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
zjing8017@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

ACM Reference Format:
Wei Song, Jing Zhang, and Jeff Huang. 2019. ServDroid: Detecting Service Usage Inefficiencies in Android Applications. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338950

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 fulfil 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

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338950

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.,

[1]https://github.com/wsong-nj/ServDroid
[2]https://developer.qualcomm.com/software/trepn-power-profiler

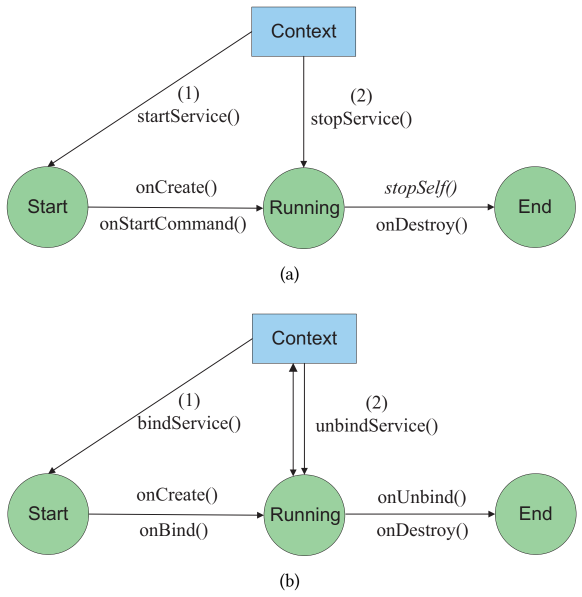

Figure 1: App service lifecycle: (a) started service, (b) bound service.

`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.

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

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

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()`), or 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).

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()`

```
public class NCBlackListActivity{
  public final void run(){
    Intent intent = new Intent(context, NotificationManagerService
    .class);
    context.bindService(intent, aqL.dYV, 1);
    oj();
    arF();
     ...
    dZm.aqC();
  }
}
public class NotificationManagerService extends Service {
  public IBinder onBind(Intent intent) {
     ...
    return this.dZm;
  }
}
```

(a)

```
public final class GoogleDriveActivity extends apf implements
  c.b, e.a{
    public final void onCreate(Bundle paramBundle){
      Intent localIntent = getIntent();
      onNewIntent(localIntent);
      q();
      . . . . . .
      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);
    }
}
```

(b)

Figure 2: Real-world premature create bugs: (a) A premature create bug in *Clean Master* (version 5.17.4), (b) A premature create bug in *WhatsApp Messenger* (version 2.17.231).

method of a hybrid service s 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 (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

```
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)

```
final class ahx extends agj implements ServiceConnection{
  final void b(){
    d();
    ......
    f();
    c();
    ......
    e();
  }
  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);
  }
}
```

(b)

Figure 3: Real-world late destroy bugs: (a) A late destroy bug in *Twitter* (version 7.0.0), (b) A late destroy bug in *YouTube* (version 12.23.60).

at all [48]. 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 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.

Figure 3 reports two late destroy bugs in real-world apps. The code snippet in Figure 3a reports a late destroy bug in the *Twitter* 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

```
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;
    }
}
```

Figure 4: A real-world premature destroy bug in *WhatsApp Messenger* (version 2.17.231).

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 *WhatsApp Messenger* app: The `startService()` method of `MessageService` is called three times, but `stopSelf()` (instead of `stopSelf(int)`) is used in the `onStartCommand()` method of `MessageService`.

Anti-pattern 4 (**Service Leak**). *A service is never destroyed after its use, even when the apps which use the service terminate.*

Anti-pattern 4 refers to the bugs that the services are never destroyed except for the situation that they are stopped (unbound) by end users. However, as aforementioned, the programmers should not rely on end users to stop (unbind) the app services but, instead, the services should be stopped (unbound) by the app itself. If a service is started (bound) but is not stopped (unbound), 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.

```
public class MpActivity extends Activity {
  public final void a(g paramg){
    startService (new Intent( this , MpService. class ));
    . . . . .
    if (b()){
      stopService (new Intent( this , MpService. class ));
    }
    else {
      . . . . .
    }
  }
}
```

(a)

