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@nju.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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPC '18, May 27–28, 2018, Gothenburg, Sweden  
© 2018 Association for Computing Machinery.  
ACM ISBN 978-1-4503-5714-2/18/05 ... $15.00  
https://doi.org/10.1145/3196321.3196336

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 Androidmanifest.xml and bytecodes of exported services of the target app to generate Intents. The generated Intents 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 lead 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](image)

Listing 1: Service Declaration Example

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.servicetest"
    android:versionCode="1">
    <application ...>
        <service>
            <intent-filter>
                <action android:name="act1" />
            </intent-filter>
            <category android:name="cat1" />
            <data android:scheme="smsto" />
            <data android:scheme="sms" />
            <android:enabled="true" />
            <android:exported="true" />
            <android:process=":remote" />
            <android:permission="Example_Permission" />
        </service>
        <service>
            <intent-filter>
                <action android:name="act2" />
            </intent-filter>
            <category android:name="cat2" />
            <data android:filter="sms" />
            <data android:filter="smsto" />
            <android:enabled="true" />
            <android:exported="true" />
            <android:process=":remote" />
            <android:permission="Example_Permission" />
        </service>
        <service>
            <intent-filter>
                <action android:name="act3" />
            </intent-filter>
            <category android:name="cat3" />
            <data android:filter="sms" />
            <data android:filter="smsto" />
            <android:enabled="true" />
            <android:exported="true" />
            <android:process=":remote" />
            <android:permission="Example_Permission" />
        </service>
    </application>
</manifest>
```
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 \( s^A_1, s^B_1, s^C_1 \) and \( s^D_1 \), where \( s^A_1 \) is an exported service instance, \( s^B_1 \) and \( s^C_1 \) are created by \( s^A_1 \) via ICC (e.g., calling \( \text{startService()} \) or \( \text{bindService()} \) with specified \( \text{Intent} \)), \( s^D_1 \) is created by \( s^A_1 \). 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.

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 Google Play 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 on\( \text{Create}() \) 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.
Listing 2: Snippet of ABServerService

```
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()++)
7                 }, 0L, 30000L);
8             }
9         }
10     }
```

3.1 Configuration Generator

Currently, we consider three different strategies for generating configurations:

3.1.1 Intent-Filter-Inensitive (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. 3.1.2, 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 behave 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 $L_{data} = \{d1,d2\}$. We also add a special element NULL to each of the three lists and result in $L_{act} = \{"act1",\text{NULL}\}$, $L_{cat} = \{"cat1",\text{NULL}\}$ and $L_{data} = \{d1,d2,\text{NULL}\}$. 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 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 $S_{IFS-PB}$ is the Cartesian product of $S_{IFS}$ and $S_{pairwise}$.

Table 1: Configurations for Service4 in Listing.1

<table>
<thead>
<tr>
<th>CFGs</th>
<th>act</th>
<th>cat</th>
<th>data</th>
<th>CFGs</th>
<th>act</th>
<th>cat</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfg0</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>cfg6</td>
<td>act1</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>cfg1</td>
<td>NULL</td>
<td>NULL</td>
<td>d1</td>
<td>cfg7</td>
<td>act1</td>
<td>NULL</td>
<td>d1</td>
</tr>
<tr>
<td>cfg2</td>
<td>NULL</td>
<td>NULL</td>
<td>d2</td>
<td>cfg8</td>
<td>act1</td>
<td>NULL</td>
<td>d2</td>
</tr>
<tr>
<td>cfg3</td>
<td>NULL</td>
<td>cat1</td>
<td>NULL</td>
<td>cfg9</td>
<td>act1</td>
<td>cat1</td>
<td>NULL</td>
</tr>
<tr>
<td>cfg4</td>
<td>NULL</td>
<td>cat1</td>
<td>d1</td>
<td>cfg10</td>
<td>act1</td>
<td>cat1</td>
<td>d1</td>
</tr>
<tr>
<td>cfg5</td>
<td>NULL</td>
<td>cat1</td>
<td>d2</td>
<td>cfg11</td>
<td>act1</td>
<td>cat1</td>
<td>d2</td>
</tr>
</tbody>
</table>

* Each configuration keeps the package name and service name as well, which are not shown in this table.
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 CFG (line 10). After that, with respects to the value of the parameter mode, ST repeats starting/stopping or binding/unbinding each of the exported service multiple times (with each configuration) determined by the corresponding threshold θpt 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 fhprof. The overall procedure of LA is shown in Procedure 2. For each dumped .hprof file fhprof, LA first finds out the set of all service instances Lsrv_all (line 4). Then, for each instance in Lsrv_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 CSRVP.keyset() if CSRVP[srv] > 0. The tuple is of the following form: ⟨cfg, srv, count⟩, 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 ⟨cfg, srv, count⟩, 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 provided by LeakCanary.

