

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

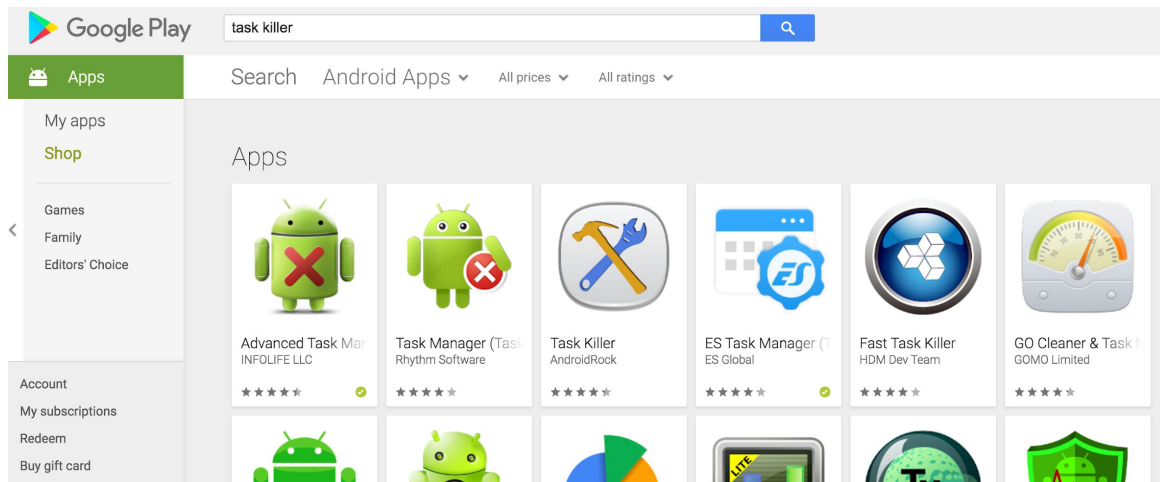
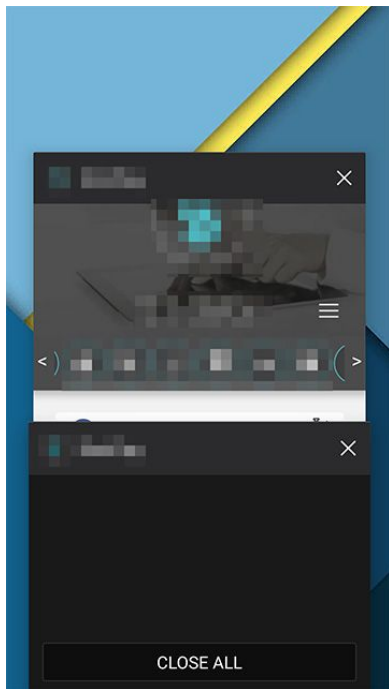
---

Yuru Shao<sup>1</sup>, Ruowen Wang<sup>2</sup>, Xun Chen<sup>2</sup>, Ahmed M. Azab<sup>2</sup>, Z. Morley Mao<sup>1</sup>  
<sup>1</sup>University of Michigan      <sup>2</sup>Samsung Research America

Presenter: Xiao Zhu (University of Michigan)

# Diehard apps

- Some apps are **hard to kill**



**“Close all” cannot kill all running apps**

# Diehard apps

- Even if they get killed they manage to **auto restart**

## [How To Disable Auto-Starting Apps On Android | PCsteps.com](https://www.pcsteps.com/16385-how-to-disable-auto-starting-apps-on-android/)

<https://www.pcsteps.com/16385-how-to-disable-auto-starting-apps-on-android/> ▼

★★★★★ Rating: 5 - 3 votes

Mar 28, 2018 - **How To Disable Auto-Starting Apps On Android** .... When we run an app, it will automatically "kill" the one we were using before that, instead of ...

Why should we prevent ... · Stop auto-starting apps on ...

## [How to stop apps from running in the background on Android ...](https://www.androidpit.com/how-to-stop-apps-running-in-the-background-on-android)

<https://www.androidpit.com/how-to-stop-apps-running-in-the-background-on-android> ▼

Aug 1, 2018 - To **stop** an app manually via the processes list, head to Settings > Developer Options > Processes (or Running Services) and click the **Stop** button. Voila! To Force **Stop** or Uninstall an app manually via the Applications list, head to Settings > Applications > Application manager and select the app you want to modify.

