Silent Battery Draining Attack Against Android Systems by Subverting Doze Mode

Ting Chen, Haiyang Tang,
Kuang Zhou and Xiaosong Zhang
School of Computer Science and Technology
University of Electronic Science and Technology of China
Chengdu, Sichuan, China
Email: brokendragon@uestc.edu.cn

Xiaodong Lin
Faculty of Business and Information Technology
University of Ontario Institute of Technology
Oshawa, Ontario, Canada
Email: Xiaodong.Lin@uoit.ca

***Abstract*—Doze mode, which was introduced from Android 6.0 aiming at reducing battery consumption when the device is unused for a long time. This work firstly reveals the internal details of the battery-saving feature, especially about the state transitions. Furthermore, we discover several defects in Androids device drivers associated with doze mode. By exploiting the defects, we implement various proof-of-concept attacks that could drain battery without acquiring any permissions by subverting doze mode. The proposed attacks are silent (hardly discerned by normal users), because they keep hidden when the smartphone is in use, while letting benign applications do battery-intensive work when the smartphone is unused rather than consuming excessive power by the attacks themselves. Google has confirmed that our attacks can reduce battery life. Finally, we discuss how to defend against the proposed attacks.**

I. Introduction

Smartphone sales have overtaken personal computers since 2011 in the market of computing platforms [1]. By the end of 2016, the global users of smartphones will surpass 2 billion, according to eMarketer [2]. Among all mobile OSes, Android dominated with an 82.8% market share in 2015 [3]. As a consequence, Android is now an ideal target for attackers.

Hardware enjoys an exponential development following Moores Law, whereas battery-related technologies do not grow at the same pace. Hence, batteries become scarce resources for smartphones since they are not always plugged. To extend battery life, various techniques have been applied, including the built-in sleeping mechanism, user-interaction-oriented low-power techniques [4], power-saving middleware [5], optimized wakelock placement [6], and power management without the involvement of device drivers [7].

Besides, Android introduced a new power-saving mechanism so called doze mode from Android 6.0 (code name: Marshmallow) that was released in September 2015. Doze mode can reduce battery consumption by switching the smartphone to *IDLE* state when the smartphone is unused for a long period [8]. In *IDLE* state, some functionalities such as network access, Wi-Fi scans, and sync operations are delayed [8], resulting in minimal power consumption.

In this work, we firstly reveal the internal details of doze mode by scrutinizing its source code. We believe our work is the first to uncover its design and implementation in such detail. Then we report several defects in Android's device drivers that implement doze mode, allowing battery exhausting attacks. Afterwards, we implement several proof-of-concept attacks that could deplete power resources without acquiring any permissions by subverting doze mode. We describe the proposed attacks as silent because they are hard to be discerned by normal users. Specifically, the attacks (1) keep hidden when the smartphone is in use; (2) allow benign applications to waste power rather than consuming excessive power by themselves. Finally, we discuss how to defend against the proposed attacks.

In brief, this work has three contributions:

- We uncover the design and implementation details of doze mode, especially about its internal state transitions and how to drive smartphone to move from one state to another.
- We discover several defects in related device drivers. Based on our observation, we implement various proof-of-concept attacks that could increase battery consumption silently.
- We discuss about approaches to battle against the proposed attacks.

Google has confirmed that the proposed attacks can reduce battery life.

II. Related Work

This section focuses on the related works about battery draining attacks. Readers who are interested in the approaches, techniques, mechanisms, strategies, architectures to extend battery life are suggested to refer to the works about green computing [9]. Besides, we do not discuss benign but flawed applications which consume excessive power, for example, applications containing forever loops.

Since the first report about battery depletion attacks in 1999 [10], this kind of attacks have received attentions in literature. Battery draining attacks could be classified into two categories according to their attacking targets. One category attacks fixed facilities, such as cloud infrastructures [11], data centers [12], aiming at increasing energy bill or greenhouse gasses. The other category targets at mobile devices with limited battery capacity. Those attacks can damage the usability of mobile devices by exhausting their battery. Obviously, our proposed attacks belong to the second category.