```
public class ArtMonitorImpl{
  public void startMonitoring () {
    ...
    this .mContext.bindService(new Intent( this .mContext,
    ArtDownloadService.class ), this .mServiceConnection, 5);
    ...
  }
}
```

(b)

Figure 5: Real-world service leak bugs: (a) A service leak bug in *Messenger* (version 123.0.0.11.70), (b) A service leak bug in *Google Play Music* (version 7.8.4818-1.R.4063206).

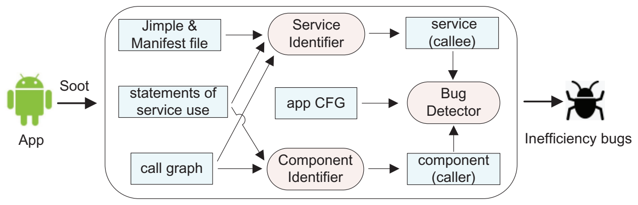

Figure 6: An overview of ServDroid.

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 (bound) 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.setClass(context, cls)`, `intent.setClassName(context, cls)`, or `intent.setComponent(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, according 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_i *is post-dominated by another node* s_j *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 `bindService()` 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 statement 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 create bugs. 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 check whether there is a path in the CFG of the app 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 following the `startService()` statement, but `bindService()` is not an immediate post-dominator of `startService()`, and there is no corresponding `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_1} that is the transitive post-dominator of stm_s. If yes, we further check the statements (nodes in the CFG) between stm_s and stm_{b_1}. 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 premature create bug is found.

4.2 Detecting Late Destroy Bugs

For a started service, its use involves a late destroy bug if `stopSelf()` 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_{un} in the CFG, we first determine the client component c and the bound service s such that c unbinds s through stm_{un}. Then, from the CFG, we find a precursor stm_u of stm_{un} such that stm_u is the last statement that c invokes a method of s. If there is no statements between stm_u and stm_{un} 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 (applying to started services and bound services, respectively) to determine 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 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 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-dominator* (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[3]. 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[4] and jd-gui[5]), 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.

Answer to 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 6P, 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

[3]https://en.wikipedia.org/wiki/List_of_most_downloaded_Android_applications (accessed in Jan 2018)
[4]https://sourceforge.net/projects/dex2jar
[5]http://jd.benow.ca
[6]It is sufficient to see the difference of battery cost.

Table 2: Total Numbers of Services and Service Usage Inefficiency Bugs in the Top 45 Most Downloaded Free Apps

App name	**Version**	**# Services**				**# Service usage inefficiency bugs**				
		Started	**Bound**	**Hybrid**	**Total**	**PCBs**	**LDBs**	**PDBs**	**SLBs**	**Total**
Google Play services	11.0.55 (436-156917137)	**130**	**17**	6	**153**	**6**	4	0	**95**	**105**
Gmail	7.6.4.158567011.release	18	2	4	24	1	1	0	5	7
Maps	9.54.1	12	9	4	25	2	1	1	9	13
YouTube	12.23.60	10	2	3	15	2	2	0	3	7
Facebook	10.2.0	22	8	4	34	0	4	0	5	9
Google	7.3.25.21.arm	21	10	6	37	5	3	0	1	9
Google+	9.14.0.158314320	23	1	4	28	0	1	0	7	8
GoogleText-to-Speech	3.11.12	4	1	0	5	0	0	0	2	2
WhatsApp Messenger	2.17.231	9	5	5	19	5	2	**2**	10	19
Google Play Books	3.13.17	8	2	1	11	0	0	0	1	1
Messenger	123.0.0.11.70	40	1	0	41	0	1	0	9	10
Hangouts	20.0.156935076	11	9	2	22	0	1	1	4	6
Google Chrome	58.0.3029.83	16	9	2	27	0	0	0	0	0
Google Play Games	3.9.08(3448271-036)	4	0	0	4	0	0	0	3	3
Google TalkBack	5.2.0	2	1	0	3	0	0	0	1	1
Google Play Music	7.8.4818-1.R.4063206	15	3	4	22	1	3	1	10	15
Google Play Newsstand	4.5.0	8	1	1	10	0	1	1	3	5
Google Play Movies & TV	3.26.5	6	4	2	12	0	0	0	4	4
Google Drive	2.7.153.14.34	6	6	3	15	1	2	0	3	6
Samsung Push Service	1.8.02	11	0	0	11	0	3	0	1	4
Instagram	10.26.0	14	4	2	20	3	1	0	7	11
Android System WebView	58.0.3029.83	8	2	1	11	0	0	0	0	0
Google Photos	2.16.0.157775819	17	8	6	31	3	2	1	4	10
Google Street View	2.0.0.157538376	6	5	2	13	0	4	0	8	12
Skype	8.0.0.44736	10	1	0	11	0	0	0	0	0
Clean Master	5.17.4	20	16	3	39	3	**10**	2	26	41
Subway Surfers	1.72.1	2	1	0	3	0	0	0	1	1
Dropbox	50.2.2	8	3	1	12	2	1	1	4	8
Candy Crush Saga	1.101.0.2	2	1	1	4	0	1	0	0	1
Viber Messenger	8.0.0.3	8	6	3	17	4	3	0	5	12
Twitter	7.0.0	9	2	2	13	0	1	0	3	4
LINE	7.5.2	11	5	3	19	3	1	1	12	17
HP Print Service Plugin	3.4-2.3.0	4	3	2	9	1	1	0	5	7
Flipboard	4.0.13	4	3	3	10	0	1	0	3	4
Samsung Print Service Plugin	3.02.170302	7	3	0	10	1	2	0	6	9
Super-Bright LED Flashlight	1.1.7	5	0	4	9	0	2	0	7	9
Gboard	6.3.28.159021150	6	4	2	12	1	0	0	2	3
Cloud Print	1.36b	8	1	1	10	0	2	0	1	3
Snapchat	10.10.5.0	6	3	0	9	1	5	1	2	9
Pou	1.4.73	4	0	0	4	0	2	0	2	4
Google Translate	5.9.0.RC07.155715800	6	0	1	7	1	4	0	3	8
My Talking Tom	4.2.1.50	17	6	2	25	0	3	2	13	18
Security Master	3.4.1	7	3	4	14	3	2	2	7	14
Facebook Lite	20.0.15	20	9	4	33	1	9	2	7	19
imo messenger	11.3.2	16	6	**7**	29	5	4	1	14	24
Sum	-	601	186	105	892	55	90	19	318	482
Average	-	13.4	4.1	2.3	19.8	1.2	2.0	0.4	7.1	10.7

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 3: Energy Consumption Before and After Bug Fixing

App name	Energy consumption	
	Original (J)	Repaired (J)
Google Play services	574.68	373.42
Gmail	437.05	388.25
