ServDroid: Detecting Service Usage Inefficiencies in Android Applications

Wei Song  
School of Computer Sci. & Eng.  
Nanjing University of Sci. & Tech.  
Nanjing, China  
wsong@njust.edu.cn

Jing Zhang  
School of Computer Sci. & Eng.  
Nanjing University of Sci.& Tech.  
Nanjing, China  
jing8017@gmail.com

Jeff Huang  
Parasol Laboratory  
Texas A&M University  
College Station, TX, USA  
jeff@cse.tamu.edu

ABSTRACT
Services in Android applications are frequently-used components for performing time-consuming operations in the background. While services play a crucial role in the app performance, our study shows that service uses in practice are not as efficient as expected, e.g., they tend to cause unnecessary resource occupation and/or energy consumption. Moreover, as service usage inefficiencies do not manifest with immediate failures, e.g., app crashes, existing testing-based approaches fall short in finding them. In this paper, we identify four anti-patterns of such service usage inefficiency bugs, including premature create, late destroy, premature destroy, and service leak, and present a static analysis technique, ServDroid, to automatically and effectively detect them based on the anti-patterns. We have applied ServDroid to a large collection of popular real-world Android apps. Our results show that, surprisingly, service usage inefficiencies are prevalent and can severely impact the app performance.

CCS CONCEPTS
- Software and its engineering → Software defect analysis.

KEYWORDS
Android app, service usage inefficiency, static analysis

1 INTRODUCTION
Mobile is eating the world. The quality of mobile apps has received an increasing attention in the research community [4, 8, 11, 13, 19, 24, 28, 29, 32, 36, 40, 42]. While GUI testing is an effective means to find acceptance bugs, most existing techniques for mobile apps have focused on foreground activities [4, 6, 11, 15, 32, 36, 41, 42], whereas background services have received little research attention [31, 48].

This paper focuses on studying Android app services, in particular their usage inefficiencies. Since such bugs are often non-functional and do not cause immediate app failures, existing testing methods fall short in detecting them.

In Android, services are components that perform long-running tasks involving few or no user interactions, e.g., file I/O, music playing, or network transactions [17]. A service usually executes on the main thread of the app 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. According to how they are used in the code, app services can be further divided into three types: started services, bound services, and hybrid services.

Although app services usually do not have a 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 and energy consumption) can be affected. Although there exists work on non-functional testing (mainly energy testing) of apps [8, 23, 24, 28, 29], non-functional testing is generally more expensive (and also labor-intensive) than functional testing [24], because its oracle relies on performance indicators.

To address this problem, instead of relying on testing, we propose four anti-patterns of service usage (distilled from our manual analysis of real-world Android apps), and develop an automated static analysis to detect service inefficiency bugs based on these anti-patterns. These anti-patterns are all defined based on the lifecycle of app services [17], and are summarized below:

- **Premature create** refers to the situation that a service is created too early before it is used. Hence, the service is in an idle state beginning from its creation 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 fulfill the new request.
- **Service leak** refers to the situation that a service is never destroyed after its use, even when the app which initiated the service terminates. While these four anti-patterns are by no means complete (i.e., there may exist other service usage inefficiencies in theory), we...
have not found any exception in our empirical study. For each of the four service usage anti-patterns, we develop a scalable static analysis to automatically detect its instances (concrete inefficiency bugs matching the anti-pattern) in apps. A main challenge we address in our static analysis is scalability. Instead of performing a globally context and path sensitive analysis, we start from a context-insensitive control flow analysis and leverage the anti-patterns to guide a relatively precise inter-procedural dominator analysis. For each service, our approach examines its uses along all potential paths, and all uses of all services are examined statically. With that, our approach not only detects service usage inefficiency bugs but also locates the components (callers) that initiate the services to facilitate debugging.

We implemented our approach in an open-source tool ServDroid based on Soot [43]. ServDroid is written in Java and is publicly available on GitHub\(^1\). To investigate service usage in real-world apps, we conducted an empirical analysis on 1,000 Android apps downloaded from Google Play (accessed in Dec 2018) by applying ServDroid on them. Our results indicate that service usage inefficiency bugs are pervasive in real-world apps: 825 (82.5%) apps involve at least one type of service usage inefficiency bugs; each app has on average 4.43 service usage inefficiency bugs. Moreover, our performance measurements with Trepn Profiler\(^2\) on 38 apps show that by fixing these bugs an app can save on average 87.14 Joule of battery in 15 minutes (as much as 15.94% energy reduction).

In summary, the key contributions of this paper are:

1. We present four novel service usage anti-patterns that may lead to unnecessary resource occupation and energy consumption in Android apps.
2. We present ServDroid, a static analysis approach and an open-source tool to effectively detect service usage inefficiency bugs based on these anti-patterns. The precision and recall of ServDroid on the top 45 most popular free Android apps are very high based on our manual inspection.
3. We present an empirical study on 1,000 real-world Android apps. The results indicate that service usage inefficiencies are severe in practice and have a significant negative impact on the app’s energy consumption.

