An Empirical Study of Android Alarm Usage for Application Scheduling

Mario Almeida[1], Muhammad Bilal[1],

Jeremy Blackburn[2], and Konstantina Papagiannaki[2]

[1] Universitat Politecnica de Catalunya,
mario.almeida@est.fib.upc.edu
[2] Telefonica Research

Abstract. Android applications often rely on alarms to schedule background tasks. Since Android KitKat, applications can opt-in for *deferrable* alarms, which allows the OS to perform alarm batching to reduce device awake time and increase the chances of network traffic being generated simultaneously by different applications. This mechanism can result in significant battery savings if appropriately adopted.
In this paper we perform a large scale study of the 22,695 most popular free applications in the Google Play Market to quantify whether expectations of more energy efficient background app execution are indeed warranted. We identify a significant chasm between the way application developers build their apps and Android's attempt to address energy inefficiencies of background app execution. We find that close to half of the applications using alarms do not benefit from alarm batching capabilities. The reasons behind this is that (i) they tend to target Android SDKs lagging behind by more than 18 months, and (ii) they tend to feature third party libraries that are using non-deferrable alarms.

1 Introduction

Todays mobile devices support a diverse set of functionality, much of which is not dependent on active user interaction. Many tasks are performed in the background, which has very clear impact on battery life and mobile data usage [4]. The impact is substantial enough that reducing and mitigating it has been the focus of a significant amount of research and development.

A promising set of solutions aim to shape applications' traffic [5,3,9,16,12,8], but suffer from severe limitations. These techniques ignore application-protocol interactions and lack integration with applications and OSes, often *increasing* energy consumption due to retransmissions and/or signaling issues [17] in real-world scenarios. Other works [4,17,14,15] highlight the need for better application knowledge and/or integration with OS/platforms.

Alarms are Android's integrated application execution scheduling mechanism (used, e.g., for background network activity) and are a primary vehicle for executing the traffic shaping techniques. Alarms are so critical to the functionality

of Android that they have been a hot topic at the last two Google IO conferences and a popular target for energy concerns[3,4]. One way Android mitigates Alarms' negative impact is *batching*, which can reduce total device awake time while increasing the chance that traffic from different applications can occur simultaneously. As of KitKat, developers can opt-in to have their alarms be *deferrable* which makes batching by the OS easier.

Unfortunately, the success of batching depends on the correct usage of alarm APIs by applications: apps themselves determine the deferrability, trigger time, and repetition interval of alarms. This leads to the situation Park et al. [13] discovered in their study of 15 Android applications: alarms are often unnecessarily set as non-deferrable. However, it is totally unclear how widespread such a practice is and thus its impact on the efficacy of alarm scheduling is unknown.

Since there is no indication that alarms will cease to be the preferred application level scheduling mechanism within Android, future design and development should be informed with an understanding of how developers use the current alarm APIs. Thus, in this paper we perform a large-scale study of 22,695 real applications from the Google Play Market (to the best of our knowledge, the largest such study to date) in order to find evidence of alarm API adoption delays and their impact on the performance of the Android OS; more specifically, the effectiveness of alarm batching in Android. We investigate how many apps use alarms, what type of alarms they use, differences in alarm usage by application category, and whether alarms are being used by apps themselves or by 3rd party libraries. We find that a shocking 46% of apps with alarms do not take advantage of Android alarm scheduling capabilities due to either targeting old SDK versions or their use of 3rd party libraries. We further discuss and analyze the problems behind Android SDK adoption and propose possible directions for improving alarm batching across applications.

2 Android Alarms

Alarms are the primary mechanism Android provides to allow applications to schedule background activities. Alarms come in two flavors: 1) time critical alarms, and 2) non-time critical. The first type is called an *exact* alarm, and the second is known as an *inexact* or *deferrable* alarm. The OS is expected to execute exact alarms on schedule, but can delay the execution of deferrable alarms. Deferrable alarms are particularly interesting due to the manner in which Android can leverage them to improve power efficiency. For example, batching alarms to multiplex network activity of multiple applications can reduce the wake up frequency of the device's radio.