## [How to Stop Android Apps From Starting By Themselves](https://www.maketecheasier.com/stop-android-apps-from-starting-by-themselves/)

<https://www.maketecheasier.com/stop-android-apps-from-starting-by-themselves/> ▼

Aug 9, 2018 - Here we'll take you through the best methods of **stopping** your **Android apps opening automatically**. Related: [How to Stop Pop-ups on Android](#) ...

## [How To Disable Auto Start Apps in Android Smartphones & Tablet](https://www.theandroidportal.com/How-To)

[https://www.theandroidportal.com/How To](https://www.theandroidportal.com/How-To) ▼

Oct 4, 2017 - You might have noticed when you boot your **Android**, some apps getting started **automatically**. Some apps like Google play services, Amazon ...

# Diehard apps implications

- Battery drain
  - Performance degradation
- 
- Reasons for being diehard
    - Bad engineering
    - Intended functionality: could be legit or illegit



# Coarse-grained app lifecycle control

## Why can apps be hard to kill?

- An app consists of a set of components
  - Activity: a component that represents visible UI that users can see and interact with
  - Service: a component that performs a longer-running operation while the app is not interacting with the user
- Services can be background for foreground
  - System considers foreground services to be more important to users
- “Close all” tries to stop all visible components, i.e., activities

# Coarse-grained app lifecycle control

## Why can apps be hard to kill?

- Before being killed, a component gets notified
  - `onStop()` / `onDestroy()` callbacks, giving the component a chance **to die gracefully**
  - Or **to revive stealthily**

Diehard techniques abuse

1. Foreground service
2. Floating view
3. Native process

```
1 // full class name: com.android.Launcher.Se
2 public class Se extends Service {
3     ...
4     // onDestroy() callback is always called by
5     // the system when a service gets killed
6     public void onDestroy() {
7         super.onDestroy();
8         ...
9         // Restart itself (the 2nd argument is the
10        // target service that will be started).
11        Intent i = new Intent(this.context, Se.class);
12        i.setFlags(268435456);
13        i.setAction("com.dai.action");
14        i.setAction("com.tdz.action");
15        startService(i);
16    }
17 }
```

HummingBad malware

# Coarse-grained app lifecycle control

## Why can apps auto restart?

- Inter-component communications (ICC) are common
  - Enable easy interactions among apps
  - Open doors for abuses
- Auto-run techniques abuse
  - Sticky service
  - System events
  - Watchdog
  - Sync service and job service
  - Cross-app wakeup

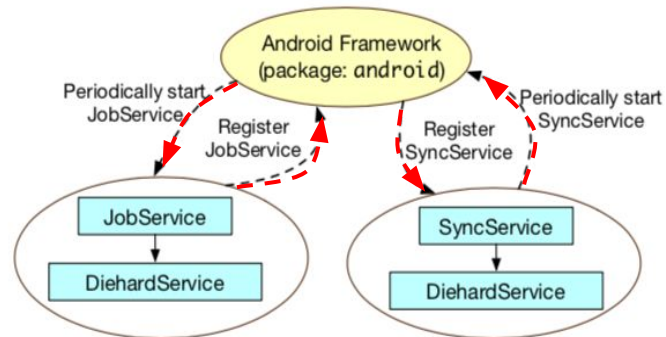
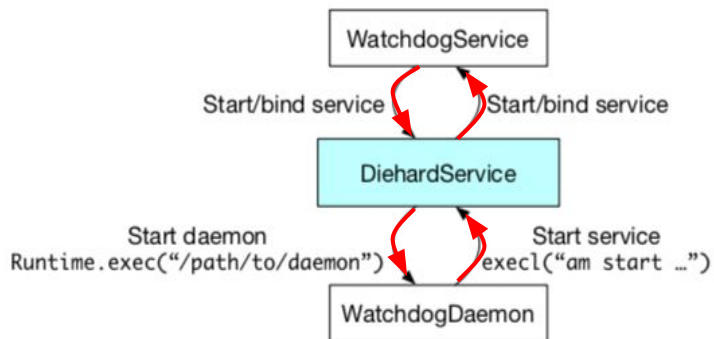
# Coarse-grained app lifecycle control

- Background Exec Limit were introduced in Android 8.0
  
- But “Background Exec Limit” has limitations
  - Too coarse-grained: per app, not per component
  - Apps can invisibly run in foreground
  - Inter-app wakeup is common among apps integrating the same 3rd-party libs



