

密级：_____



中国科学院大学
University of Chinese Academy of Sciences

硕士学位论文

Android API 演化导致的兼容性问题研究

作者姓名：_____ 何冬杰 _____

指导教师：_____ 李 炼 研究员 _____

_____ 中国科学院计算技术研究所 _____

学位类别：_____ 工学硕士 _____

学科专业：_____ 计算机科学与技术 _____

培养单位：_____ 中国科学院计算技术研究所 _____

2018 年 5 月

A Study on Android API evolution-induced Compatibility Issues

**A thesis submitted to
University of Chinese Academy of Sciences
in partial fulfillment of the requirement
for the degree of
Master of Science
in Computer Science and Technology**

By

Dongjie He

Supervisor: Professor Lian Li

Institute of Computing Technology

Chinese Academy of Sciences

May 2018

中国科学院大学
研究生学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。

作者签名：

日 期：

中国科学院大学
学位论文授权使用声明

本人完全了解并同意遵守中国科学院有关保存和使用学位论文的规定，即中国科学院有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：

导师签名：

日 期：

日 期：

摘要

Android 版本的频繁更新是导致 Android 生态碎片化问题的一个重要原因。为了使 Android 应用可以在多个 Android 版本上使用，应用程序开发者不得不为其开发的应用进行兼容处理。然而这不仅增加了开发的难度还给测试带来了许多麻烦。当前，这种因 Android 版本演化导致的兼容性问题虽然很严重但相关研究工作却很少。

本文首先围绕 API 演化导致的兼容性问题做了一些经验性的研究，我们发现 Android 应用中兼容处理是普遍存在的，91.84% 的应用做了 API 演化导致的兼容处理。另外，我们还发现导致这种兼容处理普遍的原因有 Android 演化导致 API 剧烈变化，Android 支持库对这种变化支持程度很低，应用为了在新版本提供系统添加的新功能因此需要做兼容处理。最后，通过对 10 个应用以及 1 个 Android 支持库源码的手动分析，我们发现大多数兼容修复是比较简单的，较少有使用复杂的兼容模式。

在经验性研究的基础上我们提出了一种基于数据流分析的静态检测 Android 应用中 API 不兼容使用的方法。基于该方法我们开发了一个原型工具 IctApiFinder，据我们所知，这是第一个真正意义上的静态检测 Android API 不兼容使用的工具。我们的实验表明该工具简单易用，能有效减少误报率并且能够发现真实应用中存在的 API 不兼容使用缺陷。目前我们报告的潜在缺陷中已经有五个收到了应用开发者的确认，其中三个被认为很严重并已经得到修复。

本文最后开发了一个自动识别 Android API 演化的工具，该工具能够识别出两个不同 Android 版本间的大多数 API 变化，包括添加，删除和替换等。基于该工具，我们能够为一些 API 的不兼容使用给出对应的修复建议。

关键词：Android 兼容性，API 演化，缺陷检测，程序分析

Abstract

One of the most important reason for Android fragmentation is its frequent release. In order to make their Apps to work on multiple Android versions, developers have to deal with many compatibility issues which not only increase the difficulty of development but also bring lots of trouble to the testing. Currently, Few studies are relevant to this kind of evolution-induced compatibility issues, though they are becoming more and more important.

In this paper, we firstly did some empirical works on Android evolution-induced compatibility issues. Our study found that the process of compatibility issues is very common in Apps. 91.84% Apps deal with these issues in their code. The reasons in depth may be the dramatic API changes caused by Android evolution, poor support from Android Support Library and the exploit to the new features provided by newer versions of Android. By manually analyzing ten Apps and one support library's source code, we also found that fixing compatibility issues in Apps is usually very simple and few use complicated fixing patterns.

Based on our empirical study, we have proposed a new method in detecting incompatible APIs use in Android Apps. We also develop a prototype tool called IctApiFinder. To the best of our knowledge, this is the first work to statically detect incompatible APIs use in Android Apps. Our experiments show that our tool can effectively reduce false positives and work efficiently by comparing with Android Lint and dynamic detecting tools respectively. It can also find real bugs from released Android Apps downloaded from a free application market named F-Droid. Currently, more than five Apps with incompatible API use have been confirmed by the developers and three of them have already been fixed.

In the last part of this paper, we developed a tool called Focus which can automatically identify the API changes between two different Android versions. Based on this tool, we can give repairing suggestions when IctApiFinder report incompatible API use bugs to developers.

Key Words: Android Compatibility, API Evolution, Defect detection, Program Analysis

| | |
|------------------------------------------|----|
| 第 1 章 引言 | 1 |
| 1.1 研究背景 | 1 |
| 1.2 研究现状 | 4 |
| 1.3 研究内容和意义 | 4 |
| 1.4 文章组织 | 5 |
| 第 2 章 背景知识 | 7 |
| 2.1 Android 基础知识 | 7 |
| 2.1.1 Android 系统架构 | 7 |
| 2.1.2 Android 应用程序 | 8 |
| 2.1.3 SDK 版本和兼容版本声明 | 10 |
| 2.1.4 Android Lint | 11 |
| 2.2 静态程序分析 | 12 |
| 2.2.1 控制流分析 | 12 |
| 2.2.2 数据流分析 | 13 |
| 2.2.3 别名分析 | 14 |
| 2.3 其他基本概念 | 15 |
| 第 3 章 经验性研究：Android API 演化导致的兼容性问题 | 17 |
| 3.1 研究问题 | 17 |
| 3.2 研究方法 | 18 |
| 3.2.1 数据选取 | 18 |
| 3.2.2 数据分析方法 | 20 |
| 3.3 研究结果 | 21 |
| 3.3.1 RQ1: API 演化导致的兼容问题严不严重? | 21 |
| 3.3.2 RQ2: 导致应用处理这类兼容问题的深层原因是什么? | 23 |
| 3.3.3 RQ3: 应用如何修复 API 演化导致的兼容问题? | 26 |
| 3.4 本章小结 | 29 |

| | |
|------------------------------------------------|----|
| 第 4 章 IctApiFinder: Android API 的不兼容引用检测 | 31 |
| 4.1 概述 | 31 |
| 4.2 检测方法 | 33 |
| 4.2.1 建立数据流模型 | 33 |
| 4.2.2 预分析 | 36 |
| 4.2.3 IFDS 求解 | 36 |
| 4.2.4 兼容性检测与报告 | 38 |
| 4.3 工具实现 | 39 |
| 4.4 工具评价 | 40 |
| 4.5 本章小结 | 47 |
| 第 5 章 Android API 版本演化挖掘及 API 不兼容使用修复建议 .. | 49 |
| 5.1 概述 | 49 |
| 5.2 Android API 变化识别方法 | 49 |
| 5.2.1 文本相似度 | 50 |
| 5.2.2 源码注释提取 | 53 |
| 5.2.3 类层次关系图 | 54 |
| 5.2.4 调用依赖分析 | 55 |
| 5.3 识别效果评价 | 56 |
| 5.4 讨论: 研究中遇到的困难和挑战 | 58 |
| 5.5 API 不兼容使用修复建议 | 59 |
| 5.6 相关工作 | 60 |
| 5.7 本章小结 | 61 |
| 第 6 章 结束语 | 63 |
| 6.1 主要成果 | 63 |
| 6.2 未来工作 | 64 |
| 参考文献 | 65 |
| 附录 Android 开发中最常使用 API | 69 |
| 附录 研究技术报告 | 71 |
| 致谢 | 75 |

图目录

| | | |
|-------|----------------------------------------------------------|----|
| 图 1.1 | Google Play 商店中可下载应用数量增长迅速 | 2 |
| 图 2.1 | Android 软件栈 | 9 |
| 图 2.2 | APK 文件结构 | 10 |
| 图 2.3 | Lint 工具工作流程示意图 | 12 |
| 图 2.4 | 代码, 指令 CFG 和基本块 CFG 示例 | 13 |
| 图 2.5 | 在 AndroidManifest.xml 中定义 MainActivity 组件为 程序入口点..... | 14 |
| 图 3.1 | 从 AndroZoo 下载的应用 targetSdkVersion 分布 | 18 |
| 图 3.2 | Android 应用中 VERSION.SDK_INT 使用数量箱线图..... | 21 |
| 图 3.3 | API 在 Android 应用中被引用情况 | 22 |
| 图 3.4 | 相邻版本 Android API 差异箱线图 | 23 |
| 图 3.5 | APK 中引用下一版本将被删除的 API 次数分布图 | 25 |
| 图 3.6 | 在 if 判断里直接做兼容处理 | 26 |
| 图 3.7 | 在 if 判断里间接做兼容处理 | 26 |
| 图 3.8 | 使用问号表达式进行兼容处理 | 27 |
| 图 3.9 | 简化版 ViewCompat 类对 View 的兼容支持实现 | 28 |
| 图 4.1 | Android API 不兼容引用示例 | 31 |
| 图 4.2 | Android Lint 工具分析结果..... | 32 |
| 图 4.3 | IFDS 算法检测不兼容示例 | 37 |
| 图 4.4 | IctApiFinder 报告结果示例 | 38 |
| 图 4.5 | IctApiFinder Architecture | 39 |
| 图 4.6 | F-Droid 市场上 Android 应用的不兼容引用缺陷数量分布 | 41 |

| | | |
|--------|-----------------------------------------------------------------|----|
| 图 4.7 | 触发 <code>jonas.tool.saveForOffline</code> 中不兼容 API 使用缺陷 . | 44 |
| 图 4.8 | <code>com.xargsgrep.portknocker</code> 应用开发者确认缺陷报告 | 45 |
| 图 4.9 | Batphone 应用开发者对缺陷进行确认和修复..... | 45 |
| 图 4.10 | Calendula 应用开发者采纳我们的建议简化应用代码 | 48 |
| 图 5.1 | 纯文本相似度权重变化..... | 51 |
| 图 5.2 | 标识符分词状态机 | 52 |
| 图 5.3 | 方法描述所占权重随方法参数多少的变化曲线 | 53 |
| 图 5.4 | 通过 Class Hierarchy 识别废弃类..... | 55 |
| 图 5.5 | 通过调用关系变化推断 <code>method1</code> 被 <code>method3</code> 替换 | 56 |
| 图 5.6 | 兼容修复建议示例 | 60 |
| 图 B.1 | 自动下载并编译 AOSP 项目 | 72 |
| 图 B.2 | 使用 Doop 提取 API 信息 | 72 |
| 图 B.3 | Gradle 依赖管理 | 74 |

表目录

| | | |
|-------|---------------------------------------------|----|
| 表 1.1 | 2017 年一季度智能手机市场份额 | 1 |
| 表 1.2 | Android 版本, 发布时间及市场分布 | 3 |
| 表 3.1 | 从 AndroZoo 中下载和选择应用 | 19 |
| 表 3.2 | 选用的 Android 版本, 市场分布及各类型 API 数量 | 20 |
| 表 3.3 | F-Droid 中适配次数最多的 10 个应用 | 20 |
| 表 3.4 | Android 支持库对 API 演化支持情况 | 24 |
| 表 3.5 | Android 应用源码中 VERSION.SDK_INT 被引用情况统计 | 29 |
| 表 4.1 | Android Lint 和 IctApiFinder 报告缺陷数量比较 | 42 |
| 表 4.2 | 人工分析 IctApiFinder 报告的 API 不兼容使用缺陷 | 43 |
| 表 4.3 | 国内主要移动测试平台提供服务基本概况 | 46 |
| 表 5.1 | 用于提取源码注释中含有的演化信息的正则表达式 | 54 |
| 表 5.2 | Focus 工具识别结果展示 | 57 |
| 表 5.3 | 人工分析替换关系计算工具识别正确率和准确率 | 57 |
| 表 5.4 | 各版本 SDK 中顶层 API 比率 | 58 |

第1章 引言

1.1 研究背景

近年来，伴随着移动互联网的快速发展，具有移动操作系统的智能手机亦以惊人的速度在全球范围内快速普及。利用智能手机，人们可以便利地进行收发电子邮件，与亲朋好友语音或视频交流通信，网上购物等等活动。它不仅便利了人们的生活也丰富了人们的生活，因此智能手机受到用户的广泛欢迎。

智能手机根据其搭载的操作系统的不同，可以分为 Android、iOS、Windows Phone 以及其他智能手机。根据 IDC^[1] 公布的最新统计数据 (见表1.1)，在过去一年多的时间里，Android 手机市场占有率持续高于 80%，深受广大用户的喜爱。

| Period | Android | iOS | Windows Phone | Others |
|--------|---------|-------|---------------|--------|
| 2016Q1 | 83.4% | 15.4% | 0.8% | 0.4% |
| 2016Q2 | 87.6% | 11.7% | 0.4% | 0.3% |
| 2016Q3 | 86.8% | 12.5% | 0.3% | 0.4% |
| 2016Q4 | 81.4% | 18.2% | 0.2% | 0.2% |
| 2017Q1 | 85.0% | 14.7% | 0.1% | 0.1% |

表 1.1 2017 年一季度智能手机市场份额

Android 手机深受欢迎的原因和其开放性有很大关系。Android 是一种基于 Linux 的自由及开放源代码的操作系统，一方面它允许设备厂商定制系统，于是市场涌现了一批比如华为、小米、三星等优秀手机设备，带来 Android 设备百花齐放的局面；另一方面它也吸引了更多 Android 应用开发学习者和开发者，他们为 Android 系统开发了大量生动有趣的应用程序，进而促进了 Android 生态的繁荣稳定。图1.1是 Statista 网站^① 统计的自 2009 年 12 月到 2017 年 9 月份 Google Play 商店上能够下载的 Android 应用数量分布直方图，从该图中我们可以清晰地看到 Android 应用数量增长非常迅速，平均每个月会有 3.5 万个新应用出现。

① <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store>

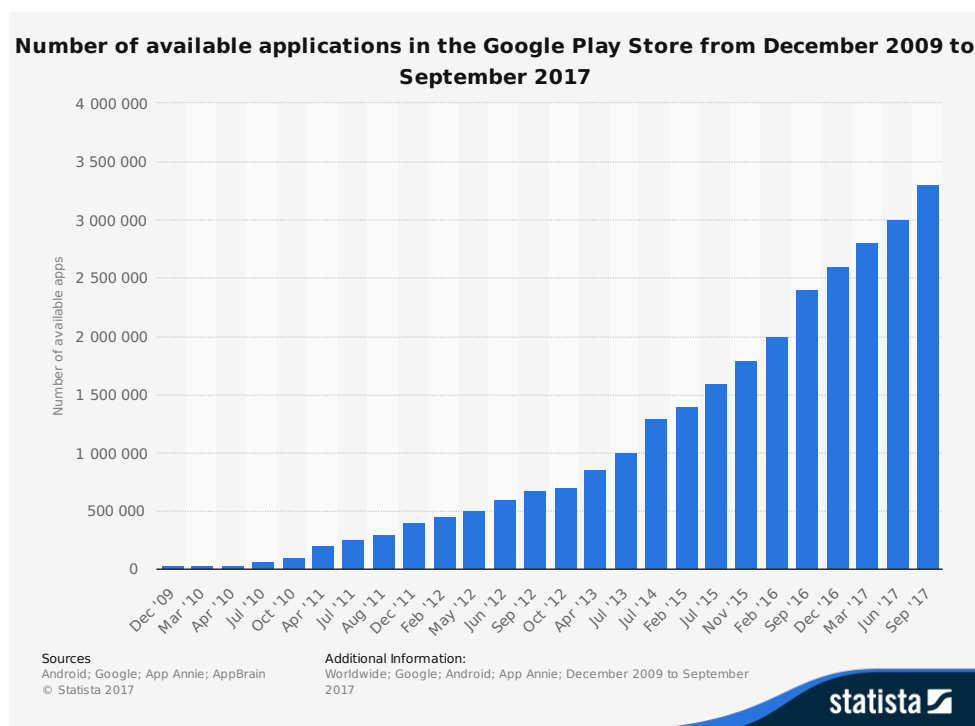


图 1.1 Google Play 商店中可下载应用数量增长迅速

开放特性使 Android 受到广泛欢迎的同时也引发了不少问题，其中兼容性问题日益严重。在 Android 生态下，数量众多的开发者和用户群体促使 Android 功能性开发和维护性开发异常活跃。根据维基百科“Android 版本历史”页面^①记录的结果（详见表 1.2），Android 系统版本更新迭代速度非常快，平均每 3 个月就有一个新版本的 Android 系统发布。频繁的版本更新导致 Android 系统不够稳定，于是经常会有用户抱怨某款应用在特定设备上不能正常使用，造成不友好的用户体验^[2]。另一方面，碎片化了的 Android 生态下存在大量不同设备型号、不同尺寸大小、不同系统版本的 Android 设备，这无疑给应用程序的开发、测试和维护带来了挑战。

从应用程序的开发角度来说，应用开发者开发一款应用时需要从设备型号、版本系统、屏幕尺寸等多个角度考虑应用的兼容性问题，这导致应用程序代码被分支判断语句划分成许多块，而某些代码块只能在特定运行环境下才能得到执行，这就是所谓的代码碎片化问题。根据开发经验，程序中分支语句会增加程序的复杂度从而增加程序开发的难度。

从应用程序的测试角度来说，测试人员需要在不同设备、不同版本的 Android 操作系统下对应用程序进行全面的测试，尽可能覆盖所有执行路径。比

^① https://en.wikipedia.org/wiki/Android_version_history

| Code name | Version number | Initial release date | API level | Share |
|--------------------|----------------|----------------------|-----------|-------|
| No codename | 1.0 | September 23,2008 | 1 | * |
| Petit Four | 1.1 | February 9,2009 | 2 | * |
| Cupcake | 1.5 | April 27, 2009 | 3 | * |
| Donut | 1.6 | September 15, 2009 | 4 | * |
| Eclair | 2.0 -2.1 | October 26, 2009 | 5 -7 | * |
| Froyo | 2.2 -2.2.3 | May 20, 2010 | 8 | * |
| Gingerbread | 2.3 -2.3.7 | December 6, 2010 | 9 -10 | 0.3% |
| Honeycomb | 3.0 -3.2.6 | February 22, 2011 | 11 -13 | * |
| Ice Cream Sandwich | 4.0 -4.0.4 | October 18, 2011 | 14 -15 | 0.4% |
| Jelly Bean | 4.1 -4.3.1 | July 9, 2012 | 16 -18 | 5.0% |
| KitKat | 4.4 -4.4.4 | October 31, 2013 | 19 -20 | 12.0% |
| Lollipop | 5.0 -5.1.1 | November 12, 2014 | 21 -22 | 24.6% |
| Marshmallow | 6.0 -6.0.1 | October 5, 2015 | 23 | 28.1% |
| Nougat | 7.0 -7.1.2 | August 22, 2016 | 24 -25 | 28.5% |
| Oreo | 8.0 -8.1 | August 21, 2017 | 26 -27 | 1.1% |

* 分布数据是 Android 官网 2018 年 2 月 5 日发布的数据，* 表示占比少于 0.1%。

表 1.2 Android 版本，发布时间及市场分布

如，通常 Android 应用兼容多个 *API Level*，对其进行测试时，需要在多个不同 Level 的系统上进行测试，以判断应用是否存在后向兼容性问题。这个过程也是非常耗费时间的，测试不全面就会导致应用程序在某些版本上出现兼容问题，在用户体验上获差评，进而流失用户。能否开发一个工具快速帮助测试人员定位应用中潜在的不兼容问题是个非常值得研究的问题。

从应用程序的维护角度来说，新版本的 Android 系统往往会增强、增添、修改、删除或废弃一些功能特性。这些功能特性的变化往往会导致一些已有的 Android 应用程序在新版本的 Android 系统上行为出现差异，即出现了前向兼容性问题。有时候出于安全性方面的考虑或为了使应用程序支持新版本引入的某些新功能，应用程序需要修改 `targetSdkVersion`，升级适配到新版本的 Android 系统。当前，该过程需要应用开发者们人工阅读新版本 Android 系统的变化，在此基础上对自己的应用进行人工适配工作。这个过程也是非常痛苦，尤其是当改动特别多的时候。能否对这个过程进行自动化以及怎样对这个过程进行自动化都是非常值得关心和研究的问题。

1.2 研究现状

Israel J. Mojica^{[3][4]} 和 Mark D. Syer^[5] 等人的研究表明 Android 应用开发严重依赖 Android 应用程序框架中提供的 API。通过继承或直接调用的方式复用这些 API 可以迅速完成程序功能实现,有效减小编码量。然而,伴随着 Android 版本的快速更新,Android 应用程序框架(application framework)也跟着快速变化,从表1.2可以看到,当前 *API Level* 已经增长到 27。Tyler McDonnell^[6] 等人的研究显示 Android 系统平均每个月有 115 个 API 被更新,且快速演化的 API 恰恰是应用程序使用较多的 API。这些研究结果意味着 Android 应用程序可能存在着严重的兼容性问题。

Android 应用的兼容性问题包括设备相关兼容性问题(Device-specific issues)和设备无关兼容性问题(Non-device-specific issue)。设备相关兼容性问题只会在特定手机设备上才会触发,而设备无关设备兼容性问题会在搭载特定 Android 版本的手机上被触发,和具体的手机设备型号无关。Lili Wei 等人的工作^[7] 显示,设备相关兼容性问题占比 59%,出现该类兼容性问题的原因主要包括有问题的硬件驱动实现(29%),设备厂商对 Android 系统的定制如功能修改,功能增强和功能删除等(19%)以及特异的硬件组成(11%)。而设备无关兼容性问题占比 41% 主要是由 Android 系统 API 的演化(35%)及原先 Android 系统缺陷(6%)造成的。本文主要研究 Android 应用的设备无关兼容性问题,这类问题主要和 Android API 的演化相关。

当前,从论文的发表情况来看,关于 Android 兼容性方面的研究还不多。虽然关于 Android API 演化的话题在网络论坛上的讨论很多^[8],但关于 API 演化对应用程序兼容性究竟有怎样的影响、影响的范围有多大以及如何帮助解决或帮助减轻 API 演化在应用程序兼容性方面造成的影响等问题目前还没看到相关研究。因此,对这些问题展开研究具有非常明显的实际意义。

