Android 应用 Activity 启动环研究

刘 奥[1] 过辰楷[1] 王伟静[1] 侯晓磊[1] 朱静雯[2] 张 森[1] 许 静[3]

[1](南开大学计算机学院 天津 300350)

[2](南开大学软件学院 天津 300350)

[3](南开大学人工智能学院 天津 300350)

摘 要 Activity 的循环启动构成了 activity 启动环(ALC),它是一种 Android 应用开发工程师为了完成特定功能而广泛使用的结构.由于缺乏对 ALC 特性的系统研究,致使当前 activity 跳转分析方法对启动方式不敏感,使其无法正确模拟使用特殊启动方式的 Android 应用的返回栈状态变化,从而产生非法路径.本文形式化的表示了以 7 种不同方式启动 activity 带来的返回栈状态变化,并提出表示 activity 间启动关系的 activity 启动图(ALG),以及启动方式敏感的 activity 跳转静态分析方法用于自动构建 ALG.该方法首先为 Android 应用构造辅助主函数:为每个 activity 类构造一个对象的堆内存分配点,对于每个 activity 对象,依据控制流组织被重写的回调函数调用顺序.然后通过面向对象的字段敏感指向分析提取 activity 启动关系中的目标 activity 类和启动方式相关配置,从而构建 activity 对象的启动图.另外,本文设计并实现了 ALC 静态分析框架 ALCAnalyzer,该框架能为 Android 应用自动生成 ALG,基于 ALG 生成 ALC 集合,并能准确模拟重复执行 ALC 时的返回栈状态变化,预测应用在运行过程中是否会产生同类型 activity 实例.对 1179 个 Android 开源应用进行自动分析及人工验证的实验结果证明了启动方式敏感的 activity 跳转分析的准确性和分析工具的实用性,同时展现了 ALC 分布的广泛性和特殊启动方式被使用的广泛性.对 Google Play 的 20 个应用进行实验,结果证明相比于启动方式不敏感的 activity 跳转分析,ALCAnalyzer 能够更准确模拟返回栈状态变化,从而防止非法路径产生,并能够为返回栈管理提供有效信息.

关键词 安卓应用;activity 启动图;启动方式;返回栈;activity 启动环

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2020.00537

Research on Activity-Launch Cycles in Android Applications

LIU Ao[1] GUO Chen-Kai[1] WANG Wei-Jing[1] HOU Xiao-Lei[1] ZHU Jing-Wen[2] ZHANG Sen[1] XU Jing[3]

[1](*College of Computer Science, Nankai University, Tianjin* 300350)

[2](*College of Software, Nankai University, Tianjin* 300350)

[3](*College of Artificial Intelligence, Nankai University, Tianjin* 300350)

Abstract Activity launching cycle (ALC) allows an activity class to be launched repeatedly, which is widely used in Android applications (apps) to support specific functions. Special launch types can be used in ALC to prevent multiple instances of each activity class in the back stack. However, existing activity transition analyses are launch-typ-insensitive, which cannot capture special launch types, thus simulating the transitions among different back stack states incorrectly and producing infeasible activity transition paths for the apps using special launch types. To address above mentioned problems, we formalize the changes of back stack states triggered by activity

收稿日期:2018-06-27;在线出版日期:2019-03-26.本课题得到国家自然科学基金项目(61402264)、天津市自然科学基金重点项目(17JCZDJC30700,19JCQNJC00300)、天津市科技支撑项目(17YFZCGX00610,18ZXZNGX00310)资助.**刘 奥**,博士研究生,主要研究方向为软件分析技术、软件测试、信息安全技术.E-mail:18271390154@163.com.**过辰楷**(通信作者),博士,讲师,主要研究方向为软件分析技术、信息安全技术、模型检测.E-mail:chenkai.guo@163.com.**王伟静**,硕士研究生,主要研究方向为软件分析技术、软件测试、信息安全技术.**侯晓磊**,硕士研究生,主要研究方向为软件分析技术、软件测试、信息安全技术.**朱静雯**,硕士,助理实验师,主要研究方向为软件分析技术、软件测试.**张 森**,硕士研究生,主要研究方向为软件分析技术、软件测试、信息安全技术.**许 静**,博士,教授,中国计算机学会(CCF)高级会员,主要研究领域为软件分析技术、软件工程、软件测试、信息安全技术.

launchings configured with 7 different launch types respectively, and propose activity launching graph (ALG). The ALG represents activity launchings in an app and can be constructed by launch-typ-sensitive activity transition analysis. Launch-typ-sensitive activity transition analysis first constructs a harness main(), which consists of one allocation site per activity class and all overridden callback calls organized according to control flows. Then the object-oriented field-sensitive point-to analysis is conducted to extract the target activity classes for activity launchings and launch type related configurations. Finally, for each activity launching, an edge from source object to target object is constructed with determined launch type. Moreover, we propose and implement a framework named ALCAnalyzer to conduct the static ALC analysis. ALCAnalyzer can generate ALGs for Android applications automatically and generate the set of ALCs based on an ALG. Based on the maximum number (infinity, two, and one) of activity instances produced for an activity class in the back stack by repeated executions of ALCs, the ALCs can be divided into three types (TYPE1, TYPE2, and TYPE3). This paper summarizes the characteristics of different type of ALCs. ALCAnalyzer can simulate the changes of back stack states accurately during the repeated executions of ALCs and predict whether there are multiple instances of an activity class in a back stack state at runtime by determining the type of an ALC. Experimental evaluations consist of two parts. The first part is conducted on 1179 open source Android applications from F-Droid. Manual examinations on the results of ALCAnalyzer analysis show the high precision of our launch-typ-sensitive activity transition analysis. We also conduct studies on the ALCs in different kinds of apps in the first part of our experiments. The results indicate that: ALCs and special launch types are widely used in Android apps; more than half of the ALCs can produce multiple activity instance for an activity class, which request reasonable back stack managements; more than sixty percent of the ALCs are longer than two which are difficult to identified manually and requests a tool to identify them automatically; news- and reading-related apps are more complex in structure. The second part of experiments is conducted on 20 apps from Google Play. The results show that comparing with launch-typ-insensitive activity transition analysis, ALCAnalyzer can model the changes of back stack states more accurately, and ALCAnalyzer can also provide the software engineers with more effective information to manage the back stack behaviors to make user experience smoother and more consistent.

Keywords Android applications; activity launching graph; launch type; back stack; activity launching cycle

1 引 言

近几年,人们已经见证了移动设备在各个领域的广泛应用. Android 系统是移动设备中最为流行的系统之一. Android 平台上的应用软件类型丰富,数量庞大,且层出不穷. Activity 作为 Android 应用(Android application,简称 app)重要组件之一,负责提供用户交互功能的界面窗口. 软件运行过程中产生的 activity 实例以栈的形式组织在返回栈中,以便用户正确返回上层界面. 用户通过与返回栈最顶端实例进行交互来完成 activity 间的跳转. Activity 跳转由 activity 的启动和返回动作触发. 不同 activity 类间的启动关系,构成了一个 app 的整体框架. 本文关注于 activity 循环启动所构成的一种特殊结构:activity 启动环(Activity Launching Cycle,简称 ALC).

ALC 的重要特性是其存在允许 app 在被使用的过程中多次启动同一 activity 类. 默认的 activity 启动方式(standard)要求 Android 平台创建一个目标 activity 的新实例并压入返回栈,所以返回栈中可能出现同一类型 activity 的多个实例. 过多的同类型 activity 实例会占用大量系统资源,以致出现卡顿甚至闪退;另外,同类型 activity 实例也会使得

用户回退操作繁复，影响用户体验[1]. Android 平台提供了多种特殊配置，使得 ALC 中的 activity 以特殊的启动方式启动. 这些特殊启动方式要求系统在软件使用过程中重用或清除特定 activity 类的实例，以避免同类型 activity 实例的产生. 例如：activity A 以 singleTop 的方式启动 A，构成了 activity 启动环 $ALC1$. Android 平台并不会在执行 $ALC1$ 时向返回栈中压入一个 A 的新实例，而是重用栈顶实例. 通过对大量开源应用中 ALC 的调研，本文根据持续重复执行 ALC 产生同类型 activity 实例的三种情况，将 ALC 归纳为 3 类. 准确对 ALC 进行分类，能够帮助开发人员调整 ALC 中的启动方式，合理管理返回栈以提升软件质量.

如上所述，开发人员为了完成特定功能，追求更流畅便捷的用户体验，常在 ALC 中使用特殊启动方式. 然而当前 activity 跳转静态分析方法[1-5]对启动方式不敏感，即它们假设所有的 activity 均以默认启动方式被启动. 所以若 ALC 中使用了特殊启动方式，当前分析方法则无法正确模拟其引起的返回栈状态变化，也就无法正确模拟返回动作所触发的 activity 跳转以及无法对 ALC 正确分类. 另外，基于 activity 跳转路径的研究与应用中，包含错误 activity 跳转的非法 activity 跳转路径会带来意料之外的错误，比如 GUI 测试和缺陷检测中的测试用例执行失败[5-8].

为了弥补当前 activity 启动关系相关分析对启动方式不敏感的缺陷，解决返回栈模拟不正确、产生非法 activity 跳转路径的问题，并正确判定 ALC 类别，本文：(1) 深入研究了 activity 启动环使用的特殊启动方式，并形式化表示了以 7 种不同启动方式启动 activity 带来的返回栈变化；(2) 提出 activity 启动图(Activity Launching Graph，简称 ALG)，该图节点表示 activity 对象，有向边表示 activity 间的启动关系且记录了启动方式. ALG 可以被用于多种上层研究，比如 Android 应用动态探索[2-3,9]、GUI 模型构建[5,8]、漏洞挖掘[10-16]、缺陷检测[6-7,17-18]等；(3) 提出启动方式敏感的 activity 跳转静态分析方法用于自动构建 ALG；(4) 提出基于 ALG 的 ALC 定位算法和 ALC 分类算法.

对 Android 应用进行启动方式敏感的 activity 跳转分析从而构建 ALG 是本文工作的关键也是难点所在. Android 平台使用 intent 对象来帮助进行组件间通信(Inter-Communication Call，简称 ICC)，从而启动目标组件. 为了确定目标 activity 和提取部分启动方式相关配置，需要通过指向分析来解析 intent 对象的字段内容. 与典型 Java 程序只有一个入口不同，Android 应用程序包含大量回调函数，用户和系统动作都会触发相应回调，即 Android 应用程序包含多个入口. 因此，如何有效处理 Android 程序的事件驱动复杂控制流是指向分析需要克服的挑战. 另外，由于 Android 平台提供大量系统 API，很多代码被隐藏，使得对 Android 应用进行精准的指向分析更加困难[1-4,11,13,16,19-21].

本文提出并实现了如图 1 所示的 ALC 静态分析框架 ALCAnalyzer 来完成 ALG 构建，ALC 定位和 ALC 分类. ALG 生成器为 Android 应用构造辅助主函数：为每个 activity 类构造一个对象的堆内存分配点，对于每个 activity 对象，依据控制流组织被重写的回调函数调用顺序. 之后通过面向对象的字段敏感指向分析构建 activity 对象的启动图. ALC 生成器以 ALG 为输入，定位 ALG 中全部 ALC 并输出 ALC 集合. 返回栈模拟器能够模拟 7 种 activity 启动方式对返回栈的影响. ALC 分类器被集成在返回栈模拟器中，根据每次模拟执行 ALC 后返回栈的状态来判断 ALC 的类别，最终给出 app 在运行过程中可能产生多个实例的 activity 集合.