The rest of the paper is organized as follows. Section 2 gives an introduction to the Android app services. Section 3 formulates the four anti-patterns that lead to service usage inefficiency bugs. Section 4 presents our static analysis for detecting instances of such anti-patterns. Section 5 reports on the results of our empirical study on real-world apps. Section 6 reviews 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 [17].

An app service can be defined by extending the Android Service class and overwriting the 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 is referred to as `intent`. Accordingly, a service declared by a `<service>` tag in the AndroidManifest XML file of the app must have the `<intent-filter>` attribute, which indicates the intents that can start it. The attribute `<exported>` indicates that components from other apps can invoke or interact with the service. 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`.

#### 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(Intent) 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 after the components are destroyed.

---

---

\(^1\)https://github.com/wsong-nj/ServDroid

\(^2\)https://developer.qualcomm.com/software/trepn-power-profiler

---

**Figure 1**: App service lifecycle: (a) started service, (b) bound service.
Bound service. The lifecycle of a bound service is shown in Figure 1b. As started services cannot interact with the components which start them, bound services are proposed, which can send data to the launching components (clients). A 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. A 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() only once). The client component’s lifecycle ends. Thus, a bound service is destroyed when no client components bind to it.

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

3 SERVICE USAGE ANTI-PATTERNS

Based on the service lifecycle and our manual analysis of real-world apps, we identify four anti-patterns with respect to different types of services that can lead to service usage inefficiencies, as summarized in Table 1. 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).

<table>
<thead>
<tr>
<th>Anti-pattern</th>
<th>Service type</th>
<th>Started</th>
<th>Bound</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Premature create</td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Late destroy</td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Premature destroy</td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Service leak</td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

Table 1: The Correlation Between Service Types and Service Usage Inefficiency Anti-patterns

**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 creation to its real use.

Once a started service is started through startService(), onCreate() and onStartCommand() are invoked successively. Therefore, the use of started services is free of premature create bugs. The bugs of premature create can exist in the use of bound services: if a component binds a service but not immediately use the service (i.e., calls the methods of the service), then the service is created too early. For example, Figure 2a shows a premature create bug in the Clean Master app, where NotificationManagerService is bound too early before it is really used via dZm.aqC(), because there are other operations between these two operations.

The bugs of premature create occurring to the use of bound services may also occur to the use of hybrid services. Besides, hybrid services could be created even earlier: If the onStartCommand() method of a hybrid service is not overwritten, when s is created by a startService() statement, s will be in an idle state until it is used as a bound service. The code snippet in Figure 2b reports such a premature create bug in the WhatsApp Messenger app: The onStartCommand() method of the GoogleDriveService is not overwritten, and the startService() method is called but not directly followed by the bindService() method.

**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 the bugs of late destroy may exist. App services can be stopped (unbind) 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.
Although the stopService() method starts a service OverlayService.

The right time to stop and to unbind a hybrid service is the

Figure 3 reports two late destroy bugs in real-world apps. The code snippet in Figure 3a reports a late destroy bug in the OverlayService app: The end represents that the service’s task is finished. Otherwise, the service is destroyed late (stopped elsewhere), or not destroyed at all (cf. Anti-pattern 4).

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).

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.

Figure 3 reports two late destroy bugs in real-world apps. The code snippet in Figure 3a reports a late destroy bug in the OverlayService app, where the a() method starts a service OverlayService. Although 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 3b shows a late destroy bug in the YouTube 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.

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 to 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, the service should be created again to respond to other components. The premature destroy bugs lead to many unnecessary destroy and recreation of the same service, 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 words, at that moment 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.

Figure 4 reports a premature destroy bug in the YouTube app: The startService() method of MessageService is called three times, but stopSelf() (instead of stopSelf(int)) is used in the onStartCommand() method of MessageService.

Figure 4: A real-world premature destroy bug in WhatsApp Messenger (version 2.17.231).
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 (unbind) by end users. However, as aforementioned, the programmers should not rely on end users to stop (unbind) the app services but, instead, the services should be stopped (unbind) by the app itself. If a service is started (bind) but is not stopped (unbind), the service is leaked. As started services can keep running even after the component which starts it has terminated, the leaked service may cause severe performance issues in the long run, especially when the service executes in an isolated process.

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 were 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 can occupy much memory, normally-used 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 started services will be stopped by the system. This may alleviate the impact of late destroy and service leak, but cannot avoid them. Moreover, this cannot prevent premature create and premature destroy.

Figure 5 reports two service leak bugs in real-world apps. The code snippet in Figure 5a shows a service leak bug in the Messenger app: The startService() method that starts the service MpService is called in the a() method of the class MpActivity, but, the corresponding stopService() method is called only when the b() method returns "true". The code snippet in Figure 5b shows a service leak bug in the Google Play Music app: the service ArtDownloadService is bound but is never unbound.