1.3 研究内容和意义

围绕 Android API 演化导致的兼容性问题,我们开展了一系列经验性研究。我们研究了 Android API 在 Android 应用程序中的使用情况、Android 应用中兼容性处理情况以及 Android 支持库对 Android API 演化的支持情况等等。另外,我们还手动分析了一些 Android 应用的源代码和 Android 支持库的源码,试图

发现应用程序在处理兼容性时存在哪些模式或规律。

在此基础之上,我们开发了一些应用工具。第一个应用工具是 `IctApiFinder`, 该工具使用静态程序分析方法分析 `Android` 应用中引用的 `API` 是否存在不兼容引用问题, 并报告出潜在不兼容性引用。第二个应用工具叫 `Focus`, 通过使用文本相似度、注释自然语言处理、调用依赖分析等方法, 该工具可以自动挖掘出两个不同版本的 `Android SDK` 中 `API` 的变化关系, 比如替换、删除、新添等。结合第一个工具, 我们将能对 `Android` 应用中发现的不兼容性问题给出一些简单的修复意见。

本工作围绕 `Android API` 演化导致的兼容性问题, 从多个角度进行经验性研究分析, 填补了该领域的研究空白。此外, 我们构建工具检测 `Android` 应用中因 `API` 变动导致的不兼容问题, 并给出修改建议。我们构建的检测和推荐工具可以帮助应用测试人员快速定位兼容性问题, 帮助开发者更快速地对应用进行兼容处理, 具有实际意义。

1.4 文章组织

本文共有六个章节。内容组织结构如下: 第一章介绍论文的研究背景、现状以及研究内容和意义。第二章介绍本文用到的一些基础知识, 主要包括 `Android` 系统、静态程序分析以及兼容性等。我们在第三章将详细阐述对 `Android API` 演化导致的兼容性问题的经验性研究。在第四章, 我们会介绍一种静态检测 `Android API` 不兼容使用的方法。第五章中我们开发工具对 `Android API` 版本演化进行自动挖掘, 并对 `API` 引用出现的兼容性问题给出修复建议。最后, 我们在第六章总结本文工作并对下一步工作进行展望。

第 2 章 背景知识

本章将介绍一些背景知识，以便后文中使用。主要包括 Android 系统架构和应用程序结构，静态程序分析中常用的技术和方法以及文中用到的一些基本概念。

2.1 Android 基础知识

Android 系统是一个基于 Linux 的开源操作系统，它由 Google 公司和开放手机联盟领导和开发。Android 系统广泛用于智能手机和平板电脑等移动设备上，深受广大用户的喜欢。下面从 Android 的系统架构、应用程序组成等方面简单介绍一下 Android 系统。

2.1.1 Android 系统架构

Android 系统并不单单是一个操作系统，它是基于 Linux 操作系统开发的一个具有完整功能的软件栈（见图2.1），从栈底到栈顶分别包括 Linux 内核、硬件抽象层、Android 运行时、本地 C/C++ 库、应用程序 API 框架以及系统原生应用^[9]。

Linux 内核层主要完成操作系统所具有的功能，该层包含许多驱动程序，系统通过这些驱动程序来与移动设备上多样的硬件设备进行交互。

硬件抽象层对底层的硬件设备能力进行抽象，为上层的 Java API 应用框架提供统一标准的接口，以方便 Android 设备厂商对 Android 的定制化。比如不同 Android 设备的相机硬件不同，对应的驱动程序也不相同，但通过硬件抽象，Android 应用程序便可以通过调用同样的 API 使用底层的相机驱动程序。

Android 运行时解析 Android 应用程序里 DEX 文件中的字节码并执行。每一个 Android 应用在其自己的进程内执行并拥有一个 Android 运行时实例。在 Android 5.0 以前 Android 运行时为 Dalvik，在该运行时下，应用每次运行都需要将字节码通过即时编译器转换为机器码，严重拖慢应用的运行效率。Android 5.0 以后，Android 运行时换成 ART，在该运行时下，应用在第一次安装时字节码会预先编译成机器码，使其成为真正的本地应用，这样应用启动和执行速度就会快很多。

本地 C/C++ 库包含 Android 系统使用的许多重要的库文件, 比如 SQLite 数据库, OpenGL ES 和 Libc 等, 这些库被系统中的许多组建和服务依赖如 ART、HAL 等。一些 Java API 也提供了对部分本地库的访问, 例如使用应用框架中的 OpenGL API 可以访问 OpenGL ES 库中功能。有时候开发 Android 应用需要使用 C 或 C++ 也可以使用 Android NDK 达到对一些本地库直接访问使用的能力。

应用程序 API 框架层使用 Java 语言编写, 它提供了 Android 系统全部可用特性的 API 集合。应用程序开发者通过使用应用框架提供的 API 复用系统组件和服务可以快速地开发自己的 Android 应用。值得注意的是, 应用程序 API 框架包含两类 API, 一类是公开给应用程序开发者使用的 API 称为 public API, 这类 API 在 Android SDK 包即 `android.jar` 中; 还有一类 API 是内部 API 或者称为隐式 (hidden) API, 这一类 API 在应用程序 API 框架包即 `framework.jar` 中存在, 但没有放到 `android.jar` 中来。Li Li^[10] 等人研究表明内部 API 多是一些功能还不够稳定在以后版本的演化中可能被删除的 API, 很少一部分内部 API 最后变为公开 (public) API。

系统应用层包含一些如邮件、SMS 通讯、日历、浏览器、通讯录、键盘等核心应用。除系统设置这个应用外, 其他应用和用户从应用商店下载安装的应用没有本质区别, 但系统应用用户一般无权限删除。

2.1.2 Android 应用程序

Android 应用程序^[11] 通常是用 Java 语言编写的具有图形用户接口 (GUI) 的程序, 它被编译为 Dalvik 字节码, 并在运行时虚拟机 (过去是 Dalvik 虚拟机, 现在是 ART) 上运行。Android 应用由四种组件组成: 活动 (Activity)、服务 (Service)、广播接收器 (Broadcast Receiver) 和内容提供者 (Content Provider)。活动组件提供可以和用户交互的界面, 服务组件通常不和用户进行交互, 主要用来在后台长时间执行计算任务, 广播接收器组件用于监听广播并对广播消息作出响应, 内容提供者组件的作用是将程序的内部数据和外部进行共享, 为数据提供外部访问接口, 被访问的数据通常以数据库形式存在。应用中组件之间通信需要使用 Intent, Intent 是信息载体, 用它可以去请求组件做相应操作。跨应用的组件间通信通常使用 Binder。

Android 应用程序以 APK(Android Package) 文件的形式发布。APK 文件实

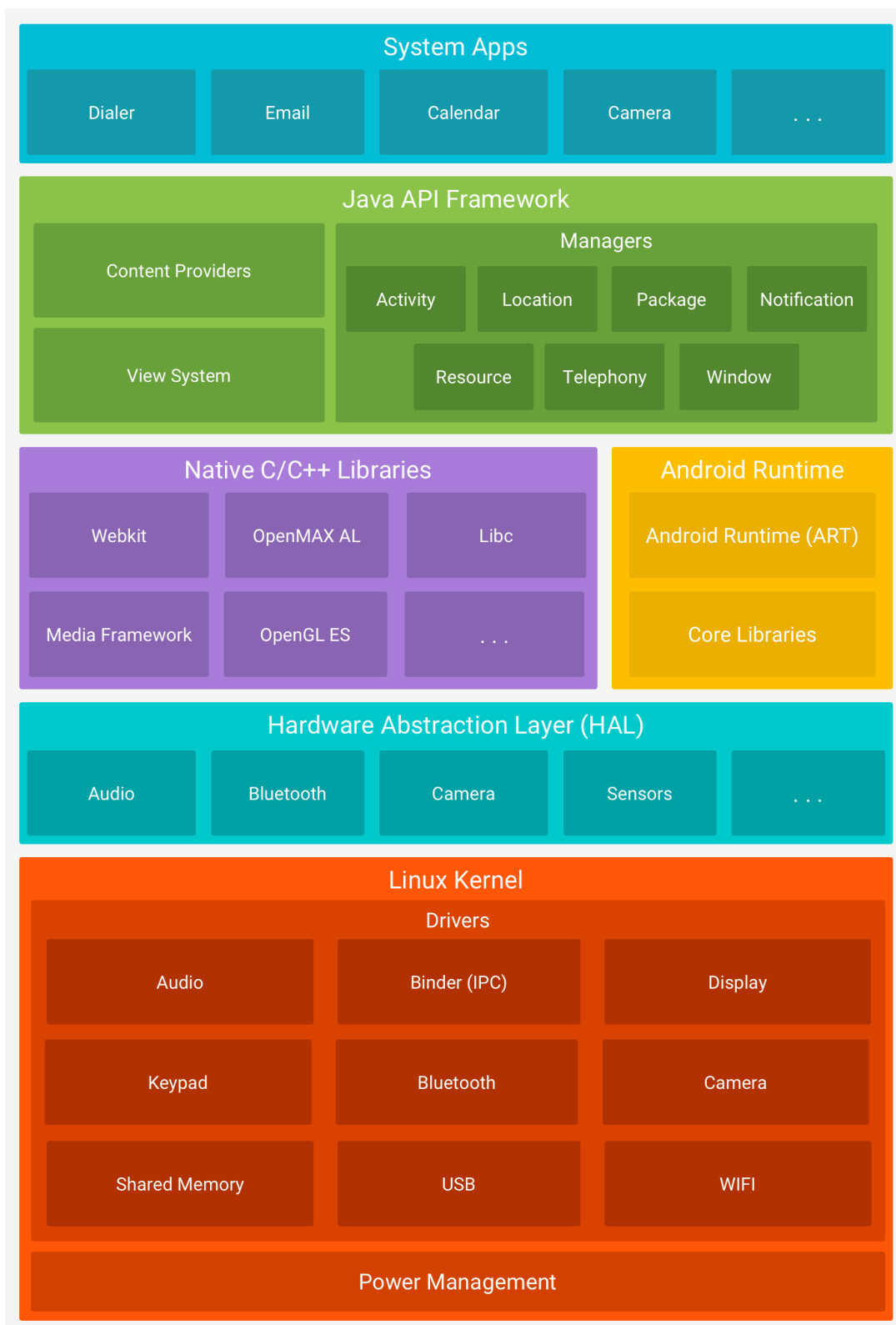


图 2.1 Android 软件栈, 该图从 Android 官网获得^②

② <https://developer.android.com/guide/platform/index.html>

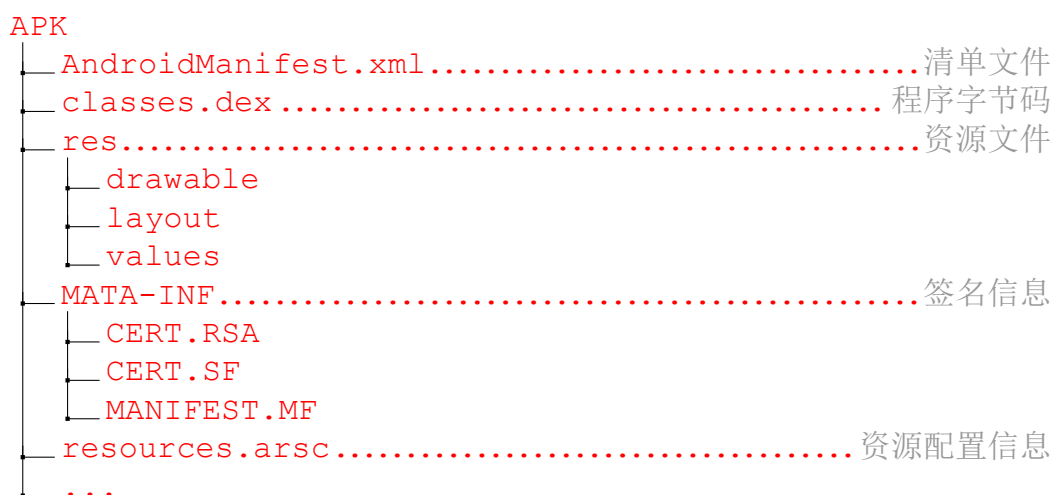


图 2.2 APK 文件结构

实际上是一个 Zip 格式的压缩包，里边包含了程序运行时需要的所有代码，数据和资源（例如，图片和视频）等。其文件结构如图2.2所示，其中：META-INF 文件夹存放应用程序的签名信息，用来确保 APK 包的完整性；res 文件夹存储资源文件如图片，界面布局文件以及一些值信息等；resources.arsc 文件，用来描述那些具有 ID 值的资源的配置信息，它的内容就相当于是一个资源索引表；classes.dex 是 APK 中最重要的一个文件，它包含了应用程序的所有字节码文件，需要注意的是随着 multidex 技术的引入，一个 APK 文件中允许有多个以 dex 为后缀的文件。

APK 中另一个非常重要的文件是 AndroidManifest.xml 文件。该文件声明了应用程序的包名，权限，使用的组件以及兼容的版本等信息。关于兼容版本的声明介绍我们将在2.1.3节详细介绍。

2.1.3 SDK 版本和兼容版本声明

应用程序框架为应用开发者提供可以重用的应用程序接口（API），这些 API 的集合被称为 Android 的软件开发工具包（SDK）。SDK 和应用程序框架均由相同源代码编译得到，分别对应 android.jar 和 framework.jar 库。这两者的区别首先是前者是开发时库而后者是运行时库，应用开发者在开发 Android 应用时使用 android.jar 库中提供的 API 进行编译链接，而当应用安装到设备上之后，应用引用的实际 API 是 framework.jar 中的 API，android.jar 中包含的 class 文件不在 Android 应用的 dex 文件中出现。另一个重要区别是，SDK 提供的 API 是应用程序框架提供的 API 的一个真子集。在

应用程序框架源码中使用 `@hide` 标记或者位于 `com.android.internal` 包中的 API 属于内部 API, 通常因不稳定或者存在缺陷等原因而不包含于 SDK 中^[10]。

伴随着 Android 版本的快速更新, 应用程序框架 (application framework) 也跟着快速变化。不同版本的应用程序框架提供的 SDK 会有较大差别, 为了区分不同版本的 SDK, Android 官方使用一个整数编号即 *API Level* 唯一表示一个 SDK 版本。需要指出的是, 同一个 *API Level* 对应的程序框架提供的 API 集合是相同的, 属于一个 SDK 版本, 因此可能多个 Android 版本对应一个 SDK 版本。从表1.2可以看到, 当前 *API Level* 已经增长到 27, 也就是共有 27 个 SDK 版本。

Android 应用通常需要能够在支持不同 SDK 版本的 Android 系统上运行, 即需要兼容多个 Android 版本。通过在其清单文件即 `AndroidManifest.xml` 文件中使用 `<uses-sdk>` 元素可以非常便利地声明其支持的最小 SDK 版本 (`minSdkVersion`), 最大 SDK 版本 (`maxSdkVersion`), 以及目标 SDK 版本 (`targetSdkVersion`)。

Android 应用只能够在 SDK 版本大于等于 `minSdkVersion` 的 Android 系统上运行, 如果系统的 SDK 版本低于应用的 `minSdkVersion`, 系统会拒绝安装该应用。`minSdkVersion` 值默认为 1。`maxSdkVersion` 从 Android 2.1 版本以后被废弃了^[12], 在此不再介绍。`targetSdkVersion` 用于告诉 Android 运行时该 APP 应用使用的行为和目标 SDK 版本的 API 行为一致。当应用程序被安装到一个 SDK 版本大于目标 SDK 版本的 Android 系统上时, 通过 `targetSdkVersion` 可以使得 Android 系统做前向兼容。`targetSdkVersion` 不设置默认等于 `minSdkVersion`。

2.1.4 Android Lint

Android Lint 是 ADT(Android Development Tools) 16 以后被引入的静态检测工具, 图 2.3 是其工作流程示意图。它通过使用多个检测过程 (checks) 来分析 Android 项目中的 Java、XML 等源文件中存在的潜在缺陷以帮助优化和提升项目质量。Lint 检测的缺陷类型非常多, 包括正确性、可用性、安全性、性能表现等多个方面的缺陷, 其报告结果不仅包含缺陷信息还包括缺陷的严重等级。用户可以通过在 `lint.xml` 文件中配置 Lint 检测的缺陷类型以及相应

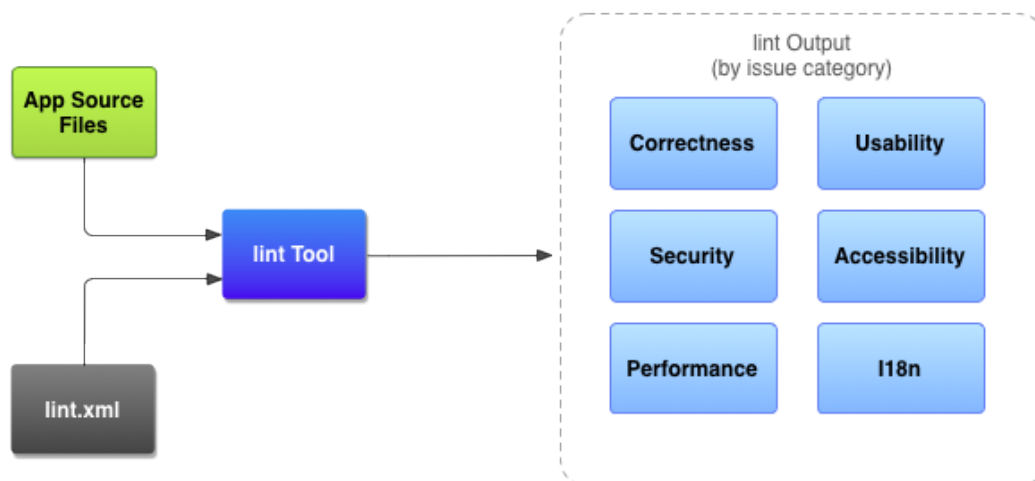


图 2.3 Lint 工具工作流程示意图

的缺陷报告等级^[13]。

当前，集成在 Android Studio 中的 Lint 工具默认配置有 200 多个检测过程 (checks)，其中一个叫 ApiDetector 的检测过程用于检测项目中不兼容引用缺陷。该工具扫描源码中所有 Android API 的引用，并对那些在应用支持的版本上不存在的 API 引用给出警告 (warn)。此外，Lint 不检测 (ignore) 使用 @TargetApi、@SuppressWarnings 等注解标记的代码片段^[14]。

2.2 静态程序分析

静态程序分析是在不执行程序的前提下获得程序运行时行为或程序性质的一类分析方法。静态程序分析技术在程序中潜在错误发现和缺陷检测等领域有广泛应用。下面简单介绍一下静态程序分析中常用的若干技术。

2.2.1 控制流分析

控制流分析技术静态刻画程序动态时刻程序可能执行情况，它将程序指令序列 (一般是三地址形式的中间表示如 Soot 中的 Jimple 格式^[15]) 按照分支跳转结构组织起来，构建成控制流图 (Control Flow Graph, CFG)。图2.4是一个示例，左侧是示例程序，中间和右侧分别是按照指令和基本块粒度构建的控制流图。控制流图中节点一般由一条指令或者一组连续执行而没有分支结构的指令 (通常称为基本块) 构成，边表示节点之间的控制流动关系。在图2.4中，我们使用数字表示指令，用大写字母表示基本块，具有相同字母的指令属于一个基本块。

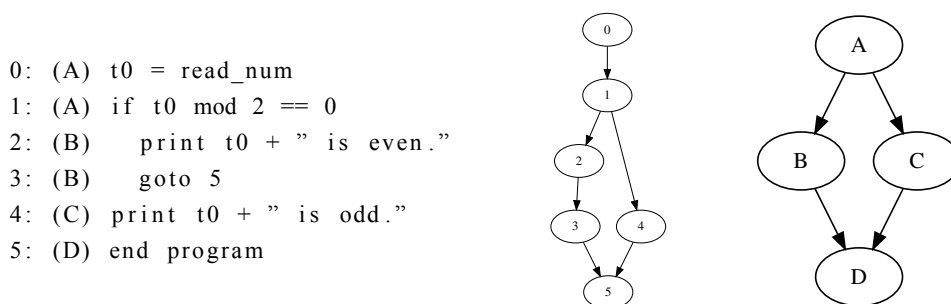


图 2.4 代码，指令 CFG 和基本块 CFG 示例

控制流图有过程内和过程间之分。为一个方法 (或函数) 构建的控制流图一般称为过程内控制流图 (Intraprocedure Control Flow Graph)，而对整个程序构建得到的控制流图一般称为过程间控制流图 (Interprocedure Control Flow Graph, ICFG)。区别在于构建过程间控制流图时，所有方法调用处 (callsite) 均会有一条指向被调用方法 (callee) 入口的边，同时也有一条从被调用方法返回处到 callsite 的边。

传统应用通常有一个 main 方法作为整个程序的入口，因此 ICFG 有唯一的入口点。Android 应用程序通常会有一个或多个入口点，这是因为通常来说一个 Android 应用程序中会包含一个或多个组件，这些组件在 AndroidManifest.xml 文件中定义时通过设置 android:exported 属性值为 true 来允许该组件被外部应用调用，进而使得该组件变成该应用的一个入口点。另外，当为组件设置 intent-filter 时，android:exported 默认为 true，即组件也会成为应用入口点。图 2.5 是绝大多数应用定义入口点的方法，活动组件 MainActivity 被定义为该应用的入口点，入口方法即为其 onCreate() 方法。在对 Android 应用进行程序分析时，为了保持和传统程序分析方法的一致性，以便于程序分析，通常我们会在 ICFG 图上添加一个辅助入口点 dummyMainMethod，然后从该入口点引边连接应用中所有入口点。在此新的 ICFG 上再进行程序分析。

在控制流图上可以做许多程序分析工作，比如计算程序语句之间的控制关系如前必经 (predominate)，后必经 (postdominate) 等，数据流分析技术一般也是在控制流图基础上进行分析。

2.2.2 数据流分析

在编译器优化中很多技术都是基于数据流分析，比如到达-定值，可用表达式和活跃变量等经典问题。数据流分析主要是计算特定程序点 (program point)

```

<application >
    .....
    <activity android:name="MainActivity">
        <intent-filter >
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter >
    </activity >
    .....
</application >

