ServDroid: Detecting Service Usage Inefficiencies in Android Applications

Anonymous Author(s)

ABSTRACT

Services are frequently-used components in Android applications, which are responsible for performing time-consuming operations on the background. While services play a crucial role, our study finds that, service usage in practice is not so efficient as expected, reducing the performance of applications, e.g., unnecessary resource occupation and energy consumption. In this paper, according to the service lifecycle, we first formulate four anti-patterns that result in service usage inefficiencies, including premature create, late destroy, premature destroy, and service leak. Since these service usage efficiency bugs do not cause application crashes, it is difficult for existing testing approaches to find them. To this end, we present a static analysis approach, ServDroid, to detect such efficiency bugs in Android applications and output the callers of the services to facilitate debugging. We apply ServDroid to 45 popular real-world Android apps, and surprisingly find that service usage inefficiencies are pervasive.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**; **Software testing and debugging**;

KEYWORDS

Android app, background service, usage inefficiency, service leak, static analysis

ACM Reference Format:
Anonymous Author(s). 2018. ServDroid: Detecting Service Usage Inefficiencies in Android Applications. In *Proceedings of ESEC/FSE'18.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Mobile applications (apps) are now eating the world. Millions of apps are available on Google Play Store and Apple App Store, and many apps have millions or even billions of downloads. The improvement of computation and memory capacity of mobile devices allows developers to design increasingly powerful and complex apps. On the other hand, more complex apps usually involve more bugs and vulnerabilities, which significantly impact the release and adoption of apps. Thus, the quality of mobile apps receives an increasing attention [4, 7, 10, 16, 21, 24, 25, 27, 31, 33, 35].

ESEC/FSE'18, November 04-09, 2018, Florida, USA
© 2018 Association for Computing Machinery.
ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

Testing is an effective means to find bugs and to improve the quality of apps. Since Android is popular and open-source, many testing approaches are presented for Android apps. However, most of them focus on GUI testing of foreground activities [4, 6, 10, 13, 27, 31, 34, 35], whereas background services have received much less research attention [40]. Services are Android components, performing long-running tasks that involve few or no user interactions, e.g., file I/O, music playing, network transaction [14]. A service executes on the main thread of the calling component's process, and thus a new thread is often created to perform the long-running operations. Services in Android fall into two categories: system services and app services. Our work focuses on the latter. According to how app services are used in the code, they can be further divided into three types: *started services*, *bound services*, and *hybrid services*.

Although app services usually do not have user interface, they can keep running even when users shut down the device screen. As a consequence, if an app involves service-related bugs, not only the functionalities of the app but its performance (e.g., resource utilization, energy consumption) will be affected significantly. In this paper, we focus on service usage inefficiencies (a kind of non-functional bugs) that do not immediately lead to app crashes, but indeed have a significant impact on the performance of the app. This problem is seldom considered in the literature. Since such bugs do not cause app crashes, existing service testing methods may meet difficulties in detecting such bugs [40]. Although there is some work on non-functional testing (mainly energy testing) of apps [7, 20, 21, 24, 25], the oracle based on performance indicators is generally more expensive and more labour intensive than functional testing [21].

To address the problem, instead of using testing, we induce four anti-patterns to facilitate the analysis of service usage efficiency bugs. The four anti-patterns are distilled from our manual analysis on real-world apps, including *premature create*, *late destroy*, *premature destroy*, and *service leak*. They are all defined based on the lifecycle of services [14]. Premature create refers to the situation that a service is created too early before it is really used. Hence, the service is in an idle state beginning from its create to its real use, occupying unnecessary memory and consuming unnecessary energy. Late destroy refers to the situation that a service is destroyed too late after its use. Similarly, the service is in an idle state beginning from the end of its final use to the moment it is destroyed. Premature destroy refers to the situation that a service is initiated by a component (caller) but it is destroyed before another component begins to use it. Consequently, the service should be created again to fulfil the new request. Service leak refers to the scenario that a service is never destroyed after its use, even when the apps which initiate the service terminate. We observe from real-world apps that these four kinds of efficiency bugs are common.

For each of the four service usage anti-patterns, we then adopt a static analysis technique to detect its instances (concrete efficiency bugs matching the anti-pattern) in an app. The analysis is based on Soot [36][1], a framework for transforming and analyzing Java and Android applications. Our approach takes an app (APK file) as input and transforms the APK file into a Jimple file (an intermediate representation supported by Soot [8]). Our approach then constructs a context-insensitive inter-procedural control flow graph of the app (Jimple file). Based on this control flow graph, our approach leverages static analysis (e.g., (post-)dominator analysis, successor analysis) to find the service usage efficiency bugs. Finally, it outputs all detected service usage efficiency bugs as well as the components (callers) which initiate the services to facilitate debugging. We implement our approach in an open-source tool ServDroid. The tool is written in Java and is publicly available at (only available after the double-blind review).

To investigate the service usage efficiency bugs in real-world apps, we apply ServDroid to a total of 45 the most popular free Android apps listed on Wikipedia[2]. The downloads of these apps are all over 500 million. Our experiment shows that service usage efficiency bugs are surprisingly pervasive in the most popular real-world apps: 42 (93.33%) apps involve at least one kind of service usage efficiency bugs; each app has on average 6.5 service usage efficiency bugs. The empirical evaluation demonstrates that service usage inefficiencies are severe in practice, but the developers have not yet realized the severity of the issue.

In summary, the main contribution of this paper is as follows:

(1) According to the service lifecycle, we formulate four service usage anti-patterns that may lead to unnecessary resource occupation and energy consumption.
(2) Based on the anti-patterns, we present a static analysis approach to detecting the service usage efficiency bugs. To the best of our knowledge, our work is among the first to detect service usage inefficiencies in mobile apps.
(3) We develop an open-source prototype tool - ServDroid - which takes an APK file as input and outputs all service usage efficiency bugs and the information for debugging.
(4) We conduct an empirical study on 45 the most downloaded free Android apps, the results of which demonstrate that the service usage inefficiencies are pervasive in practice.

The rest of the paper is organized as follows. Section 2 gives an introduction to three types of app services. Section 3 formulates four anti-patterns that lead to service usage efficiency bugs. Section 4 presents our approach to detecting such bugs. Section 5 reports on our empirical study results. Section 6 positions our approach in the related work, and Section 7 concludes the paper.

2 BACKGROUND

Along with *activities* (user interfaces), *broadcast receivers* (mailboxes for broadcast), and *content providers* (local database servers), services (background tasks) are fundamental components in Android apps. To ease the understanding of service usage anti-patterns, we introduce the lifecycle of app services and how they are used [14].

