

2020 届硕士学位论文

分类号: \_\_\_\_\_ 学校代码: 10269

密 级: \_\_\_\_\_ 学 号: \*\*\*\* \* \* \* \*



华东师范大学

East China Normal University

硕士 学位 论文  
MASTER'S DISSERTATION

论文题目:

面向工业界仿冒应用的大规模实证研究

院 系: 计算机科学与技术学院

专业名称: \*\*\*\*\*

研究方向: \*\*\*\* \* \* \*

指导教师: \*\*\* \*\*

学位申请人: \*\*\*

2020 年 5 月

Thesis for master's degree in 2020      University Code:      10269

Student ID:      \*\*\*\*

# EAST CHINA NORMAL UNIVERSITY

## A LARGE-SCALE EMPIRICAL STUDY ON INDUSTRIAL FAKE APPS

Department:      School of Computer Science and Technology

Major:      \*\*\*\*

Research Direction:      \*\*\*\*

Supervisor:      \*\*\*

Candidate:      \*\*\*

May, 2020

## 华东师范大学学位论文原创性声明

郑重声明：本人呈交的学位论文《面向工业界仿冒应用的大规模实证研究》，是在华东师范大学攻读硕士/博士（请勾选）学位期间，在导师的指导下进行的研究工作及取得的研究成果。除文中已经注明引用的内容外，本论文不包含其他个人已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中作了明确说明并表示谢意。

作者签名:\_\_\_\_\_

日期: 年 月 日

## 华东师范大学学位论文著作权使用声明

《面向工业界仿冒应用的大规模实证研究》系本人在华东师范大学攻读学位期间在导师指导下完成的硕士/博士（请勾选）学位论文，著作权归本人所有。本人同意华东师范大学根据相关规定保留和使用此学位论文，并向主管部门和学校指定的相关机构送交学位论文的印刷版和电子版；允许学位论文进入华东师范大学图书馆及数据库被查阅、借阅；同意学校将学位论文加入全国博士、硕士学位论文共建单位数据库进行检索，将学位论文的标题和摘要汇编出版，采用影印、缩印或者其它方式合理复制学位论文。

本学位论文属于（请勾选）

- ( ) 1. 经华东师范大学相关部门审查核定的“内部”或“涉密”学位论文\*，于年 月 日解密，解密后适用上述授权。
- ( ) 2. 不保密，适用上述授权。

导师签名:\_\_\_\_\_

本人签名:\_\_\_\_\_

年 月 日

\* “涉密”学位论文应是已经华东师范大学学位评定委员会办公室或保密委员会审定过的学位论文（需附获批的《华东师范大学研究生申请学位论文“涉密”审批表》方为有效），未经上述部门审定的学位论文均为公开学位论文。此声明栏不填写的，默认为公开学位论文，均适用上述授权）。

## \*\*\* 硕士学位论文答辩委员会成员名单

姓名	职称	单位	备注
***	***	*****	主席
***	***	*****	
***	***	*****	

# 摘要

伴随着移动应用迅猛的发展，研究人员开始关注如何分析移动应用的业务逻辑，了解应用的运行行为。不同于传统应用程序，Android 应用程序采用的是基于事件驱动的系统架构和面向组件的编程模式，业务逻辑对回调函数和多线程交互的依赖性高。上述特性使得程序的业务逻辑分散在不同的代码片段（例如方法、线程、组件等）中，阻碍了应用程序执行过程的建模，对了解 Android 应用程序执行细节造成了挑战。

为此，本文提出并实现了一个可用于生成 Android 应用程序动态函数调用图的系统——RunDroid。RunDroid 使用源程序代码插桩获取用户方法的执行信息，通过运行时方法拦截获取系统方法的执行信息，并将信息以日志的形式保留下来。根据应用程序运行期间产生的日志信息，RunDroid 能还原出应用的动态函数调用图。RunDroid 构建的函数调用图不仅可反映函数间的调用关系，还能反映方法对象关系和方法间的触发关系，体现 Android 中 Activity 组件的生命周期，为程序分析工作提供必要的运行时信息。

在本文中，我们从开源社区 F-Droid 中下载 9 个开源应用程序，利用 RunDroid 产生动态函数调用图，统计调用图中的函数关系数量。实验结果显示函数触发关系在应用程序的执行过程中普遍存在，并且在业务逻辑中承担着主要的作用。同时，我们还将 RunDroid 产生的动态函数调用图和 FlowDroid 产生的静态函数调用图进行对比分析。相比 FlowDroid，RunDroid 产生的函数调用图能够准确地反映应用程序的执行过程，表现函数间的调用关系和触发关系，真实地还原 Android Activity 组件的生命周期变迁。最后，我们还将 RunDroid 和错误定位相结合，进行了探索性的实验。实验结果显示，RunDroid 提供的函数调用图全面地反映了函数执行的因果关系。相比已有技术方案，RunDroid 能够使得方法间的因果关系模型更完整，体现更多的程序依赖信息，有助于提升错误定位的准确度。

**关键词:** Android，函数调用图，动态分析技术，生命周期，多线程

# ABSTRACT

With the rapid development of mobile apps, researchers have focused their attention on mobile apps analysis. They want to know the runtime behavior for the application via technical approaches. Unlike traditional programs, Android apps, based on event-driven architecture, take the components like Activity as essential building blocks and rely on callback functions and multithreaded communication heavily. These Android features split the business logic into different segments, i.e., methods, threads, and components, making trouble to the modeling for application execution , and posing the challenge to understand the execution for Android apps.

This paper proposed RunDroid, a system which can be used to build the dynamic function call graphs for Android apps. RunDroid obtains the execution information of user-level method via the source-code level instrumentation, gains the information of system-level method by runtime intercept. Using the log information, generated while app running, RunDroid can recover the dynamic function call graph for the application. The call graph, constructed by RunDroid, not only shows the calling relationship, but also reflects the method-object relationship, the trigger relationship between the methods, and the lifecycle of Activity component, and provides the important runtime details for program analysis.

In this paper, we download 9 Android apps from the open source community F-Droid and use RunDroid to build dynamic function call graphs. Results show that the trigger relationship is ubiquitous during app running and plays the major role in the Android. Also, we compare the call graphs generated by RunDroid and FlowDroid. Compared with the

FlowDroid's static call graph, the dynamic call graph generated by RunDroid can accurately display the execution detail for the apps, express the calling relationship and trigger relationship between functions, and restore the lifecycle of the Android component Activity. Finally, we apply RunDroid to fault localization for exploratory experiments. The result shows that the call graph provided by RunDroid can reflect the causal relationship of function execution comprehensively. Compared with the previous technical, RunDroid makes the causal relationship model between methods more complete, and computes more program dependency information, which helps to improve the accuracy of fault location result.

**Keywords:** *Android, Call Graph, Dynamic Analysis, Lifecycle, Multi-Thread*

# 目录

<b>摘要</b> . . . . .	i
<b>ABSTRACT</b> . . . . .	ii
<b>第一章 绪论</b> . . . . .	1
1.1 研究背景 . . . . .	1
1.2 国内外研究现状 . . . . .	2
1.2.1 Android 分析技术 . . . . .	2
1.2.2 静态分析技术 . . . . .	3
1.2.3 动态分析技术 . . . . .	3
1.2.4 分析技术的应用 . . . . .	4
1.3 论文研究意义 . . . . .	6
1.4 论文研究内容 . . . . .	6
1.5 本文遇到的困难与挑战 . . . . .	7
1.6 本文组织结构 . . . . .	8
<b>第二章 Android 系统介绍</b> . . . . .	9
2.1 Android 系统架构 . . . . .	9
2.2 Android 中的 Activity 组件 . . . . .	11
2.3 Android 中的多线程交互 . . . . .	13
2.3.1 基于 Java 的多线程交互 . . . . .	13
2.3.2 基于 Handler 的多线程消息调度 . . . . .	14

2.4	本章总结 . . . . .	16
<b>第三章</b>	<b>相关概念介绍 . . . . .</b>	<b>17</b>
3.1	概念说明 . . . . .	17
3.1.1	关于方法和对象的定义 . . . . .	17
3.1.2	关于方法间关系的定义 . . . . .	18
3.1.3	关于调用图的定义 . . . . .	19
3.2	Android 系统中的触发关系 . . . . .	20
3.2.1	基于事件响应的触发关系 . . . . .	20
3.2.2	基于 Java 多线程交互的触发关系 . . . . .	21
3.2.3	基于 Handler 多线程消息调度的触发关系 . . . . .	21
3.3	举例说明 . . . . .	22
3.4	本章总结 . . . . .	22
<b>第四章</b>	<b>RunDroid 的系统设计 . . . . .</b>	<b>23</b>
4.1	整体设计 . . . . .	23
4.2	方法信息的捕获 . . . . .	24
4.2.1	用户方法执行信息的获取 . . . . .	24
4.2.2	系统方法执行信息的获取 . . . . .	25
4.3	拓展函数调用图的构建过程 . . . . .	26
4.3.1	构建函数调用图 . . . . .	26
4.3.2	构建 Activity 的生命周期及事件回调的触发关系 . . . . .	28
4.3.3	构建多线程触发关系 . . . . .	29

4.4	本章总结 . . . . .	32
第五章	RunDroid 的系统实现 . . . . .	33
5.1	模块实现 . . . . .	33
5.1.1	预处理器 . . . . .	33
5.1.2	运行时拦截器 . . . . .	34
5.1.3	日志记录器 . . . . .	35
5.1.4	调用图构建器 . . . . .	35
5.2	本章总结 . . . . .	36
第六章	RunDroid 的系统实验 . . . . .	38
6.1	实验验证——函数触发关系的普遍性 . . . . .	38
6.2	应用结果展示 . . . . .	39
6.2.1	函数调用图的构建结果展示 . . . . .	40
6.2.2	Activity 生命周期和事件回调的效果展示 . . . . .	41
6.2.3	多线程触发关系效果展示 . . . . .	43
6.3	RunDroid 在错误定位领域的应用 . . . . .	44
6.3.1	原理简介 . . . . .	45
6.3.2	结果分析 . . . . .	46
6.4	本章总结 . . . . .	47
第七章	总结与展望 . . . . .	54
7.1	总结 . . . . .	54
7.2	展望 . . . . .	55
参考文献 . . . . .		56
致谢 . . . . .		60
攻读学位期间发表的学术论文 . . . . .		61

# 插 图

图 1.1 Google Play Store 上架的应用总数的变化趋势 . . . . .	1
图 2.1 Android 系统框架图 . . . . .	10
图 2.2 Activity 的生命周期 . . . . .	12
图 2.3 Handler 的使用实例 . . . . .	14
图 2.4 Handler 各 API 之间的调用关系 . . . . .	15
图 2.5 Handler 机制 . . . . .	16
图 4.1 RunDroid 的处理流程 . . . . .	23
图 5.1 RunDroid 的整体架构图 . . . . .	33
图 5.2 使用 Cypher 查找共用对象 $o_m$ 的方法 $m_{enqueue}$ 、 $m_{dispatch}$ . . . . .	36
图 6.1 MainActivitiy 的代码 . . . . .	49
图 6.2 斐波拉契数列相关的 RunDroid 调用图（局部） . . . . .	50
图 6.3 斐波拉契数列相关的 FlowDroid 调用图（局部） . . . . .	50
图 6.4 Activity 生命周期和事件回调的 RunDroid 调用图（局部） . . . . .	51
图 6.5 FlowDroid 生成的函数 dummyMainMethod() 的调用图 . . . . .	51
图 6.6 多线程触发关系相关的 RunDroid 调用图（局部） . . . . .	52
图 6.7 多线程触发关系-FlowDroid 生成的调用图（局部） . . . . .	52
图 6.8 使用 RunDroid 前后关于语句 Line 13 的测试用例匹配的对比 . . . . .	53

## 表 格

表 4.1 待拦截的系统方法列表 . . . . .	27
表 6.1 各应用运行过程的统计数据 . . . . .	39
表 6.2 结果对比 . . . . .	46

# 第一章 绪论

## 1.1 研究背景

根据著名网站 Statista 的统计<sup>[1]</sup> 显示，Android 官方应用平台 Google Play Store 在 2009 年 12 月至 2018 年 6 月期间的应用数量变化如图 1.1 所示。截止 2018 年 3 月，在 Google Play Store 上架的应用已经超过 330 万，而这个数字在 2013 年 7 月才刚刚突破 100 万。这反映出移动应用迅猛的增长趋势。

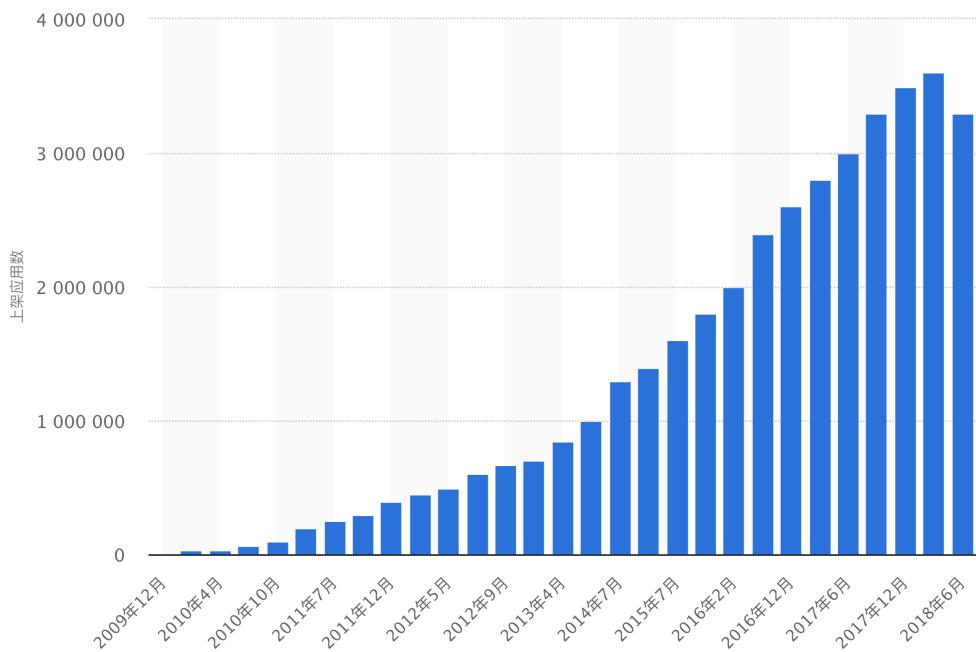


图 1.1: Google Play Store 上架的应用总数的变化趋势

在移动应用迅猛增长的同时，研究人员开始研究如何通过技术手段分析移动应用的代码内容，了解应用本身的运行时行为，进行相关学术研究和工业生产。利用程序分析技术，研究人员对应用程序的项目相关源代码、配置文件或者二进制分发文件进行分析，监控程序的运行时行为，总结出应用程序相关特征。结合具体

应用场景，我们对以上特征进行归纳总结并找出相关规律，应用在应用分析、安全风险以及质量保障等领域，进一步提升应用程序的易用性、安全性和可靠性。

## 1.2 国内外研究现状

### 1.2.1 Android 分析技术

在软件分析领域，根据分析过程中是否需要运行目标程序，我们可以将软件分析技术分为静态分析技术和动态分析技术。如果分析过程不依赖于目标程序的运行，这种分析技术称为静态分析技术，反之则为动态分析技术。

静态分析技术通常以二进制程序文件或者程序源代码作为研究主体，结合控制流分析、数据流分析技术、指针分析以及程序依赖分析，得出应用程序相关的函数调用图、UML 类图和序列图等，并在此基础上进行学术研究。

动态分析技术依赖目标程序的运行，通过修改目标程序的运行文件，搭建目标程序的运行环境，记录程序运行过程中相关操作信息，监控目标程序在运行过程中的状态变迁或指标变化，进而得出程序在运行过程中的行为特征，帮助研究人员进行程序安全性分析，提升程序的质量可靠性。

上述两种技术各有优劣。静态分析技术有着较为扎实的理论基础，覆盖较多的场景，分析较为全面，分析结果通常情况下精确可靠。但是，静态分析技术在枚举所有情况时，往往会遇到状态爆炸的问题，具体实验效果受到实验运行环境的硬件条件和算法实现程度的限制。而且，当静态分析技术分析的问题依赖于外部环境（例如，用户实时操作序列、手机所处环境因素等）时，分析得到的结果并不是非常准确。动态分析技术却能解决上述问题，通过对程序运行状态的监控，帮助研究人员了解程序的运行行为，掌握程序的安全性信息和可靠性信息。另外，动态分析技术还能解决动态代码加载的问题。但动态分析技术的短板也非常明显。动态运行环境的搭建需要一定的技术要求，构建系统的时间成本大。另外，动态分析技术的分析结果往往只适用于程序某次运行过程，无法直接推广到其它运行情况。

### 1.2.2 静态分析技术

在不执行应用程序的情况下，静态分析技术通过对应用程序的源代码或者执行文件进行控制流分析和数据流分析，进而推断应用程序在运行过程中可能产生的行为。这方面相关工作包括 [ 2–10]。Soot<sup>[2]</sup> 是传统的静态分析工具，其思路是将所有的 Java 字节码文件转化成一种中间语言 Jimple，并在 Jimple 的基础上进行常规的控制流分析、数据流分析，理论上适用于所有可以在 Java 虚拟机上运行的语言（例如 Scala、Groovy 等等）的分析。由于 Android 程序本身的字节码 Davlik 和 Java 字节码在格式上保持一致，因此，Soot 也支持 Android 应用程序的静态分析。但是，Soot 在分析过程中没有考虑一些 Android 的特性难免会出现一些问题。为此，德国达姆施塔特工业大学的 Steven Arzt 等人在 Soot 的基础上考虑 Android 程序中 Activity 的生命周期特性，推出了一个针对 Android 的静态分析工具 FlowDroid<sup>[3]</sup>，可以做到上下文、路径、对象、字段等层面上的敏感。FlowDroid 通过定义数据源点和数据泄漏点，在 Android 应用生命周期的基础上，可以实现数据流敏感的污点分析。但其不足之处在于缺少跨组件通信的分析不考虑多线程调用问题。在 FlowDroid 基础上，卢森堡大学的 Li Li 等人推出了 IccTA<sup>[5]</sup>，利用跨组件通信分析工具 IC3 提取跨组件通信的方法调用，并结合 AndroidManifest.xml 文件定义的 Intent Filter 信息，连接跨组件通信的两端组件，避免了 FlowDroid 因缺少跨组件通信而导致的数据流信息的缺失。