Fiore *et al.* proposed to drain battery stealthily by sending the victim's browser with unhearable audio files [13], for example, sounds below 20Hz. As a result, the power is wasted by playing unhearable music. Fiore *et al.*'s attack is easy to detect because users can find the abnormality of battery consumption incurred by the web browser. Besides, users can avoid the attack easily by leaving the malicious HTML5 webpage.

Researchers found that Android applications can deplete battery (deliberately or unintentionally) by misusing power management APIs. To reduce battery consumption aggressively, Android exports wakelock-related APIs to application programmers. Hence, applications can keep the smartphone awake by acquiring a wakelock, and then allow it to sleep after releasing the wakelock. However, programmers sometimes forget to release wakelocks in each path [14], [15], or place wakelocks in wrong places [6], [16], incurring power waste or faulty program logic. Various program-analysis-based approaches have been proposed to deal with wakelock-related battery draining attacks [6], [14], [15], [16].

NANSA [17] is the closest work to ours, which holds a partial wakelock, preventing CPU going to sleep and then stimulates benign applications to do power-intensive work when the screen is off. Like our work, the malware itself does not consume obvious battery resources. Instead, the battery is depleted by benign applications since the smartphone can not go to sleep. The major difference between our work with NANSA lies in that our work attempts to break doze mode, rather than the sleeping mechanism. According to Android's official website, doze mode ignores wakelocks when the smartphone is in *IDLE* state [8]. Therefore, NANSA can not subvert doze mode.

III. Dive Into Doze Mode

A. Overview

Doze mode is such a new feature that the authoritative documents concerning it we can find are only on Android's official website [8]. This subsection overviews the power-saving feature mainly according to the official descriptions [8].

If a smartphone is unplugged and stationary for a long time, with the screen off, it enters the *IDLE* state. In *IDLE* state, the smartphone reserves power by restricting applications access to the network and CPU-intensive services. Moreover, doze mode defers applications jobs, syncs, and standard alarms. Periodically, the smartphone exits *IDLE* state and does the deferred jobs. After a short while, the smartphone goes back to *IDLE* again.

Applications have to adapt to doze mode due to the restrictions in *IDLE* state. To this end, Android provides APIs like *setAndAllowWhileIdle* and *setExactAndAllowWhileIdle* that can fire alarms (very infrequently) even in *IDLE* state. Setting by *setAlarmClock*, alarms can be fired normally because it forces the smartphone to leave *IDLE* state. To maintain a

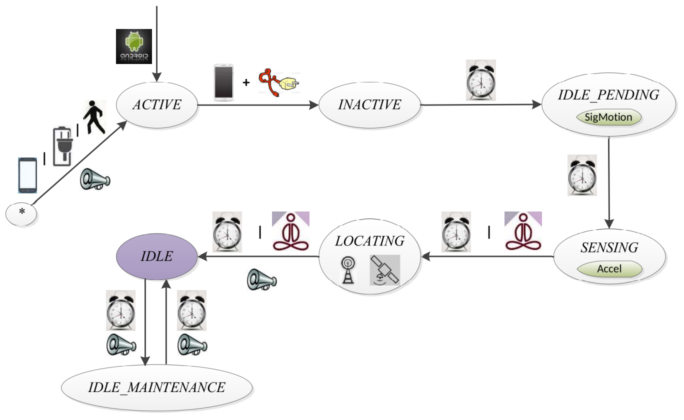

Fig. 1. State transitions

persistent network connection even in *IDLE* state, Google suggests programmers use Google cloud messaging services.

In a few cases, the restrictions applied by doze mode are too strict to one application. The application can ask users to add it to a whitelist. Note that, any third-party applications can not manipulate the whitelist directly. Instead, only the applications with system permission, for example the *Settings*, can edit the whitelist. Obviously, Android does not encourage normal applications to be exempted from doze mode. Moreover, the whitelisted applications are just partial exempt from doze. In other words, the restrictions on jobs, syncs, and regular *AlarmManager* alarms still apply.

