LESDroid - A Tool for Detecting Exported Service Leaks of Android Applications

Jun MA
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China, 210023
majun@nju.edu.cn

Shaocong LIU
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China, 210023
141270022@snju.edu.cn

Yanyan JIANG
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China, 210023
jyy@nju.edu.cn

Xianping TAO
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China, 210023
txp@nju.edu.cn

Chang XU
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China, 210023
changxu@nju.edu.cn

Jian LU
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China, 210023
lj@nju.edu.cn

ABSTRACT

Services are widely used in Android apps. However, services may leak such that they are no longer used but cannot be recycled by the Garbage Collector. Service leaks may cause an app to misbehave, and are vulnerable to malicious external apps when the service is exported or it is accessible through other exported services. In this paper, we present LESDroid for exported service leaks detection. LESDroid automatically generates service instances and workloads (start/stop or bind/unbind of exported services) of the app under test, and applies a designated oracle to the heap snapshot for service leak detection. We evaluated LESDroid using 375 commercial apps, and found 97 leaked services and 98 distinct leak entries in 70 apps.

CCS CONCEPTS

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

KEYWORDS

Android; testing; memory leak; vulnerability; service

ACM Reference Format:
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 *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196321.3196336

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196336

1 INTRODUCTION

Services are widely used in Android apps (e.g., 74,831 services are found in 28,662 of 41,537 `.apk` files in our evaluation data set) to perform long-running operations (Internet downloading, data/service updates, content provider maintenance, etc.) in the background [3]. On the other hand, services run without user supervision and are prone to errors [28]. There may be a significant portion of app logic in the background services, and bugs in such services may lead to lagging, app crash, network overuse, or energy depletion.

Sometimes, design/implementation flaws in a service may cause it to leak, named a *service leak*. A resource (a memory object, a system handler, or a service) is leaked if it is no longer used but cannot be freed up by the Garbage Collector (GC). Service leaks may cause memory bloat, app crashes, or system reboots. Especially, when a leaked service is accessible[1] to other apps, a malicious client can intentionally magnify the consequences of leaking and hurt the availability of the service's owner app. *Service leak attacks*, crafted by silently flooding the app with service creations, yield app users receiving notifications of crash reports and/or alarms of memory/energy overuse, and thus result in uninstallation of the app or negative comments in the market. For instance, according to our experiment in Sec.4.4, service leak attacks, within one hour, could bring more memory usages of tens to hundreds megabytes to the processes of about 63% distinct detected leak entry services, and cause deaths to 66 processes.

In this paper, we focus on the problem of *Exported Service Leak(ES-Leak)*, where a service leak can be triggered through the interaction with an exported service. We name the service that leaks as *Leak Service* and the exported service as *Leak Entry Service*. There are many general tools for analyzing dumped Java or Android heap files [11, 16]. However, whether the leaks can be detected and how long the process would take are highly dependent on the experience of the developer. Automated tools like LeakCanary [13] and LeakDAF [15] are for detecting leaked Android activities or fragments but not for services. To the best of our knowledge, there are only few

[1] Exporting a service is popular because this is the most convenient way to provide inter-app communications, e.g., sharing user generated contents.

existing works on testing Android services [7] [28], and none of them focus on service leaks.

This paper presents LESDroid for automated detection of ES-Leaks. We design LESDroid with the goal of detecting both leak services and leak entry services in an effective and efficient way. To achieve this goal, LESDroid faces two challenges: (a) how to access as many leak services as possible, and (b) how to identify leaked service instances. LESDroid meets these two challenges by implementing different strategies based on static analyses of both the `Androidmanifext.xml` and bytecodes of exported services of the target app to generate `Intent`s. The generated `Intent`s are then used by a client, running as a background service, as arguments to dynamically access exported services of the target app via inter-component communication (i.e., invoking the `startService()`, `bindService()` API). Finally, LESDroid determines a service as leak service if any leaked instance of the service is found in the heap snapshot. The main contributions of this paper are:

- To the best of our knowledge, we are the first work on the study of the ES-leaks problem in Android.
- We proposed an effective and efficient tool, LESDroid, for the detection of ES-leaks.
- We successfully applied LESDroid to 375 commercial apps from GooglePlay, and detected 97 distinct leak services and 98 distinct leak entry services from 70 apps.
- By further conducting service leak attacks to the detected ES-leaks, we found that 63 of these 98 leak entry services leaded to obvious memory bloats, and 47 of them resulted in the deaths of the associated processes within one hour.

The rest of this paper is organized as follows. Sec. 2 gives a brief introduction to the background of this work; Sec. 3 reveals the key ideas and structures of the LESDroid tool; Sec. 4 depicts the implementation details of our prototype of LESDroid and demonstrates the experiments carried out by applying our prototype to 375 commercial apps; Sec. 5 discusses the threats to validity of our experiments and limitations of LESDroid; after briefly introducing some related work in Sec. 6, we finally conclude this paper and discuss our future work in Sec. 7.

2 BACKGROUND

2.1 Service in Android

An Android **Service** is a simple application component that works in two different ways [1]. A service can be *started* by another component (such as an activity) via calling the `startService()` API with proper `Intent`. Usually, a started service performs a single operation and does not return a result to the caller. Meanwhile, a service can be *bound* by an application component via calling the `bindService()` API. A bound service offers a client-server interface that allows other components to interact with the service, send requests, receive results. Furthermore, a service can be both started and bound at the same time.

2.1.1 Service Lifecycle. The lifecycle of a service is much simpler than that of an activity, and can follow either of these two paths depending on whether the service is started or bound [1]. A *started service* is created when another component starts it via `startService()`. The service then runs indefinitely and must stop itself by calling `stopSelf()`. Another component can also stop the service by calling `stopService()`. When the service is stopped, the system destroys it. While a *bound service* is created when another component (a client) binds to it via `bindService()`. The client then communicates with the service through an `IBinder` interface. The client can close the connection by calling `unbindService()`. Multiple clients can bind to the same service and the system destroys the service only when all clients unbind from it.

In Android, each application process is associated with an instance of `ActivityThread`, which is in charge of scheduling and executing activities, services, broadcasts in the process. `ActivityThread` has a special field *mServices* (of type `ArrayMap`), which keeps references to all non-destroyed service instances running the corresponding process (see Fig.1 for an example). Once a service instance is destroyed, it would be removed from the corresponding *mServices* field.