Decisions related to what type of alarms to use are left to the application developers since, in theory, only they have the insight necessary to assess the impact a delayed alarm will have on their app. Unfortunately, developers will often optimize for profit (e.g., ensuring fresh ads are retrieved/displayed as often

[3]GIO'15, Doze - http://goo.gl/KEJURc

[4]GIO'14, Project Volta - https://goo.gl/aebnwF

Alarm API	SDK < 19	SDK = 19 - 22	SDK = 23
set	Exact	Inexact	Inexact
setRepeating	Exact	Inexact	Inexact
setInexactRepeating	Inexact	Inexact	Inexact
setExact	NA	Exact	Exact
setWindow	NA	Inexact	Inexact
setAndAllowWhileIdle	NA	NA	Inexact
setExactAndAllowWhileIdle	NA	NA	Exact

Table 1: Behavior of alarms based on the Target SDK level. We note that although our dataset was collected before SDK 23 was available, the continuing effort put into the alarm API highlights the critical nature of Android Alarms.

as possible) and usability rather than energy efficiency. A second wrinkle with alarm types is that developers are free to define what Android SDK their app targets. If the device the application is installed on has a different SDK than the targeted one, a compatibility mode applies which in some cases can alter the app's behavior. This can have interesting consequences for alarms because the default functionality for a given Alarm API call might differ between SDK versions (see Table 1). E.g., if the targeted SDK version is less than 19, all API calls but `setInexactRepeating` create exact alarms. For SDK 19+, a new call to explicitly create exact alarms is introduced, and the behavior for the previously existing calls is changed to create inexact alarms.

Therefore, applications with exact alarms are those which: 1) have target SDK lower then 19 *and* use `set` or `setRepeating` calls or 2) use `setExact`.

Applications with inexact alarms are those which: 1) target $\geq$ SDK 19 *and* use `set` or `setRepeating` calls or 2) use `setInexactRepeating` *or* `setWindow` calls. It is important to note, however, that despite being able to create inexact alarms for SDK $<$ 19, alarm batching across applications is only available for devices with Android KitKat (SDK 19) or higher [1].

Alternatively, Android apps can also use a new alternative designed to facilitate correct implementation of alarms and to reduce alarm occurrences based on app requirements: the JobInfo API. The JobInfo API provides new triggering conditions based on, for example, network (metered/unmetered) and device state (e.g., idle/charging), backed by more sophisticated retry mechanisms to avoid unnecessary execution, in turn allowing tuning apps with respect to battery consumption. The JobInfo APIs were introduced 6 months prior to our experiments (SDK 21), however, *none* of the apps in our dataset made use of them.

3 Results

3.1 Dataset

To understand the use of alarms in Android apps, we crawl Google Play and download up to 564 of the most popular free apps for each Google Play category.

Removing duplicates we are left with 22,695 unique apps. Although studying the most popular apps is clearly biased, it is justified for two reasons. First, these apps are more likely to be optimized than the least popular apps due to their associated revenues. Second, since these apps account for the majority of downloads, they are more representative of what users actually have on their mobile devices. This is evidenced by Viennot et al. [18] who found that the top 1% of most downloaded apps account for over 81% of the total downloads in November 30, 2013. To the best of our knowledge, our dataset (May 2015) should account for around 1.5% of the total apps of the market in 2015 (AppBrain[5] claims around 1.5 Million apps in the first quarter of 2015).

For each of the 22,695 apps, we extract their manifest; an XML file that contains application meta-data, such as the application package name, components, permissions, etc. Three of the properties listed in the manifest are the minimum, maximum, and target SDK. The target SDK is the Android API level (e.g., Android 4.4 Kitkat has an API level of 19) that the application was developed for, and, as discussed earlier, determines the types of alarms available to the developer. By default, apps that do not define a target SDK have their target default to the minimum SDK.

3.2 Static Analysis

Since the focus of this study is understanding how alarms are being used by apps, we perform static analysis on the apps we crawled. We first decompile each of the apps, which provides us with assembly-like code (smali). We then statically analyze the smali code to locate occurrences of Android Alarm API calls. In our database, each occurrence of an alarm/jobinfo API call is registered along with the respective application, alarm API, smali file name, line and annotations to the method where it occurs. Since some free apps are likely to have ads [6], opposed to their paid version, we can analyze the API call location and correlate it with the ad libraries (Section 3.5). Annotations are useful since specific methods can use the `TargetAPI` annotation to denote that they want to execute the method in compatibility mode. For apps, the meta-data (target SDK, internet usage, category) is registered. In particular, we are interested in correlating target Android SDKs with the the number of alarm API calls and their usage within different apps and app categories.