[1] https://github.com/Sable/soot
[2] App List. https://en.wikipedia.org/wiki/List_of_most_downloaded_Android_applications (accessed in Nov 2017)

To define an app service, one should inherit the `Service` class provided by Android, and overwrite corresponding methods of `Service`, e.g., `onStartCommand(Intent, int, int)`, `onBind()`, etc. A service is started in the app by an asynchronous message object which referred to as *intent*. Accordingly, a service declared by a <service> tag in the AndroidManifest XML file of the app ought to have the attribute <intent-filter> which indicates the intents that can start it. The attribute <exported> indicates the components of other apps can invoke or interact with it. The attribute <isolated-Process> indicates whether the service is executed in an isolated process. Services can be used in three manners, corresponding to three types of app services: *started service*, *bound service*, and *hybrid service*.

Started service. The lifecycle of a started service is shown in Figure 1a, which is explained as follows. A started service is started via `Context.startService()` which triggers the system to retrieve the service or to create it via the `onCreate()` method of the service if the service has not been created before, and then to invoke the `onStartCommand(Intent, int, int)` method of the service. The service keeps running until `Context.stopService()` or the `stopSelf()` method of the service is invoked. It is worth mentioning that if the service is not stopped, multiple invocations to `Context.startService()` result in multiple corresponding invocations to `onStartCommand()`, but do not create more service instances, that is, the service (instance) is shared by different callers. Once `Context.stopService()` or `stopSelf()` is invoked, the service is stopped and destroyed by the system via calling the `onDestroy()` method of the service, no matter how many times it was started. However, if the `stopSelf(int)` method of the service is used, the service will not be stopped until all started intents are processed. Note that the started service and the components which start it are loosely-coupled, i.e., the service can still keep running when the components are destroyed.

Bound service. The lifecycle of a bound service is shown in Figure 1b, which is explained in the following. Since started services cannot interact with the components which start them, bound services are presented, which can send data to the launching components (clients). The client component can invoke `Context.bindService()` to obtain a connection to a service. Similarly, this creates the service by calling `onCreate()` without `onStartCommand()` if it has not been created yet. The client component receives the `IBinder` object (a client-server interface) which is returned by the `onBind(Intent)` method of the service, allowing the two to communicate. Although multiple client components can bind to the service, the system invokes `onBind()` only once. The binding is terminated either through the `Context.unbindService()` method (the system invokes `onUnbind()`), or the client component's lifecycle ends. A bound service is destroyed when no client component binds to it.

Hybrid service. A service can be both started and have connections bound to it. This kind of services is referred to as started and bound service, or hybrid service for short. The hybrid services can be started first and then bound, or vice versa. The components that start and bind a hybrid service can be different.

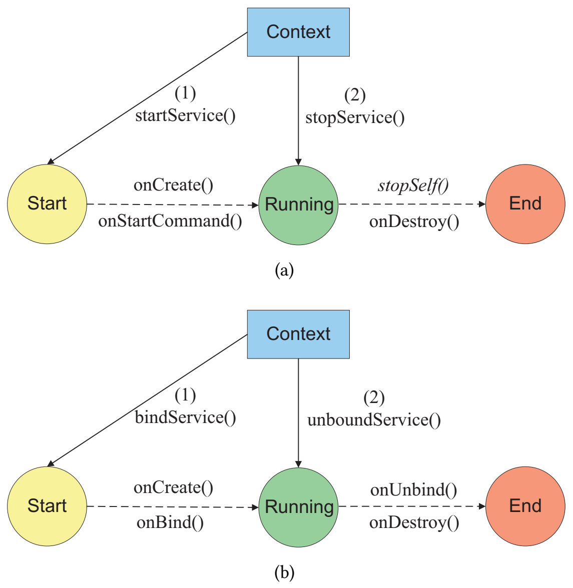

Figure 1: Service lifecycle of Android apps: (a) started service, (b) bound service.

3 SERVICE USAGE ANTI-PATTERNS

We first formulate four anti-patterns that lead to service usage inefficiencies. It is worth mentioning that these four kinds of service usage inefficiencies may occur to both local services (implemented by the app itself) and remote services (implemented by other apps).

ANTI-PATTERN 1 (**PREMATURE CREATE**). *A service is created too early before it is really used, and thus the service is in an idle state beginning from its create to its real use.*

Note that the premature create bugs can only exist in the hybrid services (i.e., started and bound services). Specifically, if `bindService()` is called after `startService()` but not immediately, and the `onStartCommand()` method of the service is not overwritten, the service will be in an idle state until `bindService()` is called. A service in an idle state occupies unnecessary resource (e.g., memory) and consumes unnecessary energy, which reduces the performance of the app.

ANTI-PATTERN 2 (**LATE DESTROY**). *A service is destroyed too late after its use, and thus the service is in an idle state beginning from the end of its final use to the moment it is destroyed.*

Let us explain why bugs of late destroy may exist. App services can be stopped (unbound) by end users via user-input events. Nevertheless, it only makes sense when end users want to stop (unbind) the services in advance. If end users want the services to complete the respective long-running tasks, they usually do not know the best moment to stop (unbind) the services. Besides, many services are non-interactive, i.e., they do not need user interaction at all [40]. Therefore, the right time to stop (unbind) a service should be determined by the app itself. More specifically,

(1) The right time to stop a started service is to call `stopSelf(int)` or `stopSelf()` at the end of its `onStartCommand()` method, because the end represents that the service's task is finished. Otherwise, the service is destroyed too late (stopped elsewhere), or not destroyed at all (cf. Anti-pattern 4).
(2) The right time to unbind a bound service is at the moment when the communication between the client component and the service is completed (i.e., the client component will not call the methods of the service any longer).
(3) The right time to stop and to unbind a hybrid service is the same as that to stop a started service and to unbind a bound service, respectively.

ANTI-PATTERN 3 (**PREMATURE DESTROY**). *Suppose that a service is used simultaneously by several components. The service is destroyed too early if one component destroys it before another component begins to use it, and thus it has to be recreated.*

Anti-pattern 3 can occur in started services and hybrid services. If there are several components that can start a service simultaneously, `stopSelf(int startID)` should be called in `onStartCommand()`. This guarantees that the service is not destroyed if the argument `startID` is not the same as that generated by the last start of the service. However, if `stopSelf()` is used instead, the created service may be destroyed too early before other components' use. Consequently, in this situation, the service should be created again to respond to other components. The premature destroy bugs lead to many unnecessary destroy and recreate of the same services, reducing the performance of the apps significantly.

Bound services are destroyed once they become unbounded. If a service becomes unbounded, it indicates that all client components which bound it have finished using it. In other word, currently, there is no other client component which is using it or ready to use it. Therefore, bound services are free of the premature destroy bugs by nature.

ANTI-PATTERN 4 (**SERVICE LEAK**). *A service is never destroyed after its use, even when the apps which use the service terminate.*

Anti-pattern 4 refers to the bugs that the services are never destroyed except for the situation that they are stopped (unbound) by end users. However, as aforementioned, the programmers should not rely on end users to stop (unbind) the app services, and, instead, the services ought to be stopped (unbound) by the app itself. If a service is started (bound) but is not stopped (unbound), the service is leaked. Since started services may be executed in an isolated process, the leaked service will keep running even when the app process terminates, which occupies unnecessary resources and reduces the performance of the app.

