ServDroid: Detecting Service Usage Inefficiencies in Android Applications

***Abstract*—Services in Android applications are frequently-used components that are responsible for performing time-consuming operations in the background. While services play a crucial role in app performance, our study finds that service usages in practice are not as efficient as expected, e.g., they tend to cause unnecessary resource occupation and energy consumption. In this paper, we present 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 develop a static analysis tool, ServDroid, which can automatically detect such bugs and also provide calling context to help debugging them. We apply ServDroid to 45 popular real-world Android apps. Surprisingly, we find that service usage inefficiencies are pervasive.**

***Index Terms*—Android app, background service, usage inefficiency, service leak, static analysis**

I. 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 the apps. Therefore, the quality of mobile apps receives an increasing attention [1]–[12].

Testing is an effective means to find bugs and to help improve the quality of apps. Since Android is popular and open-source, many testing approaches are presented for Android apps. Nonetheless, they mostly focus on GUI testing of foreground activities [2]–[4], [10], [11], [13]–[15], whereas background services have received few research attention [16], [17]. Services are Android components, performing long-running tasks that involve few or no user interactions, e.g., file I/O, music playing, network transaction [18]. 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. We focus on the latter which are further divided into three types: *started services*, *bound services*, and *hybrid services* according to how they are used in the code.

Although app services usually do not have user interface, they can keep running even when the device screen is shut down. 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) can be affected. In this paper, we focus on service usage inefficiencies (a type of non-functional bugs) that do not immediately lead to app crashes, but have a significant impact on the performance of the app. Since such bugs do not cause app crashes, existing testing methods do not work well in detecting them. Although there exists work on non-functional testing (mainly energy testing) of apps [5]–[7], [9], [19], it generally more expensive and more labour-intensive than functional testing [9], because its oracle is usually based on performance indicators.

To address this problem, instead of using testing, we first propose four anti-patterns to facilitate the analysis of service usage efficiency bugs. These anti-patterns are all defined based on the lifecycle of services [18] and are distilled from our manual analysis of real-world Android apps. The anti-patterns are described below:

- **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 situation that a service is never destroyed after its use, even when the apps which initiate the service terminate.

For each of the four service usage anti-patterns, we then propose a static analysis technique to automatically detect its instances (concrete efficiency bugs matching the anti-pattern) in an app. Our approach takes an app (APK file) as input and transforms the APK files into Jimple based on Soot [20]. It then constructs a context-insensitive inter-procedural control flow graph of the app and uses dominator analysis and successor analysis to find the service usage efficiency bugs. Finally, it reports all the detected service usage efficiency bugs as well as the components (callers) that initiate the services to facilitate debugging.

We implemented our approach in an open-source tool ServDroid. The tool is written in Java and is publicly available (links are omitted due to double-blind review). To investigate

the service usage efficiency bugs in real-world apps, we conduct an empirical analysis of 45 the most popular free Android apps listed on Wikipedia[1]. Our empirical study shows that service usage efficiency bugs are surprisingly pervasive in these real-world apps: 42 (93.33%) apps involve at least one kind of service usage efficiency bugs; each app has on average 9.8 service usage efficiency bugs. This indicates that service usage inefficiencies are severe in practice, but the developers have not yet realized their severity.

In summary, the key contributions of this paper are:

1) According to the service lifecycle, we formulate four service usage anti-patterns that may lead to unnecessary resource occupation and energy consumption.
2) We conduct an empirical study on 45 most downloaded free Android apps, the results of which demonstrate that the service usage inefficiencies are pervasive in practice and they indeed have a significant negative impact on the energy consumption.
3) Based on the anti-patterns, we present a static analysis approach and an open-source tool, ServDroid, to efficiently detect service usage efficiency bugs and also help debugging them. The precision and recall of ServDroid on the 45 apps are 97.5% and 97.28%, respectively.

The rest of the paper is organized as follows. Section II gives an introduction to the three types of app services. Section III formulates the four anti-patterns that lead to service usage efficiency bugs. Section IV presents our static analysis for detecting instances of such anti-patterns. Section V reports the results of our empirical study. Section VI reviews the related work, and Section VII concludes the paper.

II. 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 [18].

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 <isolatedProcess> 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*.

