy attack, e.g., starting an activity, binding a service, changing screen settings. Each time an event is triggered, E-Android checks the user ID of both the driving app and the driven app[1]. If different, E-Android records the user ID of both apps, as well as the type of operations, and notifies the energy accounting module. The module then updates the relevant energy data. The battery interface displays a collaborative energy view, with all related apps specified.

A. E-Android Framework

System Apps. Android contains built-in apps and internal apps. In Android, the home UI is essentially the launcher app, which is used to interact with all apps for users. Users could start an app by touching an icon in launcher or return

[1]The driving app is the app operating others, e.g., the app starting others, and the driven app is the app being operated, e.g., the app being started.



(a) Activity (b) Interrupting activity (c) Service (d) Screen (e) Wakelock

Fig. 5: Attack lifecycle

to the home UI via the home button. Another key app is the system UI. The system UI allows users to customize a device's characteristics, such as screen brightness. The "resolverActivity" is used for users to select an app responding to an implicit intent. E-Android treats these built-in apps and internal apps as system apps and excludes them from the collateral energy attack list. However, E-Android still logs events of those special apps as a vital factor to correctly calculate collateral energy consumption.

Activity. An activity possesses the screen and stays active running in foreground. An activity could be started by explicit intents or implicit intents. The component name is specified in an explicit intent. The activity will be started directly as a result. For the explicit intent case, E-Android directly keeps tracking of the driving app and the driven app. Contrarily, an implicit intent only declares a general action, instead of specifying the component name. When an implicit intent is launched, Android starts "resolverActivity", where a user could designate the app to start. Once the user makes a decision, Android starts the selected app by dispatching a new explicit intent. For the implicit intent case, E-Android tracks both intents and ignores the Android system's UI, and records both apps' user IDs after the choice is made.

E-Android views that a collateral energy attack starts when an activity is launched by another app. The attack period lasts till the next time the driven app is started, as illustrated in Figure 5a. Such a policy is reasonable since the driven activity could continuously drain the battery even in the background. On the other hand, if the driven activity incurs no extra energy consumption in the background, the addition of such a period has little impact on the driving app.

Android maintains certain task stacks to manage activities. When an activity is sent back to background, it remains in the stacks keeping all statuses at that time. Android pops out the most recent activity when one leaves foreground. Moreover, users or apps equipped with proper permissions could reorder the stack. A background activity could be simply moved to foreground without necessity to start. E-Android also accurately observes the period of a collateral energy attack by monitoring the statuses of task stacks.

Interrupting Activity. E-Android regards a foreground activity that forcibly moves the prior into background as an interrupting activity. An interrupting activity could possibly lead continuous battery draining, as the driven activity is forced in the stop state. E-Android records the user IDs of the previous foreground app and the interrupting activity if they are different. Several ways could resume the front app, including the operations from the user, IPC operations, and the operations on the activity stack. As shown in Figure 5b, the attack period ends when the previous front app resumes and the next potential attack period starts. By this way, E-Android can clearly record and allocate the energy consumption of each period without adding undeserved energy consumption to any apps.

Service. E-Android records several events related to a service, including start/stop/stopSelf and bind/unbind. Similar to activity, E-Android notifies the battery interface only when IPC happens between two apps. The period accounting for a collateral energy attack begins from start/bind and ends at stop or stopself/unbind.

Screen & Wakelock. Screen brightness could be changed either manually or automatically. If apps set a value in the auto mode, the value is saved into the settings provider but not valid until the mode is switched to manual. E-Android records two events as the start of a screen collateral attack. The first event is to enhance screen brightness under the manual mode. The second is that apps attempt to switch the auto mode to the manual mode. Once the screen collateral attack starts, E-Android also scrupulously monitors the end of period. Those events include switching into the auto mode, brightness changed by system UI (i.e., operated by users), and brightness decreasing by the attacking app. Figure 5d illustrates the details.

