



南京大學

本科畢業論文

院 系 软件学院

专 业 软件工程

题 目 安卓不可见控件内存泄漏的自动检测

年 级 2016 级 学 号 161250136

学生姓名 王冬杨

指导老师 马骏 职 称 副教授

提交日期 2020 年 4 月 29 日

南京大学本科生毕业论文(设计、作品)中文摘要

题目：安卓不可见控件内存泄漏的自动检测

院系：软件学院

专业：软件工程

本科生姓名：王冬杨

指导老师（姓名、职称）：马骏副教授

摘要：

在安卓应用中，服务和广播得到了广泛应用，提供诸如下载，数据更新，跨应用通信等功能。但由于开发者经常忽视这些不可见控件的生命周期管理，因此内存泄漏发生的几率很高。

本文将关注新版本安卓系统（Android 8+）中公开服务（Exported Services）以及静态注册的广播接收器的内存泄露问题，阐述公开服务和广播接收器内存泄露的检测方法，并编写一套供服务开发人员使用的自动分析组件内存泄露的检测工具，最后会从应用市场（App Store）中下载真实的应用，进行内存泄漏检测和分析。

关键词： 安卓系统；内存泄漏；安卓服务；安卓广播

目 录

目 录	III
1 绪论	1
1.1 研究背景	1
1.2 相关工作	1
1.3 本文主要工作	2
1.4 本文结构	2
2 背景	5
2.1 安卓服务	5
2.1.1 服务的启动方式	6
2.1.2 服务的生命周期	6
2.1.3 服务的注册方式	6
2.1.4 服务的内存泄漏	7
2.2 安卓广播接收器	8
2.2.1 广播接收器的启动方式	9
2.2.2 广播接收器的生命周期	9
2.2.3 广播接收器的注册方式	10
2.2.4 广播接收器的内存泄漏	10
3 自动化检测工具	13
3.0.1 Apk 文件解析器	13
3.0.2 测试驱动 App	14
3.0.3 测试主流程	15
3.0.4 内存分析器	16
3.0.5 实现细节	17
4 实验	19
4.1 仿真测试	19

4.1.1 仿真应用	19
4.1.2 实验结果	20
4.2 真实测试	21
4.2.1 实验结果	22
4.2.2 应用崩溃主要原因分析	22
4.3 结果分析	23
4.3.1 数据对比	23
4.3.2 实验数据局限性	24
5 结论	25
参考文献	27
致 谢	29

第一章 绪论

1.1 研究背景

在安卓应用中，服务（Services）和广播（Broadcast）得到了广泛的使用。服务可以在安卓应用的后台保持长期运行，提供诸如下载、数据更新等重要功能。然而，正因为服务长期运行于后台的特点，使其往往容易产生异常（Errors）。如果服务的编写人员缺少警惕性，服务中出现的错误（Bug）可能会导致诸多问题，严重者可能引起应用崩溃，甚至系统死机；广播可以实现跨应用通信，要接收来自系统或者其他应用的广播，应用需要编写广播接收器（Broadcast Receiver），广播接收器将在 UI 线程运行，因此不适合进行耗时操作，通常会在广播接收器中启动服务来进行后续的处理，因此广播接收器也可能通过服务或者自身导致内存泄漏。

安卓应用中的内存泄露指资源（内存对象、句柄、服务等）将不再被使用，但却无法被垃圾回收机制（GC）回收，同时也是服务中的一大类常见问题。服务如果出现内存泄露，将会导致内存使用量意外大幅度增加，进而使得系统效率降低，严重影响用户体验。

服务如果设定‘exported:true’，则该服务可以被其他应用所调用，因此内存泄露的问题将会变得更加复杂。

由于在安卓 8 及更高的版本下，安卓操作系统的“电池优化策略”禁止跨应用启动后台服务^[1]，而这一方式在安卓 7 以及更早的版本中是可行的，因此在新版本的安卓系统中，公开服务的内存泄漏检测方法与之前的方法^[2]有所差别，也正因为禁止跨应用启动后台服务，公开服务的内存泄漏问题也得到了很大的规避。

1.2 相关工作

Erika 等人在安卓 8 之前的版本中，编写了一个检测跨应用通信安全问题的工具 Com Droid^[3]，文中阐述的方法对于跨应用测试具有指导意义。

在安卓 8 之前的版本中，跨应用启动服务这一行为是被允许的，南京大学的马骏等人安卓 8 之前的版本中，实现过一个公开服务（Exported Services）内

存泄漏的检测工具 LES Droid^[2]，文中采用的方式分为四步：

1. 使用 apktool 反编译工具^[4] 获取被测试应用的 AndroidManifest.xml 文件，解析获取应用中所有的公开服务的包名和类名。
2. 将测试驱动应用、被测试应用通过 adb 安装到模拟器中，启动测试驱动应用。
3. 测试驱动应用重复启动、关闭被测试的服务，在满足一定测试强度之后，导出被测试应用的堆镜像文件（.hprof files）。
4. 基于 MAT 内存分析库^[5] 编写堆镜像文件的分析工具，自动检测内存泄漏并统计泄露的入口等。

文中的数据指出：在 41537 个被测试应用中，共在其中 28662（69%）个应用中检测出 74831 个服务，其中 3934（13.7%）个应用拥有公开服务。经过去重、安装测试以及应用商店评分筛选，有 375 个实际测试应用，最终通过不同的测试配置，最终检测到在 18.7% 的应用中有 16.8% 的服务存在内存泄漏问题。

1.3 本文主要工作

本文旨在探索一套适用于安卓 8 以上版本的公开服务和静态注册广播接收器的内存泄漏检测方法。主要工作如下：

1. 找到在安卓 8 以上版本的安卓系统上可行的跨应用测试方法。
2. 对桩应用上进行测试，并能发现所有泄露。
3. 在应用商店中下载真实应用，进行自动化测试分析实验结果。