```

图 2.5 在 `AndroidManifest.xml` 中定义 `MainActivity` 组件为程序入口点

上的数据流值。数据流值是一个抽象概念，它可以是你关心的任何程序性质或者表达式等等。比如在做到达定值分析时数据流值是到达特定程序点的定值 (definition)，在计算可用表达式时数据流值是在特定程序点时所有可用表达式的集合。

数据流分析根据是在一个方法内部分析还是在整个程序上进行分析，可以分为过程内数据流分析和过程间数据流分析。考虑到数据流传播是否顺着控制结构先后顺序，数据流分析还可以分为流敏感 (flow-sensitive) 数据流分析和流不敏感 (flow-insensitive) 数据流分析。根据数据流传播是否区分调用点 (callsite) 或调用对象，数据流分析也可以分为上下文敏感 (context-sensitive) 的数据流分析和上下文不敏感 (context-insensitive) 的数据流分析。

数据流分析的理论基础是数学上的半格理论，只要半格高度有限且传播函数满足单调性，算法必定能计算出结果来^[16]。根据半格理论，经典的数据流求解算法一般是基于 worklist 的迭代方法。

2.2.3 别名分析

别名分析^[17]用于判定程序运行时刻一个内存地址是否可能被多种方式访问，比如在类 C 语言中两个指针变量是别名表明它们指向相同的内存地址，在 Java 等语言中如果两个变量具有别名关系说明它们引用同一个对象。别名有可能别名 (may alias) 和一定别名 (must alias) 之分。可能别名关系描述两个变量在程序执行时刻可能引用同一块内存，而一定别名关系则是说两个变量在程序执行时刻一定引用同一块内存。

2.3 其他基本概念

本节主要介绍一些文中会用到的基本概念，包括最长公共子序列、编辑距离、类层次关系图、API 签名以及前向兼容和后向兼容等。

• 最长公共子序列

最长公共子序列问题^[18]是计算机科学里一个非常经典的问题，正如字面意思描述的那样，它是求两个序列的所有公共子序列中最长的那一个子序列长度。该问题可以使用动态规划算法在多项式时间内解决。动态规划方程见公式 2.1。

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1 & \text{if } a[i] = b[j] \\ \max(dp[i-1][j], dp[i][j-1]) & \text{otherwise} \end{cases} \quad \dots (2.1)$$

• 编辑距离

编辑距离又称为 Levenshtein 距离^[19]，它是 Vladimir Levenshtein 在 1965 年提出。编辑距离是一种非常有效的衡量字符串之间差异的度量方法，它表示将一个文本串编辑成另一个文本串所需要的最少编辑操作数量。编辑距离也有动态规划算法，其动态规划方程见公式 2.2。

$$dp[i][j] = \begin{cases} \max(i, j) & \text{if } \min(i, j) = 0 \\ \min \begin{cases} dp[i-1][j] + 1 \\ dp[i][j-1] + 1 \\ dp[i-1][j-1] + 1_{a[i] \neq b[j]} \end{cases} & \text{otherwise} \end{cases} \quad \dots (2.2)$$

• 类层次关系图

继承是面向对象语言的三大基本特性之一。如果将程序中的类作为节点，将类与类之间的直接继承关系作为边，便得到程序的类层次关系图（Class Hierarchy Graph，简称 CHG） $G(\bar{C}, \bar{R})$ ，其中 \bar{C} 为程序中类的集合， \bar{R} 为程序中直接继承关系的集合。在 Java 语言中，继承关系被细分为继承 (extends) 和实现 (implements) 两种关系。其中类（或接口）只可以继承一个类（或接口），但一个类可以实现多个接口。仅考虑继承关系的 CHG 是一个有向树，同时考虑继承和实现关系的 CHG 是一个有向无环图。在 CHG 图上，对于某个给定的节

点，从根节点到该节点所有可能经过的节点以及从该节点开始能够到达的所有节点可以生成一个子图，这个生成子图称为该节点在 CHG 图上的图切片。

• API 签名

本文中分析的 API 有三类，分别为类型（Type, 包括类和接口）、方法（Method）和域（Field）。为唯一区分 API，引入 API 签名概念。

类型签名包括类型名称和其所在的完整包名。比如 Android SDK 中 View 类的签名为 `android.view.View`。

方法签名包括方法名、方法返回类型签名、方法参数类型签名以及方法所属类或接口的签名。比如 Android SDK 中 View 类中的 `resolveSize` 方法其签名为 `<android.view.View: int resolveSize(int,int)>`。

域签名包括域名，域的类型签名以及域所属类或接口的签名。比如表示设备 *API Level* 的域，其签名便是 `<android.os.Build$VERSION: int SDK_INT>`。

• 前向兼容和后向兼容

应用程序需要在特定的软件系统上运行。随着时间慢慢演化，同一时期市面上一个软件系统可能存在多个版本。如果针对当前软件系统开发的应用程序也能够旧版本的软件系统上运行且行为不变，那么我们说该软件系统具有后向兼容性 (*backward compatibility*)。如果针对当前软件系统开发的应用程序在未来该软件系统的新版本上还能够运行且行为不变，那么我们说该软件系统具有前向兼容性 (*forward compatibility*)。当前，Android 生态下既存在后向兼容性问题又存在前向兼容性问题。

第 3 章 经验性研究：Android API 演化导致的兼容性问题

3.1 研究问题

Android 应用开发非常依赖 Android SDK 中提供的 API^{[3][4][5]}。然而 Android 版本迅速更新导致 Android 生态下的兼容性问题很严重^{[20][21]}。当前，这种因版本更新导致的兼容性问题为 Android 应用开发带来了前所未有的挑战：开发一个 Android 应用需要同时兼容多个版本的 Android 系统。然而，关于 Android API 演化导致的兼容性相关话题在网络论坛上讨论很多^[8]，但关于这类兼容性问题严不严重，有多严重，有哪些深层次的原因使得这类兼容性变得严重以及 Android 开发者如何应对这类兼容问题，有没有特定兼容处理模式等等大家关心的问题，目前还没有看到相关研究。本章我们将对这些问题进行经验性研究，具体来说，我们将从如下几个方面展开研究。

RQ1：API 演化导致的兼容问题严不严重？

Android 应用允许在其清单文件 `AndroidManifest.xml` 中通过 `<uses-sdk>` 元素来指定应用兼容的 Android 平台范围^[12]，因此通常一款 Android 应用开发出来后能够同时支持多个 Android 平台。而为了兼容多个平台，Android 应用可能需要使用 Android 支持库提供兼容程序包或者通过其他方式做兼容处理。那么市场上有多少 Android 应用中做了这种兼容性处理呢？在一个 Android 应用中平均会有多少处做了这种兼容性处理？我们希望通过这个研究来确认这种由 API 演化导致的兼容问题的严重性。

RQ2：导致应用处理这类兼容问题的深层原因是什么？

API 演化不是导致应用做这类兼容处理的必然原因，那么其深层原因到底是什么呢？我们了解到 Android 官方针对 Android 版本过快演化导致的应用程序兼容性问题，推出了一些兼容处理程序包即 Android 支持库。既然有了 Android 支持库，为什么应用还需要自己处理这类兼容问题？通过该研究，我们希望进一步了解和认识 Android 兼容性问题。希望搞清楚应用处理 API 演化导致的兼容性问题的真正原因。

RQ3：应用如何修复 API 演化导致的兼容问题？

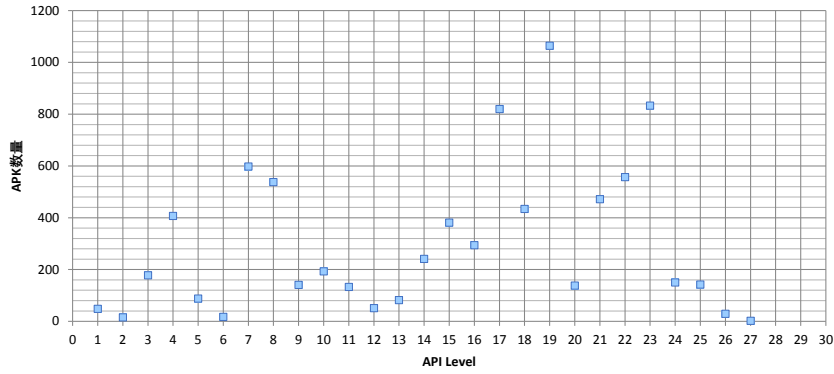


图 3.1 从 AndroZoo 下载的应用 targetSdkVersion 分布

通过本研究，我们希望了解应用开发者是如何对这种 API 演化导致的兼容性问题进行修复的，有没有一些可以复用的兼容修复模式或者这些修复有哪些规律等。通过该研究，我们希望对这种 API 演化导致的兼容性问题有个基本认识。

3.2 研究方法

在具体研究时，我们通过对大规模 Android 应用进行分析，了解 API 演化导致的兼容问题是否普遍存在；通过分析不同版本间 API 变化差异、Android 支持库对新 API 的支持情况等试图回答 Android 应用进行兼容修复的深层原因；最后，通过人工分析一些 Android 应用源码，了解应用中兼容处理的模式和规律。下面具体介绍数据选取和数据分析方法。

3.2.1 数据选取

在研究 RQ1-2 问题时，我们利用了 AndroZoo 仓库^[22]。AndroZoo 是专门为学术研究而收集的 Android 应用仓库。在我们实验收集数据时（2018 年 1 月 20 日），该仓库已经收集有 5,734,766 个 APK 文件，如表 3.1 所示，其中 Google Play 市场的 APP 应用占约 73%，剩下 27% 的 APP 应用来自 anzhi, appchina 等市场。鉴于计算资源和时间的限制，我们采用随机抽样的方法按照近 1:700 的比例随机下载了 8,047 个 APP 应用，其中 1 个 APP 的 targetSdkVersion 不在正常取值范围（1-27），我们将其剔除。对剩下的 APP 应用，我们按照其 targetSdkVersion 进行分类（其分布情况见图 3.1），再剔除 targetSdkVersion 值不在 16-26 区间的的应用，最后保留下来的应用有 4,936 个。具体情况见表 3.1 第三列和第四列。

为了研究 RQ2，我们需要选择一些 Android 版本的 SDK 以及对应的支持

| Market Name | # APKs | # Downloads | # Matches |
|-----------------|-----------|-------------|-----------|
| play.google.com | 4,205,084 | 5,947 | 4,318 |
| anzhi | 742,788 | 1,040 | 401 |
| appchina | 593,110 | 817 | 86 |
| mi.com | 113,583 | 147 | 113 |
| lmobile | 57,530 | 82 | 20 |
| angeeks | 55,794 | 86 | 3 |
| slideme | 52,467 | 74 | 35 |
| praguard | 10,186 | 4 | 0 |
| torrents | 5,294 | 8 | 0 |
| freewarelovers | 4,145 | 6 | 0 |
| proandroid | 3,683 | 0 | 0 |
| hiapk | 2,512 | 2 | 2 |
| fdroid | 2,023 | 4 | 2 |
| genome | 1,247 | 1 | 0 |
| apk_bang | 363 | 0 | 0 |
| unknown | 40 | 0 | 0 |

表 3.1 从 AndroZoo 中下载和选择应用

库。表1.2最后一列给出了各版本 Android 系统在市场上的分布情况。可以看到当前市场上 Android 应用使用的 API 主要在 Level 16-27 区间, *API Level* 16 以前的 Android 应用已经很少了。因此在我们的研究工作里, 我们将使用 *API Level* 在 16-27 区间的 Android 版本, 并为每一个 Level 选择一个 Android 发布版本。Android 版本的选择过程中我们尽可能选择能够下载并编译成功且 revision 尽可能新的版本。*API Level* 20 是专为可穿戴设备设计的 API 接口, 在我们的研究中未予考虑^[10]。我们最终选用的 Android 版本见表3.2。在确定好版本之后, 我们从 AOSP 上下载这些版本的源码进行编译, 并提取其 SDK (`android.jar`) 和对应支持库。

在研究 RQ3 时, 根据以往工作的经验^[7], Android 应用在处理 API 演化导致的兼容性时, 一般会使用 `android.os.Build.VERSION.SDK_INT` 来判断应用运行时系统支持的 *API Level*。因此我们首先对从 F-Droid^[23] 上下载的 1,425 个 Android 应用的最新版本进行预分析, 获得使用 `android.os.Build.VERSION.SDK_INT` 次数最多的 10 个应用。应用详细信息见表3.3, 其中第三列是应用 APK 大小, 第四列是应用源码在 GitHub 上被其他用户关注 (`star`) 的数量, 第五列是应用源码中 Java 代码行数, 第六列是

| Level | Selected Revision | Share | # Type | # Method | # Field |
|-------|----------------------|-------|--------|----------|---------|
| 16 | android-4.1.2_r2.1 | 1.7% | 3,217 | 30,057 | 11,679 |
| 17 | android-4.2.2_r1.2b | 2.6% | 3,259 | 30,569 | 12,004 |
| 18 | android-4.3_r3.1 | 0.7% | 3,290 | 31,104 | 12,512 |
| 19 | android-4.4_r1.2.0.1 | 12.0% | 3,412 | 32,139 | 13,325 |
| 21 | android-5.0.2_r3 | 5.4% | 3,673 | 35,426 | 16,333 |
| 22 | android-5.1.1_r9 | 19.2% | 3,683 | 35,568 | 16,380 |
| 23 | android-6.0.1_r9 | 28.1% | 3,471 | 35,239 | 16,757 |
| 24 | android-7.0.0_r7 | 22.3% | 3,823 | 39,773 | 20,016 |
| 25 | android-7.1.2_r9 | 6.2% | 3,828 | 39,896 | 20,076 |
| 26 | android-8.0.0_r9 | 0.8% | 4,181 | 44,307 | 21,419 |
| 27 | android-8.1.0_r9 | 0.3% | 4,201 | 44,455 | 21,471 |

表 3.2 选用的 Android 版本，市场分布及各类型 API 数量

应用中引用 `android.os.Build.VERSION.SDK_INT` 的次数。可以看到，我们挑选出来的应用具有代码量大，被关注度高等特点，这样大型的、热门的应用通常来说开发者在兼容处理方面做了大量努力，因此其兼容处理方式理应具有一定的代表性。

| App Name | Release | SIZE(MB) | # STAR | KLOC | # SDK_INTs |
|----------------------------------|----------------|----------|--------|-------|------------|
| org.telegram.messenger | 4.6.0a | 17 | 392 | 324.2 | 531 |
| com.poupa.vinylmusicplayer | 0.16.4.4 | 4.5 | 39 | 35.8 | 209 |
| org.glucosio.android | 1.3.0-FOSS | 8.1 | 305 | 8.2 | 195 |
| com.amaze.filemanager | 3.2.1 | 5.6 | 1,893 | 30.3 | 185 |
| im.vector.alpha | 0.8.1 | 19 | 625 | 52.5 | 185 |
| com.github.axet.maps | 8.1.0-4-Google | 49 | 2,658 | 120.9 | 179 |
| com.biglybt.android.client | 1.1.4 | 6 | 40 | 483.8 | 173 |
| eu.kanade.tachiyomi | 0.6.8 | 6.5 | 1,559 | 2.7 | 165 |
| org.bottiger.podcast | 0.160.2 | 11 | 107 | 41.9 | 165 |
| es.usc.citius.servando.calendula | 2.5.3 | 10 | 98 | 26.3 | 154 |

表 3.3 F-Droid 中适配次数最多的 10 个应用

F-Droid 是一个免费开源的 Android 应用商店，我们得到这 10 个 APP 应用的源码之后，人工分析其源码，寻找兼容性处理模式。为了了解更多兼容处理方式，我们还对 Android 支持库的源码（`android-7.0.0_r7`）进行了人工分析。

3.2.2 数据分析方法

为了回答 RQ1, 我们使用 Soot^[24] 遍历从 AndroZoo 选取的所有应用并统计每个应用中使用 `android.os.Build.VERSION.SDK_INT` 的次数，在

我们的实验中我们假设对该域的一次引用就代表一次兼容处理。为了回答 RQ2, 我们首先分析 Android 相邻版本之间的 API 差异。对于两个相邻 Android 版本, 我们还统计了有多少只在新版本中出现的 API 在新版本的支持库中被使用了。我们假设只要 API 在新版本支持库中被使用过, 该 API 即被支持库所支持, 以此来估算 Android 支持库对 API 演化的支持程度。另外, 我们还分析了那些在下一个版本中被删除的 API 在 Android 应用中被使用情况, 试图了解 API 演化导致的兼容处理和应用的升级适配之间的关系。为了回答 RQ3, 我们手动分析了选取的 10 个应用源码中所有有 `android.os.Build.VERSION.SDK_INT` 出现的代码片段, 寻找兼容处理模式并统计 `android.os.Build.VERSION.SDK_INT` 的不同种使用方式及其频数。

3.3 研究结果

3.3.1 RQ1: API 演化导致的兼容问题严不严重?

发现 1: 91.84% 应用做了兼容处理, 且平均有 50 多处做了兼容处理

在 RQ1 的研究中, 我们总共分析了 4,936 个 Android 应用, 其中有 32 个应用因 Soot 在分析时报错没能够分析成功。在剩下的 4,904 个应用中有 400 个应用没有做兼容处理, 占比仅为 8.16%, 由此可见绝大多数 Android 应用都会做 API 相关的兼容性处理。需要注意的是, 由于我们分析的是 Android 应用程序, 而 Android 应用 dex 文件中一般会包含 Android 支持库, 因此在我们分析时已经予以剔除。

此外, 我们对 Android 应用中处理兼容性次数进行分析, 如图 3.2, 我们发现每个应用中平均会有 55.45 处兼容性处理, 大于 160 以上的应用有 257 个, 有些应用甚至有 400-500 多处做了 API 相关兼容性处理, 由此可见 Android 应用开发中 API 兼容性处理问题是很严重的。

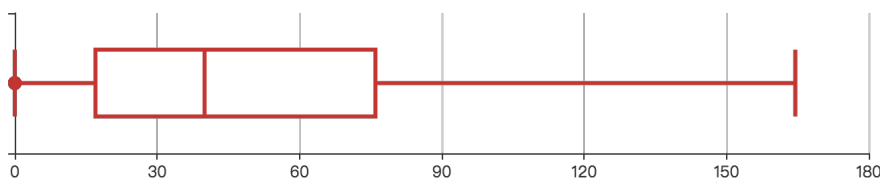


图 3.2 Android 应用中 `VERSION.SDK_INT` 使用数量箱线图

发现 2: 只有 6.74% 比例的 API 被广泛使用, 绝大多数 API 使用很少。

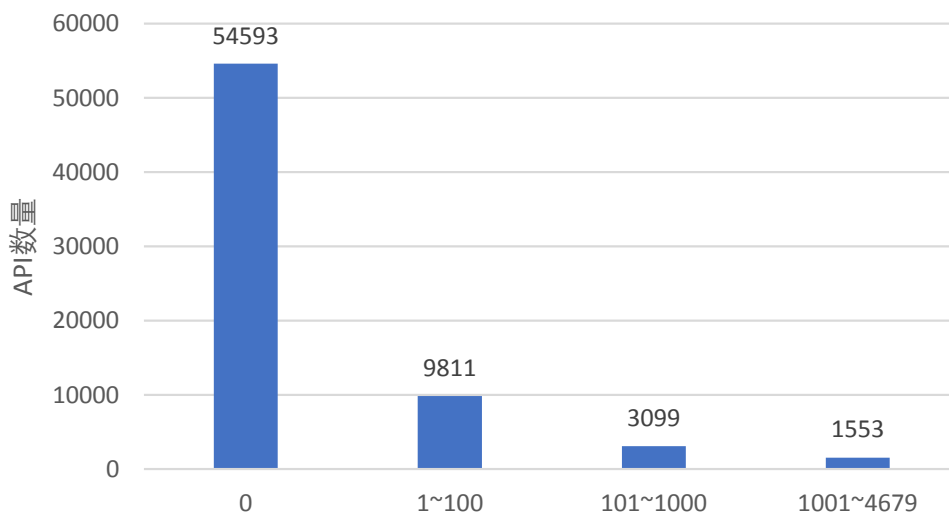


图 3.3 API 在 Android 应用中被引用情况

在 RQ1 的研究中，我们还统计了 SDK 中方法和域在 Android 应用中被使用情况。图 3.3 中我们统计了每个 API 在多少个 Android 应用中被引用过，我们发现 79.06% 的 API 在我们选择的 Android 应用中未被使用过，14.20% 的 API 被少于 100 个 Android 应用使用过（100 个应用占全部应用数量的 2.13%），只有不超过 6.74% 比例的 API 被广泛使用（被超过 100 个以上的应用引用过），这结果超过我们的预期。

同时，我们也统计了每个 API 在被分析的 Android 应用中的累计引用次数，并且得到了与此类似的结论，在此不再赘述。

另外，我们对引用次数超过 100 万的 API 进行了人工分析。在我们的统计数据里边，引用次数超过 100 万的 API 总共有 66 个，大体可以分为 3 类。最多的一类是以 `java.*` 为签名前缀的 API，共有 33 个，正好占了总数的一半。这些 API 分别属于 `java.lang.*`（18 个），`java.util.*`（12 个）和 `java.io.*`（3 个）中的方法，主要是文件，流，异常，字符串处理以及如列表，集合和映射等常用数据结构相关的 API；其次是以 `android.*` 为签名前缀的 API，共有 30 个。这些 API 主要是属于 `Parcel`（8 个），`Log`（5 个），`Bundle`（4 个），`Intent`（3 个），`Binder`（2 个），`Activity`（2 个），`View`（1 个），`IBinder`（1 个），`TextView`（1 个）等类中的方法。大致和 Android 应用开发指导书上提到的常用类一致。令我们惊讶的是唯一一个超过 100 万引用的域是 `android.os.Build.VERSION.SDK_INT`，由于应用程序一般使用该域来进行兼容处理，因此这可以间接表明 Android 应用中 API 演化导致的

兼容处理非常多。此外还有 3 个 API 属于 `org.json.JSONObject` 类，根据 Stack Overflow^① 等网络论坛上的回答，我们猜测可能 JSON 在 Android 应用开发中被广泛使用。具体 API 见附录A。

Answer to RQ1: Android 应用中这种 API 演化导致的兼容处理问题非常普遍，约 91.84% 的应用在它们的程序中做了这种兼容处理且平均做了 50 多处。