4 IMPLEMENTATION AND EVALUATIONS

In our prototype, we utilize Apktool to decode each apk file to obtain the corresponding AndroidManifest.xml.

2https://ibotpeaches.github.io/Apktool/
We use Soot\(^6\) to implement static analysis of bytecode and use PICT\(^7\) to generate \(S_{\text{service}}\) used in IFS-PB strategy. Service Trigger is implemented as an Android app running on the same device or emulator that hosting the AUT. For starting/stopping or binding/unbinding a service, ST directly invokes the \texttt{startService()} / \texttt{stopService()} or \texttt{bindService()} / \texttt{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 .\texttt{apk} files were clawed from GooglePlay from July 31st, 2015 to March 3rd, 2016. We analysed the \texttt{AndroidManifest.xml} of each .\texttt{apk} and found 74,831 services provided by 28,662 (about 69\%) .\texttt{apk}. We filtered out those declaring no exported service, retaining only 3,934 .\texttt{apk}. Then we came up with the answer to \textbf{RQ-1:} With respect to our collection of .\texttt{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 .\texttt{apk} files, finding that (1) many .\texttt{apk} files were actually different releases of the same application, and (2) some .\texttt{apk} files could be clustered as the exported services of them share the same naming pattern, despite that they belong to different apps (most .\texttt{apk} files of this kind were WallPaper apps). Thus, we randomly selected exactly one .\texttt{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 .

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

### 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 \texttt{“android:permission”} attribute of the service in the \texttt{AndroidManifest.xml} file. For example, apps on GooglePlay and only selected apps that were both popular (\#\texttt{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).

We applied LESDroid to the 375 apps with two modes\(=\{\text{“start”}, \text{“bind”}\}\), \(\theta_{\text{opt}}=10\) and three different strategies for generating configurations \{IFI, IFS, IFS-PB\}. Tab.4 shows the number of configurations generated under different settings. IFI generates exactly 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.

### Table 2: Emulator configuration

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Feature Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU/ABI</td>
<td>Intel Atom(X86_B4)</td>
</tr>
<tr>
<td>Device Ram Size</td>
<td>2GB</td>
</tr>
<tr>
<td>SD card size</td>
<td>32GB</td>
</tr>
<tr>
<td>Android API Version</td>
<td>Marshmallow (23)</td>
</tr>
</tbody>
</table>

### Table 3: Statistics of selected apps

<table>
<thead>
<tr>
<th>Apk Size</th>
<th>Rating</th>
<th>#download</th>
<th>#Exported Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>73KB</td>
<td>4.0</td>
<td>10K~50K</td>
</tr>
<tr>
<td>max</td>
<td>48,707KB</td>
<td>4.8</td>
<td>500,000K~1,000,000K</td>
</tr>
<tr>
<td>median</td>
<td>4,772KB</td>
<td>4.20</td>
<td>100K~500K</td>
</tr>
<tr>
<td>avg</td>
<td>8,082KB</td>
<td>4.24</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Start</th>
<th>Bind</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
</tr>
<tr>
<td>IFI</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>IFS</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>IFS-PB</td>
<td>1</td>
<td>30</td>
</tr>
</tbody>
</table>

\(^10\)Here, we say a service is bindable if ST could successfully obtain a \texttt{IBinder} instance from the service after calling the \texttt{bindService()} API.
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/exceptions (e.g., `NullPointerException`, `ClassNotfoundException`, `IllegalArgument`, `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 corresponding `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 permission, 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 $S_i$, we use $\gamma(S_i, cgs)$ to denote the maximum number of leak instances of the leak service $S_i$ detected by LESDroid with the strategy indicated by $cgs$, i.e., $\gamma(S_i, cgs) = \max \{t.count | t \in L_{Report}(cgs) \land t.srv = S_i \}$. 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., $\pi(S_{le}, cgs) = \max \{t.count | t \in L_{Report}(cgs) \land t.cfg.srv = S_{le} \}$. Then, Tab.8 shows the distribution of $\gamma(S_i,IFS-PB)$ for all leak services $S_i$ 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 $\gamma(S_i,IFS-PB)$ or $\pi(S_{le},IFS-PB)$.