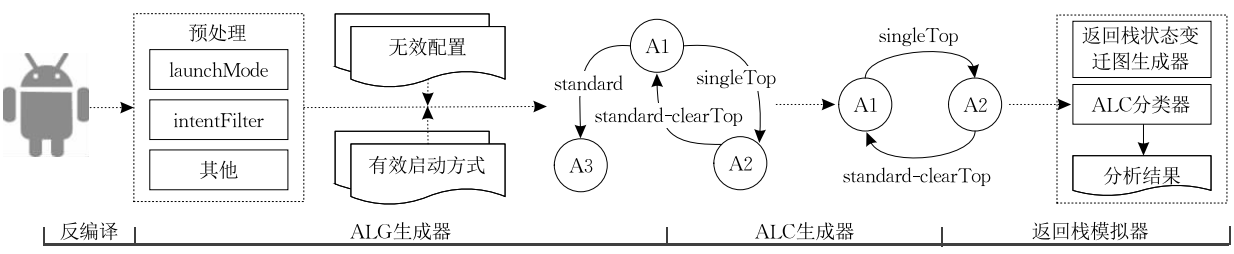


图 1　ALCAnalyzer 框架图

本文的主要贡献如下：

(1) 对 Android 应用中的特殊启动方式进行了深入研究，形式化的表示了以不同启动方式启动 activity 带来的返回栈状态变化.

(2)提出 activity 启动图的概念,以及启动方式敏感的 activity 跳转静态分析方法用于自动构建 ALG.

(3)提出并实现了 ALC 静态分析框架 ALCAnalyzer,该框架能自动完成 ALG 构建、ALC 定位、返回栈变化模拟、ALC 分类.

(4)对开源应用市场 F-Droid 中的 1179 个 Android 应用进行了自动分析.人工检查结果表明了分析方法的有效性和分析工具的实用性.对 Google Play 中 20 个 Android 应用进行分析,结果证明了 ALCAnalyzer 能够准确模拟返回栈状态变化,防止非法路径产生,并为开发人员管理返回栈提供有效信息.

2 研究动机

本节通过一款 SSH 客户端应用 ConnectBot 的例子来说明现有启动方式不敏感的 activity 跳转分析方法的局限性.

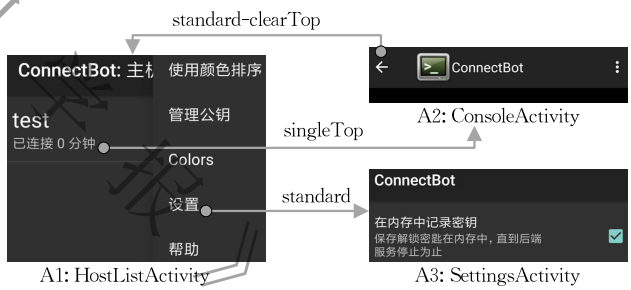
2.1 一个 ALC 实例

图 2(a)展示了 ConnectBot 中三个 activity 类以及配置文件 AndroidManifest.xml 的代码片段.代码片段 30～33 行决定了该应用 main activity 是 HostListActivity.当应用被启动时,系统会首先创建一个 HostListActivity 的实例并压入该应用的返回栈.图 2(b)展示了 ConnectBot 中三个 activity:$A1$,$A2$,$A3$ 间的启动关系.当 $A1$ 中的列表项"test"被点击时,如第 9 行所示的 ICC 函数 startActivity()被调用从而启动 $A2$.由于 ConsoleActivity 的"launchMode"设置为"singleTop"(35 行),所以 $A1$ 以 singleTop 方式启动 $A2$.第 13 行通过 API setIntent()设置目录项的响应.当 $A1$ 中右侧"设置"目录项被点击时,$A3$ 以 standard 方式被启动.当 $A2$ 左上角"←"被点击时,$A2$ 通过 ICC 函数 startActivity()来启动 $A1$(24 行).由于 intent 对象的标志设置为"FLAG_ACTIVITY_CLEAR_TOP"(23 行)且 HostListActivity 没有设置"launchMode"的值,所以 $A2$ 以 standard-clearTop 方式启动 $A1$.系统如何通过 intent 对象决定目标 activity 及如何根据返回栈相关配置判断启动方式将在 4.3 节详述.

从图 2(b)可已看出,ConnectBot 中存在 activity 启动环 $ALC2=A1\xrightarrow{\text{singleTop}}A2\xrightarrow{\text{standard-clearTop}}A1$.该环中的两个启动关系均配置为特殊的启动方式.

```
1  // HostListActivity
2  public class HostListActivity extends ListActivity{ …
3     public void onCreate(){ …
4        ListView list = this.getListView();
5     /    / 为列表项注册点击动作的监听器
6        list.setOnItemClickListener(new OnItemClickListener(){
7           public synchronized void onItemClick( … ){ …
8              Intent contents = new Intent(Intent.ACTION_VIEW, uri);
9              … HostListActivity.this.startActivity(contents) } } ); }
10    public Boolean onCreateOptionMenu(Menu menu){ …
11       MenuItem settings = menu.add(R.string.list_menu_settings);
12    /    / 为“设置”目录项添加点击响应动作
13       setting.setIntent(new Intent(
14          HostListActivity.this, SettingsActivity.class));
15       return true; } … }
16 // ConsoleActivity
17 public class ConsoleActivity extends Activity{ …
18  / / 设置“←”目录项的响应动作
19    public boolean onOptionItemSelected(MenuItem item){
20       switch (item.getItemId()){
21          case android.R.id.home:
22             Intent intent = new Intent(this, HostListActivity.class);
23             intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
24             startActivity(intent);
25             return true; } } … }
26 // SettingsActivity
27 public class SettingsActivity extends PreferenceActivity{ … }
28 // AndroidManifest.xml
29 <activity android:name=".HostListActivity"> …
30    <intent-filter>
31       <action android:name="android.intent.action.MAIN" />
32       <category android:name="android.intent.category.LAUNCHER" />
33    </intent-filter></activity>
34 <activity android:name=".ConsoleActivity"
35    android:launchMode="singleTop" … >
36    <intent-filter>
37       <action android:name="android.intent.action.View" … />
38    </intent-filter></activity>
39 <activity android:name=".SettingsActivity" … />
```

(a) 三个activity类的代码片段和AndroidManifest.xml配置文件



(b) 三个activity类间的启动关系

图 2 ConnectBot 代码片段及部分启动关系

2.2 现有分析方法局限性

当前对启动方式不敏感的 activity 跳转静态分析方法[1-5]无法提取 activity 启动方式,且在模拟返回栈状态变化时假设所有 activity 均以 standard 方式被启动.所以现有分析方法只能模拟两种返回栈变化:以默认启动方式启动 activity 带来的实例入栈和返回动作带来的 activity 实例出栈.图 3(a)展示了现有分析方法针对 $ALC2$ 所模拟的部分返回栈状态变迁,$e1$、$e2$ 表示启动 activity 带来的返回栈变化,$e3$、$e4$ 表示返回动作带来的返回栈变化,$s1$、$s2$、$s3$ 表示三个返回栈的状态.图 3(a)模拟了如下场景:初始时返回栈中只有一个 $A1$ 的实例($s1$),用户点击 $A1$ 中的列表项"test"后,一个 $A2$ 的新实例被

创建并压入返回栈中($e1$)；用户继续点击 $A2$ 中的“←”，启动方式不敏感的分析方法模拟 standard 启动方式向返回栈压入 $A1$ 的新实例($e2$)，此时返回栈状态为 $s3$；若继续遍历 $ALC2$，那么将会持续向返回栈压入 $A2$ 和 $A1$ 的新实例；若在状态 $s3$ 时，用户执行了返回动作，则栈顶实例被销毁($e4$).

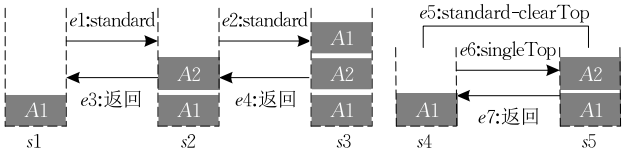


图 3　针对 $ALC2$ 的返回栈变化模拟示意图

真实的返回栈变化如图 3(b)所示，其模拟的场景如下：singleTop 要求当源 activity 的类型和目标 activity 的类型相同时，系统重用栈顶实例，否则创建一个目标 activity 的新实例并压入栈中. 由于 $A1$ 和 $A2$ 类型不同，所以点击“test”后引起的返回栈状态变化如 $e6$. standard-clearTop 要求若栈中已经存在目标 activity 的实例，则销毁该实例及以上全部实例，并创建一个目标 activity 的新实例. 所以点击“←”引起的返回栈状态变化如 $e5$.

可以看出，图 3(a)中 $s3$ 是不可能出现的错误状态，$e3$，$e4$ 为错误的状态变迁. 在基于 activity 跳转路径的研究与应用中，包含错误返回栈状态变迁的非法路径会带来意料之外的错误. 比如：在 GUI 测试和缺陷检测[2-3,5-7,9,12-15]中，需要以动作序列作为测试输入. 启动方式不敏感的分析方法允许生成非法路径“$e1$，$e2$，$e4$，$e3$”对应的动作序列 $es=$〈点击“test”，点击“←”，返回，返回〉. 显然该测试用例执行完动作序列〈点击“test”，点击“←”，返回〉后，ConnectBot 应用已经退出，根本无法继续执行后续动作从而导致测试失败. 对于 ConnectBot 来说，启动方式不敏感的分析方法允许产生 920 条长度为 4 的路径(只考虑返回动作和 activity 启动)，其中 483 条为非法路径.

显然，当 ALC 中使用了特殊启动方式，当前分析方法无法正确模拟其引起的返回栈的状态变化，也无法正确模拟返回动作所触发的 activity 跳转，从而产生非法路径，更无法对 ALC 正确分类. 本文提出的 ALCAnalyzer 则能够自动构建 ALG，准确模拟返回栈状态变化，并对 ALC 准确分类.

3　Activity 启动关系和启动环

本节对返回栈和 activity 的启动方式做了简述；形式化表示了以不同启动方式启动 activity 引起的返回栈状态变化；定义了表示 activity 间启动关系的 activity 启动图及 activity 启动环；并阐述了 ALC 分类及各类 ALC 特性.

3.1　相关 Android 特征

(1) 返回栈(task)

软件运行过程中产生的 activity 实例以栈的形式组织在返回栈中，以便用户正确返回上层界面(本文“界面”指一个 activity 实例向用户提供的可视化组件的全集. 本研究关注于 activity 间启动关系的建模及返回栈状态的模拟. Activity 内部组件内容变更引起的界面变化，如 webview 组件内容的变更，不在本文研究范围.)栈顶实例为当前 activity，用户按回退按钮时，当前 activity 会从栈顶部弹出并销毁. 返回栈可形式化为 $BS=\langle a_{c_1},a_{c_2},\cdots,a_{c_n}\rangle$，其中 a_{c_i} 表示 activity 类 c_i 的一个实例，a_{c_1} 为栈底实例，a_{c_n} 为栈顶实例. 可通过为 activity 设定启动方式来帮助管理返回栈. 若两个返回栈中 activity 实例个数相同且对应位置的两个 activity 实例类型相同，本文称这两个返回栈状态相同. Android 平台为每个应用维护一个返回栈，其名称与应用包名相同.