1.4 本文结构

本文的各章节组织结构如下：

第一章 绪论。简要说明了安卓组件内存泄漏的现象和后果。并概括地描述了检测安卓不可见控件内存泄漏的方法流程，总结了本文结构。

第二章 背景

第三章 自动化检测工具。

第四章 实验。介绍了实验进行的配置环境，测试使用应用的来源，以及实验数据结果。

第五章 总结与讨论。总结全文工作，讨论存在的问题和今后可以继续研究的方向。

第二章 背景

本节将介绍本文的测试对象：安卓服务以及安卓广播接收器。

2.1 安卓服务

在安卓系统中，每个安卓应用都对应着一个主线程，这个主线程将负责处理界面计算和渲染、负责处理用户的交互以及负责响应生命周期事件等。这些任务对主线程产生了快速响应的要求，否则会导致用户体验质量的下降。换言之，在主线程中只能进行不耗时（几毫秒级别）的计算和操作，而任何耗时较长的计算和操作都必须在主线程之外的单独的后台线程之上来完成。这样可以使得用户在积极的与应用交互时，应用在后台可以积极的响应和运行。

然而，后台任务不可避免地将会使用安卓设备的资源：例如内存资源和电池电量等。如果后台任务操作不当（如占用大量内存导致系统卡顿，长期进行密集计算导致电池电量下降变快），也可能会导致用户体验下降；同时如果安卓服务的开发人员在开发时引入了**程序缺陷**，使得服务在运行时产生了**内存泄漏**，将会使得前面的现象变得更加严重。因此随着安卓系统的更新，其对服务组件也进行了更多的限制，以尽可能保证用户体验。

- **安卓 6.0（API 级别 23）** 引入了**低电量模式以及应用待机模式**。低电量模式会在屏幕关闭且设备处于静止状态时，对运行中应用的行为进行限制，以减少电量消耗，例如：限制网络访问，限制同步等。即当开启低电量模式时，服务的行为会受到一定程度的限制。
- **安卓 7.0（API 级别 24）** 此版本限制了隐式广播的注册，并将上个版本中推出的低耗电模式推广成为一个随时进行的系统耗电优化：即当屏幕一段时间处于未唤醒状态且未接入电源，低耗电模式就会启动，而无需手动开启。即只要设备处于闲置状态时，服务的行为随时都会受到限制。
- **安卓 8.0（API 级别 26）** 对后台服务的行为进行了更严格的限制，其中影响较大的一条规则是：**禁止处于后台的应用启动新的服务**，取而代之的是需要显式经过用户许可，启动前台服务。这些限制对跨应用测试的研究带来了不便（跨应用测试一般由测试人员编写的测试驱动应用去启动其他（后

台) 应用的组件)。安卓 9.0 (API 级别 28) 引入了应用待机存储分区。系统将会根据应用的使用模式, 来动态分配给应用使用资源的优先级。

本节接下来将会详细介绍服务的启动方式, 生命周期, 注册方式, 以及内存泄漏的成因。

2.1.1 服务的启动方式

安卓应用中的服务可以通过两种方式启动^[6]:

start 方式 其他组件构造特定的 **Intent** 对象, 通过调用 **startService()** API 来启动目标服务。

bind 方式 通过调用 **bindService()** API 将目标服务与特定组件绑定。被绑定的服务提供接口供其他组件与之交互。一个服务可以同时通过以上两种方式启动。

2.1.2 服务的生命周期

服务的生命周期根据启动方式不同分为两种^[6]:

start 方式 通过 **startService()** API 启动的服务将会一直运行, 直到调用 **stopSelf()** 方法将自己停止运行。其他组件也可以通过调用 **stopService()** API 将服务停止运行。

停止运行的服务将会被 **GC(Garbage Collector)** 回收。

bind 方式 通过 **bindService()** API 启动的服务将通过 **IBinder** 接口与其他组件进行交互, 直到其他组件调用 **unbindService()** API 解除绑定。

一个服务可以同时绑定到多个组件之上, 直到所有组件都解除了绑定时, 该服务才会被 **GC** 回收。

每个安卓应用都关联一个 **ActivityThread** 实例, 负责调度和执行该应用的各种组件。**ActivityThread** 有一个 **ArrayMap** 类型的成员变量 **mServices**, 其中保存了所有没有被销毁的服务的引用。一旦某个服务的实例被销毁, 其引用将会从 **mServices** 中删除。

2.1.3 服务的注册方式

通常, 每个服务都要在 **AndroidManifest.xml** 中注册一个 **<service>** 标签 (参考 Listing. 1 中的样例)。同时服务可以通过设置 **”android:exported”** 属性

Listing 1 服务的注册方式

```

1  <manifest
2      xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:dist="http://schemas.android.com/apk/distribution"
4      package="com.example.myapplication">
5      <dist:module dist:instant="true" />
6      <application ...>
7          ...
8          <service android:name=".Service1"
9              android:enabled="true"
10             android:exported="true">
11      </service>
12      <service
13          android:name = ".Service2"
14          android:enabled = "true"
15          android:exported = "false">
16          <intent-filter>
17              <category android:name = "cat1"/>
18              <action android:name = "act2"/>
19          </intent-filter>
20      </service>
21      <service android:name = ".Service3"
22          android:enabled = "true"
23          android:permission = "Permission1">
24          <intent-filter>
25              <action android:name = "act3"/>
26              <category android:name = "cat2"/>
27              <data android:scheme = "Scheme1"/>
28              <data android:scheme = "Scheme2"/>
29          </intent-filter>
30      </service>
31  </application>
32 </manifest>

