Demystifying Diehard Android Apps

Hao Zhou
The Hong Kong Polytechnic
University
Hong Kong, China
cshaoz@comp.polyu.edu.hk

Haoyu Wang
Beijing University of Posts and
Telecommunications
Beijing, China
haoyuwang@bupt.edu.cn

Yajin Zhou
Zhejiang University
Hangzhou, China
yajin_zhou@zju.edu.cn

Xiapu Luo*
The Hong Kong Polytechnic
University
Hong Kong, China
csxluo@comp.polyu.edu.hk

Yutian Tang
ShanghaiTech University
Shanghai, China
csytang@ieee.org

Lei Xue
The Hong Kong Polytechnic
University
Hong Kong, China
cslxue@comp.polyu.edu.hk

Ting Wang
Pennsylvania State University
Pennsylvania, USA
inbox.ting@gmail.com

ABSTRACT

Smartphone vendors are using multiple methods to kill processes of Android apps to reduce the battery consumption. This motivates developers to find ways to extend the liveness time of their apps, hence the name diehard apps in this paper. Although there are blogs and articles illustrating methods to achieve this purpose, there is no systematic research about them. What's more important, little is known about the prevalence of diehard apps in the wild.

In this paper, we take a first step to systematically investigate diehard apps by answering the following research questions. First, why and how can they circumvent the resource-saving mechanisms of Android? Second, how prevalent are they in the wild? In particular, we conduct a semi-automated analysis to illustrate insights why existing methods to kill app processes could be evaded, and then systematically present 12 diehard methods. After that, we develop a system named DiehardDetector to detect diehard apps in a large scale. The experimental result of applying DiehardDetector to more than 80k Android apps downloaded from Google Play showed that around 21% of apps adopt various diehard methods. Moreover, our system can achieve high precision and recall.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

---

*The corresponding author.

1 INTRODUCTION

Battery life is one of major concerns of users. A recent survey showed that "41% of US smartphone users say a longer battery life is the design feature they want most." [9] Accordingly, smartphone vendors deploy *resource-saving mechanisms* to optimize the battery usage to kill background app processes [3].

However, from another perspective, app developers intend to increase the active time (liveness) of their apps. By doing so, they can maintain an inactive experience with users, such as providing timely services (e.g., sending notifications), making profit (e.g., pushing advertisements), etc. Indeed, recent years have witnessed many apps (diehard apps) employ various approaches to bypass resource-saving mechanisms. These approaches, which can prolong diehard apps' lifetime, are called *diehard methods*.

The diehard methods can be roughly divided into 2 categories according to their purposes: (1) keeping apps alive; (2) waking up apps. Shao et al. [26] manually studied 23 diehard apps and found a few diehard methods, most of which aim at waking up apps. However, the community still lacks a deep understanding of diehard methods and diehard apps.

Our work In this paper, we take a first step to systematically investigate diehard apps by answering 3 research questions: (1) Why could existing methods to kill app processes be circumvented? (2) How do direhard apps keep themselves alive or be waken up? (3) Do diehard behaviors widely exist in apps in the wild? Answers to these questions can facilitate the development of effective resource-saving mechanisms to benefit users (with a battery-efficient system) and keep the app ecosystem healthy by discovering diehard apps.

Hao Zhou, Haoyu Wang, Yajin Zhou, Xiapu Luo, Yutian Tang, Lei Xue, and Ting Wang

To this end, we first perform a systematic study on existing methods to kill app processes and analyze why these methods could be evaded (in §3). It is not easy to locate these methods and figure out corresponding evasion approaches because Android system has more than 10 millions lines of code and it is still evolving, let alone the lack of detailed documentation for those methods. We solve this challenge by taking a semi-automated approach that uses static code analysis to extract mostly relevant source code and then manually inspect it to understand the internal mechanisms.

Second, leveraging the observations obtained in the previous step, we systematically propose 12 diehard methods. Among them, 7 methods can keep the app alive (in §4), and 6 of them have *not* been reported before. The remaining 5 approaches can wake up diehard apps (in §5), and 1 of them is *not* mentioned in the previous work [26]. To evaluate the effectiveness of these methods, we develop 2 diehard apps (a normal app and an instant app). The extensive experiments on popular Android versions, ranging from Android 5.1 to Android 10.0, show that all of our methods can prolong apps' liveness time.

Third, to detect diehard apps in the wild, we develop DiehardDetector, a new detection tool based on the 12 diehard methods. We carefully design DiehardDetector to make it achieve low false positive and false negative rates and be able to inspect a large number of Android apps. The evaluation results of DiehardDetector show that it is accurate (i.e., the precision and the recall of the results on detecting diehard apps are 100% and 94.70%, respectively) and efficient (i.e., 45 seconds for analyzing an app on average).

Fourth, we apply DiehardDetector to analyze more than 80k apps for identifying diehard behaviors. The experimental results show that around 21% of the apps from Google Play have diehard behaviors. Moreover, 2 proposed diehard methods have not been found in the apps under investigation. It suggests that we find 2 previously unknown methods to extend the liveness time of apps.

To engage the community, we release DiehardDetector and the apps involved in the evaluation at https://github.com/moonZHH/DiehardDetector.

In summary, this paper makes the following main contributions:

- To our best knowledge, we perform the first systematic investigation on diehard apps. In particular, we are the first to reveal why the resource-saving mechanisms can be circumvented and propose 12 diehard methods to keep apps alive and wake up apps.
- We develop DiehardDetector, a new tool that can identify diehard behaviors in Android apps accurately and efficiently.
- We conduct extensive experiments to evaluate DiehardDetector and examine the prevalence of diehard methods used by apps. The experimental results on over 80k apps from Google Play show that DiehardDetector can accurately identify the diehard behaviors of apps in a scalable manner. Moreover, around 21% of apps published in Google Play have diehard behaviors.

2 BACKGROUND

Android apps can be roughly classified into 2 categories: (1) system apps, which are usually pre-installed by smartphone vendors. Since such apps are protected by Android system from being killed, they rarely use diehard methods. (2) normal apps and instant apps, which are created by developers and downloaded by the smartphone users.
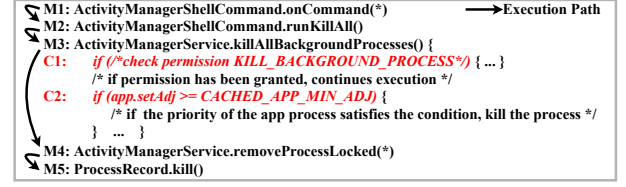


```
M1: ActivityManagerShellCommand.onCommand(*)              ──▶ Execution Path
M2: ActivityManagerShellCommand.runKillAll()
M3: ActivityManagerService.killAllBackgroundProcesses() {
C1:     if (/*check permission KILL_BACKGROUND_PROCESS*/) { ... }
        /* if permission has been granted, continues execution */
C2:     if (app.setAdj >= CACHED_APP_MIN_ADJ) {
            /* if  the priority of the app process satisfies the condition, kill the process */
        } ... }
M4: ActivityManagerService.removeProcessLocked(*)
M5: ProcessRecord.kill()
```

Figure 1: Finding the process-killing method, KAP.

Such apps may adopt diehard methods to prolong the liveness time for many purposes (e.g., pushing advertisements to make profits).

An app may have multiple processes. By default, an app's components (i.e., activities, services, broadcast receivers, and content providers) are running in the app's main process. However, each component can specify its `process` property in the manifest file to make it run in a separate process other than the main process. If all of an app's running processes are killed, the app is stopped.

Android framework provides 2 data structures, `ProcessRecord` and `TaskRecord`, to manage app processes. We leverage them to find out the methods for killing the app process or stopping the app. Note that the methods for stopping the app are special cases of methods for killing the app process because they are designed to kill all running processes associated with an app.

ProcessRecord: It is used to characterize each running app process by `ActivityManagerService` (AMS), one of the core services of Android system. Specifically, the `curAdj` (or `setAdj`) field and the `curSchedGroup` (or `setSchedGroup`) field of a `ProcessRecord` object hold an app process's priority values. There are several ways to notify AMS to create a `ProcessRecord` for an app, e.g., launching its activity, starting its service, sending a broadcast that can be responded by its receiver, accessing its content provider, etc.