(2) Activity 的启动方式

源 activity c_n 以启动方式 lt 启动目标 activity c_x 记为 $c_n \xrightarrow{lt} c_x$. Android 应用中，为 intent 对象设定的标志、finish()的调用、AndroidManifest.xml 文件中为 activity 设定的 launchMode 属性及 taskAffinity 属性共同决定了 activity 的启动方式，本文称这四个属性为 activity 启动方式相关属性. 表 1 归纳了 7 种影响单个返回栈状态的 activity 启动方式并形式化的说明每种启动方式对返回栈的影响. 各种启动方式的详细配置在第 4.3.3 节详述.

表 1　以不同启动方式启动 activity 带来的返回栈变化. 设启动关系为 $c_n \xrightarrow{lt} c_x$，当前返回栈状态为〈$a_{c_1},a_{c_2},\cdots,a_{c_x-1},a_{c_x},a_{c_x+1},\cdots,a_{c_n}$〉，其中 a_{c_1} 为栈底实例，a_{c_n} 为栈顶实例，a'_{c_x} 是区别于 a_{c_x} 的 c_x 的新实例

启动方式(lt)	启动目标 activity 后的返回栈
standard	$\langle a_{c_1},a_{c_2},\cdots,a_{c_x-1},a_{c_x},a_{c_x+1},\cdots,a_{c_n},a'_{c_x}\rangle$
singleTop	if $c_x=c_n$: $\langle a_{c_1},a_{c_2},\cdots,a_{c_x-1},a_{c_x},a_{c_x+1},\cdots,a_{c_n}\rangle$ else: $\langle a_{c_1},a_{c_2},\cdots,a_{c_x-1},a_{c_x},a_{c_x+1},\cdots,a_{c_n},a'_{c_x}\rangle$
standard-clearTop	$\langle a_{c_1},a_{c_2},\cdots,a_{c_x-1},a'_{c_x}\rangle$
singleTop-clearTop	$\langle a_{c_1},a_{c_2},\cdots,a_{c_x-1},a_{c_x}\rangle$
reorderToFront	$\langle a_{c_1},a_{c_2},\cdots,a_{c_x-1},a_{c_x+1},\cdots,a_{c_n},a_{c_x}\rangle$
singleTask	$\langle a_{c_1},a_{c_2},\cdots,a_{c_x-1},a_{c_x}\rangle$
standard|finish	$\langle a_{c_1},a_{c_2},\cdots,a_{c_x-1},a_{c_x+1},\cdots,a_{c_n-1},a_{c_x}\rangle$

(1) standard. 无论当前返回栈中是否已经包含目标 activity 的实例，一个目标 activity 的新实例均

会被创建并压入当前返回栈中.

(2) singleTop. 若栈顶实例类型与目标 activity 类型相同,则不会在栈顶创建新实例.

(3) standard-clearTop. 若栈中已经存在目标 activity 的实例 a_{c_x},那么该实例及之上的实例都会被销毁,系统重新创建一个目标 activity 的新实例 a'_{c_x} 并压入栈中.

(4) singleTop-clearTop. 若栈中已经存在目标 activity 的实例 a_{c_x},那么该实例之上的实例都会被销毁,a_{c_x} 成为栈顶实例.

(5) singleTask. 效果与 singleTop-clearTop 相同,只是配置不同.

(6) reorderToFront. 若栈中已经存在目标 activity 的实例 a_{c_x},则将该实例置于栈顶,其余元素位置不变.

(7) 伴随 finish() 的启动(standard|finish). 目标activity的新实例被创建的同时当前activity实例由于 finish() 的调用而销毁. 该种启动方式并不一定要在 activity 启动环中才能发挥作用.

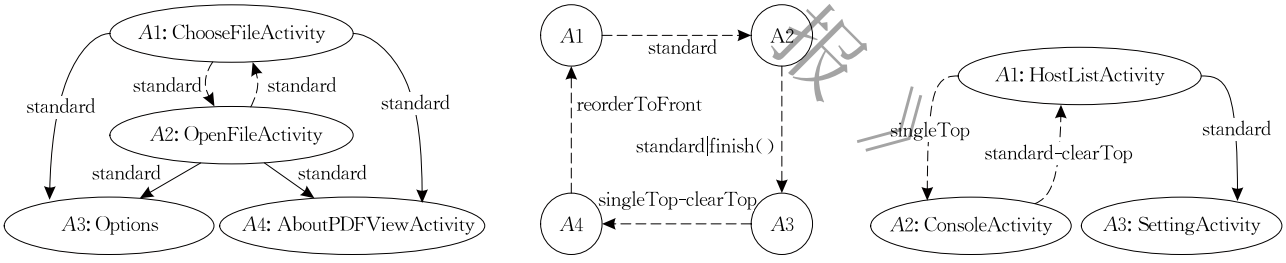
3.2 Activity 启动图和启动环

定义 1. activity 启动图(ALG)是一个三元组:$ALG=(N,E,st)$. N 表示 activity 类型集合. E 是有向边集合:$E=\{\langle n_s,n_e,lt\rangle \mid n_s\in N,n_e\in N,lt\in LT\}$,其中 n_s 表示边的始点,n_e 表示边的终点,lt 是启动方式集合 LT 中的元素,表示 n_s 以 lt 的方式启动 n_e. ALG 的起始节点为 st,为 Android 应用的 main activity.

定义 2. activity 启动环(Activity Launching Cycle,简称 ALC)是 ALG 中的一个初级回路. 若两个 ALC 有向边集合相同,则称他们具有相同的结构. 起始节点不同的两个 ALC 可能具有相同的结构.

3.3 ALC 分类

若一个 Android 应用的 ALG 中没有 ALC,那么 activity 不可能被循环启动,返回栈中肯定不会产生同类型的 activity 实例. Android 应用存在 ALC 是其产生同类型 activity 实例的必要条件. 根据重复执行 ALC 时,返回栈中产生同类型 activity 实例的情况将 ALC 分为 3 类. 针对这 3 类情况各展示了一个样例,如图 4 所示,其中 ALC 用虚线标记. 表 2 展示了这三类 ALC 被连续执行 2 次过程中返回栈的变化过程,表中用小写字母"a"表示 activity 实例,下标指示类型,上标指示其为该类型的第几个实例. 每启动一个 activity,返回栈状态就更新一次,同类型 activity 实例用黑框标出.



(a) APV中的*ALC*3　(b) Test Example中的*ALC*4　(c) ConnectBot中的*ALC*2

图 4　三类 ALC 的样例((a)第一类,(b)第二类,(c)第三类)

表 2　重复执行两次 ALC 过程中返回栈状态变化情况

ALC 名称	*ALC*3(TYPE1)					*ALC*4(TYPE2)									*ALC*2(TYPE3)				
重复次数	init	1		2		init	1				2				init	1		2	
	1	2	3	4	5	1	2	3	4	5	6	7	8	9	1	2	3	4	5
返回栈状态变化	a_1^1	a_2^1	$\boxed{a_1^2}$	$\boxed{a_2^2}$	$\boxed{a_1^3}$	a_1^1	a_2^1	a_3^1	a_4^1	a_1^1	a_2^2	$\boxed{a_3^2}$	a_4^1	a_1^2	a_1^1	a_2^1	a_1^2	a_2^2	a_1^3
		a_1^1	a_2^1	$\boxed{a_1^2}$	$\boxed{a_2^2}$		a_1^1	a_1^1	a_3^1	a_4^1	a_1^1	a_1^1	a_3^1	a_4^1		a_1^1		a_1^2	
			$\boxed{a_1^1}$	$\boxed{a_2^1}$	$\boxed{a_1^2}$				a_1^1	a_3^1	a_1^1	a_4^1		a_3^1					
				$\boxed{a_1^1}$	$\boxed{a_2^1}$						a_3^1	$\boxed{a_3^1}$							
					$\boxed{a_1^1}$														

第一类 ALC. 其特征在于：(1) ALC 中至少存在一个 activity 类 c，重复执行 ALC t 次后，返回栈中 c 的实例个数 n 满足：

$$\begin{cases} n=t+1, & c \text{ 为起始 activity 类} \\ n=t, & c \text{ 为非起始 activity 类} \end{cases}$$

且返回栈中总 activity 实例数量不会超 $tl+1$，其中 l 表示 ALC 长度. 当 t 足够大时，应用程序会因为过多的同类型 activity 实例严重占用系统而强制退出；(2) 由于持续重复执行 ALC 会产生无限个 activity 实例，所以也会有无限个返回栈状态，并且每次执行结束后的返回栈状态均不相同. 图 4(a) APV 中的 *ALC*3 即为第一类，从表 2 可以看出两次执行 *ALC*3 的过程中返回栈状态完全不同. 重复执行 *ALC*3 t 次后，ChooseFileActivity 的实例个数为 $t+1$，OpenFileActivity 的实例个数为 t. 用户想返回栈底界面则需要 $2t$ 次的回退操作.

第二类 ALC. 其特征在于：(1) 重复执行 ALC 过程中，至少存在一个能够产生两个实例的 activity 类 c，并且返回栈中每种类型 activity 最多出现两个实例，即返回栈中的实例总数最多为 $2l$，其中 l 表示 ALC 长度；(2) 重复执行 t 次后，每次执行结束后的返回栈状态均与第 t 次执行结束后返回栈状态相同且第 $t+1$ 次执行过程中的返回栈状态将会重复出现，所以返回栈状态数有限. 为详细说明该类 ALC 的特征，本文结合多种 activity 启动方式构造了 *ALC*4，如图 4(b) 所示. 从表 2 可知，返回栈只有在状态 7 时才会出现两个 *A*3 类型的实例. 第二次执行完 *ALC*4 的返回栈状态 9 和第一次执行 *ALC*4 的状态 5 相同，继续执行下去与 6、7、8、9 相同的返回栈状态会重复出现.

第三类 ALC. 其特征在于：(1) 在持续重复执行 ALC 过程中每个类型 activity 最多产生一个实例；(2) 执行过程中出现的返回栈状态是有限的. ConnectBot 中的 *ALC*2 就属于第三类. 从表 2 可以看出，返回栈的 1、3、5 状态相同，2、4 状态相同，每个状态都没有同类型 activity 实例. 继续执行 *ALC*2，有限的返回栈状态依旧会循环出现.

ALC 的分类本质上取决于 ALC 中各类 activity 间启动方式的组合.

4　启动方式敏感的 activity 跳转分析

面向对象的字段敏感指向分析能够帮助构建 Android 应用的 activity 启动图. 本节首先介绍了 Android 平台启动 activity 的组件间通信机制；然后阐述了构建 ALG 前的预处理，包括 AndroidManifest.xml 文件的解析和基于 Android 控制流的辅助主函数构造；最后详述了利用面向对象的字段敏感指向分析构造 ALG 的规则.