E-Android also records the usage of the wakelock related to screen. E-Android starts the wakelock collateral attack when the foreground app is not the app acquiring the wakelock, to defend against malware acquiring the wakelock in the service. Also, since apps use the wakelock to prevent screen off during its lifetime, a wakelock collateral energy attack begins under the condition that the wakelock is not released while the foreground app is altered. E-Android marks the end of the attack when the wakelock is released.

B. Energy Accounting Module

The energy accounting module maintains specific energy modeling for collateral energy consumption. A detailed energy map between an app and its collateral apps is maintained for each app. Each time the battery interface captures a notification, it updates the collateral energy map. While a sophisticated policy could be easily applied, currently the strategy handling basic collateral attacks is straightforward. E-Android counts

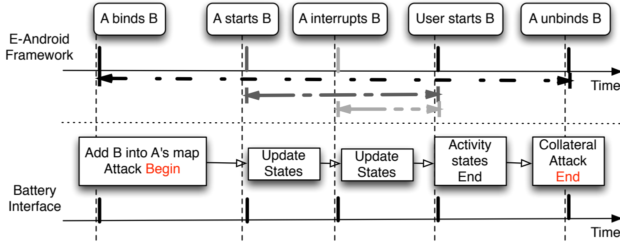


Fig. 6: Multi-collateral Attack

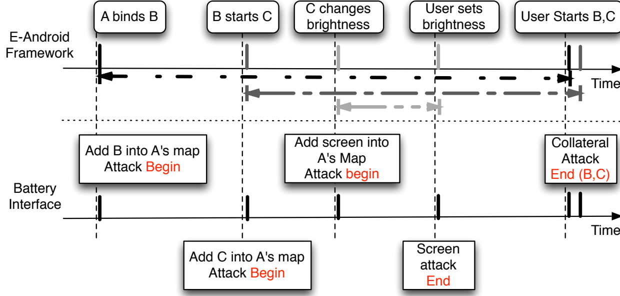


Fig. 7: Hybrid Attack

the driven app's energy consumption in the attack period to the driving app. However, one issue is multi-collateral attack (Figure 6) on the same app. An app could start activities, bind services, and interrupt activities on the same app. To avoid repeated accounting energy consumption for the same driving app, E-Android carefully maintains states for each attack to connect driving apps and driven apps. The connection will be revoked only after all collateral attacks end.

Moreover, the hybrid collateral attack complicates the accounting. The intricate IPC communications in Android easily lead to collateral attack chains. An activity started by another app could act as the man in the middle and interact with the third app, which could conduct more complex attacks. Take Figure 7 as an example. App A binds a service of app B, which starts one activity belonging to app C. App C then stealthily changes screen. Since it is app A that starts the whole collateral energy attack, it is reasonable to charge the energy drained by C and screen to A. At the same time, E-Android still needs to monitor the attack period of each attack on the chain. If one attack is over, the following attack should be updated. It is important to consider such a chain scenario, due to its frequent appearance in both malware and legitimate apps, e.g., Bob opens the Message started by the Contacts and sends a video taken by the Camera to Alice.

E-Android handles this situation by traversing each energy map and updating energy consumption, as formulated in Algorithm 1. Each time a beginning event occurs, the driving app is checked in the energy map for each host app. If it exists and the connection is alive, E-Android adds current driven app into the host's energy map. Conversely, when it comes to an end event, E-Android also removes the link brought by the driving app by updating attack states of the driven app element in the energy map.

The situation is more complicated for service events. The driven app could have already bound several energy intensive services before the triggered event. Those extra energy consumptions should also be charged for the driving app and its parents. E-Android updates the energy maps of the driving app and its parents by adding the elements in the driven app's energy map.