Maps	626.67	487.28
YouTube	803.31	687.05
Facebook	553.94	510.34
Google	498.40	426.82
Google+	438.59	422.57
GoogleText-to-Speech	395.90	353.91
WhatsApp Messenger	421.93	343.58
Google Play Books	474.77	435.57
Messenger	509.60	412.92
Hangouts	370.33	338.30
Google Play Games	549.86	475.03
Google TalkBack	458.57	429.97
Google Play Music	567.35	458.59
Google Play Newsstand	512.50	442.20
Google Play Movies & TV	363.39	317.45
Google Drive	338.49	286.74
Instagram	496.44	452.05
Google Photos	484.75	435.66
Google Street View	424.91	369.53
Clean Master	482.36	399.24
Subway Surfers	950.96	899.31
Dropbox	462.38	408.53
Candy Crush Saga	820.80	557.36
Viber Messenger	483.64	419.39
Twitter	588.59	537.88
LINE	572.07	497.71
Flipboard	486.41	438.67
Super-Bright LED Flashlight	454.12	385.26
Gboard	497.44	441.62
Snapchat	611.63	461.71
Pou	1,000.41	647.33
Google Translate	378.40	342.36
My Talking Tom	1,432.49	1,197.11
Security Master	437.09	368.08
Facebook Lite	362.16	297.52
imo messenger	492.46	325.50
Sum	20,814.81	17,471.82
Average	547.76	459.78

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%), 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. Sapienz [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.

REFERENCES

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

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

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

[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE'12, Cary, NC, USA - November 11 - 16.* 59:1–59:11.

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

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

[7] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. 2018. EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps. *IEEE Trans. Software Eng.* 44, 5 (2018), 470–490.

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

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

[10] Xiaomeng Chen, Abhilash Jindal, Ning Ding, Yu Charlie Hu, Maruti Gupta, and Rath Vannithamby. 2015. Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking, MobiCom'15, Paris, France, September 7-11,.* 40–52.

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

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

[13] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. 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.* 408–419.

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

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

[16] Google. 2017. Android intents-filters. Retrieved February 1, 2019 from https://developer.android.google.cn/guide/components/intents-filters.html

[17] Google. 2017. Android Services. Retrieved February 1, 2019 from https://developer.android.com/guide/components/services.html

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

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

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

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

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

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

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

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

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

[27] Seokjun Lee, Wonwoo Jung, Yohan Chon, and Hojung Cha. 2015. EnTrack: a system facility for analyzing energy consumption of Android system services. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp'15, Osaka, Japan, September 7-11,.* 191–202.

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

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

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

[31] Jun Ma, Shaocong Liu, Yanyan Jiang, Xianping Tao, Chang Xu, and Jian Lu. 2018. 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.* 244–254.

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

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

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

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

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

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

[38] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. 2012. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *The 10th International Conference on Mobile Systems, Applications, and Services, MobiSys'12, Ambleside, United Kingdom - June 25 - 29.* 267–280.

[39] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Morley Mao, Subhabrata Sen, and Oliver Spatscheck. 2011. Profiling resource usage for mobile applications: a cross-layer approach. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, MobiSys'11, Bethesda, MD, USA, June 28 - July 01.* 321–334.

[40] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin R. B. Butler, William Enck, and Patrick Traynor.

2016. *droid: Assessment and Evaluation of Android Application Analysis Tools. *ACM Comput. Surv.* 49, 3 (2016), 55:1–55:30.

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

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

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

[44] Mario Linares Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining energy-greedy API usage patterns in Android apps: an empirical study. In *11th Working Conference on Mining Software Repositories, MSR'14, Proceedings, May 31 - June 1, Hyderabad, India*. 2–11.

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

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

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

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