B. State Transitions

Due to the lack of public documents, we have to dig into the source of doze mode in order to reveal more interesting details. The new feature is mainly implemented in two Android's kernel drivers: *com.android.server.DeviceIdleController* and *com.android.server.AnyMotionDetector*. *DeviceIdleController* defines 7 states, which are *ACTIVE*, *INACTIVE*, *IDLE_PENDING*, *SENSING*, *LOCATING*, *IDLE*, and *IDLE_MAINTENANCE*. When the smartphone starts, the system state is initialized as *ACTIVE*. The smartphone should stay in one state at any time. When the smartphone switches to *IDLE* state, the restrictions mentioned in [8] are applied. In *IDLE_MAINTENANCE* state, the system runs the deferred jobs. The other states can be considered as preparations for stepping into *IDLE* state. We have to note that Android M, the preview version of Android 6.0 defines 5 states. Detailed state transitions are depicted in Fig.1.

ACTIVE→INACTIVE: When (1) users turn off the screen and (2) unplug the smartphone, the system switches from *ACTIVE* to *INACTIVE* immediately. *DeviceIdleController* can be informed when screen and charging events occur by registering a broadcast receiver.

INACTIVE→IDLE_PENDING: When the smartphone stays in *INACTIVE* for a long period (*i.e.*, *mInactiveTimeout*), the system changes to *IDLE_PENDING*. When the smartphone enters *IDLE_PENDING*, *DeviceIdleController* starts to monitor whether the user does significant motions, for example, walking and cycling. *DeviceIdleController* can be informed when significant motions happen by registering an event listener in a *SIGNIFICANT_MOTION* sensor object.

IDLE_PENDING→SENSING: When the smartphone stays in *IDLE_PENDING* for a long period (*i.e.*, *IDLE_AFTER_INACTIVE_TIMEOUT*), the system changes to *SENSING*. In *SENSING* state, *DeviceIdleController* starts to check for "any motion" that is tracked in the *AnyMotionDetector* driver. From technical perspective, *AnyMotionDetector* gets an accelerometer object, and then judges the smartphone is moved or stationary according to the data collected from the accelerometer.

SENSING→LOCATING: When (1) the smartphone stays in *SENSING* for a long period (*i.e.*, *SENSING_TIMEOUT*) or (2) *AnyMotionDetector* reports the smartphone is stationary, the system changes to *LOCATING*. If the later happens, *DeviceIdleController* sets a global variable *mNotMoving* as true. In *LOCATING* state, *DeviceIdleController* registers a location service through GSM, WiFi, and GPS (if GSP module is available) to perceive location changes.

LOCATING→IDLE: When (1) the smartphone stays in *LOCATING* for a long period (*i.e.*, *LOCATING_TIMEOUT*), or (2) *AnyMotionDetector* reports the smartphone is stationary, or (3) the location service observes location changes while *mNotMoving* is true, the system moves from *LOCATING* to *IDLE*. Meanwhile, *DeviceIdleController* broadcasts an intent with the type *ACTION_DEVICE_IDLE_MODE_CHANGED* to inform other components that the systems state has changed.

IDLE→IDLE_MAINTENANCE: When the smartphone stays in *IDLE* for a long period (*i.e.*, *mNextIdlePendingDelay*), the system switches to *IDLE_MAINTENANCE* in order to do the jobs which had been deferred in *IDLE* state. Without *IDLE_MAINTENANCE*, the functionalities of the smartphone could be impaired. Besides, *DeviceIdleController* broadcasts the *ACTION_DEVICE_IDLE_MODE_CHANGED* intent when the system enters *IDLE_MAINTENANCE*.

IDLE_MAINTENANCE→IDLE: To save battery aggressively, after a short period (*i.e.*, *mNextIdleDelay*), the system goes back to *IDLE* again. Through experiments, we observe that one *IDLE_MAINTENANCE* window lasts for about ten minutes. When state changes, *DeviceIdleController* also broadcasts the *ACTION_DEVICE_IDLE_MODE_CHANGED* intent.