Algorithm 1 Energy update algorithm

```
 1: app_n ← {Driven app or Screen}
 2: m_g ← {Energy map of driving app}
 3: M_p ← {Energy maps contain driving app}
 4: M_c ← {Energy maps of apps in m_g}
 5: procedure EVENT_TRIGGERED
 6:     EndLastAttack(app_n)
 7:     m_g ← AddElement(app_n)
 8:     for each app app_i ∈ M_p do
 9:         m_i ← AddElement(app_n)
10:     end for
11:     if Event is service related then
12:         for each app app_j ∈ M_c do
13:             m_g && M_p ← AddElement(app_j)
14:         end for
15:     end if
16: end procedure
```

Note that only the part of energy consumption during the attack lifecycle would be superimposed to the collateral energy of the driving app. Once all attack lifecycles end, the relation between the driving and driven apps is broken and no extra energy would be charged.

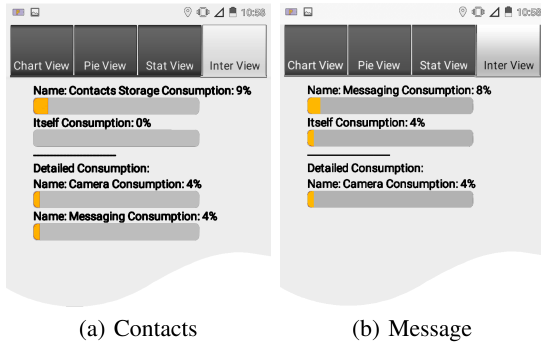
C. Battery Interface

The battery interface lists apps that consume a great deal of energy. E-Android ranks apps by total energy consumptions including collateral energy consumption. Moreover, for each app, E-Android provides a detailed inventory specifying contributions of all attack related apps. To better demonstrate the energy consumption, the apps' original energy is also listed. One sample view is provided in Figure 8, which exhibits the energy breakdown of the legitimate hybrid attack above. The breakdown assists users in better understanding the energy consumption.

V. Implementation

We modified the framework of Android 5.0.1 to implement E-Android. E-Android mainly relies on "am", which manages all components in Android, to record collateral energy events. We also utilized other classes such as "powerManagerService" to supervise the usage of wakelocks. All collateral energy events are directly input to the energy accounting module. We included the collateral attack modeling features to both Android official battery interface and powerTutor [36].

We also implemented six types of energy malware as we presented before. The malware implementation is quite straightforward, except for malware types 4 and 5. Malware #4 interrupts and switches a victim to background when a user attempts to quit the app. Most apps transfer to root activity first before opting out. Android will pop out an exit dialog to inquire users. If "OK" is clicked, the app is destroyed. Therefore, once malware successfully eavesdrops the appearance of such a



(a) Contacts (b) Message

Fig. 8: Sample view of energy breakdown by E-Android with revised PowerTutor

dialog, it can send a transparent activity to cover the victim activity. When the user clicks the position where "OK" locates in the transparent activity, malware sends an intent to start home UI. While the user feels no difference, the home UI actually induces the victim to invoke onStop(), instead of being destroyed, and thus continue draining the battery in the background.

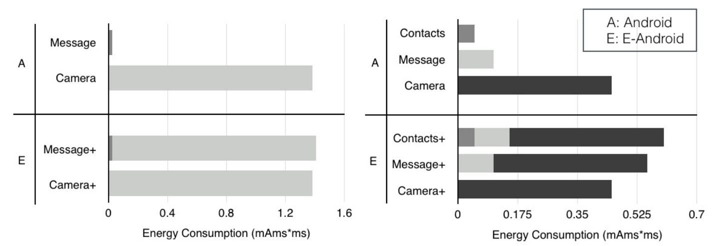
There are different ways to monitor the occurrence of the dialog. Like the technique used in the UI inference attack [8], we can utilize the shared virtual memory size of SurfaceLinger to infer activities. SurfaceLinger is the process to render UI in Android. The shared virtual memory size will change when the UI state is changed. Although a dialog is not an activity, both the root activity and the style of a dialog usually remain unchanged for most apps. Thus, the offset of the shared virtual memory is still applicable to infer the occurrence of the exit dialog. Using this technique, malware #4 is able to detect the exit dialog of the victim and sends a transparent page to cover the dialog and start the home UI once the position of "OK" is clicked.

