A Lightweight Framework for Fine-Grained Lifecycle Control of Android Applications

Yuru Shao
University of Michigan
yurushao@umich.edu

Ruowen Wang
Samsung Research America
ruowen.wang@samsung.com

Xun Chen
Samsung Research America
xun.chen@samsung.com

Ahemd M. Azab
Samsung Research America
amazab80@gmail.com

Z. Morley Mao
University of Michigan
zmao@umich.edu

Abstract

The lifecycle of Android apps is dynamically managed by the system in an ad hoc manner, which leads to apps' abusing lifecycle entry points to automatically start up and gaming the priority-based memory management mechanism to evade being killed. Such apps exhibit *diehard behaviors* that keep them long-running in the background, resulting in excessive battery consumption and device performance degradation. Existing battery-saving features are far from being effective in restricting diehard behaviors, due to the lack of systematic, fine-grained control of app lifecycle.

In this paper, we propose the Application Lifecycle Graph (ALG), a holistic modeling of system-wide app lifecycle. We present a lightweight runtime framework that builds ALG and utilizes it to realize fine-grained lifecycle control of apps. The framework exposes APIs that provide ALG information and lifecycle control capabilities to developers and device vendors, empowering them to leverage the framework to implement rich functionalities. Evaluation results show that the proposed framework is competent and incurs low performance overhead. It introduces 4.5MB additional memory usage on average, and approximately 5% and 0.2% CPU usage during system booting and at idle state.

CCS Concepts • **Software and its engineering → Dynamic analysis**; **Operating systems**; *Software performance*.

Keywords Mobile application analysis, Application behaviors, Lifecycle management, Runtime monitoring

EuroSys '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303956

ACM Reference Format:
Yuru Shao, Ruowen Wang, Xun Chen, Ahemd M. Azab, and Z. Morley Mao. 2019. A Lightweight Framework for Fine-Grained Lifecycle Control of Android Applications . In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3302424.3303956

1 Introduction

Mobile app lifecycle is dynamically managed by the system and is opaque to users. Due to the constrained resources on mobile devices, the Android system controls each app's lifecycle based on their demands and task priorities. In particular, Android imposes looser restrictions on app lifecycle and allows background execution without user awareness. On the one hand, Android's permissive lifecycle control gives apps more flexibility to react to user interactions and system events timely, and thus enables rich functionalities, such as background video recording. On the other hand, however, it also opens doors for apps to directly or indirectly alter their lifecycles. In fact, apps can easily *abuse* their entry points to automatically start up in the background, requiring no user interaction, and *game* the lifecycle management mechanism to evade being killed.

We call app behaviors that make changes to their lifecycles for the purpose of 1) keeping long-running in the background or 2) evading being killed *diehard behaviors*. Apps exhibiting such diehard behaviors are thus *diehard apps*. Diehard apps can cause battery drain and device performance degradation. Since they are oftentimes completely invisible while running in the background, it is hard for normal users to be aware of their existence and what they are actually doing. It is reported that the Amazon Shopping app operates in the background so that it remains up to date with current offers and promotions, causing high battery usage [24]. People also have privacy concerns on such apps, as they could stealthily and constantly collect sensitive user data, such as geolocations [13, 16, 37].

Essentially, diehard apps exploit two fundamental problems in Android app lifecycle. First, apps can have multiple entry points that are by default accessible to other apps on

Table 1. The changes cause lifecycle fragmentation, *i.e.,* an app's lifecycle is inconsistent in different Android frameworks.

Android version	Improvements	Diehard techniques affected
Marshmallow (6.0)	Doze, App Standby	Alarm Manager, Long-lived TCP connections
Nougat (7.0)	Fixing notification bug, Doze on the go, Background Optimization	Hiding notifications
Oreo (8.0)	Job scheduler improvements, Background Execution Limitation	Static broadcast receivers
Pie (9.0)	Background Restrictions	Foreground services

```
// full class name: com.android.Laucher.Se
public class Se extends Service {
  ...
  // onDestroy() callback is always called by
  // the system when a service gets killed
  public void onDestroy() {
    super.onDestroy();
    ...
    // Restart itself (the 2nd argument is the
    // target service that will be started).
    Intent i = new Intent(this.context, Se.class);
    i.setFlags(268435456);
    i.setAction("com.dai.action");
    i.setAction("com.tdz.action");
    this.startService(i);
  }
  ...
}
```

Figure 1. Code snippet of the HummingBad malware, decompiled by JEB Decompiler. The target of the intent object (local variable `i`) is set to `Se.class`, meaning that the service attempts to restart itself while being killed.

the same device. In addition to the user starting an app explicitly, the app can be launched by the system or another app as well, requiring no user involvement. For example, the system broadcasts signal strength changes so that apps potentially affected by weak signal strength can take actions accordingly. A diehard app, however, can also claim to handle the event and it will be automatically launched by the system to process signal changes. Second, and more importantly, app lifecycle is not strictly enforced and is hard to enforce. The lifetime of an app process is determined by the system through a combination of the parts of the app that the system knows are running, how important these things are to the user, and how much overall memory is available in the system [17]. Since apps are a sophisticated interplay between custom code and the system framework, they are able to game the system to indirectly manipulate their own lifecycle states. For instance, apps with foreground services are believed to have higher priorities. Knowing this, diehard apps usually start foreground services to escalate their priorities even though it is not a necessary functionality for them. Moreover, apps can directly alter their component lifecycle. App components implement a series of callbacks which are invoked by the system through its lifetime, but there is no limitation on what they can do inside each callback. Malicious apps have been exploiting the loosely enforced app lifecycle to be diehard, *e.g.,* the notorious HummingBad malware. As Figure 1 shows, when its service gets killed, it attempts to restart the service immediately. Not all developers are well educated or are willing to follow the guidelines. They get things done in ways they see fit, sometimes causing diehard behaviors unintentionally.

New but ad hoc features (summarized in Table 1) have been introduced to Android in an effort to limit background apps, for example, background optimization [8], Doze, and App Standby [15]. They affect diehard behaviors to a certain extent, but unfortunately, they cannot fundamentally solve the diehard behavior problem and they all have obvious limitations. First, there are legitimate cases where apps need to keep running in the background, but Background Optimization and Background Execution Limitation are both too coarse-grained, either allowing or disallowing *all* background activities. It is difficult to balance the trade-off between app functionality and user experience. For end users, neither zero control or excessively strict control is helpful. Second, apps can always find "creative" ways to bypass background restrictions. Diehard techniques evolve along with the Android framework. For example, developers have come up with approaches to escalating process priority so that their apps will not be killed when available memory is low [25]. Malware variants use social engineering to bypass a battery-saving process and stay active in the background [3].

We acknowledge that certain apps may have legitimate reasons for being diehard. However, they should comply with system regulations and development guidelines for providing the best user experience. We argue that diehard behaviors violate the system's app lifecycle control and they should be better managed. Comprehensive modeling of app lifecycle which can enable fine-grained lifecycle control is desired.

In view of this need, we make the first effort towards providing fine-grained control of app lifecycle. In particular, we categorize diehard techniques that are used by apps to keep long-running, from which we learn a valuable insight that diehard apps create high-priority app components and/or develop interdependence between component callbacks, between app components, or between other apps. The complicated app lifecycle makes it challenging to realize reliable, systematic detection and restriction. To tackle this, we

propose *application lifecycle graph (ALG)*, a systematic, informative, and precise description of app lifecycle. The problem of diehard behavior detection thus can be transformed into operations on a directed graph (*i.e.,* the ALG). Specifically, the interdependence created by diehard apps can be identified as cycles, and diehard behaviors are reflected on the ALG as edges with special properties. Leveraging ALG, we develop a lightweight framework that enables flexible and fine-grained app lifecycle enforcement at runtime. We collect and analyze 17,598 apps from Google Play and a third-party app market. Results show that diehard behaviors are very common among apps. To our surprise, diehard behaviors sometimes come from third-party libraries an app integrates, making the host app a diehard parasite.