### 1.2.3 动态分析技术

和静态分析技术相对应，动态分析技术通过执行应用程序，获取程序运行过程的相关信息，从而实现对应的研究目的。动态分析技术往往需要对运行环境做适当的修改或者调用特殊的系统接口，记录应用程序运行过程的关键信息，结合数据流追踪等技术，实现对应用程序的运行时行为的监控与记录。这方面的工作代表包括 [ 11–15] 等。Enck 等人提出的 TaintDroid<sup>[11]</sup>，是一个高效的系统级的动态污点跟踪

和分析系统。它通过修改 Dalvik 虚拟机，利用动态污点分析技术实时监控敏感数据的生成、传播和泄露，实现了变量层面、方法层面、文件层面的数据追踪。此外，TaintDroid 还支持跨进程通信（IPC）层面上的污点分析，因此可以精确分析出应用程序从消费者手机上获取和发布隐私信息的完整传播过程。TaintDroid 虽然提供了较为完备的数据流分析技术，但是不支持控制流追踪，也无法给出相关语句的执行路径。DroidBox<sup>[12]</sup> 在 TaintDroid 基础上，对 Android Framework 的部分 API 做了修改，可以记录一些开发人员感兴趣的 API（例如文件读写、网络请求、SMS 服务等）的调用，并提供分析结果的可视化。同时，DroidBox 还实现了应用程序的自动安装和执行，弥补了 TaintDroid 在软件测试自动化方面的不足。和 TaintDroid 不同，TraceDroid<sup>[13]</sup> 采用的是另一种思路，利用字节码插装技术 AspectJ，使得方法在执行时输出相应的日志信息。根据这些信息 TraceDroid 可以还原函数调用图，得到分析结果。由于 Aspect 在进行字节码编织时引入的新方法会导致方法数 65K 限制问题（即构建 APK 文件的过程中，方法总数超过 65536，进而使得 APK 文件无法成功构建<sup>[16]</sup>），因此该方案存在不稳定的情况。Inspeckage<sup>[15]</sup> 是一个基于 Xposed 框架开发的动态分析工具，通过拦截应用程序相关的 Android API，帮助用户实时了解应用程序的运行时行为（例如文件读写、加解密、网络服务等）。

#### 1.2.4 分析技术的应用

静态分析技术和动态分析技术可以分析软件行为表现，常运用在安全性分析、质量保障等领域。

安全性分析包括隐私泄露、权限机制研究、恶意软件排查等。为了弥补 Android 官方文档权限说明不完整的状况，多伦多大学的 Kathy Au 等人提出的 PScout<sup>[17]</sup> 利用 Soot 对 Android 系统程序进行静态分析，构建出 Android 系统的函数调用图，在调用图中标识权限相关的 Binder 跨进程调用，结合逆向可达性分析技术得出系统 API 接口与权限的映射关系。在 PScout 的基础上，WHYPER<sup>[18]</sup> 将自然语言处理技

术和传统静态分析技术相结合，可以帮人们找出应用市场中那些应用描述和实际使用到的权限不符的应用。Wei Yang 等人发现一部分恶意应用试图通过模仿正常应用的行为模式，以躲避安全系统的检测。为此，Wei Yang 等人提出了一种基于静态分析技术的解决方案 AppContext<sup>[19]</sup>。在 FlowDroid 提供的函数调用图的基础上，AppContext 结合 Android 常见的系统事件形成 ICFG，提取出安全敏感行为的上下文，并利用机器学习算法通过明敏感行为及其上下文将应用行为分类为良性行为以及恶意行为。通过实验分析发现，AppContext 的准确率和召回率分别达到了 87.7% 和 95%，可验证安全敏感行为的恶意性与该行为的意图（通过上下文反映）的密切关系。AppIntent<sup>[20]</sup> 利用静态分析技术提取应用程序的时间约束条件，再结合符号执行技术，最后得到用户信息泄露的序列。类似的，AppAudit<sup>[21]</sup> 采用了静动态分析结合的技术方案，利用静态分析技术找出函数调用图中潜在的数据泄露路径，并利用动态污点分析技术进行进一步确认，在一定程度上减低了结果的假阳性。相比 AppIntent，AppAudit 得到的数据泄露路径，准确度更高，同时时间开销较小。Marvin<sup>[22]</sup> 从静动态分析技术的分析结果中提取应用的特征（例如应用元数据、类结果、应用运行时行为等），并利用机器学习技术进行训练。在对大规模数据验证时，Marvin 可以识别出 98.24% 的恶意软件，假阳性低于 0.04%。由于网络通信在 Android 应用中的普遍性，文献 [ 23–25] 专门研究 Android 应用上的网络安全问题，涵盖 UNIX Domain Socket、Internet Socket 等多个不同的方面，探讨了不同使用场景下的网络安全问题。

分析技术在质量保障领域的应用包括提升软件测试的代码覆盖率<sup>[26–29]</sup>、错误定位<sup>[30,31]</sup>、错误修复分析<sup>[32,33]</sup> 以及变异测试<sup>[34–36]</sup> 等。文献 [ 26] 利用静态分析工具 SCanDroid 得到静态 Activity 迁移图，并结合基于目标探索策略和深度优先探索策略进而达到了较好的 Activity 覆盖率。Wei Yang 等人的工作<sup>[27]</sup> 通过对 APP 的状态进行建模，在深度优先算法的基础上，使得程序在当前状态无可再遍历状态的情

况下进行回溯。相比传统的深度优先算法，Wei Yang 的工作对应的时间开销更小，效率更高。此外，Stoat<sup>[28,29]</sup> 利用静态、动态分析技术标识出程序本身的状态和事件，从应用中抽象出有限状态机（FSM）模型。基于该模型，Stoat 进行模型变异、测试用例生成与执行、模型迭代，使得应用程序的覆盖率相比之前工作 [ 37,38] 提升了 17~31%。文献 [ 30,31] 等将基于频谱的错误定位技术运用在 Android 应用上，取得了不错的成绩。文献 [ 32,33] 等主要聚焦如何在 Android 应用程序出现问题时，根据异常的堆栈信息结合软件变异技术或者问答网站生成补丁代码，实现程序的自动修复。范玲玲等人的工作<sup>[39]</sup> 通过开源社区中应用的 issue 分析，从异常种类（应用异常、系统异常、库（Library）异常）、系统异常分类、错误检查工具以及错误修复方式等若干方面进行较为深入的分析、探讨和总结。

### 1.3 论文研究意义

从上述工作中，我们不难发现，分析技术的应用往往需要构建 Android 应用程序的函数调用图，对程序执行过程进行建模。但是，Android 应用程序的特性（例如，基于事件驱动的架构、面向组件的开发方式、对回调函数和多线程交互的依赖等）会导致函数调用关系缺失或者不准确，丢失函数关联的必要信息，使得分析工具无法对程序执行过程进行准确建模，对程序分析工作造成挑战。

为此，本文提出了一种可用于还原 Android 动态函数调用图的技术方案——RunDroid。RunDroid 产出的函数调用图不仅包括方法调用关系，还展示了方法对象、方法间触发关系等信息，可以帮助研究人员全面的了解函数间的关系，辅助程序分析工作的完成。

### 1.4 论文研究内容

本文的主要研究工作如下：

- 本文在传统的方法调用关系的基础上，提出了函数触发关系的定义，并定义了 Android 系统中常见的触发关系。

- 本文提出了一个可用于生成 Android 动态函数调用图的系统——RunDroid。RunDroid 通过获取应用程序运行时的执行信息，还原出程序的动态函数调用图。RunDroid 生成的函数调用图可以体现函数间的调用关系和触发关系，反映 Android 平台相关的特性，为分析程序执行提供更为详细的信息。
- 本文利用 RunDroid 构建开源 Android 应用的动态函数调用图，研究方法间触发关系在程序运行过程中的普遍性。
- 本文对 RunDroid 的运行效果进行实验，将其产生的函数调用图与经典的静态动态分析工具 FlowDroid 产生的函数调用图进行对比分析，并从不同的角度比较了两者的异同，分析两种技术的优劣。
- 本文将 RunDroid 应用在错误定位领域，RunDroid 与错误定位技术结合的实验说明 RunDroid 可以提供更多程序依赖信息，有助于提高错误定位技术技术结果的准确性。

## 1.5 本文遇到的困难与挑战

本文解决的关键问题有如下几点：

- 1) 如何获取应用程序中各个函数的执行信息？

在应用程序执行过程中，获取各函数执行信息是本文的基础。函数按照函数的声明者可以分为用户定义的方法和系统预定义的方法。对于前者，我们可以通过修改程序源代码实现；但对于后者，由于我们无法直接修改系统程序的源代码，而构建一个符合本文业务需求的系统的成本较高，为此，我们需要寻找对应的解决方案以帮助我们获取系统方法的执行信息。

- 2) 如何根据程序中各函数的执行信息，还原应用程序的函数调用图？

为了还原应用程序的函数调用图，依靠以上执行信息还不够，需要组织各种执行信息，以挖掘执行信息见的关联关系。本文将对这一问题的解决作为一个关键研究点。

3) 如何在生成的函数调用图中体现 Android 特性?

正如上文所提的, Android 系统中有很多常规应用中不具备的特性, 例如 Activity 的生命周期、基于多线程调用的触发关系。若要在生成的函数调用图上体现上述特性, 需要对 Android 系统源代码有一定了解, 并基于图中的相关信息创建与特性相对应的关系。这既是本文研究的重点, 也是本文的创新点。

## 1.6 本文组织结构

本文共分为七章, 环绕着 Android 动态函数调用图构建系统的设计与实现展开, 各章节内容如下:

**第一章** 主要介绍了本文的研究背景、相关工作、研究意义、研究内容及本文遇到的困难与挑战。

**第二章** 从 Android 的体系结构出发, 介绍了 Android 系统相关的背景知识, 包括 Activity 组件和 Android 多线程交互方式等。

**第三章** 对本文使用到的基本概念做基本的介绍, 定义 Android 系统中常见的触发关系, 并结合具体示例简单解释这些概念。

**第四章** 对 RunDroid 系统的系统功能做了基本的介绍, 从整体上阐述了 RunDroid 的工作原理, 并详细地介绍了 RunDroid 获取运行时的方法信息, 并根据日志构建拓展函数调用图的完整过程。

**第五章** 对 RunDroid 运行流程做了基本的阐述, 并详细介绍了 RunDroid 中预处理器、运行时拦截器、日志记录器、调用图构建器等组件的技术实现。

**第六章** 将利用 RunDroid 构建开源应用的动态函数调用图; 将 RunDroid 产出的调用图与静态分析工具 FlowDroid 进行对比; 并将 RunDroid 与错误定位技术结合进行探索性的实验。

**第七章** 对本文工作进行总结, 并对下一步工作进行展望。

## 第二章 Android 系统介绍

本章首先简要介绍 Android 系统架构，然后对 Android 中的 Activity 组件做基本介绍，最后着重介绍 Android 中常见的两种多线程交互方式。

### 2.1 Android 系统架构

Android 是基于 Linux 内核开发的开源操作系统，隶属于 Google 公司，主要用于触屏移动设备如智能手机、平板电脑与其他便携式设备，是目前世界上最流行的移动终端操作系统之一。

在系统架构上，Android 自下到上，可以分为四层：Kernel 层、Library 和 Android Runtime(Dalvik/ART)、Framework、Application，如图 2.1 所示。Kernel 层是硬件和软件层之间的抽象层，主要是设备的驱动程序，例如：显示驱动、音频驱动、蓝牙驱动、Binder IPC 驱动等。Library 和 Android Runtime(Dalvik/ART)：Library，顾名思义就是向上层提供各种这样的基础功能，例如 SQLite 数据库引擎，Surface Manager 显示系统管理库。Android Runtime 主要是运行 Android 程序的虚拟机。Framework 层主要是系统管理类库，包括 Activity 的管理，消息通知的管理；同时，它是 Application 的基础架构，为应用程序层的开发者提供了 API，可简化应用程序的开发过程。而 Application 就是我们平时接触的应用，例如通讯录、电话、浏览器等。

虽然 Android 应用程序是使用 Java 语言开发的，但是它和传统的 Java 程序有着很大的不同，具体有如下几点：

**基于事件驱动的系统架构：**在设计上，Android 应用程序的系统架构为事件驱动架构。在开发过程中，没有传统程序中入口函数的概念。应用程序中通用的业务逻辑（例如应用程序如何启动退出、应用的窗口如何创建销毁等）存在于 Android Framework 中。

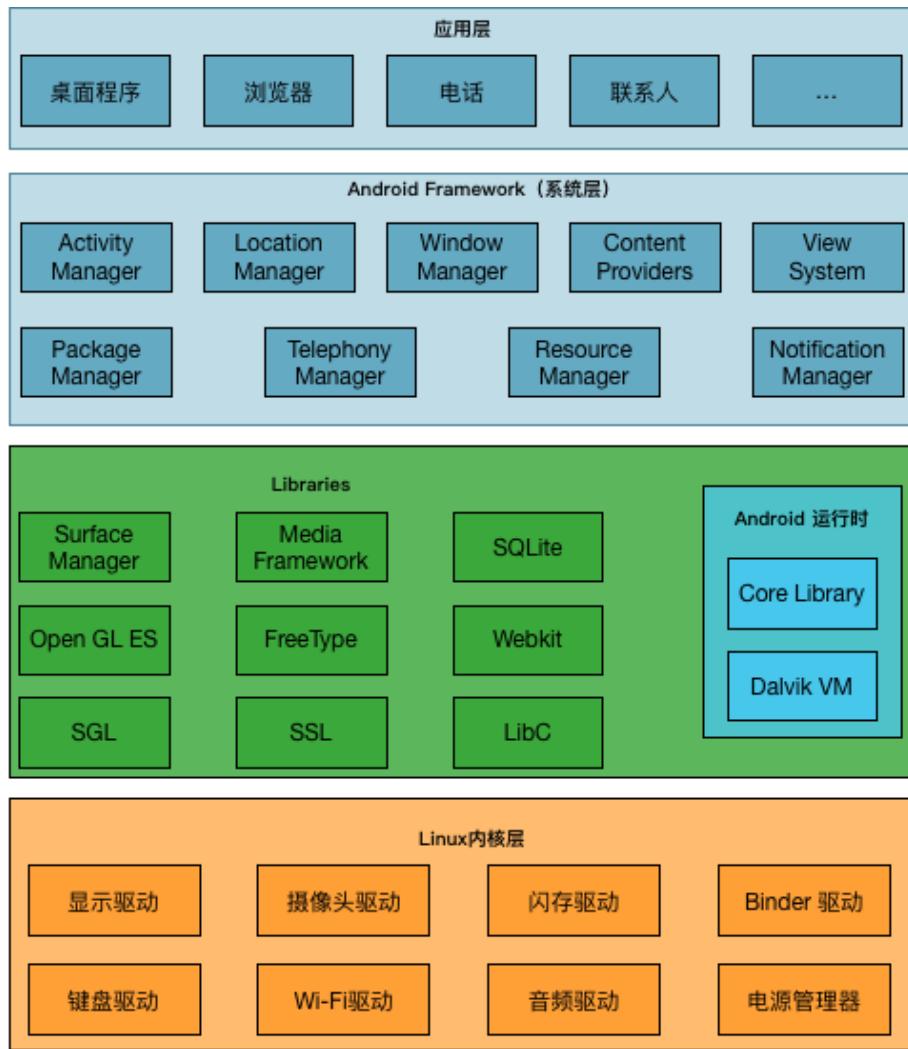


图 2.1: Android 系统框架图

**面向组件的编程模式:** Android 程序中, 常见的组件有 Activity、Service、Content Provider、Broadcast Receiver, 它们是应用程序运行的最小单元, 具有组件特有的生命周期, 生命周期受到 Android Framework 的直接调度。开发人员通过继承这些组件, 重写对应的生命周期函数, 实现对应的业务需求, 例如界面的布局、页面状态的保存。

**逻辑实现依赖于回调函数和多线程通信:** 由于 Android 应用程序采用的是基于单线程消息队列的事件驱动架构, 界面相关的操作只允许出现在主线程 (UI Thread) 中, 耗时操作只能在工作线程 (Worker Thread) 中进行。具体而言, 开发人员往往

会借助回调函数处理控件的响应事件，利用多线程交互串联界面相关操作和耗时操作，完成对应的业务。

## 2.2 Android 中的 Activity 组件

在 Android 应用程序运行过程中，Activity 向用户展示图形界面，响应用户的反馈，和其它组件一同完成相关业务，扮演着最为重要的角色。由于 Android 应用程序在架构选型上采用了事件驱动模型，为了便于协调应用内部状态的管理，Android 组件通常有生命周期的概念，Activity 也不例外。Android 系统将 Activity 的运行状态分为以下四种，Activity 的生命周期就是 Activity 在这些状态之间的跳转。

- **运行态：**在该状态下，Activity 处于页面最前端时，用户可以与 Activity 进行交互。一般而言，用户可见并可以进行交互的 Activity 均处于这个状态。
- **暂停态：**在该状态，Activity 仍然可见，但是失去了窗口的焦点。当一个 Activity 之上出现非全屏的窗体（例如对话框）但未被完全遮挡时，Activity 就处于这个状态。处于暂停状态的 Activity 仍处于存活状态，保存着所有的内存数据。
- **停止态：**当一个 Activity 被其他的 Activity 完全遮挡时，处于这个状态。处于该状态的 Activity 仍然可以保留所有的内存数据，只是对用户不可见。系统在需要内存的情况下，可以采用相应的策略对 Activity 进行杀死回收操作。
- **终止态：**当用户自动退出 Activity 时，Activity 将进入该状态；当 Activity 处于暂停态或者停止态时，系统由于内存原因可能会将上述两种 Activity 杀死并回收。

Activity 的生命周期受到 Activity 在运行时的内存分布、环境状态以及业务逻辑的影响，由 Android 系统直接负责调度。和 Activity 生命周期相关的方法包括 `onCreate()`、`onStart()`、`onResume()`、`onPaused()`、`onStop()` 和 `onDestroy()` 等，方便开发人员在 Activity 的状态发生变化时对程序的运行时数据和应用状态做适