Malware #5 secretly enhances the brightness in the background by modifying the configuration in Settings. However, a service might not be able to set window attributes and the change may not be in effect immediately. To achieve the attack goal, malware #5 sends a transparent self-close activity to set window attributes.

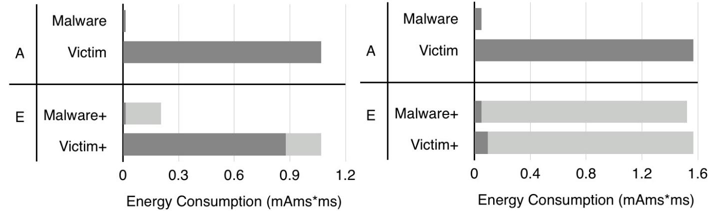
To make the attacks stealthier, our malware sets a particular flag to hide itself from recent apps so that normal users would be unaware of the existence of malware. The attacks mounted by malware appear normal, due to the fact that some apps in Android would listen for specific intents to automatically launch. For example, some apps would be opened when a user unlocks the screen by monitoring the ACTION_USER_PRESENT intent.

VI. Experiments

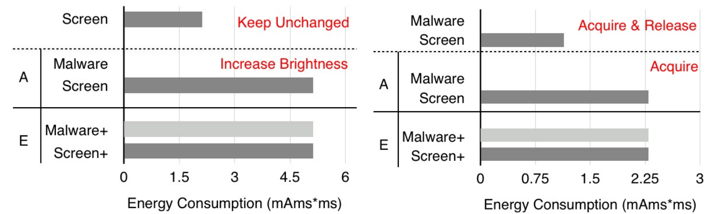
E-Android is designed to assist users to detect collateral energy consumption. To validate the effectiveness of E-Android, we ran experiments on both Android and E-Android, and compared the results. We first showed that E-Android can accurately profile collateral energy consumption in normal cases. Then we demonstrated that collateral energy attacks can sidestep Android and deplete battery, but E-Android can easily detect those attacks. We further presented that E-Android incur



(a) Scene #1 (Similar to attack #1 and attack #2) (b) Scene #2 (Similar to hybrid attack)



(c) Attack #3 (d) Attack #4



(e) Attack #5 (f) Attack #6

Fig. 9: Experimental results

little overhead on both performance and energy consumption in comparison with Android.

We utilized both Android's official interface and PowerTutor to measure the battery consumption. The results of PowerTutor are similar to those of Android's interface, thus we omitted them due to space limit.

A. *Effectiveness*

We first conducted experiments simulating real scenarios. We opened the Message app and waited 30 seconds, and then used it to take a 30 seconds short video. A more complicated scenario is that we used the Contacts to open the Message, then film a 30 seconds video exactly like the hybrid attack example. Those two cases are common and can represent normal smartphone usages.

Figures 9a and 9b illustrate the results for both normal cases. For better presentation, we use "A" to represent the results of Android, and "E" to represent the results of E-Android. Also, we use '+' to indicate the components in E-Android. As we can see, in the Android energy modeling, the Camera app expends much more energy than the Message app, regardless of the fact that the Camera is opened by the Message. However, in E-Android, the Message is also charged for the portion of energy consumed by the Camera. Thus, while the driving app would not be charged for anything in the original energy modeling, E-Android clearly reveals the collateral energy consumption and profile the battery usage by apps properly.