In summary, this paper makes the following contributions:

- We propose app lifecycle graph (ALG), a fine-grained, precise description of system-wide app lifecycle. ALG allows us to transform diehard behavior detection and restriction problems into efficient graph-based operations, *i.e.,* cycle detection and edge pruning.
- Leveraging ALG, we design and implement a lightweight runtime framework for fine-grained control of app lifecycle. This framework enables the development of new functionalities in app lifecycle management by exposing a set of easy-to-use APIs.
- We perform the first study on diehard behaviors in the wild. We find that diehard behaviors are common among apps from both Google Play and a third-party app market. One interesting observation is that app developers may not intentionally make their apps diehard, but the third-party libraries they integrate have diehard behaviors.

2 Background and Motivation

We first introduce background to app lifecycle management on Android. We then use real-world examples to demonstrate the limitations of existing app lifecycle management mechanism as our motivation.

2.1 Component Lifecycle

Components are the essential building blocks of Android apps. There are four types of components that can be used within an app, *i.e.,* Activity, Service, Broadcast Receiver, and Content Provider. Each type of component has its distinct lifecycle. A component transitions through different lifecycle states during an app's execution and the framework calls its lifecycle callbacks [7] at each state change. It is the developers' responsibility to define how a component behaves in response to lifecycle state changes. For example, while a Service is being created, its `onCreate()` is called. Developers override the default callbacks, but there is no restriction on what they can do in each lifecycle callback. An app's lifecycle is far more complicated than the aggregation of all its components' lifecycles, because there exist control flows and data flows among components.

Inter-component communications (ICCs) occur both within individual apps and between different apps. ICC relies primarily on the exchange of asynchronous messages called *Intents*, which can carry extra data in the form of key-value pairs. The ICC initiator creates an Intent instance and puts into it the target component information. ICCs enable complicated collaborations across apps. For instance, the camera app allows users to share photos on social media conveniently, by sending an Intent object with photo information to the social app. If the target app is not running, the system starts it so that desired operations can be completed. Because of this design, an app can *wake up* other apps through ICC.

2.2 Memory Management

By design, Android does not immediately kill app processes when they are switched to the background. They are cached in the background so that they can be quickly recovered when the user switches back. In this way, the system can speed up reopening apps if needed, but it also easily gets into a low free memory state, where it has to shut down certain processes in order to provide memory to processes that are more immediately serving the user [18]. If an app process gets killed, all components residing in that process are consequently destroyed. When deciding which processes to kill, the system weighs their relative importance to the user. For example, it more readily shuts down a process hosting activities that are no longer visible on screen, compared to a process hosting visible activities. The decision of whether to terminate a process, therefore, depends on the states of the components in that process. The system uses Activity Manager Service to track the importance of processes and reflect their importance by setting the `oom_adj` (latest kernels use `oom_score_adj`) value of the process under `/proc/PID/`. The higher the `oom_adj` value is, the more likely this process gets selected by the kernel's low memory killer (LMK). Since a process may host multiple components at a time and priorities are per process, the state of one single component can affect the entire process's priority.

App components have their individual lifecycles, but the call graph of callbacks is incapable of describing an app's complete lifecycle. First, in addition to component callbacks, frequent ICCs are also part of app lifecycle. Second, an app can wake up another app when inter-process communications (IPC) occur. In fact, *fragmentation* aggravates the problem, as not all devices can be upgraded to the latest version. As of September 15th, 2018, 13 months after Android Oreo was released, 85.4% devices are still running older versions [10]. Device vendors such as Huawei customize Android and add their own power-saving features. However, they impose overly strict restrictions that blindly block all app background activities.

3 Understanding Diehard Behaviors

A comprehensive understanding of diehard behaviors can provide us insights on designing fine-grained app lifecycle control. We collect cases from popular user forums [2, 4, 5] and well-known developer sites [22, 26]. We manually analyze 23 diehard apps reported by users and thoroughly inspect the techniques discussed among developers.

A key insight we learn from the analysis of existing techniques is that *diehard apps create high-priority app components and/or develop interdependence between component callbacks, between app components, or between other apps.*

3.1 Escalating Process Priority

To evade being killed, apps attempt to trick the system into believing they are important to serving the user. As a result, their process priorities will be escalated.

Foreground service. Normally, apps are put into the background when the user goes back to the home screen or switch to another app. Android, however, allows apps to start foreground services in which continuous tasks (*e.g.,* music playing, file downloading) will not be interrupted even the user is not currently using the app. Foreground services have much higher priority and therefore they will not be easily killed. This feature has been widely abused. Apps can simply call `startForeground(int, Notification)` to turn a background service into foreground state. The system considers foreground services to be user-aware and thus not candidates for killing even under heavy memory pressure. Before Android N there are bugs in displaying notifications, which are exploited to start foreground services stealthily without user awareness.

Floating view. Apps can keep a tiny, invisible floating view in the foreground, abusing the `SYSTEM_ALERT_WINDOW` permission. It is a known problem that this permission allows an app to draw overlays on top of other apps, and it is automatically granted for apps installed from Google Play [30].

Native process. Apps are allowed to run native executables using `java.lang.Runtime.exec()` APIs outside the app processes. Unlike app processes in the Android runtime, native processes are out of the control of the Android framework's memory management. They by default have higher priority, especially when they run as daemons. The native processes may not be used to perform complicated tasks but rather to guard certain app components.

3.2 Auto-run

Apps utilize auto-run techniques in order to automatically start up after reboots and restart themselves after being killed. Different from escalating process priority, auto-run behaviors create interdependence between apps or between an app and the system. Even if the user uses task management tools [1, 11] to kill background apps, diehard apps can still manage to restart with auto-run.

Sticky service. A service makes itself "sticky" by returning `START_STICKY` from its `onStartCommand()` callback. A sticky service, if uses no other diehard techniques, will be recycled by the system when available memory is low. However, the system recreates the sticky service once it gets out of the low-memory state.

Listening to system events. The system sends out broadcasts when certain events occur. Apps that are interested in specific events get notified if they have registered corresponding broadcast receivers. For example, `SIG_STR` is broadcasted out when signal strength changes, and an app listening to this broadcast will be awakened. When a receiver is registered in the manifest and the app is not running, a new process will be created to handle the broadcast. This gives the app the chances to start other components thereafter.

Watchdog. A watchdog process is used to monitor the process that needs to keep alive. If the process being watched is dead, the watchdog restarts it immediately. Watchdog processes are usually implemented as native processes, and there are several ways to monitor another process' state (*i.e.,* running or dead). For example, the watchdog could be a native daemon which establishes a local socket channel with the app process [34]. If the socket channel is somehow broken, it means the app process is dead. In this case, the native daemon tries to restart the app process immediately.

Abusing account synchronization. Apps are allowed to create a sync adapter component that encapsulates the code for the tasks that transfer data between the device and a server. Based on the scheduling and trigger provided, Android's sync framework runs the code in the sync adapter component (no matter the app is running or not), from which other components of the app can be started.

Scheduled tasks. AlarmManager allows scheduling an app to be run at some specific point in the future, even if the app is not currently running. JobScheduler first became available in Android 5.0. Apps register jobs, specify their requirements for network and timing. The system then schedules the jobs to execute at the appropriate times. Both AlarmManager and JobScheduler are abused by apps to realize auto-run. Observables can also be used to set up periodically tasks.

Cross-app wakeup. Apps developed by the same developer and apps integrating the same SDK can work together to keep long-running. For example, all Baidu apps have the same *ShareService* which periodically looks up other Baidu apps installed on the device and tries to bind to them. Since the system starts the target app for completing inter-app ICCs, one running Baidu app can thus *wake up* all other Baidu apps that are not running.

