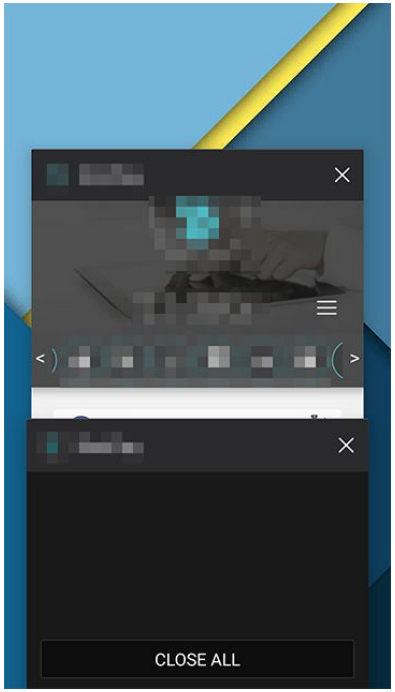
A Lightweight Framework for Fine-Grained Lifecycle Control of Android Applications

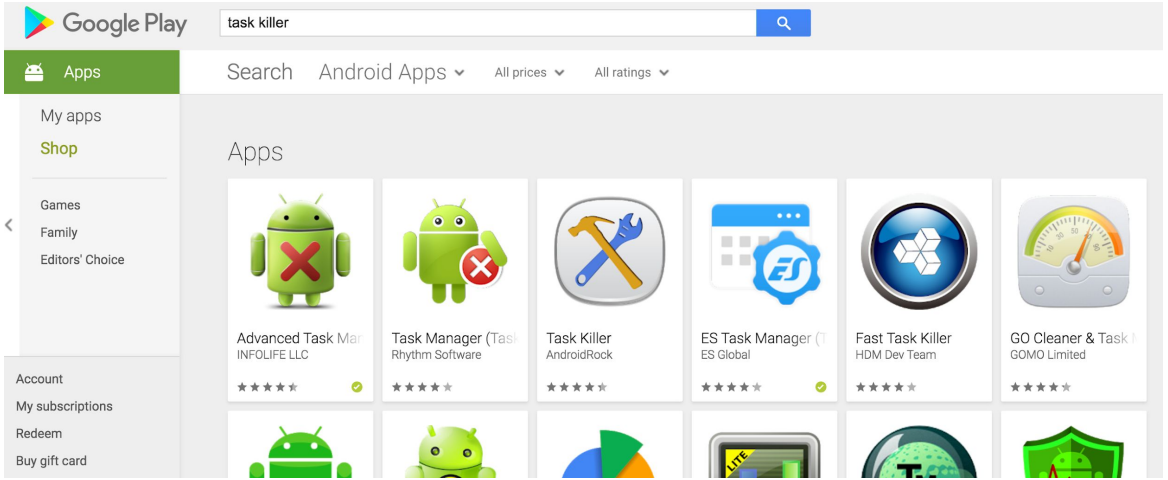
Yuru Shao[1], Ruowen Wang[2], Xun Chen[2], Ahmed M. Azab[2], Z. Morley Mao[1]
[1]University of Michigan [2]Samsung Research America

Presenter: Xiao Zhu (University of Michigan)