```

来指定该服务是否将被导出。当设置 `android:exported = "true"` 时，该服务可以被其他应用使用，反之不可。

2.1.4 服务的内存泄漏

通常，一个服务的实例不再被使用时应该被 **GC(Garbage Collector)** 回收，并释放资源。然而在某些情况下，一个被销毁的服务可能会意外的被引用，从而使得 **GC** 无法将其回收并释放资源，这样就造成了服务的内存泄漏。

例如在游戏 *com.siemdas.games2048* 中，就出现了原理如图（见 **Listing.2**）的内存泄漏。具体导致内存泄露的原理为：在 **LeakedService** 的实例被构造的时候，将会调用他的 `onCreate()` 方法，在该方法中延迟 **1000ms** 启动了一个匿名计时器，该计时器将以 **3000ms** 的周期打印调试信息，可以看到在 **TimerTask** 类中持有了 **LeakedService** 的引用，而在该服务被销毁时，其 `onDestroy()` 方法

Listing 2 服务的内存泄漏

```

1  public class LeakedService extends Service{
2      private static final String TAG = "LeakedService";
3      // Method will be called when an instance is creating.
4      public void onCreate(){
5          ...
6          new Timer().scheduleAtFixedRate(new TimerTask(){
7              public void run(){
8                  Log.d(TAG, LeakedService.this.getPackageName()
9                      + ".LeakedService is running!");
10             }
11         },1000L,3000L);
12     }
13     // Method will be called when an instance is destroying.
14     public void onDestroy(){
15         ...
16     }
17 }

```

中并没有对该匿名计时器进行销毁。因此在该服务被销毁后，将会一直存在一个匿名计时器持有该服务的引用，导致 **GC** 无法将其回收，从而导致了内存泄漏。

2.2 安卓广播接收器

安卓系统中，可以在安卓应用之间、以及安卓应用与安卓系统之间实现通讯，这种通讯的手段被称为广播^[7]，广播的设计采用了发布-订阅的设计模式。特定的广播将会在特定的事件发生时发送，例如：安卓系统将在各种系统事件（插拔耳机，插拔电源等）发生时发送系统广播；再如：一个安卓应用可以发送自定义的广播来通知其他的安卓应用，将一些消息传递给后者。

在使用广播时，应用需要注册接收特定的广播（称之为**订阅**）。在相应的广播发出，将由安卓系统负责将广播发送给订阅此种广播的应用。因此在安卓应用的角度，只需要订阅所需要的广播类型，以及关注在接收到广播时需要做出的响应动作即可，具有这种功能的组件被称为**广播接收器**。

一般而言，在跨应用通信时，广播是被广泛使用的一种方便的手段。但是在使用广播时需要格外小心，如果应用滥用广播，可能会导致系统运行变迟钝。更严重的，如果开发人员在编写广播接收器的时候引入**程序缺陷**，将会导致更严重的问题（如应用崩溃等）。

本节接下来将会介绍广播接收器的启动方式，生命周期，注册方式，以及

内存泄漏的成因。

2.2.1 广播接收器的启动方式

安卓应用中的广播接收器亦有两种方式启动^[8]:

清单声明的接收器 通过在 **AndroidManifest.xml** 中添加 **<receiver>** 标签注册广播接收器, 通过 **<intent-filter>** 标签指定接收器所订阅的广播操作。系统软件包管理器会在应用安装时注册接收器。然后, 该接收器会成为应用的一个独立入口点, 这意味着如果应用当前尚未运行, 系统可以启动应用并发送广播。系统会创建新的 **BroadcastReceiver** 组件对象来处理它接收到的每个广播。该对象仅在调用 **onReceive(Context, Intent)** 期间有效。一旦从此方法返回代码, 系统便会认为该组件不再活跃。

上下文注册的接收器 通过在代码中构造出 **BroadcastReceiver** 实例, 以及 **IntentFilter** 实例来指定订阅的广播内容, 调用 **registerReceiver(BroadcastReceiver, IntentFilter)** API 来注册接收器。只要上下文有效, 通过该方式注册的广播接收器就会接收广播。如果要停止接收广播, 需要调用 **unregisterReceiver(BroadcastReceiver)** API 来注销广播接收器

2.2.2 广播接收器的生命周期

广播接收器的生命周期根据启动方式不同亦分为两种^[8]:

清单声明的接收器 静态注册的接收器生命周期不限于 **Activity** 甚至整个应用。即使应用并不在运行, 接收器也可以接收到订阅的广播。将会在 **onReceive()** 方法结束后被销毁。

上下文注册的接收器 上下文注册的接收器, 其生命周期仅限于注册的上下文, 例如在 **Activity** 上下文注册的接收器, 在整个 **Activity** 存活期间可以持续接收广播; 在应用上下文中注册的接收器, 则会在整个应用运行期间都可以接收广播。需要注意的是: 这种方式启动的接收器必须手动进行销毁, 即调用 **unregisterReceiver()** API, 否则在上下文失效时, 系统会抛出异常 (并不会导致应用崩溃), 同时接收器会引发泄露 (见图. 2-1)。

```
2020-04-21 18:06:39.557 17978-17978/com.example.myapplication E/ActivityThread: Activity
com.example.myapplication.SecondActivity has leaked IntentReceiver com.example.myapplication
.LeakReceiver@e03b545 that was originally registered here. Are you missing a call to
unregisterReceiver()?
```