Explicitly invoking lifecycle callbacks. The system manages app component lifecycle. Different callbacks are invoked by the framework at each stage of an app component. For example, when the system destroys a service, `onDestroy` is

called. The purpose is to give apps an opportunity to save running states and die gracefully. Apps are able to override these callbacks. They can thus abuse them by explicitly calling `onStart()` inside `onStop()` so that the component will not finish. The behavior shown in Figure 1 adds an edge to the lifecycle, creating a cycle.

4 Fine-Grained Lifecycle Control

Based on the insight we learn from diehard apps and their behaviors, we believe that in an appropriate graph representation, high-priority components can be identified as special nodes and the interdependence can be captured as cycles. We propose the Application Lifecycle Graph (ALG) to accurately describe apps' lifecycles as a whole in a fine-grained manner. Diehard behaviors are reflected on the ALG as either edges with particular properties, or cycles indicating the interdependencies between apps, app components, or component callbacks. The benefits of ALG are two-fold. First, it can capture and record all app and system events (*i.e.,* edges on the graph) that affect lifecycle. Second, it allows us to convert problems such as diehard behavior detection into graph-based problems, *i.e.,* cycle detection. We design a runtime framework that utilizes ALG to dynamically track app states and realize fine-grained lifecycle control, which overcomes limitations of static analysis based approaches that lack efficiency, scalability, and extensibility. The framework also exposes the ALG and lifecycle control capabilities as a set of APIs in order to facilitate the development of new functionalities.

4.1 Application Lifecycle Graph (ALG)

ALG models lifecycles of all installed apps in three layers. From the higher level to the lower level, they are (1) cross-app ICC graph, (2) intra-app ICC graphs, and (3) component callback graphs. We have

$$ALG = (\mathcal{N}_{app}, \mathcal{E}_{cross-app-icc})$$

where $\mathcal{N}_{app}$ is the node set and $\mathcal{E}_{cross-app-icc}$ is the edge set. Each node represents an installed app: $\mathcal{N}_{app} = \{G_{app_{0\ldots n}}\}$, $G_{app_i, i\in[0,n]}$ is the intra-app ICC graph of app i. Each edge represents a cross-app ICC event. We further define $G_{app_i} = (N_{comp}, \mathcal{E}_{intra-app-icc})$, and $\mathcal{N}_{comp}$ is a set of nodes representing app components: $\mathcal{N}_{comp} = \{G_{app_i comp_{0\ldots m}}\}$. The edge set, $\mathcal{E}_{intra-app-icc}$, represents intra-app ICCs. $G_{app_i comp_j}$ $(i \in [0,n], j \in [0,m])$ is the callback graph of component j in app i. Nodes of a callback graph are callback methods, while edges are call sequences of those callback methods: $G_{app_i comp_j} = (\mathcal{N}_{callback}, \mathcal{E}_{method-call})$.

Figure 2 illustrates the ALG structure. The top level is a graph consisting of apps (also the Android framework, which will be discussed in §4.1.1) and cross-app ICCs. For example, app_0 starts app_1 with a cross-app ICC. Each app node is actually an intra-app ICC graph, whose nodes are either app components or native binaries, and edges are intra-app ICCs. For example, app_0 has three components, among which $component_0$ starts $component_1$ and $component_1$ starts $component_2$. Each component has a callback graph that models its callback sequence.

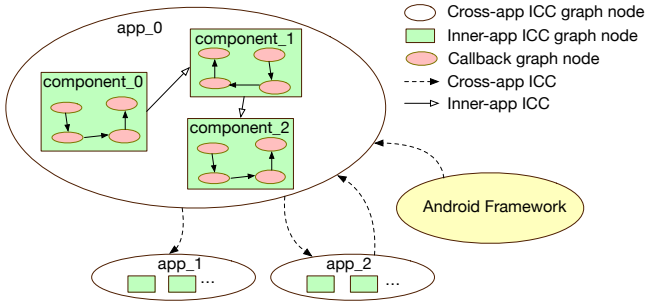

Figure 2. An ALG illustration. The Android framework is represented as a special node in the same level as apps. Edges have attributes that provide event contexts.

4.1.1 Abstract the Android Framework

The entire Android framework is abstracted into an app node in ALG, although the Android framework consists of a number of different packages and system services distributed into multiple processes. The rationale behind this abstraction is that from the apps' perspective, ICCs between the framework have no difference compared to cross-app ICCs.

Apps may interact with different system services during their lifetime. For instance, as described in §3, apps register their components to the framework, and the framework will start those registered components when required conditions are met. There are also many built-in system apps that normal apps can communicate with. Aggregating framework packages into one single node reduces graph complexity by eliminating unnecessary nodes and edges, and significantly speeds up operations on the ALG. Meanwhile, system services and system apps are critical to normal functioning of the system; they are out of the scope of our fine-grained lifecycle control. This abstraction does not have a negative impact on the precision of the ALG.

4.1.2 Lifecycle Event Context

Edges in ALG represent lifecycle-related events. We provide event context as edge attributes. We consider the four categories of attributes: (1) User interaction, *i.e.,* whether or not an app or a component is initiated by the user is an important factor for determining the legitimacy; (2) Frequency, *i.e.,* the frequency of an ICC event indicates how aggressive an app is in terms of being diehard; (3) ICC type, including app status (*i.e.,* foreground or background), triggering method call (*e.g.,* `startActivity`, `bindService`); and (4) Status, *i.e.,* enabled or disabled, for enforcing lifecycle control policies. For example, in Figure 3, ICC edges provide information on

how a component is started, and what shell command is executed to start a native component. Similarly, in Figure 4, cross-app ICC edges have information on the interactions between apps and the Android framework.

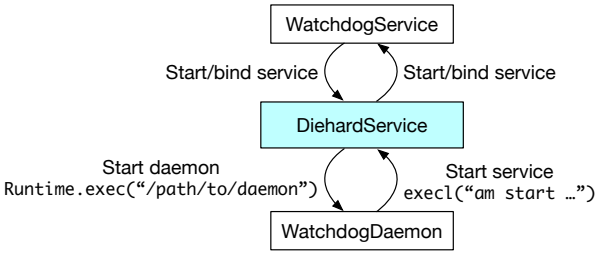

Figure 3. Partial ALG: intra-app ICC graph for an app having watchdog component. Irrelevant ALG parts are omitted.

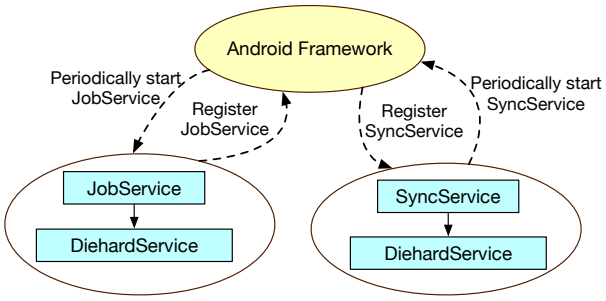

Figure 4. Partial ALG: cross-app ICC graph capturing scheduled task and account sync. Irrelevant ALG parts are omitted.

4.2 Fine-grained Lifecycle Control

To realize fine-grained, component-level app lifecycle control, we propose a lightweight runtime framework that builds ALG on-the-fly and exposes APIs to support the development of new functionalities. The advantage of a runtime system is that it can capture genuine runtime information, thus ensure accuracy, although performance overhead is inevitable and completeness cannot be guaranteed. Pure static app code analysis can gather relatively comprehensive app behaviors, but it is not precise due to the lack of source code and its inherent limitations that cause over-approximation, *e.g.*, points-to analysis [27]. The design of the framework must satisfy the following requirements.