4 DETECTING SERVICE USAGE BUGS

Since some services declared in the AndroidManifest file may not be implemented or used in the app code, we only consider the services that are actually used. The service types cannot be determined directly or statically from the AndroidManifest file or the service definition (i.e., the class that defines the service); they can only be determined according to the uses of the services in the app code: If a service is only initiated through startService() (bindService()), it is a started (bind) service; otherwise, it is a hybrid service. According to the service types, we then determine whether there are use cases of the services that may lead to the corresponding service usage inefficiencies. Our analysis covers all uses of each service.

ServDroid automatically detects service usage inefficiency bugs by matching the anti-patterns. To scale to large real-world apps, we start from a context-insensitive call graph and use each anti-pattern to guide a path-sensitive inter-procedural analysis. Figure 6 illustrates the framework of ServDroid, including three modules:

(1) Service Identifier: For each statement of service use (e.g., startService(intent), bindService(intent), stopService(intent)), this module identifies the corresponding service s (class object) according to the argument intent. If intent is explicit, s is obtained from intent’s API invocations, e.g., intent = new Intent(context, cls).intent.setClassName(context, cls), intent.setClassName(context, cls), or intent.setClassName(new ComponentName(context, cls)) [16]. The argument cls of these APIs is either String (e.g., “com.cea.eventos.NotifyService”) or class object (i.e., Service.class). Besides Service.class, developers also often use the reflection mechanism Class.forName(Service) to get the service class object, and therefore, both ways are considered to find the service class. If intent is implicit, s is obtained from the matched <intent-filter/> defined in the AndroidManifest XML file [16].

(2) Component Identifier: For each statement of service use, say, startService(intent), this module identifies the caller (i.e., component, e.g., an activity, a service) c (class object) that uses the service by back tracking the call graph starting from the statement (node) startService(intent). The call graph is generated by Soot [43].

(3) Bug Detector: The app control flow graph (InfoflowCFG, or CFG for short) can be generated by Soot. For each service
use, since the component $c$ and the service $s$ are known, ac-
tording to the CFG, Bug Detector determines whether or not
this service use may lead to service usage inefficiency bugs.

Since this is the crucial module of ServDroid, we present our
analysis for detecting each of the four kinds of bugs in more
detail in Sections 4.1-4.4.

**Service Identifier** is based on the call graph to find the service
that is used. Since the intent object to initiate a service can be
created in a nested method whose nesting depth may be large, and
its creation and its use can be in different layers, we set a maximum
depth (i.e., 50) for the recursive search to make sure that its creation
(and thus the service class) can be found while the efficiency is still
guaranteed.

A key analysis used in Bug Detector is the dominator analysis
based on the inter-procedural CFG, defined in Definitions 1 and 2:

**Definition 1 (Dominator).** In a CFG, a node (statement) $s_j$ is
dominated by another node $s_i$ if every path from the entry of the CFG
to $s_j$ contains $s_i$. $s_i$ is called a dominator of $s_j$.

**Definition 2 (Post-dominator).** In a CFG, a node (statement) $s_j$
is post-dominated by another node $s_i$ if every path from $s_i$ to the
exit of the CFG contains $s_j$. $s_j$ is called a post-dominator of $s_i$.

### 4.1 Detecting Premature Create Bugs

As mentioned in Section 3, there are two kinds of premature create
bugs. The first kind can occur to the usage of bound services and
hybrid services, whereas the second kind can only occur to the
usage of hybrid services.

We first discuss how to detect the first kind of premature create
bugs. We search in each method of the app for all the bindSer-
vice() statements. For each bindService() statement $stm_b$ in
the CFG of the app, we first determine the client component $c$
and the bound service $s$ such that $c$ binds $s$ through $stm_b$. Then, from
the CFG, we find a post-dominator $stm_u$ of $stm_b$ such that $stm_u$
is the first statement that $c$ invokes a method of $s$. If there exist no
statements between $stm_b$ and $stm_u$ that are relevant to $c$
invoked by $c$, $c$’s use of $s$ is free of premature create bugs. Otherwise, there
is a premature create bug, i.e., $c$ binds $s$ too early.

Given the example in Figure 2a, since $dZm.aqC()$ is the first state-
ment that $c$ uses $s$, and there are other operations invoked by $c$
(e.g., $oj()$, $arF()$) between context.bindService() and $dZm.aqC()$
in the CFG, a premature create bug is found.

We then discuss how to detect the second kind of premature cre-
ate bugs. We first check whether the onStartCommand() method
of the service $s$ is over-written. If it is over-written, the service use-
does not involve premature create bugs. Otherwise, we check
whether there is a path in the CFG of the app such that the fol-
lowing 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 following the start-
   Service() statement, but bindService() is not an immedi-
   ate post-dominator of startService(), and there is no cor-
   responding stopService() between startService() and
   bindService().

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 $stm_b$ that
binds the same service $s$, the first condition is satisfied. Otherwise,
we further check whether the corresponding unbindService() statement
is the dominator of $stm_b$ in the CFG, or the component
that binds $s$ is destroyed before $stm_s$. If either is met, the first
condition is satisfied. For the second condition, we check whether
there is a bindService() statement $stm_b$ that is the transitive post-
ominator of $stm_s$. If yes, we further check the statements (nodes
in the CFG) between $stm_s$ and $stm_b$. If there is no bindService() statement
that is the immediate post-dominator of $stm_s$, and there is no 
stopService() statement that stops $s$ and post-dominates $stm_s$, the second condition is satisfied.