Figure 1: An example reference chain from an `ActivityThread` to a service instance

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
        android"
    package="com.example.servicetest"
    android:versionCode="1"
    android:versionName="1.0" >
    <application ... >
        ...
        <service
            android:name=".Service1"
            android:enabled="true">
        </service>
        <service
            android:name=".Service2"
            android:enabled="true"
            android:exported="true" >
        </service>
        <service
            android:name=".Service3"
            android:enabled="true"
            android:exported="true"
            android:permission="Example_Permission">
        </service>
        <service
            android:name=".Service4"
            android:enabled="true"
            android:process=":remote">
            <intent-filter>
                <action android:name="act1" />
                <category android:name="cat1" />
                <data android:scheme="sms" />
                <data android:scheme="smsto" />
            </intent-filter>
        </service>
        <service
            android:name=".Service5"
            android:enabled="true"
            android:exported="false">
            <intent-filter>
                <action android:name="act2" />
                <category android:name="cat2" />
            </intent-filter>
        </service>
    </application>
</manifest>
```

Listing 1: Service Declaration Example

2.1.2 Service Declaration. Generally, each service must have a corresponding `<service>` declaration in the application's `AndroidManifest.xml` (see Listing.1 for an example). Specially, a service can be declared as *local* or *exported* by setting the "`android:exported`" attribute. When `android:exported`="`false`", it ensures that the service is available to only the application providing the service and stops other applications from starting or binding to it, even when using an explicit intent. Otherwise, if `android:exported` = "`true`", the service is exported. Besides, a service can declare one or many `<intent-filter>` to specify the types of intents that the service can respond to. In this case, if the "`android:exported`" attribute is not explicitly set to "`false`", the service is exported as well. Components of other applications may access an exported service. For example, Service1 and Service5 in Listing.1 are local services while Service2, Service3 and Service4 are exported services.

Furthermore, the "`android:permission`" attribute specifies the name of a permission (see Service3 in Listing.1) that a component must have in order to access the service.

For avoiding ambiguity, in the rest of this paper, the word "*Service*" indicates a service declared in `Androidmanifest.xml`, denoted as S_i, which is correspondent to a sub-class of `Service`; while "*Service Instance*" indicates a runtime object of a given service, and we use s_j^i to denote a specific instance of service S_i.

2.2 Service Leak

Generally, a service instance is no longer used and should be freed up by the Garbage Collector(GC) after it is destroyed. However, like other java objects, in some cases, a destroyed service instance might be unexpectedly referenced and thus cannot be freed up by GC as well. Such service instances become "*leaked*".

Definition 2.1 (Leak Service). A service S_A is a leak service if any instance of it is leaked.

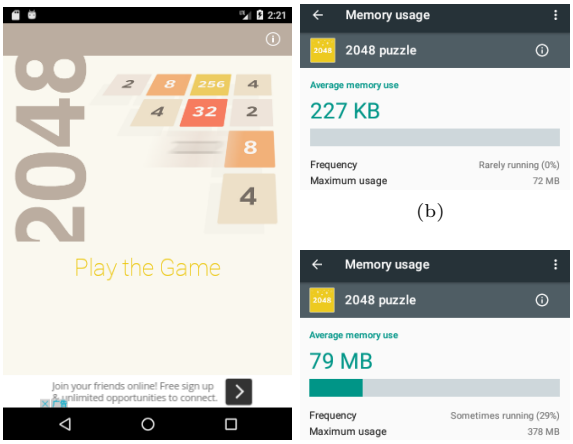
As mentioned, services can be declared as either `local` or `exported`. Once an exported service is started/bound, it might further access other services, instances of which might be leaked as well. To differentiate leak services from services that trigger leak services, we further define "*Leak Entry Service*" as follows:

Definition 2.2 (Leak Entry Service). A service S_A is a leak entry service, if (a) it is exported and (b) there is at least one instance of it causing the leak of at least one service instance (including itself).

For instance, assuming there are four leaked service instances s_1^A, s_1^B, s_1^C and s_1^D, where s_1^A is an exported service instance, s_1^B and s_1^C are created by s_1^A via ICC (e.g., calling `startService()` or `bindService()` with specified `Intent`), s_1^D is created by s_1^C. Then all S_A, S_B, S_C and S_D are leak services, while S_A is the only leak entry service.

Given a leak service, a malicious application may attack the hosting application by flooding the app with service creations, which may lead to some serious results, such as:

- In one common case, the amount of memory used by the hosting application keeps increasing. And if the memory usage finally exceeds the maximum allowed limit, an `Out-of-Memory(OOM)` error would be thrown and the application would crash.
- If a service performs any time-consuming task in the main thread, when massive instances of the service are started, newly started ones might be blocked by previous ones and an `Application Not Responding(ANR)` would be thrown.
- JNI creates references for all Java object arguments passed in to native methods, as well as all objects returned from JNI functions. These references are either *local* (stored in a local reference table) or *global* (stored in the global reference table). For example, the `AMS` service keeps a reference to each Java `ServiceRecord` instance in the global reference table to manage the corresponding service instance. Furthermore, if the number of references exceeds the local/global reference table size limits (e.g., 512 for local reference table and 51200 for global reference table with respects to our experiments), a "`local (or global) reference table overflow`" JNI error would be thrown, leading to app crash or even reboot of the Android system.



(a) (b) (c)

Figure 2: com.siendas.games2048

In each case, a user may encounter unexpected application crashes, ANR notifications or system reboots, which would affect user experience dramatically. Worse yet, users may further uninstall the application, give a low rating and submit a negative comment in the application market, which would further influence the reputation and profit of the company that releases the application.

2.3 Motivating Example

As shown in Fig.2(a), *com.siendas.games2048* is a very famous puzzle 2048 game, and it is available in GooglePlay with a rating of 4.2 and over 50,000 downloads.

Listing.2 shows a code snippet of the `ABServerService`[2] of *com.siendas.games2048*. In the `onCreate()` lifecycle method of `ABServerService`, an anonymous `Timer` is created to execute an anonymous `TimerTask` to log a piece of information indicating the running of the service at a fixed rate 30 seconds. Once an `ABServerService` instance is created, both the

[2]The code is obtained via reverse engineering tools, so it might be a little different from the original source code which is not available.