TaskRecord: It is used to characterize an activity Task [11], which is a collection of activities that the user interacts with when performing a certain job. Meanwhile, in normal cases, a `TaskRecord` object of a recently accessed app corresponds to an item in the Recent-Task list [8]. Accordingly, a `TaskRecord` object links an activity task of a recently accessed app with a Recent-Task item.

3 METHODS FOR KILLING APP PROCESSES

To find the methods for killing app processes, we conduct a semi-automated investigation on the framework and the system applications (e.g., low memory killer, a.k.a LMK) of Android 9.0 in §3.1. We learn from this investigation that the priority of an app process affects whether it will be killed or not, and describe how Android system determines an app process's priority in §3.2. In §3.3, we disclose the internals of these process-killing methods, and point out the insights why they can be evaded, which are then exploited to design diehard methods (in §4 and §5).

3.1 Finding Process-killing Methods

Diehard apps employ diehard methods to prevent their processes from being killed. Thus we look for all process-killing methods to understand why they could be circumvented. There are 2 basic approaches to terminate an app process: (1) invoking the `kill` API defined in the `ProcessRecord` class, because Android framework uses `ProcessRecord` to represent a running app process; (2) using the `kill` function exported by `libc.so`, because native applications

or libraries can invoke this function that calls syscall `__NR_kill`. We adopt a semi-automated way to find other process-killing methods that will eventually call either of the above basic approaches. Starting with the basic approaches, we use static analysis to extract most relevant source code and then manually inspect them, as detailed in the following paragraphs. It is worth noting that this semi-automated way allows us to inspect the whole Android system and can be used to find new or changed process-killing methods in the future versions of Android.

Procedure: To find the process-killing methods in Android framework, we first construct the call graph of the framework by using PScout [13]. Then, we locate the `ProcessRecord.kill` API in the call graph and perform backward reachability analysis to get the execution paths that reach this API. Each execution path refers to a specific process-killing method. Moreover, to reveal the conditions for terminating the app process, we perform control-flow analysis on each execution path to locate the conditional statements that affect whether the `kill` API will be executed or not. Subsequently, we manually analyze the extracted conditional statements to disclose how to evade the process-killing methods.

As an example, Figure 1 shows the execution path and the necessary conditional statements of the process-killing method for killing all background processes (KAP, in §3.3). We find the execution path M1→M2→M3→M4→M5 that accesses the `ProcessRecord.kill` API (i.e., M5). Then, we locate the conditions (i.e., C1 and C2) that need to be satisfied to reach M5. After inspecting these conditional statements, C2 in particular, we learn that, the priority of the app process (i.e., the value stored in the `setAdj` of the corresponding `ProcessRecord` instance) determines whether this app process will be killed or not. If an app process's priority does not satisfy C2, this app process circumvents KAP.

Since the native applications in Android system can call the `kill` function exported by `libc.so`, to discover the process-killing methods in them, we build the call graph for each native application by using SVF [27], and locate the `kill` function. Once found, we reversely traverse the call graph from the `kill` function to get the execution paths that reach this function. Then, we manually analyze the source code of the methods in each execution path to investigate how to evade the process-killing methods.

Result: We find 6 process-killing methods (in §3.3) and 4 of them check the priority of the app process to decide whether it will be killed or not. Meanwhile, we notice that, after killing an app process, Android framework will call AMS's `updateOomAdjLocked` method to adjust the priority of current running app processes. To understand how AMS calculates the priority of an app process, we perform control-flow analysis on AMS's `computeOomAdjLocked` method, which will be called by `updateOomAdjLocked`. Specifically, since the local variables, `adj` and `schedGroup`, are used to update `curAdj` and `curSchedGroup` of the `ProcessRecord` instance, respectively, we extract the conditional statements that decide whether their values will be changed. Then, we manually analyze these statements to disclose why a process has a specific priority (in §3.2).

3.2 App Process Priority

Android determines the priority of an app process using the out-of-memory (OOM) adjustment and the thread scheduling group. In the following, we first introduce how the OOM adjustment of an app process is set, which is relevant to the diehard methods proposed in §4.2, and then describe 2 thread scheduling groups that are relevant to the diehard methods proposed in §4.1. Table 1 summarizes the difference cases of app processes.

OOM Adjustment. An app process with high priority usually has a small OOM adjustment value and is less likely to be killed.

∗ `FOREGROUND_APP_ADJ`: The process of the foreground app (e.g., **C01** and **C02**) is associated with this OOM adjustment.

∗ `VISIBLE_APP_ADJ`: The app process (e.g., **C05**) that hosts visible activities is linked with this OOM adjustment.

∗ `PERCEPTIBLE_APP_ADJ`: The app process (e.g., **C08**) that hosts perceptible components (e.g., foreground services connected with visible notifications) is mapped to this OOM adjustment.

∗ `SERVICE_ADJ`: The priority of the app process (e.g., **C12**) that hosts the running services is depicted by this OOM adjustment.

∗ `CACHED_APP_MIN_ADJ`: The app process (e.g., **C13**) that hosts invisible activities is assigned with this OOM adjustment.

Remark: If an app process satisfies multiple previous descriptions, its OOM adjustment is the one that has the smallest value. For example, the OOM adjustment of the app process, hosting a visible activity and running a service, is `VISIBLE_APP_ADJ`.

Thread Scheduling Group. AMS uses it to manage the liveness of a group of app processes. Normally, processes with the same priority tend to be organized in one thread scheduling group.

∗ `SCHED_GROUP_TOP_APP`: This group contains the processes of foreground apps (e.g., **C01**), which have relatively high priority.

∗ `SCHED_GROUP_BACKGROUND`: This group includes the processes of background apps (e.g., **C12**), which have relatively low priority.

3.3 Exploring Process-Killing Methods

In this section, we elaborate on the internals of the 6 process-killing methods discovered in §3.1, especially the conditions determining whether or not an app process will be killed, and point out the insights why the methods could be circumvented.

(1) Removing Recent-Task Item (RRT).

Usage: Users can swipe away an item presented in the Recent-Task list to stop a recently accessed app.

Internal: RRT consists of 2 major steps. First, AMS removes the `TaskRecord` object associated with the removed Recent-Task item from `ActivityStackSupervisor`'s `mRecentTasks` field, which is an array recording the `TaskRecord` object of each recently accessed app. Second, AMS uses `ActivityStackSupervisor` (ASS), which is the manager of activity stacks, to obtain the `ProcessRecord` objects associated with the app to be stopped. Once a `ProcessRecord` object is found, ASS checks whether the process hosts activities. If not, the process is the candidate to be killed. Otherwise, ASS checks whether the activities hosted by the process belong to the same activity task. If so, such process is the candidate as well. Before calling `ProcessRecord`'s kill method to terminate the candidate, ASS checks whether the process's thread scheduling group is `SCHED_GROUP_BACKGROUND`. If so, the process will be killed.

Remark: RRT cannot stop the app that does not have a Recent-Task item. Moreover, it kills neither the app process that has multiple activity tasks nor the app process whose thread scheduling group is not `SCHED_GROUP_BACKGROUND`.

(2) Force-stopping App (FSA).

Table 1: A summary of different cases of app processes.

OOM Adjustment	Value	Case	Scheduling Group
FOREGROUND_APP_ADJ	0	**C01**: The process has a foreground activity, and the device is awake.	SCHED_GROUP_TOP_APP (3)
		C02: The process has a foreground activity, but the device is sleeping.	SCHED_GROUP_BACKGROUND (0)
		C03: The process has a service, to which another process with such an OOM adjustment is bound.	SCHED_GROUP_DEFAULT (2)
		C04: The process has a provider, which is acquired by another process with such an OOM adjustment.	SCHED_GROUP_DEFAULT (2)
VISIBLE_APP_ADJ	100 ≀ 200	**C05**: The process has a visible activity.	SCHED_GROUP_DEFAULT (2)
		C06: The process has a service, to which another process with such an OOM adjustment is bound.	SCHED_GROUP_DEFAULT (2)
		C07: The process has a provider, which is acquired by another process with such an OOM adjustment.	SCHED_GROUP_DEFAULT (2)
PERCEPTIBLE_APP_ADJ	200	**C08**: The process has a foreground service.	SCHED_GROUP_DEFAULT (2)
		C09: The process has an overlay window.	SCHED_GROUP_DEFAULT (2)
		C10: The process has a service, to which another process with such an OOM adjustment is bound.	SCHED_GROUP_DEFAULT (2)
		C11: The process has a provider, which is acquired by another process with such an OOM adjustment.	SCHED_GROUP_DEFAULT (2)
SERVICE_ADJ	500	**C12**: The process has a service, which was started within the last 30 minutes.	SCHED_GROUP_BACKGROUND (0)
CACHED_APP_MIN_ADJ	900	**C13**: The process has the stopped activity.	SCHED_GROUP_BACKGROUD (0)

(a) Normal App.   **(b) Instant App.**
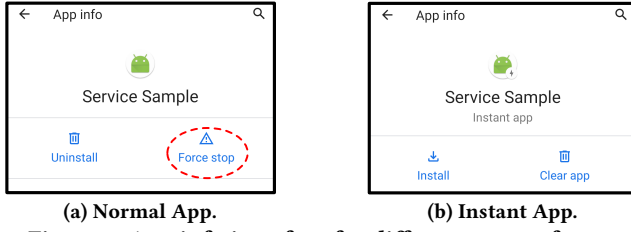
Figure 2: App info interface for different types of apps.