图 2-1: 没有回收接收器将会导致异常以及泄露

2.2.3 广播接收器的注册方式

Listing 3 广播接收器的注册方式

```

1  <manifest
2      xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:dist="http://schemas.android.com/apk/distribution"
4      package="com.example.myapplication">
5      <dist:module dist:instant="true" />
6      <application ...>
7          ...
8          <receiver
9              android:name = ".Receiver1">
10             <intent-filter>
11                 <action android:name = "act1" />
12             </intent-filter>
13         </receiver>
14         <receiver
15             android:name = ".Receiver2"
16             android:exported = "false"
17             android:enabled = "true">
18             <intent-filter>
19                 <category android:name = "cat1" />
20                 <action android:name = "act2" />
21             </intent-filter>
22         </receiver>
23     </application>
24 </manifest>

```

一般而言，清单声明的广播接收器（见 2.2.1）需要在 **AndroidManifest.xml** 文件中添加 **<receiver>** 标签（参考 **Listing. 3**），在 **<intent-filter>** 子标签中可以指定订阅的广播内容等，也可以通过设置“**android:exported**”属性来指定该广播是否将被导出。而上下文注册的广播接收器（见 2.2.1）则不需要进行前文的操作。

2.2.4 广播接收器的内存泄漏

广播接收器的内存泄漏原理类似与服务内存泄漏 2.1.4。但是由广播接收器引起的内存泄漏往往比服务更为严重，因为广播接收器被系统认为只进行不耗时的操作（如果超过 10s 未从 **onReceive()** 方法中返回，将抛出 **ANR Exception**），因此通常广播接收器在接到广播后，很有可能会启动其他的 **Service** 进行后续的耗时操作，进而可能会导致一连串的内存泄漏。

例如图中（见 **Listing. 4**）所示的广播接收器，不仅本身会导致内存泄漏，而且还会启动一个会导致内存泄漏的服务（见 **Listing. 2**），因此后果将会更加

严重。

Listing 4 广播接收器的内存泄漏

```

1  public class LeakReceiver extends BroadcastReceiver {
2      private final String TAG = "LeakReceiver";
3      private final int ID = new Random().nextInt();
4      @Override
5      public void onReceive(Context context, Intent intent) {
6          ...
7          context.startService(new
8              ↳ Intent(context, LeakService.class));
9          new Timer().scheduleAtFixedRate(new TimerTask() {
10              @Override
11              public void run() {
12                  Log.i(TAG, LeakReceiver.this.ID + " is
13                      ↳ running!");
14              }
15          }, 1000L, 3000L);
16      }
17  }

```


第三章 自动化检测工具

本章将介绍安卓控件检测内存泄露的原理。

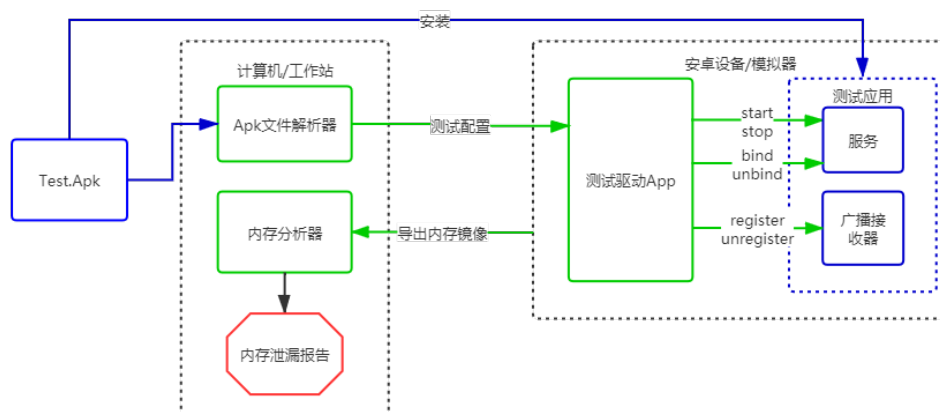


图 3-1: 自动检测工具原理

如图 3-1 所示，测试将在两个环境（工作站、安卓模拟器）中完成：

1. 在工作站中，使用 **Apk 文件解析器**（详见 3.0.1）对被测试应用 **Test.apk** 进行反编译，并解析 **AndroidManifest.xml** 清单，得到在清单中注册的公开服务以及清单声明的广播接收器，作为测试对象生成一份测试配置发送到安卓模拟器上。
2. 在安卓模拟器上安装好**测试驱动 App**（详见 3.0.2）以及被测试应用 **Test.apk**。通过 **ADB** 向安卓模拟器发送指令启动**测试驱动 App**，它会读取步骤 1 中发送来的测试配置，然后执行测试主流程（详见 3.0.3）
3. 在完成测试主流程之后，通知工作站将被测试应用的堆镜像文件（.prof）文件导出，使用**内存分析器**（详见 3.0.4）进行内存泄漏的检测，最终生成一份**内存泄漏报告**显示被测试应用的服务以及广播接收器有无内存泄漏情况，以及具体的内存泄漏的控件清单。

3.0.1 Apk 文件解析器

Apk 文件解析器旨在通过将测试应用反编译得到 **AndroidManifest.xml** 清单，对该清单进行分析，从而得到在清单中声明的公开服务以及广播接收器作为测试对象。

Listing 5 使用 apktool 工具进行 apk 的反编译

```
1 apktool d -f $test_apk$ -o temp
```

使用 apktool 工具^[4]，执行 **Listing. 5** 中的命令即可完成被测试应用的反编译得到 **AndroidManifest.xml** 清单。

Listing 6 使用 python 解析 xml 输出测试配置

```
1 from xml.dom.minidom import parse
2 def get_exported_services(xml_path):
3     exported_services = []
4     manifest = parse(xml_path).documentElement
5     for service in manifest.getElementsByTagName('service'):
6         exported = service.getAttribute('android:exported')
7         enabled = service.getAttribute('android:enabled')
8         if (exported == 'true' and (enabled == '' or enabled ==
9             ⇐ 'true')):
10             exported_services.append(service)
11     return exported_services
12
13 def get_static_receivers(xml_path):
14     exported_receivers = []
15     domTree = parse(xml_path)
16     manifest = domTree.documentElement
17     for receiver in manifest.getElementsByTagName('receiver'):
18         enabled = receiver.getAttribute('android:enabled')
19         if (enabled == '' or enabled == 'true'):
20             exported_receivers.append(receiver)
21     return exported_receivers;
```