```
1  public class ABServerService extends Service{
2      public void onCreate(){
3          ...
4          new Timer().scheduleAtFixedRate(new TimerTask(){
5              public void run(){
6                  Log.i("XXX",ABServerService.this.getPacageName()+"
           .ABServerService running");
7              }
8          }, 0L, 30000L);
9      }
10     ...
11 }
```

Listing 2: Snippet of `ABServerService`

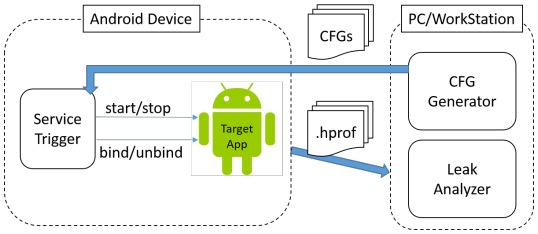


Figure 3: Overview structure of LESDroid

anonymous `Timer` instance and the `TimerTask` instance would hold a reference to the `ABServerService` instance. However, `ABServerService` does not cancel the `Timer` properly (actually, `ABServerService` provides no implementation for either `onPause()` or `onDestroy()`), so the `Timer` instance as well as the `TimerTask` instance would run permanently even after the service is destroyed. In such situation, the corresponding `ABServerService` instance will become leaked. For instance, as shown in Fig.2(b), in normal situation, the average and maximum memory usages of the app are only 227KB and 72MB accordingly; after being repeatedly started/stoped for about half a hour, the values of the average and maximum memory usages increase dramatically to 79MB and 378MB (see Fig.2(c)).

3 OVERVIEW OF LESDROID

.

Considering the complex life-cycle management mechanisms provided by the Android platform, it is not an easy job for developer to discover such leaks when the target application is under development. In this section, we demonstrate the structure of our LESDroid for automated detecting ES-Leaks, i.e., detecting both leak services and associated leak entry services.

As shown in Fig.3, LESDroid consists of three main components: (a) *Configuration Generator (CG)*, (b) *Service Trigger (ST)* and (c) *Leak Analyzer (LA)*. ST runs on the same Android device hosting the application under test (AUT), while Both CG and LA run on a stand-alone PC or workstation. CG analyzes the `Androidmanifest.xml` and decompiled codes of app under test and generates a set of configurations, each of which consists of information needed for creating an `Intent`, for each exported service; ST tries to start/stop and bind/unbind each exported service of the AUT repeatedly with `Intents` created from the configurations generated by CG. After the execution of ST, LA dumps the heap and analyzes dumped `.hprof` files to find out leaked services.

Table 1: Configurations for Service4 in Listing.1

CFGs	act	cat	data	CFGs	act	cat	data
cfg0	NULL	NULL	NULL	cfg6	act1	NULL	NULL
cfg1	NULL	NULL	d1	cfg7	act1	NULL	d1
cfg2	NULL	NULL	d2	cfg8	act1	NULL	d2
cfg3	NULL	cat1	NULL	cfg9	act1	cat1	NULL
cfg4	NULL	cat1	d1	cfg10	act1	cat1	d1
cfg5	NULL	cat1	d2	cfg11	act1	cat1	d2

⋆ Each configuration keeps the package name and service name as well, which are not shown in this table.

3.1 Configuration Generator

Currently, we consider three different strategies for generating configurations:

3.1.1 Intent-Filter-Insensitive (IFI). A basic strategy simply generates exactly one configuration per each service. The configuration contains only the package name of the AUT and the name of the given service.

3.1.2 Intent-Filter-Sensitive (IFS). As mentioned in Sec.2.1.2, a service can declare one or many `<intent-filter>` to declare its capabilities via specifying the types of intents that the service can respond to. At runtime, a service may behavior differently if any of the `action`, `category` or `data` sub-parts of the received `Intent` is different. Observing this fact, given a `<intent-filter>`, which declares a list of actions L_{act}, a list of categories L_{cat} and a list of data L_{data}, IFS tries to generate a configuration per each combination of $L_{act} \times L_{cat} \times L_{data}$. Take Service4 in Listing.1 as an example. Service4 declares an intent filter with L_{act} ={"act1"}, L_{cat} ={"cat1"} and L_{data} ={d1,d2}[3]. We also add a special element `NULL` to each of the three lists and result in L_{act} ={"act1",`NULL`}, L_{cat} ={"cat1",`NULL`} and L_{data} ={d1,d2,`NULL`}. Finally, we obtain 12 different configurations as shown in Table.1.

3.1.3 Intent-Filter-Sensitive extended with Pairwise Boolean Extras (IFS-PB). As boolean extras included in an `Intent` are usually used as path conditions, we try to extend IFS by taking these boolean extras into consideration. We apply a conservative static flow-insensitive analysis to do this.

Specifically, given a service *srv*, we first build a call graph starting from given entrypoint methods of the corresponding service (and possibly of its ancestor in the class hierarchy in case any entrypoint method is not declared by itself): for "start" mode, there are two entrypoint methods `onStart()` and `onStartCommand()`, while for "bind" mode, `onBind()` is the only one entrypoint method. Then we try to identify all fields and local variables (of all methods along the call graph) that are possible aligned to the `Intent` (and the `Bundle` associated with it) passed to these entrypoint methods. Finally, we obtain the key set $S_{Boolean}$ of related boolean extras by further identifying and checking the possible value of the first argument (i.e., a string indicating the key of the extra) for each invocation of the `Intent` API `getBooleanExtra()` and `Bundle` API `getBoolean()`.

If the key set $S_{Boolean}$ is not empty, the final set of configurations is generated on top of IFS. As the size of $S_{Boolean}$ might be large, pairwise testing is applied to $S_{Boolean}$, and $S_{pairwise}$ denotes the set of combinations obtained. Finally, the configuration set S_{cfg}^{IFS-PB} is the Cartesian product of

[3]Here, we use d1,d2 to denote the data declarations on line 29, 30 of Listing.1

Procedure 1 Overall procedure for Service-Trigger

```
1: procedure SERVICE-TRIGGER(pkgname, θ_rpt, mode)
2:     f_manifest ← GET_MANIFEST(pkgname);
3:     L_xpt ← GET_ALL_EXPORTED_SERVICES(f_manifest);
4:     for each service srv in L_xpt do
5:         CHECK(srv, θ_rpt, mode);
6:     end for
7: end procedure
8:
9: procedure CHECK(srv, θ_rpt, mode)
10:    S_cfg ← GETCFGS(srv, f_manifest);
11:    for each configuration cfg in S_cfg do
12:        intent ← BUILDINTENT(cfg);
13:        t ← 0 ;
14:        while t< θ_rpt do
15:            if mode="start" then
16:                start service srv with intent;
17:                stop service srv;
18:            else if mode="bind" then
19:                bind service srv with intent;
20:                unbind service srv;
21:            end if
22:            t++;
23:        end while
24:        //notify LA to dump the heap of corresponding process
25:        DUMPHEAP( );
26:    end for
27: end procedure
```

Procedure 2 Overall procedure for Leak Analyzer