Usage: Users can navigate to the App info interface and click the *Force stop* button to stop an app. A debugger can use the shell command, *am force-stop*, to achieve the same purpose.

Internal: FSA takes 3 steps to stop an app. First, AMS notifies the instance of `PackageManagerService` (PMS) to make the target app ignore the received broadcasts that were sent without carrying the flag, `FLAG_INCLUDE_STOPPED_PACKAGES`. Second, AMS finds the `ProcessRecord` objects associated with the app and calls `kill` method to terminate the processes. Third, AMS sends a broadcast to instruct the instances of `AlarmManagerService` and `JobSchedulerService` to clean up the pending tasks that were previously submitted by the app. Meanwhile, the broadcast will be received by `NotificationManagerService`, and the notifications linked with the target app will be dismissed accordingly.

Remark: In normal cases, FSA can kill all running processes of an app. However, we find that the instant app is an exception. As shown in Figure 2b, the App info interface of an instant app does not have the *Force stop* button. That is, FSA is unavailable for stopping instant apps, and the alive instant apps can wake up other apps.

(3) Killing Background Processes (KBP).

Usage: An app with the permission `KILL_BACKGROUND_PROCESSES` can terminate background app processes. More specifically, the app first obtains the instance of `ActivityManager` and then calls `ActivityManager`'s `killBackgroundProcesses` method to kill background app processes. A debugger can execute the shell command, *am kill*, to accomplish the same task.

Internal: The workflow of KBP contains 2 steps. First, AMS retrieves all alive `ProcessRecord` objects and examines the `setAdj` field of each to determine whether a candidate will be killed. Precisely, if the candidate's OOM adjustment value is no smaller than that of `SERVICE_ADJ`, it will be terminated. Second, AMS calls `kill` API in `ProcessRecord` to kill the target.

Remark: The first step of KBP implies it cannot kill the process, whose OOM adjustment value is smaller than that of `SERVICE_ADJ`.

(4) Killing All Background Processes (KAP).

Usage: A debugger can use the shell command, *am kill-all*, to kill all app processes running in the background, because this command makes AMS call its `killAllBackgroundProcess` method.

Internal: The workflow of KAP is similar to that of KBP, and the difference lies in the criteria for finding the target app processes. Precisely, KAP kills the app processes, whose OOM adjustment values are no smaller than that of `CACHED_APP_MIN_ADJ`.

Remark: The app process, whose OOM adjustment value is smaller than the value of `CACHED_APP_MIN_ADJ`, is excluded from the target of KAP, and thus it will not be killed by KAP.

(5) Killing Specific Process (KSP).

Usage: An app or a debugger can kill a specific process according to its PID by using the native `kill` function and the *kill* command, respectively. Note that an app can just kill its own processes while the privileged debugger does not have such constraints.

Internal: KSP calls the native `kill` function to terminate the target.

Remark: KSP cannot prevent the app from being waken up.

(6) Low Memory Killer (LMK).

Usage: If a running device's available RAM becomes insufficient, LMK's native daemon will kill a few selected app processes.

Internal: LMK includes 2 major steps. First, according to the remaining available RAM, it decides a threshold for the OOM adjustment value. Second, it lists the app processes, whose OOM adjustment values exceed the threshold. Among these processes, LMK uses the native `kill` function to kill the one that occupies the most amount of RAM. Then, LMK repeatedly executes these 2 steps until no process's OOM adjustment value exceeds the threshold.

Remark: The second step suggests that LMK rarely kills the app process with a high priority (i.e., a small OOM adjustment value).

4 KEEPING APPS ALIVE

In this section, by leveraging insights in §3.3, we propose 7 diehard methods, as listed in Table 2, to keep apps alive. Note that, if an app has a process running in the system, the app is alive.

4.1 Manipulating Recent-Task Item

We propose 2 diehard methods that manipulate the items in the Recent-Task list to evade RRT.

Table 2: Diehard methods for keeping apps alive.

Methods	Support Instant App	Method for Killing App Process					
		RRT	FSA	KBP	KAP	KSP	LMK
HTI	✓	●	◐	⊗	⊗	⊗	⊗
PMI	✓	●	◐	⊗	⊗	⊗	⊗
HFA	✓	⊗	⊗	●	●	⊗	◐
HFS	✓	●	◐	●	●	⊗	◐
COW	✓	●	◐	●	●	⊗	◐
BRS	✓	●	◐	●	●	⊗	◐
ACP	✓	●	◐	●	●	⊗	◐

1 ● denotes that the method can prevent normal apps and instant apps from being killed; ◐ denotes that the method can prevent normal apps or instant apps from being killed to a certain extent; ⊗ denotes that the method cannot prevent normal apps and instant apps from being killed.

● **Hiding Recent-Task Item (HTI).**
Design: Since RRT only stops the app that has an item in the Recent-Task list, the app can hide the item to evade RRT.
Implementation: To hide the Recent-Task item, an app can set the `excludeFromRecents` property of its main launchable activity to `true` in its manifest file or code. For the latter, the app first obtains the instance of `ActivityManager` and calls `getAppTasks` of `ActivityManager` to get the `AppTask` object that is bound with the Recent-Task item. Then, the app calls `setExcludeFromRecents` defined in the `AppTask` class to remove the Recent-Task item.

● **Producing Multiple Recent-Task Items (PMI).**
Design: Since RRT kills the process that organizes the hosted activities in an activity Task, to bypass it, the app can create multiple activity Tasks, which results in multiple Recent-Task items.
Implementation: To make Android create an additional activity Task for storing the activity, the `Intent` object used for launching the target activity should carry the flag `FLAG_ACTIVITY_NEW_TASK` [23]. Note that if the `taskAffinity` property of the source activity (i.e., intent sender) and the one of the target activity (i.e., intent receiver) are the same, Android will not create the new Task. To prevent this circumstance, the app should let the `taskAffinity` of the target activity be different from that of the source activity.

4.2　Escalating App Process Priority

Since RRT, KBP, KAP, and LMK kill app processes according to their priorities (i.e., the OOM adjustment or the thread scheduling group) as described in §3.3, we propose 5 methods to escalate an app process's priority for preventing the process from being killed or decreasing its probability of being killed.

● **Holding Foreground Activity (HFA).**
Design: As shown in Table 1, the app process (i.e., **C01** or **C02**) holding the foreground activity has the smallest OOM adjustment value. Such processes will not be killed by KBP and KAP because they select app processes with lower priorities.
Implementation: To let a background app process hold a foreground activity, the app can declare or register a receiver that listens to system/app broadcasts. Once the app receives the broadcast, it will start an activity in the foreground. Diehard apps can further make HFA hard to be noticed. For example, the apps listen to the system broadcast `ACTION_SCREEN_OFF` and start the foreground activity when the device screen turns off or launch a very small activity.

● **Hosting Foreground Service (HFS).**

Design: The OOM adjustment for the app process (i.e., **C08**) that hosts a foreground service is `PERCEPTIBLE_APP_ADJ`, whose value is smaller than that of `SERVICE_ADJ` and `CACHED_APP_MIN_ADJ`. Hence, such app processes will not be killed by either KBP or KAP. Meanwhile, since the relevant thread scheduling group is `SCHED_GROUP_DEFAULT`, RRT cannot kill such app processes.
Implementation: An app first launches a service by creating an `Intent` object and calling `startService` or `bindService`. Then, to make the service foreground, in the `onCreate`, `onStartCommand`, or `onBind` callback of the service, the app creates a `Notification` object and passes it to the `startForeground` API, which will show a notification to inform the user that there is a foreground service.