Diehard apps

- Some apps are hard to kill

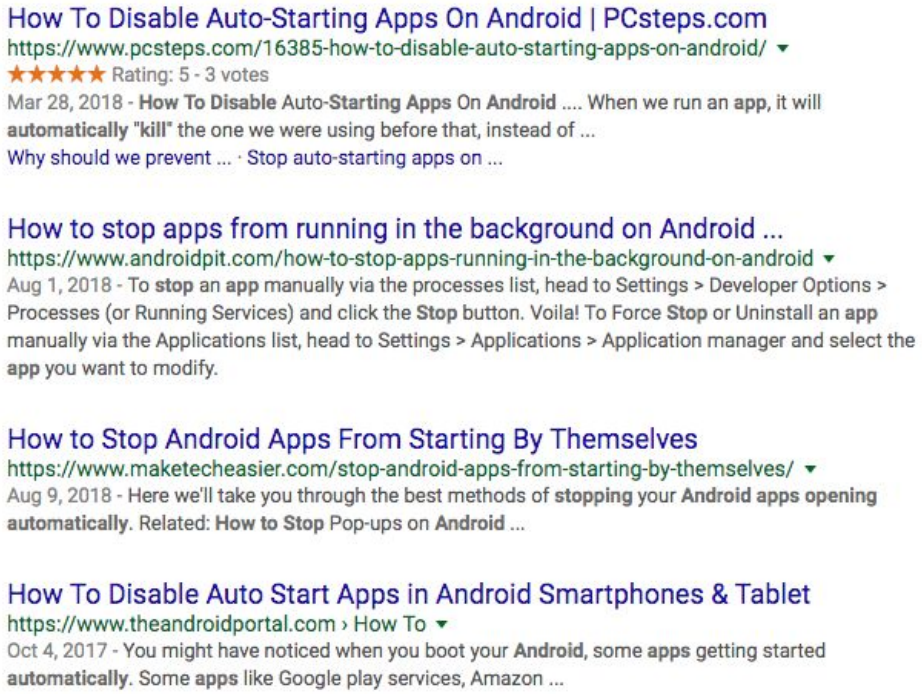




"Close all" cannot kill all running apps

Diehard apps

- Even if they get killed they manage to auto restart



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

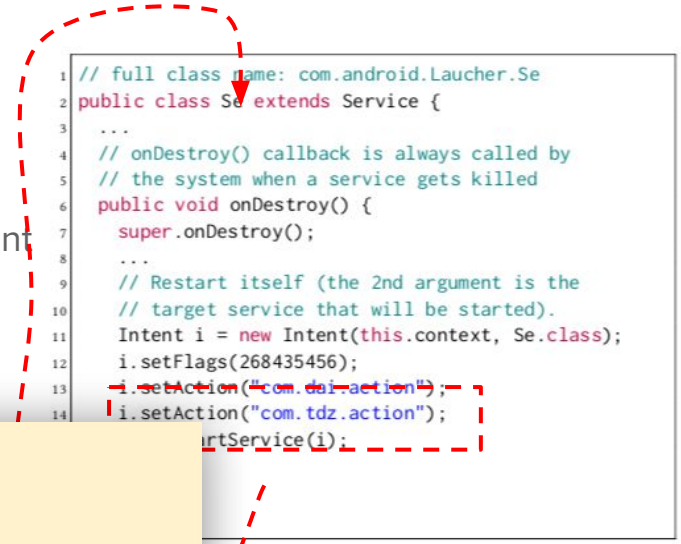
Why can apps be hard to kill?

- 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()/onDestory()` callbacks, giving the component a chance ***to die gracefully***
  - Or ***to revive stealthily***



HummingBad malware

Diehard techniques abuse

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

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

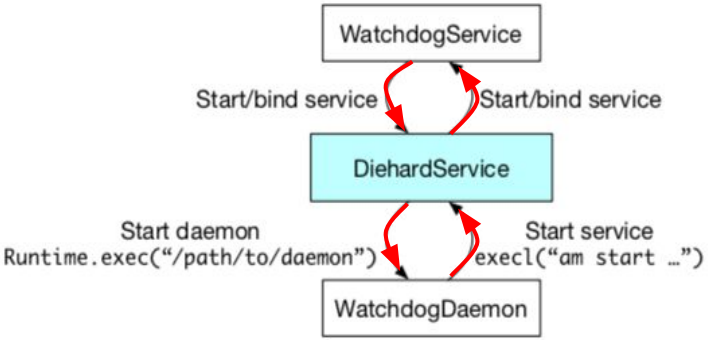
Coarse-grained app lifecycle control

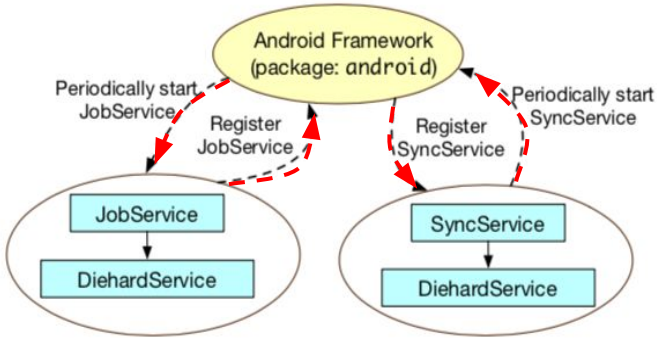
- Background Exec Limit were introduced in Android 8.0