3.3 Impact of Target SDK on Alarms

As mentioned previously, the target SDK of an application can significantly affect the behavior of its alarms. As a first step towards understanding the impact of the chosen target SDK, we plot the distribution of SDK targets from our dataset in Fig. 1. It shows that despite the efforts of Google to promote the use of their newer SDKs (e.g., Google IO conferences, extensive documentation and application design guidelines), the majority of the popular apps target SDKs

[5] http://www.appbrain.com/stats/number-of-android-apps

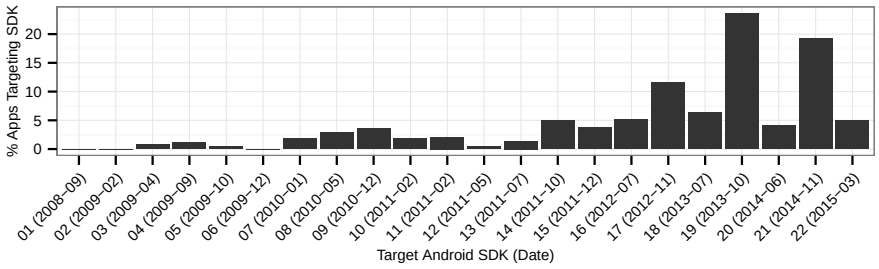


Fig. 1: Percentage of apps that define each Android SDK as the target SDK in their manifests. NB: We were unable to extract the target SDK from 1.5% of apps in our dataset.

Alarm Type	SDK < 19	SDK >= 19
AlarmInexact	8.49%	52.91%
AlarmExact	44.05%	2.31%
Alarm	46.06%	53.49%

Table 2: Fraction of apps with exact and inexact alarms grouped by SDK version. Dates represent the release dates of each Android SDK. Note that an application can make use of both exact and inexact alarms.

that were released more than 18 months ago (up to and including SDK 19, represent 71.6%). Close to half (48%) the apps target SDKs lagging behind by more than 21 months.

From the perspective of alarms, we note that 47.23% of apps have a target SDK lower than 19; i.e., they are still going to use the older alarm API behavior with defaults oriented towards exact alarms. Out of the 22+K apps in our dataset, 47.25% use alarms. Of the apps that use alarms, we see that 53.49% have target SDK versions above 19, while 46.06% target older SDKs (Table 2). As annotations can affect the targeted APIs on a per method basis, we confirmed that only 2% of the apps with SDK < 19 had occurrences of the `TargetAPI` annotation in methods containing alarm calls.

The major apparent difference between SDK < 19 apps and SDK ≥ 19 apps is the flip-flop in usage of exact and inexact alarms: only 2.31% of apps targeting SDKs ≥ 19 define exact alarms in contrast to the 44.05% of apps targeting < 19. We note that this change might not necessarily be the result of developers being aware of the impact of exact alarms, but rather an end result of targeting newer SDKs.

The reason behind Android being so conservative with maintaining the previous alarm behavior even in newer versions of Android is to avoid apps from becoming unstable when updating. Since only 2.31% of the apps targeting SDKs ≥ 19 use the exact alarm API call, if we would consider the hypothesis that apps

Application	SDK	Downloads	Version
es.lacaixa.mobile.android.newwapicon	17	1M-5M	2.0.17
com.cg.tennis	14	10M-50M	1.6.0
com.linkedin.android	15	10M-50M	3.4.8
com.rovio.angrybirds	18	100M-500M	5.0.2
com.cleanmaster.security	17	100M-500M	2.5.1
com.shazam.android	16	100M-500M	5.3.4
com.instagram.android	16	500M-1000M	6.20.2

Table 3: Example of popular and regularly updated apps with more than one million downloads and with target SDK older than 19 months (as of May 2015).