Despite the fact that the leaked services and the services destroyed too late (but not destroyed yet) can be automatically killed by the system when the system resource (e.g., memory) becomes low, the started services which was killed can be rebooted later, if the return value of their `onStartCommand()` method is "START_STICKY" or "START_REDELIVER_INTENT". In addition, since the leaked services occupy much memory, normal services may be unexpectedly killed by the system.

Table 1 summarizes the possible usage efficiency anti-patterns of different types of services.

Table 1: The correlation between service types and service usage efficiency anti-patterns

Anti-pattern \ Service type	Started	Bound	Hybrid
Premature create			√
Late destroy	√	√	√
Premature destroy	√		√
Service Leak	√	√	√

4 DETECTING SERVICE USAGE BUGS

Based on Anti-patterns 1-4, we use static analysis to detect service usage efficiency bugs in apps.

Since some services declared in the AndroidManifest XML file of the app are not implemented or used in the code, and our analysis should cover all usage of each service, we begin the static analysis by first obtaining the services that are actually implemented or used in the code. According to the methods of an implemented service, the service type is known. According to the statements that initiate services (i.e., `startService()` and `bindService()`) and the service types, we then determine whether there are use cases of the services that may lead to corresponding service usage inefficiencies based on the context-insensitive inter-procedural control flow graph of the app. Figure 2 illustrates the framework of our approach, which includes two major components: *Service Identifier* and *Bug Detector*. The former is responsible to identify from the code three list of services according to the service types. The latter is in charge of finding concrete service usage efficiency bugs for each service. Concrete methods of detecting premature create bugs, late destroy bugs, premature destroy bugs, and service leak bugs are elaborated on as follows.

4.1 Detecting Premature Create Bugs

The method of determining whether the use of a hybrid service s involves premature create bugs proceeds as follows. We first check whether the `onStartCommand()` method of the service s is overwritten. If it is overwritten, the service usage does not involve premature create bugs. Otherwise, we construct the context-insensitive inter-procedural control flow graph of the app. Based on the control flow graph, we check whether there is a path such that the following two conditions are satisfied (If both conditions are satisfied, the service s involves premature create bugs.):

(1) No component binds to service s when the `startService()` statement is executed.
(2) There is a `bindService()` statement which follows (but not immediately) the `startService()` statement.

To check the first condition, we first obtain the dominators (cf. Definition 1) of the `startService()` statement stm_s. If the list of dominators does not include a `bindService()` statement stm_b, the first condition is satisfied. Otherwise, we further check whether the corresponding `unbindService()` statement follows stm_b in the list, or the component that binds s is destroyed before stm_s (the corresponding `onDestroy()` statement follows stm_b in the list). If either is met, the first condition is satisfied.

For the second condition, we check whether there is a `bindService()` statement stm_{b_1} follows the `startService()` statement stm_s. If yes, we further determine in the path from stm_s to stm_{b_1} whether there is another `bindService()` statement that directly follows stm_s. If not, the second condition is satisfied.

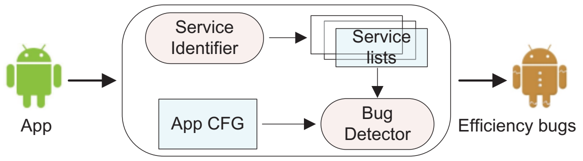

Figure 2: Approach framework of ServDroid.

Definition 1 (**Dominance and Dominator**). *In a control flow graph, a node (statement) s_j is dominated by another node (statement) s_i if each directed path from the entry of the control flow graph to s_j contains s_i. s_i is called a dominator of s_j.*

4.2 Detecting Late Destroy Bugs and Premature Destroy Bugs

The following method is used to detect late destroy bugs. The usage of a started service may result in late destroy bugs, if `stopSelf()` or `stopSelf(int)` is not called in the `onStartCommand()` method of the service. The usage of a bound service may contain late destroy bugs, if the client component does not call `unbindService()` immediately after the last invocation of the method m of the service in the control flow graph of the app (i.e., the `unbindService()` statement is not the direct successor of m). The usage of a hybrid service may lead to late destroy bugs, if `stopSelf()` (`stopSelf(int)`) is not called in `onStartCommand()`, or `unbindService()` is not called immediately after the last invocation of the method of the service in the control flow graph.

The use of a started or a hybrid service may involve premature destroy bugs, if the service is shared by two or more components (callers), and `stopSelf()` instead of `stopSelf(int)` is called in the `onStartCommand()` method of the service.

4.3 Detecting Service Leak Bugs

Normally, to bind a service, each bind statement should have a corresponding unbind statement such that the callers (client components) of the two statements are the same; however to start a service, multiple start statements may correspond to the same stop statement. If a start (bind) statement is not always followed by a stop (unbind) statement, the service may leak. As aforesaid, no matter whether end users can destroy a service or not, the app itself should have the mechanism to destroy the created service. With this in mind, we have the following steps to determine whether a service may leak or not:

(1) We first find all statements that start (bind) the service, i.e., `startService()` (`bindService()`). All the statements are summarized in a set S_1.
(2) We then find all statements that stop (unbind) the service, i.e., `stopService()` (`unbindService()`). All the statements are summarized in a set S_2.
(3) We next remove from S_2 the statements that are triggered by end users, i.e., the statements that are in the event handlers (callbacks) of the UI events (e.g., user click).