- 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

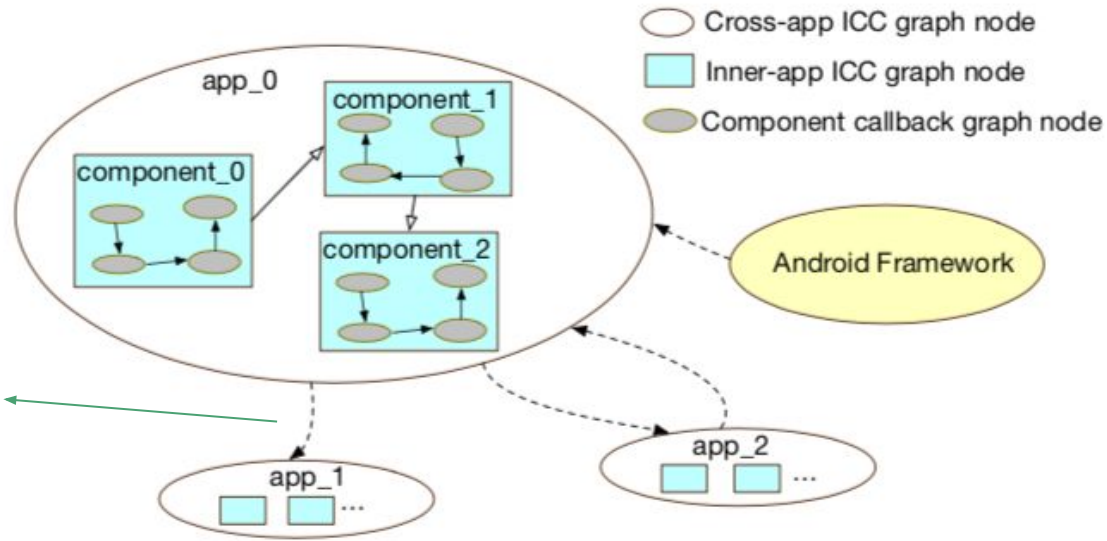
- Diehard behaviors create interdependence between:
  - component callbacks
  - app components
  - different apps
- Such interdependence can be captured as cycles on a graph





App lifecycle graph (ALG)

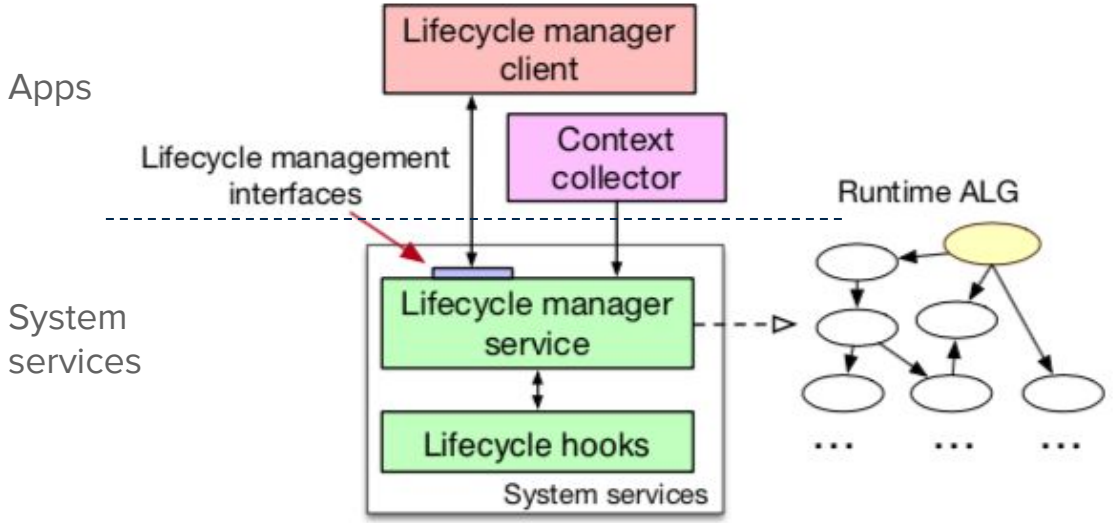
- Has multiple levels that track inner- and inter- app interactions
- Annotated with attributes that provide event contexts




```
{
  "Background": {
    "Intervals": [200, ...]
   },
  "Foreground": {
    "Intervals": [100, ...]
  },
  “Operation”: “bind”,
  "UserInitiated": 2,
  "Enabled": true
}
```