For the example in Figure 2b, since onStartCommand() is not
overwritten, and the component has other operations (e.g., $q()$
between startService() and bindService() in the CFG, a pre-
mature create bug is found.

### 4.2 Detecting Late Destroy Bugs

For a started service, its use involves a late destroy bug if stop-
Self() or stopSelf(int) is not called in the onStartCommand() method
of the service. Hence, the detection is straightforward.

We elaborate on how to detect late destroy bugs occurring to
the use of bound services. We search in the CFG of the app for
all the unbindService() statements. For each unbindService() statement
$stm_u$ in the CFG, we first determine the client compo-

nent $c$ and the bound service $s$ such that $c$ unbinds $s$ through $stm_u$.
Then, from the CFG, we find a precursor $stm_p$ of $stm_u$ such that $stm_p$
is the last statement that $c$ invokes a method of $s$. If there exist no
statements between $stm_p$ and $stm_u$ that are relevant to $c$
invoked by $c$, $c$’s use of $s$ is free of late destroy bugs. Otherwise, there
is a late destroy bug, i.e., $c$ unbinds $s$ too late.

For the example in Figure 3b, the invocation of $f()$ corresponds
to the last use of the service $s$, and the invocation of $e()$ is to unbind
$s$. Since other operations (e.g., $c()$) between these two invocations
are found from the CFG, a late destroy bug is detected.

For hybrid services, we combine the above two methods (apply-
ing to started services and bound services, respectively) to deter-
mine whether their uses may involve late destroy bugs.

### 4.3 Detecting Premature Destroy Bugs

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. Therefore, the method
of detecting premature destroy bugs is straightforward.

### 4.4 Detecting Service Leak Bugs

Normally, to bind a service, each bind statement should have a
 corresponding unbind statement such that the callers (client com-
ponents) 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
aforsaid, no matter whether end users can destroy a service or not,
the app itself should have the mechanism to destroy the created service. With that in mind, we have the following steps to determine whether a service may leak:

(1) We first find all statements that start (bind) the service, i.e., `startService()` (bindService()). All such statements are summarized in a set $S_1$.

(2) We then find all statements that stop (unbind) the same 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 triggered by the event handlers (callbacks) of the UI events (e.g., user click).

(4) For each start (bind) statement in $S_1$, we check whether the corresponding stop (unbind) statement exists in $S_2$. If not, the service leaks. If yes, we then check whether the stop (unbind) statement can be always reached from the corresponding start (bind) statement. If not, the service also leaks.

In the last step above, it is challenging to precisely determine whether a stop (unbind) statement $stm_2$ can always be reached from the corresponding start (bind) statement $stm_1$. To balance precision and efficiency, our method is based on post-domination (cf. Definition 2): if $stm_2$ is a post-dominator of $stm_1$ in the CFG of the app, then the service does not leak. Otherwise, it may leak.

Let us return to the example in Figure 5a. In the CFG of Messenger, since the `startService()` method is not post-dominated by the `stopService()` method, a service leak bug is found.

### 5 Empirical Evaluation

Our evaluation aims to answer the following research questions:

- **RQ1** - **Performance of ServDroid**: What are the precision, recall, and time overhead of ServDroid?
- **RQ2** - **Energy savings**: How much energy can be saved if these services usage inefficiency bugs are fixed?
- **RQ3** - **Service usage frequency**: Are background services widely used in Android apps? Which type of services are used most frequently?
- **RQ4** - **Pervasiveness of inefficiency bugs**: Are service usage inefficiency bugs common in practice?
- **RQ5** - **Distribution of inefficiency bugs**: How are the four kinds of service usage inefficiency bugs distributed in the three types of services?
- **RQ6** - **Dominating inefficiency bugs**: Among the four kinds of service usage inefficiency bugs, which kind is the most prevalent?
- **RQ7** - **Most vulnerable service type**: Among the three types of services, the usage of which type is more prone to inefficiency bugs?

All 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. ServDroid is implemented based on Soot [43]. The input of ServDroid is an app (APK file), and its output is the detected service usage inefficiency bugs in the app.

### 5.1 Empirical Study on 45 Apps

#### 5.1.1 Experimental Setup

We first conduct an experimental evaluation on the top 45 most downloaded free Android apps according to Wikipedia\(^1\). The first 25 apps have over one billion downloads and the rest 20 apps all have over 500 million downloads. The first two columns of Table 2 respectively report the names of these 45 apps and their versions we use. It is worth mentioning that all the app versions were the latest in early January, 2018.