[1]https://en.wikipedia.org/wiki/List_of_most_downloaded_Android_applications (accessed in Jan 2018)

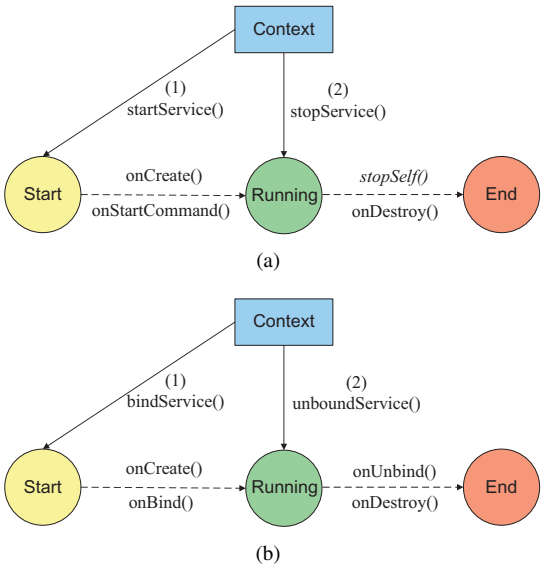

Fig. 1. App service lifecycle: (a) started service, (b) bound service.

Started service. The lifecycle of a started service is shown in Fig. 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 yet, 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 Fig. 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. Thus, 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.

III. 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).

*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.

*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 [16]. 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()` or `stopSelf(int)` 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.

*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.

*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_ST-ICKY" or "START_REDELIVER_INTENT". In addition, since the leaked services occupy much memory, normal services may be unexpectedly killed by the system. Note that Android 8.0 has taken actions on limiting background services: when the app is not in the foreground, the app's background services are stopped by the system. This may alleviate the impact of late destroy and service leak, but cannot avoid them. Moreover, this cannot reduce premature create and premature destroy.

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

IV. Detecting Service Usage Bugs

Based on the four anti-patterns, we use static analysis to detect service usage efficiency bugs in apps.

TABLE I
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	√	√	√

Since some services declared in the AndroidManifest XML file of the app may not be implemented or used in the code, and our analysis aims to discover all usages 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. Fig. 2 illustrates the framework of our approach, which includes two main components: *Service Identifier* and *Bug Detector*. The former is to identify from the code three list of services according to the service types. The latter is to find concrete service usage efficiency bugs for each service. We next present our analysis for detecting each of the four kinds of bugs in more detail.

A. 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.

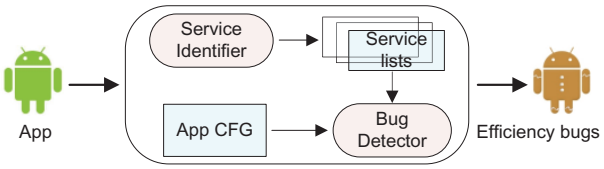

Fig. 2. Approach framework of ServDroid.

*Definition 1 (**Dominance and Dominator**):* In a control flow graph, a node (statement) s_j is dominated by another node 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.

B. Detecting Late Destroy 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 the statement which invokes 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.

C. 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 (only one) 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).
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 last step above, it is challenging to determine whether a stop (unbind) statement stm_2 can always be 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 a post-dominator of stm_1 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 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.

V. Empirical Evaluation

In this section, we study service usage inefficiencies in real-world popular Android apps, aiming at answering the following seven 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?
- **RQ6** - *Performance of ServDroid*: What are the precision, recall, and time overhead of ServDroid?
- **RQ7** - *Energy savings*: How much energy can be saved if these services usage efficiency bugs are fixed?

In the evaluation, we use both a manual inspection and a software tool (i.e., our prototype ServDroid) to analyze the service usage efficiency bugs in these apps.