4.1　Activity 间通信

Android 平台通过调用 ICC 相关 APIs 来启动 activity，这些 APIs 包括 startActivity()、startActivityForResult()、startActivityIfNeeded()，被记为 *startActivity*(*intent*). 为了全面提取启动关系，在指向分析中考虑了以上三个 APIs.

对象 intent 决定了目标 activity. Intent 可以分为两种类型，显式和隐式. 显式 intent 记录了明确的目标 activity 类型(如图 2(a)的第 13～14 行和 22 行). 隐式 intent 记录了目标 activity 需要执行的动作(如图 2(a)第 8 行)，只有具备执行该动作能力的 activities 才可能被启动. Activity 能够执行的动作被注册在 AndroidMainfest.xml 的 intent 过滤器中(如图 2(a)第 36～38 行). Android 平台提供了 APIs 来帮助 intent 对象记录目标 activity 相关信息，比如 setComponent()，setAction(). 对于显式 intent，目标 activity 类型被存储在字段 Intent.mCompont.mClass 中. 对于隐式 intent，目标 activity 需要执行的动作被存储在字段 Intent.mAction 中.

为 intent 设置标志字段能够在启动 activity 达到特殊的效果，比如“FLAG_ACTIVITY_CLEAR_TOP”标志要求系统首先查找返回栈中与目标 activity 同类型的实例，若存在，则将返回栈中该实例以上的实例全部销毁. Android 平台提供了 setFlags() 和 addFlags() 来设置和添加标志. 标志被存储在字段 Intent.mFlags 中.

4.2　预处理

预处理包括两个部分，第一部分是解析 AndroidManifest.xml 文件，第二部分是为指向分析构建辅助 main(). 从 AndroidMainfest.xml 中提取的重点信息包括：(1) Android 应用包名；(2) 注册的 activity 类名集合 *ACT*；(3) 各 activity 类的 intent 过滤器；(4) 部分返回栈相关配置信息，包括各 activity 类的 taskAffinity 和 launchMode.

与传统的 Java 程序只有一个程序入口(main 函数)不同，Android 应用程序为事件驱动，由大量回调函数构成. 这些回调函数包括生命周期回调和非生命周期回调. 非生命周期回调又包括系统回调和 GUI 回调. 为了进行指向分析，需要构建一个辅

助主函数将这些回调函数依照控制流组织起来.构建辅助 main()时,由于指向分析前 activity 间的启动关系并不可知,因此构建辅助 main()时只需考虑单个 activity 类内部的回调函数调用顺序.图 5 展示了辅助 main()的结构.对于 activity 生命周期回调函数:onCreate()和 onDestroy()分别指示 activity 生命周期的开始和结束;onStart()和 onStop()分别在 activity 可视与不可视时自动调用;onResume()和 onPause()则指示了可操作周期,用户可在这段周期内操作 activity,比如点击按钮.系统自行触发的系统回调(比如 onLocationChanged())和用户行为触发的 GUI 回调(比如 onClick())均在可交互周期内被调用.为 Android 应用构建辅助主函数时,只需要考虑被重写的回调函数.

```
1  void main(){
2    //实例化所有activity类, act_i ∈ ACT
3    r_1 = new act_1();
4    r_2 = new act_2();
5    …
6    //对每个activity类，调用其回调
7    INVOKECALLBACK(r_1);
8    INVOKECALLBACK(r_2);
9    … }
10 void INVOKECALLBACK(Activity r){
11   r.onCreate();
12   //activity可视周期
13   { r.onRestart();
14     r.onStart();
15     r.onResume();
16     //activity可交互周期
17     { //其它被重写的非activity生命周期回调函数
18       … }
19     r.onPause();
20     r.onStop(); }
21   r.onDestroy(); }
```

图 5 辅助主函数结构

4.3 面向对象的字段敏感指向分析

4.3.1 符号定义

图 6 列出了 ALG 构建方法使用的符号.第一部分是 Java 程序的抽象.C 表示类和接口的集合,ACT 表示所有 activity 类的集合.每一个方法体都可表示成语句 s.方法体可由对象实例化、赋值、字段读、字段写、方法调用五种原子语句组成.对象实例化意味着堆内存的分配.第二部分定义了指向分析过程中用到的相关方法.$Pt(r)$表示指针 r 所指向的对象集合.$FPt(r,f)$给出指针 r 的字段 f 所指向的对象集合.第三部分给出确定启动方式所用到的符号.被启动 activity 的 launchMode,taskAffinity,启动时 ICC 方法传递的 intent 对象的标志和 finish()方法的调用共同影响着返回栈状态,决定了启动方式.根据 Intent 标志对返回栈的影响将其分为无效标志($IVFLAG$)、有效标志($VFLAG$)和其它($OFLAG$).由于现阶段本文提出的返回栈建模方法无法模拟多个返回栈的交互,因此本文把引起多个返回栈的交互的 intent 标志称为无效标志.这些标志会在未来工作中考虑.第四部分给出了 ALG 相关符号.通过指向分析所构造的 activity 启动图中每个节点都为一个 activity 类的对象.

程序相关符号

类：$c\in C, act\in ACT, ACT\subseteq C$

方法：$m\in M$

变量：$x,y,z,num,cpt,action\in R$

字段：$f\in F$

语句：$s,s_i\in S$

$s::=s_1;s_2\mid$

$x:=\text{new } c()\mid$ 对象实例化

$x:=y\mid$ 赋值语句

$x:=y.f\mid$ 字段读

$x.f:=y\mid$ 字段写

$x:=y.m(z)$ 方法调用语句

对象：$o^i,o^j,o^k,o^l\in O$

指向分析相关方法

Pt：$R\rightarrow O$

FPt：$R\times F\rightarrow O$

启动方式相关符号

launchMode：$lm\in LM=\{\text{Std},\text{Top},\text{Task},\text{Inst}\}$

taskAffinity：$taf\in TAF$

intent 标志：$flag\in FLAG=VFLAG\cup IVFLAG\cup OFLAG$

有效标志：$VFTAG=\{\text{f\_singleTop},\text{f\_clearTop},\text{f\_reorderToFront}\}$

finish()调用：$fin\in FIN=\{\text{true},\text{false}\}$

启动方式：$lt\in LT=\{\text{standard},\text{singleTop},\text{standard-clearTop},\text{singleTop-clearTop},\text{reorderToFront},\text{singleTask},\text{standard}\mid\text{finish}\}$

ALG 相关符号

ALG：$ALG=(N,E,st)$

节点：$o^i\in N\subseteq O$

边：$o^i\xrightarrow{lt}o^j\in E\subseteq N\times N\times LT$

图 6 符号定义

4.3.2 面向对象的字段敏感指向分析规则

对辅主函数进行指向分析,[RULE-1]~[RULE-6]给出了五条原子语句所对应的指向规则.初始时 ALG 的节点集合 N 和边的集合 E 都为空.[RULE-1]对应于类实例化语句.[RULE-2]专门针对于 activity 类的实例化.每个 activity 对象的创建都为 ALG 新增一个节点.[RULE-3]~[RULE-5]分别对应赋值、字段读、字段写语句.[RULE-6]对应了方法的调用语句.每个方法 m 包含其方法名,形参 $this_m$ 和 p_m,返回值 r_m.对方法 m 进行指向分析可拆解为对方法体 s_m 及辅助语句 $x:=r_m$ 的分析.

$$\frac{s_i: x:=\text{new } c()\quad c\notin ACT}{o^i\notin Pt(x)}\qquad[\text{RULE-1}]$$

$$\frac{s_i: x:=\text{new } act()\quad act\in ACT}{o^i\notin Pt(x)\quad o^i\in N}\qquad[\text{RULE-2}]$$

$$\frac{x:=y\quad o^i\in Pt(y)}{Pt(y)\subseteq Pt(x)}\qquad[\text{RULE-3}]$$

$$\frac{x:=y.f \quad o^i\in Pt(y)}{FPt(o^i,f)\subseteq Pt(x)} \quad [\text{RULE-4}]$$

$$\frac{x.f:=y \quad o^i\in Pt(x)}{Pt(y)\subseteq FPt(o^i,f)} \quad [\text{RULE-5}]$$

$$\frac{x:=y.m(z) \quad o^i\in Pt(y) \quad o^j\in Pt(z)}{s_m;x:=r_m \quad o^i\in Pt(this_m) \quad o^j\in Pt(p_m)} \quad [\text{RULE-6}]$$

[RULE-7]～[RULE-10]是组件间通信相关语句指向规则. [RULE-7]对 intent 标志进行分析. Intent 标志为整数类型. Intent 对象 o^i 通过调用方法 $addFlags()$ 将标志值存储在 $mFlags$ 字段；也可以通过调用方法 $setFlags()$，将添加的新标志与已有标志进行按位或运算并存储在 $mFlags$ 字段. [RULE-8]和[RULE-9]分别对显式和隐式 intent 进行分析. 为 intent 对象 o^i 构造了字符串类型的辅助字段 $targetClassName$ 来存储目标 activity 类名. [RULE-9]中的 $FindTarget()$ 方法返回能够执行 o^j 动作的目标 activity 集合. [RULE-10]分析了所有 activity 启动相关 APIs. 在该项规则中，$this$ 表示一个 activity 实例；启动 activity 所传递的 intent 对象 o^j 的字段 $targetClasssName$ 记录了目标 activity 类名 o^k；$GetActObj()$ 根据类名获取其在堆内存中的唯一实例 o^l；$GetFlags()$ 将 intent 对象 o^j 的字段 $mFlags$ 所存储的整数转化为标志集合并返回；$GetLM()$ 和 $GetTA()$ 分别获取目标 activity 对象的 launchMode 和 taskAffinity；$GetFIN()$ 获取正被分析的方法 m 的方法体有没有调用 finish()；$(o^i,config)\mapsto o^l$ 表示 o^i 以启动方式相关配置 $config$ 下启动 o^l.

$$\frac{intent.mFlags:=num \quad o^i\in Pt(intent) \quad o^j\in Pt(num)}{\langle o^i,mFlags\rangle=o^j} \quad [\text{RULE-7}]$$

$$\frac{intent.mComponent:=cpt \quad o^i\in Pt(intent) \quad o^j\in Pt(cpt)}{FPt(o^j,mClass)\subseteq FPt(o^i,targetClassName)} \quad [\text{RULE-8}]$$

$$\frac{intent.mAction:=action \quad o^i\in Pt(intent) \quad o^j\in Pt(action)}{FindTarget(o^j)\subseteq FPt(o^i,targetClassName)} \quad [\text{RULE-9}]$$

$$\frac{\begin{array}{c}this.startActivity(intent)\\ o^i\in Pt(this) \quad o^j\in Pt(intent)\\ o^k\in FPt(intent,targetClasssName)\\ o^l=GetActObj(o^k) \quad flags=GetFlags(\langle o^j,mFlags\rangle)\\ lm=GetLM(o^l) \quad taf=GetTA(o^l) \quad fin=GetFIN(m)\end{array}}{(o^i,flags,lm,taf,fin)\mapsto o^l} \quad [\text{RULE-10}]$$