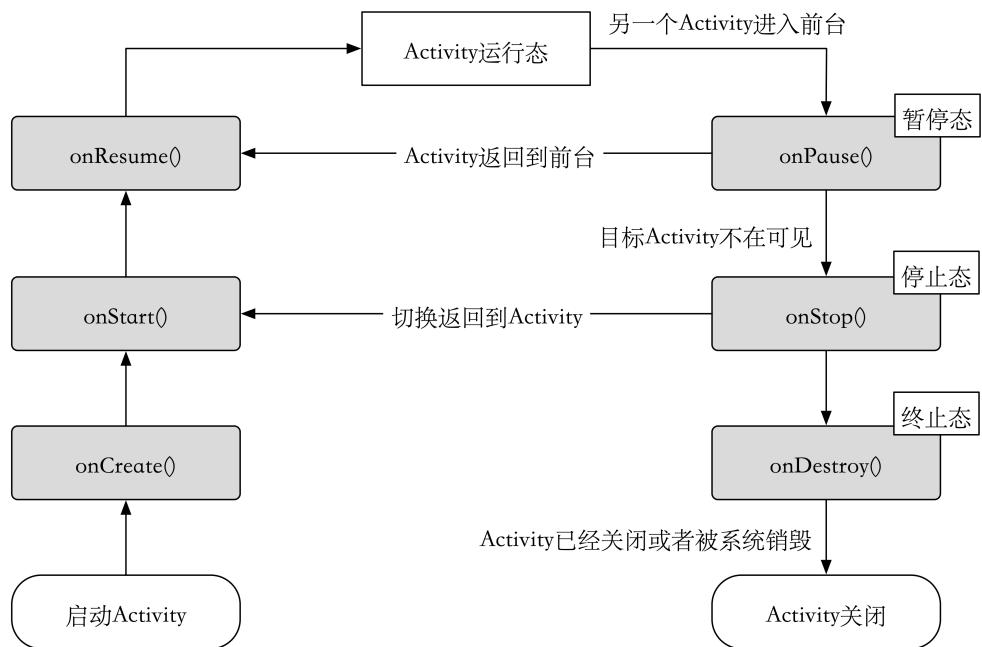


图 2.2: Activity 的生命周期

当的处理操作。Activity 的生命周期如图 2.2所示：

当用户点击应用图标，系统启动应用程序后，系统会创建、启动 Activity 并使之可以和用户进行交互。在这个过程中，`onCreate()`、`onStart()`、`onResume()` 等方法被回调，Activity 最终处于运行态；

当 Activity 被其它窗体遮挡，虽然失去交互焦点，但仍处于可见状态时，Activity 的状态从运行态变成暂停态，`onPause()` 方法会被回调；当 Activity 重新获得焦点时，`onResume()` 方法会被调用；

当 Activity 切换到另一个 Activity 时，原来 Activity 将从运行态变为停止态，`onPause()`、`onStop()` 等方法会被调用；当返回原来的 Activity 时，`onStart()`、`onResume()` 等方法会被调用，Activity 从暂停态变为运行态。

当用户点击“返回键”返回到桌面时，Activity 会失去焦点，在用户的视野中消失，直至被系统回收，对应的状态也从运行态经过暂停态、停止态，最终变为终止态，期间 `onPause()`、`onStop()`、`onDestroy()` 等方法被回调。

## 2.3 Android 中的多线程交互

Android 系统在架构设计上采用了事件驱动架构。在多线程并发访问时，若控件对于各线程均是可见的，并发对控件做读写操作会使控件处于不可预期的状态；若贸然对控件使用锁机制，访问控件的各个线程之间存在竞争关系，阻塞相关线程业务逻辑的执行，使得应用变得复杂低效。为了避免此类低效率问题，Android 系统在设计事件驱动架构时，采用了单线程的消息队列，即只允许在主线程（也称为 UI 线程）进行界面更新操作，不允许在其他线程（也称为工作线程）进行界面更新操作。另外，为了保证应用程序界面渲染和事件响应的及时性，任何在主线程上产生的耗时操作（例如加载磁盘上的图片、网络请求等）都是不被允许的。换而言之，当应用程序需要进行耗时操作时，这个操作往往会在一个新的线程中执行，而不是在主线程中。

Android 系统架构中的单线程消息队列以及主线程的非阻塞性，使得界面更新和耗时操作分散在不同的线程中。这也导致了多线程交互在 Android 应用程序运行过程中十分常见。从整体上，系统提供的交互方式分为两种：基于 Java 的多线程交互和基于 Handler 的交互方式。

### 2.3.1 基于 Java 的多线程交互

Android 应用程序是在 Java 的基础上开发的，因此，开发人员可以采用和 Java 应用相同的调用方式启动工作线程，并在对应的线程上完成业务逻辑。但是，Java API 只能实现业务逻辑从原有线程转移到新的工作线程上，不能重新返回到主线程上。为此，Android 系统在 Java API 的基础上还提供了 API `void runOnUiThread(Runnable runnable)`。该 API 可以帮助开发人员将业务逻辑的执行从工作线程转移到主线程上，该 API 也符合 Android 只允许在主线程上更新界面这一基本设计原则。但是，该 API 也存在着一些弊端，例如 `runOnUiThread(Runnable)` API 的定义位于类 `android.app.Activity`。这也就意味着，当业务逻辑的上下文脱离了 Activity 这

一组件，将无法通过该 API 将业务逻辑转移回主线程，同时基于接口的函数参数定义方式对于跨线程的参数传递也不是十分友好。为此，Android 提供了基于 Handler 的多线程交互方式。

### 2.3.2 基于 Handler 的多线程消息调度

为了满足开发人员多样化的业务在多线程间的切换，Android 提供了基于 Handler 消息调度的多线程交互方式。

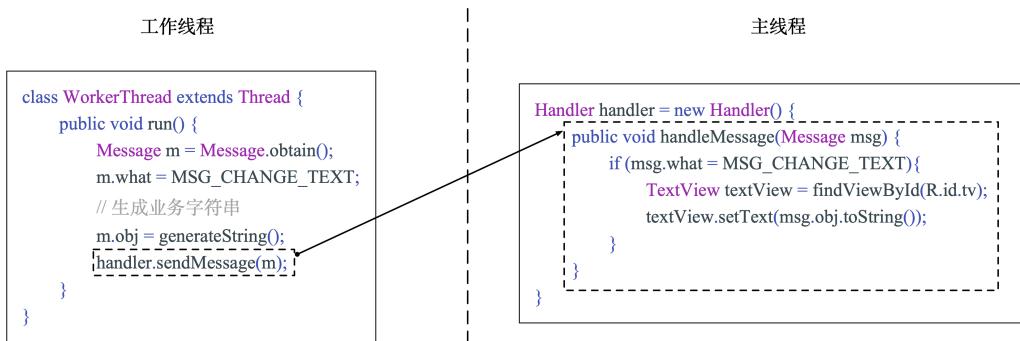


图 2.3: Handler 的使用实例

图 2.3 为关于 Handler 的使用示例。图中对应的场景为应用由于业务需要获取一个字符串，并将这个字符串展现在用户界面上。考虑到获取字符串的过程比较耗时，可能会阻塞主线程的相关业务，因此我们会在工作线程执行这部分逻辑（如图 2.3-左所示），而在主线程执行信息展现的逻辑（如图 2.3-右所示）。具体而言，在工作线程中，用户生成的字符串和对应的逻辑代码 `MSG_CHANGE_TEXT` 封装到 `Message` 对象中，通过 `Handler` 发送出去；`Handler` 在主线程中收到该 `Message` 对象时，会根据逻辑代码（即 `Message.what` 的值）决定进行对应的逻辑处理（本例中，对应逻辑为更新界面信息）。

除了示例中的调用方式，Android 系统还提供多种 API<sup>[40]</sup>（如图 2.4 所示），同时支持基于 `Runnable` 的消息调度和基于逻辑代码的消息调度：通过分析 Android 系统相关源代码，我们发现 Android 系统提供的 Handler API 间调用关系如图 2.4 所示。所有的 Handler API 最终都会调用到 Handler 类的私有方法 `enqueueMessage(MessageQueue,`

*Message, long)。*

从整体上看，Handler 机制主要由 Handler、Looper、MessageQueue、Message 等若干部分共同构成。Message 是多线程交互的核心载体；考虑到移动设备的硬件限制以及 Message 使用的频繁性，Android 系统通过对象池对 Message 对象进行管理。MessageQueue 是一个双端队列，存放所有待处理的 Message 对象。Looper 负责消息的分发。作为消息的发送者和消费者，Handler 负责将消息发送到对应的 MessageQueue 中以及消费来自 Looper 分发下来的 Message。Handler 机制中各组成部分的相互关系如图 2.5 所示。当用户要通过 Handler 传递消息时，用户将调用系统提供的 Handler API，该 API 会将通过分发方法 *Handler.enqueueMessage(MessageQueue, Message, long)* 将消息投递到消息队列 MessageQueue 中；当 Looper 从 MessageQueue 中读取该 Message 对象，分发给对应的 Handler 对象；最终，Handler 对象通过方法 *Handler.dispatchMessage(Message)* 调用方法 *Handler.handleMessage(Message)*，按照 Message 对象进行业务逻辑处理。

基于 Handler 的多线程消息调度机制充分利用了 Android 的事件驱动架构，将业务逻辑抽象出 Message 对象，借助消息队列在线程间传递 Message，达到了 Android 多线程交互的目的。Handler 机制，不仅帮助开发人员实现业务逻辑在主线程和工作线程间的自由转移，而且其灵活的 API 设计降低应用的设计复杂程度，提

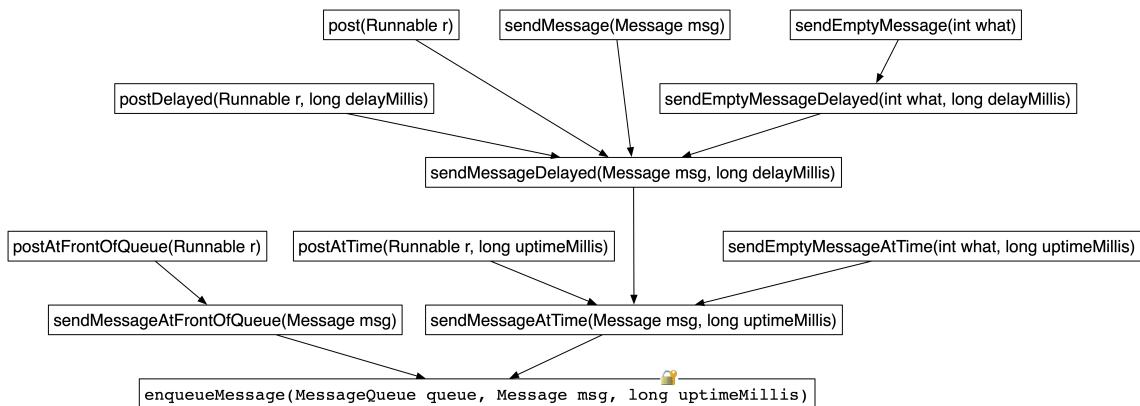


图 2.4: Handler 各 API 之间的调用关系

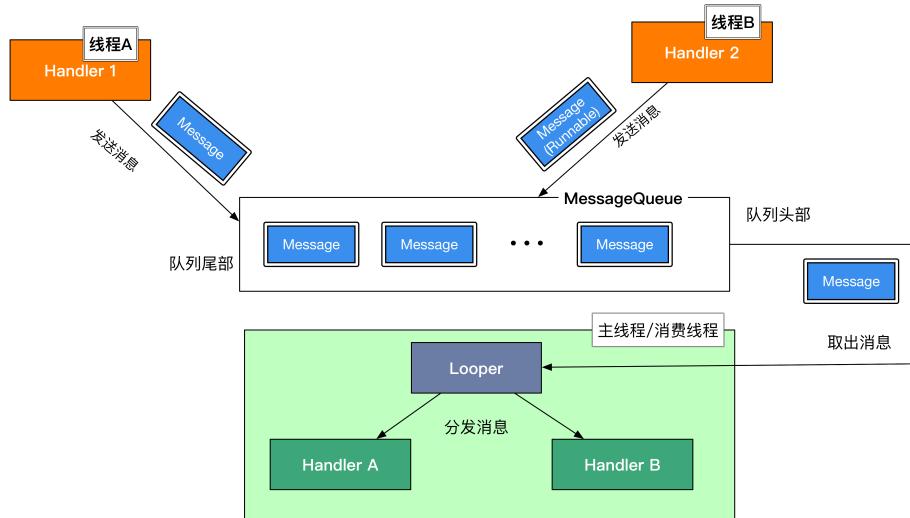


图 2.5: Handler 机制

升了系统架构的可拓展性。正因如此，在Android开发过程中，基于Handler消息调度的多线程交互十分常见。

## 2.4 本章总结

本章主要介绍了Android系统的相关背景知识，较为详细的阐述了Android的系统结构，详细介绍了Android四大组件之一的Activity生命周期。同时，本章还介绍了基于Runnable/Thread、Handler消息调度两种不同的多线程交互方式，详细地分析了Handler的运行机制，为下文基于函数调用图的多线程触发关系生成做了铺垫。

## 第三章 相关概念介绍

在本节，我们将给出拓展动态函数调用图构建过程中的基本术语，并基于这些概念定义 Android 系统中常见的触发关系，结合具体示例进行解释说明。

### 3.1 概念说明

方法执行是一个方法执行相关信息的描述，方法对象对应是和方法执行相关的对象<sup>1</sup>；调用关系和触发关系描述了方法执行之间的关系。函数调用图为所有调用关系的集合，在函数调用图上添加和方法相关的对象信息，补全方法间触发关系得到拓展函数调用图。

#### 3.1.1 关于方法和对象的定义

##### 定义 1 方法对象

和方法执行相关的对象称为方法对象，可以体现对象和执行方法的相互关系。在本文中，方法对象通常用符号  $o$  表示。

对于方法执行  $m$ ，对象和方法执行的关系（简称为方法对象关系）如下所示：

- 参数关系：若对象  $o_p$  是这个方法  $m$  的参数，记为  $m \xrightarrow{\text{parameter}} o_p$ ；
- 返回值关系：若对象  $o_r$  是这个方法  $m$  的返回值，记为  $m \xrightarrow{\text{return}} o_r$ ；
- 实例关系：若方法  $m$  是非静态方法，则方法执行时我们可以获取到关联到的 `this` 指针对象  $o_i$ ，记为  $m \xrightarrow{\text{instance}} o_i$ ；

通常情况下，一个方法执行过程中，返回值对象和实例对象不允许超过一个，而方法参数可以有多个<sup>2</sup>。一个对象可以同时是一个方法执行过程中的实例、参数或者返回值，具体情况由函数执行过程决定。

<sup>1</sup>在 Java 语言中，函数称为方法。在本文中，两者可以相互替换，不做区分。

<sup>2</sup>返回值为 `void` 的方法无返回值对象，静态方法无实例对象。

## 定义 2 方法执行

方法执行是对方法执行过程中的相关信息的描述，完整的信息包括对应方法的完整签名、执行时所处的线程以及相关的方法对象。在本文中，方法执行通常用符号  $m$  表示。

### 3.1.2 关于方法间关系的定义

#### 定义 3 调用关系

对于程序  $P$  的两个方法执行  $m_1$  和  $m_2$ ，方法  $m_1$  调用了方法  $m_2$ ，则记作  $m_1 \rightarrow m_2$ ，称为方法  $m_1$  调用方法  $m_2$ 。

$$m_0 \rightarrow m_1, m_1 \rightarrow m_2, \dots, m_n \rightarrow m (n \geq 0) \quad (3.1)$$

对于方法执行  $m$ ，若存在方法执行  $m_i$  ( $i = 0, \dots, n, n \geq 0$ )，使得式子 3.1 成立，则记作  $m_0 \rightarrow m_1 \rightarrow \dots \rightarrow m_n \rightarrow m$ ，简写为  $m_0 \xrightarrow{*} m$ ，称为方法  $m_0$  扩展调用方法  $m_n$ 。方法  $m_0$  扩展调用方法  $m$  的路径用  $[m_0, \dots, m_n, m]$  表示。特殊的，当  $n = 0$  时，对于方法  $m_0$  和方法  $m$ ， $m_0 \rightarrow m$  成立，则  $m_0 \xrightarrow{*} m$  也成立。

在系统源代码的层面上，对于方法  $m$  和  $m'$ ，若方法  $m$  的执行过程总是会调用方法  $m'$ （即  $m \rightarrow m'$  总是成立），可以记为  $m \Rightarrow m'$ ；类似的，若  $m \xrightarrow{*} m'$  总是成立，可以记为  $m \xrightarrow{*} m'$ 。

#### 定义 4 触发关系

若方法  $m_a$  和方法  $m_b$  之间同时需要满足以下三个条件，则两个方法存在触发关系，记为  $m_a \hookleftarrow m_b$  或者  $m_a \xrightarrow{\text{因果关系}} m_b$ ，称为  $m_a$  触发调用  $m_b$ ：

- 方法执行  $m_a$  的执行时间总是在方法执行  $m_b$  的执行时间之前；
- 方法  $m_a$  和  $m_b$  之间不存在一条调用路径，即  $m_a \xrightarrow{*} m_b$  不成立；
- $m_a$ 、 $m_b$  之间存在着因果关系，包括但不限于事件回调或多线程交互等。

以多线程中的 Thread 为例, 方法  $Thread.start()$  (记为  $m_{start}$ ) 的执行会使 JVM/Dalvik 虚拟机创建一个新的线程。最终, 虚拟机会在新的线程中回调该 Thread 对象的方法  $Thread.run()$  (记为  $m_{run}$ )。上述描述可以表示成  $m_{start} \hookrightarrow m_{run}$ 。由于这个触发关系和多线程相关, 也可以记作  $m_{start} \xrightarrow{\text{Thread}} m_{run}$ 。同样的, 触发关系也适用于 UI 事件注册与响应、基于 Handler 的多线程交互。

在系统源代码的层面上, 对于方法  $m_a, m_b, m_c$ , 若  $m_a \xrightarrow{*} m_b$ 、 $m_b \hookrightarrow m_c$  同时成立, 则  $m_a \hookrightarrow m_c$  成立; 若  $m_a \hookrightarrow m_b$ 、 $m_b \xrightarrow{*} m_c$  同时成立, 则  $m_a \hookrightarrow m_c$  成立。

### 3.1.3 关于调用图的定义

#### 定义 5 函数调用图

函数调用图是对程序运行时行为的描述, 用有向图  $CG = (V, E)$  表示。图中的点和方法执行  $m$  一一对应; 如果方法  $m_1$  调用方法  $m_2$  (即  $m_1 \rightarrow m_2$ ), 则有向边  $e = \langle m_1 \rightarrow m_2 \rangle$  属于有向边集合  $E$ ; 如果方法  $m_1$  触发方法  $m_2$  (即  $m_1 \hookrightarrow m_2$ ), 则有向边  $e' = \langle m_1 \hookrightarrow m_2 \rangle$  属于有向边集合  $E$ 。