Table 2: Popular Android apps studied in our experiments

App name	**Developer**	**Category**	**Version**	**# Services**			
				Started	**Bound**	**Hybrid**	**Total**
Google Play services	Google	Tools	11.0.55 (436-156917137)	**130**	**17**	6	**153**
Gmail	Google	Communication	7.6.4.158567011.release	18	1	4	23
Maps	Google	Travel & Local	9.54.1	12	9	4	25
YouTube	Google	Video Players & Editors	12.23.60	10	2	3	15
Facebook	Facebook	Social	10.2.0	22	8	4	34
Google	Google	Tools	7.3.25.21.arm	21	10	6	37
Google+	Google	Social	9.14.0.158314320	23	1	4	28
GoogleText-to-Speech	Google	Tools	3.11.12	3	0	0	3
WhatsApp Messenger	Facebook	Communication	2.17.231	9	4	3	16
Google Play Books	Google	Books & Reference	3.13.17	8	2	1	11
Messenger	Facebook	Communication	123.0.0.11.70	40	1	0	41
Hangouts	Google	Communication	20.0.156935076	11	9	2	22
Google Chrome	Google	Communication	58.0.3029.83	16	9	2	27
Google Play Games	Google	Entertainment	3.9.08(3448271-036)	1	0	0	1
Google TalkBack	Google	Tools	5.2.0	2	1	0	3
Google Play Music	Google	Music & Audio	7.8.4818-1.R.4063206	15	3	4	22
Google Play Newsstand	Google	News & Magazines	4.5.0	8	1	0	9
Google Play Movies & TV	Google	Video Players & Editors	3.26.5	5	4	2	11
Google Drive	Google	Productivity	2.7.153.14.34	6	6	3	15
Samsung Push Service	Samsung	Communication	1.8.02	11	0	0	11
Instagram	Facebook	Social	10.26.0	14	3	2	19
Android System WebView	Google	Tools	58.0.3029.83	8	2	0	10
Google Photos	Google	Photography	2.16.0.157775819	15	6	6	27
Google Street View	Google	Travel & Local	2.0.0.157538376	3	3	2	8
Skype	Microsoft	Communication	8.0.0.44736	10	1	0	11
Clean Master	Cheetah Mobile	Tools	5.17.4	14	7	5	26
Subway Surfers	Kiloo Games	Games/Arcade	1.72.1	2	1	0	3
Dropbox	Dropbox,Inc	Productivity	50.2.2	8	3	1	12
Candy Crush Saga	King	Games/Casual	1.101.0.2	2	1	1	4
Twitter	Twitter	News & Magazines	7.0.0	9	2	2	13
LINE	LINE Corporation	Communication	7.5.2	11	5	3	19
HP Print Service Plugin	HP Inc	Productivity	3.4-2.3.0	4	3	2	9
Flipboard	Flipboard	News & Magazines	4.0.13	4	3	3	10
Samsung Print Service Plugin	Samsung	Productivity	3.02.170302	7	3	0	10
Super-Bright LED Flashlight	Surpax Technology Inc	Productivity	1.1.7	3	0	2	5
Gboard	Google	Tools	6.3.28.159021150	6	4	2	12
Cloud Print	Google	Productivity	1.36b	7	1	1	9
Snapchat	Snap Inc	Social	10.10.5.0	2	0	0	2
Pou	Zakeh	Games/Casual	1.4.73	4	0	0	4
Google Translate	Google	Tools	5.9.0.RC07.155715800	6	0	1	7
My Talking Tom	Outfit7	Games/Casual	4.2.1.50	17	6	2	25
Security Master	Cheetah Mobile	Tools	3.4.1	7	3	4	14
Facebook Lite	Facebook	Social	20.0.15	20	9	4	33
imo messenger	imo.im	Communication	11.3.2	16	6	**7**	29
Sum	-	-	-	578	167	99	844
Average	-	-	-	12.8	3.7	2.2	18.8

(4) For each start (bind) statement in S_1, i.e., `startService()` (`bindService()`), we check whether or not the corresponding stop (unbind) statement, i.e., `startService()` (`unbindService()`) exists in S_2. If not, the use of the service will lead to service leak. If yes, we further determine whether or not the stop (unbind) statement can be always reached from the corresponding start (bind) statement. If not, the use of the service will also lead to service leak.

In the fourth step above, it is challenging to determine whether a stop (unbind) statement stm_2 can be always reached from the corresponding start (bind) statement stm_1. Symbolic analysis can be used to precisely solve this problem, but it is difficult to scale. To balance precision and scalability, our method of determining service leak is based on the concept of *post-dominator* (cf. Definition 2): if stm_2 is the post-dominator of stm_1 (i.e., stm_1 is post-dominated by stm_2) in the context-insensitive inter-procedure control flow graph of the app, then the service does not leak. Otherwise, it may leak.

Definition 2 (**Post-dominance and Post-dominator**). *In a control flow graph, a node (statement) s_i is post-dominated by another node (statement) s_j if each directed path from s_i to the exit (return statement) of the control flow graph contains s_j. s_j is called a post-dominator of s_i.*

5 EMPIRICAL EVALUATION

In this section, we conduct an empirical study on service usage inefficiencies in real-world popular Android apps, aiming at answering the following five research questions:

- **RQ1** - *Service usage frequency*: Are background services widely used in Android apps? Which type of services is the most frequently-used?
- **RQ2** - *Pervasiveness of efficiency bugs*: Are service usage efficiency bugs common in practice?
- **RQ3** - *Distribution of efficiency bugs*: How are the four kinds of service usage efficiency bugs distributed in the three types of services?
- **RQ4** - *Dominating efficiency bugs*: Among the four kinds of service usage efficiency bugs, which kind is the most prevalent?
- **RQ5** - *Most vulnerable service type*: Among the three types of services, the usage of which type is more prone to efficiency bugs?

5.1 Experimental Setup

To facilitate our empirical analysis, we implement our approach in an open-source prototype tool ServDroid based on Soot [36]. The input of ServDroid is an app (APK file), and its output is the detected service usage efficiency bugs in the app. If the use of a service is detected to involve efficiency bugs, the corresponding callers of the service are also outputted, which facilitates debugging. We apply ServDroid) to 45 the most downloaded free Android apps according to Wikipedia. The first 25 apps are all with over one billion downloads and the rest 20 apps all have over 500 million downloads. Table 2 summarizes the basic information of these 45 apps, including the app names, developers, versions, and the number of services (smaller than that declared in the AndroidManifest file) implemented or used in the apps. It is worth mentioning that all the app versions are the latest in the early January, 2018.

Our experiment was performed on a computer with an Intel Core i7 3.6GHz CPU and 16 GB of memory, running Windows 8, JDK 1.7, and Android 7.0, 7.1.1, and 8.0. The total runtime overhead of our prototype tool ServDroid on analyzing the 45 apps is 4,293 seconds, with 95.4 seconds on average to analyze one app, which demonstrates the efficiency of ServDroid.

5.2 Experimental Results

The last columns of Table 2 summarize the numbers of different types of services used in the 45 Android apps. From these results, we have the following finding:

Table 3: Total number of service usage efficiency bugs

App name	**Service usage inefficiency bugs**				
	# PCBs	**# LDBs**	**# PDBs**	**# SLBs**	**Total**
Google Play services	2	3	0	**93**	**98**
Gmail	0	1	0	5	6
Maps	1	1	1	9	12
YouTube	0	1	0	3	4
Facebook	0	5	0	5	10
Google	0	3	0	1	4
Google+	0	1	0	7	8
GoogleText-to-Speech	0	0	0	2	2
WhatsApp Messenger	**4**	2	**3**	10	19
Google Play Books	0	0	0	1	1
Messenger	0	1	0	9	10
Hangouts	0	1	1	3	5
Google Chrome	0	0	0	0	0
Google Play Games	0	0	0	3	3
Google TalkBack	0	0	0	1	1
Google Play Music	0	2	2	10	14
Google Play Newsstand	0	1	1	3	5
Google Play Movies & TV	0	0	0	4	4
Google Drive	0	2	0	3	5
Samsung Push Service	0	3	0	1	4
Instagram	0	1	0	7	8
Android System WebView	0	0	0	0	0
Google Photos	1	2	1	4	8
Google Street View	0	5	0	6	11
Skype	0	0	0	0	0
Clean Master	0	**12**	2	25	39
Subway Surfers	0	0	0	1	1
Dropbox	1	1	1	4	7
Candy Crush Saga	0	1	0	0	1
Viber Messenger	1	2	0	5	8
Twitter	0	1	0	3	4
LINE	1	1	1	11	14
HP Print Service Plugin	0	1	0	4	5
Flipboard	0	1	0	2	3
Samsung Print Service Plugin	0	3	0	6	9
Super-Bright LED Flashlight	0	2	0	7	9
Gboard	0	0	0	1	1
Cloud Print	0	2	0	1	3
Snapchat	0	5	0	2	7
Pou	0	2	0	2	4
Google Translate	0	4	0	3	7
My Talking Tom	0	3	2	15	20
Security Master	2	1	2	7	12
Facebook Lite	1	9	2	7	19
imo messenger	2	4	1	14	21
Sum	16	90	20	310	436
Average	0.4	2.0	0.4	6.8	9.6

Finding 1: All 45 apps use background services. The total numbers (proportions) of started, bound, and hybrid services in these apps are 578 (68.48%), 167 (19.79%), and 99 (11.73%), respectively. Each app uses 18.8 services on average. The average numbers of started, bound, and hybrid services used in an app are 12.8, 3.7, and 2.2, respectively.
Implication: Services are widely used in Android apps. Started services are the most frequently-used type of services.

The reason why started services are the most-frequently used type of services lies in that they require less user interaction, and are thus particularly suitable for time-consuming tasks running on the background.

Table 3 summarizes the total number of detected service usage efficiency bugs in these apps. Table 4 reports on the numbers of premature create bugs (PCBs), late destroy bugs (LDBs), premature destroy bugs (PDBs), and service leak bugs (SLBs) with respect to the started, bound, and hybrid services in the 45 Android apps, respectively. In summary, we have the following findings:

Finding 2: Surprisingly, service usage efficiency bugs are common in the 45 Android apps. 42 (93.33%) of them are infected by at least one kind of efficiency bugs; 34 (75.56%) of them involve at least two kinds of efficiency bugs; 15 (33.33%) of them have no less than three kinds of efficiency bugs; and 8 (17.78%) of them are found to have all the four kinds of efficiency bugs.
Implication: Service usage efficiency bugs are common in real-world popular Android apps.

Finding 3: The numbers (proportions) of premature create bugs occurring on the started services, bound services, and hybrid services are 0 (0%), 0 (0%), and 16 (100%), respectively. The numbers (proportions) of late destroy bugs occurring on the started services, bound services, and hybrid services are 27 (30%), 48 (53.33%), and 15 (16.67%), respectively. The numbers (proportions) of premature destroy bugs occurring on the started services, bound services, and hybrid services are 14 (70%), 0 (0%), and 6 (30%), respectively. The numbers (proportions) of service leak bugs occurring on the started services, bound services, and hybrid services are 231 (74.52%), 32 (10.32%), and 47 (15.16%), respectively.
Implication: This confirms the conclusion in Table 1 that the premature create bugs only happen to the usage of hybrid services; premature destroy bugs do not occur on the usage of bound services; late destroy bugs and service leak bugs can happen to all the three types of services.

Finding 4: Among the 45 apps, 10 (22.22%) apps have premature create bugs, 35 (77.78%) apps are infected by late destroy bugs; 13 (28.89%) apps are found to have premature destroy bugs; and 41 (91.11%) apps involve service leak bugs.
Implication: Service leak bugs and premature create bugs are the most and least prevalent kinds of service usage inefficiencies, respectively.

Finding 5: The total numbers of premature create bugs, late destroy bugs, premature destroy bugs, and service leak bugs in the 45 apps are 16, 90, 20, and 310, respectively. The proportions of the four kinds of service usage efficiency bugs are 3.67%, 20.64%, 4.59%, and 71.10%, respectively. Each app involves 9.6 service usage efficiency bugs on average. The average numbers of premature create bugs, late destroy bugs, premature destroy bugs, and service leak bugs in an app are 0.4, 2.0, 0.4, and 6.8, respectively.
Implication: The number of service leak bugs is much larger than the total number of the other three kinds of service usage efficiency bugs. Service leak bugs are the dominant kind of service usage inefficiencies.

Compared to the other three kinds of service usage efficiency bugs, the negative effects of service leak bugs are the biggest, because service leak bugs lead to more unnecessary memory occupation and energy consumption. The conclusion of Findings 4 and 5 indicate that service leak bugs are the most severe kind of service usage inefficiencies in apps.

Finding 6: A total of 272 service usage efficiency bugs are relevant to the usage of 578 started services, 80 service usage efficiency bugs happen to the usage of 167 bound services, and 84 service usage efficiency bugs occur on the usage of 99 hybrid services.
Implication: Among the three types of services, the usage of hybrid services has the highest possibility to involve efficiency bugs, whereas the usage of started services and bound services has the lower possibilities to involve efficiency bugs.

We also applied ServDroid to other 60 real-world but not so popular Android apps, and found that those apps involve even more service usage efficiency bugs. In summary, Findings 1-6 of our empirical study indicate that service usage efficiency bugs are pervasive in Android apps, and the developers have not yet been aware of the severity of service usage efficiency bugs.

5.3 Case Studies

In this part, we reports on some service usage efficiency bugs detected from three Android apps.

YouTube is a well-known video-sharing platform all over the world. For its Android app of version 12.23.60, the number of started, bound, and hybrid services implemented or used in *YoutTube* is ten, two, and three, respectively. Our tool ServDroid finds totally four service usage efficiency bugs from this app. The code snippet in Figure 3a shows a late destroy bug found in this app, where the class `ahx` binds a service from a third party. The `bindService()` method is called in the `d()` method of `ahx`, but the corresponding `unbindService()` method is not immediately called after the last invocation of the method (i.e.,`post()`) of the service in `f()`. The service remains idle until the `e()` method of `ahx` is called. The code snippet in Figure 3b exhibits a service leak bug found from *YoutTube*: The `startService()` method that starts `AnalyticsService` is called in the `onReceive` method of `jth`, but no `stopSelf()` or `stopService()` exists to stop `AnalyticsService`.