**Oracle.** In this experiment, our oracle is constructed based on a substantial manual inspection. Using the reverse engineering tools (dex2jar\(^2\) and jd-gui\(^3\)), each APK file is transformed into Java code, and three graduate students who are instructed with the four anti-patterns independently search the bugs in the Java code. The inconsistencies among them were solved through iterative checking and coordination. The inspection took them about two weeks. We use the code inspection results as the oracle to evaluate the effectiveness of our approach. We also use ServDroid to detect the service usage inefficiency bugs in these 45 apps, and compare the detected results with those of the manual inspection.

#### 5.1.2 Experimental Results

The third column of Table 2 summarizes the numbers of different types of services used in the 45 Android apps. The last column of Table 2 summarizes the numbers of premature create bugs (PCBs), late destroy bugs (LDBs), premature destroy bugs (PDBs), service leak bugs (SLBs), and all service usage inefficiency bugs detected by ServDroid in the 45 apps. Taking the manual inspection results as the ground truth, the accuracy of ServDroid is reported as follows.

| RQ1 | The overall precision and recall of the results returned by ServDroid are both 100%. The total runtime overhead of ServDroid on analyzing the 45 apps is 4,325 seconds, with about 96 seconds to analyze one app on average. |
| Implication | The lightweight analysis of ServDroid for detecting such particular anti-patterns has a high accuracy, and is scalable. |

**Threats to validity.** The manual inspection inevitably suffers from limitations; for example, many variable values cannot be inferred and thus some infeasible paths cannot be known, and potential paths of service usages may be missed due to the huge search space. Thereby, the 100% precision and recall of ServDroid may not be the real case. However, our empirical study shows that ServDroid can find many service usage inefficiencies, and thus is useful.

To investigate the energy impact of the inefficiency bugs, we first use Soot to instrument the infected apps to fix the inefficiency bugs and repackage them (Based on our static analysis results, the statements to fix a bug and the position where they should be instrumented are all clear, and thus the fixing is fully automated). Then, we run the original and repackaged apps independently on a phone (Google Nexus 6\(^5\), running Android 8.0) with the same setting (e.g., user operations) for 15 minutes\(^6\), and use the tool (also an app) Trepn Profiler to measure the battery cost of them, respectively. Based on the static analysis result of ServDroid, we ensure


\(^2\)https://sourceforge.net/projects/dex2jar

\(^3\)http://jd.benow.ca

\(^4\)It is sufficient to see the difference of battery cost.
that the app behavior related to the bugs is exercised. The above measurement is repeated three times. The average battery cost is summarized in Table 3, including the data of 38 apps (the other 7 apps are not measured because they are either free of the bugs or cannot run independently). We do not break down the energy savings for each kind of bugs, as their numbers are imbalanced, and service leak and late destroy are often correlated, which cannot be fixed separately. The results in Table 3 answer RQ2:

**Answer to RQ2:** The average energy consumptions (15 minutes) of an app before and after the service usage inefficiency bugs are fixed are 546.73 Joule and 459.59 Joule, respectively. That is, an app can save on average 87.14 Joule in 15 minutes if the service usage inefficiency bugs are fixed.

**Implication:** The energy consumption is reduced after the service usage inefficiency bugs are fixed, and thus the bugs have a significant impact on energy consumption.