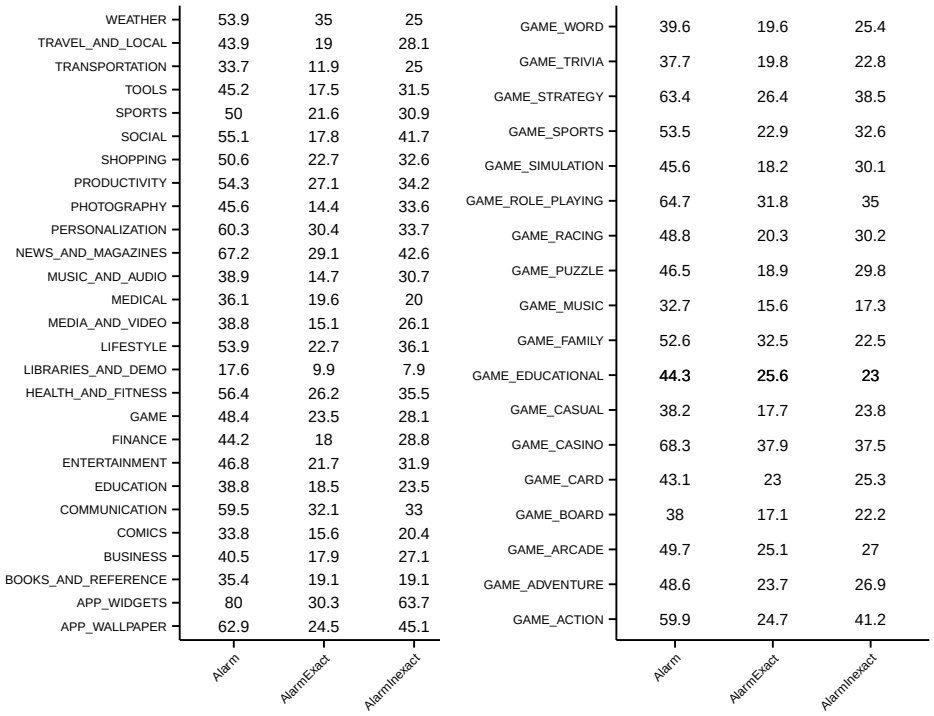
with target SDK $\geq$ 19 updated from an older SDK, it is probable that either most apps did not have exact time constraints after all or that the ones that do willingly avoided updating their SDKs. If the first is true, then Android is being very conservative with their approach regarding alarms batching behavior, which has a big impact on the power consumption of devices. Although we did not study apps update rates, it would have been interesting to determine if the second case holds by, for example, determining how many of these apps were updated after the release of Android API level 19. Our intuition is that even regularly updated apps often do not update their SDK. As an example, in Table 3, we show a few well known apps which, by the time of our study, had target SDKs lower than 19. Which means that these apps are unable to utilize the new energy efficient alarm APIs provided by the latest Android SDKs.

Even if the device is supported and up-to-date, apps can target old versions of the Android SDKs, which can have a negative impact on the overall performance of the device. Our results clearly demonstrate that there is slow adoption of new SDK versions by application developers. More importantly, we see that despite the efforts to make Android more energy efficient with respect to alarm handling (e.g., through JobInfo and the introduction of inexact alarms), backwards compatibility (a necessary evil at this point due to fragmentation), lack of developer awareness about new SDK benefits, and misuse of alarms by developers makes it hard to succeed.

3.4 Type of Alarms depending on app category

Considering the conservative behavior of Android regarding non-deferrable alarms, we now wonder which type of apps require exact alarms. To this end, we explore how different categories (as retrieved from Google Play) of apps make use of alarms (Figure 2).

Surprisingly, categories of apps such as widgets (80%), wallpapers (63%) and personalization (60%) have a bigger fraction of apps with alarms than communication (59%) and social categories (55%). While having more alarm definitions does not necessarily mean that there will be more alarm occurrences during runtime, we found that, for example, there are 308 widget apps defining



(a) All categories (b) Game apps

Fig. 2: Percentage of apps per category (avg. 523 apps) that have any kind of alarms, have exact alarms and inexact alarms. Due to the high amount of Game categories, a) groups this categories into GAME. Note that an application can make use of both exact and inexact alarms.

repeating alarms (`setRepeating` and `setInexactRepeating`) (in Sec. 3.6 we manually analyze some of these apps). Regarding time critical alarms, the five application categories with most apps with exact alarms are respectively: casino games (37.9%), weather (35%), family games (32.5%), communication (32.1%) and role-playing games (31.8%). Finally, the average number of alarms defined by apps per category is shown in Figure 3.