**注意:** 在应用执行过程中, 方法 A 被调用了两次, 方法 A 的每次执行都调用了方法 B, 则对应的函数调用图  $CG$  如式子 3.2 所示。在调用图  $CG$  中,  $m_a$  和  $m_b$  各有两个, 分别对应的两次方法执行。 $\langle m_{a_1} \rightarrow m_{b_1} \rangle$  对应的是第一次函数 A 调用函数 B,  $\langle m_{a_2} \rightarrow m_{b_2} \rangle$  对应的是第二次函数 A 调用函数 B,

$$\left\{ \begin{array}{l} CG = (V, E), \\ V = \{m_{a_1}, m_{b_1}, m_{a_2}, m_{b_2}\}, \\ E = \{\langle m_{a_1} \rightarrow m_{b_1} \rangle, \langle m_{a_2} \rightarrow m_{b_2} \rangle\}. \end{array} \right. \quad (3.2)$$

#### 定义 6 拓展函数调用图

在函数调用图的基础上, 添加了方法对象和函数间的触发关系。拓展函数调用图中的节点包括方法执行节点和方法对象节点。图中的边包括描述方法间关系的边和描述方法和对象间的边: 前者的方法间关系包括调用关系和触发关系; 而后者

的关系包括和方法对象相关的三个关系（即参数关系、返回值关系与实例关系）。

### 3.2 Android 系统中的触发关系

触发关系描述的是方法执行间的关系。触发定义通常由对象与方法的属性约束条件和对象与方法的关系约束条件两个部分组成。前者描述的是对象和方法执行的属性约束条件，即对对象  $o$  本身的类型  $o.class$ 、定义方法  $m$  的类  $m.class$ 、方法执行  $m$  的方法签名  $m.sign$  和方法执行  $m$  的方法完整签名  $m.methodSign$ （即  $m.class$  和  $m.sign$  的组合）等属性的约束；后者对应的则是对象和方法之间的关系，包括方法对象关系、方法间的调用关系。本文中涉及的触发关系包括以下几种。

#### 3.2.1 基于事件响应的触发关系

Andoid 事件回调的触发关系描述的是控件点击事件的注册和响应之间的因果关系，即方法  $View.setOnClickLister(View$OnClickListener)$ （用  $m_{register}$  表示）和  $View$OnClickListener.onClick(View)$  间的因果关系（用  $m_{click}$  表示）。在上述方法中，方法  $m_{register}$  的实例对象  $o_{view}$  是方法  $m_{click}$  的参数对象，而方法  $m_{register}$  的参数对象  $o_{listener}$  是方法  $m_{click}$  的实例对象。因此，当两个方法  $m_{register}$ 、 $m_{register}$  及其方法对象  $o_{view}$ 、 $m_{register}$  满足式子 3.3 时， $m_{register} \xrightarrow{UiEvent} m_{register}$  成立。

$$\left\{ \begin{array}{l} o_{view}.class = View; \\ o_{listener}.class \in \{c \mid c \text{ 为类 } View$OnClickListener \text{ 的子类}\}; \\ m_{register}.methodSign = View.setOnClickLister(View$OnClickListener); \\ m_{click}.class \in \{c \mid c \text{ 为类 } View$OnClickListener \text{ 的子类}\}; \\ m_{click}.sign = onClick(View); \\ m_{register} \xrightarrow{\text{instance}} o_{view}, m_{register} \xrightarrow{\text{parameter}} o_{listener}; \\ m_{click} \xrightarrow{\text{parameter}} o_{view}, m_{click} \xrightarrow{\text{instance}} o_{listener}. \end{array} \right. \quad (3.3)$$

### 3.2.2 基于 Java 多线程交互的触发关系

基于 Java 的多线程交互往往是以 `Runnable` 作为传递对象，通常通过调用方法 `Thread.start()`（用  $m_{start}$  表示）和 `Activity.runOnUiThread(Runnable)`（用  $m_{runOnUiThread}$  表示）等 API，进而触发类 `Thread` 和 `Runnable` 的方法 `run()`（用  $m_{run}$  表示）的执行。

通过方法  $m_{start}$  触发方法  $m_{run}$  的执行时，方法  $m_{start}$  和  $m_{run}$  的实例对象均为同一个 `Thread` 对象  $o_{thread}$ 。因此，当两个方法  $m_{start}$ 、 $m_{run}$  及 `Thread` 对象  $o_{thread}$  满足式子 3.4 时， $m_{start} \xrightarrow{\text{Thread}} m_{run}$  成立。

$$\left\{ \begin{array}{l} m_{start}.methodSign = \textbf{Thread.start}(); \\ m_{run}.class \in \{c \mid c \text{ 为类 } \textbf{Thread} \text{ 的子类}\}; \\ m_{run}.sign = \textbf{run}(); \\ o_{thread} \in \{c \mid c \text{ 为类 } \textbf{Thread} \text{ 的子类}\}; \\ m_{start} \xrightarrow{\text{instance}} o_{thread}, m_{run} \xrightarrow{\text{instance}} o_{thread}. \end{array} \right. \quad (3.4)$$

同样的，对于方法 `Activity.runOnUiThread(Runnable)`，也存在类似的关系：在该触发关系中，`Runnable` 类型的对象  $o_r$ ，既是方法  $m_{runOnUiThread}$  的参数，又是方法  $m_{run}$  的实例。因此，当两个方法  $m_{runOnUiThread}$ 、 $m_{run}$  及其方法对象  $o_r$  满足式子 3.5 时， $m_{runOnUiThread} \xrightarrow{\text{runOnUiThread}} m_{run}$  成立。

$$\left\{ \begin{array}{l} m_{runOnUiThread}.methodSign = \textbf{Activity.runOnUiThread(Runnable)}; \\ m_{run}.class \in \{c \mid c \text{ 为接口 } \textbf{Runnable} \text{ 的实现类}\}; \\ m_{run}.sign = \textbf{run}(); \\ o_r.class \in \{c \mid c \text{ 为接口 } \textbf{Runnable} \text{ 的实现类}\}; \\ m_{runOnUiThread} \xrightarrow{\text{parameter}} o_r, m_{run} \xrightarrow{\text{instance}} o_r. \end{array} \right. \quad (3.5)$$

### 3.2.3 基于 Handler 多线程消息调度的触发关系

根据第二章中关于 `Handler` 的介绍，我们知道 `Handler` 通过 `Message` 进行多线程消息调度：通过方法 `enqueueMessage(Message)`（用  $m_{enqueue}$  表示）向消息队列从放入消息（用  $o_m$  表示）；通过方法 `dispatchMessage(Message)`（用  $m_{dispatch}$  表示）

从消息队列中取出消息  $o_m$ , 进行对应业务逻辑处理。因此, 当两个方法  $m_{enqueue}$ 、 $m_{dispatch}$  及其方法对象  $o_m$  满足式子 3.6 时,  $m_{enqueue} \xrightarrow{\text{Handler}} m_{dispatch}$  成立。

$$\left\{ \begin{array}{l} m_{enqueue}.methodSign = \textbf{Handler.enqueueMessage(Message)}; \\ m_{dispatch}.methodSign = \textbf{Handler.dispatchMessage(Message)}; \\ o_m.class = \textbf{Message}; \\ m_{enqueue} \xrightarrow{\text{parameter}} o_m, m_{dispatch} \xrightarrow{\text{parameter}} o_m. \end{array} \right. \quad (3.6)$$

### 3.3 举例说明

以第二章中的图 2.3 为例, 我们将简要阐述上述概念。在线程 WorkerThread 中, 方法  $run()$  依次调用了方法  $Message.obtain()$  (用  $m_{obtain}$  表示) 和方法  $Handler.sendMessage(Message)$  (用  $m_{send}$  表示), 则有  $m_{run} \rightarrow m_{obtain}$  和  $m_{run} \rightarrow m_{send}$ 。对于方法  $m_{obtain}$ ,  $o_m \xrightarrow{return} m_{obtain}$  成立。对于方法  $m_{send}$ ,  $o_m \xrightarrow{\text{parameter}} m_{send}$ 、 $o_{handler} \xrightarrow{\text{instance}} m_{send}$  成立。通过对 Android Handler 运行机制的分析, 我们知道  $m_{enqueue} \leftrightarrow m_{dispatch}$ 。从图 2.4、图 2.5 中可知, 所有的 Handler API 最后就会调用到  $Handler.enqueueMessage(MessageQueue, Message, long)$  方法, 即  $m_{send} \xrightarrow{*} m_{enqueue}$ , 以及  $m_{dispatch} \xrightarrow{*} m_{handle}$ , 因此, 方法  $m_{send}$  触发调用了方法  $m_{handle}$ , 即  $m_{send} \rightarrow m_{handle}$  或  $m_{send} \xrightarrow{\text{Handler}} m_{handle}$ 。

### 3.4 本章总结

本章介绍了本文使用到的基本概念, 从方法和对象的关系、方法间关系、调用图等几个方面对方法关系、方法执行、调用关系、触发关系、函数调用图、拓展调用图等概念做了符号化的定义。在此基础上, 本章从对象与方法的属性约束条件、对象与方法的关系约束条件两个方面对 Android 系统中的触发关系做了详细的定义。同时, 本章还结合第二章的 Handler 例子简单阐述了上述概念。

## 第四章 RunDroid 的系统设计

本文提出了一个可用于还原 Android 应用程序运行时动态函数调用图的系统——RunDroid。RunDroid 利用源程序代码插桩和运行时方法拦截的相结合的方式，捕获应用程序在应用层面和系统层面的方法执行信息，还原方法间的调用关系。在此基础上，RunDroid 利用对象和方法的关系结合具体的触发规则，进而还原出方法间触发关系，在调用图中展现运行过程中的 Android 特性行为。RunDroid 提供的函数调用图从方法调用关系、方法间的触发关系以及方法执行的相关对象信息等多个方面较为全面地刻画了 Android 应用程序的执行过程。

### 4.1 整体设计

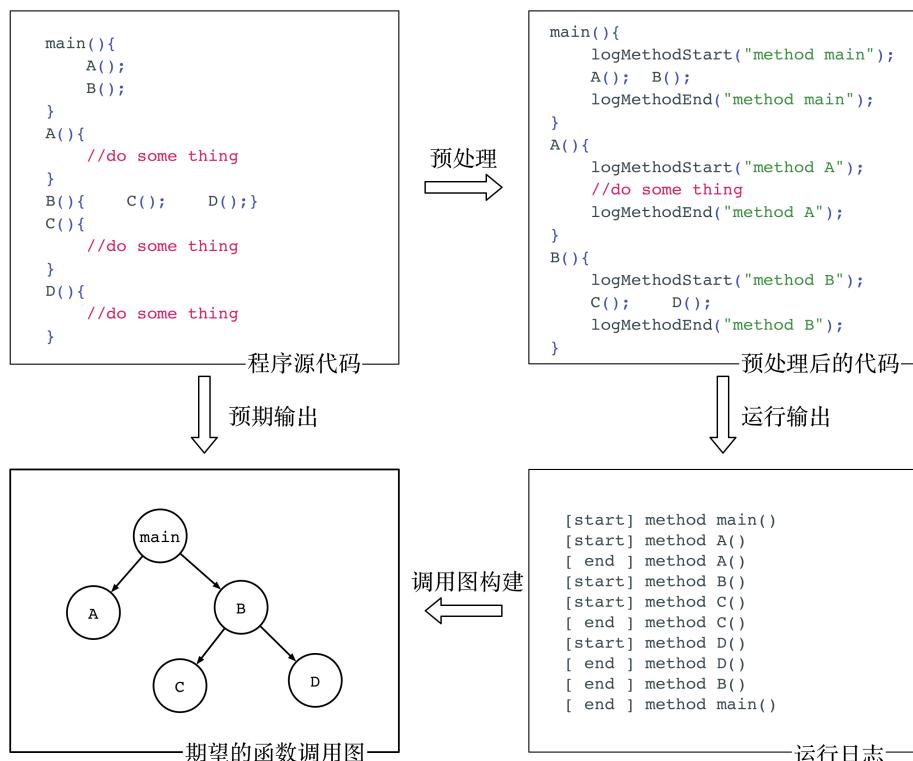


图 4.1: RunDroid 的处理流程

图 4.1-左上为一段示例代码：在 main 函数执行时，程序会依次调用 A、B 两个函数，而 B 函数则会调用了 C、D 两个函数。图 4.1-左下则是程序的运行时函数调用图，即 RunDroid 实际的输出产物。程序执行过程可以看做函数调用图的深度优先遍历过程。还原函数调用图的关键点，在于如何在程序执行过程中输出树的遍历序列，并根据遍历序列进行还原函数调用图。

本文采用的技术方案：RunDroid 通过对源程序（图 4.1-左上）进行预处理，得到包含日志记录功能的运行代码（图 4.1-右上）；程序在函数执行前后可以输出和方法执行相关的日志信息，（图 4.1-右下）；最后，我们根据这些日志信息构建出函数调用图（图 4.1-左下）。另外，在函数调用图的基础上，RunDroid 利用日志中包括的方法对象信息，挖掘和方法对象相关联的方法，结合触发规则，进而建立方法触发关系，形成最终的拓展函数调用图。

## 4.2 方法信息的捕获

在 Android 系统中，方法分为用户方法和系统方法两类。用户方法是由用户定义的，直接修改的成本较低，易于修改其运行行为。而系统方法在系统中定义，属于系统运行环境的一部份，行为修改成本较高。虽然记录这两种方法的执行信息的方式是不一样的，但是两者最终的效果是一致的：当应用程序运行时，相关方法的执行信息均会以日志的形式记录下来，用于后续调用图的构建。

### 4.2.1 用户方法执行信息的获取

为了获取应用程序中用户方法的执行信息，RunDroid 通过源代码插桩技术对应用做不影响业务逻辑的代码修改，产出包含日志代码的 APK 文件。该 APK 在运行期间会以日志形式输出用户方法的执行信息。

RunDroid 对用户方法进行源代码修改的处理过程如算法 4.1 所示：RunDroid 会遍历项目源代码中的 Java 源文件，将源文件和对应的抽象语法树转化成 XML 格式的文件，并以 DOM 的形式加载到程序中（第 3 行）。对于抽象语法树中的每

一个类和每一个方法，RunDroid 会计算出编译后所处的类的类名及方法签名（第4~7行）。全限类名和方法签名的组合<全限类名，方法签名>可以作为方法体的唯一标识，和抽象语法树中的方法体一一对应。在每个方法内部，预处理器会插入用户方法日志记录的代码，实现日志代码编织，达到用户方法运行时信息记录的目的（第8~10行）。另外，预处理器还对每个方法体进行了异常捕获处理，防止方法体内部的异常导致日志记录过程的中断，影响函数调用图的构建（第9行）。上述操作都是对 DOM 对象 *document* 直接操作的。当一个 Java 文件修改完毕后，预处理器会将 DOM 对象重新转换成 Java 源文件，并替换原来的文件（第11行）。最后，预处理器会利用编织后得到的源代码构建成 APK 文件（第12行）。

---

#### 算法 4.1：日志代码插桩过程

---

输入： *javaFiles*，应用程序的源代码

输出： *apk*，包含插桩代码的 APK 文件

```

1 Function buildApk(log):
2   for javaFile ∈ javaFiles do
3     将源文件 javaFile 的抽象语法树转化成 DOM 对象 document;
4     for classEle ∈ document do
5       className ← computeClassName(classEle,xmlFile);
6       for methodEle ∈ classEle do
7         methodId ← computeMethodId(methodEle,className );
8         addMethodExitLogCode(methodEle,methodId);
9         addMethodCatchLogCode(methodEle,methodId);
10        addMethodEnterLogCode(methodEle,methodId);
11
12     将 document 转化成新的 Java 文件 newJavaFile;
13     apk ← buildApk(newJavaFile)
14   return apk;

```

---

#### 4.2.2 系统方法执行信息的获取

由于无法直接对系统方法进行源代码修改，RunDroid 利用动态拦截技术对系统方法的执行进行拦截，达到获取系统方法的执行信息的目的。RunDroid 为待拦

截的系统方法维护了一个系统方法列表，具体如表 4.1 所示。列表中的方法通常和 Android 特性相关，涵盖 Activity 的生命周期方法、Java Thread API 以及 Handler 机制等多个方面。

在目标应用程序开始运行前，RunDroid 会为上述目标方法注册执行回调函数。在目标方法执行前和执行后，对应的回调函数会被执行，进而通过日志记录器记录下这些方法执行的相关信息。在 RunDroid 中，系统方法执行信息的获取，可以填补调用图缺失的系统方法执行，进而还原出应用层和系统层之间以及系统内部的方法调用，使得产生的调用图更加完整。

### 4.3 拓展函数调用图的构建过程

RunDroid 构建拓展函数调用图的过程分为如下几个阶段：根据程序运行时的日志提取函数间的调用关系，创建函数调用图；根据 `AndroidManifest.xml` 的 Activity 组件声明，在函数调用图上标识出完整的 Activity 生命周期流转序列；利用方法执行与方法对象的关联关系，结合触发关系规则，补全方法间的触发关系，形成最终的拓展函数调用图。

#### 4.3.1 构建函数调用图

由于在程序执行过程中，不存在一个调用关系跨越两个线程，因此，在整个构建过程中，RunDroid 以产生日志的线程为基本构建单元，向调用图添加方法调用关系。函数调用图的构建过程如算法 4.2 所示。

对于每个线程，RunDroid 顺序遍历对应的日志，使用栈  $stack$  的入栈、出栈操作来模拟对应线程的函数执行的过程，还原调用关系（第 2~20 行）：当读取到方法执行的开始日志时，系统会在调用图创建一个节点表示该方法的执行（第 7~8 行），同时也在调用图中添加方法参数、方法实例对应的对象节点  $o_p$ 、 $o_i$  以及方法对象和方法的关系  $\langle o_p \xrightarrow{\text{parameter}} m \rangle$ 、 $\langle o_i \xrightarrow{\text{instance}} m \rangle$ （第 9~10 行）。如果此时当前线程栈  $stack$  的栈顶方法元素  $top$  存在，系统会创建从方法  $top$  到当前方法  $m$  的调

表 4.1: 待拦截的系统方法列表

方法签名	说明
Activity.onCreate(Bundle)	
Activit.onStart()	
Activity.onResume()	
Activity.onPause()	
Activity.onStop()	
Activity.onDestroy()	
Thread.start()	和 Java 线程启动相关的方法
Message.obtain()	
Handler.enqueueMessage(MessageQueue,Message,long)	
Handler.dispatchMessage(Message)	
Handler.post(Runnable)	
Handler.postAtTime(Runnable,long)	
Handler.postAtTime(Runnable,Object,long)	
Handler.postDelayed(Runnable,long)	
Handler.postAtFrontOfQueue(Runnable)	
Handler.sendMessage(Message)	
Handler.sendEmptyMessage(int)	
Handler.sendEmptyMessageDelayed(int,long)	
Handler.sendEmptyMessageAtTime(int,long)	
Handler.sendMessageAtFrontOfQueue(Message)	
Handler.sendMessageDelayed(Message,long)	
Handler.sendMessageAtTime(Message,long)	和 Hanlder 机制相关的方法