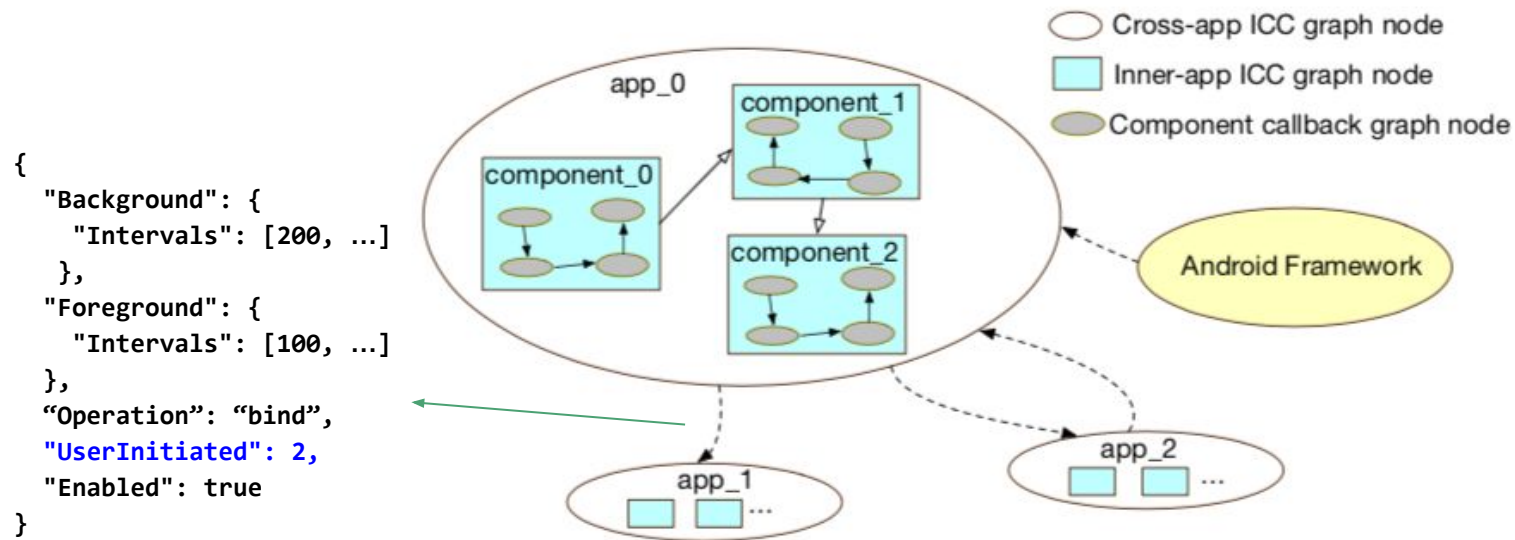
# Key insights

- Diehard behaviors create **interdependence** between:
  - component callbacks
  - app components
  - different apps
- Such interdependence can be captured as cycles on a graph



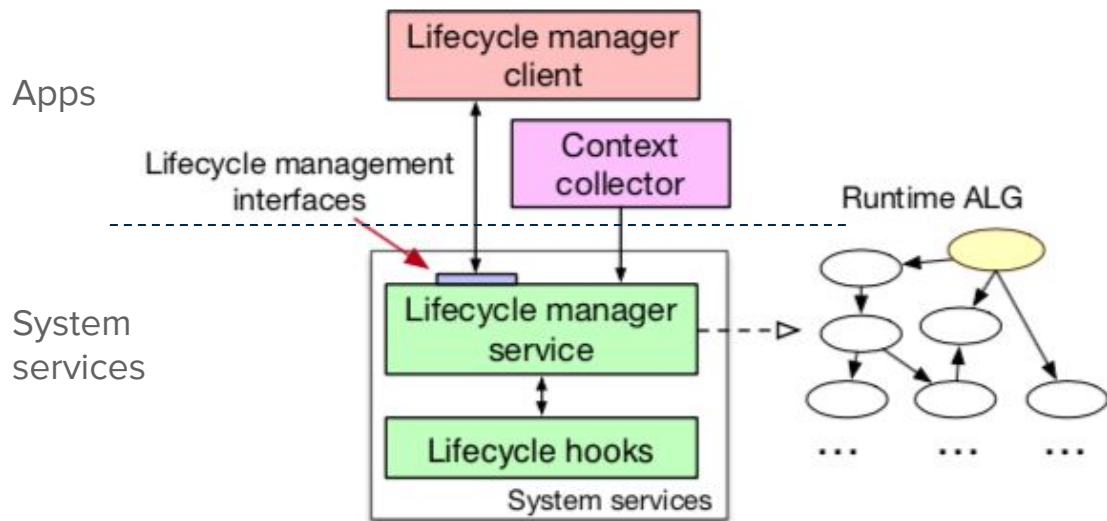
# App lifecycle graph (ALG)