```
1: procedure LEAK-ANALYZER(cfg, f_heap)
2:     C_SRV: a map for counting leaked instances of each service
   type;
3:     L_Report: a list storing report tuples;
4:     L_srv_all ← GETALLSRVINS(f_heap);
5:     for each s ∈ L_srv_all do
6:         if ISLEAKED(s) then
7:             C_SRV[s.type] ← C_SRV[s.type]+1;
8:         end if
9:     end for
10:    for each srv in C_SRV.keyset() do
11:        if C_SRV[srv]> 0 then
12:            add tuple "⟨cfg, srv, C_SRV[srv]⟩" to L_Report;
13:        end if
14:    end for
15:    return L_Report;
16: end procedure
17:
18: procedure ISLEAKED(s)
19:    if ISDESTORYED(s) then
20:        L_P← GETALLPATHSTOGCROOT(s);
21:        L_P← FILTERPATHS(L_P);
22:        if L_P is empty then
23:            return false;
24:        end if
25:        return true;
26:    else
27:        return false;
28:    end if
29: end procedure
```

S^{IFS}_{cfg} and $S_{pairwise}$, where S^{IFS}_{cfg} denotes the set of configurations generated by IFS.

3.2 Service Trigger

Procedure.1 shows the overall procedure of ST. It first obtains the list of all exported services of the AUT by parsing its `AndroidManifest.xml` (line 2-3). Then, for each exported service *srv*, ST would check it by calling `check()`. In `check()`, ST first parses the `<service>` part of the service and gets a set of configurations from CG (line 10). After that, with respects to the value of the parameter *mode*, ST repeats starting/stoping or binding/unbinding each of the exported service multiple times (with each configuration) determined by the corresponding threshold θ_{rpt} accordingly (line 13-23). Finally, for each configuration, ST notifies LA to dump the heap of the process in which the service runs. When ST starts or binds to a service *srv* with a given configuration *cfg* (line 16, 19 of Procedure.1), it creates an instance of `Intent`, explicitly sets the component name to the name of *srv* and set action, category and data parts of the intent accordingly (when each corresponding field of *cfg* is not `NULL`).

3.3 Leak Analyzer

Leak Analyzer(LA) is responsible for identifying and counting leaked service instances remained in each dumped `.hprof` file f_{heap}. The overall procedure of LA is shown in Procedure.2. For each dumped `.hprof` file f_{heap}, LA first finds out the set of all service instances L_{srv_all} (line 4). Then, for each instance in L_{srv_all}, LA checks whether it is leaked(via calling the procedure `isLeaked()`), and counts leaked instances for each service found in the heap file (line 5-9). Finally, LA generates a report tuple for each service *srv* in $C_{SRV}.keyset()$ if $C_{SRV}[srv]> 0$. The tuple is of the following form: $\langle cfg, srv, count\rangle$, where cfg indicates the configuration for building the `Intent` used by ST to access the service (see Procedure.1 line 14-23), *srv* indicates a leak service, and *count* stands for the number of leaked instances of *srv* found in the heap file.

Given a report tuple $\langle cfg, srv, count\rangle$, then the service *srv* is a leak service, and the configuration cfg is a leak entry configuration, while the corresponding service $cfg.service$ is judged as a leak entry service for the leak service *srv*.

The procedure `isLeaked()` checks whether a service instance *s* is leaked. It first determines whether *s* is destroyed or not by checking whether there is a reference chain from an `ActivityThread` instance to it via the *mServices* field. If there is no such a chain, *s* is judged as destroyed. Then, for a destroyed service instance *s*, the procedure collects the set of all paths from *s* to every GC root. After that, `filterPaths()` is called to filter paths that should not be view as leaks. If any path exists after `filterPaths()`, LA judges *s* as leaked and we call each of the remaining path a "*L-path*". Specifically, `filterPaths()` first excludes all paths consisting of any object of type `FinalizerReference`, `SoftReference`, `WeakReference` or `PhantomReference`. Besides, as leaks may be caused by the Android SDK and there is little a developer can do to fix them, `filterPaths()` also ignores such leaks by excluding paths that match any known leak pattern defined in the `AndroidExcludedRefs.java`[4] provided by LeakCanary.

4 IMPLEMENTATION AND EVALUATIONS

In our prototype, we utilize Apktool[5] to decode each .apk file to obtain the corresponding `Androidmanifest.xml`.

[4]https://github.com/square/leakcanary/blob/master/leakcanary-android/ src/main/java/com/squareup/leakcanary/AndroidExcludedRefs.java, accessed Nov.20, 2016

[5]https://ibotpeaches.github.io/Apktool/

Table 2: Emulator configuration

Feature Name	Feature Value
CPU/ABI	Intel Atom(X86_64)
Device Ram Size	2GB
Sdcard size	2GB
Android API Version	Marshmallow (23)

We use Soot[6] to implement static analysis of bytecode and use PICT[7] to generate $S_{pairwise}$ used in IFS-PB strategy. Service Trigger is implemented as an Android app running on the same emulator or device that hosting the AUT. For starting/stoping or binding/unbinding a service, ST directly invokes the `startService()` / `stopService()` or `bindService()` / `unBindService()` API accordingly[8]. Leak Analyzer is built on top of AndroMAT [2].

We evaluate LESDroid on real-world Android apps. All the experiments are done on a Dell workstation with Win10 as OS, 32GB RAM and Intel Xeon E5-1650 CPU. Android 6.0 Marshmallow, one of the most popular versions installed on 32.3% of all devices [4], is used in our experiments. We create a fresh emulator (with configuration shown in Table.2) for each app, and run totally 5 emulators simultaneously to speed up the overall running process. We want to answer following research questions via the experiments:

- **RQ-1:** How widely are services and, especially, exported services used in Android apps?
- **RQ-2:** What are the main factors preventing LESDroid from accessing exported services? How would the three different configuration generation strategies affect the accessibility?
- **RQ-3:** Can LESDroid detect leak services and their associated leak entry services effectively and efficiently?
- **RQ-4:** What consequences would exported service leaks lead to?

4.1 Data Set Collection

A collection of 41,537 `.apk` files were clawed from GooglePlay from July 31st, 2015 to March 3rd, 2016. We analysed the `AndroidManifest.xml` of each `.apk` and found 74,831 services provided by 28,662 (about 69%) `.apk`. We filtered out those declaring no exported service, retaining only 3,934 `.apk`. Then we came up with the answer to **RQ-1**: *With respects to our collection of `.apk` files, services are widely used in about 69% Android apps, over 13.7% of which provide exported services.*

We manually reviewed the names and corresponding exported services of the remaining `.apk` files, finding that (1) many `.apk` files were actually different releases of the same application, and (2) some `.apk` files could be clustered as the exported services of them share the same naming pattern, despite that they belongs to different apps (most `.apk` files of this kind were WallPaper apps). Thus, we randomly selected exact one `.apk` file per each application and each cluster, resulting a data set consisting of 2,557 apps. Among them, 1,109 could not be installed on the emulator[9]. To remove toy apps, we manually checked the remaining 1,348 apps on GooglePlay and only selected apps that were both popular ($\#_{download} \geq$10,000) and high rated (rating $\geq$ 4.0). Finally, 375 apps remained and formed the data set for our evaluations (See Tab.3).

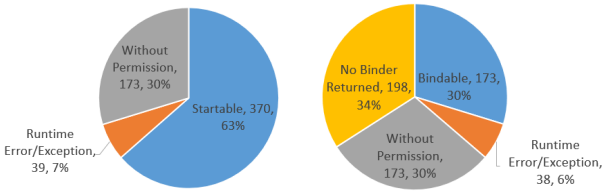
[6]http://sable.github.io/soot/
[7]https://github.com/microsoft/pict
[8]ST can bind a bound service without having the class definition of the returned `Ibinder` instance, as ST does not invoke any method provided by the `Ibinder` instance.
[9]Many `.apk` files were compiled for armv7 only while the emulator in our experiment used the Intel architecture.

Table 3: Statistics of selected apps