● **Creating Overlay Window (COW).**
Design: An app process (i.e., **C09**) can launch an overlay window [17, 22], which will be rendered on top of any other app windows (e.g., interfaces of app activities), to adjust its thread scheduling group to `SCHED_GROUP_DEFAULT` and elevate its OOM adjustment to `PERCEPTIBLE_APP_ADJ`. Consequently, such app processes will not be killed by RRT, KBP, and KAP.
Implementation: Before Android 8.0, to make an app window become an overlay, the app calls `setAttribute` or `setType` defined in the `Window` class to specify the window's type with `TYPE_PHONE`, `TYPE_SYSTEM_ALERT`, `TYPE_SYSTEM_ERROR`, or `TYPE_TOAST`. For newly released Android systems (i.e., those published no earlier than Android 8.0), `TYPE_APPLICATION_OVERLAY` is usually adopted to construct the overlay window [32] because the aforementioned window types are deprecated.

● **Binding Running Service (BRS).**
Design: An app can run its services in separate processes (e.g., service processes) to reduce the resource occupation of its main process. Since the service process will usually not host any user perceptible components (e.g., UI), its OOM adjustment value is larger than that of `PERCEPTIBLE_APP_ADJ`. Hence, to prevent the service process from being killed, the app needs to escalate the priority of its service processes. After scrutinizing how AMS calculates and updates the priority of the app process, we find that the priority of the service process can be affected by the priorities of its client processes (i.e., **C03**, **C06**, and **C10**) that bind themselves to the service. Specifically, if the OOM adjustment value of the service process is bigger than the values of some of its client processes, its value will be adjusted to the smallest one of all its client processes, and the thread scheduling group will be set to `SCHED_GROUP_DEFAULT`. Thus, the service process will not be killed by RRT, KBP, and KAP.
Implementation: An app process calls `bindService` to bind itself to the service that is running in a separate process. Note that once all client processes disconnect themselves from the service by calling `unbindService`, the service process's priority will not be affected by the client processes.

● **Acquiring Published Content Provider (ACP).**
Design: A content provider can run in a separate app process (e.g., provider process), and its priority will be affected by its client processes (i.e., **C04**, **C07**, and **C11**). The OOM adjustment and the thread scheduling group of the provider process will be changed following the same rules as the service process mentioned in BRS. Thus, the provider process will not be killed by RRT, KBP, and KAP.
Implementation: To connect itself to the provider process, the app process calls the `acquireContentProviderClient` API or

Table 3: Diehard methods for waking up apps.

Methods	Support Instant App	Method for Killing App Process					
		RRT	FSA	KBP	KAP	KSP	LMK
CSS	✓	●	◐	●	●	●	●
MSB	✗	●	◐	●	●	●	●
LAS	✓	●	◐	●	●	●	●
UJS	✗	●	⊗	●	●	●	●
MAB	✓	●	●	●	●	●	●

1 Meanings of ●, ◐, and ⊗ are the same as those in Table 2.

the `acquireUnstableContentProviderClient` API, which are declared in the `ContentResolver` class. Note that the priority of the provider process will no longer be affected by its client processes if they are disconnected by calling the APIs, `close` or `release`, which are provided by the `ContentProviderClient` class.

5 WAKING UP APPS

Since none of the methods in Table 2 can prevent an app from being killed by KSP, solely keeping apps alive alone is not enough. Hence, diehard apps also use other methods to wake themselves up after being stopped. Note that, if a component of the stopped app has been launched, the app is waken up.

We propose 5 methods listed in Table 3 to wake up diehard apps. These methods are divided into 3 categories for different scenarios. First, the app utilizes its own service (e.g., a sticky service) to wake itself up (in §5.1). Since this method cannot relaunch normal apps stopped by FSA, we further propose methods in other categories. Second, the app leverages system components to wake itself up. For example, the app can use Alarm service to periodically launch its component to wake itself up. Note that Android system sets several constraints for apps to use the system components. For example, some of the system components are not available for instant apps and cannot timely wake up the app. Hence, apps may adopt the method in the third category. Third, the app relies on other apps to wake it up. That is, the stopped app waits for the wake-up signal (e.g., broadcasts) from other apps. Once received, the signal launches the stopped app's component that responds to the signal, and then the app is waken up.

5.1 Utilizing App Component

A stopped app can wake itself up through its specific components.
• **Constructing Sticky Service (CSS)**.
Design: The unique feature of the sticky service makes it suitable for executing long-time tasks [10]. Specifically, if the process running a sticky service is killed after the service has been started (i.e., its `onStartCommand` callback has been executed), Android will relaunch the service. Thus, diehard apps, which are stopped by RRT, KBP, KAP, KSP, or LMK, can use the sticky service to wake themselves up. However, this method cannot wake up the app stopped by FSA, because FSA makes Android ignore the app's sticky services.
Implementation: To turn a started service to a sticky one, the service's `onStartCommand` callback should return one of the following constants: `START_STICKY_COMPATIBILITY`, `START_STICKY`, or `START_REDELIVER_INTENT`. Using the former 2 constants, the service can be relaunched by the framework multiple times whereas the service can be relaunched only once if the last constant is

Table 4: Partial System broadcasts for implementing MSB.

Action Name	System Event
ACTION_BOOT_COMPLETED	The device has finished booting.
ACTION_LOCALE_CHANGED	The device's locale has changed.
ACTION_MEDIA_MOUNTED	An external media has been mounted.
ACTION_NEW_OUTGOING_CALL	An outgoing call is about to be placed.
ACTION_TIMEZONE_CHANGED	The timezone has changed.

used. To ensure the `onStartCommand` callback will be invoked by the framework, the app should start the sticky service using the `startService` API. Note that the relaunched service is a background service by default, which will be stopped by AMS if it has been running in the background for more than a minute [1]. To tackle this issue, the diehard app can use our method HFS in §4.2 to bring the relaunched service to the foreground.

5.2 Leveraging System Functionality

Diehard apps can be waken up by exploiting particular system functionality (e.g., system services and broadcasts). We elaborate 3 diehard methods based on app-accessible system functionality.
• **Monitoring System Broadcast (MSB)**.
Design: An app can use broadcast receivers to monitor system events [36]. Once a concerned event happens, the receiver will be launched. Thus, the app, which is stopped by RRT, KBP, KAP, KSP, or LMK, can use this method to wake itself up. However, since FSA makes the stopped app only receive the broadcasts that carry the flag, `FLAG_INCLUDE_STOPPED_PACKAGES`, which is not included in the system broadcasts that can be received by the app [6], MSB cannot wake up normal apps stopped by FSA.
Implementation: The broadcast receiver can be declared in the app's manifest file or dynamically registered by code through the framework API `registerReceiver`. Note that, if the app has been stopped, only the receivers declared in its manifest file can be launched, because Android will only inspect the app's manifest file to determine whether its receivers can respond to the system broadcasts [2]. Thus, a few system broadcasts (e.g., `ACTION_SCREEN_ON`), which can only be received by dynamically registered receivers, cannot wake up the app. Moreover, from Android 8.0, statically declared receivers are no longer permitted to receive the implicit broadcasts (e.g., most of system broadcasts), which do not specify the package name of the target receiver [1]. Hence, only the system broadcasts, which are exempted from these constraints [6], can be used to implement MSB, and we list partial of them in Table 4.
• **Leveraging Alarm Service (LAS)**.
Design: `AlarmManager` allows an app to repeatedly launch its component for executing a certain task. Accordingly, the app, which is stopped by RRT, KBP, KAP, KSP, and LMK, can use this function to wake itself up . However, LAS cannot wake up the app stopped by FSA, because FSA removes the task pending in the Alarm service.
Implementation: To repeatedly launch an app component, the app first constructs a `PendingIntent` task, which specifies the app component to be launched. Then, the app calls `setRepeating` or `setInexactRepeating` declared in the `AlarmManager` class to set the time interval for `AlarmManagerService` to schedule the task.
• **Using Job Scheduling Service (UJS)**.

```
1   // The value of the "intervalMillis" field of oldJob is no smaller than 15*60*1000.
2   public void exploit(JobInfo oldJob) {
3       // Step 1: Store the original JobInfo object using a Parcel container.
4       Parcel in = Parcel.obtain();
5       oldJob.writeToParcel(in, PARCELABLE_WRITE_RETURN_VALUE);
6       // Step 2: Revise the "intervalMillis" value to a smaller value.
7       Parcel out = Parcel.obtain();
8       // reviseParcel(*) is a customized method manipulates the buffer in the Parcel.
9       reviseParcel(in, PARCELABLE_WRITE_RETURN_VALUE, out);
10      // Step 3: Create the JobInfo object using the buffer of the Parcel container.
11      JobInfo newJob = JobInfo.CREATOR.createFromParcel(out);
12      jobScheduler.schedule(newJob);
13  }
```

Figure 3: Adjusting the time interval stored in JobInfo.