- Has multiple levels that track inner- and inter- app interactions
- Annotated with attributes that provide event contexts



# A component-level lifecycle control framework

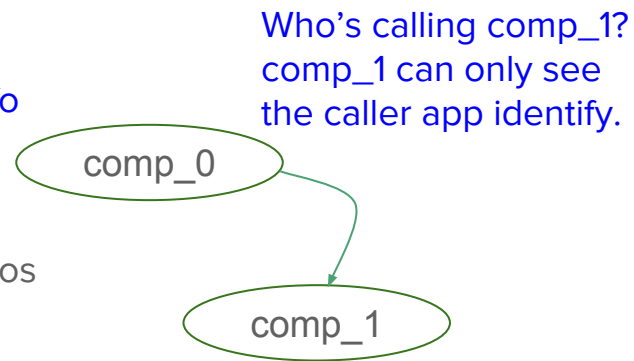
- Maintains a global ALG in memory to enable efficient graph operations
- Installs async hooks to monitor all ICC events and collects ICC info
- Provides query & control capabilities as APIs



# A component-level lifecycle control framework

## Requirements and challenges

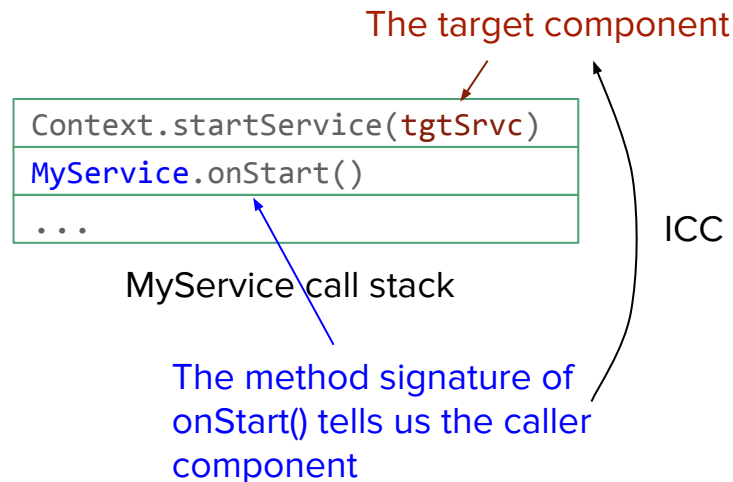
- Accurately identifying ICC caller component
  - No existing mechanisms to provide component-level caller info
  - Limited caller app info: only app UID/PID/package name
- Nonblocking, hooks don't block ongoing operations
  - There's no single best hook placement strategy for all scenarios
- Nondisruptive, avoid causing app crashes
  - Hard to gracefully shut down apps/components



# Identifying caller component

- Target component is called by an app, no caller component info provided
- Naïve approach: inspecting call stack when starting an ICC

```
class MyService extends Service {  
    ...  
    public void onStart() {  
        // start a target service  
        this.startService(tgtSrvc);  
    }  
    ...  
}
```

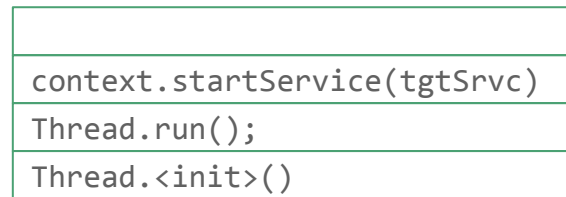


# Identifying caller component

- Target component is called by an app, no caller component info provided
- ~~Naïve approach: inspecting call stack when starting an ICC~~
  - Call stack is per thread
  - Doesn't work if the caller starts a new thread in which the target is called

```
class MyService extends Service {  
    ...  
    public void onStart() {  
        // start a target service in a new thread  
        new Thread() {  
            Public void run() {  
                MyService.this.startService(tgtSrvc);  
            }  
        }.start();  
    }  
    ...  
}
```