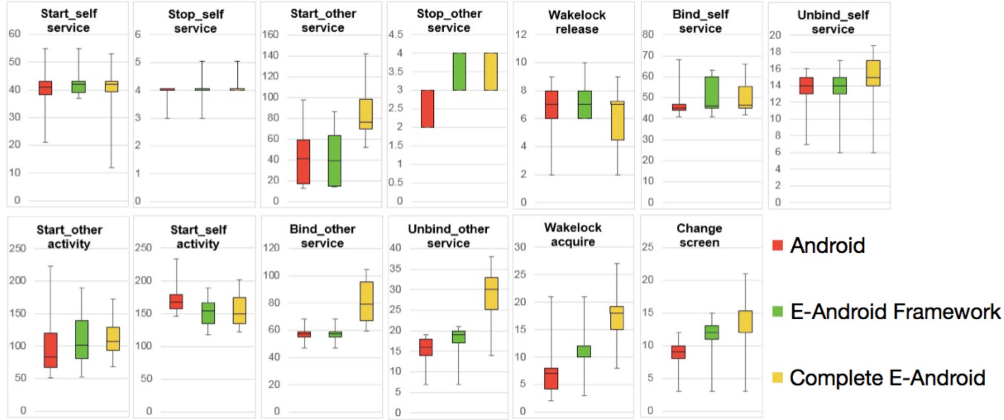


Fig. 10: Boxplot of time cost (ms)

We then launched six types of collateral energy attacks using our implemented malware. Each attack lasts 60 seconds. In the first two attacks, malware either directly or simultaneously initiates activities of other apps. For malware #3, it binds the victim's service once it detects the service is started. The attacked app starts its service and stops it immediately. However, the connection bound by malware forces the service to run continuously. We measured the energy consumption after malware #4 interrupts the victim to background. For attack #5, we first measured the regular screen energy consumption. Then, we measured the energy consumption after malware enhances brightness. While Android turns screen off after 30 seconds, we measured the energy consumed by screen for 60 seconds under the conditions that malware #6 releases/does not release the wakelock.

For attacks #1 and #2, their results are similar to that of the normal case #1. We displayed the results of attacks #3 and #4 in Figures 9c and 9d, as well as the results of attacks #5 and #6 in Figures 9e and 9f, respectively. For Figures 9e and 9f, the upper part indicates the energy consumption under normal circumstances. The lower part presents the results of both Android and E-Android under attacks. As the figures show, energy attacks consume much more energy than normal usage.

Notation	Definition
Start_self_service	Start a service belongs to same app by `startService()`.
Stop_self_service	Stop a service belongs to same app by `stopService()`.
Start_other_service	Start a service belongs to different app by `startService()`.
Stop_other_service	Stop a service belongs to different app by `stopService()`.
Bind_self_service	Bind a service belongs to same app by `bindService()`.
Unbind_self_service	Unbind a service belongs to same app by `unbindService()`.
Bind_other_service	Bind a service belongs to different app by `bindService()`.
Unbind_other_service	Unbind a service belongs to different app by `unbindService()`.
Start_self_activity	Start an activity belongs to same app by `startActivity()`.
Start_other_activity	Start an activity belongs to different app by `startactivity()`.
Wakelock_acquire	Acquire a wakelock by `acquire()`.
Wakelock_release	Release a wakelock by `release()`.
Change_screen	Change screen brightness.

TABLE I: Notations of micro operations.

Also, all those attacks can successfully bypass the supervision of the Android battery interface. However, E-Android can detect all these collateral energy attacks. Moreover, from the result of attack #3, we can also clearly see that only the energy consumed during the period of a collateral attack is attributed to malware. E-Android does not charge the energy consumption beyond that attack to malware.

B. *Overhead*

Micro benchmark. To measure the overhead of E-Android, we first recorded the time cost of several critical events that E-Android monitors. Table I lists all micro operations we measured. We run each operation 50 times on both Android and E-Android.