用关系,  $\langle top \rightarrow m \rangle$ , 并将当前方法  $m$  压入栈  $stack$  中 (第 11~14 行)。当读取到方法执行的结束日志时, 该日志对应的方法必然是栈顶方法  $top$ , 若栈顶方法  $top$  存在返回对象  $o_r$ , 则只需要将  $o_r$  和  $top$  的关系添加到调用图中即可, 最后弹出栈顶的  $top$  即可 (第 16~18 行)。在上述过程中, 如果待添加的方法对象在调用图中已存在, 该对象复用调用图原有的对象节点即可, 无须重新添加, 只需添加方法与对象间的关系即可。

### 4.3.2 构建 Activity 的生命周期及事件回调的触发关系

#### § 构建 Activity 的生命周期

Android 应用的 Activity 生命周期的构建就是将 Activity 生命周期方法按照时间顺序串联。生命周期的构建过程以 AndroidManifest 文件作为输入，在原有的函数调用图的基础上，添加 Activity 生命周期变化的有向边，具体过程如算法 4.3 所示：对于 AndroidManifest 文件中声明的每一个 Activity 组件对象  $o_{activity}$ ，系统都会遍历以该对象为方法实例的方法  $m$ （即  $m$ 、 $o_{activity}$  满足条件  $o_{activity} \xrightarrow{\text{instance}} m$ ）；

---

#### 算法 4.2：函数调用图的构建过程

---

输入： $logs$ ，应用程序的运行时日志

输出： $ecg$ ，拓展函数调用图

```

1 Function buildExtendedCallGraph(log):
2   |   ecg ← new ExtendedCallGraph();
3   |   for thread ∈ logs.threads do
4   |     |   stack ← new Stack();
5   |     |   for log ∈ logs.get(thread) do
6   |       |     |   top = stack.peek();
7   |       |     |   if isMethodStartLog(log) then
8   |         |       |   m ← generateMethodInfo(log);
9   |         |       |   ecg.addMethodNode(m);
10  |         |       |   ecg.addMethodObjectsIfNotExists( $o_p, o_i$ );
11  |         |       |   ecg.addMethodObjectRels( $\langle o_p \xrightarrow{\text{parameter}} m \rangle, \langle o_i \xrightarrow{\text{instance}} m \rangle$ );
12  |         |       |   if top ≠ null then
13  |           |           |   ecg.addInvokeRel( $\langle top \rightarrow m \rangle$ );
14  |           |           |   stack.push(m);
15  |       |   else
16  |           |           |   ecg.addMethodObjectIfNotExists( $o_r$ );
17  |           |           |   ecg.addMethodObjectRel( $\langle o_r \xrightarrow{\text{return}} m \rangle$ );
18  |           |           |   stack.pop();
19   |   return ecg;

```

---

若方法  $m$  同时满足三个条件，则将它添加到列表  $lifecycleList$  中（第 7~11 行）。最后，将列表  $lifecycleList$  按照时间顺序进行排序，并依次连接起来，即可得到 Activity 的生命周期（第 12~13 行）。

这三个条件分别为：(1) 方法  $m$  的方法签名和图 2.2 中的任何一个生命周期方法的签名保持一致；(2) 方法  $m$  执行时所处的线程为主线程；(3) 方法  $m$  在调用图  $cg$  中不在调用者，即在调用图  $cg$  中不存在方法  $m' \in cg$ ，使得  $m' \rightarrow m$  成立。

通常情况下，条件 (1) ~ (2) 可以筛选出 Activity 组件相关的生命周期方法。但是，当生命周期方法重写时，子类方法往往需要回调父类方法，执行系统在该状态下的业务逻辑。例如，在重写方法  $onCreate()$  时，开发人员会调用  $super.onCreate()$  以完成 Activity 窗体的初始化。若只考虑前面两个条件，两个方法均会被看作生命周期方法。子类方法作为系统回调的一部分，应当保留在 Activity 的生命周期中，父类方法被子类方法调用，属于用户调用的范畴。为了区分上述两类场景下的方法调用，我们需要在筛选生命周期方法时考虑条件 (3)。

### § 构建 Android 事件回调的触发关系

在构建 Android 事件回调的触发关系时，我们会遍历拓展调用图中的所有 View 类型的对象节点  $o_{view}$  和 View.OnClickListener 类型的对象节点  $o_{listener}$  的组合。对于组合  $\langle o_{listener}, o_{view} \rangle$ ，若调用图中存在方法节点  $m_{register}, m_{click}$  满足式子 3.3，方法节点  $m_{register}, m_{click}$  之间添加从  $m_{register}$  指向  $m_{click}$  的有向边，即  $m_{register} \xrightarrow{UiEvent} m_{click}$ （第 15~18 行）。

#### 4.3.3 构建多线程触发关系

基于函数调用图构建多线程触发关系主要分为两个方面，基于 Java 的多线程交互与基于 Handler 的多线程消息调度，具体的过程如算法 4.4 所示。

## § 基于 Java 的多线程交互

基于 Java 的多线程交互往往是以 Runnable 作为传递对象，通常通过调用方法 *Thread.start()* 和 *Activity.runOnUiThread(Runnable)* 等 API，进而触发类 Thread 和 Runnable 的方法 *run()* 的执行。

对于方法 *Thread.start()* 相关的触发关系，我们会遍历拓展调用图中的所有

---

### 算法 4.3: 构建 Activity 的生命周期和事件回调

---

输入: *manifestFile*, AndroidManifest 文件

*ecg*, 函数调用图

输出: *ecg*, 包括 Activity 生命周期的函数调用图

```

1 Function patchActivityLifecycleAndUiEvent(ecg):
2     patchActivityLifecycle(ecg,manifestFile);
3     patchUiEvent(ecg);
4     return ecg;
5 Function patchActivityLifecycle(ecg,manifestFile):
6     lifecycleList ← new List();
7     for oactivity ∈ {act | act 为文件 manifestFile 中定义的 Activity} do
8         for m ∈ {m | m 为调用图 ecg 中以 oactivity 为实例的方法节点} do
9             if m 为 Activity 的生命周期方法 && m 的执行线程为主线程
10                && ⟨m' → m⟩ ∉ ecg then
11                    lifecycleList.add(m);
12
13 sortListByTime(lifecycleList);
14 linkItemsByTime(lifecycleList,ecg);
15
16 Function patchUiEvent(ecg):
17     for oview ∈ {o | o 为调用图 ecg 中 View 类型的对象节点} do
18         for olistener ∈ {o | o 为调用图 ecg 中 View.OnClickListener 类型的对象节点}
              do
                  if oview, olistener, mregister 和 mclick 满足式子 3.3 then
                      ecg.addTriggerRel(mregister  $\xrightarrow{UiEvent}$  mclick)

```

---

---

**算法 4.4: 扩展函数调用图的构建过程**


---

输入:  $ecg$ , 拓展函数调用图  
 输出:  $ecg$ , 包含触发关系的拓展函数调用图

```

1 Function addTriggerRels( $ecg$ ):
2   // 在调用图  $ecg$  中创建基于 Java 多线程交互的触发关系
3   for  $o_{thread} \in \{o \mid o$  为调用图  $ecg$  中的 Thread 类型的对象节点 } do
4     if  $o_{thread}$ 、 $m_{start}$  和  $m_{run}$  满足式子 3.4 then
5        $ecg.addTriggerRel(m_{start} \xrightarrow{\text{Thread}} m_{run})$ 
6   for  $o_r \in \{o \mid o$  为调用图  $ecg$  中的 Runnable 类型的对象节点 } do
7     if  $o_r$ 、 $m_{runOnUiThread}$  和  $m_{run}$  满足式子 3.5 then
8        $ecg.addTriggerRel(m_{runOnUiThread} \xrightarrow{\text{runOnUiThread}} m_{run})$ 
9   // 在调用图  $ecg$  中创建基于 Handler 多线程消息调度的触发关系
10  for  $o_m \in \{o \mid o$  为调用图  $ecg$  中的 Message 类型的对象节点 } do
11    if  $o_m$ 、 $m_{handle}$  和  $m_{dispatch}$  满足式子 3.6 then
12       $m_{send} \leftarrow calculateSendMethod(ecg, m_{enqueue});$ 
13       $m_{handle} \leftarrow calculateHandleMethod(ecg, m_{dispatch});$ 
14       $ecg.addTriggerRel(m_{send} \xrightarrow{\text{Handler}} m_{handle})$ 
15
    return  $ecg$ ;
  
```

---

Thread 类型的对象节点  $o_{thread}$ 。判断调用图中是否存在方法节点  $m_{start}$ 、 $m_{run}$  以及对象节点  $o_{thread}$  满足式子 3.4, 方法节点  $m_{start}$ 、 $m_{run}$  之间添加从  $m_{start}$  指向  $m_{run}$  的有向边, 即  $m_{start} \xrightarrow{\text{Thread}} m_{run}$  (第 2~4 行)。

对于方法  $Activity.runOnUiThread(Runnable)$  相关的触发关系, 我们会遍历拓展调用图中的所有 Runnable 类型的对象节点  $o_r$ 。判断调用图中是否存在方法节点  $m_{runOnUiThread}$ 、 $m_{run}$  以及对象节点  $o_r$  满足式子 3.5, 方法节点  $m_{runOnUiThread}$ 、 $m_{run}$  之间添加从  $m_{runOnUiThread}$  指向  $m_{run}$  的有向边, 即  $m_{runOnUiThread} \xrightarrow{\text{runOnUiThread}} m_{run}$  (第 5~7 行)。

## § 基于 Handler 的多线程消息调度

根据第二、三章的介绍可知，用户通过调用 Handler 提供的 API，将相关业务逻辑借助 Message 对象传递给目标线程的 Handler 对象，在目标线程执行相应的业务逻辑处理。在构建 Handler 触发关系过程中，首先，我们利用类 Handler 的方法 *enqueueMessage(Message)*（用  $m_{enqueue}$  表示）和 *dispatchMessage(Message)*（用  $m_{dispatch}$  表示）公用同一个 Message 对象的特点，找到所有的 Handler 底层函数触发关系， $m_{enqueue} \hookrightarrow m_{dispatch}$ （第 8 ~9 行）；对于每一个触发关系  $m_{enqueue} \hookrightarrow m_{dispatch}$ ，从  $m_{enqueue}$  顺着调用关系往上找到最上层的 Handler API 方法（即用户调用的 Handler API 方法， $m_{send}$ ）（第 10 行），从  $m_{dispatch}$  顺着调用关系往下找到用户定义的方法 *Handler.handleMessage(Message)*  $m_{handle}$ （第 11 行），最后在调用图中提交方法  $m_{send}$  和  $m_{handle}$  之间的触发关系  $m_{send} \xrightarrow{\text{Handler}} m_{handle}$ （第 12 行）。最终，我们得到用户通过 Handler 进行多线程消息调度的用户层面方法的触发关系。

## 4.4 本章总结

本章详细介绍了 RunDroid 的设计。首先，本章简要说明了 RunDroid 实现的基本功能，然后具体解释说明 RunDroid 的整体运行流程。在此基础上，本章介绍了 RunDroid 各个的流程的设计细节。RunDroid 利用源代码修改和运行时方法拦截相结合的方式，捕获用户方法和系统方法的执行信息，并以日志的形式保存方法执行信息。RunDroid 遍历各线程的方法执行日志结合栈的入栈、出栈操作，模拟各个线程的函数执行的过程，通过线程内部各个方法日志的嵌套关系还原出函数调用图，并在此基础上构建 Activity 的生命周期，补全方法间的触发关系，构建最终的拓展函数调用图。

## 第五章 RunDroid 的系统实现

RunDroid 使用 Java 作为开发语言，使用了 srcML、Xposed、Neo4j 等技术，由预处理器、运行时拦截器、日志记录器、调用图构建器等部分组成，整体架构图如图 5.1 所示。预处理器通过源代码插桩技术实现在用户方法运行时触发用户方法的信息记录，而运行时拦截器则是通过方法劫持技术实现对系统方法执行的拦截，进而触发日志记录。在应用运行时，日志记录器会以日志的形式将用户方法和系统方法对应的执行信息记录下来。最后，调用图构建器会根据应用程序运行时输出的日志，构建拓展函数调用图。

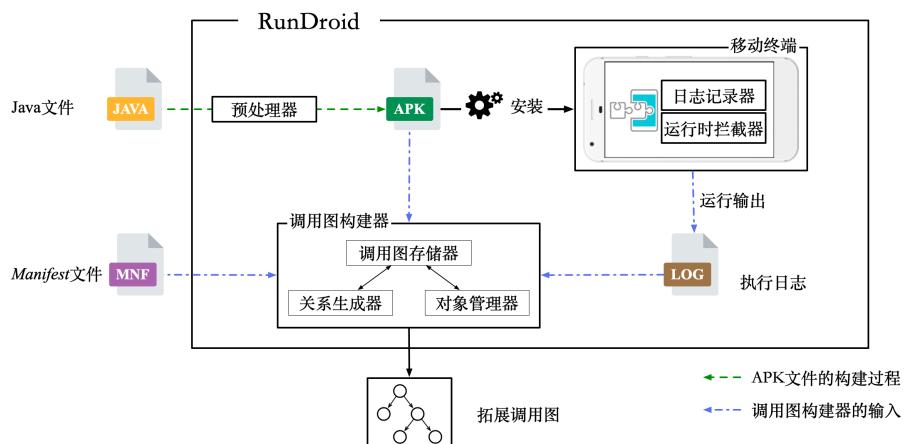


图 5.1: RunDroid 的整体架构图

### 5.1 模块实现

#### 5.1.1 预处理器

预处理器在 RunDroid 中的作用主要是用户方法的日志代码编织。在现有的技术上，代码编织主要分为两类：基于源代码的代码编织和基于字节码的代码编织。字节码编织技术以技术成熟、应用广泛的 AspectJ<sup>[41]</sup> 为代表技术，但是，字节码

插桩技术在运行过程中会生成过多的方法数，在 Android 应用构建过程中可能存在方法数 65K 限制问题。为此，预处理器采用的方案是基于源代码的代码编织方案。

在实现上，预处理器利用轻量级源代码分析工具 srcML<sup>[42]</sup> 对程序的源代码进行语法解析，将程序的抽象语法树转化成 XML 文件。在 XML 文件的基础上，预处理器直接对每个方法体进行修改，实现日志代码的编织，最后重新转换成源代码，经过构建得到可以运行的 APK 文件。该方案通过直接修改源程序，将日志记录代码写入在方法体内部，避免了新方法的引入，规避了方法数 65K 限制问题。

### 5.1.2 运行时拦截器

由于无法对系统方法的源代码进行直接修改，因此预处理器无法实现对系统方法的日志代码编织，无法达到系统方法信息记录的目的。为了弥补预处理器的不足，RunDroid 中的运行时拦截器需要在系统层面上实现对系统方法的拦截，实现后续方法信息的日志记录。

在实现上，运行时拦截器是基于 Xposed<sup>[43]</sup> 实现的插件，它维护的列表包括了所有需要拦截的系统方法。每当目标应用程序启动时，运行时拦截器通过 Xposed 提供的 API *XposedHelpers.findAndHookMethod()* 将表 4.1 中的目标方法绑定到方法钩子（即类 HookCallBack）上。在应用程序的运行过程中，HookCallBack 对象的方法 *beforeHookedMethod(MethodHookParam)* 会在目标方法执行之前被调用，方法 *afterHookedMethod(MethodHookParam)* 会在目标方法执行后调用。上述两个方法的参数 *MethodHookParam* 中会包括该方法执行时的相关方法对象。最终，HookCallBack 对应会将方法执行的相关信息传递给日志记录器。

相比定制化 Android 系统，通过 Xposed 实现的运行时拦截器兼容性良好，适用于当前主流的 Android 系统版本，实现成本低。同时，待拦截方法列表的编辑功能的引入，使得在不修改插件代码的情况下，支持动态添加删除待拦截的方法，避免了 Xposed 插件修改后的系统重启操作，提高了 RunDroid 整体的执行效率。

### 5.1.3 日志记录器

日志记录器的职责是将方法执行的消息以日志文件的形式进行持久化存储。针对不同的方法类型（静态方法与非静态方法、用户方法与系统方法等），日志记录器提供了不同的 API，帮助我们记录相应方法执行的日志信息。在日志内容上，日志记录器还会记录每个方法执行的时间戳、所处线程、方法签名标识、所处阶段（方法开始执行阶段/方法执行完毕阶段）以及相关的方法对象信息。方法对象信息主要包括对象的类型、属性和全局 ID（通过 Java API `System.identityHashCode(Object)` 获取）等。同时，日志记录器还支持对象信息的自定义：开发人员可以根据自身的需要为不同类型的对象输出不同对象数据信息。

### 5.1.4 调用图构建器

调用图构建器由调用图存储器、对象管理器和关系生成器组成，如图 5.1 所示。

调用图存储器将 Neo4j<sup>[44]</sup> 作为调用图的存储引擎，承担着拓展函数调用图的存储、查询、展示的职责。Neo4j 的数据分为节点和关系两种类型，支持自定义键值属性。同时，Neo4j 可为节点指定标签，为关系指定类型，用于区分不同含义的节点和关系。拓展函数调用图中的数据在 Neo4j 中的表现形式如下：调用图中的方法和对象会以节点的形式出现在调用图中，方法节点的标签分为 *METHOD* 和 *FRAMEWORK*，对象节点的标签为 *OBJECT*；方法间关系分为调用关系和触发关系，分别用关系 *INVOKER* 和 *TRIGGER* 表示；方法和对象之间的关系分为参数关系、返回值关系和实例关系，分别用关系 *PARAMETER*、*RETURN*、*INSTANCE* 表示。

对象管理器负责将日志中的对象信息映射成调用图中的节点。通常情况下，我们认为对象和全局 ID 一一对应，所以，我们将全局 ID 相同的对象信息映射成调用图中的一个节点。但考虑到有些类型（例如 Handler 机制中的 Message）使用对象池技术，我们还在全局 ID 的基础上引入对象版本号，避免对象在不同生命周期的串用。以 Message 为例，调用图构建过程中，如果对象管理器发现待提交的 Message