Caller component info  
unavailable on the new  
thread's call stack

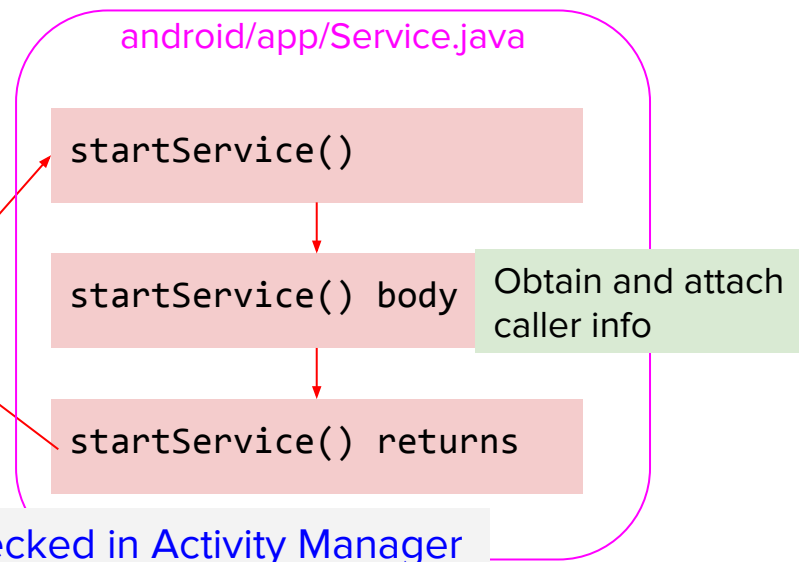


New thread call stack

# Identifying caller component

- No caller component info provided by the system
- Proposed approach: attaching caller info in the base Service class

```
class MyService extends Service {  
    ...  
    public void onStart() {  
        // start a target service in a new thread  
        new Thread() {  
            Public void run() {  
                MyService.this.startService(tgtSrvc);  
            }  
        }.start();  
    }  
    ...  
}
```



Attached caller info will be checked in Activity Manager Service later, in case an app wants to bypass it.

# Using event contexts

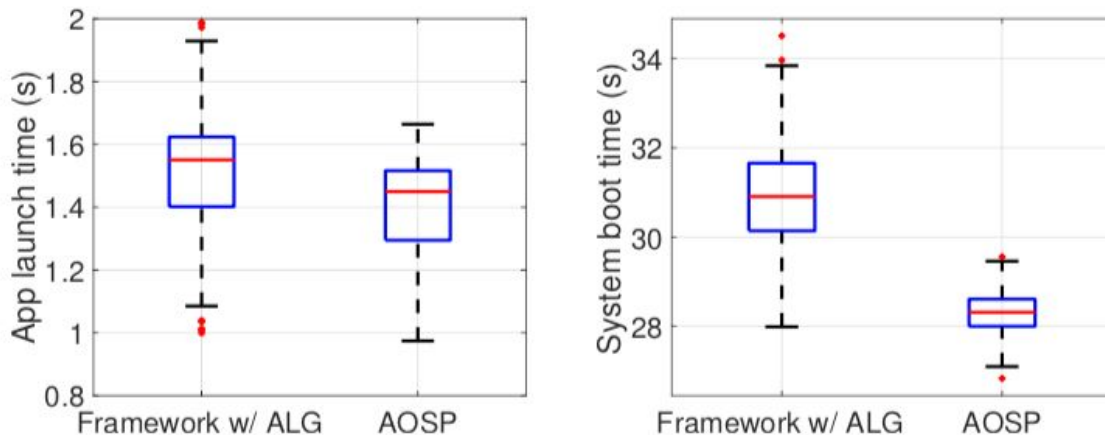
- ICC event contexts are helpful for distinguishing legit and illegit diehard behaviors
- Example policy: If a service is in **foreground** and only started by **non-user-initiated** components, then it's an illegit diehard component

```
for (String app : listOfApps) {  
    AppCompGraph appCompGraph = LMS.getAppCompGraph(app);  
    for (Node comp : appCompGraph.Nodes) {  
        if (comp.getProperty("foreground") == true) {  
            // check all incoming edges' "userInitiated" property  
            // if all > 0, this component is a diehard service  
        }  
    }  
}
```