<table>
<thead>
<tr>
<th>App name</th>
<th>Version</th>
<th>Started</th>
<th>Bound</th>
<th>Hybrid</th>
<th>Total</th>
<th>PCBs</th>
<th>LDBs</th>
<th>PDBs</th>
<th>SLBs</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Play services</td>
<td>11.0.55 (436-156917137)</td>
<td>130</td>
<td>17</td>
<td></td>
<td>153</td>
<td>6</td>
<td>4</td>
<td>95</td>
<td></td>
<td>105</td>
</tr>
<tr>
<td>Gmail</td>
<td>7.6.4.158567011.release</td>
<td>18</td>
<td>2</td>
<td>4</td>
<td>24</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Maps</td>
<td>9.54.1</td>
<td>12</td>
<td>9</td>
<td>4</td>
<td>25</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>YouTube</td>
<td>12.23.60</td>
<td>10</td>
<td>2</td>
<td>3</td>
<td>15</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Facebook</td>
<td>10.2.0</td>
<td>22</td>
<td>8</td>
<td>4</td>
<td>34</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>Google</td>
<td>7.3.25.21.arm</td>
<td>21</td>
<td>10</td>
<td>6</td>
<td>37</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>Google+</td>
<td>9.14.0.158314320</td>
<td>23</td>
<td>1</td>
<td>4</td>
<td>28</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>GoogleText-to-Speech</td>
<td>3.11.12</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>WhatsApp Messenger</td>
<td>2.17.231</td>
<td>9</td>
<td>5</td>
<td>5</td>
<td>19</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>19</td>
</tr>
<tr>
<td>Google Play Books</td>
<td>3.13.17</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Messenger</td>
<td>123.0.0.11.70</td>
<td>40</td>
<td>1</td>
<td>0</td>
<td>41</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Hangouts</td>
<td>20.0.156935076</td>
<td>11</td>
<td>9</td>
<td>2</td>
<td>22</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Google Chrome</td>
<td>58.0.3029.83</td>
<td>16</td>
<td>9</td>
<td>2</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Google Play Games</td>
<td>3.9.08(3448271-036)</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Google TalkBack</td>
<td>5.2.0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Google Play Music</td>
<td>7.8.4818-1.R.4063206</td>
<td>15</td>
<td>3</td>
<td>4</td>
<td>22</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>Google Play Newsstand</td>
<td>4.5.0</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Google Play Movies &amp; TV</td>
<td>3.26.5</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Google Drive</td>
<td>2.7.153.14.34</td>
<td>6</td>
<td>6</td>
<td>3</td>
<td>15</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Samsung Push Service</td>
<td>1.8.02</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Instagram</td>
<td>10.26.0</td>
<td>14</td>
<td>4</td>
<td>2</td>
<td>20</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>Android System WebView</td>
<td>58.0.3029.83</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Google Photos</td>
<td>2.16.0.157775819</td>
<td>17</td>
<td>8</td>
<td>6</td>
<td>31</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>Google Street View</td>
<td>2.0.0.157538376</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>13</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>Skype</td>
<td>8.0.0.44736</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Clean Master</td>
<td>5.17.4</td>
<td>20</td>
<td>16</td>
<td>3</td>
<td>39</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>26</td>
<td>41</td>
</tr>
<tr>
<td>Subway Surfers</td>
<td>1.72.1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dropbox</td>
<td>50.22</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>12</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Candy Crush Saga</td>
<td>1.101.0.2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Viber Messenger</td>
<td>8.0.0.3</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>17</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>Twitter</td>
<td>7.0.0</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>13</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>LINE</td>
<td>7.5.2</td>
<td>11</td>
<td>5</td>
<td>3</td>
<td>19</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>17</td>
</tr>
<tr>
<td>HP Print Service Plugin</td>
<td>3.4-2.3.0</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Flipboard</td>
<td>4.0.13</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Samsung Print Service Plugin</td>
<td>3.02.170302</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>Super-Bright LED Flashlight</td>
<td>1.17</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Gboard</td>
<td>6.3.28.159021150</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>12</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Cloud Print</td>
<td>1.36b</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Snapchat</td>
<td>10.10.5.0</td>
<td>6</td>
<td>3</td>
<td>0</td>
<td>9</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>Pea</td>
<td>1.4.73</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Google Translate</td>
<td>5.9.0.ROC07.155715800</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>My Talking Tom</td>
<td>4.2.1.50</td>
<td>17</td>
<td>6</td>
<td>2</td>
<td>25</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>13</td>
<td>18</td>
</tr>
<tr>
<td>Security Master</td>
<td>3.4.1</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>14</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>Facebook Lite</td>
<td>20.0.15</td>
<td>20</td>
<td>9</td>
<td>4</td>
<td>33</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>19</td>
</tr>
<tr>
<td>imo messenger</td>
<td>11.3.2</td>
<td>16</td>
<td>6</td>
<td>7</td>
<td>29</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>14</td>
<td>24</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td></td>
<td>601</td>
<td>186</td>
<td>105</td>
<td>892</td>
<td>55</td>
<td>90</td>
<td>19</td>
<td>318</td>
<td>482</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td>13.4</td>
<td>4.1</td>
<td>2.3</td>
<td>19.8</td>
<td>1.2</td>
<td>2.0</td>
<td>0.4</td>
<td>7.1</td>
<td>10.7</td>
</tr>
</tbody>
</table>
**Table 3: Energy Consumption Before and After Bug Fixing**