3.3.2 RQ2: 导致应用处理这类兼容问题的深层原因是什么?

发现 3: Android SDK 相邻版本间 API 数量差异巨大

为了回答 RQ2，我们统计了各 SDK 版本中含有的各种类型 API 的数量，见表 3.2 第 4-6 列。图 3.4 是根据各种类型 API 相邻版本之间数量差异绘制得到的箱线图。我们发现相邻 SDK 版本间，API 数量差异很大：类型平均差异（增加或减少）为 140.8 个，方法平均差异 1,505.6 个，域平均差异 979.2 个。其中方法差异最大，有些版本之间方法数量差异竟达到 5000 多个。

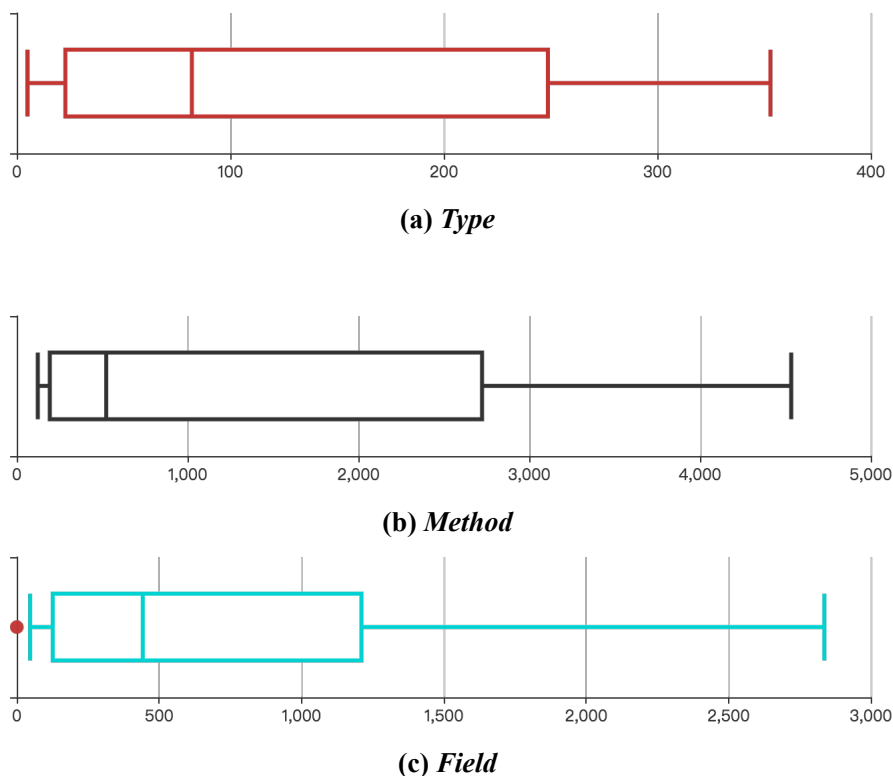


图 3.4 相邻版本 Android API 差异箱线图

发现 4: Android 支持库对 Android API 变化的支持平均不足 23%

① <https://stackoverflow.com/questions/23167167/why-do-we-use-json-in-android>

我们注意到 Android 官方为了消减 API 演化导致的兼容性问题，很早就设计了 Android 支持库。但为什么应用中还有那么多处兼容处理呢？这让我们觉得很奇怪。因此我们对比了每两个相邻版本的 Android SDK，对于那些在新版本中出现的 API，如果其在对应版本的 Android 支持库中被引用过，我们就认为该 API 被支持库做了兼容处理支持。由于 Android 支持库出来较晚，17 以前的版本没有支持库，因此我们从 level 18 的支持库开始统计。我们统计的数据见表 3.4，其中“/”之后的数字表示在新版本的 SDK 中新出现的 API 数量，“/”之前的数字表示在新版本支持库中被支持的 API 数量。

| 比较 Level | # Supported / # New introduced | | |
|----------|--------------------------------|-------------|-----------|
| | Type | Method | Field |
| 17vs18 | 0/67 | 0/744 | 0/571 |
| 18vs19 | 6/122 | 75/1,044 | 91/813 |
| 19vs21 | 11/265 | 136/3,383 | 3/3,022 |
| 21vs22 | 0/10 | 4/154 | 0/64 |
| 22vs23 | 2/152 | 2/1,970 | 0/823 |
| 23vs24 | 102/355 | 1,100/4,605 | 179/3,267 |
| 24vs25 | 1/5 | 7/132 | 0/60 |
| 25vs26 | 164/357 | 2,424/4,450 | 261/1,350 |

表 3.4 Android 支持库对 API 演化支持情况

从表 3.4 可以看到，方法的平均支持率最高但仅为 22.74%，域的平均支持率最低仅有 5.36%。虽然我们的统计方法比较粗糙，但可以肯定的是 Android 支持库对 API 演化支持程度不高。演化过程中新出现的 API 要么是对原有 API 的替换，要么是引入的新功能。应用开发时，如果支持库不提供相应兼容支持，应用程序就需要自己做兼容处理。如此低的支持率或许是大多数应用程序都做兼容处理的原因。

值得注意的是，Android 支持库第 26 版本对 SDK 从 25 到 26 的变化支持率较高，特别是方法的支持率超过了 50%，这可能说明 Google 官方在有意加强支持库对 Android API 变化的支持。

发现 5：不考虑 API 行为变化，绝大多数应用可以不做修改或做很少修改即可在下一版本的 Android 系统上运行

在研究 RQ2 时，我们统计了 Android 应用中方法和域的引用次数 (Total-Cnt)，SDK 中 API 的引用次数 (SdkCnt)，引用被标明为 `@Deprecated` 的 API

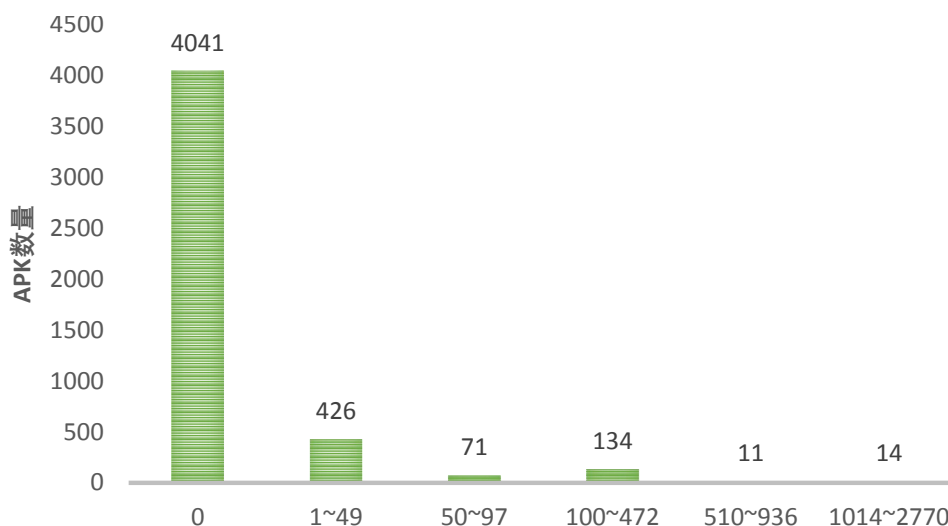


图 3.5 APK 中引用下一版本将被删除的 API 次数分布图

次数 (DepreCnt) 以及引用在下一 SDK 版本中被删除的 API 次数 (RmCnt)。根据我们的统计结果,我们发现 SdkCnt 平均为 15,026, 而 TotalCnt 平均为 83,526, 占比 23%, 验证了之前工作^[3-5]中 Android 应用开发严重依赖 Android 应用程序框架中提供的 API 的结论。DepreCnt 平均为 26, 且只有 290 个应用没有使用 Deprecated 的 API, 占比仅为 6.17%, 这再次验证了之前工作^[6]中过时 API 使用很普遍的结论。

此外,我们发现尽管平均来说一个 APK 引用在下一 SDK 版本中将被删除的 API 多达 16 次,但引用次数分布非常不均匀。图3.5是我们根据统计结果绘制出来的分布直方图。从直方图中我们发现 4,041 个 APK 没有引用将会在下一版本被删除的 API, 占比为 86%。在剩下有引用下一版本被删除 API 的 APK 当中,我们发现 426 个应用引用次数小于 50 次, 占比 64.94%。这说明一方面 Android SDK 演化时,开发者可能有意识避免 Android 应用的向前不兼容问题;另一方面说明如果不考虑 API 行为变化,那么绝大说 Android 应用可以在不做太多修改的情况下在 Android 新版本上运行。这个研究发现意味着新版本 Android 系统发布以后,Android 应用支持新版本而做的兼容处理大多数属于对新版本 API 提供的新功能特性的支持而非 API 替换适配。同时这个发现也意味着对 Android 程序进行简单升级适配以让其在新版本上正常工作意义不大。

Answer to RQ2: Android 应用中广泛处理 API 演化导致的兼容性问题的原因可以归纳为：Android 演化导致 API 剧烈变化，然而 Android 支持库对这种变化支持度很低，Android 应用为了在新版本向用户支持系统的新功能特性因而需要做兼容处理。

3.3.3 RQ3: 应用如何修复 API 演化导致的兼容问题?

发现 6: Android 应用中的兼容处理方式大多简单直接，较少使用特定的兼容模式

在 RQ3 中，我们研究分析了 10 个开源 Android 应用源码和一个 Android 支持库的源码，并总结了一些常见兼容处理模式。我们发现在 Android 应用程序的代码中绝大多数兼容处理方式是非常简单的，其中使用最多的一种方式是在 `if` 语句中判断运行时系统版本然后选择执行对应兼容处理的 API。比如图 3.6 是从 `com.amaze.filemanager` 源码中截取的一个片段，它在 *API Level* 18 以后调用了 `android.os.HandlerThread` 的 `quitSafely` 方法替代了旧版本的 `quit` 方法。

```

1  if (SDK_INT >= 18) {
2      // Let it finish up first with what it's doing
3      handlerThread.quitSafely();
4  } else
5      handlerThread.quit();

```

图 3.6 在 `if` 判断里直接做兼容处理

有时候判断语句中做兼容处理代码长度会比较长或者这种兼容代码片段会被多次使用，兼容片段会被单独提取出来作为一个独立的方法，然后在 `if` 判断语句中调用对应兼容处理的方法达到间接处理兼容性的效果。例如图 3.7 中，`CryptUtil` 类中提供了两种密码加密算法，分别封装在 `aesEncryptPassword` 和 `rsaEncryptPassword` 方法中。程序运行时根据应用程序所运行的设备的版本信息动态选择调用不同的密码加密算法。

```

1  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
2      return CryptUtil.aesEncryptPassword(plainText);
3  } else if (Build.VERSION.SDK_INT >= 18) {
4      return CryptUtil.rsaEncryptPassword(context, plainText);
5  } else
6      return plainText;

```

图 3.7 在 `if` 判断里间接做兼容处理

```

1 LayoutHelper.createFrame(Build.VERSION.SDK_INT >= 21 ? 56 : 60,
2   Build.VERSION.SDK_INT >= 21 ? 56 : 60,
3   (LocaleController.isRTL ? Gravity.LEFT : Gravity.RIGHT) |
4   Gravity.BOTTOM, LocaleController.isRTL ? 14 : 0, 0,
5   LocaleController.isRTL ? 0 : 14, 14);

```

图 3.8 使用问号表达式进行兼容处理

另外一种适配方式是使用问号表达式对一些数值按照 Android 版本进行适当调整。比如图3.8中根据 SDK 版本创建大小不同的 frame 组件。

上面方法多是对方法 API 的兼容处理模式。有时候程序会对 Android SDK 中提供的一个完整类型进行兼容适配，这种情况下，适配方法变得稍微复杂，且具有一定的适配模式。比如 level 为 24 的 Android 支持库中 ViewCompat, MenuItemCompat, KeyEventCompat 等以 Compat 为后缀的类具有相同的适配模式。我们以 ViewCompat 为例简单介绍这种兼容模式。图3.9是 Android 支持库中对 android.view.View 类的兼容处理类 ViewCompat 的简化版本。该版本里有一个 setElevation 方法需要做兼容处理（实际上 View 类不止这一个方法需要做兼容处理），为此在 ViewCompat 类内部引入一个 ViewCompatImpl 接口，该接口中对 setElevation 进行了声明。接下来，针对不同 Android SDK 版本，分别定义一个类比如 BaseViewCompatImpl, ..., LollipopViewCompatImpl 类等等，这些类中需要对 setElevation 方法进行特殊兼容处理的就在类中对该方法进行重新定义（比如第 15-17 行），否则继承之前版本的默认实现即可。最后在第 22-37 行，根据运行时系统的 SDK 版本实例化对应的兼容实例。当应用程序开发需要调用 View 的 setElevation 方法时，只需要调用 ViewCompat 的 setElevation 方法（第 39-40 行），该方法会自动处理好兼容性问题。Android 应用中也有类似处理方式，比如 org.bottiger.podcast 中的 RevealAnimator 以及 org.bottiger.podcast 中对 SupportAnimator 的处理方式。

发现 7: 88.65% 与控制结构有关的 `android.os.Build.VERSION.SDK_INT` 引用是直接在 if 判断语句中和常数 level 进行比较

在研究 RQ3 问题的时候，我们顺便分析了一下 `android.os.Build.VERSION.SDK_INT` 在我们选择的 10 个 Android 应用源码中被引用的方式。我们发现所有的引用大体可以归为四类：第一类是直接

```

1  public class ViewCompat {
2      interface ViewCompatImpl {
3          void setElevation(View view, float elevation);
4      }
5      static class BaseViewCompatImpl implements ViewCompatImpl {
6          @Override
7          public void setElevation(View view, float elevation) {
8              }
9      }
10     static class EclairMr1ViewCompatImpl extends BaseViewCompatImpl { ... }
11     static class GBViewCompatImpl extends EclairMr1ViewCompatImpl { ... }
12     .....
13     static class KitKatViewCompatImpl extends JbMr2ViewCompatImpl { ... }
14     static class LollipopViewCompatImpl extends KitKatViewCompatImpl {
15         public void setElevation(View view, float elevation) {
16             view.setElevation(elevation);
17         }
18     }
19     static class MarshmallowViewCompatImpl extends LollipopViewCompatImpl { ... }
20     static class Api24ViewCompatImpl extends MarshmallowViewCompatImpl { ... }
21
22     static final ViewCompatImpl IMPL;
23     static {
24         final int version = android.os.Build.VERSION.SDK_INT;
25         if (BuildCompat.isAtLeastN()) {
26             IMPL = new Api24ViewCompatImpl();
27         } else if (version >= 23) {
28             IMPL = new MarshmallowViewCompatImpl();
29         } else if (version >= 21) {
30             IMPL = new LollipopViewCompatImpl();
31         } .....
32         } else if (version >= 7) {
33             IMPL = new EclairMr1ViewCompatImpl();
34         } else {
35             IMPL = new BaseViewCompatImpl();
36         }
37     }
38
39     public static void setElevation(View view, float elevation) {
40         IMPL.setElevation(view, elevation);
41     }
42 }

```

图 3.9 简化版 ViewCompat 类对 View 的兼容支持实现

出现在 if 判断语句中并且直接和常数 level 值进行关系比较，偶尔会以一定别名 (must alias) 的变量替代该 API 出现在 if 判断语句中，这一类出现直接影响程序的控制流向 (C_1)；第二类是 `android.os.Build.VERSION.SDK_INT` 直接和常数 level 值进行关系比较运算后直接赋值给一个 bool 变量，然后在 if 语句中判断 bool 变量的 true 或 false，这一类出现会间接影响程序的控制流向 (C_2)；第三类是其他比较复杂的间接影响控制流向的引用，比如说用了一个函数封装 `android.os.Build.VERSION.SDK_INT` 和 *API Level* 的比较，使用时直接调用该函数进行判断 (C_3)；最后一类引用是和控制结构无关的引用，比如说出现在打印语句中，我们将问号表达式也归入此类，因为这种使用基本上都是非常简单的常数适配 (C_4)。

表3.5是我们统计得到的结果，可以看到直接在 if 判断语句中引用 `android.os.Build.VERSION.SDK_INT` 和 level 常数进行比较

| App Name | # C ₁ | # C ₂ | # C ₃ | # C ₄ |
|----------------------------------|------------------|------------------|------------------|------------------|
| org.telegram.messenger | 394 | 15 | 84 | 109 |
| com.poupa.vinylmusicplayer | 48 | 0 | 0 | 4 |
| org.glucosio.android | 5 | 0 | 0 | 0 |
| com.amaze.filemanager | 127 | 0 | 0 | 1 |
| im.vector.alpha | 40 | 0 | 0 | 3 |
| com.github.axet.maps | 39 | 0 | 2 | 4 |
| com.biglybt.android.client | 38 | 0 | 1 | 1 |
| eu.kanade.tachiyomi | 31 | 0 | 0 | 0 |
| org.bottiger.podcast | 79 | 1 | 2 | 0 |
| es.usc.citius.servando.calendula | 27 | 0 | 1 | 0 |

表 3.5 Android 应用源码中 VERSION.SDK_INT 被引用情况统计

的占绝大多数。如果仅考虑和程序控制结构相关的引用，我们发现 有 88.65% 比例的引用属于 C₁，其中 org.telegram.messenger 应用中 C₃ 出现 84 次，出现这么多次的原因是其在 Util 类中定义了一个 SDK_INT 静态域对 android.os.Build.VERSION.SDK_INT 进行了封装，然后在源码中多次使用了 Util.SDK_INT 替代对 android.os.Build.VERSION.SDK_INT 的引用。

Answer to RQ3: 根据我们的人工分析，Android 应用中 API 演化导致的兼容处理方法通常是非常简单的，较少具有特定的处理模式。另外，我们发现对 android.os.Build.VERSION.SDK_INT 的使用大致可以分为 4 类，其中三类和程序控制流有关。这三类中 88.65% 使用属于第一类，即在“if”条件表达式中直接与常数 level 进行比较。

3.4 本章小结

本章围绕 Android API 演化导致的兼容性问题，从应用程序中兼容处理是否普遍、导致应用做兼容处理的深层原因以及应用程序如何修复兼容问题等多个角度做了许多经验性的研究工作。

根据我们的研究结果，我们发现绝大多数应用都做了 API 相关的兼容处理，而且平均每个应用会有 50 多处在做这种兼容处理。我们还发现被广泛使用的 API 比例很少，不超过 6.74%，然而 android.os.Build.VERSION.SDK_INT 是被使用最多的域，这可能侧面证明 Android 应用中 API 演化导致的兼容处理

是普遍的。

另外，我们对 Android 应用中这种兼容问题导致的深层原因进行了分析，我们发现 Android 演化导致相邻版本间 Android API 数量差异很大，但 Android 支持库对 Android API 变化的支持程度很低。我们的试验结果显示 Android 支持库对 Android API 的变化平均支持度不足 23%。应用为了使用 Android 新版本提供的一些新特性，因此需要自己进行兼容处理。

最后通过对 10 个应用以及 1 个支持库的源码进行人工分析，我们发现应用中大多数兼容处理方式是比较简单直接的，88.65% 的兼容处理是通过在 if 语句中直接判断运行时系统版本，复杂的兼容处理方式比较少见。

第4章 IctApiFinder: Android API 的不兼容引用检测

4.1 概述

Android 应用在开发时通常在其清单文件 `AndroidManifest.xml` 中声明其可以同时兼容多个版本的 Android 系统。通过第三章的研究，我们的确发现绝大多数 Android 应用都会做 API 相关的兼容性处理。但实际开发中许多应用由于人力或资金不足等原因并没能够在其声明支持的所有 Android 系统上实际测试过。少数应用虽然在发布之前会使用大公司提供的兼容测试服务比如腾讯的 WeTest 等进行测试，但由于动态执行路径覆盖有限等原因，一些不兼容性问题在测试时未能充分暴露出来，因此应用上线之后仍然常会有闪退现象出现。

根据我们的调研，我们发现 Android 实际上提供了一个叫 Lint 的静态检测工具，该工具可以在 Android 开发时扫描应用程序源码。对所有使用的 API，如果该 API 是 `minSdkVersion` 以后添加的并且开发者没有使用 `@SupressLint` 或 `@TargetApi` 等进行注解，则 Lint 会给开发者报告错误提醒。图4.1是我们人为构造的一个不兼容 API 引用实例。`startDrag` 方法 (API_1) 在 Level 11 被引入，在 Level 24 被废弃，而 `startDragAndDrop` 方法 (API_2) 只在 Level 24 以后的 SDK 中存在，用于替换被废弃了的 `startDrag`

```

1 // minSdkVersion: 10; targetSdkVersion 27.
2 public class MainActivity extends Activity {
3     private TextView mView;
4     protected void onCreate(Bundle bundle) {
5         ...
6         if (Build.VERSION.SDK_INT >= 24)
7             wrapper(mView, c, s, null, i);
8         else
9             mView.startDrag(c, s, null, i); // API1 [11, 23]
10    }
11    private wrapper(View v, ClipData c, ...) {
12        v.startDragAndDrop(c, s, o, i); // API2, [24, 27]
13    }
14 }

```

图 4.1 Android API 不兼容引用示例

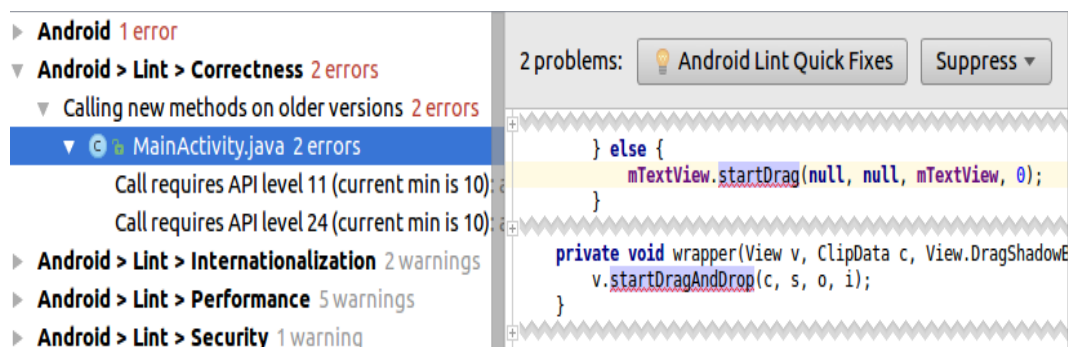


图 4.2 Android Lint 工具分析结果

方法。这段程序所在的 Android 应用其最小 SDK 版本为 10，目标 SDK 版本是 27。容易发现虽然 API_2 在 Level 10-23 并不存在，但对 API_2 的引用却并不是不兼容引用。这是因为 API_2 引用上下文中对 Level 有限制。对 API_1 引用是不兼容引用，因为当该应用运行在 Level 为 10 的 Android 系统上，由于其应用框架不提供 API_1 的实现接口，该应用程序将会崩溃。