# Results: overhead

- Evaluated on a Nexus 6P (3GB RAM) running Android 8.0
- The framework incurs low overhead on app launch time and system boot time

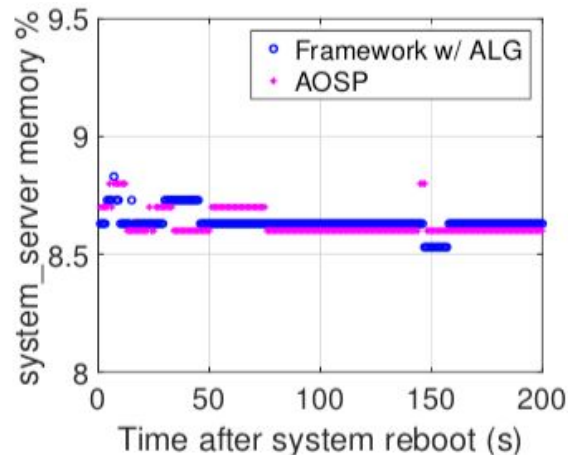
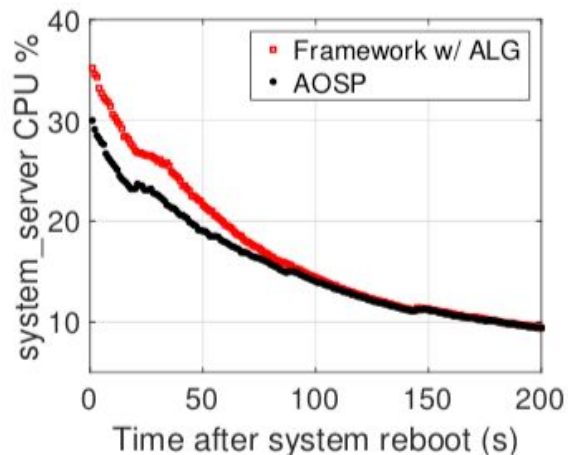


Measured with Android activity manager service

- < 0.1s app launching delay
- ~2.5s system boot delay

# Results: overhead

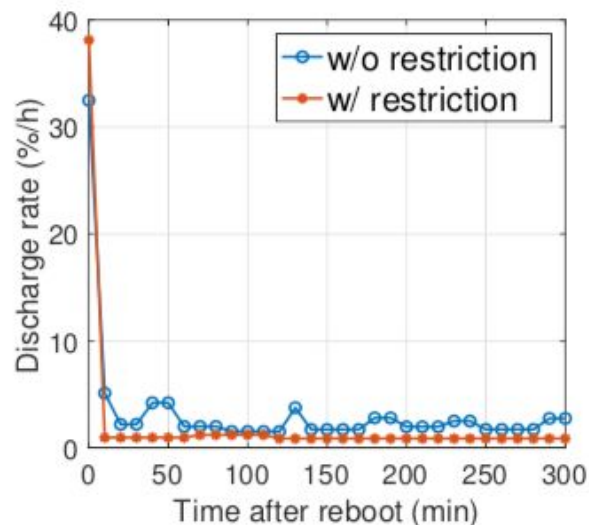
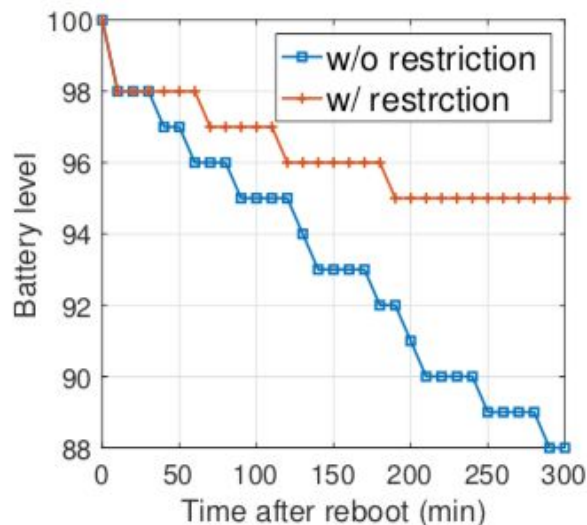
- Evaluated on a Nexus 6P (3GB RAM) running Android 8.0
- The framework incurs negligible overhead on CPU and memory usage