****→ACTIVE:*** Any states except *ACTIVE* will switch to *ACTIVE* directly if any of the following events happen (1) the user turns on the screen, (2) the smartphone is in charging, (3) *AnyMotionDetector* reports the smartphone is moved. When the state changes to *ACTIVE*, the *ACTION_DEVICE_IDLE_MODE_CHANGED* intent is broadcasted.

In the whole state transition circle, the *AlarmManager* plays an important role that controls state transitions triggered by timeout. The *AlarmManager* service is obtained when *DeviceIdleController* starts. Meanwhile, a pending intent of the type *ACTION_STEP_IDLE_STATE* is created, allowing the *AlarmManager* to send the pending intent at a later time.

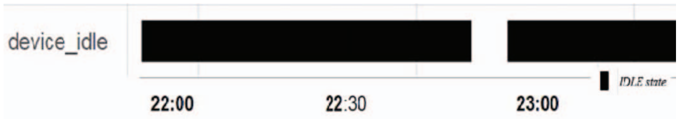

Fig. 2. *IDLE* windows without attacks

For example, when *SENSING_TIMEOUT* expires, the *AlarmManager* sends the *ACTION_STEP_IDLE_STATE* intent to *DeviceIdleController* to drive the smartphone from *SENSING* to *LOCATING*.

IV. Battery Draining Attacks

A. Defects in Device Drivers

Through scrutinizing the source code of doze mode, we discover two defects. The first is that there are no restrictions in sending *ACTION_STEP_IDLE_STATE* intent, allowing any third-party applications to change system's state. Though the mentioned intent is defined privately in *DeviceIdleController*, hackers can dig it out like us and take advantage of it.

The second lies in that Android puts no restrictions in invoking the public API *setAlarmClock*. As stated in official website [8], alarms set with *setAlarmClock* could be fired normally because the system exits *IDLE* state shortly before those alarms fire. As a consequence, malware can subvert doze mode by invoking *setAlarmClock* frequently.

For comparison, we present the *IDLE* windows of a smartphone without attacks in Fig.2 which is depicted by Battery Historian [18]. We collected data for about ten hours. For the sake of presentation, we show the data collected in the first 1h30m. Please note that the black box indicates the smartphone is in *IDLE* state. To quicken experiments, we force the tested smartphone into *IDLE* state at 21h:52m:10s by the command *adb shell dumpsys deviceidle force-idle* and then leave the smartphone untouched.

During 10 hours, there are three *IDLE_MAINTENANCE* windows, that are 22h:37m:35s $\sim$ 22h:42m:37s, 00h:34m:55s $\sim$ 00h:44m:57s, and 04h:39m:23s $\sim$ 04h:49m:44s. The accumulated time in *IDLE_MAINTENANCE* state is about 25m25s, whereas the smartphone stays in *IDLE* in the rest of the time. Hence, without attacks, *IDLE* state occupies about 95.8% of the total testing period.

B. Attack by Sending The Internal Intent

The basic idea of the attack is to force the smartphone to move from *IDLE* to *IDLE_MAINTENANCE* immediately after it enters *IDLE* state, because *IDLE_MAINTENANCE* state does not save power. The attack consists of two steps: (1) detecting whether the smartphone is in *IDLE* state; if so (2) changing state by sending *ACTION_STEP_IDLE_STATE* intent. Hence, the key point is how to detect *IDLE* state. We find two ways, one public while the other is unknown before. Please note that system state is stored in a private variable *mState* of *DeviceIdleController*. Unfortunately, we can not get access to *mState* directly.