WhatsApp Messenger is a popular mobile messaging app that allows people to exchange messages using the devices' Internet connections. For its version 2.17.231, the number of started, bound, and hybrid services implemented or used in *WhatsApp Messenger* is nine, four, and three, respectively. Although the service quantity is not large, 19 service usage efficiency bugs are found by our tool ServDroid, including four premature create bugs, two late destroy bugs, three premature destroy bugs, and 10 service leak bugs. The code snippet in Figure 4a reports a premature create bug in this app: The `onStartCommand()` method of the `GoogleDriveService` is not overwritten, and the `bindService` method is called after (but not immediately) the `startService` method. The code snippet in Figure 4b shows a late destroy bug, where `SearchActionVerificationClientService` binds a remote service from Google. The `bindService()` method is called in the `onCreate()` method of `SearchActionVerificationClientService`, but the corresponding `unbindService()` method is not immediately called after the last invocation of the method `isSearchAction()` of the Google

Table 4: The number of service usage efficiency bugs occurring to different types of services

App name	**Started service**				**Bound services**			**Hybrid services**				
	# LDBs	**# PDBs**	**# SLBs**	**Total**	**# LDBs**	**# SLBs**	**Total**	**# PCBs**	**# LDBs**	**# PDBs**	**# SLBs**	**Total**
Google Play services	1	0	**90**	**91**	2	0	2	2	0	0	3	5
Gmail	1	0	4	5	0	1	1	0	0	0	0	0
Maps	0	0	6	6	1	0	1	1	0	1	3	5
YouTube	0	0	3	3	1	0	1	0	0	0	0	0
Facebook	3	0	5	8	2	0	2	0	0	0	0	0
Google	1	0	0	1	2	1	3	0	0	0	0	0
Google+	0	0	7	7	1	0	1	0	0	0	0	0
Google Text-to-Speech	0	0	2	2	0	0	0	0	0	0	0	0
WhatsApp Messenger	0	**2**	2	4	2	0	2	**4**	0	1	**8**	**13**
Google Play Books	0	0	1	1	0	0	0	0	0	0	0	0
Messenger	1	0	9	10	0	0	0	0	0	0	0	0
Hangouts	0	1	0	1	1	1	2	0	0	0	2	2
Google Chrome	0	0	0	0	0	0	0	0	0	0	0	0
Google Play Games	0	0	3	3	0	0	0	0	0	0	0	0
Google TalkBack	0	0	1	1	0	0	0	0	0	0	0	0
Google Play Music	1	0	2	3	0	**3**	3	0	1	**2**	5	8
Google Play Newsstand	1	1	3	5	0	0	0	0	0	0	0	0
Google Play Movies & TV	0	0	4	4	0	0	0	0	0	0	0	0
Google Drive	0	0	3	3	2	0	2	0	0	0	0	0
Samsung Push Service	1	0	0	1	2	1	3	0	0	0	0	0
Instagram	1	0	3	4	0	**3**	3	0	0	0	1	1
Android System WebView	0	0	0	0	0	0	0	0	0	0	0	0
Google Photos	0	1	3	4	1	0	1	1	1	0	1	3
Google Street View	2	0	2	4	1	0	1	0	2	0	4	6
Skype	0	0	0	0	0	0	0	0	0	0	0	0
Clean Master	1	**2**	15	18	**8**	**3**	**11**	0	**3**	0	7	10
Subway Surfers	0	0	0	0	0	1	1	0	0	0	0	0
Dropbox	0	1	1	2	1	2	3	1	0	0	1	2
Candy Crush Saga	0	0	0	0	1	0	1	0	0	0	0	0
Viber Messenger	0	0	4	4	1	1	2	1	1	0	0	2
Twitter	1	0	2	3	0	1	1	0	0	0	0	0
LINE	0	1	6	7	1	2	3	1	0	0	3	4
HP Print Service Plugin	0	0	3	3	1	1	2	0	0	0	0	0
Flipboard	0	0	1	1	1	1	2	0	0	0	0	0
Samsung Print Service Plugin	0	0	5	5	3	1	4	0	0	0	0	0
Super-Bright LED Flashlight	0	0	5	5	1	1	2	0	1	0	1	2
Gboard	0	0	1	1	0	0	0	0	0	0	0	0
Cloud Print	0	0	1	1	2	0	2	0	0	0	0	0
Snapchat	2	0	2	4	2	0	2	0	1	0	0	1
Pou	2	0	2	4	0	0	0	0	0	0	0	0
Google Translate	1	0	2	3	1	1	2	0	2	0	0	2
My Talking Tom	1	1	11	13	2	2	4	0	0	1	2	3
Security Master	0	1	3	4	0	2	2	2	1	1	2	6
Facebook Lite	**4**	**2**	4	10	3	1	4	1	2	0	2	5
imo messenger	2	1	10	13	2	2	4	2	0	0	2	4
Sum	27	14	231	272	48	32	80	16	15	6	47	84
Average	0.6	0.3	5.1	6.0	1.1	0.7	1.8	0.4	0.3	0.1	1.0	1.8

service in onHandleIntent(). The Google service remains idle until the onDestroy() method of the SearchActionVerificationClientService is called. The code snippet in Figure 4c corresponds to a premature destroy bug: The startService() method of MessageService is called three times, but the stopSelf() (instead of stopSelf(int)) method is used in the onStartCommand() method of MessageService. The code snippet in Figure 4d exhibits a service leak bug: The startService() method that starts ExternalMediaManager is called in the onReceive method of ExternalMediaStateReceiver, but no stopSelf() or stopService() exists to stop ExternalMediaManager.

Twitter is an app designed to reduce busywork, which allows users to focus on the things that matter. For its version 7.0.0, the number of started, bound, and hybrid services implemented or used

```java
final class ahx extends agj implements ServiceConnection{
  final void d(){
    localIntent = new Intent("android.media.MediaRouteProviderService");
    this.o = this.a.bindService( localIntent , this, 1);
  }
  final void f(){
    localahy.h.j.post(new ahz(localahy));
  }
  final void e(){
    this.a.unbindService( this );
  }
}
```

(a)

```java
class jth extends BroadcastReceiver{
  public void onReceive(Context paramContext, Intent paramIntent){
    localIntent .setComponent(new ComponentName(localContext,
      "com.google.android.gms.analytics.AnalyticsService"));
    localContext. startService ( localIntent );
  }
}
```

(b)

Figure 3: Service usage efficiency bugs in *YouTube* (version 12.23.60): (a) A late destroy bug, (b) A service leak bug.

in *Twitter* is nine, two, and two, respectively. Although the number of services is not large, four service usage efficiency bugs are detected by our tool ServDroid, including one late destroy bug and three service leak bugs. The code snippet in Figure 5a reports the late destroy bug, where the `a()` method starts a service `OverlayService`. Despite the fact that the `stopService()` method is called in the `b()` method, the time to stop the `OverlayService` is too late. To address this problem, the `stopSelf()` method should be invoked in the `onStartCommand()` method of `OverlayService`. The code snippet in Figure 5b shows a service leak bug found from *Twitter*: The `startService()` method that starts the service `OemHelperService` is called in the `a()` method of the class `OemIntentReceiver`, but, surprisingly, there is no `stopSelf()` or `stopService()` available in the app code to stop `OemHelperService`.