Design: Since Android 5.0, an app can ask Android system to periodically execute a job by defining it in the app's `JobService` component. More specifically, the app submits a `JobInfo` object to notify `JobSchedulerService` to launch the `JobService` component and execute the job. Thus, the app, which is stopped by RRT, KBP, KAP, KSP, or LMK, can use `JobSchedulerService` to relaunch its `JobService` component and wake itself up. However, UJS cannot wake up the app stopped by FSA, because FSA cancels the scheduled job accepted by `JobSchedulerService`. Moreover, since the `JobService` component of the instant app cannot be accessed by `JobSchedulerService` via the package name and class name [7], UJS cannot wake up instant apps.

Implementation: Diehard apps first create the `JobInfo` object by using the `JobInfo.Builder` class, and then invokes `setPeriodic` to specify the time interval for periodically performing the job defined in the `JobService` component. Note that, after Android 8.0, `setPeriodic` adjusts the minimum time interval (i.e., the value stored in the `intervalMillis` field of the `JobInfo` object) to 15 minutes for optimizing the battery usage. To bypass this limit, we discover and exploit the inconsistency of the time interval check conducted by the local `JobInfo` object and the remote `JobSchedulerService` instance[1]. Specifically, Android only checks the validity of the time interval in the client side when the app calls `setPeriodic`, rather than in the server side when `JobSchedulerService` is going to schedule the job according to the time interval stored in the `JobInfo` object. Figure 3 illustrates a proof-of-concept (PoC) that can be adopted by the diehard app to bypass the time interval constraint.

5.3 Relying on Third-party Apps

A diehard app can be waken up by colluded third-party apps, which can launch the activity, start the service, or acquire the content provider of the diehard app by explicitly specifying the names of these components. Since the relationship between the diehard app and the colluded app can be easily figured out according to the explicitly specified component names, we do not study this method in this paper. Instead, we design an approach for third-party apps to implicitly wake up the diehard app.

● **Monitoring App Broadcast (MAB).**

Design: An app can use the receiver declared in its manifest file to monitor the broadcast from third-party apps to wake itself up. Unlike system broadcasts, a well-crafted implicit broadcast can wake up the app stopped by FSA. Hence, the apps stopped by any of 6 process-killing methods can use MAB to wake themselves up.

---

[1] In the latest released Android 10.0, such vulnerability has been fixed.
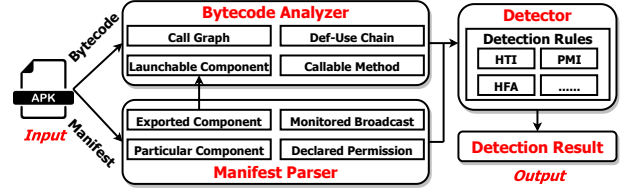

Figure 4: The overview of DiehardDetector.

Implementation: In accordance with the design, the broadcast receiver needs to be statically declared in the manifest file. The implicit app broadcast sent by other apps should satisfy 2 requirements. First, the broadcast should bypass the implicit broadcast constraint since Android 8.0 (i.e., Android no longer permits apps to send implicit broadcasts). Third-party apps can circumvent it by sending broadcast with the flag `FLAG_RECEIVER_INCLUDE_BACKGROUND`. Second, the app stopped by FSA should be able to receive the implicit broadcast. To achieve it, the third-party apps should let the implicit broadcast carry the flag `FLAG_INCLUDE_STOPPED_PACKAGES`.

6 DETECTING DIEHARD APPS

We are interested in how prevalent the diehard methods are used by apps in the wild. To conduct a large-scale study on it, we develop a new tool named DiehardDetector to check whether apps have employed the diehard methods described in §4 and §5. After giving an overview of DiehardDetector in §6.1, we detail the modules of DiehardDetector in §6.2 - §6.4.

6.1 Overview

As shown in Figure 4, DiehardDetector consists of 3 modules: manifest parser, bytecode analyzer, and detector. It takes in an app's APK file and outputs the diehard methods used by this app if any.

● **Manifest Parser** (§6.2): It parses the app's manifest file to collect the information about exported app components to facilitate the subsequent static analysis on the app's Dex file. To obtain the necessary information for detecting deihard methods, this module also retrieves the broadcasts monitored by the app, finds the app components with special attributes, and collects the declared permissions.

● **Bytecode Analyzer** (§6.3): It first constructs the call graph and builds the def-use chain by analyzing the app's Dex files. Then, it find the app's components, which cannot be launched, to prune invalid call graph edges caused by the conservative graph construction strategy [12], which assumes that all the app's components are launchable. Subsequently, it collects the nodes, which denote methods, including app defined methods and framework APIs, in the pruned call graph for identifying diehard methods.

● **Detector** (§6.4): Taking the results of the manifest parser and the bytecode analyzer as inputs, this module discovers the diehard methods used by the app according to the rules defined for the diehard method described in §4 and §5.

6.2 Parsing Manifest File

We collect the following information from the manifest file.
● **Exported App Components**. To collect app components that can be launched externally, DiehardDetector inspects the `exported`

attribute of each declared app component and stores exported ones to the set $S_{exported}$. Note that, if a component has `intent-filters`, it can receive external intents, and thus DiehardDetector also saves such components to $S_{exported}$.
• **Monitored Broadcasts**. To obtain the broadcasts monitored by the statically declared receivers, DiehardDetector examines the `intent-filters` of each broadcast receiver, and puts the system broadcasts and app broadcasts to 2 sets, namely $S_{syscast}$ and $S_{appcast}$, respectively.
• **Special App Components**. To find the app components that are relevant to diehard methods, DiehardDetector resolves each app component's attributes. Precisely, the service or the content provider with the `process` attribute, which makes the component run in a separate process, will be stored to the set $S_{process}$. Moreover, the activity with the `taskAffinity` attribute, which allows the activity to be placed in a separate activity Task, will be saved to the set $S_{affinity}$. If an activity enables its `excludeFromRecents` attribute, which makes the corresponding activity Task not be associated with a Recent-Task item, such the activity will be recorded to the set $S_{recents}$.
• **Declared Permissions**. To identify the diehard methods (e.g., COW) that need specific permissions, DiehardDetector parses the `uses-permission` tags in the manifest file and stores the declared permissions to the set $S_{permission}$.

6.3 Analyzing Bytecode

To facilitate the detection in §6.4, DiehardDetector performs static analysis to construct an accurate call graph, build def-use chain, and identify launchable components as well as callable methods.
• **Call Graph**: To find the execution path for identifying HFA and HFS, DiehardDetector first uses FlowDroid [12] to build the app's call graph and then prune invalid edges.

Given an app, FlowDroid creates an entry-point and connects it with the callbacks of app components under the assumption that all app components are launchable. That is, the callbacks of app components can always be executed. Then, FlowDroid finds the methods that are reachable from the callbacks to build the call graph. However, since some app components can never be launched, the call graph contains invalid edges, which may bring false positives to the detection results of diehard apps.

To prune invalid edges, DiehardDetector finds launchable app components, keeps the edges that are reachable from the callbacks of launchable app components, and removes the others (i.e., invalid edges). Normally, there are 3 types of launchable app components: (1) exported app components, which can be launched from the external (e.g., users or other apps); (2) the app components, which can be launched by exported app components; (3) the app components, which can be launched by launchable app components. Accordingly, since exported app component are always launchable, DiehardDetector finds the launchable unexported app components to prune invalid edges.