We excluded the two biggest and smallest values as outliers in our dataset. Figure 10 illustrates the box plots of execution time of all operations. The unit of vertical coordinate is millisecond. We first disabled the energy accounting module of *E-Android*. The difference between Android and E-Android is actually the overhead of the E-Android framework, which monitors all events related to collateral energy consumption. The results show that the E-Androids framework has almost the same performance overhead as Android. *Complete E-Android* indicates the time cost when E-Android enables the energy accounting module. We can see that E-Android only induces trivial overhead when events occur within a same app, e.g., starting another activity of a same app. This is because the event within a same app is not a collateral energy event, and thus it does not bring any extra workload to the energy accounting module and the battery interface. For a case with multiple apps, E-Android does cost more time on those operations. However, the overhead still remains the same order of magnitude with less than few milliseconds. Moreover, the delay only occurs when collateral energy related events are triggered. Otherwise, E-Android works exactly like Android, without inducing any extra overhead. Overall, the little overhead induced by E-Android will not degrade users' experience.

AnTuTu Benchmark. We also used AnTuTu benchmark to measure the CPU and memory overhead. AnTuTu evaluates performance in several aspects, including memory, CPU performance for both float and integer, and I/O. The bigger score means better performance. We listed the total scores as well as

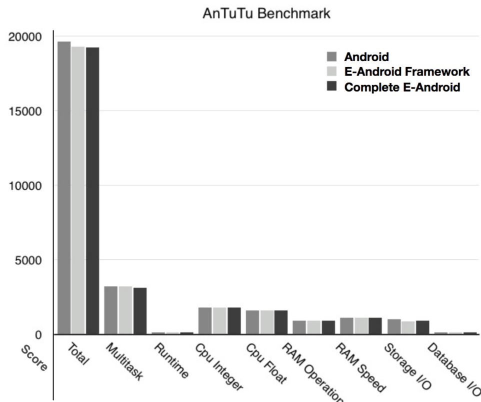


Fig. 11: AnTuTu benchmark

several key factors in Figure 11. The results demonstrate that E-Android has a similar overhead as Android.

Energy Efficiency. We also recorded the battery level to validate the energy efficiency of E-Android. In all above experiments, the decreased energy level is the same between Android and E-Android. Since E-Android only takes additional actions when collateral energy events are triggered, it will not drain extra energy at other times.

C. Discussion

In comparison with traditional energy profiling, E-Android profiles the energy consumption of apps from a different perspective, with an emphasis on the interactions among apps. Since almost no extra overhead is introduced if the enhanced energy accounting module in E-Android is disabled, E-Android would not affect users who prefer to use the original energy modeling. On the other hand, as our experimental results show, with energy modeling enabled, E-Android will provide a more accurate and complete energy profiling and detect energy malware with minor overheads.

VII. Related Work

Energy Modeling. Many prior studies [6], [17], [29], [31], [34], [36] focus on energy modeling. Dong et al. [12] escalated the rate of energy modeling from 1-10hz to 100hz while keeping high accuracy. Mittal et al. [25] presented an energy emulation tool that estimates the energy usage of an emulated workstation. Other works [9], [13] focus on energy modeling on screen. While these works have achieved high accuracy and high frequency, none of them consider IPC. Different from these prior works focusing on a single app, we take the interaction between apps into account and propose new policies to model energy. Our work enhances existing approaches with a comprehensive and impartial energy modeling.

Energy Bug. Several studies investigate abnormal battery drain issues. Pathak et al. [28] used Eprof [29] to profile power consumption. They found most energy spent on a small part of routines. Pathak et al. collected posts from four mobile user forums where various energy bugs were reported, including bugs in hardware, OS, framework, and apps [27]. Thiagarajan et al. [32] studied how mobile browsers consume energy. Pathak et al. [30] conducted a comprehensive analysis on no-sleep bugs using static methods and reported a variety of wakelock bugs, such as releasing a wakelock improperly. More works [21], [35] develop tools to automatically diagnose these energy bugs. While those works focus on special energy bugs of independent apps, they do not consider the interaction between apps.