对于图4.1这个例子，我们使用 Android Lint 工具进行分析。图4.2是 Android Studio 中 Lint 工具报告给出的结果。可以看到 Lint 工具给开发者报告了两个错误，该工具认为 API_1 和 API_2 分别是 Level 11 和 24 之后添加进 SDK 中的，均超过了应用设定的 `minSdkVersion` 的值。虽然这样的报告对开发者来说是有用的，但还不够精确。由于不考虑调用上下文信息，Lint 工具对 API_2 的报告实际上是一个误报。根据我们从相关开发者论坛中了解到的情况来看，开发者在开发过程中很少会使用 Lint 工具去分析优化应用程序，这可能和 Lint 的误报率过高有很大关系。另外，Android 应用开发过程中会依赖许多外部库或模块，Lint 对这些外部库或模块的分析能力有限。如果应用使用的第三方库中有引用不兼容的 API，Lint 工具将无法检测出来。但这样的引用同样会引发应用上线后崩溃，而且检测起来非常困难。

本章，我们提出一种静态检测 Android API 不兼容使用的方法，基于这个方法我们开发了一个检测 Android API 不兼容使用的原型工具 IctApiFinder^① 并即将开源。据我们所知，目前这还是第一个使用静态程序分析的方法检测 Android API 不兼容使用的工具。

① <https://bitbucket.org/DongjieHe/ictapifinder>

4.2 检测方法

为了检查 Android 应用中的不兼容 API 引用，我们首先要知道什么样的引用是不兼容引用。定理 4.1 给出了不兼容引用的充分必要条件。

定理 4.1 对于一个应用， API_i 的使用是不兼容的当且仅当它满足三个条件：

- 存在一个能让应用可以安装上去的 Android 版本，即要求该 Android 版本的 API Level 大于等于应用 `AndroidManifest.xml` 中声明的 `minSdkVersion` 值。
- 在这个 Android 版本上应用至少存在一条可执行路径可以执行到 API_i 的被引用处。
- 在能够执行到 API_i 的 Android 版本上，其对应的 SDK 中没有 API_i 的定义。

第一个条件限定了不兼容 SDK 版本的可能范围，如果应用在某个 Android 平台上存在 API 不兼容使用问题，那么第一个必要条件要求该 Android 平台支持的 SDK 版本必须大于等于该应用的 `minSdkVersion`。第三个条件是否满足也比较容易检测，只需要拿到对应版本的 Android SDK，然后判断里边是否含有目标 API 即可。但是第二个条件是否能够被满足检测起来就不那么容易了，需要考虑到该 API 被引用的上下文情况。在图 4.2 中，Lint 工具误报原因正是因为没有考虑到 API 被引用的上下文。

为了确定在哪些 Android 版本上存在执行路径到达 API 的引用处，最简单直接的方法就是在所有版本的 Android 系统上去测试，构造一条执行路径。然而这种方法非常耗时费力。因此，我们提出一种使用静态程序分析的方法来检测 API 的不兼容使用。我们首先将该问题建模成数据流问题，然后应用成熟的数据流分析算法进行求解。

4.2.1 建立数据流模型

对于我们这个问题，一个程序点（program point）的数据流值显然就是能够到达该点的所有 Android SDK 版本的集合。当前，Android SDK 版本是一个有限集合即数字 1-27。由于是求存在可达路径的 API 版本，因此建模好的数据

流模型应当是前向的 (forward) 的数据流分析, 且合并操作为集合并操作。为了构造数据流方程, 我们首先需要确定会对数据流值产生影响的程序指令。根据第三章经验研究 (特别是发现 7), 我们发现会对数据流值产生影响的程序语句有 C_1 , C_2 和 C_3 三类, 且均是 “if” 分支语句。我们定义这类 “if” 分支语句为数据流值相关的 “if” 分支语句 (Data-Flow-fact Relevant “if” statement, 简称 DFR “if” 分支语句)。关于 DFR “if” 分支语句的获取方法我们将在 4.2.2 介绍。

得到这些 if 指令后, 我们就很容易得到其数据流分析方程, 具体见公式 4.1 和 4.2。 $KILL_i$ 和具体的 if 指令中的条件判断有关, 比如如果判断表达式是 $SDK_INT \geq 24$, 那么 $KILL_i = \{1, 2, \dots, 23\}$, $\overline{KILL}_i = \{24, 25, 26, 27\}$ 。

$$IN_i = \bigcup_{p \in pred_i} (OUT_p) \quad \dots (4.1)$$

$$OUT_i = \begin{cases} IN_i - KILL_i & \text{DFR if 指令 true 分支后继指令} \\ IN_i - \overline{KILL}_i & \text{DFR if 指令 false 分支后继指令} \\ IN_i & \text{其他指令} \end{cases} \quad \dots (4.2)$$

上面构建的数据流模型, 其数据流值的集合和合并操作一起构成了一个高度有限的完全格。其中 $\top = \{1, 2, \dots, 27\}$, $\perp = \emptyset$, \sqcup 是集合并操作 \cup , 对任意数据流值 x , 满足 $\top \sqcup x = \top$, $\perp \sqcup x = x$ 。此外, 容易看出来转移函数公式 4.2 是单调的。根据 2.2.2 节提到的数据流分析理论, 半格高度有限且传播函数单调可以保证数据流值在有限时间内计算出来。因此我们这个问题可以通过该数据流模型进行计算求解。

本工作在建立好上述数据流计算模型之后, 我们在算法 1 中给出一个基于数据流分析的、检测 Android 应用中 API 不兼容使用缺陷的、通用的算法伪代码描述。分析过程 (第 1~7 行) 可以分成五个步骤, 首先进行预分析收集应用程序中使用的 API 集合以及和数据流值相关的分支条件语句, 然后为 Android 应用构建过程间控制流图, 接着使用数据流分析算法计算每个 API 使用点处的数据流值即能够执行到此程序点的 Android SDK level 集合 (第 4 行), 再接着检查确认不兼容的 API 使用情况, 最后对检测出来的, 有问题的 API 报告给用户。在设计算法具体实现时, 我们使用了高效且精度更高的过程间数据流分析算法 IFDS 算法^[25]。我们将在 4.2.3 详细介绍求解方法。

Algorithm 1 Android 应用中 API 不兼容使用检测算法

```

1: procedure runAnalysis(app)
2:    $\langle if2kill, apiSet \rangle \leftarrow$  preAnalysis(app)
3:   icfg  $\leftarrow$  constructCallGraph(app) ▷construct ICFG for app
4:   DataFlowSolver.solve(if2kill, icfg) ▷calculate data flow facts
5:   bugList  $\leftarrow$  checkAPICompatibility(apiSet, app)
6:   reportBug(bugList) ▷report incompatible API use bugs
7: end procedure
8:
9: procedure preAnalysis(app) ▷collect API use and our concern IfStmt
10:  for all method sm  $\in$  app do
11:    aliasSet  $\leftarrow$  SdkIntMustAliasAnalysis(sm)
12:    for all statement stmt  $\in$  sm do
13:      if stmt is IfStmt then
14:         $if2kill \leftrightarrow$  DFRIfStmt(stmt, aliasSet)
15:      end if
16:      if stmt is InvokeStmt then
17:         $apiSet \leftrightarrow$  sdkApi(stmt) ▷add sdk api into apiSet
18:      end if
19:    end for
20:  end for
21:  return  $\langle if2kill, apiSet \rangle$ 
22: end procedure
23:
24: procedure checkAPICompatibility(apiSet, app)
25:   $[minLevel, maxLevel] \leftarrow$  apiLevelRange(app)
26:  for api  $\in$  apiSet do
27:    livingSet  $\leftarrow$  DataFlowSolver.livingFacts(api)
28:    for i  $\in$  livingSet do
29:      if  $i \in [minLevel, maxLevel] \wedge api \notin SDK_i$  then
30:         $noLiving \leftrightarrow i$  ▷add level i into not-living list
31:      end if
32:    end for
33:     $bugList \leftrightarrow \langle api, noLiving \rangle$  ▷collect potential bugs
34:  end for
35:  return bugList
36: end procedure

```

4.2.2 预分析

在预分析阶段（第 9~22 行），我们需要收集 Android 应用程序中所有和数据流值相关的 if 分支语句以及所有 API 引用语句。收集 if 分支语句主要用于 4.2.3 中数据流计算求解阶段计算数据流转移函数。收集 API 引用语句是出于效率考虑和节省存储空间，我们仅存储 API 引用语句以及一些合并点的数据流值，其他中间无关语句的数据流值不存储，从而达到节省空间的效果。另外，在最后检查是否是不兼容 API 引用时，我们不需要再次遍历程序指令去寻找 API 引用。

我们重点考虑的是属于 C_1 中的 if 分支语句（占比 88.65%）。这类待收集的 if 分支语句其条件表达式具有特定形式的二元关系运算表达式，关系运算包括“大于 >”，“大于等于 >=”，“等于 ==”，“不等于 !=”，“小于等于 <=”，“小于 <”。两个操作数中有一个一定是整数常量，另外一个应该和 SDK_INT 是 Must alias 关系。因此我们首先使用了一个过程内的数据流分析计算每个程序点和 SDK_INT 具有 Must alias 关系的变量集合（第 11 行），然后遍历该过程内的每一条语句，收集该过程内符合条件的 if 分支语句和 API 引用语句（第 12~19 行）。出于简化考虑，在算法 1 中我们并没有给出其他类型语句（如赋值语句）中出现的 API 引用。

在计算 SdkIntMustAliasAnalysis 这个过程内数据流分析时，我们假定方法中使用 SDK_INT 的方式相对直接，因此仅考虑赋值语句这种简单的别名传播方式。详细过程在此不再介绍。

4.2.3 IFDS 求解

在我们的工作中，我们使用 IFDS 算法计算 Android 应用程序中 API 引用点的数据流值（第 4 行），即存在执行路径到达该 API 使用点的 Android SDK 版本集合。IFDS^[25] 是一个非常高效且计算精度很高的过程间数据流分析算法，它是一个同时支持上下文敏感和流敏感的数据流分析方法。IFDS 是由 Interprocedural, Finite, Distributive 以及 Subset 四个英文单词的首字母缩写得到，它要求使用该框架求解的数据流分析问题必须同时满足四个特点：第一个是要求该数据流分析必须是跨过程的，仅做过程内分析不需要使用该方法；第二个要求该数据流分析过程中，所有可能的数据流值必须是确定的，有限个数的；第三个要求是合并操作对所有数据流方程满足分配率，即满足

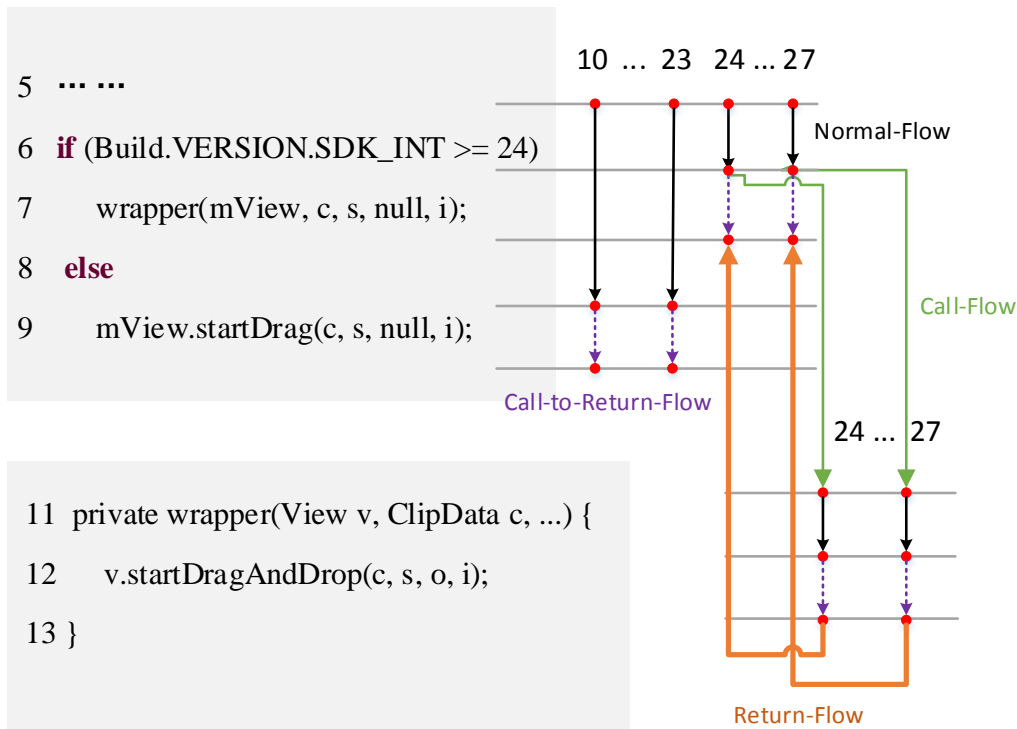


图 4.3 IFDS 算法检测不兼容示例

$f(x_1) \cup f(x_2) = f(x_1 \cup x_2)$; 最后一点要求所有程序点算出来的结果必须是所有可能数据流值的一个子集。我们的研究问题正好同时满足这四个性质，因此可以使用 IFDS 算法求解。

IFDS 算法将数据流分析问题转化成图可达问题，它根据数据流方程将每一个数据流值沿着过程间控制流图 (ICFG) 进行扩展得到超图，在此基础上提出一个动态规划算法即 Tabulation 算法进行求解。超图上的边有四类即 call-to-return-flow 边、call-flow 边、return-flow 边以及 normal-flow 边。算法实现需要为这四类边构造四类转移函数。对我们的研究问题来说，除了和 DFR “if” 分支语句相关的普通边需要按照公式 4.2 特殊处理之外，其他边都是 $OUT_i = IN_i$ 。

图 4.3 是使用 IFDS 算法检测图 4.1 示例中 API 不兼容使用的示例。程序最开始数据流值为 $\{10, \dots, 27\}$ ，由于第 6 行的 if 语句和版本判断有关，因此根据普通边的数据流方程公式，到达第 7 行之前数据流值变为 $\{24, \dots, 27\}$ ，进入第 9 行时数据流值为 $\{10, \dots, 23\}$ 。程序第 7 行调用 wrapper 方法，数据流值通过 call-flow 边传到第 11 行，在 wrapper 方法体中没有版本判断的 if 语句，因此数据流值保持不变，即第 12 行对 startDragAndDrop 方法的引用只会在 $\{24, \dots, 27\}$ 区间的 Android 版本上被调用。

```

1 app-debug.apk minSdkVersion: 10, targetSdkVersion: 27
2 BUG: <android.widget.TextView: boolean startDrag(android.content.ClipData, android.view.
3     View$DragShadowBuilder, java.lang.Object, int)> called in <com.example.hedj.sdkinttest.
4     MainActivity: void onCreate(android.os.Bundle)> on line 9 not in [10]
5
6 reachable paths:
7     --><dummyMainClass: void dummyMainMethod(java.lang.String[])>
8     --><com.example.hedj.sdkinttest.MainActivity: void onCreate(android.os.Bundle)>

```

图 4.4 IctApiFinder 报告结果示例

4.2.4 兼容性检测与报告

兼容性检测方法 `checkAPICompatibility` 相对来说比较简单。首先调用 `apiLevelRange` 方法 (算法1第 25 行) 从 Android 应用的 `AndroidManifest.xml` 文件中读取应用的 `minSdkVersion` 和 `targetSdkVersion` 信息, 获知应用支持的 Android 版本范围。接下来, 对于在预处理阶段收集到的每一个 API 引用, 我们向数据流求解器查询该 API 引用处的数据流值, 即可能执行到该处的 Android 版本的集合 (第 27 行), 记为 `livingSet`。对于 `livingSet` 中的每一个值如果它在应用能够运行的版本范围内且在对应的 SDK 版本中又不包含该 API, 那么该引用就是一个不兼容引用 (第 28~32 行)。最后我们将有问题的 API 引用和其不被兼容的版本集合一起存入 `bugList` 中 (第 33 行) 在缺陷报告阶段报告给用户 (第 6 行)。

在图4.1中, 由于 `startDrag` 方法在 SDK 版本 11 以后才出现, 因此我们检测工具会报第 9 行的 API 引用在 SDK 版本为 10 的系统上会不兼容。程序第 7 行调用 `wrapper` 方法, 数据流值通过 `call-flow` 边传到第 11 行, 到达第 12 行时保持不变, 对 `startDragAndDrop` 方法的引用只会在 {24, ..., 27} 区间的 Android 版本上被调用。所以虽然该 Android 应用声明兼容的最小版本为 10, 但对 `startDragAndDrop` 方法的引用不会导致兼容性问题。

图4.4是我们的工具检测图4.1中 API 不兼容使用例子时给出的报告。从图中可以看出我们的报告可以非常精确的给出 API 在源码中的位置, 以及在哪个 API Level 上使用会存在不兼容问题。另外, 我们还编写了一个路径追踪器, 该工具沿着 ICFG 深度搜索, 能够给用户最多 10 条可能的到达路径供用户进行缺陷确认时使用。

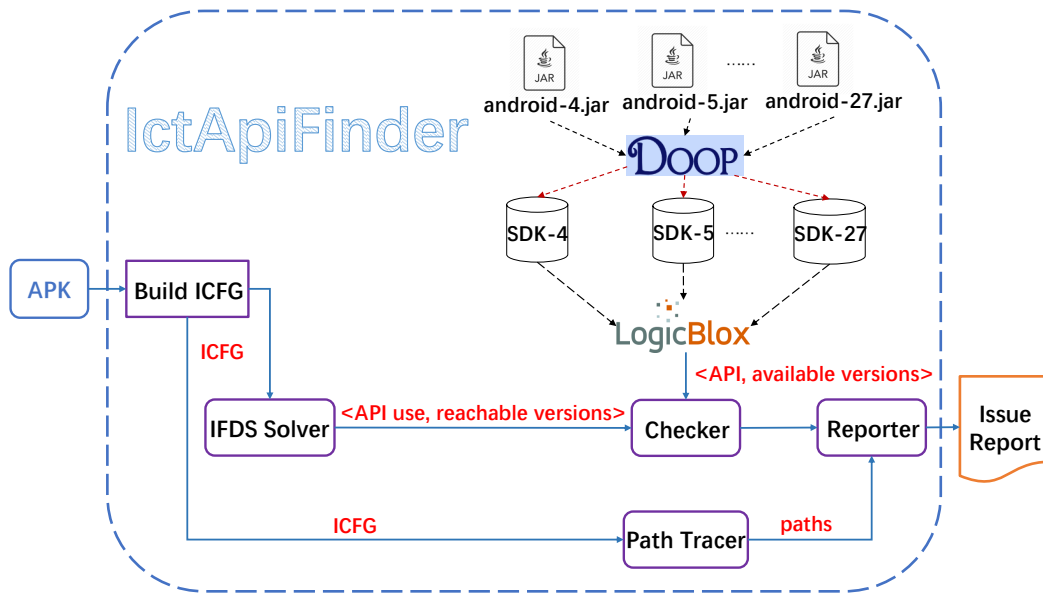


图 4.5 IctApiFinder Architecture.