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

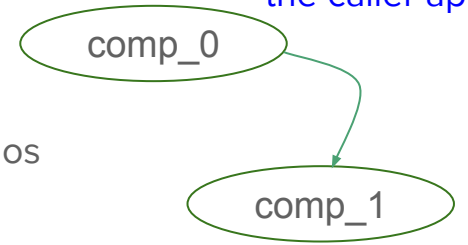


A component-level lifecycle control framework

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

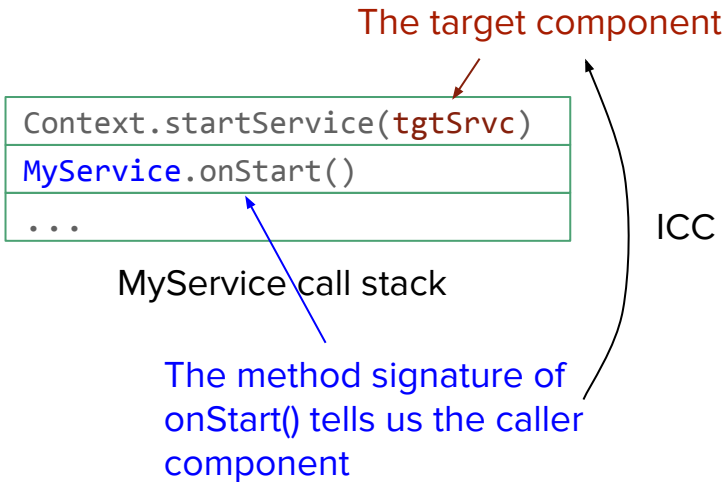
Who’s calling comp_1? comp_1 can only see the caller app identify.



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

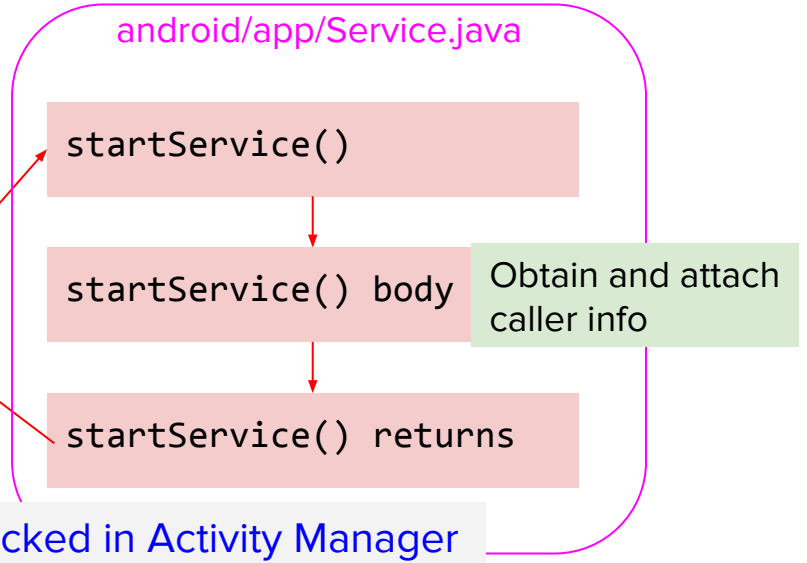
context.startService(tgtSrvc)
Thread.run();
Thread.<init>()

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.