4.3.3　Activity 启动边构建规则

[RULE-11]～[RULE-18]给出了根据 activity 间启动关系和配置信息生成 ALG 启动边的规则. [RULE-11]对应于无效配置的判定，其中 $pkName$ 是当前 Android 应用的包名. 当 taf 与当前包名不相同时，若系统安装了与 taf 相同的应用，则会在该应用的返回栈中启动目标 activity. 无效配置会引起多个返回栈的交互，没有为无效配置的 activity 启动生成启动边. [RULE-12]～[RULE-18]对应有效边的生成，其中方法 $ConfigVali()$ 根据规则 11 来判定配置信息是否有效，若有效，则返回 true.

$$\frac{\begin{array}{c}(o^i,flags,lm,taf,fin)\mapsto o^l\\ (lm \text{ is Inst})\vee\neg(taf \text{ is } pkName)\vee\\ (\exists flag\in flags: flag\in IVFLAG)\end{array}}{o^i\xrightarrow{\text{invalide}}o^l} \quad [\text{RULE-11}]$$

$$\frac{\begin{array}{c}(o^i,flags,lm,taf,fin)\mapsto o^l\\ (ConfigVali(flags,lm,taf))\wedge(lm \text{ is Task})\end{array}}{o^i\xrightarrow{\text{singleTask}}o^l\in E} \quad [\text{RULE-12}]$$

$$\frac{\begin{array}{c}(o^i,flags,lm,taf,fin)\mapsto o^l\\ (ConfigVali(flags,lm,taf))\wedge\\ (\exists flag\in flags: flag \text{ is f\_clearTop})\wedge\\ ((\exists flag\in flags: flag \text{ is f\_singleTop})\vee(lm \text{ is Top}))\end{array}}{o^i\xrightarrow{\text{singleTop-clearTop}}o^l\in E} \quad [\text{RULE-13}]$$

$$\frac{\begin{array}{c}(o^i,flags,lm,taf,fin)\mapsto o^l\\ (ConfigVali(flags,lm,taf))\wedge\\ (\exists flag\in flags: flag \text{ is f\_clearTop})\wedge\\ \neg((\exists flag\in flags: flag \text{ is f\_singleTop})\vee(lm \text{ is Top}))\end{array}}{o^i\xrightarrow{\text{standard-clearTop}}o^l\in E} \quad [\text{RULE-14}]$$

$$\frac{\begin{array}{c}(o^i,flags,lm,taf,fin)\mapsto o^l\\ (ConfigVali(flags,lm,taf))\wedge\\ (\exists flag\in flags: flag \text{ is f\_reorderToFront})\end{array}}{o^i\xrightarrow{\text{reorderToFront}}o^l\in E} \quad [\text{RULE-15}]$$

$$\frac{\begin{array}{c}(o^i,flags,lm,taf,fin)\mapsto o^l\\ (ConfigVali(flags,lm,taf))\wedge\\ \neg(\exists flag\in flags: flag \text{ is f\_clearTop})\wedge\\ ((\exists flag\in flags: flag \text{ is f\_singleTop})\vee(lm \text{ is Top}))\end{array}}{o^i\xrightarrow{\text{singleTop}}o^l\in E} \quad [\text{RULE-16}]$$

$$\frac{(o^i, flags, lm, taf, fin) \mapsto o^l \quad (ConfigVali(flags, lm, taf)) \wedge (\forall flag \in flags: flag \notin VFTAG) \wedge fin \vee (lm \text{ is Std}) \quad o^i \xrightarrow{\text{standard}|\text{finish}} o^l \in E}{\text{[RULE-17]}}$$

$$\frac{(o^i, flags, lm, taf, fin) \mapsto o^l \quad (ConfigVali(flags, lm, taf)) \wedge (\forall flag \in flags: flag \notin VFTAG) \wedge \neg fin \vee (lm \text{ is Std}) \quad o^i \xrightarrow{\text{standard}} o^l \in E}{\text{[RULE-18]}}$$

5 ALC 静态分析框架

本文提出并实现了 ALC 自动静态分析框架 ALCAnalyzer，如图 1 所示. 该框架以 Android package (简称 apk)作为输入，包括三个核心组件：ALG 生成器、ALC 生成器和返回栈模拟器，返回栈模拟器中集成了 ALC 分类器. ALG 生成器实现了第 4 节基于面向对象字段敏感指向分析的 ALG 构建方法，有效启动方式可随着对无效配置的深入研究和对多个返回栈变迁建模而得到扩展. ALC 生成器以 ALG 作为输入，输出 Android 应用程序中的 ALC 集合. 返回栈模拟器依据表 1 对 7 种不同有效方式启动 activity 带来的返回栈变化进行模拟. ALC 分类器以 ALC 集合作为输入，输出每个 ALC 的分类结果并预测 Android 应用程序在运行过程中能够产生同类型实例的 activity 类的集合. 本节对 ALC 生成器和 ALC 分类器进行详述.

5.1 ALC 生成器

ALC 生成器实现了如算法 1 所示的基于 ALG 的 ALC 生成算法. 算法 1 以 ALG 的 main activity 节点为起点对 ALG 进行递归深度优先遍历来探测 ALC. *trace* 记录了访问过的节点，当 *trace* 中已经存在即将要访问的当前节点 *currentNode* 时，表明存在一个 ALC(第 4～8 行). 第 5 行 *ALC* 不仅记录了一个初级回路 *cycle* 和 *cycle* 的起始节点 *startNode*，还记录了从 *ALG* 起始节点到 *startNode* 的路径. *cycle* 即为 *trace* 中 *currentNode* 之后的节点序列，而 *preCycle* 为 *currentNode* 之前的节点序列. 使用第 6 行的 *AddALC*()向 *ALCs* 添加新的 *ALC* 时，若 *ALCs* 中已经存在与 *ALC* 结构相同的元素，则保留具有较短 *preCycle* 的 ALC. 若没有发现 ALC，那么继续访问当前节点的后继节点(第 10～13 行). 第 10 行通过 *GetSuccessors*()获取当前节点的后继节点集合，第 11～13 行处理各后继节点.

如果最终的有效 *ALC* 集合为空，说明该应用在使用过程中不可能出现同类型 activity 实例.

算法 1. ALC 集合生成算法.

输入：*ALG*

输出：Android 应用的有效 ALC 集合 *ALCs*

1. *ALCs*←∅，initialize *trace* with an empty list;
2. *Traverse*(*entryNode*(*ALG*));
3. PROCEDURE *Traverse*(*currentNode*)
4. IF *currentNode* in *trace* THEN
5. *ALC*←*GetALC*(*currentNode*, *trace*);
6. *AddALC*(*ALCs*, *ALC*);
7. RETURN;
8. END IF
9. add *currentNode* to *trace*;
10. *successors*←*GetSuccessors*(*currentNode*)
11. FOREACH *nextNode*∈*successors* DO
12. *Traverse*(*nextNode*);
13. END FOR
14. remove the last node from *trace*;
15. END PROCEDURE

5.2 ALC 分类器

依据 3.3 节阐述的各类 ALC 特征，可以对 ALC 类别进行判定，方法为：(1) 若重复执行 ALC 过程中发现某次执行结束后的返回栈状态与前一次结束后相同，则肯定是第二类或第三类；(2) 根据执行过程中是否产生同类型 activity 实例来区分第二类和第三类 ALC；(3) 重复执行第二类 ALC，返回栈中最多有 $2\times Length(ALC)$ 个实例，重复执行第三类 ALC，返回栈中最多有 $Length(ALC)$ 个实例，因此当返回栈中实例个数大于 $2\times Length(ALC)$ 时，即可判定其为第一类.

具体判别过程如算法 2 所示. 4～8 行模拟了一次 ALC 的执行过程，*tStack* 和 *oldStack* 分别记录当前返回栈状态及上次执行结束后返回栈状态. 9～17 行根据返回栈状态判别 ALC 类型，第 9 行中 *SameStates* 用来判定两个返回栈状态是否相同.

算法 2. ALC 分类算法.

输入：*ALC*; /* 待分析的 ALC */

输出：*ALC* 的类别 *type*，具有多个实例的 activity 类集合 *s*

1. *s*←∅，push startNode of *ALC* into *oldStack* and *tStack*; /* 初始化 *s*，*oldStack* 和 *tStack* */
2. WHILE true DO
3. *tNode*←*startNode* of *ALC*;
4. FOR *i* from 1 to *Length*(*ALC*) DO

```
5.      tNode←GetNextNode(tNode);
                    /* 更新 tNode 为其后继节点 */
6.      tStack←UpdateStack(tStack,tNode);
        /* 模拟 tNode 的启动,并更新 tStack 状态 */
7.      s←CheckStack(tStack);/* 检查 tStack 状态,
          返回具有多个实例的 activity 类的集合 */
8.    ENFOR
9.    IF SameStates(tStack,oldStack) THEN
10.     IF s≠∅ THEN
11.       type←2,BREAK;
12.     ELSE
13.       type←3,BREAK;
14.     END IF
15.   ELSE IF Length(tStack)>2×Length(ALC)
      THEN
16.     type←1,BREAK;
17.   END IF
18.   oldStack←tStack;
19. END WHILE
```

6　实验与结果

本文基于Android程序静态分析工具包Gator[5,8]实现了 Android 应用 activity 启动图自动构建、activity 启动环定位及分类的原型系统 ALCAnalyzer. 本节将展示 ALCAnalyzer 的有效性评估结果、对 Android 开源应用市场 F-Droid 中 1179 个应用的分析结果以及 ALCAnalyzer 对 20 个 Google Play 市场应用的分析结果.

6.1　实验设置

实验设计主要为了解答以下两个问题：

(1) ALCAnalyzer 有效性如何，即能否准确构建 ALG、能否准确定位 ALC、能否对 ALC 正确分类.

(2) 在真实应用场景中，ALCAnalyzer 能否在以下两方面发挥作用：避免非法路径生成；为管理返回栈提供有效信息.

针对问题一，本文以 Android 开源应用市场 F-Droid 中的开源应用作为实验对象. 截止至 2017 年 12 月 15 日，在 F-Droid 中下载到 1381 个 Android 开源应用. 由于每个应用包括多个版本，根据更新时间选取最新版本为实验对象. 人工将这些应用分成 8 类：新闻、阅读类；工具类；游戏类；旅游购物类；视频播放、摄影、社交娱乐类；辅助插件类；其它. 每个 app 只在一个类别中. 辅助插件类的 202 个辅助应用需要与其它 apps 联合使用，其工程结构与典型 Android 应用不同. 因此在 apk 反编译过程中会遇到资源解析不正确或在预分析阶段无法找到 main activity 的问题，导致无法进行下一步分析. 而剩余 6 类的 1179 个 apps 则能成功完成预分析过程，表 4 第 2 列展示这 6 类各自包含的应用个数. 根据开源应用实验结果，我们不仅分析了 ALCAnalyzer 的有效性，还分析了开源应用所使用的各种启动方式的分布、不同长度 ALC 的分布、各类应用包含 ALC 的分布以及 ALC 静态分析各个阶段的时间代价.

针对问题二，从 Google play 中选取了 20 个高下载量的 Android 应用为实验对象. 表 5 第 2 列展示了这些应用的名称.