4.3 工具实现

基于算法1, 我们构建了一个 Android 应用不兼容 API 使用检测的独立的原型工具 IctApiFinder (Incompatible Api use Finder)。图 4.5 是该工具的架构设计图。在该工具的构建过程中我们使用了 Soot^[24] 工具自带的 SPARK 调用图构建算法^[26] 构建 Android 应用的过程间控制流图。根据 Arzt, Steven 等^[27] 的实验结果, 基于 SPARK 算法构建的调用图在后续 IFDS 数据流分析中能在时间和空间上达到更好的效率。

在 2.2.1 节中, 我们提到 Android 应用由于存在多个入口点, 因此 ICFG 不只一个 entry point。为此需要引入一个 dummyMainMethod 辅助点将 Android 应用的 ICFG 变成传统的 Java 程序的 ICFG, 以便于使用传统的 Java 程序分析方法。在我们的实现中我们借鉴了 FlowDroid^[28] 中的处理方法, 为 Android 应用构建了 dummyMainMethod 方法来模拟组件的生命周期, 确定回调方法之间的先后执行顺序。同样, 我们也假设应用中组件的执行没有先后顺序。

另外, IFDS 求解器我们使用的是 Heros^[29], 这是一个对 IFDS 框架^[25] 的多线程实现, 具有非常好的可扩展性。

在检测 API 兼容性时, 我们需要知道每个 API 存在于哪些 SDK 版本中, 因此需要构建一个 API 到 API levels 的映射关系。以往的工作^{[6][12]} 通过分析伴随 Android SDK 一起发布的 api-versions.xml 文档以及 api_diff 目录中的 HTML 文件来获取 API 及其对应的 API Level 信息。但 Wu 等人^[12] 指

出这种基于文档的分析方法会由于文档本身存在的一些错误而导致分析结果不够准确。因此在我们的工作中，我们决定直接对 Android SDK 进行分析获得 API 相关信息。具体来说，我们对选定的 Android 版本进行编译得到其 SDK 文件即 `android.jar` 文件，然后我们借助 Doop 工具从 `android.jar` 文件中提取 API 签名信息并生成到 Datalog 数据库中，最后我们编写程序使用 Datalog 引擎工具从数据库中直接读取我们需要的 API 信息。Doop^[30] 是一个优秀的 Java 静态程序分析框架，它使用 Soot^[24] 工具从 jar 包中提取程序信息，并使用 Datalog^[31] 语言编写程序分析逻辑，然后使用 Datalog 引擎如 LogicBlox^[32] 对程序进行分析，程序分析结果最终保存在普通文件或数据库中。借助 Doop 提取 SDK 中的 API 信息大大便利了我们的研究工作。

4.4 工具评价

我们使用 IctApiFinder 对从 F-Droid 网站下载得到的最新的 1,426 个 Android 应用程序进行检测，其中 1 个应用因为 soot 解析时出错没有检测结果。图 4.6 是按照不兼容引用缺陷数量统计得到的应用分布情况。在剩下的 1,425 个应用中，工具报告有 1,064 个应用没有 API 不兼容使用问题，剩下 361 个应用我们的工具报告有一到多个 API 不兼容使用问题。这说明 F-Droid 网站上的应用至少有 74.67% 比例的应用是不存在 API 不兼容使用问题的，考虑到我们的工具还有误报，因此实际存在 API 不兼容使用的应用会比 24.74% 要低。这与第 3 章发现 1 的结论并不矛盾，可能正是因为多数应用中做了兼容性处理才不至于使得应用发布出来以后还存在这类 API 不兼容使用问题。

我们从上述分析报告有缺陷的应用中随机挑选了 20 个进行下一步实验分析。主要从能否有效降低误报率，能否发现真实应用中存在的不兼容使用缺陷以及检测效率等多个角度进行评价。

- 与 Android Lint 工具比较，IctApiFinder 可以有效降低误报率

对于我们挑选的 20 个 Android 应用，我们用我们的工具和 Android Studio 自带的 Lint 工具进行比较。我们从 F-Droid 网站上下载了这 20 个应用及其源代码。我们同时使用 IctApiFinder 和 Android Lint 工具检测这些应用中 API 不兼容使用情况。我们的工具在检测时使用的是一种过近似 (over-approximate) 的策略，不会存在漏报的情况。然而 Android Lint 工具对使用 `@SuppressWarnings`

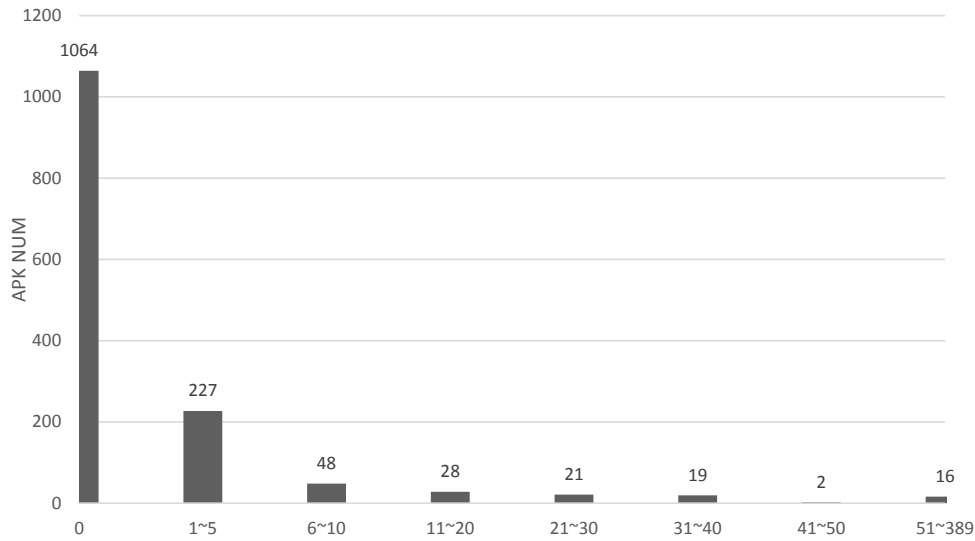


图 4.6 F-Droid 市场上 Android 应用的不兼容引用缺陷数量分布

等注解标记的代码片段不做检测，因此会存在漏报情况。我们在使用 Android Lint 检测应用时会事先去掉应用源码中 `@SuppressWarnings`, `@targetApi` 和 `@RequiresApi` 等注解信息以减少这种漏报对工具效果评测的影响。我们的工具要做到对添加注解信息的 API 不报告的效果其实也很容易。另一个导致 Android Lint 漏报的原因是 Lint 仅扫描 Android 应用源代码而对于应用依赖的第三方库不做考虑。然而根据我们了解到的情况，在 Android 应用开发时，应用通过 Gradle 自动构建工具^[33] 添加依赖并使用依赖的场景非常常见。对于这种漏报情况，我们在统计数据时，对于 IctApiFinder 报告的属于第三方库中的 API 不兼容引用缺陷，我们给 Lint 的结果也相应添加上。

表4.1是我们最后统计得到的结果，其中第三列是 Lint 工具报告的缺陷数量，第四列是我们开发的工具报告的数量。从表中数据可以看到，我们的工具报告缺陷数量明显少于 Android Studio 自带的 Lint 工具，平均来说 Lint 报告数量是我们的 5.5 倍，最多时报告的数量是我们的 20 多倍。由此可见，我们的工具能够平均有效减少 Lint 工具 82.1% 的误报数量。这在开发 Android 应用时对开发者来说是非常有用的，它既不会因误报过多使得开发者失去耐心，也不会因为漏报导致开发的程序还存在 API 不兼容使用问题。

- **IctApiFinder 能够检测出真实应用中存在的不兼容 API 使用缺陷**

我们对表4.1中 IctApiFinder 报告出来的结果进行人工检查并将我们认为比较有可能为真实的缺陷提交给开发者。表4.2是我们人工分析后得到的结果。

| ID | APP Name | Version | # Lint | # Finder |
|----|--------------------------------------------|-----------------|--------|----------|
| 1 | com.github.premnirmal.tickerwidget | 2.4.04 | 17 | 3 |
| 2 | de.christinecoenen.code.zapp | 1.10.0 | 21 | 1 |
| 3 | ca.rmen.android.networkmonitor | 1.30.0 | 46 | 13 |
| 4 | com.easytarget.micopi | 3.6.11 | 2 | 1 |
| 5 | com.prhlt.aemus.Read4SpeechExperiments | 1.1 | 1 | 1 |
| 6 | com.vonglasow.michael.qz | 1.1 | 32 | 7 |
| 7 | com.xargsgrep.portknocker | 1.0.11 | 44 | 17 |
| 8 | com.ymber.eleven | 1.0 | 15 | 9 |
| 9 | com.zegoggles.smssync | 1.5.11-beta7 | 5 | 3 |
| 10 | de.devmil.muzei.bingimageofthedayartsource | 1.4 | 37 | 37 |
| 11 | de.kromke.andreas.unpopmusicplayerfree | 1.41 | 29 | 14 |
| 12 | it.feio.android.omninotes.foss | 5.4.3 | 37 | 28 |
| 13 | jackpal.androidterm | 1.0.70-rebuild | 52 | 14 |
| 14 | jonas.tool.saveForOffline | 3.1.6 | 3 | 1 |
| 15 | net.opendasharchive.openarchive.release | 0.0.17-alpha-1 | 12 | 8 |
| 16 | org.servalproject | 0.93 | 5 | 1 |
| 17 | org.openintents.notepad | 1.5.4 | 4 | 3 |
| 18 | org.sensors2.osc | 0.2.0 | 25 | 14 |
| 19 | org.smssecure.smssecure | 0.16.8-unstable | 93 | 5 |
| 20 | org.softteg.slartus.forpdaplus | 3.4.8.2 | 732 | 37 |

表 4.1 Android Lint 和 IctApiFinder 报告缺陷数量比较

其中第四列是人工确认为真实缺陷的数量，第五列是人工确认为误报的数量。对于所有我们认为可能是真实缺陷的报告结果，我们首先尝试人工触发，对于难以触发出来的缺陷，我们提交给应用开发者等待开发者进一步确认。

表4.2中被矩形标记的是我们人工成功地触发出来的缺陷。出现该缺陷的应用saveForOffline 是一个非常简单的应用，此应用接收从浏览器中分享来的网页内容并允许用户离线时继续浏览。该应用在版本 3.1.6 时设置支持的最小 SDK 版本为 16。当使用该应用浏览离线的页面时，应用程序源码中会调用“<android.webkit.WebSettings: void setMediaPlaybackRequiresUserGesture(boolean)>”，该API在 Level 17 才被添加到 SDK 中，因此对该 API 的调用属于不兼容使用。我们在分析清楚不兼容原因之后，从腾讯的WeTest系统上租用了一个 GALAXY S3 云真机（支持的 API level 是 16），然后在上面运行该应用。不兼容引用缺陷很快便被触发出来。图4.7是缺陷触发出来时的 log 截图，可以看到出现缺陷的原因是“<android.webkit.WebSettings: void

| ID | APP Name | IctApiFinder | # TP | # FP |
|----|--------------------------------------------|--------------|------|------|
| 1 | com.github.premnirmal.tickerwidget | 3 | 3 | 0 |
| 2 | de.christinecoenen.code.zapp | 1 | 0 | 1 |
| 3 | ca.rmen.android.networkmonitor | 13 | 12 | 1 |
| 4 | com.easytarget.micopi | 1 | 0 | 1 |
| 5 | com.prhlt.aemus.Read4SpeechExperiments | 1 | 0 | 1 |
| 6 | com.vonglasow.michael.qz | 7 | 7 | 0 |
| 7 | com.xargsgrep.portknocker | 17 | 13 | 4 |
| 8 | com.ymber.eleven | 9 | 9 | 0 |
| 9 | com.zegoggles.smssync | 3 | 0 | 3 |
| 10 | de.devnil.muzei.bingimageofthedayartsource | 37 | 37 | 0 |
| 11 | de.kromke.andreas.unpopmusicplayerfree | 14 | 0 | 14 |
| 12 | it.feio.android.omninotes.foss | 28 | 24 | 4 |
| 13 | jackpal.androidterm | 14 | 0 | 14 |
| 14 | jonas.tool.saveForOffline | 1 | 1 | 0 |
| 15 | net.opendasharchive.openarchive.release | 8 | 0 | 8 |
| 16 | org.servalproject | 1 | 1 | 0 |
| 17 | org.openintents.notepad | 3 | 2 | 1 |
| 18 | org.sensors2.osc | 14 | 0 | 14 |
| 19 | org.smssecure.smssecure | 5 | 2 | 3 |
| 20 | org.softeg.slartus.forpdaplus | 37 | 35 | 2 |

表 4.2 人工分析 IctApiFinder 报告的 API 不兼容使用缺陷

setMediaPlaybackRequiresUserGesture (boolean) >”不存在，程序抛出 `java.lang.NoSuchMethodError` 异常。

出现这种缺陷的原因很直接，调用栈深度也不算太深，因此相对比较容易触发。应用中出现这类不兼容缺陷的原因主要是因为应用开发时可能没有利用 Android Studio 提供的检错功能，程序发布时也没有在其支持的所有版本上做过仔细的兼容测试。

表4.2中圆形标记的是已经被开发者确认过的缺陷。其中portknocker是一个端口碰撞应用，端口碰撞是一种通过对事先商议好的端口进行碰撞以获得特殊授权的防火墙技术。具体原理不在此讨论。该应用在版本 1.0.11 时支持的最小 SDK 版本为 10。该应用中使用了一个第三方库提供的组件“`com.ianhanniballake.localstorage.LocalStorageProvider`”，这个组件在代码实现时继承了“`android.provider.DocumentsProvider`”类，而这个类是在 Level 19 之后才添加到 SDK 中的，这就导致应用中调用的所有继承自“`android.provider.DocumentsProvider`”类中的 API 均

| 时间 | PID | 级别 | Tag | 日志内容 |
|--------------------|-------|----|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 03-30 23:38:34.495 | 12952 | I | SurfaceFlinger | SurfaceFlinger : eMagnificationFactorChanged. magnificationFactor=1.0 |
| 03-30 23:38:34.495 | 12952 | I | SurfaceFlinger | SurfaceFlinger : eZoomPositionChanged. zoomX=0.0 zoomY=0.0 |
| 03-30 23:38:34.510 | 23932 | I | am_on_paused_called | jonas.tool.saveForOffline.MainActivity |
| 03-30 23:38:34.525 | 13043 | I | am_restart_activity | [1122667056,5,jonas.tool.saveForOffline/ViewActivity] |
| 03-30 23:38:34.580 | 23932 | I | dalvikvm | Could not find method android.webkit.WebSettings.setMediaPlaybackRequiresUserGesture, referenced from method jonas.tool.saveForOffline.ViewActivity.setupWebView |
| 03-30 23:38:34.580 | 23932 | W | dalvikvm | VFY: unable to resolve virtual method 254: Landroid/webkit/WebSettings;.setMediaPlaybackRequiresUserGesture (Z)V |
| 03-30 23:38:34.645 | 23932 | I | webclipboard | clipservice: android.sec.clipboard.ClipboardExManager@42460010 |
| 03-30 23:38:34.665 | 23932 | W | dalvikvm | threadid=1: thread exiting with uncaught exception (group=0x416d92a0) |
| 03-30 23:38:34.670 | 23932 | E | AndroidRuntime | FATAL EXCEPTION: main |
| 03-30 23:38:34.670 | 23932 | E | AndroidRuntime | java.lang.NoSuchMethodError: android.webkit.WebSettings.setMediaPlaybackRequiresUserGesture |
| 03-30 23:38:34.670 | 23932 | E | AndroidRuntime | at jonas.tool.saveForOffline.ViewActivity.setupWebView(ViewActivity.java:125) |
| 03-30 23:38:34.670 | 23932 | E | AndroidRuntime | at jonas.tool.saveForOffline.ViewActivity.onCreate(ViewActivity.java:87) |
| 03-30 23:38:34.670 | 23932 | E | AndroidRuntime | at android.app.Activity.performCreate(Activity.java:5206) |
| 03-30 23:38:34.670 | 23932 | E | AndroidRuntime | at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1083) |
| 03-30 23:38:34.670 | 23932 | E | AndroidRuntime | at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2064) |
| 03-30 23:38:34.670 | 23932 | E | AndroidRuntime | at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2125) |

图 4.7 触发 `jonas.tool.saveForOffline` 中不兼容 API 使用缺陷

只能在 Level 19 之后的设备上运行，在 19 以前的设备上会出现不兼容使用缺陷。我们将该缺陷提交给了应用开发者并很快得到了他们的确认。图4.8是开发者回复截图，在后续的交流中，他们对我们发现并告知这个缺陷表示欢迎和感谢。当前该缺陷已经被开发者修复。

Android 应用开发时，Android Studio 默认使用 Gradle^[33] 作为项目自动构建工具。Android 应用开发通过在 `build.gradle` 这个 Groovy^[34] 脚本中添加依赖库链接的方式使用第三方库。这种开发模式导致第三方库中引用的一些 API 可能不在 Android 应用支持的版本范围内从而导致应用运行时可能会出现各种不兼容问题。这类不兼容问题在开发者开发时一般难以被发现，Android Lint 工具也无法检测这类不兼容问题。该例子表明使用我们的工具能够检测出这类第三方库中存在的不兼容 API 使用缺陷。

另一个被开发者确认的应用是 Serval Mesh（又被称为 Batphone）。该应用能够让用户通过 Wi-Fi 进行语音呼叫，发短信或文件共享而不需要 SIM 卡及商业移动电话运营商的支持。该应用在 0.93 版本时支持的最小 SDK 版本是 8。然而其源码中使用的“`java.lang.String: void String(byte[],int,int,java.nio.charset.Charset)`”构造函数却是在 level 9 之后才被添加到 SDK 中。我们将该缺陷报告给开发者，开发者不仅对这个缺陷进行了确认还迅速做了修复。图4.9左侧是开发者对我们提交的 *bug issue* 进行确认，右侧是对该缺陷的修复。

更多确认实例不在此一一列举了。表4.2中椭圆形标记的是当前被开发者



图 4.8 com.xargsgrep.portknocker 应用开发者确认缺陷报告

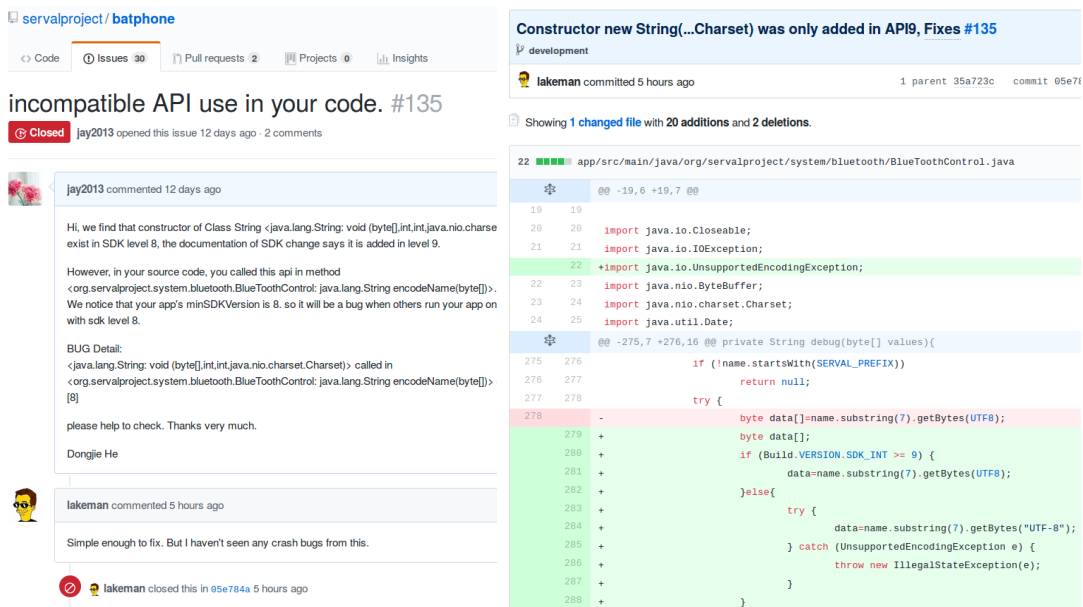


图 4.9 Batphone 应用开发者对缺陷进行确认和修复

标记但还没得到正式回应的缺陷，其他应用目前还没有收到开发者确认邮件。

从上述或触发或被确认的实例中可以看到我们的工具能够检查出实际应用中存在的 API 不兼容使用缺陷。这些不兼容使用缺陷大致可以分为两类，其中第一类是因应用中直接使用了不兼容的 API 但没有做任何兼容处理导致的。这类缺陷不是特别多，主要出现在应用声明的 `minSdkVersion` 附近，可以推测此类应用的开发者应该没有做过仔细测试。另一类缺陷通常出现在应用程序依赖的第三方库中。这些第三方库中也可能会引用一些 SDK 中的 API，但是这些 API 并不在应用支持的版本范围内，这就会导致应用在使用时可能因为第三方库中调用了不支持的 API 而导致应用崩溃。这类不兼容问题在应用开发时相对难以发现，Android Lint 工具也检测不到，因此可能更容易出现。

- 和动态检测工具比较，IctApiFinder 简单易用，检测速度快

我们了解到当前许多大公司内部都有专门测试 Android 兼容性的平台，例如国内腾讯的 WeTest，百度的 MTC，阿里的 MQC 以及中国移动的贯众云测试等。也有专门提供移动应用测试的服务平台如 Testin。表 4.3 是这些测试平台提供测试服务的基本概况。

| | 兼容测试 | 性能测试 | 功能测试 | 遍历测试 | 稳定测试 | 网络场景测试 | 真机租用 |
|---------------|-----------------------------|------|------|------|------|--------|------|
| Testin | ✓ 自动遍历，指定脚本 | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| WeTest | ✓ 急速 50 款，一小时交付 | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| MTC | ✓ 定制测试脚本，覆盖主要页面/功能，适配主流测试机型 | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| MQC | ✓ 安装，启动，monkey，卸载，自动登录 | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ |
| 贯众云 | ✓ 安装，启动，卸载，关键环节 | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ |

表 4.3 国内主要移动测试平台提供服务基本概况

我们特别关注了这些测试平台提供的兼容测试功能，我们发现这些测试平台均是通过动态测试的方法来检测移动应用的兼容性，其测试的机器主要是当前市场占有率较高的一些主流测试机型，因而不能够保证对应用支持的所

有版本进行测试。其次，这些测试平台通过遍历主要组件或者定制测试脚本的方式进行测试只能覆盖主要页面或功能，对一些不太容易到达的路径，动态测试方法不能够检测是否会存在兼容性问题。此外，这些测试平台进行测试的时间和需要覆盖测试的设备数量有关，一般快点的也需要至少半个小时才能出测试结果。

而我们的工具通过静态程序分析的方式可以覆盖到所有可能的执行路径，这是动态检测工具所无法比拟的。另外，我们的检测工具不需要定制测试脚本，简单易用。根据我们测试 F-Droid 网站上应用的经验，平均检测一个应用仅需 6.08 秒，最耗时的一个应用也只需要 3 分 45 秒即可分析完成，因此检测速度非常快。

- **IctApiFinder 能够帮助开发者简化代码，减少不必要的兼容处理结构**

在 Android 应用程序开发过程中，一些过去做得兼容处理程序随着应用本身的演化可能不再被支持。这些代码存在程序中不仅得不到执行，由于其总是出现在一些分支语句中因而会增加程序的代码复杂程度，不利于应用程序的维护。IctApiFinder 在检测 API 不兼容使用的同时，对于那些不可能被执行到的 API，我们也会报告给用户并建议用户删除。

图4.10是Calendula应用的开发者采纳我们的建议简化其应用代码。此应用原先在 *API Level* 小于 5 时调用 “`NotificationManager:cancel(String)`” 方法，然而由于此应用的最小 SDK 版本已经设为 16, 原先对 `cancel` 方法的兼容处理成为多余。开发者采纳了我们的建议对其代码进行了简化。从图4.10可以看到，原先 5 行的分支判断被一行代码取代，极大简化了应用程序代码。

4.5 本章小结


本章我们提出了一种静态检测 Android 应用程序中 API 不兼容引用的方法，并构建了一个检测工具 IctApiFinder。据我们所知，目前这还是第一个真正意义上用于检测 Android 应用 API 不兼容使用问题的静态分析工具。通过和 Android Studio 自带的 Lint 工具比对，我们的工具能够平均有效减少其 82.1% 的误报数量。和动态工具比较，我们的工具具有使用简单，检测速度快且无漏报等优点。通过对 F-Droid 上一些开源程序进行分析以及人工对分析结果的

PickupNotification: unwrap useless version checks

We have minApi 16, so there's no need to check if version is >= ECLAIR (5).