接下来使用 **python** 中的 **xml** 库解析此 **xml** 文件（详见 **Listing. 6**），筛选出所有指定 **android:exported = "true"** 以及 **android:enabled = "true"**（或者不指定，默认值为"true"）的服务，以及在清单中声明的广播接收器。将这些控件的包名，类名，以及所需的权限等信息写入**测试配置**中

3.0.2 测试驱动 App

测试驱动 App 负责控制所有安卓模拟器上的测试行为（即测试主流程 **3.0.3**）。首先驱动会读取**测试配置**，获得要测试的服务和广播接收器，之后进行**测试主流程**，在完成测试流程之后，使用 **Socket** 与**工作站**进行通信，通知**工作站**测试任务已经完成，导出被测试应用的堆镜像文件进行最后的分析。

3.0.3 测试主流程

算法 3.1 测试主流程：公开服务测试

```

1: 读取测试配置
2: for 遍历测试配置中的公开服务 do
3:   重置安卓模拟器
4:   重置计数器
5:   if 使用 bind 方式测试 then
6:     启动仿真 Activity
7:   end if
8:   while 计时器 < 测试重复次数 do
9:     if 使用 start 方式测试 then
10:      调用 startService() API 启动服务
11:      调用 stopService() API 停止服务
12:     else
13:      调用 bindService() API 将服务绑定到 Activity 上
14:      调用 unbindService() API 解除服务绑定
15:     end if
16:     计数器自增 1
17:   end while
18:   if 使用 bind 方式测试 then
19:     销毁仿真 Activity
20:   end if
21:   通知工作站导出堆镜像
22: end for

```

算法 3.2 测试主流程：清单声明的广播接收器

```

1: 读取测试配置
2: for 遍历测试配置中的广播接收器 do
3:   重置安卓模拟器
4:   重置计数器
5:   while 计时器 < 测试重复次数 do
6:     构造特定广播，定向发送给该接收器
7:     计数器自增 1
8:   end while
9:   通知工作站导出堆镜像
10: end for

```

在测试主流程中，对服务和接收器进行逐一测试：

服务测试（见算法 3.1）测试原理为对被测试应用进行大量重复的启动和关闭（根据测试要求不同分为 start/bind 和 stop/unbind），确保潜在的内存泄漏被触发，之后将被测试应用的堆镜像导出，送入**内存分析器**（详见 3.0.4）进行检测和分析。

广播接收器测试（见算法 3.2）广播接收器的测试不需要事先启动应用，因为在清单中注册的广播接收器在应用安装时就会在系统中进行注册，在任何时候都可以接收广播（即使应用未运行），因此只需要构造接收器订阅的广播，定向发送给该接收器就可以触发该接收器。同样的为了确保潜在的内存泄漏被触发，需要重复发送大量的广播。最终将被测试应用的堆镜像导出，送入**内存分析器**（详见 3.0.4）进行检测和分析。

3.0.4 内存分析器

算法 3.3 内存分析器：服务分析

```

1: 导入并解析.hprof 文件
2: 查询所有 Service 的衍生类集合
3: for 遍历 Service 的衍生类集合 do
4:   if 该实例已经被销毁 then
5:     获得所有该实例到 GC Root 的路径集合
6:     去除路径集合中不合理的路径
7:     if 路径集合为空 then {证明该实例确实已经被销毁}
8:       该实例不存在内存泄漏
9:     else {该实例仍然被引用，产生内存泄漏}
10:      该实例产生内存泄漏
11:      if 该实例内存泄漏原因为已知的安卓系统 BUG then
12:        不判定构成人为内存泄漏
13:      else
14:        由开发者构成的人为内存泄漏
15:      end if
16:    end if
17:  end if
18: end for

```

内存分析器负责从堆镜像文件（.hprof 文件）中识别和统计服务（广播接收器）的内存泄漏实例，基于开源工具 MAT^[5] 进行定制化的开发，具体原理（详见算法 3.3 及 3.4）为：首先导入并解析堆镜像文件，找到所有的服务（广播接收器）实例，接下来对于每个找到的实例，逐个检测判断是否产生了内存泄露，对于产生内存泄漏的控件，生成一份内存泄漏分析报告。

服务的内存泄露判定 所有仍然活跃的服务都会被 **ActivityThread** 的 **mServices** 变量所引用，而所有被销毁的服务都会被从 **mServices** 中删除。一个对象的实例如果实际上处于存活状态，则一定会拥有至少一条有效的 **GC Root Path**。因此服务内存泄漏的判定充要条件为：拥有至少一条有效的 **GC Root Path** 且不在 **mServices** 中。

算法 3.4 内存分析器：服务分析

```

1: 导入并解析.hprof 文件
2: 查询所有 BroadcastReceiver 的衍生类集合
3: for 遍历 BroadcastReceiver 的衍生类集合 do
4:   if 该实例已经被销毁 then
5:     获得所有该实例到 GC Root 的路径集合
6:     去除路径集合中不合理的路径
7:     if 路径集合为空 then {证明该实例确实已经被销毁}
8:       该实例不存在内存泄漏
9:     else {该实例仍然被引用，产生内存泄漏}
10:      该实例产生内存泄漏
11:      if 该实例内存泄漏原因为已知的安卓系统 BUG then
12:        不判定构成人为内存泄漏
13:      else
14:        由开发者构成的人为内存泄漏
15:      end if
16:    end if
17:  end if
18: end for