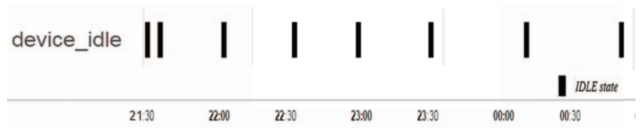

Fig. 3. *IDLE* windows under attack, *IDLE* state is detected by a public API, the state transition is driven by sending *ACTION_STEP_IDLE_STATE* intent

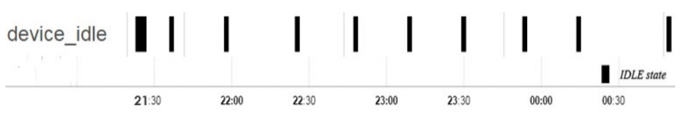

Fig. 4. *IDLE* windows under attack, *IDLE* state is detected by the side channel method, state transition is driven by sending *ACTION_STEP_IDLE_STATE* intent

1) Public method: Developers can invoke a public API *isDeviceIdleMode* to check whether current state is *IDLE* [19]. By registering a broadcast receiver to receive *ACTION_DEVICE_IDLE_MODE_CHANGED* intent dynamically, applications can check state only when the state changes. Unsurprisingly, the officially recommended method corresponds to our analysis of source code that *DeviceIdleController* broadcasts an *ACTION_DEVICE_IDLE_MODE_CHANGED* intent whenever the state transits to *IDLE*, *IDLE_MAINTENANCE*, or *ACTIVE*. The *IDLE* windows under attack are shown in Fig.3.

It is noticeable that the tested smartphone exits *IDLE* state immediately after *DeviceIdleController* receives the *ACTION_STEP_IDLE_STATE* intent. Besides, each *IDLE* window only lasts for one or two seconds. Between each *IDLE* window is an *IDLE_MAINTENANCE* window, which lasts for about ten minutes. Consequently, the *IDLE* windows are accumulated to merely 61s during the 10-hour experiment.

2) Side-channel method: We propose to detect *IDLE* model without invoking power-related APIs based on the fact that Android puts several restrictions when the smartphone is in *IDLE*. Hence, basically, we propose to check whether some functionalities, such as network access and Wi-Fi scans, are usable periodically. In the implementation of the proof-of-concept attack, we (1) create a working thread containing a forever loop that checks a number in every 7s; (2) set an alarm which will be fired 5s later by invoking *setExact*; (3) flip the number and set the alarm again when receives the alarm. Recall that the alarms set by *setExact* will be deferred when the smartphone is in *IDLE*. Therefore, if the number keeps unchanged in two consecutive checks, we detect the *IDLE* state. The *IDLE* windows under attack are shown in Fig.4.

Unsurprisingly, Fig.4 is similar with Fig.3 indicating that we are successful to detect *IDLE* state using side channel. The only discernible difference is that *IDLE* windows last for slightly longer than the last experiment. For example, the longest *IDLE* window in this experiment is 12s. Moreover, the accumulative time of all *IDLE* windows during the 10-hour experiment is 427s, about 7 times longer than that using the public API. Although the proposed side channel method is not so sensitive to state changes, it is stealthier because no power-related APIs are called.

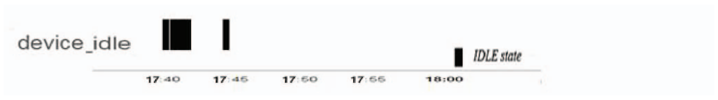

Fig. 5. Battery draining attack by abusing *setAlarmClock*

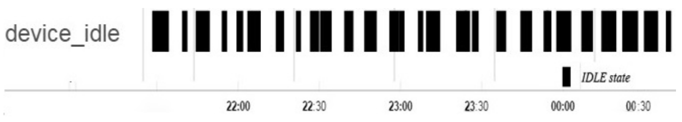

Fig. 6. The attack for disordering state transitions

C. Attack by Abusing The Public API

In this proof-of-concept attack, we create a working thread containing a forever loop that set an alarm by invoking *setAlarmClock* in every 10s. The attacking result is shown in Fig.5. We can see that the system exits *IDLE* state (at 17h:36m:35s) immediately after it enters *IDLE* (at 17h:36m:31s). More obviously, after that, the system can not go to *IDLE* at all. This attack even does not need to know whether the current state is *IDLE*. It is obvious that the fastest way an attacker can drain battery resources is invoking *setAlarmClock* periodically. But some attackers may prefer the approaches mentioned in Section IV.B, because *setAlarmClock* is a public power-related API that may attract the attentions of security experts.