对象  $m$  是方法  $Message.obtain()$  的返回值，处理逻辑如下：如果调用图中不存在一个全局 ID 和  $m$  一致的节点，则对象管理器会创建一个全新的节点来表示  $m$ ，设置其对象版本号为 1；而调用图已经存在一个全局 ID 和  $m$  一致，版本号为  $version$  的节点  $m'$ ，则对象管理器会创建一个全新的节点来表示  $m$ ，并将  $version + 1$  作为节点  $m$  的版本号，而不是复用原有节点象  $m'$ 。因此，在对象管理器的角度看来，两个  $Message$  对象只有当全局 ID 和对象版本号都一致时，才是同一个对象。

关系生成器是算法 4.2 ~ 算法 4.4 的具体实现，基于 Cypher 脚本和 Soot 等技术共同实现。当我们需要找到使用  $o_m$  作为参数的方法  $m_{enqueue}$ 、 $m_{dispatch}$ （算法 4.4 第 11 行）时，可以直接使用脚本图 5.2 直接在调用图中找到所有符合条件的节点。相比传统的编程语言，Cypher 在实现相同逻辑时具有表达形式简介、代表阅读性好、开发效率好的特点。Soot 的工作主要是提供应用程序相关的类继承检索服务。例如，算法 4.4 第 7 行中，我们需要知道所有的 `Runnable` 对象，此处便需要通过 Soot 查询所有的 `Runnable` 子类。在算法 4.3 中，我们也会将 Soot 和 Cypher 相结合用于查找所有 `Activity` 的生命周期方法。

---

```
MATCH (m_enqueue:METHOD)-[ :PARAM ] -> (o_m:OBJECT) ,
  (m_dispatch:METHOD)-[ :PARAM ] -> (o_m:OBJECT)
return o_m, m_enqueue, m_dispatch
```

---

图 5.2: 使用 Cypher 查找共用对象  $o_m$  的方法  $m_{enqueue}$ 、 $m_{dispatch}$

## 5.2 本章总结

本章侧重介绍 RunDroid 的系统实现。在整体上，RunDroid 由预处理器、运行时拦截器、日志记录器、调用图构建器等部分组成。预处理器和运行时拦截器分别对应应用程序在执行过程中用户方法和系统方法的执行拦截，以触发日志记录器记录方法执行信息。预处理器是通过源代码编织实现的，避免了新方法的引入，规

避了方法数 65K 限制问题。运行时拦截器借助 Xposed 框架提供的方法执行劫持技术实现对系统方法的执行拦截，在实现上规避了 Xposed 插件修改后的重启系统操作。调用图构建器将运行过程中方法执行的日志信息作为输入，在 Neo4j 图数据库上构建函数调用图，利用图中方法和对象之间的关系，利用 Soot 和 Cypher 语句实现 Activity 生命周期的构建、方法触发关系的生成，形成最终的扩展函数调用图。

## 第六章 RunDroid 的系统实验

实验相关配置的环境如下：实验平台是 ThinkPad E430，CPU 为 Intel 酷睿 i5 3210M，内存为 12GB。手机型号为小米 MI 5，处理器型号为骁龙 820，RAM 为 3GB，系统为 Android 6.0，内置 Xposed 运行环境。

### 6.1 实验验证——函数触发关系的普遍性

相比其它分析技术，RunDroid 最为重要的创新点为在函数调用图中展示函数触发关系（例如多线程相关的触发关系）。为了验证函数触发关系在 Android 程序运行过程中的普遍性，我们从开源社区 F-Droid<sup>[45]</sup> 中随机抽取了 9 个应用，利用 RunDroid 获取各个应用的运行时日志信息，构建对应的动态函数调用图。最终，我们对调用图中的用户方法节点数、系统方法节点数、方法间调用关系及函数触发关系等数据进行了统计，各项数据如表 6.1 所示。

观察用户方法、系统方法、函数调用关系三列，我们发现等式“用户方法 + 系统方法 - 函数调用关系 = 1”并不成立。函数调用图的构建过程保证了函数调用图本身是一个有向无环图。因此，从图论的角度看，函数调用图不是图论中的树，而是有若干个树组成的森林。这也反映 Android 基于事件驱动的系统架构：在 Android 应用程序中，逻辑代码分散在不同片段中，通过系统调度在时间维度上串联起来，进而实现完整的业务逻辑。

从表 6.1 中看，应用程序在执行过程中，被 RunDroid 捕获到的函数触发关系大都集中在 10~39 之间。而在这个数值上，应用 osmtracker-android 和 archwiki-viewer 属于两个不同的极端。osmtracker-android 中的触发关系只有 0 个。经过分析，我们发现 osmtracker 中，用户操作大多与菜单、对话框与 ListView 产生交互操作，多线程相关操作是通过 AsyncTask 进行的，而不是通过 Handler 进行。而 archwiki-

viewer 捕获到的触发关系多达 102 个，是所有应用中函数触发关系最多的。分析发现，archwiki-viewer 是一个基于浏览器的应用程序。RunDroid 捕获到的 102 个触发关系中并不属于事件回调相关的触发关系，都是基于 Handler 的触发关系，大部分对应的 Handler 是 *org.chromium.base.SystemMessageHandler*，属于浏览器内部业务逻辑。archwiki-viewer 中的浏览器需要根据网页的加载情况，定时地更新浏览器界面，因此产生大量和 Handler 相关的触发关系。

由此，我们可以看出函数触发关系在 Android 应用中的普遍性：Android 应用中的逻辑实现依赖于回调函数和多线程通信等函数触发关系。

表 6.1: 各应用运行过程的统计数据

应用	用户方法	系统方法	函数调用关系	函数触发关系
AnyMemo	1021	322	1105	38
Microphone	31	230	161	31
Quran For My Android	156	204	225	35
ReGeX	1415	95	1323	39
upm-android	118	337	355	25
screenrecorder	1301	320	879	10
chanu	1086	353	397	34
osmtracker-android	997	328	1162	0
archwiki-viewer	195	360	310	102

## 6.2 应用结果展示

在本节中，我们会对 RunDroid 运行结果做出相应的展示，并将展现结果和静态分析工具 FlowDroid 的输出结果进行对比，分析各自的优劣。

我们研究的 APK 的主体代码如图 6.1 所示。在图 6.1 中，我们声明了 Activity 组件 *MainActivity*，他是应用层的主 Activity，该 Activity 界面主要有 3 个按钮组成，方便对应的是斐波拉契数量的计算、启动一个 Activity（生命周期相关）以及基于 Handler 的异步事件。

### 6.2.1 函数调用图的构建结果展示

在应用程序运行时，点击按钮 button1，应用会计算斐波拉契数列中的第 5 项，并将这个数以 Toast 的形式展示给用户。上述过程中，方法调用同时涉及到普通方法调用、递归函数调用。RunDroid 的动态分析结果如图 6.2 所示：每一个蓝色节点对应的是方法执行，每一个粉色节点对应是一个对象；如果对象是方法执行的方法对象，则在图中会有一条边从方法指向该对象，并在边上标识两种之间的关系（蓝色有向边表示参数关系、黄色有向边表示返回值关系等）；如果两个方法执行之间存在调用关系，则他们之间会通过从调用发起方指向被调用方的绿色有向边进行连接<sup>3</sup>。

在本案例中，*doHandleButton1()* 调用了方法 *doFibonacci()*，因此前者有条有向边指向后者；通过观察虚线框中各节点的关系，我们知道，对于方法 *doFibonacci()*，当参数为 5 时，对应的结果为 5。FlowDroid 的静态分析结果如图 6.3 所示。通过比较图 6.2 和图 6.3，我们可以发现以下有趣的现象：

首先，两者在方法 *doFibonacci()* 数量上不同的：RunDroid 得到的结果中，方法节点 *doFibonacci()* 共有 9 个，即在运行过程中，*doFibonacci()* 被调用了 9 次。由于在一个应用中，每个方法的方法签名是唯一的，所以 FlowDroid 给出的结果中 *doFibonacci()* 只有一个节点。该节点上存在一个指向自己的环，表示这个方法在执行过程中可能出现递归调用的情况。FlowDroid 得到的函数调用图往往以方法体本身作为研究的基本单元，因此只能表示方法之间的可能存在调用关系，不能刻画具体的执行过程，而 RunDroid 的函数调用图可以细化方法执行之间的关系，详细地反映了程序执行的具体过程。

其次，在运行过程中，斐波拉契数列的计算结果最终以 Toast 的形式展现在界

<sup>3</sup>由于所有方法的实例对象都是 MainActivity，所以实例关系在图中不做展示；图中未标出方法名的方法节点均为方法 *doFibonacci()*。

面上。但方法 `Toast.makeText(Context,CharSequence,int)` 没有出现在 RunDroid 的结果中，却出现在 FlowDroid 给出的结果。出现这个现象的原因如下：该方法属于系统定义的方法，而运行时拦截器待拦截的方法列表中并未包含该方法，因此运行时未收集到相关的方法执行信息，导致 RunDroid 无法在调用图中还原相应方法的方法执行。这也反映了 RunDroid 的不足：关于系统函数的运行时信息的捕获需要提前将相应的方法告知运行时拦截器，生成的调用图才会包括相应的方法。虽然 RunDroid 得到函数调用图缺失了部分方法，存在局部不完整，但是调用图中只包含研究人员关心的方法执行信息，突出研究重点，避免了分析信息的繁杂。

最后，FlowDroid 给出的结果中包括了大量的 `StringBuilder` 相关的方法，而 RunDroid 的结果并不包括这些方法：对比源码发现，源程序并未直接使用 `StringBuilder`。通过查阅文献 [ 46]，在编译过程中，Java 编译器会通过对源代码做适当等价修改，通过引入 `StringBuilder`，避免表达式求值过程中产生过多的中间字符串对象，提高字符串连接的性能。由于上述过程发生在 Java 程序编译阶段，并最后以字节码的形式保存在 APK 文件中，而 FlowDroid 从字节码层面对应用进行分析，因此 FlowDroid 的分析结果会包含该方法。对于 RunDroid，上述方法既在源代码中未出现相关方法定义（无法进行日志代码编织），又不在运行时拦截器的目标方法列表中，所以 RunDroid 给出的调用图自然也不会包括 `StringBuilder` 相关的方法。

### 6.2.2 Activity 生命周期和事件回调的效果展示

在本节中，应用运行时，我们将点击按钮 `button2`，在 `MainActivity` 启动另一个 `NewActivity`，对比 RunDroid 和 FlowDroid 在 Activity 生命周期和事件回调的效果展示。最终，我们得到的 RunDroid 的运行结果如图 6.4 所示。针对 Android 组件 `Activity` 的生命周期特性，FlowDroid 进行了针对性的建模，将 `Activity` 的生命周期和 UI 事件回调串联起来，构造应用层面的方法 `dummyMainMethod()` 表示应用的执行过程，相应的结果如图 6.5 所示。

对比图 6.4 和图 6.5，我们可以发现 RunDroid 和 FlowDroid 在组件生命周期和事件回调上的设计思想上是不同的：RunDroid 可以捕获到组件生命周期方法以及回调事件方法的实际执行，因此，RunDroid 对于上述行为的展现是按照时间的先后顺序将这些方法节点串联起来，形成完整的事件序列。而 FlowDroid 在生成函数 *dummyMainMethod()* 时，会为 AndroidManifest.xml 文件中定义的每一个 Activity 单独创建包含生命周期方法和事件回调方法的状态迁移图（图 6.5 中的虚线框部分表示每个 Activity 各的整体状态迁移，灰色部分为 MainActivity 中的 UI 事件响应方法），最后将这些 Activity 的状态迁移串联起来，并将 Action 为 *android.intent.action.MAIN* 并且 category 为 *android.intent.category.LAUNCHER* 的 Activity 组件作为结果的默认启动的 Activity，最终形成方法 *dummyMainMethod()*。

在结果展现上，RunDroid 倾向于展现所有和 Activity 生命周期相关的方法。例如，虽然开发人员在源代码中没有定义 *onResume()* 等方法，但在运行过程中，MainActivity 的父类方法 *onResume()* 被调用了，便会出现 RunDroid 的结果中。而 FlowDroid 在这点上的处理方式恰恰相反：FlowDroid 认为一个父类方法未被重写，则该方法运行行为表现保持一致性，FlowDroid 并不会在结果中展示父类 Activity 未重载的生命周期方法。

另外，我们发现在应用从 MainActivity 切换到 NewActivity 过程中，对应的生命周期变化为 *MainActivity.onPause() → NewActivity.onCreate() → NewActivity.onStart() → NewActivity.onResume() → MainActivity.onStop()*，MainActivity 和 NewActivity 的生命周期方法是交替出现的，并不是 MainActivity 的生命周期方法全部执行完毕后才执行 NewActivity 的生命周期方法。而 FlowDroid 给出的结果将 MainActivity 和 NewActivity 分开处理，因此得到的结果属于后者。相比 FlowDroid，RunDroid 的运行结果是程序运行时的直接反映，真实地反映了应用的状态变化。

### 6.2.3 多线程触发关系效果展示

多线程开发是 Android 开发中过程的典型开发要求。本文针对此场景，我们进行了相关的测试。在本节，我们将点击按钮 button3，获取 *doHandleButton3()* 相关的调用图情况。经过实验，RunDroid 和 FlowDroid 的运行结果分别如图 6.6、图 6.7 所示。两幅图在以下节点上存在不同。

#### § Java API 多线程的触发方式

RunDroid 在方法 *Thread.start()* 和方法 *WorkerThread.run()* 之间添加了一条有向边。这条有向边从前者指向后者，表示方法 *Thread.start()* 的执行触发了方法 *WorkerThread.run()* 的执行，边上标识着触发原因（*Thread*, 通过 *Thread* 方式触发）。FlowDroid 对方法 *doHandleButton3()* 进行推算出图 6.1 第 42 行中的变量 *workerThread* 类型为 *WorkerThread*。同时，FlowDroid 可以推算出方法 *Thread.start()* 的执行会导致方法 *WorkerThread.run()* 的执行。在 FlowDroid 的设计者看来，方法 *Thread.start()* 可以替换成方法 *WorkerThread.run()*。因此，FlowDroid 的结果中，方法 *doHandleButton3()* 和 *WorkerThread.run()* 之间存在调用边。这条有向边并不是像 RunDroid 从 *Thread.start()* 出发。虽然 RunDroid 和 FlowDroid 都可以在函数调用图表现 Java API 多线程的触发方式，但相比之下，RunDroid 针对性地标识了函数间的关系，呈现方式更为全面严谨，更体现程序的运行过程。

#### § 基于 Handler 的多线程触发方式

在 *WorkerThread* 的方法 *run()* 中，我们通过 *Handler* 进行了异步 UI 操作，触发了方法 *MainActivity\$1.handleMessage(Message)* 的执行。在构建拓展调用图过程中，RunDroid 充分挖掘了这些方法和对应的方法对象的关联关系，进而补全方法 *Handler.sendMessage()* 和 *MainActivity\$1.handleMessage(Message)* 之间的触发关系。因此，在 RunDroid 展示的结果中，上述两个方法之间存在一个从前者指向后者的有向边，边上标识着相应的触发原因（通过 *Handler* 方式触发，*Handler*）

<sup>4</sup>。但是，FlowDroid 分析相关代码时，由于缺少运行时上下文信息，无法分析出 *WorkerThread.run()* 中 handler 的具体类型，因此无法将这两个方法连接起来。

### § 静态方法的使用

另外，在 FlowDroid 给出的结果中，我们发现方法 *doHandleButton3()* 还调用了类 *Message* 的类初始化方法 *<clinit>()*。原因是方法 *doHandleButton3()* 调用类 *Message* 的静态方法 *obtain()*，因此可能需要进行类的初始化。由于在程序运行过程中，方法 *doHandleButton3()* 在执行方法 *Message.obtain()* 前，类 *Message* 已经完成了类的初始化，方法 *Message.<clinit>()* 并没有被调用。在 RunDroid 的结果中，上述两个方法不存在调用关系。这从一个侧面方面反映出 FlowDroid 分析的结果和动态运行的过程可能存在一定的偏差。

## 6.3 RunDroid 在错误定位领域的应用

在本节中，我们将采用因果关联模型<sup>[47,48]</sup>作为错误定位技术。相比传统错误定位技术（基于频谱的错误定位）只考虑了程序语句的覆盖情况，基于因果关联模型的错误定位技术还考虑程序在执行过程中的程序依赖（即控制依赖和数据依赖），分析的准确度较高。

$$\tau(s) = E[Y = 1|T = 1] - E[Y = 0|T = 0] \quad (6.1)$$

注：  $E[Y = 1|T = 1]$ ：实验组执行失败的期望， $E[Y = 0|T = 0]$ ：对照组执行通过的期望。

$$Y = \alpha + \tau T + \beta X \quad (6.2)$$

注：其中  $Y$  表示测试用例是否执行失败（1 为执行失败，0 为执行通过）， $T$  和  $X$  分别为语句  $s$  和  $Pred(s)$  中各语句的覆盖情况， $\beta$  为  $Pred(s)$  中各语句的错误估计值， $\alpha$  为公式中待计算的定值。

---

<sup>4</sup>由于我们定义的 Handler 是 *MainActivity* 的匿名内部类，因此编译后的类名为 *MainActivity\$1*。

### 6.3.1 原理简介

基于因果关联模型的错误定位技术，以程序本身  $P$  和程序在一组测试用例下的覆盖率信息作为输入，通过期望模型（式子 6.1）和线性回归模型（式子 6.2）计算程序  $P$  中的语句  $s$  的相应的错误估计值  $\tau$ 。 $\tau$  的值越大，语句  $s$  是错误的可能性越大。具体过程如算法 6.1 所示。

---

#### 算法 6.1：错误定位的计算方法

---

输入:  $P$ , 应用程序

输入:  $CoverageInfo$ , 应用程序的覆盖信息

输出:  $sorted_{\tau}$ , 程序语句的逆向排序

1 **Function** *FaultLocazilation*( $P, CoverageInfo$ ):

```

2   for  $s \in P$  do
3       计算语句  $s$  在程序  $P$  中前驱节点集合  $Pred(s)$ ;
4       根据  $Pred(s)$  的覆盖率情况筛选待计算的数据  $Mdata(s)$ ;
5       if  $Mdata(s)$  为  $\emptyset$  then
6           使用  $Mdata(s)$  根据式子 6.1 计算  $\tau(s)$  ;
7       else
8           根据式子 6.2 计算  $\tau(s)$  ;
9   对各语句按照  $\tau$  逆向排序得到  $sorted_{\tau}$ ;
10  return  $sorted_{\tau}$ ;
```