The widgets category not only has the largest number of apps with alarms and one of the highest time critical alarms usage (30.3%), but also it also has the highest average number of alarms (4.9) defined within an application. The analyzed apps had up to 70 alarm definitions[6], e.g., Whatsapp defines 28 alarms, Instagram 11 and Facebook only 2. Again, we point that although Facebook

[6] `com.ecare.android.womenhealthdiary`

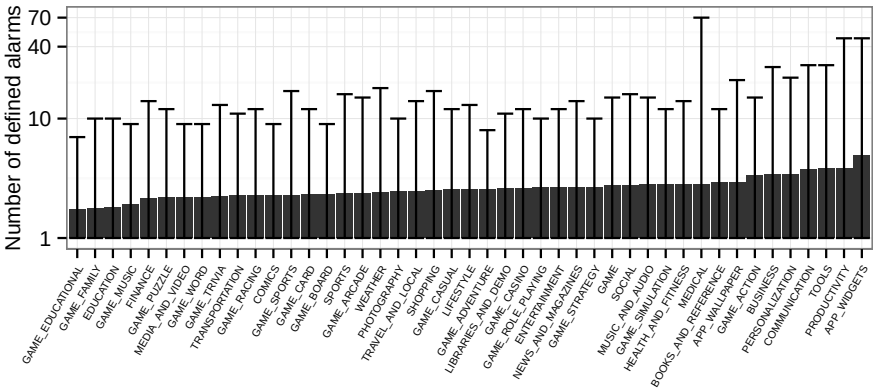


Fig. 3: Average number of alarms per application for each Google Play category. Error bars depict the maximum number of alarms for each category.

has only 2 alarm definitions, its alarms are actually very frequent during runtime (Section 3.6).

3.5 The impact of 3rd party libraries

From our experience while studying apps, we have also seen that many apps have a big proportion of 3rd-party content. For example, consider Skype, only about 36.4% of its code is actually Skype-specific functionality, while 31.8% accounts for 3rd-party SDKs (e.g., roboguice, jess, qik, android support) and 32.8% belongs to ads/analytics (e.g., flurry, Microsoft ads).

Hence one important aspect to check is whether defined alarms are native to the application itself or if they originate from 3rd party libraries. We analyzed the package names of the files where the alarms were detected and compared them to 93 ads and analytics libraries available for Android, retrieved from a public list provided by AppBrain[7]. The library package names and matches were manually confirmed to eliminate false positives.

Figure 4 shows the number of apps where alarms defined by these ads/analytics libraries were found. Alarms of ads/analytics libraries found in less than 10 apps are omitted (e.g., cellfish, inmobi, mopub). Although our approach might not cover all possible ads/analytics libraries, we were able to detect that 10.65% of the unique apps (22.55% of apps with alarms) have alarms defined by third-party ads/analytics, and around 10.42% of all alarm API calls found belong to these libraries.

Finally, considering the number of alarms defined across all apps, we have discovered that 31.5% of all alarms are repeating, while nearly 40.5% of alarms

[7]http://www.appbrain.com/stats/libraries/ad

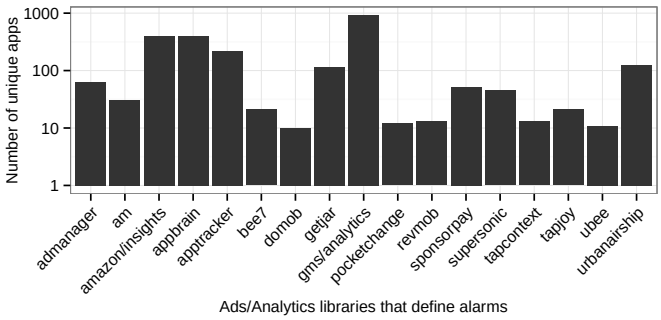


Fig. 4: Number of apps with alarms defined by third-party ads/analytic libraries.

are non-deferrable. Regarding 3rd-party ads and analytics libraries, their alarms account for 10.4% of all alarm occurrences. From these occurrences, 72.6% of them are repeating and 22.3% of them are non-deferrable. Even though we only explored ads/analytics, given the large coverage of these 3rd-party libraries, optimizing their resource consumption and having them use inexact alarms (e.g., using `TargetAPI` annotation) would certainly lead to appreciable gains in terms of energy consumption.

3.6 Occurrence of alarms at execution time

To confirm the impact of alarms on Android KitKat (SDK 19), the first to introduce batching by default, we perform two experiments. The experiments use two different sets of 30 apps. The first set is the top 30 most popular free apps of the Google Play market. The second set is the 30 apps with the largest number of `setRepeating` alarm definitions that also target SDK lower than 19. The latter was chosen since these alarms should be deferred if the target SDKs were set to ≥ 19 and notably includes apps with >1K to >500M downloads.