<table>
<thead>
<tr>
<th>App name</th>
<th>Energy consumption</th>
<th>Energy consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original (J)</td>
<td>Repaired (J)</td>
</tr>
<tr>
<td>Google Play services</td>
<td>574.68</td>
<td>373.42</td>
</tr>
<tr>
<td>Gmail</td>
<td>437.05</td>
<td>388.25</td>
</tr>
<tr>
<td>Maps</td>
<td>626.67</td>
<td>487.28</td>
</tr>
<tr>
<td>YouTube</td>
<td>803.31</td>
<td>687.05</td>
</tr>
<tr>
<td>Facebook</td>
<td>533.94</td>
<td>510.34</td>
</tr>
<tr>
<td>Google</td>
<td>498.40</td>
<td>426.82</td>
</tr>
<tr>
<td>Google+</td>
<td>438.59</td>
<td>422.57</td>
</tr>
<tr>
<td>GoogleText-to-Speech</td>
<td>395.90</td>
<td>353.91</td>
</tr>
<tr>
<td>WhatsApp Messenger</td>
<td>421.93</td>
<td>343.58</td>
</tr>
<tr>
<td>Google Play Books</td>
<td>474.77</td>
<td>335.57</td>
</tr>
<tr>
<td>Messenger</td>
<td>509.60</td>
<td>412.92</td>
</tr>
<tr>
<td>Hangouts</td>
<td>370.33</td>
<td>338.30</td>
</tr>
<tr>
<td>Google Play Games</td>
<td>549.86</td>
<td>475.03</td>
</tr>
<tr>
<td>Google TalkBack</td>
<td>458.57</td>
<td>429.97</td>
</tr>
<tr>
<td>Google Play Music</td>
<td>567.35</td>
<td>458.59</td>
</tr>
<tr>
<td>Google Play Newsstand</td>
<td>512.50</td>
<td>442.20</td>
</tr>
<tr>
<td>Google Play Movies &amp; TV</td>
<td>363.39</td>
<td>317.45</td>
</tr>
<tr>
<td>Google Drive</td>
<td>338.49</td>
<td>286.74</td>
</tr>
<tr>
<td>Instagram</td>
<td>496.44</td>
<td>452.05</td>
</tr>
<tr>
<td>Google Photos</td>
<td>484.75</td>
<td>435.66</td>
</tr>
<tr>
<td>Google Street View</td>
<td>424.91</td>
<td>369.53</td>
</tr>
<tr>
<td>Clean Master</td>
<td>482.36</td>
<td>399.24</td>
</tr>
<tr>
<td>Subway Surfers</td>
<td>950.96</td>
<td>899.31</td>
</tr>
<tr>
<td>Dropbox</td>
<td>462.38</td>
<td>408.53</td>
</tr>
<tr>
<td>Candy Crush Saga</td>
<td>820.80</td>
<td>557.36</td>
</tr>
<tr>
<td>Viber Messenger</td>
<td>483.64</td>
<td>419.39</td>
</tr>
<tr>
<td>Twitter</td>
<td>588.59</td>
<td>537.88</td>
</tr>
<tr>
<td>LINE</td>
<td>572.07</td>
<td>497.71</td>
</tr>
<tr>
<td>Flipboard</td>
<td>486.41</td>
<td>438.67</td>
</tr>
<tr>
<td>Super-Bright LED Flashlight</td>
<td>454.12</td>
<td>385.26</td>
</tr>
<tr>
<td>Gboard</td>
<td>497.44</td>
<td>441.62</td>
</tr>
<tr>
<td>Snapchat</td>
<td>611.63</td>
<td>461.71</td>
</tr>
<tr>
<td>Pou</td>
<td>1,000.41</td>
<td>647.33</td>
</tr>
<tr>
<td>Google Translate</td>
<td>378.40</td>
<td>342.56</td>
</tr>
<tr>
<td>My Talking Tom</td>
<td>1,432.49</td>
<td>1,197.11</td>
</tr>
<tr>
<td>Security Master</td>
<td>437.09</td>
<td>368.08</td>
</tr>
<tr>
<td>Facebook Lite</td>
<td>362.16</td>
<td>297.52</td>
</tr>
<tr>
<td>imo messenger</td>
<td>492.46</td>
<td>325.50</td>
</tr>
<tr>
<td>Sum</td>
<td>20,814.81</td>
<td>17,471.82</td>
</tr>
<tr>
<td>Average</td>
<td>547.76</td>
<td>459.78</td>
</tr>
</tbody>
</table>

### 5.2 Experiments on 1,000 Apps

To assess the service usage and the relevant inefficiency bugs more extensively in practice, we further conducted an empirical study on 1,000 real-world Android apps using ServDroid. These 1,000 apps were randomly selected from Google play (accessed in Dec 2018). We summarize the results below.

**Answer to RQ3:** Among the 1,000 apps, 939 use background services. The total numbers (proportions) of started, bound, and hybrid services used in these apps are 4,952 (60.87%), 2,468 (30.34%), and 715 (8.79%), respectively. Each app uses about 8 app services on average.

**Implication:** Services are widely used in Android apps, and started services are the most frequently used type of services.

**Answer to RQ4:** Surprisingly, service usage inefficiency bugs are common in the 1,000 Android apps. 825 (82.5%) of them are infected by at least one kind of inefficiency bug; 608 (60.8%) of them involve at least two kinds of inefficiency bugs; 304 (30.4%) of them have no less than three kinds of inefficiency bugs; and 59 (5.9%) of them are found to have all the four kinds of inefficiency bugs. Each app has 4.43 service usage inefficiency bugs on average.

**Implication:** Service usage inefficiency bugs are common in real-world Android apps.

**Answer to RQ5:** The numbers (proportions) of premature create bugs occurring to the started services, bound services, and hybrid services are 0 (0%), 400 (71.05%), and 163 (28.95%), respectively. The numbers (proportions) of late destroy bugs occurring to the started services, bound services, and hybrid services are 491 (38.18%), 620 (48.21%), and 175 (13.61%), respectively. The numbers (proportions) of premature destroy bugs occurring to the started services, bound services, and hybrid services are 137 (78.29%), 0 (0%), and 38 (21.71%), respectively. The numbers (proportions) of service leak bugs occurring to the started services, bound services, and hybrid services are 1,720 (71.61%), 392 (16.32%), and 290 (12.07%), respectively.