1. Non-blocking monitoring. To reduce app perceived delay, we must not block app executions. This brings challenges in placing hooks in the Android framework for collecting runtime information.
2. ALG accuracy. Our lifecycle control framework relies heavily on the ALG. An accurate ALG is the foundation of new functionalities developed on it. However, there is no existing mechanism in the Android framework to support ICC caller component identification. In fact, very limited caller information is available, including only app UID, PID, and package name.
3. Nondisruptive control. If an app or an app component is being restricted, we need to gracefully shut it down, without causing crashes that will be perceived by the user.

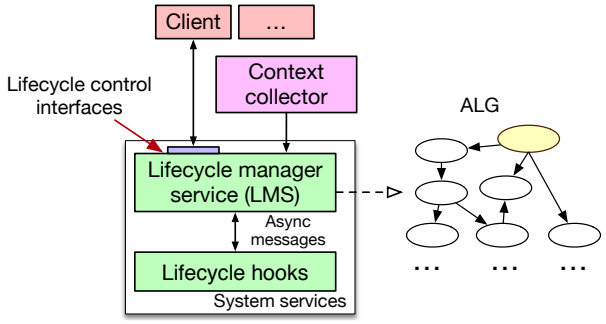

Figure 5. Overview of the framework. There could be multiple client apps that use the lifecycle control APIs.

Figure 5 depicts the architecture of our proposed framework. We add a system service, Lifecycle Manager Service (LMS), into the Android framework to maintain an ALG at runtime. To collect runtime lifecycle information we place various hooks into existing system services such as Activity Manager Service and Job Service. All hooks report collected data to LMS, which updates the ALG accordingly. Meanwhile, LMS exposes a set of APIs that provide the ALG and fine-grained lifecycle control capabilities to apps. We overcome the challenge of accurately identifying caller component of an ICC using a Context Collector (§4.2.2). These interfaces enable various use cases. System-level developers (*e.g.*, device vendors) can leverage them to restrict diehard behaviors. Developers of task manager apps and battery saver tools can use the interfaces to better manage running tasks and to implement more effective battery saving policies.

4.2.1 Lifecycle Manager Service (LMS) and Hooks

LMS works with lifecycle hooks to build and maintain the ALG. Hooks are placed into several system services for collecting runtime app lifecycle information and controlling lifecycle events. We choose to hook services instead of app logics for two reasons. First, Android adopts a client-server model where apps send their requests to handling services. For example, ICCs are eventually executed by Activity Manager Service, which acts like a switch that looks up the target app and component, thus connects the caller and the target. Placing hooks in services allows us to centralize monitoring and enforcement logics so that we can keep minimal communication channels with LMS and therefore reduce overhead. Second, we cannot trust information coming directly from the apps, because apps have the capability of manipulating its own memory and bypassing the hooks.

Table 2. APIs provided by our framework for fine-grained app lifecycle control. `Bundle` objects are essentially key-value pairs. They are used to update one or multiple edge/node properties at a time.

API	Type	Description
`AppLifecycleGraph getLifecycleGraph()`	Sync	Return a copy of ALG
`AppCompGraph getAppCompGraph(String pkg)`	Sync	Return an app component graph with given package name
`CompCallbackGraph getCompCallbackGraph(String pkg, String comp)`	Sync	Return callback graph of an app component
`void setAppProperties(Bundle p)`	Async	Set properties of an app node on the ALG
`void setAppComponentProperties(Bundle p)`	Async	Set properties of an app component
`void setCrossAppEdgeProperties(Bundle p)`	Async	Set properties of an cross-app ICC edge
`void setIntraAppEdgeProperties(Bundle p)`	Async	Set properties of an intra-app ICC edge
`void setCompCallbackEdgeProperties(Bundle p)`	Async	Set properties of component callback graph edge

For an operation that affects app lifecycle, there could be many intermediate procedures (*i.e.,* method calls) between the API being called and the internal method that eventually performs the intended operation. To balance overhead, accuracy, and extensibility, we must carefully choose appropriate locations for installing hooks. In general, we have three different hook placement options, as illustrated in Figure 6.

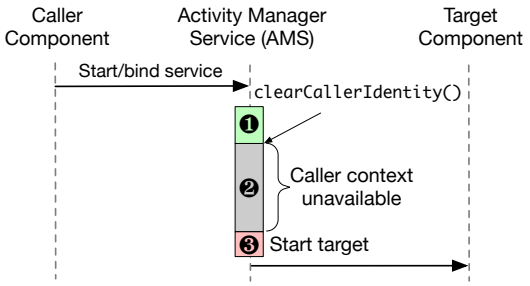

Figure 6. Hook placement options during service launching/binding ICC. The identity of the caller app is completely unavailable after the AMS calls `clearCallerIdentity()`.

Close to the caller. If hooks are close to the caller, we can more easily collect the calling app UID, PID, and package name, as they are still available until the AMS calls `clearCallerIdentity`. Nonetheless, not all lifecycle-related operations can finally reach their targets, because they could fail at any intermediate method calls. As a result, we would collect false-positive ICCs and create ALG edges that do not really exist.

Close to the target. Lifecycle hooks can also be placed close to the target component. In this case, we would have very accurate information about the target without additional efforts. The downside is we lose the caller app and component information completely.

Somewhere in-between. Placing hooks at certain points of the intermediate method calls allows us to balance accuracy and overhead. We can keep caller information before it gets cleared, and reuse intermediate return values to obtain target component identity. However, this option requires efforts in understanding system services code, which may change drastically across different Android versions. The cost of maintenance is the highest.

We choose to place lifecycle hooks close to the target, as our top goal is to ensure accuracy and eliminate false positives. The tradeoff is that we need to store caller information before it is cleared. Evaluations in §5 show that the overhead for this tradeoff is totally acceptable. Hooks could also be placed at both the caller side and the target side, but this would result in higher overhead.

4.2.2 Identify Caller Component

In Android's client-server model of app-framework interactions, apps are identified by their UID, PID, or package name, which means system services see all components of an app as a whole. The granularity of all access control mechanisms is per app, not per app component. As for our lifecycle control framework, we aim to achieve component-level granularity, and the ALG requires caller component and target component for each ICC. Identifying target component is trivial, but accurate identification of caller component is challenging.

To overcome this challenge, we modify base app component classes to attach caller component information automatically, as illustrated in Figure 7. Since everything from the app side could be manipulated by the app itself, we validate received caller component information in LMS. Specifically, all app components extend base classes from the Android SDK, *e.g.,* `android.app.Service`, `android.app.Activity`. We add into base component classes a `getIdentity()` method that returns class full name, including the package. Leveraging the polymorphism feature of the Java language, calling the method on concrete sub-class instances returns specific component names. The whole process is completely transparent to app and it requires no effort from app developers.

Everything coming from the app side cannot be trusted, because apps have the capability to manipulate anything in their own memory space. This means caller component information from client side could be manipulated if the app wants to bypass or trick our caller identification method. To

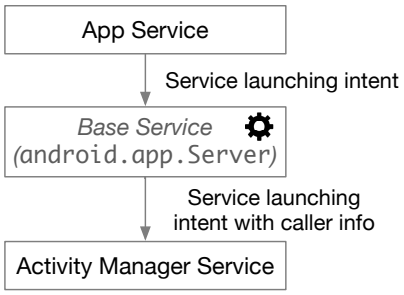

Figure 7. Attaching caller component information (using service as an example).

mitigate this potential problem, LMS validates caller component information it receives. First, the caller component must belong to the caller app, who can be identified by caller UID or package name. Second, the caller component must have been started already.