TABLE II
Popular Android Apps Studied in the Empirical Study

App name	Version
Google Play services	11.0.55 (436-156917137)
Gmail	7.6.4.158567011.release
Maps	9.54.1
YouTube	12.23.60
Facebook	10.2.0
Google	7.3.25.21.arm
Google+	9.14.0.158314320
GoogleText-to-Speech	3.11.12
WhatsApp Messenger	2.17.231
Google Play Books	3.13.17
Messenger	123.0.0.11.70
Hangouts	20.0.156935076
Google Chrome	58.0.3029.83
Google Play Games	3.9.08(3448271-036)
Google TalkBack	5.2.0
Google Play Music	7.8.4818-1.R.4063206
Google Play Newsstand	4.5.0
Google Play Movies & TV	3.26.5
Google Drive	2.7.153.14.34
Samsung Push Service	1.8.02
Instagram	10.26.0
Android System WebView	58.0.3029.83
Google Photos	2.16.0.157775819
Google Street View	2.0.0.157538376
Skype	8.0.0.44736
Clean Master	5.17.4
Subway Surfers	1.72.1
Dropbox	50.2.2
Candy Crush Saga	1.101.0.2
Viber Messenger	8.0.0.3
Twitter	7.0.0
LINE	7.5.2
HP Print Service Plugin	3.4-2.3.0
Flipboard	4.0.13
Samsung Print Service Plugin	3.02.170302
Super-Bright LED Flashlight	1.1.7
Gboard	6.3.28.159021150
Cloud Print	1.36b
Snapchat	10.10.5.0
Pou	1.4.73
Google Translate	5.9.0.RC07.155715800
My Talking Tom	4.2.1.50
Security Master	3.4.1
Facebook Lite	20.0.15
imo messenger	11.3.2

A. Experimental Setup

Data set. Our empirical evaluation is based on 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 II summarizes the basic information of these 45 apps, including the app names and their versions. It is worth mentioning that all the app versions are the latest in the early January, 2018.

Oracle. We conduct an empirical study on these 45 apps by manual inspection. Specifically, we decompile the apps into Java source code, and ask three graduate students to manually and independently inspect the source code to identify all service efficiency bugs. The inconsistencies among them are solved through double check and coordination. We use the code inspection results as the oracle (ground truth) to evaluate the effectiveness of our approach.

TABLE III
TOTAL NUMBER OF SERVICES AND SERVICE USAGE EFFICIENCY BUGS

App name	# Services				# Service usage inefficiency bugs						
	Started	**Bound**	**Hybrid**	**Total**	**PCBs**	**LDBs**	**PDBs**	**SLBs**	**Total**	**FPs**	**FNs**
Google Play services	**130**	**17**	6	**153**	2	3	0	**95**	**100**	0	1
Gmail	18	2	4	24	0	1	0	5	6	0	0
Maps	12	9	4	25	1	1	1	9	12	0	0
YouTube	10	2	3	15	0	1	0	3	4	0	1
Facebook	22	8	4	34	0	5	0	5	10	1	0
Google	21	10	6	37	0	3	0	1	4	0	0
Google+	23	1	4	28	0	1	0	7	8	0	0
GoogleText-to-Speech	4	1	0	5	0	0	0	2	2	0	0
WhatsApp Messenger	9	4	3	16	**4**	2	**2**	10	18	1	0
Google Play Books	8	2	1	11	0	0	0	1	1	0	0
Messenger	40	1	0	41	0	1	0	9	10	0	0
Hangouts	11	9	2	22	0	1	1	3	5	0	1
Google Chrome	16	9	2	27	0	0	0	0	0	0	0
Google Play Games	1	0	0	1	0	0	0	3	3	0	0
Google TalkBack	2	1	0	3	0	0	0	1	1	0	0
Google Play Music	15	3	4	22	0	2	2	10	14	1	1
Google Play Newsstand	8	1	1	10	0	1	1	3	5	0	0
Google Play Movies & TV	6	4	2	12	0	0	0	4	4	0	0
Google Drive	6	6	3	15	0	2	0	3	5	0	0
Samsung Push Service	11	0	0	11	0	3	0	1	4	0	0
Instagram	14	4	2	20	0	1	0	7	8	1	1
Android System WebView	8	2	1	11	0	0	0	0	0	0	0
Google Photos	17	8	6	31	1	2	1	4	8	0	0
Google Street View	3	3	2	8	0	5	0	7	12	1	1
Skype	10	1	0	11	0	0	0	0	0	0	0
Clean Master	18	9	5	32	0	**12**	2	25	39	3	1
Subway Surfers	2	1	0	3	0	0	0	1	1	0	0
Dropbox	8	3	1	12	1	1	1	4	7	0	0
Candy Crush Saga	2	1	1	4	0	1	0	0	1	0	0
Viber Messenger	8	6	3	17	1	2	0	5	8	0	1
Twitter	9	2	2	13	0	1	0	3	4	0	0
LINE	11	5	3	19	1	1	1	12	15	0	1
HP Print Service Plugin	4	3	2	9	0	1	0	5	6	0	0
Flipboard	4	3	3	10	0	1	0	2	3	0	1
Samsung Print Service Plugin	7	3	0	10	0	3	0	6	9	1	0
Super-Bright LED Flashlight	3	0	2	5	0	2	0	7	9	0	0
Gboard	6	4	2	12	0	0	0	1	1	0	1
Cloud Print	8	1	1	10	0	2	0	1	3	0	0
Snapchat	2	1	0	3	0	5	0	2	7	0	1
Pou	4	0	0	4	0	2	0	2	4	0	0
Google Translate	6	0	1	7	0	4	0	3	7	0	0
My Talking Tom	17	6	2	25	0	3	2	13	18	1	0
Security Master	7	3	4	14	3	2	2	7	14	0	0
Facebook Lite	20	9	4	33	1	9	2	7	19	1	0
imo messenger	16	6	**7**	29	2	4	1	14	21	0	0
Sum	596	181	104	881	17	91	19	313	440	11	12
Average	13.2	4.0	2.3	19.5	0.4	2.0	0.4	7.0	9.8	0.24	0.27