	Apk Size	Rating	#download	#Exported Service
min	73KB	4.0	10K∼50K	1
max	48,707KB	4.8	500,000K ∼ 1,000,000K	19
median	4,772KB	4.20	100K∼500K	1
avg	8,082KB	4.24	–	1.55
total	–	–	–	582

Table 4: Configurations generated by different strategies (per service)

	Start				Bind			
	min	max	avg	total	min	max	avg	total
IFI	1	1	1	582	1	1	1	582
IFS	1	9	2.29	1,335	1	9	2.29	1,335
IFS-PB	1	30	2.67	1,553	1	15	2.33	1,355



(a) "start" mode (b) "bind" mode

Figure 4: Service accessibility

We applied LESDroid to the 375 apps with two modes={"start","bind"}, $\theta_{rpt} = 10$ and three different strategies for generating configurations {IFI, IFS, IFS-PB}. Tab.4 shows the number of configurations generated under different settings. IFI generates exact one configuration per each service, and IFS generates same number of configurations for each service in both "start" and "bind" modes. The number of configurations generated by IFS-PB for a given service may be different in different modes, as boolean extras are obtained via different entrypoint methods in the two modes accordingly. Besides, as there are only a few services involving boolean extras (especially in "*bind*" mode), the number of configurations generated by IFS-PB is not much larger than that generated by IFS.

4.2 Service Accessibility

As shown in Fig.4, among the 582 exported services in our data set, there are only 370 services that can be started by LESDroid with at least one of the three strategies, while the number of bindable services[10] is only 173. The main reasons for unaccessible services are three folds: 1) lack of permissions, 2) runtime errors/exceptions, and 3) intentional un-bindable.

4.2.1 Lack of Permissions. As mentioned in Sec.2.1.2, a developer could protect an exported service from unauthorized access by specifying the '`android:permission`" attribute of the service in the `Androidmanifest.xml` file. For example,

[10]Here, we say a service is bindable if ST could successfully obtain a `IBinder` instance from the service after calling the `bindServive()` API.

Table 5: Accessible applications, services and configurations under different experimental settings

	Start			Bind		
	IFI	IFS	IFS-PB	IFI	IFS	IFS-PB
# app with accessible service	247	248	248	119	121	122
# accessible service	359	366	370	170	172	173
# accessible cfg	359	790	975	170	369	391

Table 6: Statistics of evaluation results

	Start			Bind		
	IFI	IFS	IFS-PB	IFI	IFS	IFS-PB
# app with leak	61	63	64	41	46	46
# leak entry service	86	90	91	42	54	54
# leak entry cfg	86	206	242	42	107	107
# leak report tuple	89	214	261	47	108	108
# leak service ⋆	83	87	88	43	50	50

⋆ Leak services detected in different apps are considered as different, even if they share the same name.

183 services in our data set are protected by 27 different permissions (including 12 system permissions as well as 15 permissions defined by applications).

We built ST with these 27 permissions, which enabled ST to successfully access 10 permissions-guarded services, including 6 services guarded by `READ_EXTENSION_DATA` system permissions and 4 services guarded by 3 different permissions defined by applications. We inspected the manifest files of the apps providing these 10 accessible services and found that all of the 6 accessible services guarded by `READ_EXTENSION_DATA` system permission belonged to a single app, which redefined the `READ_EXTENSION_DATA` permission in its manifest file. Furthermore, the redefined system permission together with other three application defined permissions were specified with a protection level of "normal". Therefore, other applications (e.g., LESDroid) can access these services.

4.2.2 Runtime Error/Exceptions. There were nearly forty services (about 7%) that cannot be successfully started or bound to, as runtime errors or exceptions were encountered when LESDroid tried to access them. Most of these errors/excpetions (e.g., `NullPointerException`, `ClassNotFoundException`, `IllegalArgumentException`, `NumberFormatException`, etc.) were thrown because the intents generated by LESDroid to access these services did not contain valid extras required by them and the services themselves did not provide proper mechanisms for handling these errors/exceptions. Services without elegant mechanisms for handling such runtime errors/exceptions are vulnerable to DoS attacks. However, they are out of the scope of this paper. In this paper, we only focus on detecting leak services and their associated leak entry services.

4.2.3 Intentional un-bindable Services. Besides, as shown in Fig.4(b), in our data set, 198 services were intentionally implemented as un-bindable where no `IBinder` instance would be returned in the corressponding `onBind()` methods. As a result, the number of bindable services and the number of apps hosting bindable services are much smaller than those of startable services and corresponding hosting apps.

Tab.5 shows the statistics of accessible services as well as associated applications and configurations under different settings. Then, the answer to **RQ-2** is: *Lack of permissions, runtime errors/exceptions, and intentional un-bindable services are the main factors preventing LESDroid from accessing exported services. By generating configurations with more detailed information, IFS and IFS-PB could try more execution paths to bypass some runtime errors/exceptions and thus improve the numbers of accessible services (e.g., 11 more services could be started with IFS-PB than with IFI).*

4.3 Detected Leaks

As shown in Tab.6, IFS and IFS-PB outperformed IFI on accessing more services and detecting more leaks, and IFS-PB performed the best. In both modes, with respects to our experiments, the set of leak services and the corresponding set of leak entry services detected by LESDroid with IFI are strictly covered by those detected with IFS, which are further strictly covered by those detected with IFS-PB. The difference between IFS and IFS-PB in detecting leaks is not significant. For example, in "*start*" mode, IFS-PB detects only one more leak service (together with its associated leak entry service) than IFS does; while they perform the same in "*bind*" mode. As the results provided by LESDroid with IFS-PB cover those obtained with IFI and IFS, in the following discussions, we only focus on results obtained via IFS-PB.

In most cases, leak services are actually the leak entry services of themselves. For example, out of the 88 and 54 leak services detected by LESDroid with IFS-PB in "*start*" and "*bind*" modes (see Tab.6), 85 and 46 are the entry services of themselves. There are only three entry services in "*start*" mode and one entry service in "*bind*" mode which cause leaks of multiple services. Meanwhile, there are also three leak services in "*start*" mode and two leak services in "*bind*" mode whose leaks can be triggered through more than one entry service.

Given a target app, let $L_{Report}(cgs)$ denote set of all report tuples obtained via the configuration generation strategy indicated by cgs. Then, for a given detected leak service S_l, we use $\gamma(S_l, cgs)$ to denote the maximum number of leak instances of the leak service S_l detected by LESDroid with the strategy indicated by cgs, i.e., $\gamma(S_l, cgs) = max\{t.count | t \in Ł_{Report}(cgs)$ and $t.srv = S_l\}$. And, for a leak entry service S_{le}, we use $\pi(S_{le}, cgs)$ to denote the maximum number of leak instances of any leak service S' detected by LESDroid with the strategy cgs, where S' is a leak service triggered by S_{le}, i.e., $\pi(S_{le}, cgs) = max\{t.count | t \in Ł_{Report}(cgs)$ and $t.cfg.srv = S_{le}\}$. Then, Tab.8 shows the distribution of $\gamma(S_l$,IFS-PB) for all leak services S_l detected by LESDroid with IFS-PB strategy, while Tab.9 shows the distribution of $\pi(S_{le}$,IFS-PB) for all detected leak entry services S_{le}, accordingly.