4.2.3 Nondisruptive Control

In addition to monitoring app lifecycle and building the ALG, our framework provides fine-grained, component-level lifecycle control. The idea is that lifecycle control policies can be stored as ALG node and edge properties. For example, if a service component is considered to be diehard we can simply set its `enabled` status to `false`, and let lifecycle hooks enforce it. We cannot return an error or throw out an exception within the hooks, because the hooks have no idea whether the caller app is able to properly handle the errors or exceptions. To avoid crashing the caller app unexpectedly, the hooks redirect ICCs to dummy components created by LMS. Those dummy components only execute minimal code and exit immediately.

4.2.4 Asynchronous Operations

To avoid hooks blocking the execution flow of apps, it is important to reduce the running time of hooks. Considering that hooks send ICC information to LMS and LMS takes time to process it, we let all hooks send asynchronous messages to LMS. The caller side (*i.e.,* hook points) does not have to wait for return values before proceeding. Moreover, whenever there is an update on ALG, LMS client should be aware of it. Instead of clients keeping polling ALG from LMS interfaces, LMS sends out a permission protected broadcast to notify clients. The protected broadcast can only be received by apps that have been granted the permission `android.permission.LIFECYCLE_UPDATES_ACCESS`, and user consent is required to grant an app this permission.

4.2.5 Exposed APIs

Table 2 lists a set of APIs that our framework provides to clients. To protect them from being abused, we enforce the permission `android.permission.LIFECYCLE_GRAPH_ACCESS`. This permission also requires user consent in order to be granted to an app. There are two categories of APIs according to how results get returned, *i.e.,* synchronous and asynchronous APIs. The principle is that reading operations are synchronous and writing operations are asynchronous. In this way, the ALG obtained by clients are consistent with the one in LMS, and updating ALG does not block the caller components. The code snippet in Figure 8 shows how easy it is to use the APIs to query different levels of graphs from ALG and detect cycles.

```
void detectCycles() {
  // detect cycles on component callback graph
  for (app : installedApps) {
    for (comp : getAppComponents(app)) {
      callbackGraph =
          lms.getCompCallbackGraph(app, comp);
      bfs(callbackGraph);
    }
  }
  // detect cycles on inner-app ICC graph
  for (app : installedApps) {
    compGraph = lms.getAppCompGraphs(app);
    bfs(compGraph);
  }
  // detect cycles on cross-app ICC graph
  bfs(lms.getLifecycleGraph);
}
```

Figure 8. Querying different levels of lifecycle graphs and detecting cycles. Certain variable types are omitted. `lms` is a reference pointing to the Lifecycle Manager Service.

5 Evaluations

We implement the proposed framework on Android Open Source Project (AOSP) 8.0.0_r4 codebase and install it on a Nexus 6P Android phone with 3GB memory. In this section, we evaluate the accuracy of our approach to building ALG and the performance of the fine-grained lifecycle control framework. To demonstrate the usability and the capabilities of the APIs, we showcase two example client apps. We also present our findings based on the analysis of 17,598 apps from Google Play and a third-party app market.

5.1 ALG Accuracy

We use analysis results of the apps described in §3 as ground truth to evaluate ALG accuracy. Since the ALG is built from runtime information collected by the hooks, there are no false-positive ICCs. Our hook placement strategy ensures accurate identification of ICC target components. The only factor that could result in inaccuracy is our caller component identification approach. In our experiments, the framework captures 149 unique ICCs and accurately identifies all of their caller components.

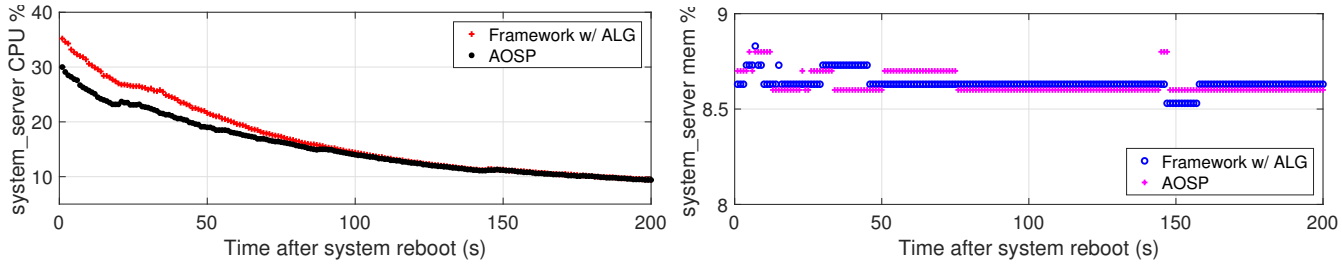

Figure 9. `system_server` CPU and memory usage after device reboot.

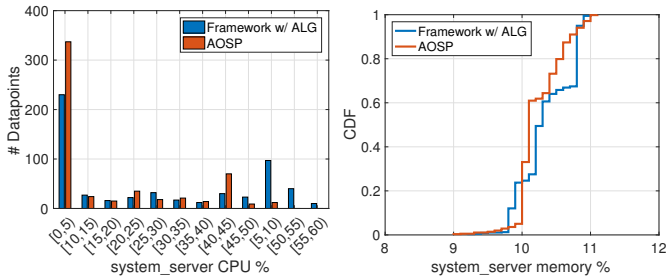

Figure 10. `system_server` CPU and memory usage while repeatedly launching apps.

5.2 Overhead

Lifecycle hooks and LMS are integrated into the Android framework, running in the `system_server` process. We compare CPU and memory usages with the original AOSP build, both have the same set of apps installed. At the very beginning of device booting, the lifecycle control framework adds approximately 5% CPU usage, as Figure 9 shows. This is reasonable as the initialization of the LMS takes additional CPU times. The difference becomes negligible after around 100 seconds, and the framework imposes only 0.15% additional memory usage. Most of the additional memory is used for storing the ALG at runtime, whose size is less than a few megabytes, depending on the number of installed apps. This is acceptable even on low-end devices with much less memory. We repeatedly launch an app for 560 times to measure CPU and memory usages during app launches. Results are shown in Figure 10. The framework incurs less than 20% peak CPU usage, due to a high number of ALG updates. Still, the memory usage difference is small.

We also evaluate app launch time and system boot time with and without the proposed framework. We follow the official recommendation on launch time performance measurements [14]. Results are presented in Figure 11(a). The median app launch time of AOSP is 363ms, while our framework increases that by 93ms. This small change can be barely noticed by users. We then reboot the device 100 times to measure system boot time. Results shown in Figure 11(b) suggest that boot time increase is also insignificant. The median increases from 28.345s to 30.932s. Figure 11(d) is the cumulative distribution of time consumed at hooking points. 99% hooks are executed within 2 millisecond.

5.3 API Usability

We implement two example client apps that leverage lifecycle control APIs to (1) detect and report cycles on ALG and thus detect diehard behavior and (2) restrict background ICCs that launch apps without user interactions.

Leveraging event contexts provided by ALG as edge properties, we implement an app that demonstrates the effectiveness of fine-grained lifecycle control, also using the interfaces provided by our framework. We enforce a policy that prevents an invisible app component (no matter it is in the background or foreground status) to launch inactive components in other apps. We fully charge the device, reboot it, and leave it for five hours without performing any operations on it. We measure battery level changes and battery discharge rate every 10 minutes with the Battery Historian tool [9]. Results in Figure 12 suggest that by the restriction of diehard behaviors is effective. Battery life can be extended significantly. Similar to CPU and memory usages after reboot, the discharge rate is higher with the framework at the very beginning due to additional initialization efforts.