Implementation. We implement our approach in an open-source prototype tool ServDroid based on Soot [20]. The input of ServDroid is an app (APK file), and its output is the detected service usage efficiency bugs in the app. ServDroid also returns the calling context to help developers debug these inefficiencies. We also use ServDroid to detect the service usage efficiency bugs in the 45 apps, and compare the detected results with those of the manual inspection.

B. Experimental Results

In this subsection, we report on the results and findings of our empirical evaluation. The second column of Table III summarizes the numbers of different types of services used in the 45 Android apps.

Answer to RQ1: All 45 apps use background services. The total numbers (proportions) of started, bound, and hybrid services in these apps are 596 (67.65%), 181 (20.55%), and 104 (11.80%), respectively. Each app uses 19.5 services on average. The average numbers of started, bound, and hybrid services used in an app are 13.2, 4.0, and 2.3, respectively.
Implication: Services are widely used in Android apps, and 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.

The first five sub-columns of the third column of Table III summarize the numbers of premature create bugs (PCBs),

late destroy bugs (LDBs), premature destroy bugs (PDBs), service leak bugs (SLBs), and all service usage efficiency bugs detected by ServDroid from the 45 apps. According to the code inspection results, the numbers of false positives (FPs) and false negatives (FNs) of ServDroid are also listed in the last two sub-columns, respectively. According to the code inspection results, we have the following findings:

Answer to RQ2: 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; 16 (35.56%) 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. Each app has 9.8 bugs on average.
Implication: Service usage efficiency bugs are common in real-world popular Android apps.

Answer to RQ3: The numbers (proportions) of premature create bugs occurring on the started services, bound services, and hybrid services are 0 (0%), 0 (0%), and 17 (100%), respectively. The numbers (proportions) of late destroy bugs occurring on the started services, bound services, and hybrid services are 27 (30.34%), 46 (51.68%), and 16 (17.98%), respectively. The numbers (proportions) of premature destroy bugs occurring on the started services, bound services, and hybrid services are 14 (73.68%), 0 (0%), and 5 (26.32%), respectively. The numbers (proportions) of service leak bugs occurring on the started services, bound services, and hybrid services are 237 (75.0%), 32 (10.13%), and 47 (14.87%), respectively.
Implication: This confirms the conclusion in Table I 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.

Answer to RQ4: The total numbers of premature create bugs, late destroy bugs, premature destroy bugs, and service leak bugs in the 45 apps are 17, 89, 19, and 316, respectively. The proportions of the four kinds of service usage efficiency bugs are 3.85%, 20.18%, 4.31%, and 71.66%, respectively. 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 7.0, 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.