D. Disorder State Transitions: A Risk

We believe that it could be a potential attack in principle by sending *ACTION_STEP_IDLE_STATE* intent irregularly. Our analysis shows that this intent plays a critical role in maintaining state transitions. However, attackers can send the intent when they want, resulting in disordered state transitions. In the proof-of-concept attack, we send the intent in every T seconds. T is a random number chosen from (0, 20). Fig.6 gives the attacking result.

Fig.6 shows significant differences with Fig.3 to Fig.5, indicating that the attack does not aim at draining battery power. Moreover, the *IDLE* windows show an obviously different trend with those in normal condition as shown in Fig.2. To develop robust applications, programmers have to consider two program workflows, one for *IDLE* state, one for the others due to the functionality restrictions in *IDLE* state. We believe the proposed attack may make trouble with the applications and system components if they can not react to state transitions properly.

E. Summary

To summarize, we propose (1) two ways to detect *IDLE* state, one public, one unknown before; (2) two methods to change states based on the internal intent or one public API respectively. The proposed attacks have the following features.

- The attacks are not discernible when the smartphone is in use.

- No dangerous permissions are required.
- The battery draining attacks do not deplete power by themselves. Instead, they allow other benign applications to waste power.
- The combination of side-channel-based *IDLE* state detection and internal-intent-based state transition even do not need to call any power-related APIs.
- All proposed attacks are examined successfully in all releases of Android 6.0, including Android M Preview. The latest release we have tested is Android-6.0.1_r10 until January 21, 2016.

V. Case Study

In this section, we present a case study about evaluating the battery consumption in normal condition and under attack in practice. The experiment is conducted on an LG Nexus 5X installed with Android 6.0.1. We install an application, YouKu (China's YouTube) on it. We run a downloading task in YouKu that fetches episodes from YouKu's official servers. In normal condition, we force the smartphone to enter *IDLE* state by the *adb*'s command, immediately after starting the downloading task. Then, the smartphone is left untouched with the screen turned off for 3 hours. After the experiment, the battery capacity reduces from 100% to 95% (*i.e.*, the consumption is 5%). Besides, about 196MB is downloaded.

For comparison, we set the attacking scene as follows. We use side-channel to detect *IDLE* state and send the internal intent to compel the smartphone to leave *IDLE* state, because it is the stealthiest way among the proposed attacking methods. We launch the attack, followed by starting the downloading task, then turn off the screen and keep the smartphone untouched. The experiment also lasts 3 hours. The result is that the battery capacity reduces from 100% to 68% (*i.e.*, the consumption is 32%). During the experiment, about 5200MB is downloaded. Hence, the experiment validates the existence of the defects of Android doze mode and the effectiveness of the attack. We can expect a more significant difference in battery consumption between the attacking scene with the normal condition, if more applications are running in the background.

Fig.7 demonstrates the battery consumption by each application under attack. The figure is depicted by the default battery statistics tool on LG Nexus 5X, that ranks the applications in descending order of the consumed power. For convenience, we omit several items between Youku and our attacking application (DeviceIdleModeTest). The observation is as expected that YouKu consumes the most battery power among all applications (83.95%), while the attacking application itself consumes little (0.15%) that is fifth from the bottom. Hence, normal users are likely to mistakenly consider YouKu as the malware rather than the real one, just depending on battery consumption.

As expected, YouKu downloads more under attack than in normal condition, because the downloading task is delayed in *IDLE* state. In normal condition, the accumulative time in