The process for pruning invalid edges consists of 3 steps. In the first step, DiehardDetector finds the unexported app components that can be launched by other app components. In detail, Diehard-Detector locates the framework APIs (e.g., those listed in Table 6) for launching the app components. Then, DiehardDetector performs

def-use analysis on the `Intent` objects past to these APIs because the uses of these `Intent` objects contain the names of the launchers (i.e., the app components that launch the unexported app components) and the unexported app components to be launched. In the second step, to ensure the launched unexported app components are launchable app components, DiehardDetector checks whether they are launched by launchable app components. More specifically, DiehardDetector inspects whether the launchers of the unexported app components are launchable app components. If so, such the unexported app components are launchable app components. In the third step, to prune invalid edges, DiehardDetector traverses the call graph from the callbacks of identified launchable app components, and then removes the edges that cannot be accessed.
• **Def-Use Chain**: To find the variable definitions and uses for detecting COW, BRS, and ACP, DiehardDetector builds def-use chains [18] of selected variables. More specifically, to get the window type of the created overlay for detecting COW, DiehardDetector builds def-use chains of the variables that will be assigned to the `a` parameter of Window's `setAttributes` method or the `type` parameter of Window's `setType` method. To find the app services that will not be unbound for detecting BRS, DiehardDetector builds def-use chains of `ServiceConnection` objects, which will be passed to the `bindService` API or the `unbindService` API. To find the content providers that will not be released or closed for detecting ACP, DiehardDetector builds def-use chains of `ContentProviderClient` objects, which are either the returned objects of `acquireContentProviderClient` or the base objects of invocations to `ContentProviderClient`'s `release/close` method.

Moreover, to find the app components that will be launched using special `Intent` flags, which are essential for identifying PMI, CSS, and MAB, DiehardDetector builds def-use chains of Intent objects passed to framework APIs that are used to launch app components. DiehardDetector stores the activities launched with the `FLAG_ACTIVITY_NEW_TASK` flag to the set S_{task}, saves the services started by `startService` to the set $S_{startService}$, and puts the broadcasts sent with flags, `FLAG_RECEIVER_INCLUDE_BACKGROUND` and `FLAG_INCLUDE_STOPPED_PACKAGE` to the set $S_{sndcast}$.
• **Launchable Components**: To find launchable app components for identifying HFA and HFS, DiehardDetector examines the call graph to get the launchable app components' callbacks, whose method signatures contain the names of the app components. DiehardDetector saves the launchable app components to the set $S_{launchable}$.
• **Callable Methods**: To get the callable methods for recognizing HTI, COW, LAS, and UJS, DiehardDetector stores the nodes in the call graph, each of which corresponds to a method that is reachable from callbacks of launchable app components, to the set $S_{callable}$.

6.4 Detecting Diehard Methods

DiehardDetector identifies various diehard methods according to the rules summarized in Table 5. We explain them as follows.
(1) HTI: DiehardDetector examines $S_{exported}$ and $S_{recents}$ to find whether an exported activity's `excludeFromRecents` attribute is set to `true`. If no activity is found, DiehardDetector will inspect $S_{callable}$ to find whether `setExcludeFromRecents` can be called.

Table 5: Core rules for detecting diehard methods.

Behavior	Detection Rule
HTI	$(S_{exported} \cap S_{recents} \neq \varnothing) \vee ((\texttt{AppTask.setExcludeFromRecents} \in S_{callable}) \wedge (args[0] = true))$
PMI	$(S_{affinity} \cap S_{task}) \neq \varnothing$
HFA	$\exists$ receiver, (receiver $\in S_{launchable}) \wedge ((\text{path}(onReceive, *.startActivity) \neq \varnothing) \vee (\text{path}(onReceive, PendingIntent.getActivity) \neq \varnothing))$
HFS	$\exists$service, (service$\in S_{launchable})\wedge((\text{path}(onCreate, startForeground) \neq \varnothing) \vee (\text{path}(onBind, startForeground) \neq \varnothing) \vee (\text{path}(onStartCommand, startForeground) \neq \varnothing))$
COW	$(\texttt{SYSTEM\_ALERT\_WINDOW} \in S_{permission}) \wedge (((\texttt{Window.setAttributes} \in S_{callable}) \wedge (args[0].type \geq \texttt{TYPE\_PHONE})) \vee ((\texttt{Window.setType} \in S_{callable}) \wedge (args[0] \geq \texttt{TYPE\_PHONE})))$
BRS	$\exists$ serviceConnection, (serviceConnection $\in S_{bind}) \wedge$ (serviceConnection $\notin S_{unbind}$)
ACP	$\exists$ contentProviderClient, (contentProviderClient $\in S_{acquire}) \wedge$ (contentProviderClient $\notin (S_{release} \cup S_{close})$)
CSS	$\exists$ service, ((service $\in S_{exported}) \vee$ (service $\in S_{startService})) \wedge$ (service $\in S_{sticky}$)
MSB	$\exists$ broadcast, (broadcast $\in$ system broadcasts that are exempted from the implicit broadcast constraint) $\wedge$ (broadcast $\in S_{syscast}$)
LAS	$(\texttt{AlarmManager.setInexacRepeating} \in S_{callable}) \vee (\texttt{AlarmManager.setRepeating} \in S_{callable})$
UJS	$\texttt{JobInfo\$Builder.setPeriodic} \in S_{callable}$
MAB	$((S_{appcast} \cap S_{sndcast}) \neq \varnothing) \wedge (S_{sndcast}$ comes from another app)

1 The syntax *args[index]* returns the specific argument of the callable method; The syntax *path(src,tgt)* obtains the invocation chains that start from the *src* method and end with the *tgt* method.

Table 6: Partial APIs for launching app components.

Component	Framework APIs
Activity	Context.startActivity(Intent) Activity.startActivity(Intent) PendingIntent.getActivity(Context,int,Intent,int)
Service	Context.startService(Intent) PendingIntent.getService(Context,int,Intent,int) Context.bindService(Intent,ServiceConnection,int)
Broadcast Receiver	Context.sendBroadcast(Intent) PendingIntent.getBroadcast(Context,int,Intent,int) Context.registerReceiver(BroadcastReceiver,IntentFilter)

If any one of conditions is satisfied, which implies the app will hide its Recent-Task item, HTI is detected.

(2) PMI: If an activity is included in both $S_{affinity}$ and S_{task}, PMI is detected, because launching the activity will let the app have multiple activity Tasks, corresponding to multiple Recent-Task items.

(3) HFA: DiehardDetector analyzes the call graph to decide whether the `onReceive` callback of a launchable broadcast receiver will call `PendingIntent`'s `startActivity` or `getActivity` to launch the activity. If found, HFA is recognized, because a foreground activity can be launched when the app is running in the background.

(4) HFS: DiehardDetector detects HFS by examining callbacks (i.e., `onCreate`, `onBind`, and `onStartCommand`) of each app service in $S_{launchable}$ to determine whether `startForeground` will be invoked by the callbacks. If so, it denotes that the app will launch the foreground service, and thus HFS is detected.

(5) COW: DiehardDetector first checks $S_{permission}$ to find out whether the `SYSTEM_ALERT_WINDOW` permission has been requested by the app. If so, DiehardDetector analyzes the def-use chains of variables passed to the `Window`'s `setAttributes` or `setType` method to get the type of the window created by the app. If the window type's value is no smaller than that of `TYPE_PHONE`, COW is found because the app will create an overlay [32].

(6) BRS: The core for detecting BRS is to find the remote app service that can be connected through `bindService`, and such connection will never be broke via `unbindService`. Accordingly, DiehardDetector analyzes the def-chains of `ServiceConnection` objects passed to `bindService` or `unbindService`, and stores the found variable definitions to sets, S_{bind} and S_{unbind}, respectively. If the intersection of these two sets is not a null set, which implies some service bindings will never be broken, DiehardDetector then analyzes the

def-use chains of `Intent` objects passed to `bindService` to find the bound service. If the service is also included in $S_{process}$, the remote service will use BRS to escalate its process priority.

(7) ACP: DiehardDetector identifies ACP by finding the acquired `ContentProviderClient` objects that will not be released or closed. We store the definitions of the acquired objects (i.e., the returned objects of either `acquireUnstableContentProviderClient` or `acquireContentProviderClient`), the closed objects (i.e., the base objects of the invocations to `ContentProviderClient`'s `close` method), and the released objects (i.e., the base objects of the invocations to `ContentProviderClient`'s `release` method), to the sets $S_{acquire}$, S_{close}, and $S_{release}$, respectively. If there is an object, which is included in $S_{acquire}$ but is excluded from $S_{release}$ and S_{close}, and its corresponding content provider is included in $S_{process}$, we find a remote content provider that will use ACP to elevate its process priority.