---

对于语句  $s$ , 语句  $s$  的前驱节点集合  $Pred(s)$  指的是语句  $s$  在程序  $P$  中的控制依赖和数据依赖的合集（第 3 行）。在匹配过程中，每个测试用例按照语句  $s$  前驱节点集合  $Pred(s)$  的覆盖率情况表示成一维向量，并按照语句  $s$  是否覆盖将测试用例分为实验组和对照组（实验组 ( $T=1$ ) 和对照组 ( $T=0$ ) 分别与覆盖语句  $s$  的测试用例、未覆盖语句  $s$  的测试用例对应）；如果向量表示结果相同的测试用例同时出现在实验组和对照组中，则保留对应的测试用例到集合  $Mdata(s)$  中，反之不留（第 4 行）。匹配完毕后，当  $Mdata(s)$  不为空集，我们将使用式子 6.1 计算语句  $s$  的错误估计值  $\tau(s)$ （第 6 行）。当  $Mdata(s)$  为空集，我们将通过线性回归式子 6.2 公

式计算得到关于  $T$  的斜率  $\tau$  作为语句  $s$  的错误估计值  $\tau(s)$  (第 8 行)。最后, 对于所有的语句, 按照错误估计值逆向排序输出, 算法完毕。

### 6.3.2 结果分析

为了验证 RunDroid 提供的动态调用图对因果影响模型有一定的提升作用, 我们按照 [ 47,48] 的实验设置, 使用原始因果影响模型的计算结果 (没有 RunDroid) 作为基线, 和并使用 RunDroid 后的因果影响模型的计算结果进行比较。我们的实验对象如表 6.2 所示, 代码 *Line .13* 为程序的错误 (方法 *loadData(int)* 的参数不可为负数)。在编写的 5 个测试用例  $t_1 \sim t_5$  中,  $t_2$ 、 $t_5$  未通过测试。列  $\tau$  为未使用 RunDroid 情况下的计算结果, 列  $\tau'$  为使用 RunDroid 情况下的计算结果。

表 6.2: 结果对比

v num	$t_1$ btn0	$t_2$ btn1	$t_3$ btn1	$t_4$ btn2	$t_5$ btn1	$\tau$	$\tau'$
<i>Line .1</i> void onClick(View v) {							
<i>Line .2</i> num = getNumber();	1	1	1	1	1	NA	NA
<i>Line .3</i> if(v.getId() == R.id.btn1) {	1	1	1	1	1	NA	NA
<i>Line .4</i> if( num == 0 ) {	0	1	1	0	1	0.67	0.67
<i>Line .5</i> num=1;	0	0	1	0	0	-1.0	-1.0
<i>Line .6</i> }							
<i>Line .7</i> }							
<i>Line .8</i> Thread t = createThread(v.getId());	1	1	1	1	1	NA	NA
<i>Line .9</i> t.start();	1	1	1	1	1	0.67	0.67
<i>Line .10</i> }							
<i>Line .11</i> TaskThread.run() {							
<i>Line .12</i> if(v.getId() == R.id.btn1) {	1	1	1	1	1	NA	NA
<i>Line .13</i> loadData(num); /* FAULT */	0	1	1	0	1	0.67	1.0
<i>Line .14</i> }							
<i>Line .15</i> }							
	0	1	0	0	1		

注: 第 2 列到第 6 列分别代表测试用例  $t_1 \sim t_5$ ; 每个测试用例列的标题分别显示在第 1 行和第 2 行使用的  $v$  和  $num$  的值; 测试用例列的值指示相应的程序语句是否由测试用例执行, 1 表示覆盖, 0 表示未覆盖; 最后一行表示各个测试用例的执行结果, “1” 表示测试未通过 ( $Y = 1$ ), “0” 表示测试通过 ( $Y = 0$ )。

在  $\tau$  的结果中，分数最高包括 *Line .4*、*Line .7* 和 *Line .13*，均为 0.67；在  $\tau'$  的结果中，分数最高只有 *Line .13*，分数从原来的 0.67 提升到 1。而其他行的分数保持不变。在 RunDroid 接入后，错误语句的分数有所提高，同时错误语句与非错误语句得到区分。因此， $\tau'$  的结果比结果  $\tau$  更好。

*Line .13* 分数提高的原因如下：在接入 RunDroid 前，方法 *onClick(View)* 和 *TaskThread.run()* 没有任何调用关系。因此，*Line .13* 的前驱节点集合中只包括控制依赖语句 *Line .12*，实验组和对照组的 *Mdata( Line .13)* 数据如图 6.8-(a) 所示，所有的测试用例参与计算： $\tau = \frac{1+1+0}{3} - \frac{0+0}{2} = 0.67$ 。而接入 RunDroid 后，方法 *onClick(View)* 和 *TaskThread.run()* 通过方法触发关系产生方法间的因果关联，进而形成新的程序依赖关系。*Line .13* 的前驱节点增加了 *Line .2*、*Line .5* 和 *Line .9*，变为  $\{Line .2, Line .5, Line .9, Line .12\}$ ，实验组和对照组的 *Mdata( Line .13)* 如图 6.8-(b) 所示。实验组中的测试用例  $t_3$  在对照组中找不到对应的测试用例，因此不参与计算。最后，根据测试用例  $t_1$ 、 $t_2$ 、 $t_4$ 、 $t_5$  得到 *Line .13* 的错误估计值： $\tau = \frac{1+1}{2} - \frac{0+0}{2} = 1$ 。*Line .12* 的前驱节点也发生了变化，但是由于实验组和对照组匹配结果保存不变，因此结果保存不变。而对于其他语句，前驱节点的信息并没有发生变化，因此数值保持不变。

RunDroid 提供的动态调用图，从方法调用关系和方法触发关系两个方面反映方法间的因果关系，辅助补全函数间的控制依赖和数据依赖。相比之前的技术只从函数调用角度分析程序，RunDroid 辅助下的程序分析结果更为全面，对因果影响模型有一定的提升作用。因此，RunDroid 和错误定位技术的结合能使得结果得到一定的提升。

## 6.4 本章总结

我们从开源社区 F-Droid 中下载 9 个开源应用程序，利用 RunDroid 产生这些应用程序的动态函数调用图，统计图中的函数触发关系。我们发现函数触发关系（事

件回调、Java 多线程以及 Handler) 普遍存在于应用程序的业务中，特别是 Handler 机制在 Android 应用程序中的使用尤为广泛。

我们将 RunDroid 产生的动态函数调用图和 FlowDroid 产生的静态函数调用图进行对比分析。我们发现，RunDroid 生成的调用图的完整性依赖于源代码处理以及待拦截的目标系统方法列表的完整程度，而 FlowDroid 的分析对象是 Android 应用程序 APK 文件，调用图的完整性受到自身算法实现的限制。另外，两者在设计思想上不同的，RunDroid 关心的是在程序执行过程中的方法之间的依赖关系，而 FlowDroid 则是站在函数调用可能性的角度，体现两个方法（包括递归调用）之间是否存在调用的可能性。前者是某一应用运行的具体、微观的表现，而后者更多的是反映方法调用关系在宏观、理论上的可能性。综上，RunDroid 和 FlowDroid 体现方法的调用关系上各有千秋。

同时，我们将 RunDroid 和错误定位技术结合，进行了探索性的实验。实验结果显示，RunDroid 生成的调用图中的方法触发关系可以将异步方法调用（例如 Java 多线程和 Handler）关联起来，补全函数间的控制依赖和数据依赖关系。在 RunDroid 提供的信息的帮助下，基于因果模型的错误定位技术相关实验结果得到一定的提升，提高了实验结果的准确度。

```
1 package cn.mijack.rundroidtest;
2
3 public class MainActivity extends Activity implements View.OnClickListener {
4     Button button1,button2,button3;
5     Handler handler = new Handler() {
6         public void handleMessage(Message msg) {
7             if (msg.what == 1)
8                 Toast.makeText(MainActivity.this, "handle", Toast.LENGTH_SHORT).show();
9         }
10    };
11    protected void onCreate(Bundle savedInstanceState) {
12        super.onCreate(savedInstanceState);
13        setContentView(R.layout.activity_main);
14        button1 = findViewById(R.id.button1);
15        button1.setOnClickListener(this);
16        // button2、button3进行相同的操作，此处省略
17    }
18    public void onClick(View view) {
19        switch (view.getId()) {
20            case R.id.button1:
21                doHandleButton1();
22                return;
23                // button2、button3进行相同的操作，此处省略
24        }
25    }
26    public void doHandleButton1() {
27        int fibonacci = doFibonacci(5);
28        Toast.makeText(this, "fibonacci: " + fibonacci, Toast.LENGTH_SHORT).show();
29    }
30    private int doFibonacci(int i) {
31        if (i < 1)      return -1;
32        if (i == 1 || i == 2)      return 1;
33        return doFibonacci(i - 1) + doFibonacci(i - 2);
34    }
35    public void doHandleButton2() {
36        Intent intent = new Intent(this, NewActivity.class);
37        startActivity(intent);
38    }
39    public void doHandleButton3() {
40        Thread workerThread =new WorkerThread(handler);
41        workerThread.start();
42    }
43 }
```

图 6.1: MainActvitiy 的代码

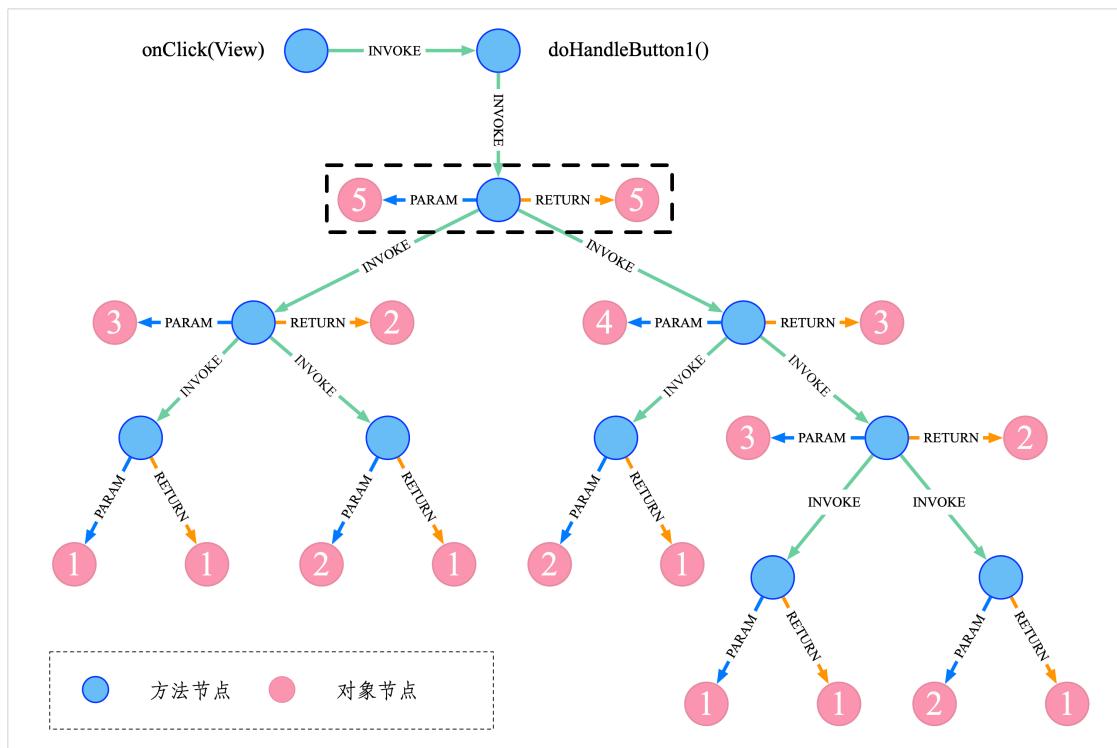


图 6.2: 斐波拉契数列相关的 RunDroid 调用图 (局部)

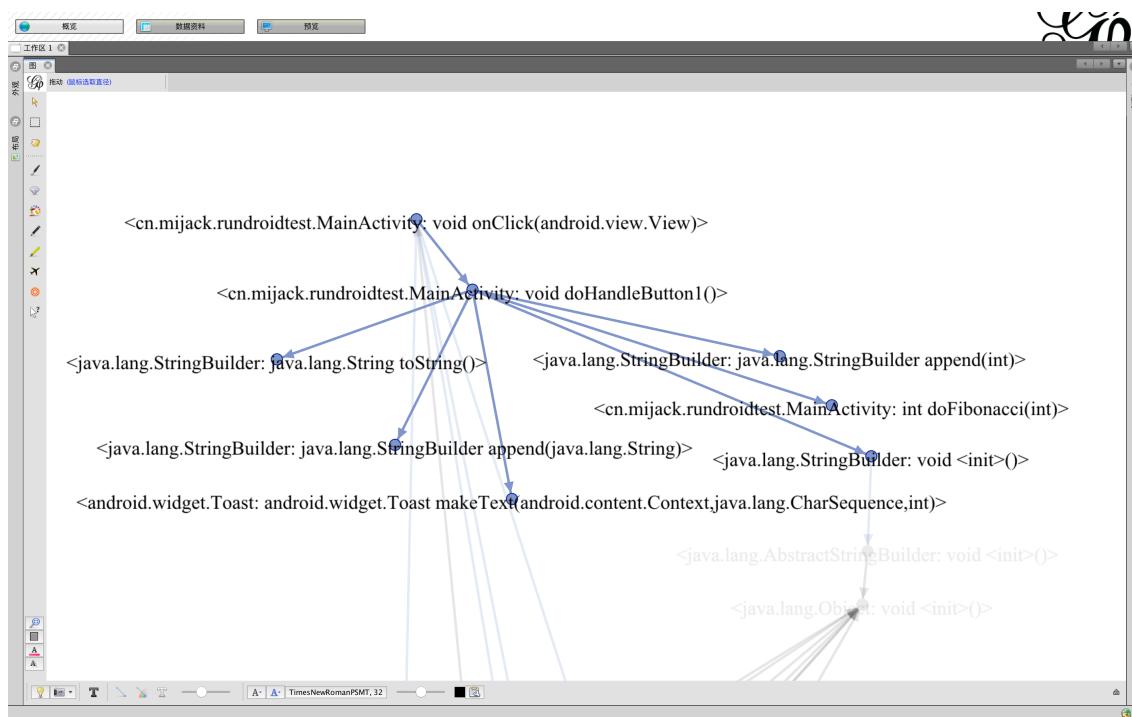


图 6.3: 斐波拉契数列相关的 FlowDroid 调用图 (局部)

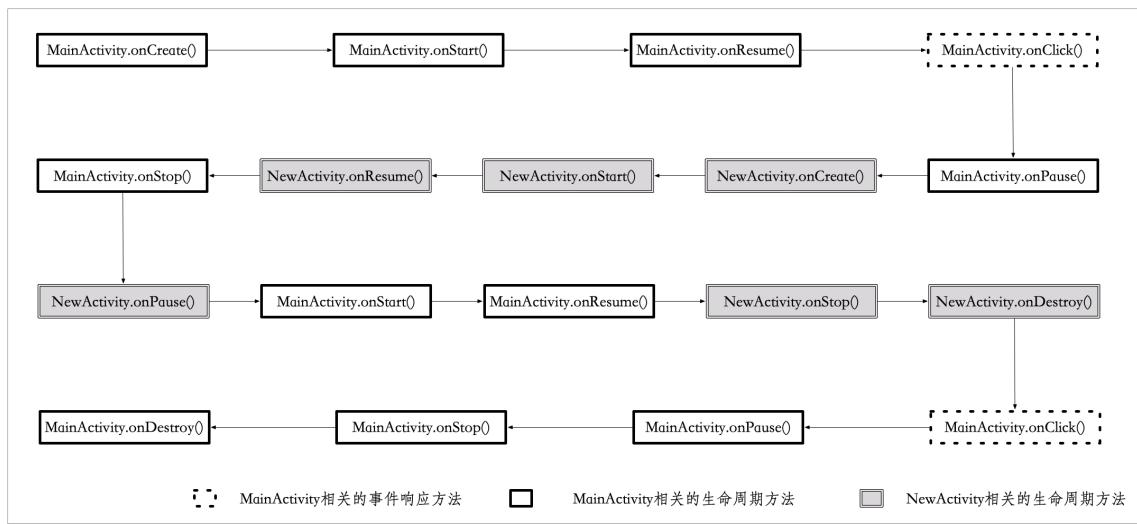


图 6.4: Activity 生命周期和事件回调的 RunDroid 调用图 (局部)