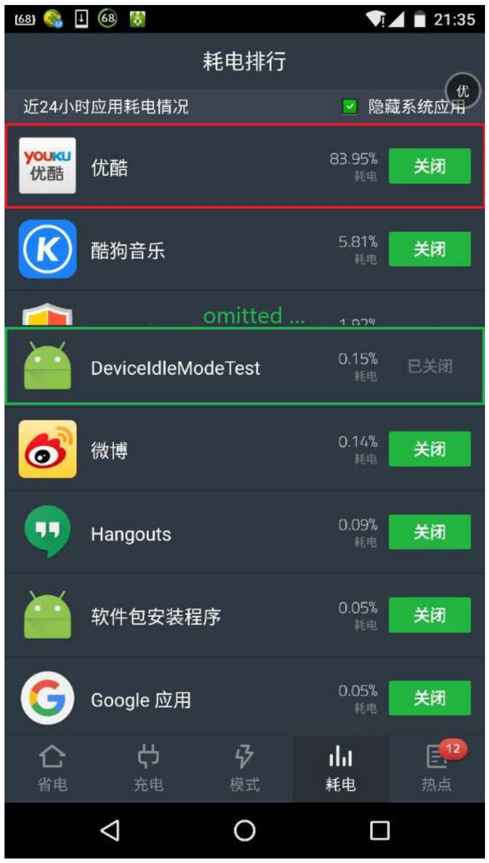

Fig. 7. Battery consumption by each application under attack

IDLE state is about 2h51m04s, so the downloading task is active just for about 9m during the 3-hour experiment.

VI. Defend Against The Proposed Attacks

When submitting an application to Google Play, the application could be analyzed both statically and dynamically before published, in order to discover malicious behaviors. However, the market access mechanism is insufficient to find the proposed attacks. First, the attacks do not acquire permissions, defeating any permission-based approaches [20], [21]. Second, API-based methods [22], [23] can observe the invocations of *isDeviceIdleMode* and *setAlarmClock*. However, the two public APIs may not be good indicators of malicious behaviors, because Android allows benign applications to use them. Therefore, API-based methods are likely to produce high false positives. More seriously, the combination of side channel and the internal intent does not call any power-related APIs, completely defeating API-based methods.

Third, Java reflection and dynamic code loading can defeat static program analysis that aims at finding the manipulation of *ACTION_STEP_IDLE_STATE* intent. Afterwards, the proposed attacks can circumvent Googles dynamic analysis (*i.e.*, Google Bouncer), because they keep hidden when the smartphone is in use.

It is straightforward to detect the proposed attacks by monitoring battery usage. However, normal users can not find the attacks using the default battery tool in the smartphone, even if alert users can be aware of the abnormal battery consumption. Using professional tools like Battery Historian [18], analysts can find the abnormality of *IDLE* windows. However, it is difficult to pinpoint the malware because the attacks themselves do not consume excessive power.

We believe the best countermeasure is fixing related device drivers by Google. For example, Android should associate a privilege permission with the *ACTION_STEP_IDLE_STATE* intent or simply forbid third-party applications to send it. We are aware that Android has listed the broadcasts which should not be sent by third-party applications in *AndroidManifest.xml*. In other words, many broadcasts have been protected by Android system. However, the intent *ACTION_STEP_IDLE_STATE* is not included. We plan to investigate how many unprotected internal intents and how can we influence Android system by exploiting them as future work. Second, Android should put some restrictions in calling *setAlarmClock*, for example, only allowing applications to invoke the API very infrequently.

VII. Conclusion

Doze mode is a new feature introduced from Android 6.0, attempting to reduce battery consumption. In this work, we firstly reveal the design and implementation details of doze mode by inspecting its source code. Then we report the discovered defects in related device drivers. Taking advantage of those defects, we implement various proof-of-concept attacks that could drain battery silently. Google has confirmed that the proposed attacks can reduce battery life. Finally, we discuss countermeasures.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China, No.61402080, No.61572115, No.61502086, No.61572109, and China Postdoctoral Science Foundation founded project, No.2014M562307.

References

[1] C. Taylor, "Smartphoe sales overtake PCs for the first time," http://mashable.com/2012/02/03/smartphone-sales-overtake-pcs/#8_JVn5UuEsqn, 2012.

[2] "2 billion consumers worldwide to get smart(phones) by 2016," http://www.emarketer.com/Article/2-Billion-Consumers-Worldwide-Smartphones-by-2016/1011694, 2014.