Among the 45 apps, 10 (22.22%) apps have premature create bugs, 35 (77.78%) apps are infected by late destroy bugs; 14 (31.11%) apps are found to have premature destroy bugs; and 41 (91.11%) apps involve service leak bugs. Service leak bugs and premature create bugs are the most and least prevalent kinds of service usage inefficiencies, respectively.

Answer to RQ5: 278 service usage efficiency bugs are relevant to the usage of 596 started services, 78 service usage efficiency bugs happen to the usage of 181 bound services, and 85 service usage efficiency bugs occur on the usage of 104 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.

The 45 apps are analyzed with ServDroid 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. Taking the manual inspection results as ground truth, the accuracy of ServDroid is reported as follows.

Answer to RQ6: The overall precision and recall of the results returned by ServDroid are 97.5% and 97.28%, respectively. The total runtime overhead of ServDroid on analyzing the 45 apps is 4,319 seconds, with 96 seconds on average to analyze one app.
Implication: The lightweight analysis of ServDroid has a high accuracy, and is scalable.

To investigate the energy impact of the efficiency bugs, we first use Soot to instrument the infected apps to fix the efficiency bugs, and repackage them. Then, we run the original and repackaged apps independently on a phone with the same setting (OPPO R9m with a 2.0GHz 8-core CPU and 4GB of memory, running Android 8.0) for 15 minutes, and use the tool (also an app) Trepn Profiler to measure the battery cost of them, respectively. The results are summarized in Table IV, where seven apps are not measured because they are either free of the efficiency bugs or cannot run independently. Based on the results in Table IV, we have the following finding:

Answer to RQ7: The average energy consumptions (15 minutes) of an app before and after the service usage efficiency bugs are fixed are 579.32 mW and 475.14 mW, respectively. That is, an app can save on average 104.18 mW in 15 minutes if the bugs are fixed.
Implication: The energy consumption is reduced after the bugs are fixed, and thus the efficiency bugs have a significant impact on energy consumption.

We have also applied ServDroid to different versions of some apps, and find that, the numbers of service usage efficiency bugs decrease in newer versions (see Fig. 3, where concrete version numbers are abstracted away for simplicity.). This implies that some developers fixed such bugs, though not completely. In summary, our empirical study indicate that service usage efficiency bugs are pervasive in Android apps, but most developers have not yet been aware of the severity of the problem.

C. Case Studies

In this subsection, we elaborate on some service usage efficiency bugs in three popular Android apps.

TABLE IV
ENERGY CONSUMPTION BEFORE AND AFTER THE BUGS ARE FIXED

App name	Energy consumption	
	Original (mW)	Repaired (mW)
Google Play services	586.29	383.6
Gmail	473.39	420.94
Maps	670.91	503.92
YouTube	859.51	747.76
Facebook	605.51	557.39
Google	506.82	446.85
Google+	482.57	470.16
GoogleText-to-Speech	435.06	384.02
WhatsApp Messenger	451.96	357.27
Google Play Books	521.09	432.18
Messenger	580.37	473.29
Hangouts	396.16	362.58
Google Play Games	613.26	491.57
Google TalkBack	529.34	402.96
Google Play Music	502.18	421.69
Google Play Newsstand	506.2	457.06
Google Play Movies & TV	381.62	349.27
Google Drive	375.36	312.28
Instagram	513.61	500.39
Google Photos	481.95	345.28
Google Street View	452.96	390.17
Clean Master	498.26	432.87
Subway Surfers	1037.81	903.29
Dropbox	486.87	351.27
Candy Crush Saga	705.18	403.92
Viber Messenger	497.31	409.31
Twitter	651.92	610.47
LINE	479.61	357.19
Flipboard	529.27	470.62
Super-Bright LED Flashlight	498.68	426.83
Gboard	538.92	460.39
Snapchat	652.21	503.8
Pou	1106.85	672.31
Google Translate	436.75	365.75
My Talking Tom	1637.91	1457.3
Security Master	481.37	400.72
Facebook Lite	371.06	313.28
imo messenger	478.15	319.75
Sum	22014.25	18055.50
Average	579.32	475.14