**Implication:** This confirms the analysis results in Table 1 that the premature create bugs do not occur on the usage of started 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 RQ6:** The total numbers of premature create bugs, late destroy bugs, premature destroy bugs, and service leak bugs in the 1,000 apps are 563, 1,286, 175, and 2,402, respectively. The proportions of the four kinds of service usage inefficiency bugs are 12.72%, 29.06%, 3.95%, and 54.27%, respectively.

**Implication:** The number of service leak bugs is much larger than the total number of the other three kinds of service usage inefficiency bugs. Service leak bugs are the dominant kind of service usage inefficiencies.

**Answer to RQ7:** 2,348 service usage inefficiency bugs are relevant to the usage of 4,952 started services, 1,412 bugs to the usage of 2,468 bound services, and 666 bugs to the usage of 715 hybrid services.

**Implication:** Among the three types of services, the usage of hybrid services has the highest possibility to involve inefficiency bugs, whereas the usage of started services and bound services has a lower possibility to involve inefficiency bugs, but still significant.

To make sure that the readers can reproduce our empirical results, we archive all apps used in the our study along with our tool ServDroid on GitHub.

We have also applied ServDroid to different versions of some apps, and found that the numbers of service usage inefficiency bugs decrease in newer versions. This implies that developers fixed some of these bugs, though not completely. Overall, our empirical
study indicates that service usage inefficiency bugs are prevalent in Android apps.

6 RELATED WORK

Our research is related to GUI testing, service analysis and testing, and performance (energy) testing for mobile (Android) apps. In the following, we review representative existing work on these topics.

**GUI testing.** Most existing testing approaches for Android apps focus on GUI testing. According to the exploration strategies employed, Choudhary et al. [12] summarize three main categories of testing approaches: random testing [15, 21, 32, 41], model-based testing [3, 5, 11, 42, 47], and advanced testing [4, 25, 33, 35]. Despite the fact that Monkey [15] 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 [12]. Dynodroid [32] 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 [41] 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, 11, 42, 47]. 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 [11] to learn the model. The third category approaches leverage advanced techniques to efficiently generate effective event sequences for app testing [4, 25, 33, 35]. For example, ACTEve [4] uses symbolic execution and EvoDroid [33] employs evolutionary algorithm to generate event sequences. Sapience [35] formulates the event sequence generation as a multiple-objective optimization problem and employs a 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, 14, 22, 30, 40, 45]. In terms of app services, Khanmohammadi et al. [26] find that malware may use background services to perform malicious operations with no communication with the other components of the app. 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 gains little attention. Lee et al. present a system facility for analyzing energy consumption of Android system services [27]. Snowdrop [48] 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 is able to find common bugs (functional bugs that lead to app crashes) in services, but may meet difficulties in detecting service usage inefficiency bugs (non-functional bugs that may not lead to app crashes) targeted in our work. Recently, a dynamic analysis tool LESDroid [31]

is presented to find exported service leaks, whereas ServDroid is a static analysis tool and is more general, which could not only find both private and exported service leaks but also detect the other three kinds of service usage inefficiency bugs.

**Performance testing.** Non-functional or performance bugs in apps are also important for user experience. Liu et al. [28] 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, 20, 34, 37], energy bugs and the corresponding detection and testing solutions draw increasing attention [8, 18, 23, 24, 29, 38, 39, 44, 46]. Pathak et al. study app energy bugs which arise from mishandling power control APIs [38]. Based on a hardware power monitor, Vásquez et al. mine and analyze energy-greedy API usage patterns that correspond to suboptimal usage or choice of APIs. Chen et al. show that apps can drain battery even while running in background, and they present a system that can suppress background activities that are not required to the user experience [10]. However, these researches do not consider service usage anti-patterns presented in this paper. Energy bugs are found to be highly relevant to resource leaks [8, 18, 29, 46]. Banerjee et al. [7, 8] 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 resource (also service) leak bugs, the test oracle based on the power consumption is expensive and time-consuming. To reduce the cost of testing, Jabbarvand et al. [24] propose an approach to minimizing the energy-aware test-suite. Wu et al. [46] present a static analysis approach to detecting GUI-related energy-drain bugs, whereas our approach aims to detect service usage inefficiency bugs.

7 CONCLUSIONS

It is extremely challenging for testing techniques to reveal service usage inefficiencies in mobile apps, because such latent bugs do not exhibit immediate bug symptoms such as crashes. We have conducted an in-depth study of Android services, and presented four anti-patterns that lead to service usage inefficiency bugs in real-world apps. We have also presented ServDroid, a scalable static analysis approach to automatically detect such bugs based on these anti-patterns. To our knowledge, ServDroid is among the first to detect service performance bugs using static analysis. Our empirical evaluation on a large collection of real-world Android apps indicates that ServDroid is highly effective and service usage inefficiency bugs are prevalent in practice, which severely impact the apps on energy consumption.

**ACKNOWLEDGMENTS**

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61761136003, the Natural Science Foundation of Jiangsu Province under Grant No. BK20171427, and the Collaborative Innovation Center of Novel Software Technology and Industrialization.