[3] "martphone OS Market Share, 2015 Q2," http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2015.

[4] W. Song, N. Sung, B. G. Chun and J. Kim, "Reducing energy consumption of smartphones using user-perceived response time analysis," in Proceedings of the 15th Workshop on Mobile Computing Systems and Applications, pp. 1-20, 2014.

[5] H. Galeana-Zapién, C. Torres-Huitzil and J. Rubio-Loyola, "Mobile phone middleware architecture for energy and context awareness in location-based services," *Sensors*, 14(12): 23673-23696, 2014.

[6] F. Alam, P. R. Panda, N. Tripathi, N. Sharma and S. Narayan, "Energy optimization in Android applications through wakelock placement," in Proceedings of Europe Conference and Exhibition on Design, Automation and Test, pp. 1-4, 2014.

[7] C. Xu, F. X. Lin and L. Zhong, "Device drivers should not do power management," in Proceedings of 5th Asia-Pacific Workshop on Systems, pp. 1-11, 2014.

[8] "Optimizing for doze and app standby," http://developer.android.com/training/monitoring-device-state/doze-standby.html#restrictions, 2015.

[9] A. Hooper, "Green computing," *Communication of the ACM*, 51(10): 11-13, 2008.

[10] F. Stajano and R. J. Anderson, "The resurrecting duckling: security issues in ad-hoc wireless networks," in Proceedings of the 7th International Workshop on Security Protocols, pp. 172-194, 1999.

[11] F. Palmieri, S. Ricciardi, U. Fiore, M. Ficco and A. Castiglione, "Energy-oriented denial of service attacks: an emerging menace for large cloud infrastructures," *The Journal of Supercomputing*, 71(5): 1620-1641, 2013.

[12] F. Palmieri, S. Ricciardi and U. Fiore, "Evaluating network-based DoS attacks under the energy consumption perspective: new security issues in the coming green ICT area," In Proceeding of International Conference on Broadband and Wireless Computing, Communication and Applications, pp. 374-379, 2011.

[13] U. Fiore, F. Palmieri, A. Castiglione, V. Loia and A. De Santis, "Multimedia-based battery drain attacks for android devices," in Proceedings of 11th IEEE Conference on Consumer Communications and Networking, pp. 145-150, 2014.

[14] A. Jindal, A. Pathak, Y. C. Hu and S. Midkiff, "Hypnos: understanding and treating sleep conflicts in smartphones," in Proceedings of the 8th ACM European Conference on Computer Systems, pp. 253-266, 2013.

[15] 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 Proceedings of the 10th International Conference on Mobile systems, Applications, and Services, pp. 267-280, 2012.

[16] A. Jindal, A. Pathak, Y. C. Hu and S. Midkiff, "On death, taxes, and sleep disorder bugs in smartphones," in Proceeding of the Workshop on Power-Aware Computing and Systems, pp. 1-5, 2013.

[17] M. Bauer, M. Coatsworth and J. Moeller, "NANSA: a no-attribution, no-sleep battery exhaustion attack for portable computing devices," 2015.

[18] "Battery Historian 2.0," https://github.com/google/battery-historian, 2016.

[19] "PowerManager," http://developer.android.com/reference/android/os/PowerManager.html, 2016.

[20] Z. Aung and W. Zaw, "Permission-based Android malware detection," *International Journal of Scientific and Technology Research*, 2(3): 228-234, 2013.

[21] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas and G. Álvarez, "Puma: permission usage to detect malware in Android," in Proceedings of the International Joint Conference CISIS12-ICEUTE12-SOCO12, pp. 289-298, 2013.

[22] D. J. Wu, C. H. Mao, T. E. Wei, H. M. Lee and K. P. Wu, "Droidmat: Android malware detection through manifest and API calls tracing," in Proceeding of Seventh Asia Joint Conference on Information Security, pp. 62-69, 2012.

[23] Y. Aafer, W. Du and H. Yin, "DroidAPIMiner: mining API-level features for robust malware detection in Android," *Security and Privacy in Communication Networks*, pp. 86-103, 2013.