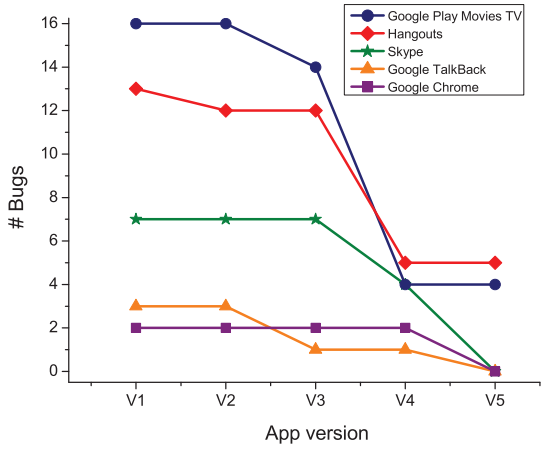

Fig. 3. The number of services usage efficiency bugs decreases in newer versions.

```
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)

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

(b)

Fig. 4. Service usage efficiency bugs in *YouTube* (version 12.23.60): (a) A late destroy bug, (b) A service leak bug.

YouTube is a well-known video-sharing platform all over the world. Here, we show two service usage efficiency bugs in its Android app of version 12.23.60. The code snippet in Fig. 4a presents 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 called immediately 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 Fig. 4b 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, several service usage efficiency bugs are found. The code snippet in Fig. 5a 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 Fig. 5b 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 called immediately after the last invocation of the method `isSearchAction()` of the Google service in `onHandleIntent()`. The Google

```
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)

```
public abstract class SearchActionVerificationClientService extends
  Service{
    private final Intent mServiceIntent = new Intent("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);
      }
    }
    public final void onDestroy(){
      unbindService(this.mSearchActionVerificationServiceConnection);
    }
  }
```