Rel #104

develop

 ontherunvaro committed 7 days ago Verified 1 parent 4502203 commit d37fc6de285a5

Showing 1 changed file with 2 additions and 10 deletions.

12 ■■■■ Calendula/src/main/java/es/usc/citius/servando/calendula/activities/PickupNotification.java

| Line | Change | Code |
|------|--------|--------------------------------------------------------------|
| 113 | 113 | public static void cancel(final Context context) { |
| 114 | 114 | final NotificationManager nm = (NotificationManager) context |
| 115 | 115 | .getSystemService(Context.NOTIFICATION_SERVICE); |
| 116 | - | if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.ECLAIR) { |
| 117 | - | nm.cancel(NOTIFICATION_TAG, 0); |
| 118 | - | } else { |
| 119 | - | nm.cancel(NOTIFICATION_TAG.hashCode()); |
| 120 | - | } |
| 116 | + | nm.cancel(NOTIFICATION_TAG, 0); |
| 121 | 117 | } |

图 4.10 Calendula 应用开发者采纳我们的建议简化应用代码

确认和复现，IctApiFinder 能够找到真实应用中一些不太容易被发现的不兼容 API 使用的缺陷，表现出非常不错的检测效果。

第 5 章 Android API 版本演化挖掘及 API 不兼容使用修复建议

5.1 概述

Android 版本更新迅速，在第 3 章中，我们对 Android API 变化从宏观上做了一些经验性的了解，但是我们对于每一个 API 个体在 Android 版本更替的过程中做了怎样的演化并不了解。在第 4 章中，我们的工具能够检测出一些兼容性相关的缺陷并给出一些可能的触发路径。当前，我们想对我们工具报告出的缺陷给出修复建议；下一步，我们可能还会对这种兼容性缺陷进行自动修复方面的研究。然而，为了达到这些目的，首先我们需要知道每个 API 是如何演化的，为此我们需要开发一个自动识别 Android API 演化的工具。

另一方面，根据第 3 章的研究，我们发现绝大多数应用程序在开发时都会处理 API 相关的兼容性问题。当开发者发现自己的应用存在 API 不兼容性使用时，通过检索 Android 官方文档中去了解 API 如何变化的方法似乎是非常低效率的。绝大多数开发者倾向于准备好的答案。我们以“Android API NoSuchMethodError Issue”为关键词使用 Google 搜索引擎检索，得到 137,000 条检索结果（2018 年 03 月 23 日）。通过人工查看前几页的条目，我们发现这些结果中多数是 SDK 中 API 演化导致，由此可见 API 的演化的确给应用开发者带来不少困扰。如果可以开发一个工具自动识别和收集 Android API 不同版本间的演化，供应用开发者查询使用，应该也是一件非常有意义的事情。

本章，我们开发了一个自动化识别 Android SDK 中 API 演化的工具 Focus，该工具可以识别任意两个版本 Android SDK 中相当比例的 API 的变化关系如替换、新添、删除等。借助该工具，我们为 Android SDK 中存在的每一个 API 构建其到其他 SDK 版本上的替换关系。结合第 4 章的工作，在报告兼容性缺陷的同时我们可以提供一份修复兼容性缺陷的建议。

5.2 Android API 变化识别方法

借鉴以往类似工作^{[35][36]}中的方法，我们开发了一个自动识别 API 变化的工具 Focus，主要使用了文本相似度、代码注释提取、类层次关系图以及调用

依赖分析等识别方法。本节将详细介绍这些识别方法。

5.2.1 文本相似度

API 的变化很多是因为代码重构导致^[37]，而简单的代码重构多以修改类，方法或域的名称，添加方法参数等形式出现，此类变化前后 API 的相似度往往很高，因此可以通过文本相似度来识别此类变化。

API 有类型、方法和域之分。不同类型的 API 相似度计算也不相同。为此，我们为 API 相似度计算进行了细致的建模工作。

• 普通文本的相似性

普通文本的相似性一般用文本之间最长公共子序列长度以及编辑距离（也称 levenshtein 距离）来综合度量。用 $lcs(a, b)$ 表示文本 a 和文本 b 最长公共子序列长度， $ld(a, b)$ 表示文本 a 和文本 b 最小编辑距离，则普通文本相似性可用公式5.1计算：

$$plainsim(a, b) = \frac{lcs(a, b)}{ld(a, b) + lcs(a, b)} \quad \dots (5.1)$$

• 标识符相似性

当前程序源码中出现频次最多的就是标识符。Java 语言中，程序开发人员通常以驼峰式^[38](Camel Case) 命名程序中用到的标识符。这些驼峰将标识符分隔成一个个有意义的单词。软件代码重构时往往是对其中某个单词进行修改或者调正单词之间的顺序。因此在计算标识符相似性的时候，我们不仅仅要考虑标识符本身的相似性，还要考虑标识符结构的相似性。例如“HelloWorld”和“WorldHello”单词文本相似度角度考虑不怎么相似，但它们的结构编辑距离为 1，因此又很相似。假定用 $structsim(a, b)$ 表示标识符 a 和标识符 b 的结构相似度，则我们用公式5.2来定义标识符的相似性。

$$idsim(a, b) = (1 - w) * structsim(a, b) + w * plainsim(a, b) \quad \dots (5.2)$$

其中 w 为纯文本相似度的权重因子。令 $|split(a)|$ 表示字符串 a 可以分成的单词个数，则 w 可以由公式5.3来定义。

$$w = e^{-\frac{\min(|split(a)|, |split(b)|) - 1}{10}} \quad \dots (5.3)$$

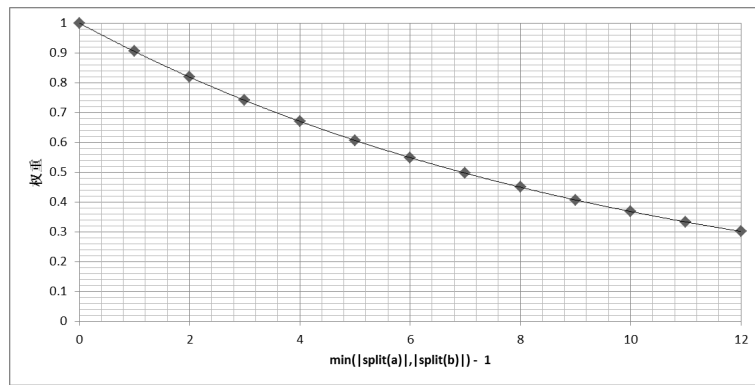


图 5.1 纯文本相似度权重变化

从图5.1可以直观的看到纯文本相似度的权重随着标识符分词数量先快后慢的下降，这比较符合我们对标识符相似度计算的要求。当标识符仅是一个单词构成时，公式5.2 退化成纯文本相似度计算公式，但当标识符由多个单词构成时，结构相似性权重会逐渐变高。计算结构相似性需要先对标识符分词，然后将每一个单词用一个唯一的字符代替得到标识符的结构表示，最后对结构表示再进行纯文本相似度计算得到最后的结构相似度。令 $rep(a)$ 表示标识符的结构表示，则结构相似度计算方法见公式5.4。

$$structsim(a, b) = plainsim(rep(a), rep(b)) \quad \dots (5.4)$$

为了对标识符进行分词，我们构建了如图5.2所示的有限状态自动机。S 和 E 分别是开始和结束状态，其他节点表示中间状态。边上标记的小写字母 u 表示当前遇到的字符是大写字母时进行的状态转移，小写字母 l 表示遇到小写字母进行的状态转移，遇到其他字符按照标有 x 字符的边进行状态转移。对于标有字符 D 的边，我们不仅要进行状态转移还需要将当前获得的子字符串处理成新单词并重新开始积累子字符串。例如，标识符“HEVCHighTierLevel51”通过该分词状态机将被分成 {“HEVC”, “High”, “Tier”, “Level51”} 四个单词，其结构表示可以为任意四个不同的字符构成的串比如“ijkl”。同样，对标识符“HighTierHEVCLLevel51”将获得 {“High”, “Tier”, “HEVC”, “Level51”} 四个单词，结构表示为“jkil”，因此这两个标识符的结构相似度就是“ijkl”和“jkil”的纯文本相似度。

- 类型，方法和域的相似性

有了纯文本和标识符相似性计算公式，类型，方法和域的相似性计算就比

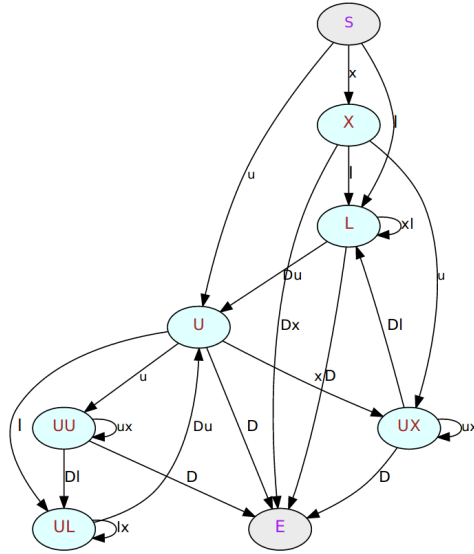


图 5.2 标识符分词状态机

较容易了。类型签名由类型名和类型所属包 (package) 名共同构成。定义类型相似性为:

$$typesim(a, b) = (1 - w) * packagesim(a, b) + w * namesim(a, b) \quad \dots (5.5)$$

其中, $packagesim(a, b)$ 表示类型 a 和类型 b 的包名的相似度, 为简单起见, 我们在具体实现时使用了纯文本相似性去计算。 $namesim(a, b)$ 表示类型 a 和类型 b 的类型名称的相似度, 使用标识符相似性公式进行计算。 w 表示名称相似度的权重, 定义为:

$$w = 1 - packagesim(a, b)/2 \quad \dots (5.6)$$

用于调节包名和类型名的权重。

方法签名由方法名, 方法所属类型, 方法返回类型以及方法参数类型列表构成。其中除方法名之外的元素又被称为**方法描述**。我们按照公式5.7计算方法间的相似性。

$$methodsim(a, b) = \begin{cases} w + (1 - w) * AVG(argsim(i)) & \text{方法名和参数数目相同} \\ w + (1 - w) * plainsim(a.desc, b.desc) & \text{方法名相同, 参数数目不同} \\ W + (1 - W) * idsim(a.name, b.name) & \text{方法描述相同} \\ 0 & \text{其他情况} \end{cases} \quad \dots (5.7)$$

其中, $argsim(i)$ 表示第 i 个参数类型的相似性, AVG 表示平均值函数。 $a.desc$

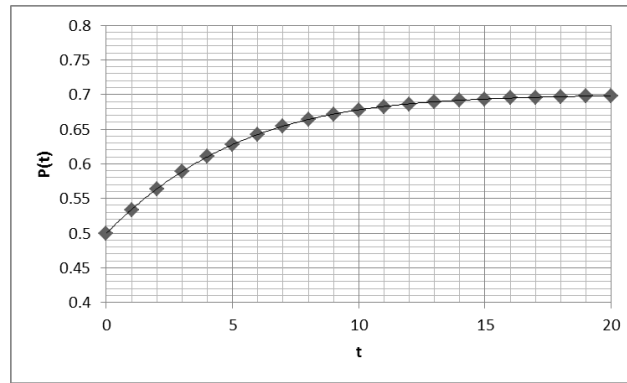


图 5.3 方法描述所占权重随方法参数多少的变化曲线

表示方法 a 的方法描述, $a.name$ 表示方法 a 的方法名。权重因子 w 一般取合适的常数值, 而权重因子 W 使用 logistic 函数计算:

$$W(t) = \frac{K * P * e^{r*t}}{K + P * (e^{r*t} - 1)} \quad \dots (5.8)$$

其中, t 为方法参数个数。具体实现时, 我们取 $K = 0.7, P = 0.5, r = 0.25$ 。如图 5.3, 这样取值可以保证方法描述的权重在 0.5 和 0.7 之间, 方法参数越多, 方法描述所占权重越大。

域签名由域所属类的类型和域的类型以及域的名称共同构成。我们按照公式计算域的相似性。

$$fieldsim(a, b) = \begin{cases} \frac{1}{2} + namesim(a, b) & \text{if } a.type = b.type \\ \frac{1}{2} + typesim(a, b) & \text{if } a.name = b.name \\ 0 & \text{otherwise} \end{cases} \quad \dots (5.9)$$

需要强调的是, 我们假定只有属于同一个类中的域才有必要计算相似性。属于不同类中的域即使类型名和域名都相同, 相似度也为 0。

5.2.2 源码注释提取

Java 项目源码中提供了两种标记 API 废弃的方法^[39]: 在 J2SE1.1 版本开始支持在代码注释中添加 javadoc 的 @deprecated 标签; J2SE5.0 以后也可以使用 @Deprecated 注解。程序开发人员在将 API 废弃时往往会在注释中添加一些文字用来解释这些 API 为什么被废弃或者废弃之后可以使用什么 API 进行替代。对于这些替换 API, Javadoc 规范推荐使用一些注解来标记, 其中 Javadoc1.1 版本推荐使用 @see 来标记替换 API, Javadoc1.2 以后推荐使用 @link 来标记。下面是从 Android Framework 源码中提取出的一些例子:

```

1  /**
2  * @deprecated Use {@link #getBlockCountLong()} instead.
3  */
4  @Deprecated
5  public int getBlockCount() {
6      return (int) mStat.f_blocks;
7  }

1  /**
2  * @deprecated This type is no longer used.
3  */
4  @Deprecated
5  public static final int IMAGE_ANCHOR_TYPE = 6;

1  /**
2  * .... To do that, use {@link WebViewDatabase#clearFormData}.
3  */
4  public void clearFormData() {
5      .....
6  }

```

这些源码注释中存在一定的模式，可以帮助我们挖掘 API 的演化。

我们使用 Eclipse 框架中提供的 JDT Core 库^[40] 解析 Android 应用框架项目源码。该库提供非常便利的方法供我们访问源码的抽象语法树 (AST 树)。通过遍历源码的 AST 树，寻找含有 deprecation 信息的节点，然后对这些节点使用我们编写好的正则表达式就可以进行匹配并提取其中含有的演化信息。部分提取模式见表 5.1。

| 模式 | 正则表达式 |
|------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| 替换模式 1 | replaceVerb{@link *}replaceSuffix |
| 替换模式 2 | @deprecated(\s)use(link)?replacements(\s)instead.\$ |
| 替换模式 3 | @deprecated(\s)replaced(\s)by(\s)replacements.\$ |
| 替换模式 4 | @deprecated(\s){@link}(preferred used instead).\$ |
| 替换模式 4 | @deprecated(\s)see {(.*)} |
| Unuse 模式 1 | @deprecated(\s)do(\s)not(\s)use(.\$) |
| Unuse 模式 2 | (no longer used obsolete no longer support ignored no longer needed not support no longer available unused no longer in use never used no longer called) |

表 5.1 用于提取源码注释中含有的演化信息的正则表达式

5.2.3 类层次关系图

通过比较同一个类型在前后两个版本的类层次关系图上的切片差异，可以启发我们识别出一部分类型的演化信息。例如图 5.4 中，图 5.4(a) 和 图 5.4(b) 是类 “android.graphics.Xfermode” 在 API Level 23 和 24 类层次关系图中的切片，从这两幅图中我们可以发现类 “android.graphics.PixelXorXfermode” 和

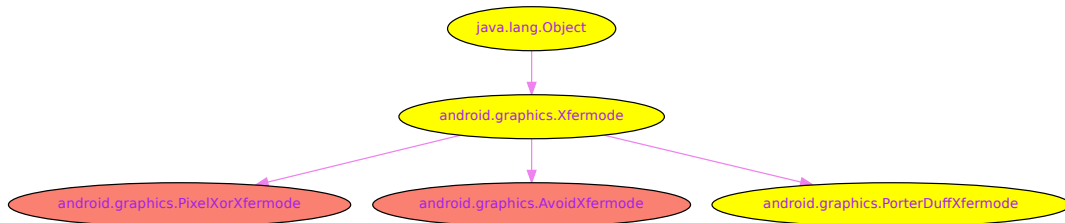
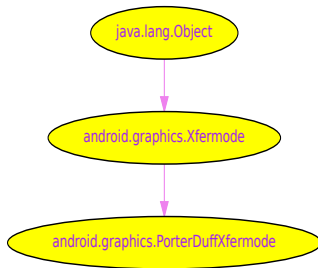
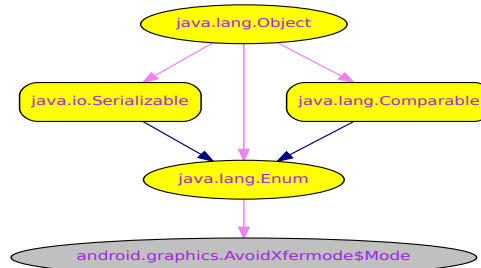
(a) SDK6.0 对 *xfermode* 图切片(b) SDK7.0 对 *xfermode* 图切片(c) SDK6.0 对 *AvoidXfermodes\$Mode* 图切片

图 5.4 通过 Class Hierarchy 识别废弃类

“android.graphics.AvoidXfermode”在演化过程中消失了，而这两个类在 Level 23 时已经被标为废弃，因此推测这两个类在这个演化过程中使被删除了。图5.4(c)中“android.graphics.AvoidXfermode\$Mode”是“android.graphics.AvoidXfermode”类的内部类，虽然两者之间继承关系差异很大，但当外部类被认定为删除类时，内部类也将被认定为删除类。

此外，通过类型使用信息可以识别出更多 API 演化信息。比如，在比对 API Level 23 和 24 中方法的演化时，我们发现方法“addOnRoutingChangedListener”和“removeOnRoutingChangedListener”在前后版本中方法名没有变，返回类型没有变，参数变量名没有变化，唯一变化的是第一个参数的类型。因此我们猜测第一个参数的类型在前后版本之间具有替换关系，即“android.media.AudioRecord\$OnRoutingChangedListener”和“android.media.AudioRouting\$OnRoutingChangedListener”之间具有替换关系，这个猜测在 Android API 变化文档中得到验证。

5.2.4 调用依赖分析

通过调用依赖关系的变化挖掘 API 的替换关系的基本思想是如果两个方法有替换关系，那么方法体中使用的 API 也很有可能具有替换关系。比如

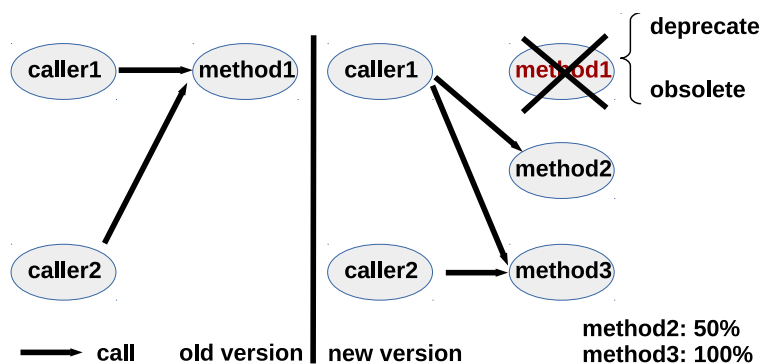


图 5.5 通过调用关系变化推断 method1 被 method3 替换

图5.5中, caller1 在旧版本中调用了 method1, caller2 也调用了 method1。到新版本后, caller1 不再调用 method1 而是改成调用 method2 和 method3, caller2 也不再调用 method1, 而是改成调用 method3。因此可以推断出在新版本中 method1 有可能被 method2 替换也有可能被 method3 替换, 但更有可能被 method3 替换。

通过仔细分析, 我们发现这种方法有一定的局限性。首先, 它只能为被调用者 (callee) 挖掘替换信息, 不能够为调用者挖掘替换信息 (caller), 因此无法对系统中从未被系统内部调用过的 API (称为顶层 API) 进行有效的挖掘。其次, 它要求有足够多这样的调用关系变化才能够进行挖掘。举例来说, 如果图5.5中旧版本里 method1 只被 caller1 调用, 到新版本中 caller1 不再调用 method1 而是改成调用 method2 和 method3, 这时我们可能就会推断出 method1 被 method2 和 method3 替换, 然而这样的推断是错误的, method1 实际上只是被 method3 替换。

为此, 我们考虑挖掘 Android Framework 以及 Android 系统原生应用程序。希望借此可以解决以往工作中不能够识别顶层 API 演化信息的问题, 并通过挖掘更多的调用依赖关系提升演化识别的准确性。

5.3 识别效果评价

本节我们对 Focus 的识别效果进行评价。为了便于人工比较, 我们选择 *API level* 相邻的两个版本进行比较 (Android 官方提供了相邻两个 *API level* 之间的差异, 我们的工具不限于识别相邻版本)。观察表3.2, 我们发现 Level 23 和 24 之间的 API 差异最大, 于是我们选择 android-6.0.1_r9 和 android-7.0.0_r7 两个 Android 版本进行分析。表5.2是我们工具分析

得到的结果。我们先对表中数据做简单解释，API 分为三种类型即类型（包括

| API 类型 | 共有未 废弃 | 共废 弃 | 变为 废弃 | 变回 普通 | 新添 | 删除 | 旧废 弃 | 替换 | 废弃不 支持 | 旧未 知 | 新未 知 |
|---------------|-----------|---------|----------|----------|----|----|---------|-----|-----------|---------|---------|
| Type | 3,309 | 122 | 37 | 0 | 79 | 3 | 2 | 46 | 14 | 0 | 272 |
| Method | 34,424 | 693 | 48 | 3 | 0 | 2 | 2 | 219 | 153 | 6 | 899 |
| Field | 16,078 | 640 | 31 | 0 | 0 | 0 | 0 | 179 | 73 | 5 | 579 |

表 5.2 Focus 工具识别结果展示

类和接口)，方法以及域。第二列数据表示 API 既在 Level 23 中也在 24 中并且没有用 `@Deprecated` 标记；第三列表示虽然两个版本中都存在，但两个版本中都被 `@Deprecated` 标记了；第四列表示在 level 23 时是普通 API，到 24 后被 `@Deprecated` 标记；第五列表示原先是被 `@Deprecated` 标记为废弃到 24 之后又变为普通 API；剩下的依次是新添加 API 数量、被删除的 API 数量、旧版本被废弃新版本找不到的 API 数量、工具识别出的替换关系数量、识别出的不再被支持了的 API 数量、旧版本未识别的 API 数量以及新版本还未识别出来的 API 数量。