<table>
<thead>
<tr>
<th>Table 5: Accessible applications, services and configurations under different experimental settings</th>
<th>Table 6: Statistics of evaluation results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start</strong></td>
<td><strong>Bind</strong></td>
</tr>
<tr>
<td>IFS</td>
<td>IFS-PB</td>
</tr>
<tr>
<td># app with accessible service</td>
<td>241</td>
</tr>
<tr>
<td># accessible service</td>
<td>359</td>
</tr>
<tr>
<td># accessible cfg</td>
<td>359</td>
</tr>
</tbody>
</table>

*Leak services detected in different apps are considered as different, even if they share the same name.*
Table 7: Processing time of LESDroid with IFS-PB (unit: ms)

<table>
<thead>
<tr>
<th></th>
<th>CG</th>
<th>ST</th>
<th>LA</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>1</td>
<td>8,024</td>
<td>9</td>
<td>1,384</td>
</tr>
<tr>
<td>max</td>
<td>531,095</td>
<td>392,232</td>
<td>354,437</td>
<td>867,688</td>
</tr>
<tr>
<td>median</td>
<td>13,700</td>
<td>17,501</td>
<td>3,308</td>
<td>37,891</td>
</tr>
<tr>
<td>avg</td>
<td>26,830</td>
<td>26,127</td>
<td>7,357</td>
<td>60,313</td>
</tr>
</tbody>
</table>

Table 8: Distribution of $\gamma(S_{le}, IFS-PB)$ for all leak entry services detected by LESDroid with IFS-PB

<table>
<thead>
<tr>
<th></th>
<th>$\gamma(S_{le}, IFS-PB)$</th>
<th>Start</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Bind</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>...</td>
</tr>
<tr>
<td>total</td>
<td>7</td>
<td>37</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>$\pi(S_{le}, IFS-PB)$</th>
<th>Start</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Bind</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>...</td>
</tr>
<tr>
<td>total</td>
<td>7</td>
<td>38</td>
</tr>
</tbody>
</table>

con.astra.aasramc.gu 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 were tested. Unlike setting $\theta_{ret} = 10$ 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 led 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 usage 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 useages. 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 $\pi(S_{le}, IFS-PB) = 1$, 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 $\pi(S_{le}, IFS-PB) \geq 5$ (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 $\pi(S_{le}, IFS-PB)$ bringing massive memory growths as well. For example, there are six leak entry services (with $\pi(S_{le}, IFS-PB) = 1$) 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

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

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

<table>
<thead>
<tr>
<th>Memory Growth (unit:MB)</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(0, 1]$</td>
<td>12</td>
</tr>
<tr>
<td>$(1, 2]$</td>
<td>1</td>
</tr>
<tr>
<td>$(2, 3]$</td>
<td>6</td>
</tr>
<tr>
<td>$(3, 4]$</td>
<td>9</td>
</tr>
<tr>
<td>$(4, 5]$</td>
<td>4</td>
</tr>
<tr>
<td>$(5, 6]$</td>
<td>7</td>
</tr>
<tr>
<td>$(6, 7]$</td>
<td>6</td>
</tr>
<tr>
<td>$(7, 8]$</td>
<td>3</td>
</tr>
<tr>
<td>$(8, 9]$</td>
<td>2</td>
</tr>
<tr>
<td>$(9, 10]$</td>
<td>1</td>
</tr>
<tr>
<td>$(10, 12]$</td>
<td>0</td>
</tr>
<tr>
<td>$\geq 12$</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\pi(S_{le}, IFS-PB)$</th>
<th>Start</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bind</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td>total</td>
<td>7</td>
</tr>
</tbody>
</table>
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 usage.

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. 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 logs.

Furthermore, in our experiment, the intervals between two consecutive rounds of starting/bindng 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 $\pi(S_i, IFS-PS)$ values tend 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 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.

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 AServerService 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 from 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

<table>
<thead>
<tr>
<th>Reason</th>
<th>start</th>
<th>bind</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Not Responding</td>
<td>17</td>
<td>7</td>
</tr>
<tr>
<td>Global Reference Table Overflow</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>Out Of Memory</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>NullPointerException</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SQLiteCanOpenDatabaseException</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>RejectedExecutionException</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>RuntimeException</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Signal-6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Signal-9</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>44</td>
<td>22</td>
</tr>
</tbody>
</table>
provide a systematic testing framework for Android services and combine different techniques (e.g., symbolic execution, NLP-based heuristic) to generate 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 code which may contain 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

ReIda2 [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 an 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 Intents. Then it tries to reveal leaks by starting/stoping or binding/unbinding each exported service of the app under test repeatedly with the Intents 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