Energy Malware. Several works study energy malware on mobile device. Martin et al. [22] first implemented three types of attacks aiming to drain energy of a mobile phone: (1) sending repeated network requests to a victim, (2) replacing a still image with an animated GIF, and (3) making cache miss by repeatedly writing and reading an array with different length. Chandra et al. [7] proposed a covert channel attack based on smartphone battery. Although these attacks are proved to be effective, they are detectable by battery interface. Users could easily detect them by simply checking the energy consumption. Kim et al. [18] proposed power signatures to detect energy malware. While they achieved promising results of detecting four types of bluetooth worms and a DoS-attack-like bomber, power signature cannot tackle collateral energy malware that drains energy via an indirect approach.

Human-Battery Interface. The human battery interface that allows users to monitor the energy consumption is also a well-studied topic [14], [16], [24], [33]. Previous research attempts to develop a more accurate interface with various forms and different contents to illustrate the power consumption. While our work could easily incorporate with these advanced battery interfaces, we just choose the basic interface as the tool to display the energy consumption of a smartphone.

Security and Privacy. Numerous researchers have studied security and privacy issues on mobile phones [5], [15], [38] The privilege escalation [11], component hijacking [20], and content provider pollution attacks [39] leverage IPC in Android for malicious purposes. Several works [26], [37] have been presented to defend against those IPC-based malware. However, energy malware depleting energy behaves normally, and thus it can avoid being detected by traditional malware defense mechanisms. Curti et al. [10] detected malware based on energy consumption, but they used existing energy modeling tools to measure an app's energy consumption. Therefore, such a work still falls into the weakness of current energy modeling.

VIII. Conclusion

In this paper, we are the first to introduce the concept of collateral energy consumption and related bugs. We have revealed several mechanisms in Android such as IPC that could confuse existing energy modeling approaches to trigger collateral energy bugs. We have further presented a set of new attacks based on collateral energy bugs and demonstrated that Android is vulnerable to these attacks. Specifically, exploiting the collateral energy bugs, energy malware can shorten the battery's lifetime and evade the supervision of current energy accounting. We have proposed E-Android to assist mobile users to tackle collateral energy bugs. E-Android can not only detect energy malware, but also provide a more accurate energy accounting under normal conditions. We have evaluated both Android and E-Android in terms of accounting energy consumption under six implemented malware and two normal scenarios. While all the proposed attacks can stealthily drain battery without being caught in Android, E-Android can clearly reveal collateral energy bugs and detect the attacks. Finally, we have shown that the overhead of E-Android is minor.

IX. ACKNOWLEDGMENTS

We would like to thank our shepherd Feng Qian and the anonymous reviewers for their insightful and detailed comments. This work was partially supported by NSF grant CNS-1618117.

REFERENCES

[1] Antutu benchmark. http://www.antutu.com/en/index.shtml.

[2] Smartphone user penetration as percentage of total global population from 2011 to 2018. http://www.statista.com/statistics/203734/global-smartphone-penetration-per-capita-since-2005/.

[3] A tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/.

[4] Top 10 android battery-sucking vampire apps. http://betanews.com/2014/02/27/top-10-android-battery-sucking-vampire-apps/.

[5] M. Becher, F. C. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf. Mobile Security Catching Up? Revealing the Nuts and Bolts of the Security of Mobile Devices. In *IEEE S&P*, 2011.

[6] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *USENIX ATC*, 2010.

[7] S. Chandra, Z. Lin, A. Kundu, and L. Khan. Towards A Systematic Study of the Covert Channel Attacks in Smartphones. In *SECURECOMM*, 2014.

[8] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security*, 2014.

[9] X. Chen, Y. Chen, Z. Ma, and F. C. Fernandes. How is Energy Consumed in Smartphone Display Applications? In *ACM HotMobile*, 2013.