Services behave different in "*start*" and "*bind*" modes. As a result, there are 46 leak services as well as 44 leak entry services can only be detected in "*start*", while there are only 7 leak services and 7 leak entry services can only be detected in "*bind*" mode. Even in the 44 leak services and 47 leak entry services (see the blue bold digits in Tab.8 and Tab.9) that are detected in both modes , there are a few services showing different $\gamma(S_l$,IFS-PB) or $\pi(S_{le}$,IFS-PB). Take the `SipService` service (which is actually a leak service and a leak entry services of itself at the same time) of

Table 7: Processing time of LESDroid with IFS-PB (unit: ms)

	CG	ST	LA	Total
min	1	8,021	9	1,854
max	531,095	392,232	354,437	867,688
median	13,700	17,501	3,308	37,891
avg	26,830	26,127	7,357	60,313

Table 8: Distribution of $\gamma(S_l$, IFS-PB) for all leak services detected by LESDroid with IFS-PB

		$\gamma(S_l$, IFS-PB) : **Start**									total
		0	1	2	3	4	5	...	9	10	
$\gamma(S_l$, IFS-PB) : **Bind**	0	-	14	4	2	-	1	...	1	24	46
	1	5	**23**	-	-	**1**	-	...	-	**1**	30
	2	-	-	**5**	-	-	-	...	-	-	5
	3	-	-	-	-	-	-	...	-	-	0
	4	-	-	-	-	**1**	-	...	-	-	1
	...	...	...	...	...	...	...	...	...	...	...
	8	1	-	-	-	-	-	...	-	-	1
	9	-	-	-	-	-	-	...	-	-	0
	10	1	-	-	-	-	-	...	-	**13**	14
total		7	37	9	2	2	1	...	1	38	97

Table 9: Distribution of $\pi(S_{le}$, IFS-PB) for all leak entry services detected by LESDroid with IFS-PB

		$\pi(S_{le}$, IFS-PB) : **Start**									total
		0	1	2	3	4	5	...	9	10	
$\pi(S_{le}$, IFS-PB) : **Bind**	0	-	13	4	2	-	1	...	1	23	44
	1	5	**25**	-	-	**1**	-	...	-	**2**	33
	2	-	-	**5**	-	-	-	...	-	-	5
	3	-	-	-	-	-	-	...	-	-	0
	4	-	-	-	-	**1**	-	...	-	-	1
	...	...	...	...	...	...	...	...	...	...	...
	8	1	-	-	-	-	-	...	-	-	1
	9	-	-	-	-	-	-	...	-	-	0
	10	1	-	-	-	-	-	...	-	**13**	14
total		7	38	9	2	2	1	...	1	38	98

`com.aastra.aastramc.gui` for example, there are five leak instances of it found in "*start*" mode , while only one leak instance found in "*bind*" mode.

LESDroid is efficient as well. As shown in Tab.7, it takes LESDroid (with IFS-PB) about one minute to process an app on average, while the maximum processing time is less than 15 minutes. The average proportions of processing time consumed by CG, ST and LA are 39.6%, 50.9% and 9.5% accordingly.

We conclude the answer to **RQ-3** as: *Even with the simplest IFI strategy, LESDroid could detect 83 and 43 leak services as well as 86 and 42 leak entry services in* "start" *and* "bind" *modes accordingly. LESDroid with IFS-PB performed the best and it successfully detected 98 leak entry services (16.8%) from 70 different apps (18.7%) in the data set consisting of 375 apps. The results indicate that the problem of exported service leak is common among Android apps.*

4.4 Consequences of Leaks

To see what consequences caused by leak services and leak entry services would be, we further conducted an experiment for stress testing those leak entry services detected in the first experiment. Specifically, in this experiment, we applied LESDroid to the 98 distinct leak entry services detected in the first experiment. We tested a leak entry service, which was detected in both modes in the first experiment, twice in different modes accordingly. As a result, 91 leak entry services in "*start*" mode and 54 leak entry services in "*bind*" mode

Table 10: Distribution of memory growth for all leak entry services detected by LESDroid with IFS-PB

			Memory Growth (unit:MB)									total
			$[0,1]$	$(1,2]$	$(2,4]$	$(4,8]$	$(8,16]$	$(16,32]$	$(32,64]$	$(64,128]$	$(128,\infty)$	
$\pi(S_{le}$, IFS-PB)	**Start**	1	12	2	-	1	6	2	2	9	4	38
		2	2	-	-	-	1	-	2	3	1	9
		3	-	-	-	-	-	-	-	-	2	2
		4	-	-	-	-	1	-	-	-	1	2
		5	-	-	-	-	-	-	1	-	-	1
		9	-	-	-	-	-	1	-	-	-	1
		10	-	-	-	1	5	9	10	7	6	38
		total	14	2	-	2	13	12	15	19	14	91
	Bind	1	10	8	1	-	4	2	3	3	2	33
		2	1	-	1	-	-	1	-	1	1	5
		4	-	-	1	-	-	-	-	-	-	1
		8	-	-	-	-	-	-	-	-	1	1
		10	-	-	-	-	-	1	6	2	5	14
		total	11	8	3	-	4	4	9	6	9	54

were tested. Unlike setting $\theta_{rpt} = 10$ in the first experiment, we instead let LESDroid run each leak entry service for one hour, or the number of start/bind calls exceeds 512,000 (i.e., 10X of the size of the JNI global reference table), or until any error/exception took places and the corresponding process dies. For each leak entry service S_{le}, we chose only one configuration which leaded to $\pi(S_{le}$, IFS-PB) in the first experiment to run [11]. As a comparison, we also started/bound each leak entry service exactly one time with the same configuration for a hour. In both situations, we logged the memory usages via the "`dumpsys meminfo`" shell cmd. Finally, for each leak entry service, we obtained the maximum memory usages in both situations from the logs. Tab.10 demonstrates the memory usage growths (i.e., difference between the maximum memory usages in the two situations) for all leak entry services.

As shown in Tab.10, most leak entry services lead to massive memory usages. For example, the memory usage growths brought through 60 leak entry services in "*start*" mode and 28 leak entry services in "*bind*" mode are over 16MB memory, while the maximum memory growth is almost 500MB. At the same time, there are in total 40 leak entry services (including 18 in "*start*" mode and 22 in "*bind*" mode) do not bring memory growths greater than 8MB. We found that 39 of the 40 leak entry services have $\pi(S_{le}$, IFS-PS) values less than 5, and 34 of them have a $\pi(S_{le}$, IFS-PS) $= 1$, indicating that the leaked service instances caused by them are limited, which could be an explanation for the low corresponding memory growths.

Only six (including two whose running processes died only after tens of rounds), among the 55 leak entry services with $\pi(S_{le}$, IFS-PS) $>= 5$ (including 40 in "*start*" mode and 15 in "*bind*" mode), have brought less than 16MB memory growths. At the same time, there are a few leak entry services with lower $\pi(S_{le}$, IFS-PS) bringing massive memory growths as well. For example, there are six leak entry services (with $\pi(S_{le}$, IFS-PS) $= 1$) having memory growth larger than 128MB. Despite these massive memory growths are brought through leak entry services, the real sources of memory leaks might be some other resources accessed by the leak entry

[11] If multiple configurations leading to $\pi(S_{le}$, IFS-PB), we randomly chose one.

Table 11: Reasons for dead processes within one hour

Reason	start	bind
Application Not Responding	17	7
Global Reference Table Overflow	9	4
Out Of Memory	9	6
NullPointerException	2	2
SQLiteCantOpenDatabaseException	1	-
RejectedExecutionException	1	-
RuntimeException	2	-
Signal-6	2	1
Signal-9	1	2
Total	44	22

services. It is left for our future work to detect these leak sources. Furthermore, 63 out of the 98 distinct leak entry services have brought over 16MB more memory[12].

The memory growth should be more serious than shown in Tab.10, as 47 out of the 98 distinct leak entry services have brought deaths to 66 associated processes before the time limit[13]. Tab.11 shows the reasons for process deaths and the affected numbers of leak entry services. The top three reasons for dead processes are `ANR`, `OOM`, and `Global Reference Table Overflow`, causing the deaths of 52 processes. All of the remaining 14 dead processes did not die immediately when the service was accessed for the first time but after tens to thousands rounds of launches. Specially, one process died from `RejectedExecutionException` as the app exceeded the number of tasks that can be queued by the `ThreadPoolExecutor`. Furthermore, there were six processes died from different signals (e.g., Signal-6, Signal-9, etc.) without any explicit trace of the causes in the log.

Furthermore, in our experiment, the intervals between two consecutive rounds of starting/binding for different apps were different. As a result, some apps could finish over 100,000 rounds, while others could only finish less than 10,000 rounds without death. This could be another possible explanation for the low memory growths brought by some of these leak entries.