6.2　系统有效性分析

6.2.1　有效性评估实验过程

使用 ALCAnalyzer 对 1179 个开源应用进行分析，设置最长分析时间阈值 30 min. 若超过该阈值，则分析失败. 表 3 中步骤“ALCAnalyzer”指示了 ALCAnalyzer 的总体分析结果：“包含 ALC”和“ALC”分别展示包含 ALC 的应用个数及 ALC 的总个数；3～5 列分别展示了包含第一类、第二类、第三类 ALC 的应用个数，一个应用可能包含多种类型的 ALC；7～9 列分别展示了第一类、第二类、第三类 ALC 的总数. 针对自动检测出的 167 个 apk 中的 1522 个 ALC，分配本文作者中的 3 名有 Android 开发经验的组员进行人工验证. 首先，3 人各自分配数量相当的应用，并花费 2 周时间完成第一轮检查；然后，3 人两两交换上一轮检查的应用，再花费 2 周时间进行人工验证；对于两轮验证结果不一致的地方，3 人共同讨论并确认最后结果. 人工验证过程为：

(1) 查看源码，确认 ALCAnalyzer 检测出的 ALC 是否真实存在. 若一个 ALC 中包含不存在的启动关系，则将该 ALC 删除. 比如，ALCAnalyzer 为某 apk 生成的 ALG 包含启动关系 $A \xrightarrow{standard} B$，但人工检查时发现该启动关系不存在，则将全部包含该边的 ALC 删除. 我们称该例中的 $A \xrightarrow{standard} B$ 为误报边，那些包含误报边的 ALC 即为被误报 ALC. 对确认存在的 ALC，ALCAnalyzer 分类结果如表 3 步骤“Step 1”所示.

(2) 针对确认存在的 ALC，手工提取 ALC 的各 activity 启动方式，结合 Android 应用动态运行时产生同类型 activity 实例情况，人工确认其分类. 表 3 步骤“Step 2”是人工确认的 ALC 分类结果. ALCAnalyzer 共将 14 个确认存在的 ALC 错分，具体情况为：

①1 个第一类错分为第三类；
②1 个第一类错分为第二类；
③1 个第二类错分为第三类；
④7 个第三类错分为第一类；
⑤4 个第三类错分为第二类.

人工确认后各类应用包含三类 ALC 的详细情况如表 4 所示. 表 4 第 3 列展示了各类别中被成功分析的应用数量. "ML"列展示了各类应用中最长 ALC 的长度. "AML"为平均最大长度，表示对于一类应用中各应用最长 ALC 的长度均值，通过以下公式计算：

$$AML=\sum_{i=0}^{N}\mathrm{Max}(l_{i1},l_{i2},\cdots,l_{ij},\cdots,l_{iM_i})/N,$$

其中 N 为该类中 app 的数目，l_{ij} 为该类第 i 个应用中第 j 个 ALC 的长度，M_i 为该类应用第 i 个应用包含 ALC 的总数.

表 3 ALCAnalyzer 对 1179 个开源应用的分析结果与人工验证结果

步骤	包含 ALC	包含 TYPE1	包含 TYPE2	包含 TYPE3	ALC	TYPE1	TYPE2	TYPE3
ALCAnalyzer	167	97	1	94	1522	823	5	694
Step 1	154	85	1	88	1306	667	5	634
Step 2	154	82	1	88	1306	662	1	643

表 4 各类 Android 应用包含三类 ALC 的详细情况

类别	数量	成功分析	包含 ALC	包含 TYPE1	包含 TYPE2	包含 TYPE3	ALC	TYPE1	TYPE2	TYPE3	ML	AML
新闻阅读	139	130	23	13	1	13	257	127	1	129	8	2.58
工具	399	390	44	21	0	26	332	281	0	51	4	2.01
游戏	132	130	12	7	0	6	150	17	0	133	7	2.15
购物	88	80	15	8	0	8	60	37	0	23	4	2.27
娱乐	180	163	28	16	0	15	232	108	0	124	8	2.04
其它	241	230	32	17	0	20	275	92	0	183	6	2.18
总数	1179	1123	154	82	1	88	1306	662	1	643	—	—

6.2.2 有效性评估实验结果

结果显示，超过 95%的 app 能够成功完成分析过程，极少数应用由于预分析阶段时间过长或 ALG 过于复杂，导致系统超时. 根据成功完成分析的 Android 应用的实验结果，使用以下三个指标来评估 ALCAnalyzer 的有效性.

(1) ALC 误报率 $ALC\_FPR=1-ALC\_CN/ALC\_DN=14.19\%$，其中 ALC_CN 指被确认的 ALC 数目，ALC_DN 指被检测出来的 ALC 数目. 检测出 ALC 中有 216 个不存在. 产生 ALC 误报的原因是 ALG 生成器在静态分析时对上下文不敏感，因此在构建 ALG 时为不存在的启动关系构建了启动边. 例如，为社交新闻类应用 Reddinator 构建的 ALG 中包含启动关系：

$$\text{WebViewActivity}\xrightarrow{\text{standard}}\text{ViewALLSubredditsActivity},$$

但人工验证该启动关系不存在，导致与该误报边相关的 33 个第一类 ALC 全部为误报.

(2) 分类错误率 $CER=CEN/ALC\_CN=1.07\%$，其中 CEN 为分类错误的 ALC 数目. CER 能反映提取的 activity 启动方式和 ALC 分类的准确性. 人工检查发现的 14 个 ALC 错分均为 activity 启动方式提取错误造成. 比如一款管理笔记的应用 OI Notepad 中，DialogHostingActivity 通过调用 FilenameDiaglog 的对象才能启动一个新的 DialogHostingActivity，并且在启动新实例后调用了 finish() 销毁当前 DialogHostingActivity实例. 它是第三类 ALC. 但由于本文静态分析方法暂时还捕捉不到如此精细的控制流，致使 ALCAnalyzer 找到的是第一类 ALC：

$$\text{DialogHostingActivity}\xrightarrow{\text{standard}}\text{DialogHostingActivity}.$$

(3) ALG 精确度：$ALG\_A=ARN/ALC\_DN=84.89\%$，其中 $ARN=ALC\_CN-CEN$，表示被正确分类的 ALC 数目. 构建的 ALG 是否精确体现在两方面：一是启动关系查找是否准确，即源 activity 是否启动了目标 activity；二是启动方式的提取是否准确. ALC 误报源于启动关系查找不准确，ALC 分类错误源于启动方式提取不准确. ALG_A 即为被正确分类的 ALC 占被检测出的总 ALC 的百分比，反映了 ALCAnalyzer 构建 ALG 的精确度.

6.3 开源应用实验结果分析

对成功完成自动分析的 1123 个 Android 应用进行人工检查后统计所有 activity 启动关系的启动方式. 对于启动方式为 singleTop 的启动关系，若其源 activity 与目标 activity 类型不相同，那么它对返回栈的影响与 standard 相同. 因此，我们把以上情况归为 standard. 各启动方式所占比例如图 7. 由于以 singleTop 方式启动的 activity 所占比例仅为

0.34%，所以在图 7 中没有显示. 可以发现，37.87% 的启动关系使用了特殊启动方式. 其中，singleTask、standard-clearTop、singleTop-clearTop、singleTop、reorderToFront 若不在 activity 启动环中，其作用都与 standard 对返回栈的作用相同，这些启动方式比例高达 18.94%. 伴随 finish() 的启动方式不在 ALC 中依然能够发挥作用，其所占比例为 18.93%. 对于以特殊启动方式启动 activity 所触发的返回栈状态变化，启动方式不敏感的 activity 跳转分析默认启动方式为 standard 的做法无法对其准确模拟，从而导致非法返回栈变迁路径的产生. 这体现出构建启动方式敏感的 activity 跳转分析的必要性.

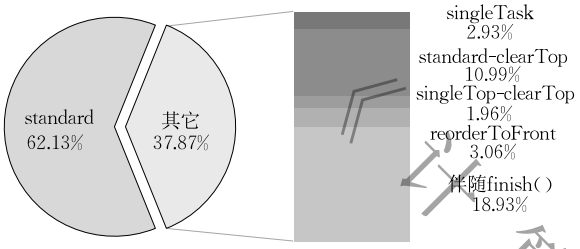


图 7　1123 个应用中各 activity 启动方式所占比例

从表 4 可以得出，在成功完成测试的应用中，13.7%包含 ALC，可见 ALC 是开发者为了完成特定功能而广泛使用的结构. 包含 ALC 的应用中，53.25%包含第一类 ALC，57.14%包含第三类 ALC. 所有 ALC 中，50.67%为第一类，49.23%为第三类. 唯一的第二类 ALC 在应用 Reddinator 中：SubmitActivity 以 standard-clearTop 方式启动 ViewRedditActivity，并且 ViewRedditActivity 以 standard 方式回到 SubmitActivity，致使返回栈中有两个 SubmitActivity 的实例. 包含第一类和(或)第二类 ALC 的应用在使用过程中可能产生多个同类型 activity 实例，这样应用所占比例高达 7.39%. 这也验证了对 ALC 进行系统研究的必要性.

图 8 展示了不同长度 ALC 的个数. 可以看出，长度为 1 或 2 的 ALC 所占比例为 37.74%，长度为 3 或 4 的 ALC 所占比例为 36.57%. 长度在 5 以上的 ALC 也占有相当比例，为 25.69%. ALC 长度越长，开发人员在开发过程中就越难发现，越不容易使用合适的启动方式来管理返回栈. 这也反映出自动定位 ALC 的必要性.

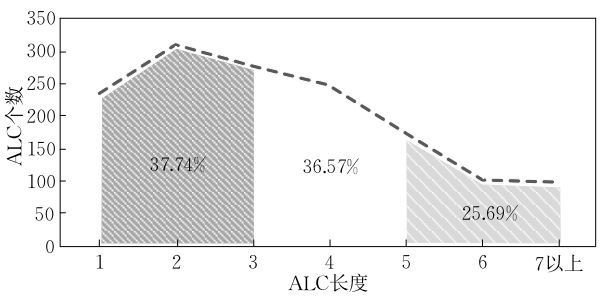


图 8　不同长度 ALC 的个数

图 9 中柱形总体高度表示包含 ALC 的应用在其分类中所占的百分比，深色部分表示包含第一类和(或)第二类 ALC 的百分比. 可以发现购物相关和阅读相关的应用包含 ALC 的比例以及包含第一类和(或)第二类 ALC 的比例高于其它，而游戏和工具类应用则较低. 各类 ALC 平均最大长度的折线图趋势与柱状图的趋势相似. 阅读相关和购物相关类的 AML 高于其它应用. 这也反映出购物相关和阅读相关应用的功能需求更需要 activity 启动环结构的支持，其软件中的启动关系也较为复杂，更容易出现第一类和第二类 ALC.

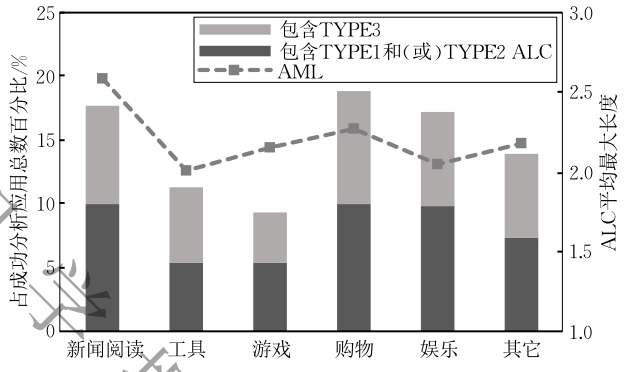


图 9　每类应用包含第一类和(或)第二类 ALC 的百分比及每类应用中 ALC 的平均最大长度

6.4　时间分析

本文把 ALCAnalyzer 对 apk 的分析过程分为五个阶段：反编译、预分析、生成 ALG、生成 ALC 集合、ALC 分类. 统计包含 ALC 的 167 个应用的分析时间，计算得到平均总耗时为 54.1 s，各阶段平均耗时依次为：12.5 s、30 s、11 s、14.18 ms、6.92 ms. 可见 ALCAnalyzer 非常快捷. 另外，反编译，预分析和生成 ALG 三个阶段占据分析过程的主要时间，另外两个阶段用时极少.