For each experiment we flash a new Android firmware (KitKat), install the 30 apps and create new accounts with no contacts/friends when needed (e.g., Gmail, Facebook, Twitter, etc.). All apps were started once to ensure Android gives them permission to execute on reboot if required, and then the phone is left on for around 30 minutes. We then reboot the phone, turn off its screen, and let it run for around 3 hours. Finally, we gather the alarm and wakeup counts as reported by Android Dumpsys (`adb shell dumpsys alarm`) for the installed apps. Both experiments were repeated to confirm the patterns we observed.

There were a total of 261 alarms registered by the apps in our first experiment. Only 53 (20%) caused the device to wakeup and we found no significant correlation between the number of registered alarms and the number of alarms that woke the device ($r = 0.11$, $p = 0.55$). That said, we were quite surprised to find that the two Facebook apps (messenger and the regular app) were responsible for the majority of wakeups (15 per hour). Upon closer examination,

we determined that they were waking the phone to maintain a connection to a message queue, even though the accounts used had literally zero social activity.

A total of 1,041 alarms were registered by apps in our second experiment. Of these, 636 (61%) woke up the device and we found a strong and significant correlation between the number of registered alarms and the number of alarms that woke the device ($r = 0.86$, $p < 0.01$). The worst offending application was the social network Spoora (10K-50K downloads) which registers *only* `setRepeating` alarms and also has its SDK target set to 9. Spoora was responsible for 372 wakeups and is a clear example of the negative impact of careless alarm usage which could be easily mitigated by simply targeting a newer SDK. Interestingly, this type of scenario is not unique to less popular apps: Norton Security and Antivirus (10M-50M downloads) has a target SDK of 17 and caused 141 wakeups.

From these two experiments we have clear evidence that poor alarm API usage can cause substantial impact on the device, and it is not limited to small time developers. In particular, our results highlight how even a simple misconfiguration (i.e., setting a target SDK too low) can have significant negative impacts in execution behavior. In the future, we intend to run similar experiments on a larger scale, taking direct battery measurements, manually modifying the target SDK to quantify exactly how the impact on battery consumption changes between target SDKs, and more closely examining the relationship between alarm type declaration and registration/wakeups.

4 Discussion and Conclusion

Research on energy efficiency in mobile devices tends to propose solutions focused on batching activity to amortize the cost of waking up the mobile device and its radio. The efficiency of such solutions depends on the ability of the operating system to schedule background activity at the most appropriate time. In Android, alarms are a popular mechanism to schedule background activities. To understand apps' usage of alarms, we crawled the Google Play store and downloaded over 22 thousand of the most popular apps.

We found that nearly 50% of apps define their alarms to be *non-deferrable* by the operating system, thus hamstringing Android's ability to optimize scheduling at all. When examining the prevalence of alarms, we found that they existed across all categories of apps with some having up to 70 alarms declared. For apps with alarms, 22.5% have them defined by 3rd party ads/analytics libraries they use, and these libraries account for at least 10.4% of all declared alarms. We also showed the inefficiencies of alarms by manually analyzing 60 apps at runtime, finding apps waking up the device an inordinate number of times.

While Android fragmentation has been studied in the past [7,10,11], it was generally approached from the perspective of the wide distribution of Android versions, heterogeneous hardware, and lack of updates. In this work we have revealed another facet of this problem: even if the device is supported and up-to-date, apps often target old versions of the Android SDKs, which can have a negative impact on the overall performance of the device. Via static analysis, we discovered that a substantial number of apps' alarms are non-deferrable due to

targeting older versions of the Android SDK and that by simply changing the target SDK to > 19 these apps would likely benefit from advanced OS alarm scheduling mechanisms. Furthermore, while previous work [10] which studied a much smaller set of 10 open-source apps found that 28% of method calls were outdated with a median lag time of 16 months, we also show that in the case of alarms, close to half the API calls are outdated by more than 18 months.

Ads and analytics are a particularly interesting subject of study since they have been shown to have a big impact on energy consumption [6]. We found that the majority of alarms related to ads and analytics are repeating, meaning that they most likely result in background operations that might have no real end-user benefit. This seems to be a problem that is core to Android in particular, since iOS does not have a direct analogue to alarms and has an extremely limited background execution environment [2]. Since from our experience a large proportion of Android apps make use of third-party code, future large-scale studies of energy consumption, optimization, and alarm usage should focus on common third-party libraries.