(8) CSS: DiehardDetector examines the return value of each app service's `onStartCommand` callback to find sticky services, and stores them to the set S_{sticky}. If the intersection of S_{sticky} and $S_{startService}$ is not a null set, CSS is recognized, because the app will start the sticky service by using `startService`,

(9) MSB: If the app declares receivers to monitor the system broadcasts, which are exempted from the implicit broadcast constraint, we consider that such an app implements MSB.

(10) LAS: To recognize LAS, DiehardDetector inspects $S_{callable}$ to find out whether `setRepeating` or `setInexactRepeating` declared in the `AlarmManager` class will be called by the app. If so, it suggests the app will use Alarm service to relaunch its component.

(11) UJS: DiehardDetector checks $S_{callable}$ to figure out whether the `setPeriodic` method defined in the `JobInfo$Builder` class will be called by the app. If so, the `JobService` component of the app will be relaunched, and thus UJS is found.

(12) MAB: If an app broadcast included in $S_{appcast}$ is presented in $S_{sndcast}$ of another app, MAB is detected because the app under examination listens to the well-crafted broadcast from another app,

7 EVALUATION

In this section, we evaluate the applicability of our diehard methods, the performance of DiehardDetector, and the prevalence of diehard apps in the wild by answering the following 3 research questions.

Table 7: Diehard methods found in data-2k apps.

Behavior	#app	#app'	Δapp	Behavior	#app	#app'	Δapp
HTI	118	118	+0	ACP	0	0	+0
PMI	10	13	+3	CSS	72	91	+19
HFA	2	2	+0	MSB	452	452	+0
HFS	139	240	+101	LAS	158	165	+7
COW	11	14	+3	UJS	19	19	+0
BRS	0	0	+0	MAB	0	0	+0

RQ1: Do our diehard methods work on most popular Android versions, ranging from Android 5.1 to Android 10.0?
RQ2: How is the performance of DiehardDetector?
RQ3: How prevalent are diehard apps?

Data Set: We use 3 data sets including data-3, data-2k, and data-80k for answering the 3 research questions, respectively. To answer **RQ1**, we developed a normal app and an instant app, both of which implement all available diehard methods for their app types. We also developed an auxiliary normal app for verifying MAB. To answer **RQ2**, we downloaded 2,080 open-source apps from F-Droid [4] to form the benchmark. To answer **RQ3**, we randomly crawled 81,237 apps from Google Play [5].

7.1 RQ1: Applicability of Diehard Methods.

Methodology: To assess the applicability of our diehard methods, we run each app in data-3 on various versions of official Android systems, which are released by Google.
Result: All our diehard methods can prolong the liveness time of apps running on the official Android systems, ranging from 5.1 to 10.0. Since Android 10.0 has patched the vulnerability in the time interval check (mentioned in §5.2), we cannot change the minimum time interval (15 mins) for `JobSchedulerService` to execute the periodic job and thus UJS cannot timely relaunch the app.

> **Answer to RQ1:** All our 12 diehard methods are applicable to popular Android systems, ranging from Android 5.1 to Android 10.0. However, Android 10.0 constraints the ability of UJS.

Table 8: Apps using diehard methods in data-80k.

Behavior	#app	Ratio	Behavior	#app	Ratio
HTI	1,878	10.79%	ACP	0	0.00%
PMI	424	2.44%	CSS	1,933	11.14%
HFA	125	0.72%	MSB	12,413	71.30%
HFS	2,243	12.88%	LAS	6,452	37.06%
COW	264	1.52%	UJS	130	0.75%
BRS	21	0.12%	MAB	0	0.00%

Table 9: Number of diehard methods used in each app.

	data-2k (632)			data-80k (17,410)	
Number	#app	Ratio	Number	#app	Ratio
1	363	57.44%	1	11,213	64.41%
2	205	32.44%	2	4,444	25.53%
3	50	7.91%	3	1,375	7.90%
4	12	1.90%	4	272	1.56%
5	2	0.31%	5	75	0.43%
6	0	0.00%	6	31	0.17%

7.2 RQ2: Performance of DiehardDetector.

Methodology: To evaluate the performance of DiehardDetector, we first use it to analyze the apps in data-2k, and then manually analyze the source code of the apps to determine whether they do (or do not) have diehard behaviors. More specifically, we manually analyzed 500 randomly selected apps in data-2k, 250 of which are detected diehard apps while the remaining 250 are normal apps.
Result: Table 7 lists the detection results, where #*app* denotes the number of apps having a specific diehard behavior. Since FlowDroid fails to process 9 apps in data-2k due to the over-long analysis time (> 30 mins), DiehardDetector successfully analyzed 2,071 apps, and found 632 (30.52%) of them use diehard methods.

After analyzing 250 detected diehard apps manually, we did not find any false positives. For the other 250 apps, we found 14 false negatives, which make FlowDroid throw an exception when constructing the call graph. Consequently, their call graphs include neither nodes nor edges so that DiehardDetector cannot detect their diehard methods (except for PMI) and thus treats them as non-diehard apps. Accordingly, the precision and the recall of DiehardDetector is 100% and 94.70%, respectively.

To evaluate the effect of pruning the call graph (§6.3), we compare the detection results (#*app*) with those (#*app'*) achieved without removing invalid call graph edges. Specifically, Δ*app* shows the increased number of identified apps having a specific diehard behavior (i.e., Δ*app* = #*app'* - #*app*). In detail, if invalid edges are not removed, DiehardDetector will cause 53 false positives. Moreover, we notice that over 40% of #*app'* for HFS are false positives because the recognized foreground services will never be launched. Hence, removing invalid call graph edges is important to DiehardDetector.

> **Answer to RQ2:** DiehardDetector can identify diehard apps with high precision (100%) and recall (94.7%).

Table 10: Ratio of diehard apps in various app categories.

Category	Ratio	Category	Ratio	Category	Ratio
Personalization	40.26%	Social	25.26%	Entertainment	18.26%
Communication	39.44%	Video Players	24.91%	Photography	18.04%
News&Magazines	31.80%	Lifestyle	22.33%	Finance	17.31%
Tools	31.47%	Health&Fitness	22.17%	Education	14.39%
Music&Audio	29.40%	Business	21.87%	Book&Reference	13.92%
Shopping	27.60%	Travel&Local	20.02%	Games	10.10%

7.3 RQ3: Prevalence of Diehard Apps.

Methodology: To detect diehard apps in the wild and investigate the popularity of diehard methods, we apply DiehardDetector to analyzing the apps in data-80k crawled from Google Play.
Result: The average time for DiehardDetector to analyze an app is 45 seconds. Table 8 lists the overall result, where *Ratio* is the ratio of diehard apps with a specific diehard method. It shows that 17,410 (around 21%) of Google Play apps in our dataset use diehard methods, and thus diehard apps are prevalent in the app ecosystem. Moreover, MSB, LAS, CSS, HFS, and HTI are the most popular diehard methods, and roughly 70% of diehard apps implement MSB. It is interesting that ACP and MAB were not found in data-2k and data-80k. Thus, we discover 2 new ways for apps to keep their processes alive and wake themselves up.

Since a diehard app may adopt more than one diehard method, we also analyze the number of diehard methods used in each diehard app, and the results are shown in Table 9. In detail, more than 35% of diehard apps adopt more than one diehard methods, but very few diehard apps employ more than four diehard methods.

We also study the distribution of diehard apps in different app categories, and the results are listed in Table 10. It shows that more than 40% of apps in Personalization category use diehard methods. Moreover, the categories including Communication, News & Magazines, and Tools have more than 30% diehard apps.

> **Answer to RQ3:** Around 21% of Google Play apps in our dataset adopt diehard methods. The top 5 ones are MSB, LAS, CSS, HFS, and HTI. ACP and MAB have not been widely used yet. Moreover, diehard methods are widely used by apps in categories like Personalization, Communication, News & Magazines, and Tools.