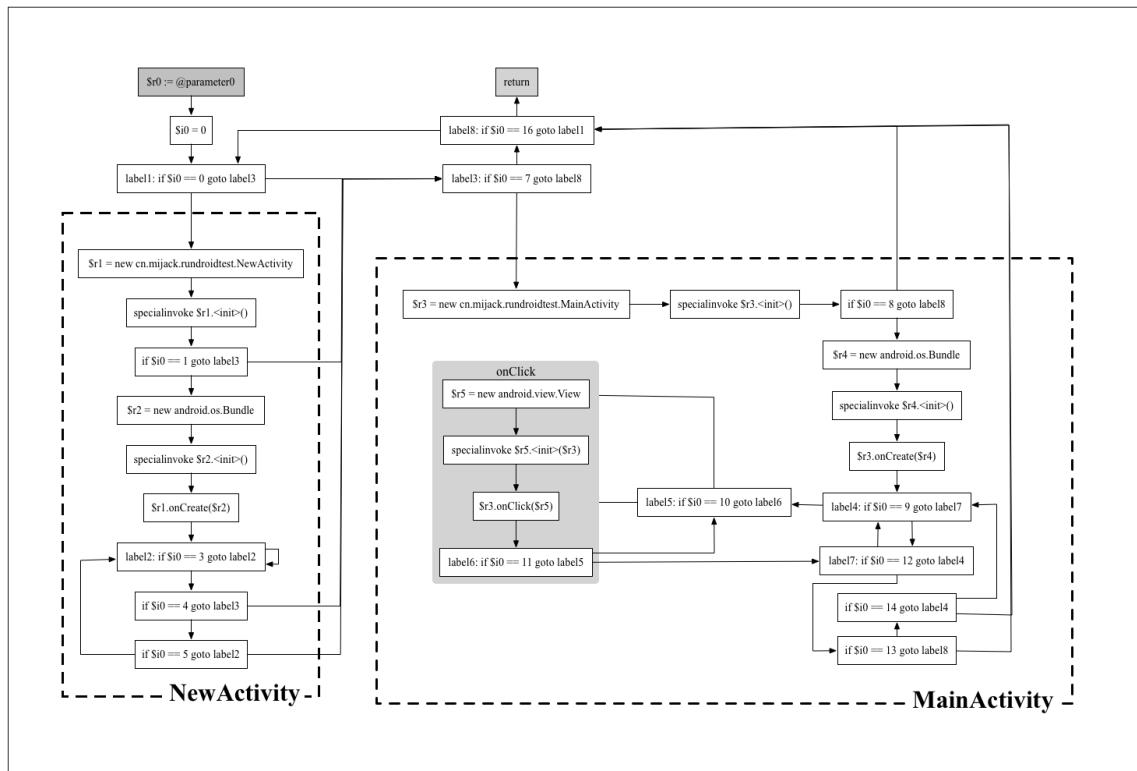


图 6.5: FlowDroid 生成的函数 dummyMainMethod() 的调用图

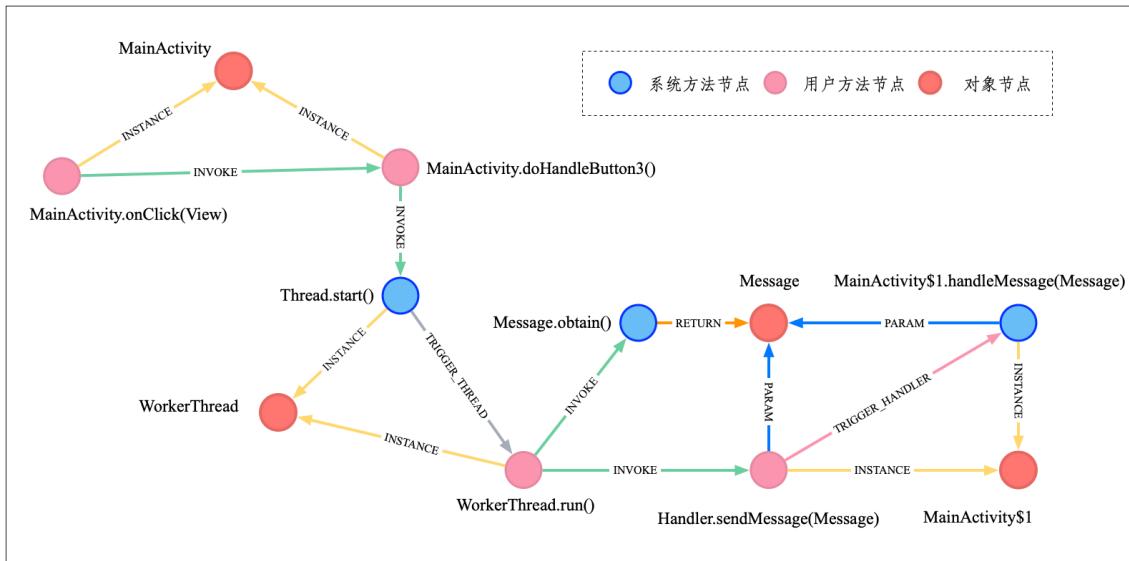


图 6.6: 多线程触发关系相关的 RunDroid 调用图 (局部)

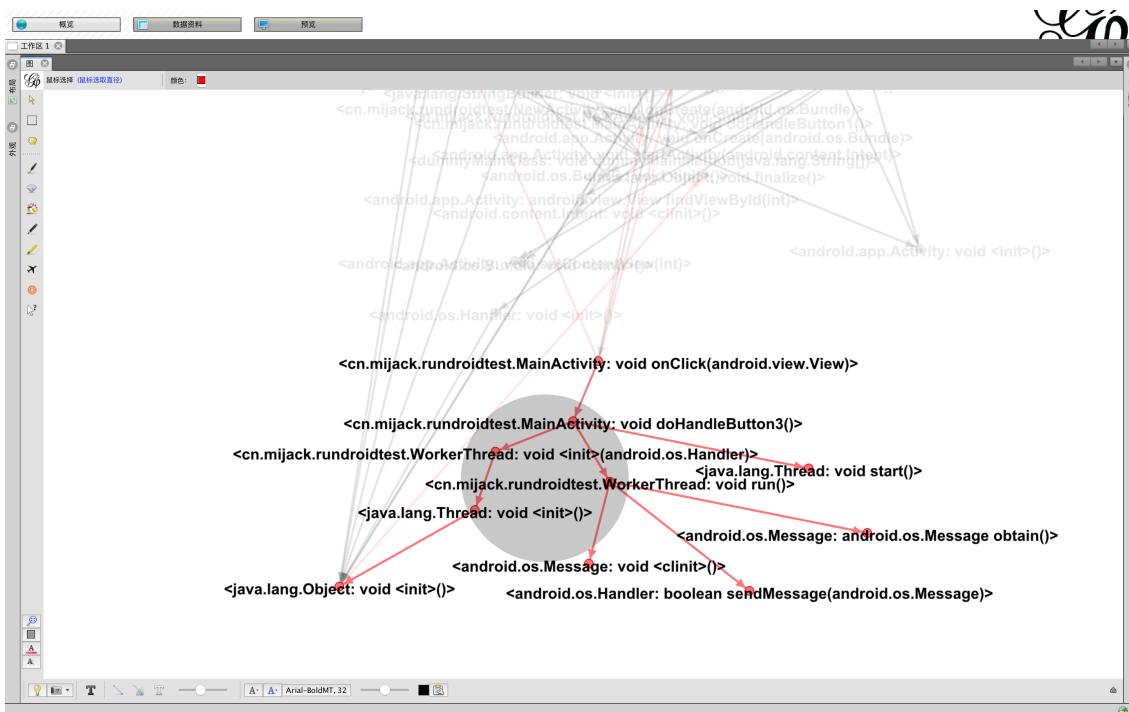
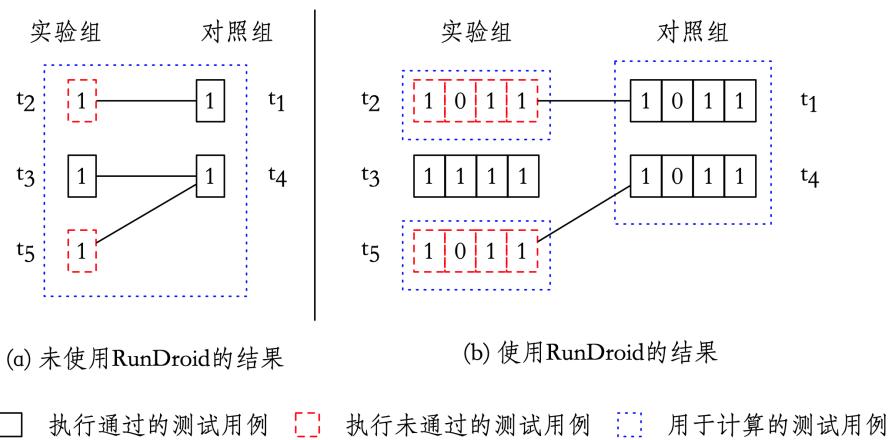


图 6.7: 多线程触发关系-FlowDroid 生成的调用图 (局部)

图 6.8: 使用 RunDroid 前后关于语句 *Line .13* 的测试用例匹配的对比

## 第七章 总结与展望

### 7.1 总结

本文提出并实现了一个可用于生成 Android 应用程序动态函数调用图的技术方案——RunDroid。RunDroid 生产的函数调用图，可以准确的反映应用程序的执行过程，取得了有意义的研究成果。本文的主要工作如下：

1) 拓展函数调用图：本文在传统的函数调用概念的基础上，提出了函数触发关系，用于描述两个方法之间的因果关系；并结合方法触发关系、方法对象等概念提出了拓展函数调用图的概念。

2) RunDroid 的设计与实现：RunDroid 利用源程序代码插桩和运行时方法拦截相结合的方式，获取应用方法执行信息，构建函数调用图；并在此基础上，利用方法和对象的关系补全到调用图中的方法间触发关系，展现运行过程中的 Android 特性行为。

3) 静动态工具的实验结果对比：本文将 RunDroid 产生的动态函数调用图和 FlowDroid 产生的静态函数调用图进行对比。相比 FlowDroid，RunDroid 产生的函数调用图能够体现应用程序的执行过程，表现函数间的调用关系和触发关系，准确地还原 Android 组件的生命周期。

4) 开源应用的统计实验：利用 RunDroid 构建开源 Android 应用的动态函数图，统计数据佐证了事件回调、Handler 等函数间触发关系在 Android 应用中的普遍性。

5) RunDroid 在错误定位领域的应用：相关实验结果实现，从 RunDroid 提供的函数关系信息可以反映出更多程序依赖信息，相比之前技术方案，方法间的因果关系模型更健全，实验的可靠性有所提升。

## 7.2 展望

虽然通过 RunDroid 还原得到的 Android 应用程序动态函数调用图，反映程序的运行时状态，但在实验过程中我们发现以下问题：

- 1) RunDroid 在捕获应用用户层方法时，采用的方案是源代码插桩方案。调用图构建的前置条件需要提供 Android 应用的源代码，因此，RunDroid 的运行对源代码高度依赖。
- 2) 在实验阶段，我们发现，当应用程序长时间运行时，应用程序会产生较多的日志。通常的，移动设备上的存储是有限的。因此，对于一些调用关系较为复杂的应用，RunDroid 的日志方案比较容易遇到日志存储的瓶颈。
- 3) RunDroid 中的运行时拦截器是基于 Xposed 框架实现的。Xposed 框架并不是适用于所有的 Android 手机，在一定程度给 RunDroid 的实验环境提出了额外的要求。

整体上，本文提出的 RunDroid 较为准确地还原出 Android 应用程序在运行过程的函数调用图。针对 RunDroid 实验中发现的不足，RunDroid 的后续工作可以从以下几个方面进行改进：利用字节码修改技术代替源代码修改方案以减少 RunDroid 运行过程中对源代码的依赖；引入基于 JVMTI 的调试环境，借助调试技术实现系统方法执行的拦截，摆脱对 Xposed 环境的依赖；通过静态分析技术确定运行过程的确定性路径，缩减待插桩的用户方法数量，进而减少运行时日志的产出量。

## 参考文献

- [1] Google Play Store: number of apps 2018 | Statistic[EB/OL]. 2018.  
[https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.](https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/)
- [2] VALLÉE-RAI R, CO P, GAGNON E, et al. Soot-a Java bytecode optimization framework[C] // Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research. 1999 : 13.
- [3] ARZT S, RASTHOFER S, FRITZ C, et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps[J]. Acm Silan Notices, 2014, 49(6) : 259–269.
- [4] WEI F, ROY S, OU X, et al. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps[C] // CCS '14 : Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. 2014.
- [5] LI L, BARTEL A, BISSYANDÉ T F, et al. IccTA: Detecting Inter-component Privacy Leaks in Android Apps[C] // ICSE '15 : Proceedings of the 37th International Conference on Software Engineering - Volume 1. 2015.
- [6] androguard/androguard: Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja !)[EB/OL]. 2018.  
[https://github.com/androguard/androguard.](https://github.com/androguard/androguard)
- [7] pxb1988/dex2jar: Tools to work with android .dex and java .class files[EB/OL]. 2018.  
[https://github.com/pzb1988/dex2jar.](https://github.com/pzb1988/dex2jar)
- [8] java-decompiler/jd-gui: A standalone Java Decompiler GUI[EB/OL]. 2018.  
[https://github.com/java-decompiler/jd-gui.](https://github.com/java-decompiler/jd-gui)
- [9] iBotPeaches/Apktool: A tool for reverse engineering Android apk files[EB/OL]. 2018.  
[https://github.com/iBotPeaches/Apktool.](https://github.com/iBotPeaches/Apktool)
- [10] maaaaz/androwarn: Yet another static code analyzer for malicious Android applications[EB/OL]. 2018.  
[https://github.com/maaaaz/androwarn.](https://github.com/maaaaz/androwarn)
- [11] CHUN S H V T B, MCDANIEL L C J J P, ENCK A S W, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones[J]. ACM Transactions on Computer Systems (TOCS), 2014.
- [12] pjplantz/droidbox: Dynamic analysis of Android apps[EB/OL]. 2018.  
[https://github.com/pjplantz/droidbox.](https://github.com/pjplantz/droidbox)
- [13] VAN DER VEEN V, BOS H, ROSSOW C. Dynamic analysis of android malware[J], 2013.
- [14] YAN L-K, YIN H. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis.[C] // USENIX security symposium. 2012 : 569–584.

- [15] ac-pm/Inspeckage: Android Package Inspector - dynamic analysis with api hooks, start unexported activities and more. (Xposed Module)[EB/OL]. 2018.  
<https://github.com/ac-pm/Inspeckage>.
- [16] Configure Apps with Over 64K Methods | Android Studio[EB/OL]. 2017.  
<https://developer.android.com/studio/build/multidex.html>.
- [17] AU K W Y, ZHOU Y F, HUANG Z, et al. Pscout: analyzing the android permission specification[C] // Proceedings of the 2012 ACM conference on Computer and communications security. 2012 : 217–228.
- [18] PANDITA R, XIAO X, YANG W, et al. WHYPER: Towards Automating Risk Assessment of Mobile Applications.[C] // USENIX Security Symposium : Vol 2013. 2013.
- [19] YANG W, XIAO X, ANDOW B, et al. Appcontext: Differentiating malicious and benign mobile app behaviors using context[C] // Proceedings of the 37th International Conference on Software Engineering-Volume 1. 2015 : 303–313.
- [20] YANG Z, YANG M, ZHANG Y, et al. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection[C] // Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013 : 1043–1054.
- [21] XIA M, GONG L, LYU Y, et al. Effective real-time android application auditing[C] // Security and Privacy (SP), 2015 IEEE Symposium on. 2015 : 899–914.
- [22] LINDORFER M, NEUGSCHWANDTNER M, PLATZER C. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis[C] // Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual : Vol 2. 2015 : 422–433.
- [23] SHAO Y, OTT J, JIA Y J, et al. The Misuse of Android Unix Domain Sockets and Security Implications[C] // ACM Sigsac Conference on Computer and Communications Security. 2016 : 80–91.
- [24] JIA Y J, CHEN Q A, LIN Y, et al. Open Doors for Bob and Mallory: Open Port Usage in Android Apps and Security Implications[C] // IEEE European Symposium on Security and Privacy. 2017 : 190–203.
- [25] BU W, XUE M, XU L, et al. When program analysis meets mobile security: an industrial study of misusing Android internet sockets[C] // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017 : 842–847.
- [26] AZIM T, NEAMTIU I. Targeted and depth-first exploration for systematic testing of android apps[C] // Acm Sigplan Notices : Vol 48. 2013 : 641–660.
- [27] YANG W, PRASAD M R, XIE T. A grey-box approach for automated GUI-model generation of mobile applications[C] // International Conference on Fundamental Approaches to Software Engineering. 2013 : 250–265.
- [28] SU T. FSMdroid: guided GUI testing of android apps[C] // Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on. 2016 : 689–691.
- [29] SU T, MENG G, CHEN Y, et al. Guided, Stochastic Model-based GUI Testing of Android Apps[C] // ESEC/FSE 2017: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017 : 245–256.
- [30] MIRZAEI H, HEYDARNOORI A. Exception fault localization in Android applications[C] // Mobile Software Engineering and Systems (MOBILESoft), 2015 2nd

- ACM International Conference on. 2015 : 156 – 157.
- [31] MACHADO P, CAMPOS J, ABREU R. MZoltar: automatic debugging of Android applications[C] // Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile. 2013 : 9 – 16.
- [32] TAN S H, DONG Z, GAO X, et al. Repairing Crashes in Android Apps[J], 2018.
- [33] GAO Q, ZHANG H, WANG J, et al. Fixing recurring crash bugs via analyzing q&a sites (T)[C] // Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on. 2015 : 307 – 318.
- [34] DENG L, OFFUTT J, AMMANN P, et al. Mutation Operators for Testing Android Apps[J/OL]. Inf. Softw. Technol., 2017, 81(C) : 154 – 168.  
<https://doi.org/10.1016/j.infsof.2016.04.012>.
- [35] DENG L, MIRZAEI N, AMMANN P, et al. Towards mutation analysis of android apps[C] // Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on. 2015 : 1 – 10.
- [36] LINARES-VÁSQUEZ M, BAVOTA G, TUFANO M, et al. Enabling mutation testing for android apps[C] // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017 : 233 – 244.
- [37] HAO S, LIU B, NATH S, et al. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps[C] // Proceedings of the 12th annual international conference on Mobile systems, applications, and services. 2014 : 204 – 217.
- [38] AMALFITANO D, FASOLINO A R, TRAMONTANA P, et al. MobiGUITAR: Automated model-based testing of mobile apps[J]. IEEE software, 2015, 32(5) : 53 – 59.
- [39] FAN L, SU T, CHEN S, et al. Large-scale Analysis of Framework-specific Exceptions in Android Apps[C] // ICSE '18 : Proceedings of the 40th International Conference on Software Engineering. 2018 : 408 – 419.
- [40] Handler | Android Developers[EB/OL]. 2018.  
<https://developer.android.com/reference/android/os/Handler>.
- [41] The AspectJ Project | The Eclipse Foundation[EB/OL]. 2018.  
<https://www.eclipse.org/aspectj/>.
- [42] COLLARD M, DECKER M, MALETIC J. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code[C] // Proceedings of 29th IEEE International Conference on Software Maintenance (ICSM' 13) Tool Demonstration Track, Eindhoven, The Netherlands. 2013 : 1 – 4.
- [43] Xposed Installer | Xposed Module Repository[EB/OL]. 2017.  
<http://repo.xposed.info/module/de.robv.android.xposed.installer>.
- [44] Neo4j, the world's leading graph database - Neo4j Graph Database[EB/OL]. 2017.  
<https://neo4j.com/>.
- [45] F-Droid - Free and Open Source Android App Repository[EB/OL]. 2018.  
<https://f-droid.org/>.
- [46] GOSLING J, JOY B, STEELE G. The Java language specification[M]. [S.l.] : Addison-Wesley Professional, 2000.
- [47] BAAH G K, PODGURSKI A, HARROLD M J. Causal inference for statistical fault localization[C] // Proceedings of the 19th international symposium on Software test-

- ing and analysis. 2010 : 73 – 84.
- [48] BAAH G K, PODGURSKI A, HARROLD M J. Mitigating the Confounding Effects of Program Dependences for Effective Fault Localization[C] // ESEC/FSE ’11 : Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. 2011.

## 攻读学位期间发表的学术论文

1. Chongbin Tang, Sen Chen, Lingling Fan, Lihua Xu, Yang Liu, Zhushou Tang, and Liang Dou, “A large-scale empirical study on industrial fake apps,” in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, IEEE Press, 2019: 183-192. (发表于 CCF 软件工程方向顶级会议 ICSE2019)