Similar service usage efficiency bugs are found in other 42 Android apps used in our study. The reason why service usage efficiency bugs are so prevalent is that their effects are not so severe as those of the functional bugs, that is, they do not cause app crashes, and thus developers do not regard them as a tricky problem.

6 RELATED WORK

Our research is related to the work on GUI testing, service analysis and testing, and performance (energy) testing of Android apps. In the following, we review the state-of-the-art of these there aspects.

GUI testing. Most existing testing approaches for Android apps focus on GUI testing. According to the exploration strategies employed, Choudhary et al. [11] summarize three main categories of testing approaches: *random testing* [13, 18, 27, 34], *model-based testing* [3, 5, 10, 35, 39], and *advanced testing* [4, 22, 28, 30]. Although Monkey [13] is among the first generation techniques for Android testing, compared with many follow-up approaches, it still shows good performance and advantages in app testing [11]. Dynodroid [27] improves Monkey by reducing the possibility of

```java
public final class GoogleDriveActivity extends apf implements c.b, e.a{
  public final void onCreate(Bundle paramBundle){
    Intent localIntent = getIntent ();
    onNewIntent(localIntent);
    . . . . . .
    getApplicationContext (). bindService (new Intent( this ,
      GoogleDriveService. class ), this . af, 1);
  }
  protected final void onNewIntent(Intent paramIntent){
    m();
  }
  final void m(){
    Intent localIntent = new Intent( this , GoogleDriveService. class );
    getApplicationContext (). startService ( localIntent );
  }
}
```

(a)

```java
public abstract class SearchActionVerificationClientService extends Service{
  private final Intent mServiceIntent = new Inten("com.google.android.
    googlequicksearchbox.SEARCH_ACTION_VERIFICATION_SERVICE").
    setPackage("com.google.android.googlequicksearchbox");
  public final void onCreate(){
    bindService( this . mServiceIntent,
    this . mSearchActionVerificationServiceConnection , 1);
  }
  protected final void onHandleIntent(Intent paramIntent){
    if ((bool2) && (this.mIRemoteService.
      isSearchAction( localIntent , localBundle))){
        performAction( localIntent , bool1, localBundle);
        return;
    }
  }
  public final void onDestroy(){
    unbindService( this . mSearchActionVerificationServiceConnection );
  }
}
```

(b)

```java
public class MessageService extends Service{
  public void a(Context paramContext){
    Intent intent =new Intent(paramContext.this, MessageService. class );
    startService ( intent );
  }
  public void b(Context paramContext){
    Intent intent =new Intent(paramContext.this, MessageService. class );
    startService ( intent );
  }
  public void m(Context paramContext){
    Intent intent =new Intent(paramContext.this, MessageService. class );
    startService ( intent );
  }
  public int onStartCommand(Intent paramIntent,int paramInt1, int paramInt2){
    . . . . . .
    stopSelf ();
    return 1;
  }
}
```

(c)

```java
public static class ExternalMediaStateReceiver extends BroadcastReceiver{
  public void onReceive(Context paramContext, Intent paramIntent){
    paramContext. startService (paramIntent. setClass (paramContext,
      ExternalMediaManager.class));
  }
}
```

(d)

Figure 4: Service usage efficiency bugs in *WhatsApp Messenger* (version 2.17.231): (a) A premature create bug, (b) A late destroy bug, (c) A premature destroy bug, (d) A service leak bug.

```
public class OverlayService extends Service{
  public static void a(Context paramContext){
    paramContext.startService(new Intent(paramContext, OverlayService.class)
        );
  }
  public int onStartCommand(Intent paramIntent, int paramInt1, int paramInt2
        ){
    return;
  }
  public static void b(Context paramContext){
    paramContext.stopService(new Intent(paramContext, OverlayService.class))
        ;
  }
}
```

(a)

```
public class OemIntentReceiver extends BroadcastReceiver{
  public static void a(Context paramContext){
    localIntent.setClassName("com.twitter.twitteroemhelper",
      "com.twitter.twitteroemhelper.OemHelperService");
    paramContext.startService(localIntent);
  }
}
```

(b)

Figure 5: Service usage efficiency bugs in *Twitter* (version 7.0.0): (a) A late destroy bug, (b) A service leak bug.

generating redundant events. It achieves this by monitoring the reaction of an app upon each event and basing on the reaction to generate the next event. Recently, EHBDroid [34] is presented to first instrument the invocations of event callbacks in each activity and then directly trigger the callbacks in a random order. This approach is more efficient as it bypasses the GUI for test input generation. Since random testing may generate redundant events, several model-based testing approaches are proposed [2, 3, 5, 6, 10, 35, 39]. These approaches first obtain a model of the app GUI, and then generate test input according to the model. While most of them utilize program analysis techniques to obtain the model, machine learning is used in [10] to learn the model. The third category approaches leverage advanced techniques to efficiently generate effective event sequences for app testing[4, 22, 28, 30]. For example, ACTEve [4] uses symbolic execution and EvoDroid [28] employs evolutionary algorithm to generate event sequences. Sapienz [30] formulates the event sequence generation as a multiple-objective optimization problem and employs search-based algorithm to generate the shortest event sequences that can maximize the code coverage and bug exposure.

Service analysis and testing. A deal of work concentrates on the security vulnerabilities (e.g., denial of service, single point failure) of Android system services [1, 12, 19, 26, 33, 37]. In terms of app services, Khanmohammadi et al. find that malware may use background services to perform malicious operations with no communication with the other components of the app [23]. They propose to use classification algorithms to differentiate normal and malicious apps based on the service features related to their lifecycle. In contrast to GUI testing for activities, background service testing gain little attention. Snowdrop [40] is among the first to automatically and systematically testing background services in apps. Since not all *Intent* messages can be directly derived from the app bytecode, Snowdrop infers field values based on a heuristic that leverages the similarity in how developers name variables. This approach can find general bugs (functional bugs that lead to app crashes) in services, but may meet difficulties in detecting service usage efficiency bugs targeted in this paper.

Performance testing. Non-functional or performance bugs in apps are also important for user experience. Liu et al. [24] conduct an empirical study on 29 popular Android apps and find three types of performance bugs: GUI lagging, energy leak, and memory bloat. They also summarize common performance bug patterns (including lengthy operations in main threads, wasted computation for invisible GUI, and frequently invoked heavy-weight callbacks) and propose method to detect them. Since energy is a major concern in app performance and green software engineering [9, 17, 29, 32], energy bugs and the corresponding testing solutions draw increasing attention [7, 15, 20, 21, 25, 38]. It is worth mentioning that energy bugs are highly relevant to resource leak [7, 15, 25, 38]. Banerjee et al. present a testing framework to detect energy bugs and energy hotspots in apps based on the measurement of the power consumption through a power meter. Although the framework also considers service leak bugs, the test oracle based on the power consumption is expensive and time-consuming. To reduce the cost of the test, Jabbarvand et al. propose an approach to minimize the energy-aware test-suite [21]. Wu et al. present a static analysis approach to detecting GUI-related energy-drain bugs [38], whereas our approach aims to detect service usage efficiency bugs.