```

广播接收器的内存泄漏判定 由于广播接收器被系统规定为不耗时控件（超过 10s 未完成 `onReceive()` 方法，将会抛出 **ANR Exception** 导致应用崩溃），因此对于广播接收器而言，只需要检测是否有实际处于存活状态的实例即可。即：拥有至少一条有效的 **GC Root Path**。

安卓系统自身原因产生内存泄漏 由于安卓系统自身可能存在 **BUG** 导致正常的控件产生内存泄漏，此类问题并非由于安卓应用开发者的失误而导致，因此要将此类内存泄漏排除。

3.0.5 实现细节

后台服务限制^[1] 在安卓 8 中，新增了“电池优化策略”，系统不允许后台应用创建后台服务，因此跨应用对服务进行测试将会失败。解决办法为：禁用系统的电池管理服务；在测试时，将被测试应用置于前台。

广播限制^[9] 在安卓 8 及更高版本系统的应用中禁止将隐式广播注册为清单声明的广播接收器。而显示广播和需要签名授权的广播接收器不受限制，可以继续注册为清单声明的广播接收器。**注意：**随着安卓系统的更新，很多隐式广播已经不再受到此规定的限制，具体的广播列表可以参考隐式广播例外^[10]。

超级用户限制 在安卓 7 以及更早的版本中，开发者经常会将系统增加超级用户权限，以便于进行测试（亦成为 **root**），大致的方法为将兼容的二进制

文件 **su** 拷贝到安卓设备中，使安卓设备可以执行超级用户指令 **sudo**，进而为测试带来便利。然而再**安卓 8** 以及更高的系统中，由于用于 **root** 操作的二进制文件 **su** 维护团队相继解散，获得非安卓系统开发人员获得超级用户权限变得困难，因此本文的自动化检测工具的实现不要求对系统进行 **root**，而将使用 **socket** 建立安卓设备与工作站的通信，由工作站使用 **adb** 指令完成跨应用的特殊权限操作：比如**跨应用导出堆镜像文件**操作需要应用拥有超级用户权限，而使用 **adb** 指令则不需要超级用户权限即可导出任意应用的堆镜像文件。

第四章 实验

在实验中，首先使用自动化检测工具对仿真应用进行了测试，以验证可行性和正确性。接下来对真实的应用进行了测试，收集了测试数据。

4.1 仿真测试

4.1.1 仿真应用

仿真应用（见 **Listing. 10**）应同时具有两个公开服务以及两个清单声明的广播接收器。其中每个控件之一需要人为制造内存泄漏（称为 **LeakedService(Listing. 7)** 与 **LeakedReceiver(Listing. 8)**），另一个则需要确保不存在内存泄漏（称为 **NormalService(Listing. 9)** 与 **NormalReceiver(Listing. 9)**）。在对仿真应用进行测试时，预期的实验结果为：能够检测到 **LeakedService** 和 **LeakedReceiver** 的泄露实例，以证明该工具可以发现内存泄漏问题。而检测不到 **NormalService** 和 **NormalReceiver** 的泄露实例，以证明该工具不会将正常的组件误检。

Listing 7 LeakedService 主体代码

```

1 public class LeakedService extends Service {
2     private static final String TAG = "LeakedService";
3     @Override
4     public void onCreate() {
5         super.onCreate();
6         new Timer().scheduleAtFixedRate(new TimerTask() {
7             @Override
8             public void run() {
9                 Log.i(TAG, LeakedService.this.getPackageName() +
10                     ↵ ".LeakedService running ");
11             }
12         }, 1000L, 3000L);
13     }
14 }

```

Listing 8 LeakedReceiver 主体代码

```

1  public class LeakedReceiver extends BroadcastReceiver {
2      private static final String TAG = "LeakedReceiver";
3      private final Random random = new Random();
4      @Override
5      public void onReceive(Context context, Intent intent) {
6          new Timer().scheduleAtFixedRate(new TimerTask() {
7              @Override
8              public void run() {
9                  Log.i(TAG, LeakedReceiver.this.random.nextInt() +
10                     ↵ ".LeakedReceiver running ");
11              }
12          }, 1000L, 3000L);
13      }
14  }

```

Listing 9 NormalReceiver 与 NormalService 主体代码

```

1  public class NormalReceiver extends BroadcastReceiver {
2      private static final String TAG = "NormalReceiver";
3      private final Random random = new Random();
4      @Override
5      public void onReceive(Context context, Intent intent) {
6          Log.i(TAG, NormalReceiver.this.random.nextInt() +
7             ↵ ".NormalReceiver running ");
8      }
9  }
10
11 public class NormalService extends Service{
12     private static final String TAG = "NormalService";
13     @Override
14     public void onReceive(Context context, Intent intent) {
15         Log.i(TAG, NormalService.this.getPackageName() +
16            ↵ ".LeakedService running ");
17     }
18 }

```

4.1.2 实验结果

实验结果（见图.4-2及图.4-1）能正确检测到 **LeakedService** 和 **LeakedReceiver** 的内存泄漏实例，而没有误检 **NormalService** 以及 **NormalReceiver**，证明检测工具实际有效。