Finally, the answer to **RQ-4** is: *Most detected exported service leaks lead to growths in memory usages in the first place. Leak services with high $\pi(S_{le}, IFS\text{-}PS)$ values tent to bring more memory usages. Furthermore, apps with exported service leaks are vulnerable to service leak attacks, which may bring kinds of runtime exceptions/errors that would not take place in normal use to the target apps.*

5 DISCUSSION

5.1 Threats to Validity

Like any empirical study, there are potential threats to the validity of the results of our experiments.

First, our results will not necessarily generalise beyond the 375 commercial apps to which we have applied LESDroid. Second, we only evaluated LESDroid on the Marshmallow version (23) of Android platform, and our approach may fail if mechanisms for managing services change in subsequent versions. Third, leak services would be one, but may not be the unique one, cause of the massive memory usages shown in Sec.4.4. They might be caused by leaks of some objects other than services. But it is clear that these leaks can be triggered by LESDroid via starting/stopping or binding/unbinding given exported entry services. Fourth, exported services can be accessed with or without the target app running in the front, which may affect the results of our experiments. To overcome this threat, in our experiments, we ran LESDroid for both situations where the AUT is (1) hidden in the background and (2) running in the front. In both situations, we detected exported ES-leaks, but there were fewer detected ES-leaks in the second situation. One possible explanation for this difference might be that the AUT running in the front could bind to services by itself and prevent them from being destroyed. However, for the sake of limited space, we only show the results where the target apps running in the background in this paper.

[12]If a leak entry service found in both "*start*" and "*bind*" modes, its growth is set as the maximum of the two growths found separately in different modes

[13]There are 14 leak entry services whose corresponding processed died in both "*start*" and "*bind*" modes.

5.2 Service Protection

To avoid exported services leak, a developer should keep his/her services private unless necessary. And for an exported service, a developer should protect it by specifying given permission with a protection level "`signature`" or higher. For those services exported without proper permission protection, a developer should be careful in implementing not only the service but also all other components which are accessible through the service. A developer may use LESDroid to identify leak services and the associated (unprotected) leak entry services and further fix the app to avoid such leaks.

5.3 Limitations

Currently, LESDroid can detect leak services whose leak is triggered in given life-cycle APIs (i.e., `onCreated()`, `onStart()`, `onStartCommand()`, `onBind()`, `onUnBind()`, `onDestroy()`, etc.) of some exported services. It is not able to detect leaks introduced by methods provided by the `IBinder` interface of a bindable service. Besides, the current configuration generation strategies implemented by LESDroid (i.e., IFI, IFS and IFS-PB) are not effective enough to explore paths that start from the three entry methods `onStart()`, `onStartCommand()` and `onBind()` and require complex path conditions involving values provided by other payloads included in the `Intent` argument. Although, LESDroid can detect exported service leaks, which can be view as signs of memory leaks, yet it cannot determine other implicit sources of memory leaks; besides, LESDroid fails to report further details on the objects that are the root causes of detected leak services, e.g, the `Timer` object in the `ABServerService` in Listing.2.

6 RELATED WORK

6.1 Android Service Testing

There are few works on testing Android services [7] [28]. Buzzer [7] provides an input validation vulnerability scanner for Android devices. It fuzzes all the Android system services by sending requests with malformed arguments to them. Snowdrop [28] is a testing framework that systematically identifies and automates background services in Android apps.

Our work differs form them in that we focus on detecting leak services accessible through exported services in Android apps. While Buzzer [7] tries to detecting vulnerabilities of different Android system services, and Snowdrop [28] tries to

provide a systematic testing framework for Android services and combine different techniques (e.g., symbolic execution, NLP-based heuristic) to generates test inputs to maximize service code path coverage. The input generation part of Snowdrop can be applied to LESDroid to obtain a better configuration generation strategy.

6.2 Inter-Component Communication Vulnerability Detection

There are massive works on Inter-Component Communication (ICC) vulnerability detection. On the one hand, many approaches relied primarily on static analysis [8] [18] [14] [20] [5]. For example, IC3 [18] extracts information about Intents in a flow-sensitive manner, while FlowDroid [5] applies a static taint analysis to identify flows or privacy leakages from sensitive Android API sources and sinks. On the other hand, some approaches rely purely on dynamic analysis [7] [9] [12]. For example, Stowaway [9] dynamically detects permission over privilege. IntentDroid [12] dynamically explores an app's Intent interface to identify vulnerabilities.

Generally speaking, pure static approaches fail in determining program paths and the corresponding Intents needed to execute them; while pure dynamic approaches cannot analyze non-executed codes which may contains potential ICC vulnerabilities. So, there are a variety of works combining both static and dynamic analysis to detect ICC vulnerabilities [29] [21] [23]. For example, AppCaulk [23] combines both static and dynamic analysis to detect and prevent data leaks.

Unlike most works focusing on identifying vulnerabilities in Android apps, one recent interesting work LetterBomb [10] goes a step further. It attempts to determine exploitability of those detected vulnerabilities. It provides an approach for automatically generating exploits for Android apps, relying on a combined path-sensitive symbolic execution-based static analysis, and the use of software instrumentation and test oracles.

Exported service leaks can be viewed as a special type of ICC vulnerability, through which a malicious client can intentionally magnify the consequences of leaking and hurt the availability of the service's owner app. Besides, the idea of LetterBomb's exploit generation technique can be applied to LESDroid to generate valid configurations as well.

6.3 Memory and Resource Leak Detection

Relda2 [22] provides a light-weight static tool for analyzing potential resource leaks caused by missing release operations for resources. [26] [27] proposed a systematic GUI model-based approach for testing resource leaks for Android apps. It generates comprehensive test cases to trigger repeated executions of neutral cycles (special sequences of GUI events) to discover resource leaks in apps. [19] goes a step further by prioritizing test cases according to their likelihood to cause memory leaks in a given test suite.

There have been many works on detecting memory leaks in Java programs. LeakBot [17] detects memory leaks by formulating structural and temporal properties of reference graphs. [24] [25] try to optimize the static analysis to produce precise leak reports based on some observations about containers/loops. An interesting project, Leak Pruning [6], tries to bound the memory consumption of Java programs with leaks by pruning likely leaked data structures when a program runs out of memory.