7 CONCLUSIONS

It is of great difficulty for testing techniques to reveal service usage efficiency bugs in Android apps, because such bugs do not cause apps to crash immediately. In this paper, we propose to use static analysis to address this problem. To this end, we first formulate four anti-patterns that lead to service usage efficiency bugs. Based on Soot, we present the approach to detecting all such bugs in Android apps. To the best of our knowledge, our work is among the first to detect service performance bugs using static analysis. Our approach is implemented as an open-source tool ServDroid, based on which, we conduct an empirical study. The empirical results demonstrate that service usage efficiency bugs are surprisingly pervasive in real-world popular apps, and the developers have not yet been aware of the severity of the bugs.

REFERENCES

[1] Huda Abualola, Hessa Alhawai, Maha Kadadha, Hadi Otrok, and Azzam Mourad. 2016. An Android-based Trojan Spyware to Study the NotificationListener Service Vulnerability. In *The 7th International Conference on Ambient Systems, Networks and Technologies (ANT'16) / The 6th International Conference on Sustainable Energy Information Technology (SEIT'16) / Affiliated Workshops, May 23-26, Madrid, Spain*. 465–471.

[2] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. 2011. A GUI Crawling-Based Technique for Android Mobile Application Testing. In *the Fourth IEEE International Conference on Software Testing, Verification and Validation, Berlin, Germany, 21-25 March, Workshop Proceedings*. 252–261.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7*. 258–261.

[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *20th ACM SIGSOFT Symposium*

on the Foundations of Software Engineering, FSE'12, Cary, NC, USA - November 11 - 16. 59.

[5] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'13, part of SPLASH'13, Indianapolis, IN, USA, October 26-31.* 641–660.

[6] Young Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE'16, Singapore, September 3-7.* 238–249.

[7] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE'14, Hong Kong, China, November 16 - 22.* 588–598.

[8] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Dexpler: converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP'12, Beijing, China, June 14.* 27–38.

[9] Coral Calero and Mario Piattini (Eds.). 2015. *Green in Software Engineering.* Springer.

[10] Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'13, part of SPLASH'13, Indianapolis, IN, USA, October 26-31.* 623–640.

[11] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE'15, Lincoln, NE, USA, November 9-13.* 429–440.

[12] Huan Feng and Kang G. Shin. 2016. BinderCracker: Assessing the Robustness of Android System Services. *CoRR* abs/1604.06964 (2016).

[13] Google. 2015. The Monkey UI Android testing tool. http://developer.android.com/tools/help/monkey.html. (2015).

[14] Google. 2017. Android Services. (2017). Retrieved February 1, 2018 from https://developer.android.com/guide/components/services.html

[15] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. 2013. Characterizing and detecting resource leaks in Android applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE'13, Silicon Valley, CA, USA, November 11-15.* 389–398.

[16] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. 2015. Tracking the Software Quality of Android Applications Along Their Evolution (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE'15, Lincoln, NE, USA, November 9-13.* 236–247.

[17] Mohammad Ashraful Hoque, Matti Siekkinen, Kashif Nizam Khan, Yu Xiao, and Sasu Tarkoma. 2016. Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices. *ACM Comput. Surv.* 48, 3 (2016), 39:1–39:40.

[18] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test, AST'11, Waikiki, Honolulu, HI, USA, May 23-24.* 77–83.

[19] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. 2015. From System Services Freezing to System Server Shutdown in Android: All You Need Is a Loop in an App. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6.* 1236–1247.

[20] Reyhaneh Jabbarvand and Sam Malek. 2017. μDroid: an energy-aware mutation testing framework for Android. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE'17, Paderborn, Germany, September 4-8.* 208–219.

[21] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2016. Energy-aware test-suite minimization for Android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA'016, Saarbrücken, Germany, July 18-20.* 425–436.

[22] Casper Svenning Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20.* 67–77.

[23] Kobra Khanmohammadi, Mohammad Reza Rejali, and Abdelwahab Hamou-Lhadj. 2015. Understanding the Service Life Cycle of Android Apps: An Exploratory Study. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM'15, Denver, Colorado, USA, October 12.* 81–86.

[24] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07.* 1013–1024.

[25] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Jian Lu. 2014. GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. *IEEE Trans. Software Eng.* 40, 9 (2014), 911–940.

[26] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. 2017. System Service Call-oriented Symbolic Execution of Android Framework with Applications to Vulnerability Discovery and Exploit Generation. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'17, Niagara Falls, NY, USA, June 19-23.* 225–238.

[27] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26.* 224–234.

[28] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE'14, Hong Kong, China, November 16 - 22.* 599–609.

[29] Irene Manotas, Christian Bird, Rui Zhang, David C. Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori L. Pollock, and James Clause. 2016. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering, ICSE'16, Austin, TX, USA, May 14-22.* 237–248.

[30] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA'16, Saarbrücken, Germany, July 18-20.* 94–105.

[31] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of Android applications. In *Proceedings of the 38th International Conference on Software Engineering, ICSE'16, Austin, TX, USA, May 14-22.* 559–570.

[32] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. 2016. What Do Programmers Know about Software Energy Consumption? *IEEE Software* 33, 3 (2016), 83–89.

[33] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin R. B. Butler, William Enck, and Patrick Traynor. 2016. *droid: Assessment and Evaluation of Android Application Analysis Tools. *ACM Comput. Surv.* 49, 3 (2016), 55:1–55:30.

[34] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: beyond GUI testing for Android applications. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE'17, Urbana, IL, USA, October 30 - November 03.* 27–37.

[35] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE'17, Paderborn, Germany, September 4-8.* 245–256.

[36] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, Mississauga, Ontario, Canada.* 13.

[37] Kai Wang, Yuqing Zhang, and Peng Liu. 2016. Call Me Back!: Attacks on System Server and System Apps in Android through Synchronous Callback. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28.* 92–103.

[38] Haowei Wu, Shengqian Yang, and Atanas Rountev. 2016. Static detection of energy defect patterns in Android applications. In *Proceedings of the 25th International Conference on Compiler Construction, CC'16, Barcelona, Spain, March 12-18.* 185–195.

[39] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE'13, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS'13, Rome, Italy, March 16-24. Proceedings.* 250–265.

[40] Li Lyna Zhang, Chieh-Jan Mike Liang, Yunxin Liu, and Enhong Chen. 2017. Systematically testing background services of mobile apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE'17, Urbana, IL, USA, October 30 - November 03.* 4–15.