Listing 10 仿真应用的 AndroidManifest.xml 清单

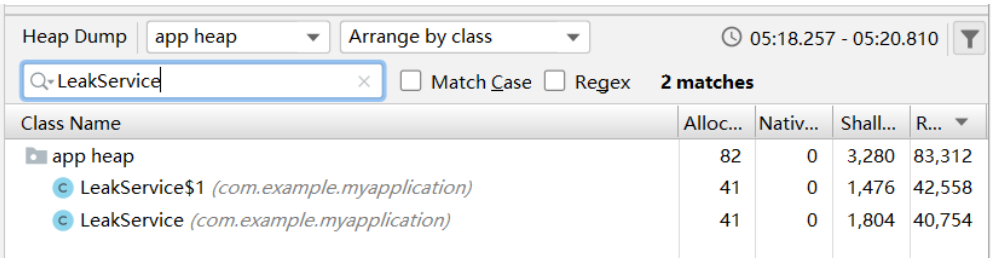
```

1  <manifest
2      xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:dist="http://schemas.android.com/apk/distribution"
4      package="com.example.myapplication">
5      <dist:module dist:instant="true" />
6
7      <permission
8          android:name="app.custom.permission"
9          android:protectionLevel="signature" />
10     <application ...>
11         <activity android:name=".MainActivity">
12             <intent-filter>
13                 <action
14                     ↪ android:name="android.intent.action.MAIN"
15                     ↪ />
16                 <category
17                     ↪ android:name="android.intent.category.LAUNCHER"
18                     ↪ />
19             </intent-filter>
20         </activity>
21
22         <receiver
23             android:name=".LeakedReceiver"
24             android:exported="true">
25             <intent-filter>
26                 <action android:name="TestActionForLeaked" />
27             </intent-filter>
28         </receiver>
29
30         <receiver
31             android:name = ".NormalReceiver"
32             android:exported = "true">
33             <intent-filter>
34                 <action android:name = "TestActionForNormal" />
35             </intent-filter>
36         </receiver>
37
38         <service
39             android:name=".LeakedService"
40             android:exported="true">
41         </service>
42
43         <service
44             android:name = ".NormalService"
45             android:exported = "true">
46         </service>
47     </application>
48 </manifest>

```

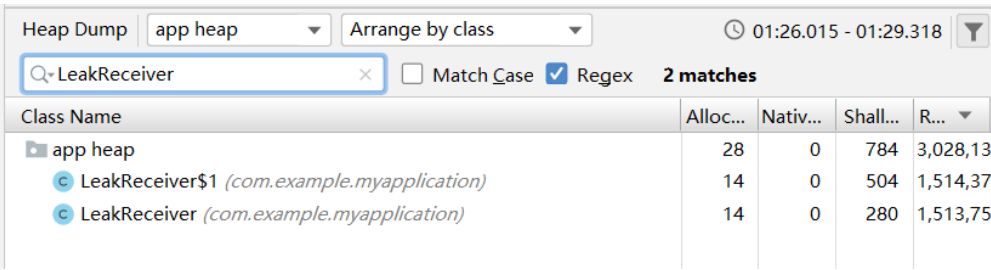
4.2 真实测试

本文选取了 **AppChina 应用市场**^[11] 作为测试应用的来源，在其中选取了各种类别应用中下载量最高的若干应用进行测试。对 **22** 个实际应用的测试结果



Heap Dump	app heap	Arrange by class	05:18.257 - 05:20.810	2 matches
Q LeakService				
Class Name	Alloc...	Nativ...	Shall...	R...
app heap	82	0	3,280	83,312
LeakService\$1 (com.example.myapplication)	41	0	1,476	42,558
LeakService (com.example.myapplication)	41	0	1,804	40,754

图 4-1: 检测到 **LeakedService** 内存泄漏实例



Heap Dump	app heap	Arrange by class	01:26.015 - 01:29.318	2 matches
Q LeakReceiver				
Class Name	Alloc...	Nativ...	Shall...	R...
app heap	28	0	784	3,028,13
LeakReceiver\$1 (com.example.myapplication)	14	0	504	1,514,37
LeakReceiver (com.example.myapplication)	14	0	280	1,513,75

图 4-2: 检测到 **LeakedReceiver** 内存泄漏实例

如下：

4.2.1 实验结果

表现正常 **12(54.5%)** 这些应用的组件表现正常，并没有出现内存泄露问题。

内存泄漏 **3(13.6%)** 这些应用包含了存在内存泄漏问题的组件，并成功被检测工具检测到。

应用崩溃 **7(31.8%)** 这些应用在测试时抛出了 **ANR(Application Not Response)** 异常，导致应用崩溃，无法完成测试。

4.2.2 应用崩溃主要原因分析

无法正常启动 部分应用在启动时即发生了崩溃，导致应用停止运行。这类应用崩溃的原因可能是因为应用的版本和模拟器系统的版本不兼容，例如使用了不再符合开发规范的接口，某些接口不再支持，或没有在 **AndroidManifest.xml** 中声明所需的权限（缺少 **<uses-permission>** 标签）等。

在测试过程中崩溃 部分应用可以正常启动，但是在主测试流程中发生了应用崩溃问题。这类应用崩溃的原因可能是因为应用的组件启动流程存在问题，比如：进行了风险操作，进行了线程不安全操作，没有对启动环境进行检查等；也可能是由于应用的组件确实存在内存泄漏问题，而且泄露表现的十分严

重，触发了安卓系统的安全限制，导致安卓系统介入将应用停止运行。

空指针异常 部分应用的组件在启动时会抛出 **NullPointerException**，该异常表示，在组件的启动过程中，对未实例化的对象进行了读写操作。这类问题大多数是因为组件的开发人员的疏忽，导致的程序缺陷。可能的成因有组件之间使用了共享资源，但是并没有进行专门管理，也有可能是因为这类组件的启动严格遵循**状态自动机**，但是测试时无法得知组件启动需要满足的前置条件，导致运行出错。