When we examined alarm usage at runtime we discovered that the implications of the static analysis held true for the most part. The apps with the highest number of defined alarms were in fact executing the alarms at an exceedingly high rate. In one egregious case, a single application was responsible for 372 wakeups in a 3 hour period.

This work serves as an initial large-scale look into alarms and their impact. Overall, our findings indicate that research on energy efficiency on mobile devices needs to incorporate an understanding around the use of alarms. Deeper examinations into the use and abuse of Android alarms should provide more fruitful insight and solutions, leading to increased energy efficiency and device performance.

References

1. Alarmmanager. http://goo.gl/ncrGaO
2. iOS Developer Library: Background execution. https://goo.gl/xZd16w
3. Athivarapu, P.K., Bhagwan, R., Guha, S., Navda, V., Ramjee, R., Arora, D., Padmanabhan, V.N., Varghese, G.: RadioJockey: mining program execution to optimize cellular radio usage. In: Proceedings of the 18th annual international conference on Mobile computing and networking (2012)
4. Aucinas, A., Vallina-Rodriguez, N., Grunenberger, Y., Erramilli, V., Papagiannaki, K., Crowcroft, J., Wetherall, D.: Staying online while mobile: the hidden costs. In: Proceedings of the ninth ACM conference on Emerging networking experiments and technologies (2013)
5. Balasubramanian, N., Balasubramanian, A., Venkataramani, A.: Energy consumption in mobile phones: a measurement study and implications for network applications. In: Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference (2009)
6. Gui, J., Mcilroy, S., Nagappan, M., Halfond, W.G.: Truth in advertising: The hidden cost of mobile ads for software developers. In: Proceedings of the 37th International Conference on Software Engineering (2015)

7. Han, D., Zhang, C., Fan, X., Hindle, A., Wong, K., Stroulia, E.: Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs. In: 19th Working Conference on Reverse Engineering (2012)
8. Higgins, B.D., Reda, A., Alperovich, T., Flinn, J., Giuli, T.J., Noble, B., Watson, D.: Intentional networking: opportunistic exploitation of mobile network diversity. In: Proceedings of the sixteenth annual international conference on Mobile computing and networking (2010)
9. Liu, H., Zhang, Y., Zhou, Y.: TailTheft: leveraging the wasted time for saving energy in cellular communications. In: Proceedings of the sixth international workshop on MobiArch (2011)
10. McDonnell, T., Ray, B., Kim, M.: An Empirical Study of API Stability and Adoption in the Android Ecosystem. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance (2013)
11. Mulliner, C., Oberheide, J., Robertson, W., Kirda, E.: PatchDroid: Scalable Third-party Security Patches for Android Devices. In: Proceedings of the 29th Annual Computer Security Applications Conference (2013)
12. Nguyen, N.T., Wang, Y., Liu, X., Zheng, R., Han, Z.: A Nonparametric Bayesian Approach for Opportunistic Data Transfer in Cellular Networks. In: Wireless Algorithms, Systems, and Applications (2012)
13. Park, S., Kim, D., Cha, H.: Reducing Energy Consumption of Alarm-induced Wake-ups on Android Smartphones. In: Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications (2015)
14. Qian, F., Wang, Z., Gao, Y., Huang, J., Gerber, A., Mao, Z., Sen, S., Spatscheck, O.: Periodic transfers in mobile applications: network-wide origin, impact, and optimization. In: Proceedings of the 21st international conference on World Wide Web (2012)
15. Shi, C., Joshi, K., Panta, R.K., Ammar, M.H., Zegura, E.W.: CoAST: collaborative application-aware scheduling of last-mile cellular traffic. In: Proceedings of the 12th annual international conference on Mobile systems, applications, and services (2014)
16. Vergara, E.J., Nadjm-Tehrani, S.: Energy-aware cross-layer burst buffering for wireless communication. In: Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet (2012)
17. Vergara, E.J., Sanjuan, J., Nadjm-Tehrani, S.: Kernel level energy-efficient 3g background traffic shaper for android smartphones. In: Proceedings of the 9th International Wireless Communications and Mobile Computing Conference (2013)
18. Viennot, N., Garcia, E., Nieh, J.: A Measurement Study of Google Play. In: The 2014 ACM international conference on Measurement and modeling of computer systems (2014)