5.4 Diehard Apps in the Wild

To the best of our knowledge, there is no prior study on diehard apps and their behaviors. To understand diehard behaviors in the wild, we analyze a large number of apps downloaded from Google Play and a third-party app market[1]. Due to the lack of an update-to-date Google Play dataset[2], we choose to download Google Play's top 500 best selling free apps from each of the 29 categories[3] by ourselves. 11,339 were successfully downloaded in early June of 2018. We

[1] `http://www.appchina.com/`. We choose this market for two reasons. First, Google Play is inaccessible for Chinese users. Second, it is one of the most popular 3rd-party markets according to Alexa Rank.
[2] The popular PlayDrone app dataset [36] used in other work is very outdated (updated in Nov. 2014).
[3] The category `ANDROID_WEAR` is excluded. Android wear apps are currently not in the scope of our study.

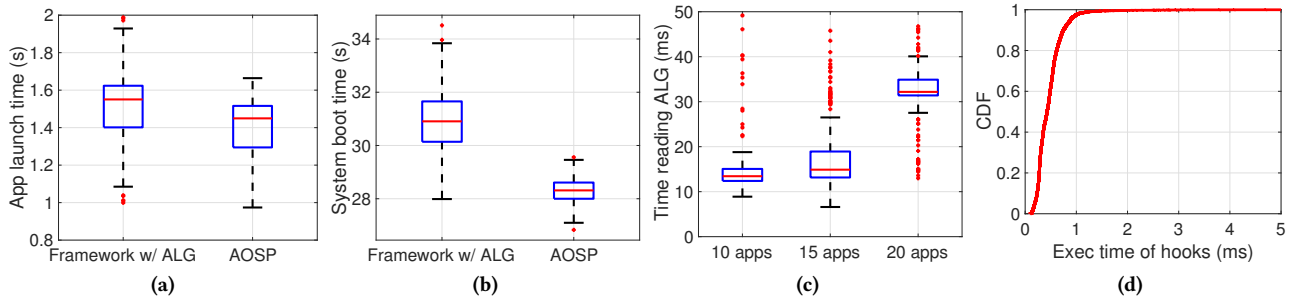

Figure 11. The comparison of app launch time and system boot between our framework and AOSP are shown in (a) and (b). (c) shows the difference in ALG reading time with different numbers of apps installed on the device. The cumulative distribution of hooks' execution time is presented in (d).

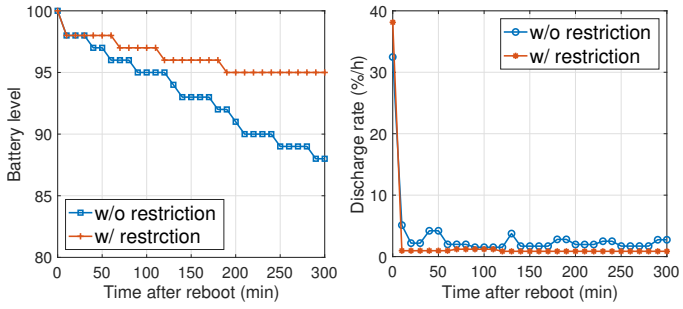

Figure 12. Battery life can be extended if diehard behaviors are restricted.

also collect 6,259 best selling apps from the third-party app market covering all its 15 categories, excluding duplicated ones that also appear in Google Play. We first group apps by their developers and then install apps in the same group altogether. We use `aapt` to identify app user interfaces (*i.e.,* Activity components), and use the command line tool `am` to launch them, mimicking user interactions. App analysis results show that diehard behaviors are common in both Google Play apps and apps from the third-party market. We also find diehard behaviors coming from widely used SDKs, although some of the host apps are not intentionally diehard.

Table 3 lists the percentages of apps that use each diehard technique. It is obvious that all diehard techniques except for account sync are more widely used by apps from the third-party market. Sticky service is the most prevalent among apps from both app markets. The percentages of Google Play apps having a floating view, native process, and explicit callbacks are significantly lower than that of apps from the third-party market. Figure 13 is the cumulative distribution of numbers of diehard techniques apps use. 38% Google Play apps have no diehard behaviors at all, while only 17% apps from the third-party market are non-diehard. These numbers clearly indicate that Google Play apps are less aggressive in keeping themselves long-running.

Table 3. Percentage of apps that use each diehard technique.

Technique	3rd-Party Market	Google Play
Foreground service	16.3%	13.1%
Native process	5.6%	1.0%
Floating activity	25.2%	9.3%
Sticky service	29.3%	25.1%
System events	19.4%	18.5%
Watchdog	7.3%	2.8%
Account sync	0.3%	0.3%
Inter-app wakeup	20.1%	17.5%
Scheduled tasks	0.9%	0.7%
Explicit callbacks	5.4%	1.8%

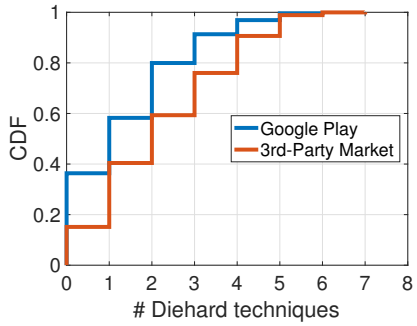

Figure 13. Numbers of diehard techniques used by apps from Google Play and the third-party market. Google Play apps tend to use less diehard techniques.

5.4.1 Purposes of Being Diehard

We investigate the purposes of apps for being diehard. By manually examining the ALG and reversing APKs, we classify diehard behavior purposes into the six categories. Results are summarized in Table 4.

Sensor monitoring. Certain apps want to constantly monitor system events and user activities. Since Android N, most of the system broadcasts can only be received by dynamically registered broadcast receivers. For example, apps are

Table 4. Purposes of being diehard.

Purpose	# Apps	
	3rd-Party Market	Google Play
Sensor monitoring	40	61
Ads/promotions	59	51
Push notification	1,192	124
Keyguard	84	75
Hot patching	48	39
Downloading/uploading	280	57

no longer able to receive screen lock events with a static broadcast receiver. In order to monitor system events and take actions accordingly, apps have to keep alive so that their dynamic receivers are active. There are also apps constantly sensing the ambient environment, *e.g.,* lighting. Other examples include fitness apps that track user activities. They need to keep long-running in the background, otherwise, their would produce inaccurate results.

Displaying ads/promotions. A common business model for app developers is to make profits from ads displayed inside their apps. They usually keep an ad service running in the background to retrieve ads and show them to users. Some apps show promotion notifications from while running in the background. Such apps would like to be diehard so that they can maximize profits.

Push notifications. Google recommends Firebase Cloud Message (FCM) for sending push notifications to devices from the server side. However, FCM is not available in certain regions (*e.g.,* China blocks Google services). The lack of OS-level message push service leaves developers no choice but to use third-party message push SDKs, or to implement their own message push services. One major metric for evaluating the quality of message push services is the delivery rate. To make sure push notifications can be delivered timely and successfully, push services have to leverage diehard techniques to keep themselves long-running in the background. We observe apps coming with multiple push services, all of which attempt to stay long-running in the background, creating several diehard services.

Keyguard. Non-system apps are not allowed to replace the lock screen. But apps can create UI components that looks like a screen lock to users. In order to provide self-implemented keyguard functionalities while the device is locked, apps need a long-running service, which has to be diehard so that it can provide the required functionalities.

Hot patching. Android allows apps to load and execute dynamically at runtime. For example, they can download plugins in the format of `dex` files and load them by user demand. This feature is also used by some apps to realize hot patching. Users do not need to reinstall the app anymore. Instead, hot patching services can replace the out-dated code by updating the corresponding `dex` file.

Downloading/uploading. Certain apps download data from or upload local updates to their servers periodically. They use a diehard service to prevent the downloading/uploading process from being accidentally terminated. In fact, the recommended approach to downloading and uploading data is to use `AsyncTask`.