4.3 结果分析

本小节会将本文的测试结果（见4.2.1）与之前的结果^[2]进行对比，进行分析。

4.3.1 数据对比

表 4-1: 不同安卓版本上实验数据对比

	表现正常	内存泄漏	运行异常	泄露正常比
安卓 6	63.8%	13.7%	22.5%	0.215
安卓 9	54.5%	13.6%	31.8%	0.250

实验数据的对比表明：

- 存在内存泄漏的应用比例大致相同，但是表现正常的应用比例有下降，相应的运行异常的应用比例增加。
- 泄露正常比（检测到内存泄漏问题的应用数量与表现正常的应用数量之比）上升，这表明在**稳定可用的安卓应用**中，内存泄漏问题比早先的版本变得更多了。
- 运行异常的应用比例增加，原因可能是安卓系统版本升级后，将系统权限收紧，导致在旧的安卓版本上可以获取相应权限正常运行的应用，在新版本的安卓系统上无法正常获取到系统权限（可能是系统权限不再可以获取，或者获取权限方式改变等），从而导致应用权限不足，无法正常运行；另外的可能原因是安卓版本升级之后，新增或变更了部分开发规范，导致原有应用的代码在新版本安卓系统上不在兼容（例如新版本不在允许启动后台服务，而需要启动前台服务，并显式获得用户许可等）。

4.3.2 实验数据局限性

由于实验设备和实验环境的限制（参考表.4-2），本文只能使用效率较慢的方式对少量小规模的应用进行串行的测试（参考表.4-3）。因此本文得到的实验数据结果具有局限性，说服力较小。

表 4-2: 工作站配置对比

	操作系统	内存容量	处理器型号	模拟器系统版本
LESDroid 测试配置^[2]	Windows 10	32 GB	Intel Xeon E5-1650	Android 6
本文检测工具的测试配置	Windows 10	8 GB	Intel(R) Core(TM) i7-8565U	Android 9

表 4-3: 测试方法对比

	并发能力	模拟器内存	模拟器 SD 卡容量	测试强度
LESDroid 测试配置^[2]	5 线程并发	2 GB	2 GB	每个应用测试 3 次
本文检测工具的测试配置	单线程	2 GB	512 MB	每个应用测试 2 次

第五章 结论

本文实现了一个自动化的检测工具，它可以帮助安卓应用的开发人员对应用进行测试，帮助发现存在内存泄露问题的公开服务以及清单声明的广播接收器，以便于给用户带来更好的体验。

本自动化检测工具首先对被测试应用进行反编译，分析得到公开服务和清单声明的广播接收器等被测试组件信息；启动安卓模拟器，将被测试应用以及测试驱动应用安装到模拟器上进行测试流程；接下来对被测试组件进行反复的启动/关闭，绑定/解绑定，注册/解注册操作，最大程度确保内存泄漏问题被实际触发；最终为了找到内存泄漏的证据，将被测试应用的堆镜像文件导出到工作站中，基于开源工具开发了检测安卓组件内存泄漏的工具，最终导出内存泄漏报告。

安卓应用的开发人员只需要运行本文中的工具对其开发的应用进行检测，然后阅读内存泄漏报告，就可以对应用进行安全性的评估。

参考文献

- [1] Android Background Service Limit[K/OL]. 2020 [2020-04-27].
<https://developer.android.google.cn/about/versions/oreo/background#services>.
- [2] JUN M, SHAOCONG L, JIANG Y, et al. LESDroid-A Tool for Detecting Exported Service Leaks of Android Applications[C] // 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC). 2018 : 244 – 24410.
- [3] CHIN E, FELT A P, GREENWOOD K, et al. Analyzing inter-application communication in Android[C] // Proceedings of the 9th international conference on Mobile systems, applications, and services. 2011 : 239 – 252.
- [4] CONNOR TUMBLESON R W. A Tool for Reverse Engineering Android Apk Files[K/OL]. 2019 [2020-04-06].
<https://ibotpeaches.github.io/Apktool/>.
- [5] AndrodMat 2012.[K/OL]. 2012 [2020-04-06].
<https://github.com/joebowbeer/andromat>.
- [6] Android Service Guide[K/OL]. 2020 [2020-04-20].
<https://developer.android.com/guide/components/services.html>.
- [7] Android Broadcasts Guide[K/OL]. 2020 [2020-4-29].
<https://developer.android.google.cn/guide/components/broadcasts>.
- [8] Android Broadcast Guide[K/OL]. 2020 [2020-04-21].
<https://developer.android.com/guide/components/broadcasts>.
- [9] Android Broadcast Receiver Limit[K/OL]. 2020 [2020-04-27].
<https://developer.android.google.cn/about/versions/oreo/background#broadcasts>.
- [10] Implicit Broadcast Exceptions[K/OL]. 2020 [2020-04-27].
<https://developer.android.google.cn/guide/components/broadcast-exceptions>.

-
- [11] AppChina: An App Market for Chinese Mobile Users[K/OL]. 2020 [2020-04-28].
<http://www.appchina.com/>.

致 谢

由衷地感谢我的导师马骏副教授对我进行了悉心的指导，恩施思维敏捷，脚踏实地，对我实验过程中遇到的苦难和疑惑，都进行了仔细耐心的讲解和指导，对我的毕业设计的完成给与了巨大的帮助。

我也要感谢实验室的同学们，在三年的训练中，我们一起互相帮助，同甘共苦，攻坚克难，是我奋斗路上可靠的队友。愿我们的友谊地久天长，愿你们今后取得更高的成就。

另外，我还要感谢开源社区中无数勤劳无私的贡献者们，没有你们开发的众多的开源工具，本文的工作无法完成。在此感谢开源工作者，以及众多博主的技术博客，让我学到了很多。

最后，还要感谢我的家人，无论我身在哪里，家人都一如既往的支持鼓励着我，是我前进路上最坚强的后盾。

感谢所有支持帮助我的人，也祝各位同学前途无量。