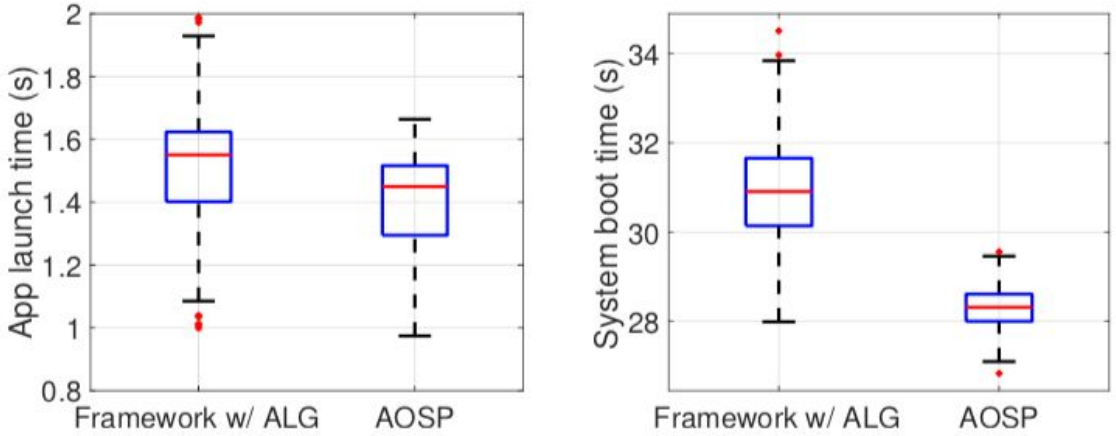
Using event contexts

- 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

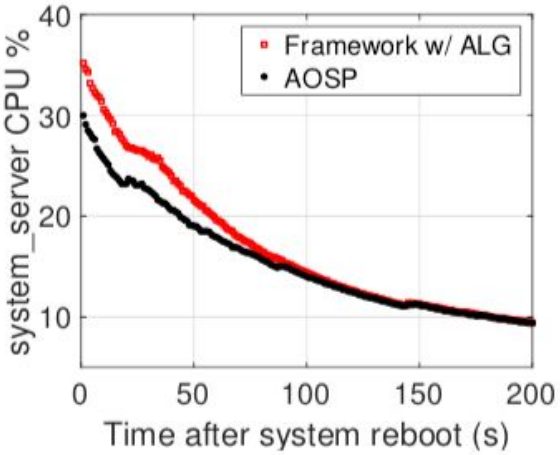


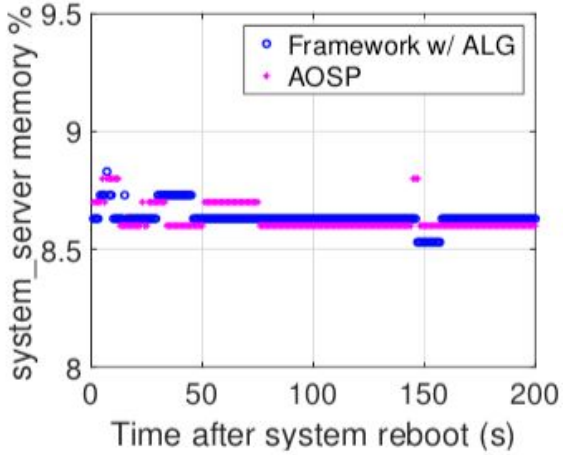
Measured with Android activity manager service