5.4.2 Third-Party Libraries

We find that apps may not intend to be diehard. Their diehard behaviors could come from third-party libraries. A summary of third-party libraries having diehard behaviors is listed in 5. To our surprise, we observe dedicated libraries that implement state-of-the-art diehard techniques and their primary goal is to keep apps long-running. For example, a library with package name `com.daemon.keepalive` is found in several high-rating apps with millions of installs such as Smart Cooler [20], RAM Master [19], and SPARK [21]. An industry-leading Android anti-virus service provider, Qihoo 360, offers a malware scanning library with a diehard service `com.qihoo.magic.service.KeepLiveService`. This service appears in Qihoo family apps as well as non-Qihoo apps such as Super Antivirus Cleaner [23], which has a rating of 4.7 and more than 10 million installs. We also find an open-source daemon library that offers out-of-the-box diehard components [12].

Tencent Xinge is one of the most popular message push SDKs. It actively queries installed apps on the device and looks for apps that also have the same SDK integrated, *i.e.,* apps that also listen to the broadcast `com.tencent.android.tpush.action.SDK`. If another app is found to have Xinge SDK but is not currently alive, it tries to launch that app in various ways, one of which is presented in Figure 14. The built-in command line tool, `am`, is called to start the target Xinge services in other apps.

```
for (Map.Entry localEntry : localMap.entrySet()) {
  try
  {                    Map key is app package name and value is service name.
    String str = "am startservice -n " + (String)localEntry.getKey()
    Process localProcess = Runtime.getRuntime().exec(str);
    int i = localProcess.waitFor();
    if (i != 0)
    {
      str = "am startservice --user 0 -n " + (String)localEntry.getK
      localProcess = Runtime.getRuntime().exec(str);
      i = localProcess.waitFor();
    }
```

Figure 14. Tencent message push SDK has diehard behavior that wakes up all its services using shell command `am` that can bypass background execution limitation. Code decompiled from version 4.0.3 jar file using JD-GUI.

Third-party libraries, in particular, those exhibiting diehard behaviors, are potentially leaking user privacy. SDKs such as Tencent Xinge, JPush, Xiaomi Push asks for sensitive permissions, including reading phone state, accessing WiFi/network state, accessing fine location, and accessing coarse location.

Table 5. Third-party libraries coming with diehard behaviors, their purposes, techniques they use, and whether they request sensitive permissions.

SDK	Purpose	Techniques	Sensitive Permissions
Tencent Xinge	Message push & Notification	Native watchdog, foreground service, sticky service, system events, inter-app wakeup	Yes
JPush	Message push & Notification	Foreground service, floating activity, inter-app wakeup	Yes
iGenxin	Notification	Scheduled tasks, sticky service, foreground service, system events, watchdog	Yes
Baidu Share	Cross-app data sharing	Native watchdog, sticky service, foreground service, system events, inter-app wakeup	Yes
Xiaomi Push	Message push & Notification	Sticky service, foreground service, system events, inter-app wakeup	Yes
Eguan	Monitoring	Scheduled tasks, sticky service, foreground service, system events	No
EMChat	Notification	Scheduled tasks, sticky service, foreground service, system events, inter-app wakeup	Yes
Jiubang	Notification	Scheduled tasks, sticky service, foreground service, system events, watchdog	Yes

5.4.3 A Real-World ALG

Figure 15 presents a real-world ALG visualization from 10 apps. It shows the interactions between apps and app components, as well as lifecycle events. We find that Baidu apps perform cross-app wakeup intensively. Tencent apps tend to utilize framework services to be diehard. The ES File Explorer app (package name `com.estrong.android.app`) and Tencent Input app (package name `com.tencent.qqim`) have watchdog services.

6 Discussion

While this work presents a fine-grained lifecycle control framework and makes the first step toward understanding diehard behaviors, there are limitations we plan to address in the future. First, we collected apps from Google Play and only one third-party market, which might lead to biases in results. We plan to do a larger scale study across multiple app markets. Second, due to the nature of runtime analysis, the framework cannot capture potential lifecycle events that are not triggered by the user, therefore the completeness of the ALG is not guaranteed. Third, a user study could be helpful for us to design better APIs for empowering app developers. As the wearable platforms become increasingly popular, we also plan to implement the framework on the Android Wear platform and investigate diehard behaviors of wear apps.

Legitimacy of diehard behaviors. We acknowledge that apps may have legitimate reasons for being diehard. For instance, a fitness app has to monitor user activities and locations constantly. However, we argue that apps should more gracefully achieve long-running and clearly indicate their background activities using Android recommended approaches, instead of abusing app lifecycle or gaming the system. Our proposed framework provides foundations for developing robust diehard behavior detection and restriction mechanisms. Device vendors and developers can leverage the ALG and event contexts our framework provides to realize a crowd-sourced tool that can identify the legitimacy of diehard behaviors with a large dataset.

Background-running apps on iOS. Unlike Android, background processing in iOS is highly regulated. Long-running tasks require specific permissions to run in the background without being shut down, and only specific app types are allowed to do so [6]. Additionally, all iOS apps submitted to the App Store are manually reviewed to ensure that they do not violate Apple's guidelines. Developers are required to present a compelling reason for background activities. We argue that Android cannot simply adopt the iOS approach to restricting background executions. Android is meant to be a customizable platform and the whole ecosystem is open. Developers are believed to be responsible and follow the guidelines, which is unfortunately not true.

Circumventions. We place hooks into the system based on our understanding of current system implementation. In the future, new Android APIs might be introduced that could be abused to realize diehard behaviors and circumvent being captured by ALG. We argue that our framework is extensible and ALG can also be extended in order to adapt to future Android frameworks. As long as we build a runtime ALG, we can always rely on it and upgrade the detection algorithms.

Lessons learned. Benign diehard apps call for system-level support of long-running mechanisms that are transparent and controllable to users. Third-party libraries should be better inspected before being integrated, and library providers are supposed to provide configuration options to app developers. Nevertheless, it is challenging to prevent abuses of legitimate functionalities, because oftentimes this is a cat-and-mouse game and API designers are unaware of the

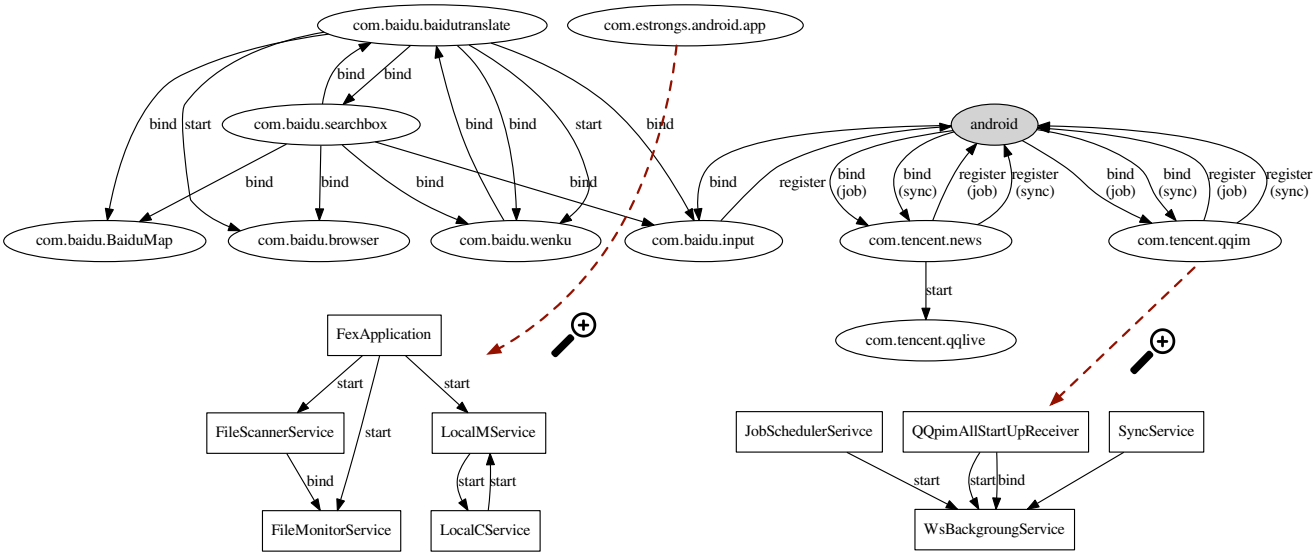

Figure 15. The topmost level (*i.e.,* cross-app ICC graph) of a real ALG visualized by Graphviz. `com.estrongs.android.app` and `com.tencent.qqim` are further inspected with one of their intra-app ICC graphs.

potential side effects. Android API designers can leverage our framework to fix loopholes and better manage app lifecycle.