[10] M. Curti, A. Merlo, M. Migliardi, and S. Schiappacasse. Towards Energy-Aware Intrusion Detection Systems on Mobile Devices. In *IEEE HPCS*, 2013.

[11] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege Escalation Attacks on Android. In *Information Security*, 2011.

[12] M. Dong and L. Zhong. Self-Constructive High-Rate System Energy Modeling for Battery-Powered Mobile Systems. In *ACM MobiSys*, 2011.

[13] M. Dong and L. Zhong. Chameleon: A Color-Adaptive Web Browser for Mobile OLED Displays. *IEEE Trans. on Mobile Computing*, 11(5), 2012.

[14] D. Ferreira, E. Ferreira, J. Goncalves, V. Kostakos, and A. K. Dey. Revisiting Human-Battery Interaction with an Interactive Battery Interface. In *ACM UbiComp*, 2013.

[15] X. Gao, D. Liu, H. Wang, and K. Sun. PmDroid: Permission Supervision for Android Advertising. In *IEEE SRDS*, 2015.

[16] W. Jung, Y. Chon, D. Kim, and H. Cha. Powerlet: An Active Battery Interface for Smartphones. In *ACM UbiComp*, 2014.

[17] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha. DevScope: A Nonintrusive and Online Power Analysis Tool for Smartphone Hardware Components. In *CODES+ISSS*, 2012.

[18] H. Kim, J. Smith, and K. G. Shin. Detecting Energy-Greedy Anomalies and Mobile Malware Variants. In *ACM MobiSys*, 2008.

[19] M. Lentz, J. Litton, and B. Bhattacharjee. Drowsy Power Management. In *ACM SOSP*, 2015.

[20] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *ACM CCS*, 2012.

[21] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker. eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones. In *USENIX NSDI*, 2013.

[22] T. Martin, M. Hsiao, D. S. Ha, and J. Krishnaswami. Denial-of-Service Attacks on Battery-powered Mobile Computers. In *IEEE PerCom*, 2004.

[23] M. Martins, J. Cappos, and R. Fonseca. Selectively Taming Background Android Apps to Improve Battery Lifetime. In *USENIX ATC*, 2015.

[24] G. Metri, W. Shi, M. Brockmeyer, and A. Agrawal. BatteryExtender: An Adaptive User-Guided Tool for Power Management of Mobile Devices. In *ACM UbiComp*, 2014.

[25] R. Mittal, A. Kansal, and R. Chandra. Empowering Developers to Estimate App Energy Consumption. In *ACM MobiCom*, 2012.

[26] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *USENIX Security*, 2013.

[27] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices. In *ACM HotNets*, 2011.

[28] 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 *EuroSys*, 2012.

[29] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-Grained Power Modeling for Smartphones Using System Call Tracing. In *EuroSys*, 2011.

[30] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake? Characterizing and Detecting No-Sleep Energy Bugs in Smartphone Apps. In *ACM MobiSys*, 2012.

[31] A. Shye, B. Scholbrock, and G. Memik. Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures. In *MICRO*, 2009.

[32] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh. Who Killed My Battery: Analyzing Mobile Browser Energy Consumption. In *WWW*, 2012.

[33] K. N. Truong, J. A. Kientz, T. Sohn, A. Rosenzweig, A. Fonville, and T. Smith. The Design and Evaluation of a Task-Centered Battery Interface. In *ACM UbiComp*, 2010.

[34] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring. In *USENIX ATC*, 2012.

[35] L. Zhang, M. S. Gordon, R. P. Dick, Z. M. Mao, P. Dinda, and L. Yang. ADEL: An Automatic Detector of Energy Leaks for Smartphone Applications. In *CODES+ISSS*, 2012.

[36] 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 *CODES+ISSS*, 2010.

[37] M. Zhang and H. Yin. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *NDSS*, 2014.

[38] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE S&P*, 2012.

[39] Y. Zhou and X. Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *NDSS*, 2013.