图 10 将分析各 apk 过程中反编译，预分析和生成 ALG 这三个阶段的时间用平滑曲线连接起来. 可以发现反编译耗时保持在稳定水平，与 apk 本身无关. 预分析和生成 ALG 阶段的耗时曲线波动很大，且呈现一致的变化趋势. 原因是预分析阶段需要深度遍历逆向工程得到的 smali 代码，生成 ALG 阶段需要查找启动、销毁 activity 及 intent 标签设置

等关键语句，所以 apk 间代码复杂度的差异造成了分析时间的差异.

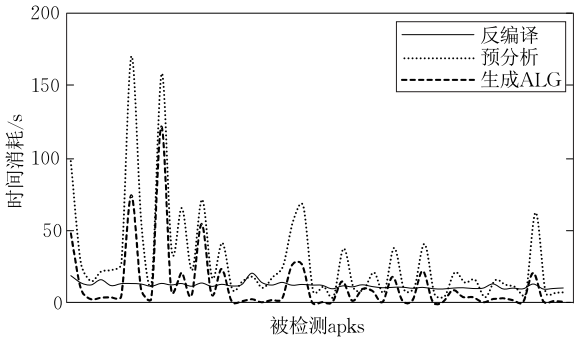


图 10 分析 apk 的时间消耗

6.5 ALCAnalyzer 应用示例

为了验证 ALCAnalyzer 能否在实际应用场景中发挥作用，使用其对 20 个 Google Play 中的热门应用进行分析，结果如表 5 所示.“N”、“E”分别表示 ALG 的节点数和边数，“$E_{special}$”表示配置了特殊启动方式的边数，“ALC”表示包含 ALC 的数目，“ML”表示各应用最长 ALC 的长度，“Time”表示分析时间，其中不包括反编译时间. 表 5 中，第 1 至第 12 个应用包含 ALC 且使用了特殊启动方式，第 13 至第 16 个应用包含 ALC 但没有使用特殊启动方式，第 17 至第 20 个应用不包含 ALC.

表 5 ALCAnalyzer 对 20 个 Google Play 应用的分析结果

	应用名称	N	E	$E_{special}$	ALC	ML	LC	$MIAC$	路径(l=4)			Time/s
									$Lt\_sensitive$	$Lt\_insensitive$	Δ	
1	Xmp Mod Player	8	9	4	2	2	1.00	0	61	69	0.12	34.45
2	KouChat	5	4	2	2	2	1.00	0	16	20	0.20	13.19
3	Scrum Chatter	6	6	1	1	1	1.00	0	22	32	0.31	37.27
4	MPDroid	15	15	3	1	2	0.20	0	9	12	0.25	11.84
5	Wifi Fixer	4	9	9	3	2	1.00	0	41	87	0.53	76.82
6	Shorty	7	5	1	1	2	1.00	0	23	26	0.12	123.01
7	Transportr	8	5	2	1	1	0.40	0	8	18	0.56	22.10
8	Metrodroid	13	10	5	2	2	0.50	0	25	29	0.14	20.23
9	BeeCount	8	14	13	4	4	0.50	0	44	48	0.08	100.60
10	NetGuard	7	4	1	1	1	1.00	0	9	60	0.85	43.73
11	Port Knocker	6	5	3	2	2	0.80	0	16	20	0.20	33.14
12	Offine Calendar	3	3	1	1	2	1.00	0	9	11	0.18	27.16
13	BART Runner	4	5	0	1	2	0.80	2	38	38	0	84.67
14	OpenSudoku	10	12	0	3	2	0.92	2	167	167	0	71.72
15	Quran	12	14	0	1	2	0.86	1	608	608	0	104.15
16	Rental Calc	12	15	0	1	2	1.00	1	264	264	0	321.68
17	Tickmate	10	11	0	0	0	1.00	0	285	285	0	45.00
18	SipDroid	13	12	0	0	0	0.25	0	13	13	0	74.38
19	ParkenDD	6	8	0	0	0	0.50	0	64	64	0	80.55
20	Graph 89	5	5	0	0	0	1.00	0	73	73	0	220.95

若 app 使用了无效配置，则其 ALG 为非连通图. 因此，在定位 ALC 的时候，其最大搜索范围是与 main activity 节点相连通的图.“LC”列则表示 ALC 生成器的最大搜索范围对于整个 ALG 的有效启动边覆盖率. 20 个应用的“LC”均值为 78.65%，可以反映出所构建的 ALG 对于启动关系的覆盖较为全面，生成的 ALC 集合也较为齐全.

“$MIAC$”列表示能够产生同类型 activity 实例的 activity 类的个数. 可以发现第 1 至第 12 个应用虽然包含 ALC，但由于使用了特殊启动方式，使得应用在使用过程中不会产生同类型 activity 实例. 第 17 至第 20 个应用不包含 ALC，所以即使没有使用特殊启动方式，也不会产生同类型 activity 实例. 第 13 至第 16 个应用包含 ALC 且没有使用特殊启动方式，使得当这些 ALC 被重复执行时，至少有一个 activity 类在某一返回栈状态中具有多个实例. 用户多次执行这些 ALC 会造成返回栈中 activity 实例过多，影响用户体验. 因此第 13 至第 16 个应用中的 ALC 应该引起开发人员的重视，通过调整 activity 启动方式或限制返回栈存放同类型 activity 实例数目等措施来优化用户体验，提升软件质量.

当前 Android 应用测试常通过遍历 GUI 模型路径生成测试用例[5,7]. 通过模拟返回栈状态能够帮助判断返回动作的目标 activity，从而有效避免非法路径[6]. ALCAnalyzer 中的返回栈模拟器组件不仅能够进行 ALC 的分类，还能基于 ALG 生成返回栈状态变迁图. 由于包含第一类 ALC 应用的返回栈状态无穷多，所以对任一 activity 类来说，设定返回栈中最多有该 activity 类的 k 个实例. 图 3(a)和(b)分别是当 k=2 时，启动方式敏感和启动方式不

敏感的返回栈状态模拟技术为 *ALC*2 构建的返回栈状态变迁图.

通过遍历返回栈状态变迁图，生成形式为“$p=e_1,e_2,\cdots,e_l$”的路径集合，其中 l 为路径长度.“*Lt_sensitive*”列和“*Lt_insensitive*”列分别表示遍历启动方式敏感和不敏感的返回栈状态变迁图($k=5$)产生的长度为 4 的路径数目.“Δ”列表示相较于启动方式不敏感的 activity 跳转分析，ALCAnalyzer 能够避免产生非法路径的比例，计算公式为 $\Delta=(Lt\_insensitive-Lt\_sensitive)/Lt\_insensitive$. 由于第 13 至第 20 个应用没有使用特殊启动方式，因此启动方式敏感和不敏感的返回栈状态变迁图一样，生成的路径数目也一样. 对于第 1 至第 12 个使用特殊启动方式的应用来说，相较于当前启动方式不敏感的 activity 跳转分析，基于 ALG 所构建的返回栈状态变迁图平均能够避免 29.5% 的非法路径. 这能在基于模型的 Android 应用测试工作中大大提高测试准确性和效率.

通过以上实验分析得出，ALCAnalyzer 的意义主要体现在两方面：一是能够对 Android 应用进行启动方式敏感的 activity 跳转分析，生成 ALG，准确模拟返回栈变化并防止非法路径产生；二是能够准确定位 ALC 并预测应用在使用过程中是否会产生同类型 activity 实例，为开发人员管理返回栈提供有效信息.

7　相关工作

为事件驱动的 Android 系统或应用提取特定的模型已经成为理解程序结构和行为，帮助测试 Android 应用的重要手段，因而被广泛应用. 文献[5]处理了 Android 应用在运行过程中可能出现的窗口序列为 Graphical User Interface(GUI)建立了 window 转换图(Window Transition Graph，简称 WTG)，并用于测试用例的生成. 文献[22]表明基于 GUI 模型生成的测试输入对 Android 应用中特定错误的查找效率较高. 文献[23]关注事件交互且提供了语言 P 用于将事件驱动的异步系统表示成状态机. 文献[24]关注 Android 系统等事件驱动系统的消息流机制建模. 文献[8]提出了针对 Android 应用的回调函数控制流图(Callback Control Flow Graph，简称 CCFG)，文献[25]提出从应用中有选择的提取有用控制流模型的方法. 文献[26-27]建立了 Android 应用的回调序列模型. 这些模型均可被用于测试用例生成. 文献[1-4]构建了表示 activity 间跳转关系的 ATG. 相较于 ATG，本文提出的 ALG 的各启动边记录了启动方式.

基于模型的 Android 测试输入生成技术作为三大类 Android 测试输入生成技术之一，在检测特定问题方面效果突出[22]. 在资源泄露检测方面，Guo 和 Zhang 等人[17]提出了一种轻量级的静态分析方法，通过分析函数调用关系和应用的生命周期从而进行检测，并且实现了资源泄露检测工具 Relda 用以验证其方法. 而后在此基础上，Zhang 等人在文献[18]中深入的分析程序内部结构，并利用模型检测分析程序流从而得到资源泄露细粒度的检测结果，并给出了升级版本的 Relda2. 另外，Yan 及 Rountev 等人[7-8]实现了系统地检测 Android 应用中资源泄露问题的系统 LeakDroid，该系统围绕 GUI 的事件序列，提出了“neutral cycle”. 根据其生成的测试用例有效暴露资源泄露. 除此之外，大量研究关注基于模型的 Android 应用 GUI 测试. 文献[2,9,28-30]动态完成 GUI 测试同时构建 GUI 模型以便回归测试，其目标均为提升代码覆盖率. 文献[9]提出多标准的 GUI 模型构建测试技术. 文献[2]静态构建了 activity 变迁图(ATG)作为探索目标 activity 的基础，该模型由启动方式不敏感的 activity 跳转分析构建，模型的边不记录启动方式. 传统 Android 应用测试以动作序列作为测试输入，EHBDroid[31] 为 Android 应用测试提供了新思路：通过直接调用响应 GUI 动作的回调函数，来获取更高的代码覆盖率.

与本文最相关的是对返回栈状态变化进行模拟方面的工作. 文献[5]中提出了 window 栈的概念将只存储 activity 的栈扩展为存储 window 的栈. 但其对 window 栈的模拟只停留在初级的阶段：静态分析过程没有考虑各 activity 的启动方式对返回栈的影响，并不能通过遍历 WTG 路径来真实模拟软件运行过程中窗口栈状态. 本文针对可能产生同类型 activity 实例的 activity 循环启动结构进行研究，不但对 ALC 进行了全面查找，还精确提取了 activity 启动方式，真实模拟返回栈变化，为基于 activity 跳转分析、返回栈状态变迁分析的下游研究提供基础支持.