7 Related Work

App lifecycle control mechanisms and diehard behaviors have not been studied before. Previous efforts in identifying and addressing battery drain problems [29, 31, 32, 35] mainly focus on bugs instead of intentional app behaviors and lack the insights into app lifecycle as a whole.

Chen *et. al.* present a study on low-level background activities of apps by looking into CPU idle and busy time [29]. They conducted a large-scale measurement study performing in-depth analysis of background app activities, quantified the amount of battery drain, and developed a metric called background to foreground correlation to measure the usefulness of background activities. Our work is different than theirs in two aspects. (1) Our focus is an important root cause of energy drain, *i.e.,* behaviors that make apps long-running and evade being killed. (2) We not only look at individual apps but also interactions among apps. There is also work that helps understand where and how energy drain happens in smartphones [28]. The authors developed a hybrid utilization-based and finite state machine based model that accurately estimates energy breakdown among activities and phone components. Pathak *et. al.* perform a characterization study of no-sleep energy bugs in smartphone apps and proposes a compile-time solution to automatically detect no-sleep bugs [33]. This work focuses on a different problem. While they look at bugs that may cause unusual energy drain, we study intentional behaviors of apps that keep them long-running in the background. To mitigate no-sleep bugs, Vekris *et. al.* [35] implement a tool that verifies the absence of this kind of energy bugs with regard to a set of WakeLock specific policies using a precise, inter-procedural data flow analysis framework to enforce them. Tamer [32] is an OS mechanism that interposes on events and signals that cause task wake-ups and allows for their detailed monitoring, filtering, and rate-limiting. It helps reduce battery drain in scenarios involving popular apps with background tasks.

8 Conclusion

In this paper, we present a fine-grained app lifecycle control framework that leverages app lifecycle graph (ALG) to accurately describe app lifecycles. We overcome challenges such as caller component identification, nondisruptive app control. App study results suggest that diehard behaviors are common in apps from both Google Play and a third-party market. Evaluations show that our framework is capable of efficiently capturing app lifecycle events, imposing negligible performance overhead. The proposed framework provides easy-to-use APIs on which new features can be developed. It empowers device vendors and app developers to leverage the ALG and event contexts to realize accurate detection of diehard behaviors and component-level, fine-grained control on app lifecycle.

Acknowledgments

We would like to thank Haining Chen and Shengtuo Hu for their input during the early stage of this work. We would also like to thank the anonymous reviewers, as well as Roxana Geambasu, our shepherd, for their valuable feedback. This work is partially funded by NSF under the grants CCF-1628991, CNS-1629763 and by ONR under the grant N00014-18-1-2020.

References

[1] Advanced task manager. https://play.google.com/store/apps/details?id=mobi.infolife.taskmanager.

[2] Android authority forums. https://www.androidauthority.com/community/.

[3] Android banking malware whitelists itself to stay connected with attackers. https://www.symantec.com/connect/blogs/android-banking-malware-whitelists-itself-stay-connected-attackers.

[4] Android forums. https://androidforums.com/.

[5] Android forums at androidcentral. https://forums.androidcentral.com/.

[6] App programming guide for ios — background execution. https://goo.gl/jryM9q.

[7] Application fundamentals. https://developer.android.com/guide/components/fundamentals.html.

[8] Background optimizations. https://developer.android.com/topic/performance/background-optimization.html.

[9] Battery historian. https://github.com/google/battery-historian.

[10] Dashboards. https://developer.android.com/about/dashboards/.

[11] Es task manager (task killer). https://play.google.com/store/apps/details?id=com.estrongs.android.taskmanager.

[12] Hello daemon. https://github.com/xingda920813/HelloDaemon.

[13] How to turn off smartphone apps that track you in the background. http://www.ibtimes.com/how-turn-smartphone-apps-track-you-background-1657868.

[14] Launch-time performance. https://developer.android.com/topic/performance/launch-time.html.

[15] Optimizing for doze and app standby. https://developer.android.com/training/monitoring-device-state/doze-standby.html.

[16] Privacy issues: Data abuse on certain mobile apps uncovered. https://www.sciencedaily.com/releases/2012/07/120705133714.htm.

[17] Processes and application life cycle. https://developer.android.com/guide/topics/processes/process-lifecycle.html.

[18] Processes and threads. https://developer.android.com/guide/components/processes-and-threads.html.

[19] Ram master – memory optimizer. https://play.google.com/store/apps/details?id=com.speedbooster.optimizer.

[20] Smart cooler. https://play.google.com/store/apps/details?id=com.cooler.smartcooler.

[21] Spark – live random chat. https://play.google.com/store/apps/details?id=com.video.chat.spark.

[22] Stack overflow android questions. https://stackoverflow.com/questions/tagged/android.

[23] Super antivirus cleaner & booster. https://play.google.com/store/apps/details?id=com.oneapp.max.

[24] These 5 apps are killing your battery. https://www.androidpit.com/battery-draining-apps.

[25] Who lives and who dies? process priorities on android. https://medium.com/google-developers/who-lives-and-who-dies-process-priorities-on-android-cb151f39044f.

[26] Xda developers. https://forum.xda-developers.com/android/software.

[27] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. of the ACM PLDI* (2014).

[28] Chen, X., Ding, N., Jindal, A., Hu, Y. C., Gupta, M., and Vannithamby, R. Smartphone energy drain in the wild: Analysis and implications. In *Proc. of the ACM SIGMETRICS* (2015).

[29] Chen, X., Jindal, A., Ding, N., Hu, Y. C., Gupta, M., and Vannithamby, R. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *Proc. of the ACM MobiCom* (2015).

[30] Fratantonio, Y., Qian, C., Chung, S. P., and Lee, W. Cloak and dagger: From two permissions to complete control of the ui feedback loop. In *Proc. of the IEEE S&P* (2017).

[31] Ma, X., Huang, P., Jin, X., Wang, P., Park, S., Shen, D., Zhou, Y., Saul, L. K., and Voelker, G. M. Edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *Proc. of the USENIX NSDI* (2013).

[32] Martins, M., Cappos, J., and Fonseca, R. Selectively taming background android apps to improve battery lifetime. In *Proc. of the USENIX ATC* (2015).

[33] Pathak, A., Jindal, A., Hu, Y. C., and Midkiff, S. P. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proc. of the ACM MobiSys* (2012).

[34] Shao, Y., Ott, J., Jia, Y. J., Qian, Z., and Mao, Z. M. The misuse of android unix domain sockets and security implications. In *Proc. of the ACM CCS* (2016).

[35] Vekris, P., Jhala, R., Lerner, S., and Agarwal, Y. Towards verifying android apps for the absence of no-sleep energy bugs. In *Proc. of the USENIX HotPower* (2012).

[36] Viennot, N., Garcia, E., and Nieh, J. A measurement study of google play. In *Proc. of the ACM SIGMETRICS* (2014).

[37] Wei, F., Li, Y., Roy, S., Ou, X., and Zhou, W. Deep ground truth analysis of current android malware. In *In Proc. of DIMVA* (2017).