- < 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

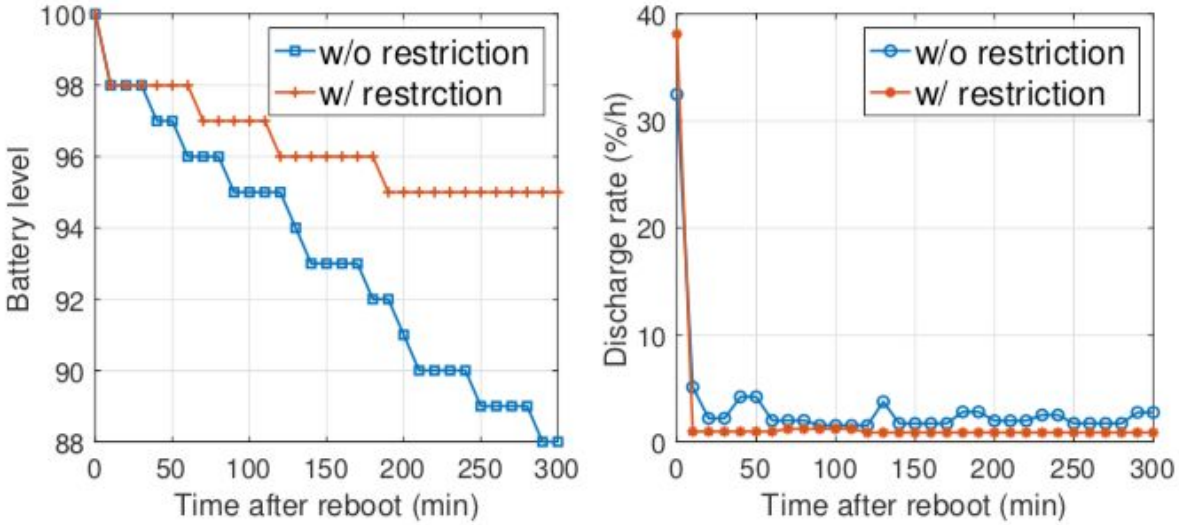




- ~5% additional CPU usage during initialization
- ~4.5MB (0.15%) additional memory usage

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

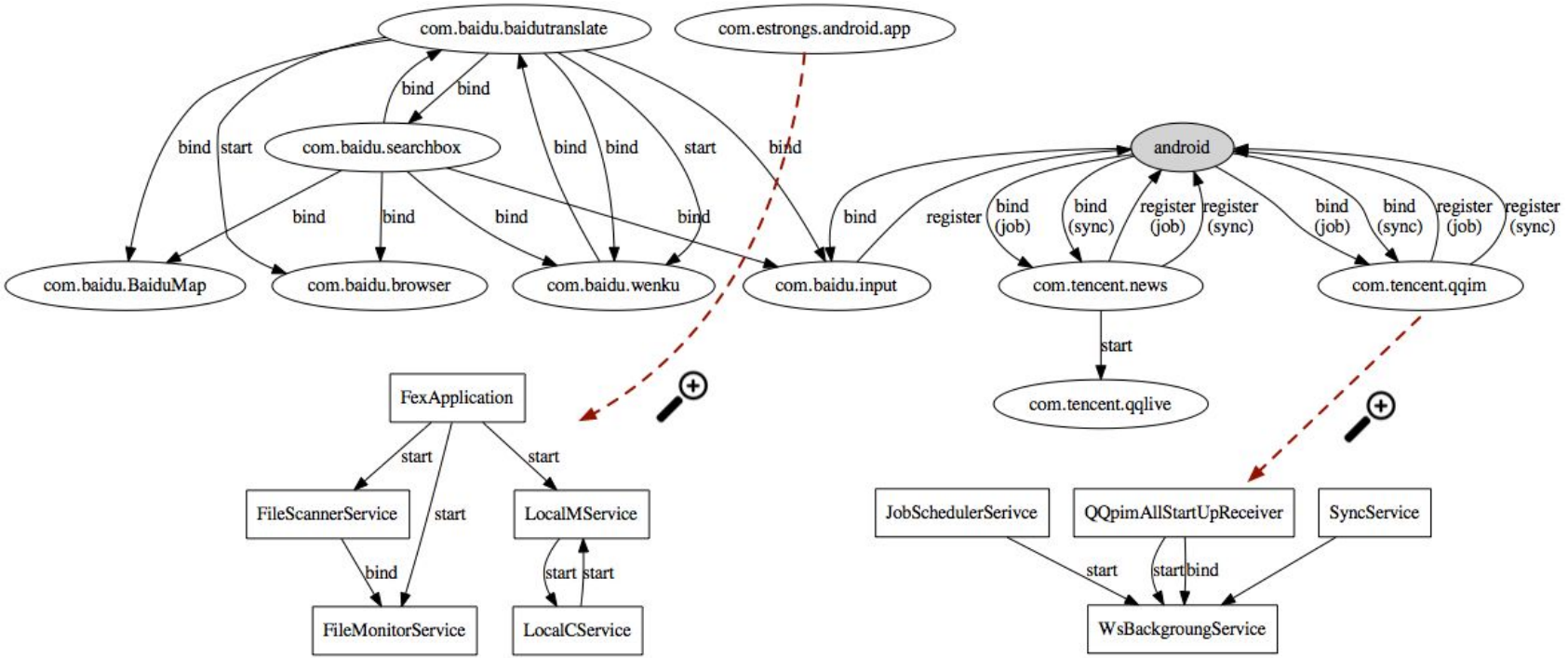


Summary

- 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