8　总　结

ALC 是一种 Android 应用开发工程师为了完

成特定功能而普遍使用的结构.为了保证软件质量,常常需要对 app 进行以 activity 跳转分析和返回栈变迁分析为基础的测试工作.由于缺乏对 ALC 特性的系统研究,致使当前 activity 跳转分析无法正确模拟使用特殊启动方式的 Android 应用的返回栈状态变化,导致非法路径的产生.

本文形式化表示了 7 种 activity 启动方式对返回栈的影响,系统分析了重复执行各类 ALC 的表现,提出启动方式敏感的 activity 跳转分析方法、基于 ALG 的 ALC 定位方法以及通过模拟重复执行 ALC 时返回栈状态变化来对 ALC 进行分类的方法.实验结果证明方法的有效性.该工作能够支持基于 activity 跳转分析的上层应用,解决当前 activity 跳转分析对启动方式不敏感、返回栈模拟不正确以及非法路径的问题,并能够为开发人员管理返回栈提供有效信息.

由于 ALCAnalyzer 的实现基于 Gator,因此也继承了如下缺陷:使用了上下文不敏感的方法调用图,造成 ALG 中启动边的误报;只为主线程相关的回调函数控制流建模,可能造成 ALG 构建不完全.另外,ALCAnalyzer 没有模拟以无效配置启动 activity 引起的返回栈变化.因此,下一步的工作主要包括:(1)通过上下文敏感的指向分析来构建更加精确的 activity 启动图;(2)提高提取伴随 finish()启动方式的精度;(3)考虑以无效配置启动 activity 对返回栈的影响,对多个返回栈交互引起的返回栈状态变化建模.

参 考 文 献

[1] Zhang Y, Sui Y, Xue J. Launch-mod-aware context-sensitive activity transition analysis//Proceedings of the International Conference on Software Engineering. Gothenburg, Sweden, 2018: 598-608

[2] Azim T, Neamtiu I. Targeted and depth-first exploration for systematic testing of Android apps//Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. Indianapolis, USA, 2013: 641-660

[3] Mirzaei N, Garcia J, Bagheri H, et al. Reducing combinatorics in GUI testing of Android applications//Proceedings of the International Conference on Software Engineering. Texas, USA, 2016: 559-570

[4] Bhoraskar R, Han S, Jeon J, et al. Brahmastra: Driving apps to test the security of third-party components//Proceedings of the USENIX Security Symposium. San Diego, USA, 2014: 1021-1036

[5] Yang S, Zhang H, Wu H, et al. Static window transition graphs for Android//Proceedings of the International Conference on Automated Software Engineering. Lincoln, USA, 2015: 658-668

[6] Yan D, Yang S, Rountev A. Systematic testing for resource leaks in Android applications//Proceedings of the IEEE International Symposium on Software Reliability Engineering. Pasadena, USA, 2013: 411-420

[7] Zhang H, Wu H, Rountev A. Automated test generation for detection of leaks in Android applications//Proceedings of the International Workshop on Automation of Software Test. Austin, USA, 2016: 64-70

[8] Yang S, Yan D, Wu H, et al. Static control-flow analysis of user-driven callbacks in Android applications//Proceedings of the International Conference on Software Engineering. Firenze, Italy, 2015: 89-99

[9] Baek Y M, Bae D H. Automated model-based Android GUI testing using multi-level GUI comparison criteria//Proceedings of the International Conference on Automated Software Engineering. Singapore, 2016: 238-249

[10] Lee Y K, Bang J Y, Safi G, et al. A SEALANT for inter-app security holes in Android//Proceedings of the International Conference on Software Engineering. Buenos Aires, Argentina, 2017: 312-323

[11] Arzt S, Rasthofer S, Fritz C, et al. FlowDroid: Precise context, flow, field, object-sensitive and lifecycl-aware taint analysis for Android apps//Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation. Edinburgh, UK, 2014: 259-269

[12] Huang W, Dong Y, Milanova A, et al. Scalable and precise taint analysis for Android//Proceedings of the International Symposium on Software Testing and Analysis. Baltimore, USA, 2015: 106-117

[13] Wei F, Roy S, Ou X. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps//Proceedings of the ACM Conference on Computer and Communications Security. Scottsdale, USA, 2014: 1329-1341

[14] Chen Q A, Qian Z, Mao Z M. Peeking into your app without actually seeing it: UI state inference and novel Android attacks//Proceedings of the USENIX Security Symposium. San Diego, USA, 2014: 1037-1052

[15] Lu L, Li Z, Wu Z, et al. CHEX: Statically vetting Android apps for component hijacking vulnerabilities//Proceedings of the International Conference on Computer and Communications Security. Munich, Germany, 2012: 229-240

[16] Gordon M I, Kim D, Perkins J, et al. Information-flow analysis of Android applications in DroidSafe//Proceedings of the Network and Distributed System Security Symposium. San Diego, USA, 2015: 60-75

[17] Guo C, Zhang J, Yan J, et al. Characterizing and detecting resource leaks in Android applications//Proceedings of the

International Conference on Automated Software Engineering. Silicon Valley, USA, 2013: 389-398

[18] Wu T, Liu J, Xu Z, Zhang Y, et al. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. IEEE Transactions on Software Engineering, 2016, 42(11): 1054-1076

[19] Li L, Mcdaniel P, Bartel A, et al. IccTA: Detecting inter-component privacy leaks in Android apps//Proceedings of the International Conference on Software Engineering. Florence, Italy, 2015: 280-291

[20] Octeau D, Jha S, Dering M, et al. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis//Proceedings of the ACM IGPLAN-SIGACT Symposium on Principles of Programming Languages. St. Petersburg, USA, 2016: 469-484

[21] Octeau D, Luchaup D, Dering M, et al. Composite constant propagation: Application to Android inter-component communication analysis//Proceedings of the International Conference on Software Engineering. Florence, Italy, 2015: 77-88

[22] Choudhary S R, Gorla A, Orso A. Automated test input generation for Android: Are we there yet?//Proceedings of the International Conference on Automated Software Engineering. Lincoln, USA, 2015: 429-440

[23] Desai A, Gupta V, Jackson E, et al. P: Safe asynchronous event-driven programming//Proceedings of the ACM SIGPLAN Notices. Seattle, USA, 2013: 321-332

[24] Garcia J, Popescu D, Safi G, et al. Identifying message flow in distributed event-based systems//Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. Saint Petersburg, Russian Federation, 2013: 367-377

[25] Blackshear S, Sridharan M, Sridharan M. Selective control-flow abstraction via jumping//Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. Pittsburgh, USA, 2015: 163-182

[26] Guo C, Dong N, Bai G, et al. App genome: Callback sequencing in Android//Proceedings of the International Conference on Software Engineering. Buenos Aires, Argentina, 2017: 149-151

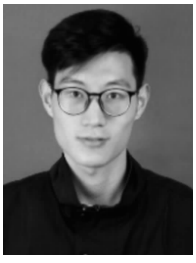
[27] Guo C, Ye Q, Dong N, et al. Automatic construction of callback model for Android application//Proceedings of the International Conference on Engineering of Complex Computer Systems. Dubai, United Arab Emirates, 2016: 231-234

[28] Amalfitano D. Using GUI ripping for automated testing of Android applications//Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. Essen, Germany, 2012: 258-261

[29] Amalfitano D, Fasolino A R, Tramontana P. A GUI crawling-based technique for android mobile application testing//Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops. Berlin, Germany, 2011: 252-261

[30] Machiry A, Tahiliani R, Naik M. Dynodroid: An input generation system for Android apps//Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. Saint Petersburg, Russian Federation, 2013: 224-234

[31] Song W, Qian X, Huang J. EHBDroid: Beyond GUI testing for Android applications//Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. Urbana, USA, 2017: 27-37



LIU Ao, Ph. D. candidate. His main research interests include software analysis techniques, software testing and information security techniques.

GUO Chen-Kai, Ph. D., lecturer. His main research interests include software analysis techniques, information security techniques and model checking.

WANG Wei-Jing, M. S. candidate. Her main research interests include software analysis techniques, software testing and information security techniques.

HOU Xiao-Lei, M. S. candidate. His main research interests include software analysis techniques, software testing and information security techniques.

ZHU Jing-Wen, M. S., assistant experimentalist. Her main research interests include software analysis techniques, software testing.

ZHANG Sen, M. S. candidate. His main research interests include software analysis techniques, software testing and information security techniques.

XU Jing, Ph. D., professor. Her main research interests include software analysis techniques, software engineering, software testing and information security techniques.

Background

Activity, as a key component of Android, is responsible for providing GUI windows for users. Activity instances created during a software runtime are organized in the back stack (activity stack), so that users can return from the current instance to the previous one correctly. Interactions with the top instance of stack trigger activity launchings (activity transitions), which change the back stack state. In order to ensure the quality of software, it is necessary to carry out the testing work based on activity transition analysis for Android applications (apps). The activity transition analysis is a foundation for tasks like dynamic exploration, GUI model construction, vulnerability detection and defects detection. In Android, developers can set up the configurations of activity launchings for back stack management. These different configurations are named launch types in this paper. Our study on various launching types used in F-Droid applications shows that almost 25.31% of the apps use special launch types and 37.87% of activity launchings are configured with special launch types.

However, existing modeling techniques on activity transitions are launch-typ-insensitive, which suppose all activities are launched with standard. Launch-typ-insensitive activity transition analyses cannot simulate the changes of back stack invoked by activity launchings with special launch types correctly, thus resulting in infeasible back stack states transitions and infeasible paths which consist of at least one infeasible path. Such infeasible paths may lead to unexpected errors such as test case execution failure in GUI testing and defects detection which raises researches interests. This raises a key request for launch-typ-sensitive activity transition analysis.

All special launch types aim to request the Android platform to reuse an activity instance of a specific activity class or terminate specific activity instances in back stack to avoid multiple activity instances of one activity class. We found that activity launching cycles (ALCs), which are widely used in Android applications and allow an activity to be launched repeatedly, is necessary for the production of multiple instances of one activity. Therefore, we make a systematic research on activity launching cycles.

In this paper, we propose the activity launching graph (ALG) and the launch-typ-sensitive activity transition analysis to construct ALG automatically. Moreover, we propose and implement a framework named ALCAnalyzer to conduct the static ALC analysis. ALCAnalyzer can generate ALGs for Android applications automatically and generate the set of ALCs based on ALG. It can also simulate the changes of back stack states accurately during the repeated executions of ALCs and predict whether there are multiple instances of any activity class in a back stack state in runtime. Experimental evaluations on 1179 open source Android applications from F-Droid show the high precision of our launch-typ-sensitive activity transition analysis. Experiments on 20 applications from Google Play show that ALCAnalyzer can model the changes of back stack states accurately, and provide the software engineers with more effective information to manage the back stack behaviors.

This research work is supported by the National Natural Science Foundation of China (Grant No. 61402264), the Tianjin Natural Science Foundation (Grant Nos. 17JCZDJC30700 and 19JCQNJC00300), and the Science and Technology Planning Project of Tianjin (Grant Nos. 17YFZCGX00610 and 18ZXZNGX00310).