- ~5% additional CPU usage during initialization
- ~4.5MB (0.15%) additional memory usage

# Results: a restriction rule

- Disable background auto-start services by **cutting off background edges**
- 7 Baidu family apps and 3 Tencent apps installed
- Left phone idle after reboot

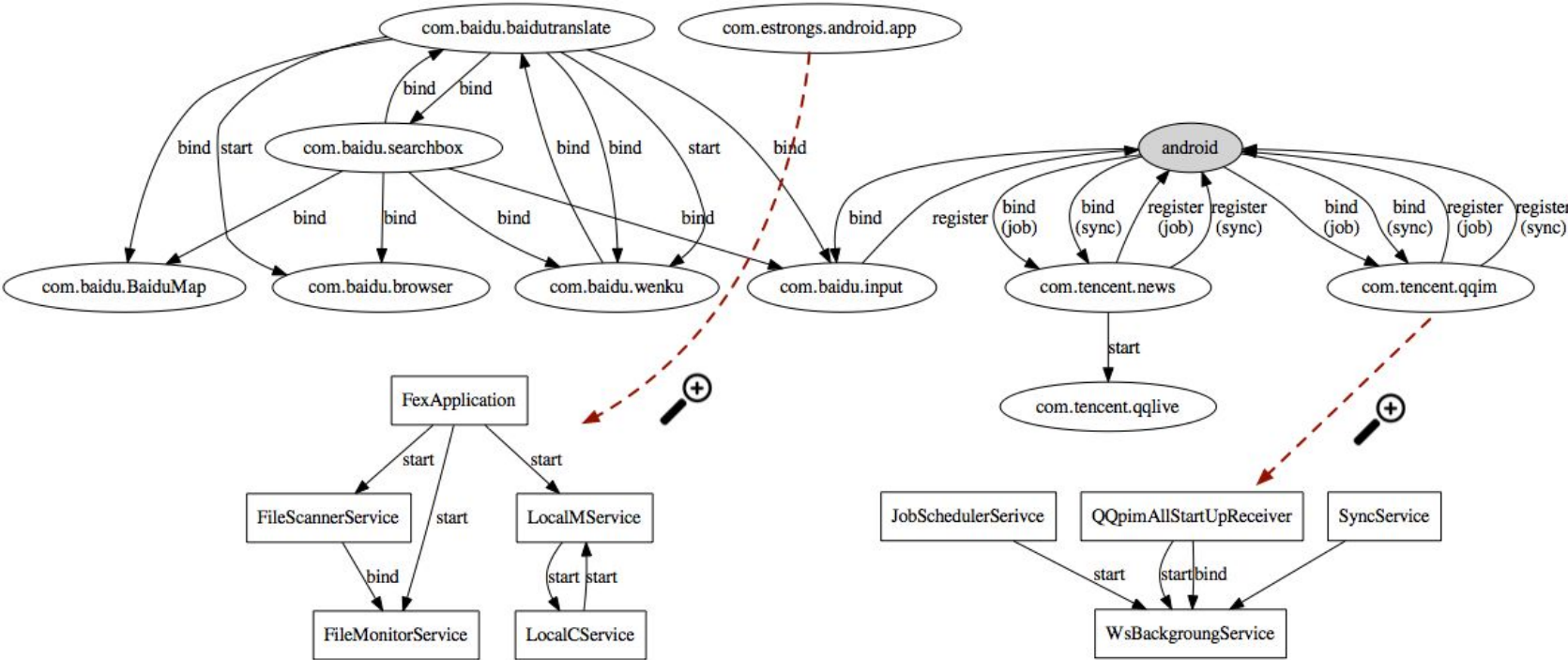


# Summary

- Diehard apps abusing system features is a known but previously unstudied problem
  - Apps from 3rd-party markets tend to be more aggressive
- Propose ALG for complete, precise app lifecycle description
  - Diehard behavior analysis and detection problems are transformed into graph problems
- Leveraging ALG, a lightweight framework is presented to provide fine-grained lifecycle enforcement
- Future work includes using user feedback to build better policies for restricting diehard behaviors

Thank you!

# Results: ALG example



# Results: diehard apps

- 17,598 apps from Google Play and a 3rd-party market
- 13.1% Google Play apps and 16.3% 3rd-party market apps have foreground services
- Apps from the 3rd-party market are more aggressive