General tools for analyzing memory(e.g., MAT [16], AndroMat [2], HAHA [11]) simply report objects or set of objects which are suspiciously big as leak suspects, and do not provide specific supports for detecting leaked components of Android; besides, whether the leaks can be detected and how long the process would take depend highly on the experience of the developer/tester. LeakCanary [13], built on top of HAHA, is a open source memory leak detection library specific for Android. However, to use LeakCanary, a developer needs to modify part of the source codes (required code injection/modification might be missed and tedious) and add given building dependencies into the `build.gradle` file of the app; besides, manual interactions are required to run the app. LeakDAF [15] makes use of UI testing technique to execute automatically the app under test, and applies memory analysis technique to inspect dumped heap files to identify leaked activities and fragments based on Androids mechanisms for managing them.

Compared to the work mentioned , LESDroid is an automated tool specifically for detecting leak services and leak entry services of Android apps. Unlike LeakCanary working as a library, LESDroid works as a stand-alone tool and can be directly applied to any app released in markets without requiring the source code.

7 CONCLUSION AND FUTURE WORK

This paper proposed LESDroid, an automated tool for the detection of exported service leaks in an Android app. LESDroid applies a static analysis to generate a collection of configurations for creating `Intent`s. Then it tries to reveal leaks by starting/stoping or binding/unbinding each exported service of the app under test repeatedly with the `Intent`s as ICC arguments. Finally, LESDroid analyzes heap snapshots for identifying leaked service instances. Without requiring source code, LESDroid not only benefits developers/testers, but also provides app market administrators a simple way for evaluating memory leaks of published apps. We applied LESDroid to 375 commercial apps and detected 97 distinct leak services as well as 98 distinct leak entry services from 70 apps.

In the future, we plan to learn lessons from symbolic execution and exploit generation techniques to build more effective strategies for generating configurations. Besides, we also plan to detect leaks that only happen if given methods, provided by the `IBinder` interface obtained when a service is bound, are invoked (in some special sequences). Furthermore, we want to extend LESDroid to a more general situation, where an exported service may lead to memory leak even without any leaked service instance.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (Grant No. 2017YFB1001801), National Natural Science Foundation (Grant Nos. 61690204, 61472174) of China, and the Collaborative Innovation Center of Novel Software Technology and Industrialization.

REFERENCES

[1] 2017. Android Service Guide. (2017). Retrieved Aug. 7, 2017 from https://developer.android.com/guide/components/services.html

[2] AndrodMat 2016. AndroMAT. (2016). Retrieved July 31, 2016 from https://bitbucket.org/joebowbeer/andromat/overview

[3] Android Services API 2017. Android Service API Reference. (2017). Retrieved Aug. 7, 2017 from https://developer.android.google.cn/reference/android/app/Service.html

[4] Android Versions 2017. Platform Versions of Android. (2017). Retrieved August 8, 2017 from https://developer.android.com/about/dashboards/index.html

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[6] Michael D. Bond and Kathryn S. McKinley. 2009. Leak Pruning. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 277–288.

[7] Chen Cao, Neng Gao, Peng Liu, and Ji Xiang. 2015. Towards Analyzing the Input Validation Vulnerabilities Associated with Android System Services. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015)*. ACM, New York, NY, USA, 361–370.

[8] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys '11)*. ACM, New York, NY, USA, 239–252. https://doi.org/10.1145/1999995.2000018

[9] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, New York, NY, USA, 627–638. https://doi.org/10.1145/2046707.2046779

[10] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. 2017. Automatic Generation of Inter-component Communication Exploits for Android Applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 661–671. https://doi.org/10.1145/3106237.3106286

[11] HAHA 2016. Headless Android Heap Analyzer(HAHA). (2016). Retrieved July 31, 2016 from https://github.com/square/haha

[12] Roee Hay, Omer Tripp, and Marco Pistoia. 2015. Dynamic Detection of Inter-application Communication Vulnerabilities in Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 118–128. https://doi.org/10.1145/2771783.2771800

[13] LeakCanary 2016. (2016). Retrieved July 11, 2016 from https://github.com/square/leakcanary

[14] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 280–291. http://dl.acm.org/citation.cfm?id=2818754.2818791

[15] Jun MA, Sheng LIU, Shengtao YUE, Xianping TAO, and Jian LU. 2017. LeakDAF: An Automated Tool for Detecting Leaked Activities and Fragments of Android Applications. In *Proceedings of the 41th IEEE International Computer Software and Applications Conference (COMPSAC XIV)*.

[16] MAT 2016. (2016). Retrieved July 11, 2016 from http://www.eclipse.org/mat/

[17] Nick Mitchell and Gary Sevitsky. 2003. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *European Conference on Object-Oriented Programming*. Springer, 351–377.

[18] D. Octeau, D. Luchaup, S. Jha, and P. McDaniel. 2016. Composite Constant Propagation and its Application to Android Program Analysis. *IEEE Transactions on Software Engineering* 42, 11 (Nov 2016), 999–1014. https://doi.org/10.1109/TSE.2016.2550446

[19] Ju Qian and Di Zhou. 2016. Prioritizing Test Cases for Memory Leaks in Android Applications. *Journal of Computer Science and Technology* 31, 5 (2016), 869–882.

[20] Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2015. Analysis of Android Inter-app Security Vulnerabilities Using COVERT. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 725–728. http://dl.acm.org/citation.cfm?id=2819009.2819149

[21] Julian Schütte, Dennis Titze, and J. M. De Fuentes. 2014. AppCaulk: Data Leak Prevention by Injecting Targeted Taint Tracking into Android Apps. In *Proceedings of the 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM '14)*. IEEE Computer Society, Washington, DC, USA, 370–379. https://doi.org/10.1109/TrustCom.2014.48

[22] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang. 2016. Light-Weight, Inter-Procedural and Callback-Aware Resource Leak Detection for Android Apps. *IEEE Transactions on Software Engineering* 42, 11 (Nov 2016), 1054–1076. https://doi.org/10.1109/TSE.2016.2547385

[23] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. 2015. Effective Real-Time Android Application Auditing. In *2015 IEEE Symposium on Security and Privacy*. 899–914. https://doi.org/10.1109/SP.2015.60

[24] Guoqing Xu and Atanas Rountev. 2008. Precise memory leak detection for java software using container profiling. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 151–160.

[25] Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. 2014. LeakChecker: Practical Static Memory Leak Detection for Managed Languages. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 87, 11 pages. https://doi.org/10.1145/2544137.2544151

[26] D. Yan, S. Yang, and A. Rountev. 2013. Systematic testing for resource leaks in Android applications. (Nov 2013), 411–420. https://doi.org/10.1109/ISSRE.2013.6698894

[27] H. Zhang, H. Wu, and A. Rountev. 2016. Automated Test Generation for Detection of Leaks in Android Applications. (May 2016), 64–70. https://doi.org/10.1109/AST.2016.018

[28] Li Lyna Zhang, Mike Chieh-Jan Liang, Yunxin Liu, and Enhong Chen. 2017. Systematically Testing Background Services of Mobile Apps, In ASE (International Conference on Automated Software Engineering). https://www.microsoft.com/en-us/research/publication/systematically-testing-background-services-mobile-apps/

[29] Yajin Zhou and Xuxian Jiang. 2013. Detecting Passive Content Leaks and Pollution in Android Applications. In *In Proceedings of the 20th Annual Symposium on Network and Distributed System Security, NDSS 13*.