8 THREAT TO VALIDITY

The threats to the external validity of DiehardDetector come from 2 aspects. First, some special app samples may negatively affect the performance of DiehardDetector. More precisely, due to the intrinsic problem of static analysis, DiehardDetector cannot handle the apps that adopt advanced obfuscations or packing techniques to prevent the app's bytecode from being analyzed. To tackle this issue, we may use deobfuscators (e.g., TIRO [29]) and unpackers (e.g., PackerGrind [30, 31]) to recover the protected bytecode. Moreover, some apps may use native code to call the methods (e.g., `startActivity`) relevant to constructing the app's call graph and detecting diehard methods. FlowDroid may miss them because it cannot analyze the native code of the app. To mitigate this problem, we may employ JN-SAF [28] to get the method invocations implemented in the app's native code. Second, although we design 12 diehard methods, we might miss some other diehard methods adopted by apps in the wild. Nevertheless, this is the most comprehensive study of diehard behaviors, and over half of the proposed diehard methods have never been mentioned by existing studies.

9 DISCUSSION

• **Applicability of Diehard Methods**: In section §7.1, we evaluate the applicability of diehard methods using different versions of the Android systems released by Google. However, to achieve the purpose of prolonging the battery life, mobile vendors will introduce their own process-killing methods to their customized Android systems, which may compromise the effectiveness of the presented diehard methods. In future work, we will further evaluate the applicability and the effectiveness of the proposed diehard methods using different mobile vendors' customized Android systems.

• **Intentional Diehard Behaviors**: In this paper, we do not distinguish the diehard behavior that is intentionally implemented by the adversary and the one that is carelessly produced by the benign developer. In future work, we intend to differentiate these two types of diehard behaviors. For instance, if an app relaunches a dead app process after it checks the aliveness state of this process, we may consider such procedure implies that the diehard behavior is intentionally performed. Additionally, if the behavior for keeping the app process alive does not involve any user interactions, we may also treat it as an intentionally performed diehard behavior.

10 RELATED WORK

This section introduces the studies that are most related to finding diehard apps. First, recent studies identify unusual app behaviors. Shan et al. [24] propose a static analysis tool to detect self-hiding behaviors, which prevent the app from being noticed by users. OverlayChecker [32] runs the app in an emulator to examine whether an overlay window will be launched by the app at runtime. DDAX [25] performs static analysis to find the app that abuses the device administrator, a sensitive functionality provided by Android system. Shao et al. [26] developed a tool to analyze an app's lifecycle, which may be used to detect some diehard methods, e.g., HFS, CSS, MSB, LAS, and UJS. However, it cannot detect other diehard methods proposed by us. Moreover, we are the first to reveal the limitations of existing process-killing methods, and exploit them to design diehard methods. It is worth noting that 7 of 12 proposed diehard methods were not mentioned in [26].

Second, researchers studied the factors affecting the energy consumption of Android apps. EnergyPatch [14] and μDroid [19] pointed out that improper operations on energy-intensive hardware components cause apps to exhibit poor energy consumption. HOT-PEPPER [15] reveals that the bad implementation practices in apps have negative impacts on their energy consumption. Oliveira et al. [21] and Chowdhury et al. [16] found that the programming languages used to develop apps and the runtime execution logs affect their energy consumption. GEMMA [20] optimizes the colors used by apps to reduce the energy consumption on smartphones' displays. However, none of them mentioned that diehard behaviors can affect the energy consumption of apps.

11 CONCLUSION

We conduct the first systematic investigation on diehard apps and diehard methods. By revealing and exploiting the limitations of existing process-killing methods, we design 12 practical diehard methods for keeping alive app processes or waking up the stopped app, which work on popular Android versions from 5.1 - 10.0. To automatically identify the presence of diehard methods, we develop DiehardDetector that can accurately and quickly detect our diehard methods. The extensive experimental results show that DiehardDetector achieves high precision and recall. By applying DiehardDetector to more than 80k apps fetched from Google Play, we observe that around 21% of apps have adopted diehard methods, suggesting that diehard apps are prevalent in the wild. To help users understand the diehard behaviors, app stores may ask developers to describe such behaviors in their app's privacy policies[34, 35] and check whether the apps have disclosed all their diehard behaviors[33].

12 ACKNOWLEDGEMENT

REFERENCES

[1] 2020. Background Execution Limits. https://developer.android.com/about/versions/oreo/background.

[2] 2020. Broadcasts Overview. https://developer.android.com/guide/components/broadcasts.

[3] 2020. Don't kill my app! https://dontkillmyapp.com/.

[4] 2020. F-Droid. https://f-droid.org.

[5] 2020. Google Play Store. https://play.google.com/store.

[6] 2020. Implicit Broadcast Exceptions. https://developer.android.com/guide/components/broadcast-exceptions.

[7] 2020. Launch Instant App. https://developers.google.com/android/reference/com/google/android/gms/instantapps/Launcher.

[8] 2020. Recents Screen. https://developer.android.com/guide/components/activities/recents.

[9] 2020. Smartphone users still want long-lasting batteries more than shatterproof screens. https://today.yougov.com/topics/technology/articles-reports/2018/02/20/smartphone-users-still-want-longer-battery-life.

[10] 2020. Sticky Service. https://developer.android.com/reference/android/app/Service#START_STICKY.

[11] 2020. Understand Tasks and Back Stack. https://developer.android.com/guide/components/activities/tasks-and-back-stack.

[12] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flow-Droid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. PLDI*.

[13] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proc. CCS*.

[14] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. 2018. EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps. *IEEE Transactions on Software Engineering* (2018).

[15] Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2017. Investigating the energy impact of Android smells. In *Proc. SANER*.

[16] Shaiful Chowdhury, Silvia Di Nardo, Abram Hindle, and Zhen Ming Jiang. 2018. An exploratory study on assessing the energy impact of logging on Android applications. *Empirical Software Engineering* (2018).

[17] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proc. S&P*.

[18] Mary Jean Harrold and Mary Lou Soffa. 1994. Efficient Computation of Interprocedural Definition-Use Chains. *ACM Trans. Program. Lang. Syst.* (1994).

[19] Reyhaneh Jabbarvand and Sam Malek. 2017. μDroid: An Energy-Aware Mutation Testing Framework for Android. In *Proc. FSE*.

[20] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2017. GEMMA: Multi-objective Optimization of Energy Consumption of GUIs in Android Apps. In *Proc. ICSE*.

[21] Wellington Oliveira, Renato Oliveira, and Fernando Castor. 2017. A Study on the Energy Consumption of Android App Development Approaches. In *Proc. MSR*.

[22] Chuangang Ren, Peng Liu, and Sencun Zhu. 2017. WindowGuard: Systematic Protection of GUI Security in Android. In *Proc. NDSS*.

[23] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. 2015. Towards Discovering and Understanding Task Hijacking in Android. In *Proc. USENIX Security*.

[24] Zhiyong Shan, Iulian Neamtiu, and Raina Samuel. 2018. Self-hiding Behavior in Android Apps: Detection and Characterization. In *Proc. ICSE*.

[25] Zhiyong Shan, Raina Samuel, and Iulian Neamtiu. 2019. Device Administrator Use and Abuse in Android: Detection and Characterization. In *Proc. MobiCom*.

[26] 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 *Proc. EuroSys*.

[27] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering* (2014).

[28] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. JN-SAF: Precise and Efficient NDK/JNI-Aware Inter-Language Static Analysis Framework for Security Vetting of Android Applications with Native Code. In *Proc. CCS*.

[29] Michelle Y Wong and David Lie. 2018. Tackling runtime-based obfuscation in Android with TIRO. In *Proc. USENIX Security*.

[30] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive unpacking of Android apps. In *Proc. ICSE*.

[31] L. Xue, H. Zhou, X. Luo, L. Yu, D. Wu, Y. Zhou, and X. Ma. 2020. PackerGrind: An Adaptive Unpacking System for Android Apps. *IEEE Transactions on Software Engineering* (2020).

[32] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. 2019. Understanding and Detecting Overlay-based Android Malware at Market Scales. In *Proc. MobiSys*.

[33] L. Yu, X. Luo, J. Chen, H. Zhou, T. Zhang, H. Chang, and H. Leung. 2019. PPChecker: Towards Accessing the Trustworthiness of Android Apps' Privacy Policies. *IEEE Transactions on Software Engineering* (2019).

[34] Le Yu, Tao Zhang, Xiapu Luo, and Lei Xue. 2015. AutoPPG: Towards Automatic Generation of Privacy Policy for Android Applicationss. In *Proc. SPSM*.

[35] L. Yu, T. Zhang, X. Luo, L. Xue, and H. Chang. 2017. Towards Automatically Generating Privacy Policy for Android Apps. *IEEE Transactions on Information Forensics and Security* (2017).

[36] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *Proc. S&P*.