(b)

```
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)

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

(d)

Fig. 5. 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)

Fig. 6. Service usage efficiency bugs in *Twitter* (version 7.0.0): (a) A late destroy bug, (b) A service leak bug.

service remains idle until the `onDestroy()` method of the `SearchActionVerificationClientService` is called. The code snippet in Fig. 5c 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 Fig. 5d 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, there are four service usage efficiency bugs, including one late destroy bug and three service leak bugs. The code snippet in Fig. 6a 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 Fig. 6b shows a service leak bug: 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 studied in our empirical evaluation. 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.

VI. 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 three aspects.

GUI testing. Most existing testing approaches for Android apps focus on GUI testing. According to the exploration strategies employed, Choudhary et al. [21] summarize three main categories of testing approaches: *random testing* [3], [13], [15], [22], *model-based testing* [4], [11], [23]–[25], and *advanced testing* [2], [26]–[28]. 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 [21]. Dynodroid [3] improves Monkey by reducing the possibility of 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 [15] 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 [4], [11], [14], [23]–[25], [29]. 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 [4] to learn the model. The third category approaches leverage advanced techniques to efficiently generate effective event sequences for app testing [2], [26]–[28]. For example, ACTEve [2] uses symbolic execution and EvoDroid [27] employs evolutionary algorithm to generate event sequences. Sapienz [28] 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], [30]–[34]. 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 [35]. 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 [16] 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. A dynamic analysis tool LESDroid [17] is presented to find exported service leaks, whereas ServDroid is a static analysis tool and is more general, which can not only find both private and exported service leaks but also detect the other three kinds of service usage efficiency bugs.

Performance testing. Non-functional or performance bugs in apps are also important for user experience. Liu et al. [5] 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 [36]–[39], energy bugs and the corresponding testing solutions draw increasing attention [6], [7], [9], [19], [40], [41]. It is worth mentioning that energy bugs are highly relevant to resource leak [6], [7], [40], [41]. 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 [9]. Wu et al. present a static analysis approach to detecting GUI-related energy-drain bugs [41], whereas our approach aims to detect service usage efficiency bugs.

VII. 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. We implement our approach in an open-source tool ServDroid and conduct an empirical evaluation on 45 real-world Android apps. The empirical results demonstrate that service usage efficiency bugs are severe in practice and they have a significant negative impact on the energy consumption.

References

[1] B. Reaves, J. Bowers, S. A. G. III, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife, B. Wright, K. R. B. Butler, W. Enck, and P. Traynor, "*droid: Assessment and evaluation of Android application analysis tools," *ACM Comput. Surv.*, vol. 49, no. 3, pp. 55:1–55:30, 2016.

[2] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "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*, 2012, pp. 59:1–59:11.

[3] A. Machiry, R. Tahiliani, and M. Naik, "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*, 2013, pp. 224–234.

[4] W. Choi, G. C. Necula, and K. Sen, "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*, 2013, pp. 623–640.

[5] Y. Liu, C. Xu, and S. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07*, 2014, pp. 1013–1024.

[6] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "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*, 2014, pp. 588–598.

[7] Y. Liu, C. Xu, S. Cheung, and J. Lu, "Greendroid: Automated diagnosis of energy inefficiency for smartphone applications," *IEEE Trans. Software Eng.*, vol. 40, no. 9, pp. 911–940, 2014.

[8] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "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*, 2015, pp. 236–247.

[9] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "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*, 2016, pp. 425–436.

[10] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "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*, 2016, pp. 559–570.

[11] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "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*, 2017, pp. 245–256.

[12] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *Proceedings of the 40th International Conference on Software Engineering, ICSE'18, Gothenburg, Sweden, May 27 - June 03*, 2018, pp. 408–419.

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

[14] Y. M. Baek and D. Bae, "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*, 2016, pp. 238–249.

[15] W. Song, X. Qian, and J. Huang, "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*, 2017, pp. 27–37.

[16] L. L. Zhang, C. M. Liang, Y. Liu, and E. Chen, "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*, 2017, pp. 4–15.

[17] J. Ma, S. Liu, Y. Jiang, X. Tao, C. Xu, and J. Lu, "Lesdroid: a tool for detecting exported service leaks of android applications," in *Proceedings of the 26th Conference on Program Comprehension, ICPC '18, Gothenburg, Sweden, May 27-28*, 2018, pp. 244–254.

[18] Google. (2017) Android services. [Online]. Available: https://developer.android.com/guide/components/services.html

[19] R. Jabbarvand and S. Malek, "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*, 2017, pp. 208–219.

[20] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, "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*, 1999, p. 13.

[21] S. R. Choudhary, A. Gorla, and A. Orso, "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*, 2015, pp. 429–440.

[22] C. Hu and I. Neamtiu, "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*, 2011, pp. 77–83.

[23] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon, "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*, 2012, pp. 258–261.

[24] W. Yang, M. R. Prasad, and T. Xie, "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*, 2013, pp. 250–265.

[25] T. Azim and I. Neamtiu, "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*, 2013, pp. 641–660.

[26] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20*, 2013, pp. 67–77.

[27] R. Mahmood, N. Mirzaei, and S. Malek, "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*, 2014, pp. 599–609.

[28] K. Mao, M. Harman, and Y. Jia, "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*, 2016, pp. 94–105.

[29] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "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*, 2011, pp. 252–261.

[30] H. Huang, S. Zhu, K. Chen, and P. Liu, "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*, 2015, pp. 1236–1247.

[31] K. Wang, Y. Zhang, and P. Liu, "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*, 2016, pp. 92–103.

[32] H. Feng and K. G. Shin, "Bindercracker: Assessing the robustness of android system services," *CoRR*, vol. abs/1604.06964, 2016.

[33] H. Abualola, H. Alhawai, M. Kadadha, H. Otrok, and A. Mourad, "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*, 2016, pp. 465–471.

[34] L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, and P. Liu, "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*, 2017, pp. 225–238.

[35] K. Khanmohammadi, M. R. Rejali, and A. Hamou-Lhadj, "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*, 2015, pp. 81–86.

[36] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?" *IEEE Software*, vol. 33, no. 3, pp. 83–89, 2016.

[37] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, and S. Tarkoma, "Modeling, profiling, and debugging the energy consumption of mobile devices," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 39:1–39:40, 2016.

[38] I. Manotas, C. Bird, R. Zhang, D. C. Shepherd, C. Jaspan, C. Sadowski, L. L. Pollock, and J. Clause, "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*, 2016, pp. 237–248.

[39] C. Calero and M. Piattini, Eds., *Green in Software Engineering*. Springer, 2015.

[40] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "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*, 2013, pp. 389–398.

[41] H. Wu, S. Yang, and A. Rountev, "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*, 2016, pp. 185–195.