对于我们的工具最后仍然识别不了其演化方式的 API，我们姑且认为是新添加或被删除的 API（表 5.2 最后两列）。我们对所有新版本中处于 `@Deprecated` 状态的 API 以及所有在旧版本中存在但在新版本找不到的 API 进行人工检查，判断它们是否存在替换关系，以及这些替换关系是否被我们的工具挖掘出来，挖掘出来的替换关系是对还是不对等等。最后我们分析得到的结果见表 5.3。

| | 挖掘出替换关系 | 正确的替换关系 | 总共替换关系 | 正确率 | 召回率 |
|--------|---------|---------|--------|--------|--------|
| Type | 47 | 43 | 92 | 91.49% | 46.74% |
| Method | 219 | 209 | 392 | 95.43% | 52.32% |
| Field | 179 | 179 | 515 | 100% | 34.76% |

表 5.3 人工分析替换关系计算工具识别正确率和准确率

从表 5.3 中数据可以看到我们的工具识别准确率很高，平均超过 90%，这主要是我们在识别时候策略比较保守导致的。另一方面也需要看到我们的工具召回率不是特别高，平均低于 50%。召回率不高的原因主要有以下几个：首先我们在工具实现时对于跨类型的替换关系没做处理，比如说一个域被

一个方法替换。另外一个原因是许多方法或域被移到一个类中去了，然而不论是文档还是注释都是让开发者去看类的实现并不告诉开发者直接替换关系，这些类在 SDK 中也没怎么被引用，导致工具挖掘时挖掘不到，比如 `android.provider.Contacts` 及其内部类中的许多域被废弃，文档和注释让看 `android.provider.ContactsContract` 这个类，我们到这个类里边也找不到对应的名字一样的域。对于这类我们也算在替换关系总数里边（有 187 个），这导致我们域的召回率特别低，对于这种情况，我们建议官方文档和源码注释等都有待进一步优化以提升对开发者的友好性。

5.4 讨论：研究中遇到的困难和挑战

在我们最初的设计中，我们试图使用应用程序开发框架中对 API 的调用依赖关系来挖掘 Android API 的替换关系。然而根据我们的研究，我们发现 Android SDK 中顶层 API 所占比例过高，使用调用依赖关系挖掘替换规则的方法难以起到作用。所谓顶层 API 就是在应用程序开发框架中未被引用过但允许外部应用调用的 API，这类 API 不能够通过调用依赖关系来挖掘替换关系。我们统计了一下当前主流 Android 版本中顶层 API 的比例，数据见表 5.4。

| API level | Type | Ratio | Method | Ratio | Field | Ratio |
|-----------|-------------|-------|---------------|-------|---------------|-------|
| 16 | 1,580/3,217 | 0.49 | 22,308/30,057 | 0.74 | 10,990/11,679 | 0.94 |
| 17 | 1,589/3,259 | 0.49 | 22,735/30,568 | 0.74 | 11,308/12,004 | 0.94 |
| 18 | 1,606/3,290 | 0.49 | 23,195/31,104 | 0.75 | 11,846/12,512 | 0.95 |
| 19 | 1,652/3,412 | 0.48 | 24,482/32,139 | 0.76 | 12,554/13,325 | 0.94 |
| 21 | 1,668/3,673 | 0.45 | 25,832/35,426 | 0.73 | 15,289/16,333 | 0.94 |
| 22 | 1,668/3,683 | 0.45 | 25,906/35,568 | 0.73 | 15,334/16,380 | 0.94 |
| 23 | 1,302/3,471 | 0.38 | 24,726/35,239 | 0.70 | 15,650/16,757 | 0.93 |
| 24 | 1,565/3,823 | 0.41 | 28,882/39,773 | 0.73 | 18,867/20,016 | 0.94 |
| 25 | 1,565/3,828 | 0.41 | 28,940/39,896 | 0.73 | 18,925/20,076 | 0.94 |
| 26 | 1,765/4,181 | 0.42 | 33,443/44,307 | 0.75 | 20,168/21,419 | 0.94 |
| 27 | 1,778/4,201 | 0.42 | 33,238/44,455 | 0.75 | 20,221/21,471 | 0.94 |

表 5.4 各版本 SDK 中顶层 API 比率

表中“/”前是顶层 API 数量，“/”后是该版本该类型 API 总数。通过表 5.4，我们发现除类型外，方法和域在 Android 应用框架内部被使用的很少。SDK 中有一半以上的 API 在 Android 系统内部未被使用过。SDK 中顶层域占比接近 94%，顶层方法占比平均 73%。而当我们利用 Android 应用框架中的调用依赖

关系对 SDK 中废弃 API 以及处于未知状态的 API 进行替换关系挖掘时，我们发现只能提取出 40 多条调用关系变化，调用关系变化数量太少以至于使用调用依赖关系进行挖掘基本上不可行。

按照原计划，我们希望从 Android 系统原生应用的版本变化中挖掘更多替换规则，根据我们的分析结果，我们发现 Android 系统应用使用 SDK 中在新版本被新废弃或新删除的 API 比例也很少，因此也难以挖掘。我们也曾试图从第三方 Android 应用中挖掘更多 API 替换规则，但结果令人失望。Android 应用几乎都做了版本兼容，没有办法通过调用依赖分析的方法来挖掘 API 的替换关系。

在研究的后期，我们试图从兼容处理模式中挖掘 API 演化，但基本上尝试失败。根据第 3 章的研究结论，我们发现 Android 支持库对 Android API 支持度很低，许多 API 的变化从支持库中挖掘不出来。另外，我们人工分析了一些兼容处理模式，我们发现简单的替换关系很少。要么是在新版本调用一个方法，在旧版本不做任何处理，要么是在旧版本用一段代码兼容新版本的一个方法调用。虽然看上去具有替换关系，但是通过软件自动提取，实现起来比较困难，首先不知道如何界定这两段代码是否具有替换关系，多个应用中对同一个新 API 的兼容处理方法可能并不相同。

从当前工具挖掘的效果来看，通过提取文档注释的方法是最行之有效的办法，根据我们的经验，绝大多数替换关系是通过从文档注释中提取出来的。文本相似度等其他方法也有一些效果，但总体来说效果不明显。

5.5 API 不兼容使用修复建议

基于 Focus 演化识别工具，我们为 Android SDK 所有版本中的 API 进行一个预处理。该预处理过程最终构建一个哈希映射，key 是 Android SDK 中的 API，value 是一个大小为 27 的数组，每一个数组元素表示该 API 在对应版本上的替换。如果在对应版本上没有替换且 API 不在那个版本中存在则对应位置为 null；如果没有替换但 API 本身在那个 SDK 版本中存在，则对应位置就是 API 本身。

结合第 4 章的工作，当 IctApiFinder 工具在报告某个 API 在哪些 SDK 版本上不存在时，工具会查看我们预处理得到的哈希映射。如果在对应版本上有替换 API，我们的工具会建议开发者使用替换 API，从而达到给出修复建议的效

```

1 // minSdkVersion: 22; targetSdkVersion 27.
2 public class MainActivity extends Activity {
3     private TextView mView;
4     protected void onCreate(Bundle bundle) {
5         ...
6         mView.startDragAndDrop(c, s, o, i); // [24,27]
7     }
8 }

```

修复建议: recommend using <boolean startDrag(...)> on level [22,23]

图 5.6 兼容修复建议示例

果。

图5.6是我们构造的一个兼容修复示例。该示例中第 6 行使用的 API 在 Level 24 以后添加到 SDK 中，但该 Android 应用支持的最低版本是 22，因此对该 API 的引用属于不兼容引用。当 IctApiFinder 在检测到该不兼容引用时会查询我们预分析得到的哈希映射结构，然后发现该 API 在版本 22 和 23 可以用 startDrag 方法进行替换，因此我们会建议开发者使用 startDrag 进行兼容修复处理。

5.6 相关工作

从我们前期论文调研结果来看，目前已有一些挖掘 API 演化相关的研究工作。根据这些论文重点强调的研究方法来看，大致可以分为两类：一类是以 AURA^[35] 为代表的使用调用依赖关系的方法挖掘 API 的替换关系；另外一类是以 HiMa^[41] 为代表的通过挖掘软件版本管理仓库的方法挖掘演化信息。这两类方法中都混合使用了文本相似性以及代码注释的自然语言处理等方法。

AURA 工具混合使用了文本相似度和调用依赖两种方法，其基本思想认为两个已知的替换方法对 (文中称为 anchor) 中调用的方法也很有可能具有替换关系。AURA 工具首先为新旧版本中已知替换方法对的被调用方法 (callee) 生成候选替换方法对，然后结合置信度和文本相似性计算得到新的替换关系，并将新生成的替换方法对加入到已知替换方法对集合中。算法不断迭代直到没有新的替换关系生成为止。AURA 能够找到一对一，一对多，多对一，多对多等替换关系。和之前的工作比较，它能够在损失较少精度的情况下达到 53.07% 的召回率 (recall)。然而这种方法的局限性在于它不能够为框架或软件库中顶层 API 即在库代码中从未被引用过的 API 挖掘出替换关系。根据我们

的研究，Android SDK 中顶层 API 占比特别高，使用这种调用依赖关系的方法挖掘 Android API 变化效果并不好。我们试图从 Android 系统应用程序中 API 的使用来减弱这种因顶层 API 过高带来的消极影响，但我们第 3 章的工作发现 Android 应用中使用到的 API 并非均匀分布，只有很少比例的 API 被广泛使用，绝大多数 API 很少被应用程序使用，因此这方面的努力也未能成功。

HiMa 使用一些自然语言处理的方法分析版本控制系统中处于两个发布版本之间所有相邻的小版本 (revision) 之间的提交 (commit) 注释信息，得到相邻版本 (revision) 之间的 API 变化；然后将所有相邻版本 (revision) 之间的 API 变化信息进行聚合得到两个大版本 (release) 之间 API 的变化信息。文中说 HiMa 比 AURA 计算更耗费时间，但在召回率和精确性方面优于 AURA。然而这种方法严重依赖提交时注释信息的质量。根据我们日常使用版本控制系统的经验，对于较为稳定成熟型的软件框架，由于每次变动较小，其 commit 信息相对比较详细。但是对于那些功能添加型或演化剧烈型的软件框架，即使 commit 信息比较详细也不可能覆盖所有变化，比如框架使用一个新类替换另一个类，在它的 commit 信息中可能只有“replace class A with class B”字样，一般不会详细到类中每一个方法和域的替换。因此我们认为 HiMa 可能不适合挖掘演化剧烈型的软件框架。我们的研究对象 Android 系统是一个更新异常频繁的软件系统，每一个版本之间 SDK 变动都很大，因此在调研初期，我们没有采用从 Android 版本管理库中挖掘演化信息的方法。后期，我们会再仔细研究通过这种方法提升工具的演化识别效果的可能性。

5.7 本章小结

本章开发了 Focus 工具用于挖掘 Android API 的版本演化，我们尝试了文本相似度、源码注释提取、调用依赖关系等多种挖掘方法，我们的工具能够识别的准确率非常高，但召回率还不够好。主要原因是 Android SDK 中顶层方法比例过高等导致文本相似度和调用依赖关系等方法挖掘效果显现不出来。有些替换关系在源码和文档中都介绍的不够详细，通过注释提取也没办法提取出对应替换关系。因此我们认为 Android 官方文档和源码注释还有待进一步优化。

本章最后通过使用我们开发的演化识别工具，我们能够为第 4 章检测出来的潜在缺陷提供一些有用的修复建议。

第 6 章 结束语

在 Android 智能手机正风靡全球的同时，Android 应用因为 API 兼容性方面的问题在开发，测试和维护方面正经历前所未有的困难和挑战。Android 系统快速的更新迭代导致 Android 生态碎片化问题严重，带来的不良结果便是 Android 应用开发不得不考虑对多个 Android 系统的兼容。不同 Android 版本间 API 的巨大差异使得应用开发人员难以搞清楚一个 API 可以使用的版本范围以及在新版本上可以使用什么 API 进行替代。对于测试人员来说，他们不得不在应用兼容的多个 Android 版本上进行逐个测试。

本文对 Android API 变化导致的应用兼容性问题进行了全面的研究，并提出了一种有效的检测方法。此外，我们还开发了一个自动识别 Android API 演化的工具。它不仅可以为不兼容引用的 API 提供修复建议还可以用于帮助 Android 开发者了解 Android API 变化，进而让他们开发出兼容性更好的应用程序。

6.1 主要成果

综合前面章节的介绍，本文的研究成果可概括为如下三点：

- 开展了一个关于 Android API 演化导致的兼容处理情况的经验研究。

我们从 Android 应用程序中常用兼容处理方式，Android 应用中与 API 相关的兼容性处理情况是否普遍，导致 Android 应用处理这类兼容问题的深层原因等多个角度做了大量细致的经验研究工作，我们发现了许多有意思的结论，这些结论对我们后续的工作有非常多的指导意义。

- 提出了一种检测 Android 应用中 API 不兼容性使用的方法。

基于之前的经验性研究工作，我们发现应用程序中处理 API 相关的兼容性的方式具有非常普遍的相似性。通过观察这些兼容处理方式，我们提出了一种检测 Android 应用中 API 不兼容使用的方法并开发了原型工具，实验效果很好。

- 开发了一个自动识别不同版本间 API 演化的工具。

最后，我们开发了一个自动识别不同 Android 版本间 API 变化的工具，该工具可以有效帮助开发者了解 Android 不同版本之间 API 的演化，进而开发出兼容性更好的 Android 应用程序。基于该工具，我们能够为一些 API 不兼容使用缺陷提出修复建议。

6.2 未来工作

Android 应用中不兼容问题还很严重，但这个方向上的研究目前还很少，因此也还有很多工作要做。在后续的研究工作中，我计划从检测工具的性能优化，API 不兼容使用的自动确认（复现）和自动修复这三个方面继续探索。

在第4章，我们提出检测兼容性问题的方法，同时开发了工具。但由于研究时间的限制，我们并没有在性能优化上做努力。然而，我们的方法中真正关心的仅仅是 ICFG 图中和特定分支语句相关的节点。对其他节点，我们或许可以在不影响数据流的前提下进行缩点，比如说对不含有我们关心的分支语句的方法，我们可以在 ICFG 图上用一个点表示，这将大大缩小计算规模，有助于提升程序计算性能。

我们的检测工具不仅能够报告出哪里的 API 使用可能是不兼容的使用，还能够使用追踪器 (Tracer) 报告所有从程序入口到使用点的可能路径。然而，我们发现人工去确认缺陷报告是否是真实缺陷是一件很困难的事情。能否构造输入指导模拟器自动沿着追踪器给的可能路径执行进而复现和确认是否是真实缺陷是一件有价值也很有挑战的工作。我希望后续有时间可以在这方面进行探索。

当缺陷被确认为一个真实缺陷时，理所当然的，我们希望能够对缺陷进行自动修复。自动修复需要知道在对应版本上可以用哪些 API 替换不兼容使用的 API。这个工作我们在第 5 章已经做了初步探究，后续还会继续研究，争取在召回率和准确率上进一步提升该工具的效果。而要做自动修复，我们需要知道有哪些种修复方式以及如何进行修复。我们在第3章最后对 Android 应用以及支持库中常用的兼容模式做了一些经验性了解，下一步工作的重点便是如何进行自动修复。

参考文献

- [1] IDC: Smartphone OS Market Share. [J]. URL <https://www.idc.com/promo/smartphone-market-share/os>, 2018.
- [2] Joorabchi M E, Mesbah A, Kruchten P. Real Challenges in Mobile App Development[C]// 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. 2013: 15–24. doi: 10.1109/ESEM.2013.9.
- [3] Ruiz I J M, Nagappan M, Adams B, et al. Understanding reuse in the Android Market[C]// 2012 20th IEEE International Conference on Program Comprehension (ICPC). 2012: 113–122. doi: 10.1109/ICPC.2012.6240477.
- [4] Mojica I J, Adams B, Nagappan M, et al. A Large-Scale Empirical Study on Software Reuse in Mobile Apps[J]. IEEE Software, 2014, 31(2): 78–86. doi: 10.1109/MS.2013.142. issn: 0740-7459.
- [5] Syer M D, Adams B, Zou Y, et al. Exploring the Development of Micro-apps: A Case Study on the BlackBerry and Android Platforms[C]// 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation. 2011: 55–64. doi: 10.1109/SCAM.2011.25.
- [6] McDonnell T, Ray B, Kim M. An Empirical Study of API Stability and Adoption in the Android Ecosystem[C]// 2013 IEEE International Conference on Software Maintenance. 2013: 70–79. doi: 10.1109/ICSM.2013.18.
- [7] Wei L, Liu Y, Cheung S C. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps[C]// 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). 2016: 226–237.
- [8] Linares-Vásquez M, Bavota G, Di Penta M, et al. How do api changes trigger stack overflow discussions? a study on the android sdk[C]// Proceedings of the 22nd International Conference on Program Comprehension. ACM. 2014: 83–94.
- [9] Brahler S. Analysis of the android architecture[J]. Karlsruhe institute for technology, 2010, 7(8).
- [10] Li L, Bissyandé T F, Le Traon Y, et al. Accessing inaccessible android apis: An empirical study[C]// Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on. IEEE. 2016: 411–422.
- [11] Meier R. Professional Android 4 application development[M]. John Wiley & Sons, 2012.
- [12] Wu D, Liu X, Xu J, et al. Measuring the declared SDK versions and their consistency with API calls in Android apps[C]// International Conference on Wireless Algorithms, Systems, and Applications. Springer. 2017: 678–690.
- [13] Improve Your Code with Lint. [J]. URL <https://developer.android.com/studio/write/lint.html>, 2018.

- [14] Lint API Check. [J]. URL <http://tools.android.com/recent/lintapicheck>, 2018.
- [15] Vallee-Rai R, Hendren L J. Jimple: Simplifying Java bytecode for analyses and transformations[J]. 1998.
- [16] Nielson F, Nielson H R, Hankin C. Principles of program analysis[M]. Springer, 2015.
- [17] Smaragdakis Y, Balatsouras G, et al. Pointer analysis[J]. Foundations and Trends® in Programming Languages, 2015, 2(1): 1–69.
- [18] Longest common subsequence problem. [J]. URL https://en.wikipedia.org/wiki/Longest_common_subsequence_problem, 2018.
- [19] Levenshtein distance. [J]. URL https://en.wikipedia.org/wiki/Levenshtein_distance, 2018.
- [20] Han D, Zhang C, Fan X, et al. Understanding android fragmentation with topic analysis of vendor-specific bugs[C]// Reverse Engineering (WCRE), 2012 19th Working Conference on. IEEE. 2012: 83–92.
- [21] Mutchler P, Safaei Y, Doupé A, et al. Target fragmentation in Android apps[C]// Security and Privacy Workshops (SPW), 2016 IEEE. IEEE. 2016: 204–213.
- [22] Allix K, Bissyandé T F, Klein J, et al. Androzo: Collecting millions of android apps for the research community[C]// Proceedings of the 13th International Conference on Mining Software Repositories. ACM. 2016: 468–471.
- [23] Free and Open Source Android App Repository. [J]. URL <https://f-droid.org>, 2018.
- [24] Vallée-Rai R, Co P, Gagnon E, et al. Soot: A Java bytecode optimization framework[C]// CASCON First Decade High Impact Papers. IBM Corp. 2010: 214–224.
- [25] Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability[C]// Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM. 1995: 49–61.
- [26] Lhoták O, Hendren L. Scaling Java Points-to Analysis Using Spark[C]// International Conference on Compiler Construction. Springer. 2003: 153–169.
- [27] Arzt S. Static data flow analysis for android applications[D]. Technische Universität, 2017.
- [28] 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 Sigplan Notices, 2014, 49(6): 259–269.
- [29] Bodden E. Inter-procedural data-flow analysis with ifds/ide and soot[C]// Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis. ACM. 2012: 3–8.
- [30] Smaragdakis Y, Bravenboer M. Using Datalog for fast and easy program analysis[G]// Datalog Reloaded. Springer, 2011: 245–251.
- [31] Abiteboul S, Hull R, Vianu V. Foundations of databases: the logical level[M]. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [32] Green T J, Aref M, Karvounarakis G. Logicblox, platform and language: A tutorial[G]// Datalog in Academia and industry. Springer, 2012: 1–8.

-
- [33] Gradle build tool. [J]. URL <https://gradle.org>, 2018.
- [34] Groovy: A multi-faceted language for the Java platform. [J]. URL <http://groovy-lang.org/>, 2018.
- [35] Wu W, Guéhéneuc Y.-G, Antoniol G, et al. Aura: a hybrid approach to identify framework evolution[C]// Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. ACM. 2010: 325–334.
- [36] Yu P, Yang F, Cao C, et al. API Usage Change Rules Mining based on Fine-grained Call Dependency Analysis[C]// Proceedings of the 9th Asia-Pacific Symposium on Internetware. ACM. 2017: 10.
- [37] Dig D, Johnson R. How do APIs evolve? A story of refactoring[J]. Journal of Software: Evolution and Process, 2006, 18(2): 83–107.
- [38] Camel case. [J]. URL https://en.wikipedia.org/wiki/Camel_case, 2018.
- [39] Brito G, Hora A, Valente M T, et al. On the use of replacement messages in API deprecation: An empirical study[J]. Journal of Systems and Software, 2018, 137: 306–321.
- [40] JDT Core Component. [J]. URL <https://www.eclipse.org/jdt/core/index.php>, 2018.
- [41] Meng S, Wang X, Zhang L, et al. A history-based matching approach to identification of framework evolution[C]// Software Engineering (ICSE), 2012 34th International Conference on. IEEE. 2012: 353–363.
- [42] Android Open Source Project. [J]. URL <https://source.android.com/>, 2018.
- [43] Alll B, Tumbleson C. Dex2jar: Tools to work with android. dex and java. class files[J]. Oceau D, Jha S, McDaniel R Retargeting,
- [44] Dupuy E. JD-Gui: Yet another fast java decompiler[J]. URL <http://java.decompiler.free.fr>, 2012.
- [45] Apache Ant. [J]. URL <https://ant.apache.org/>, 2018.
- [46] Apache Maven Project. [J]. URL <https://maven.apache.org/>, 2018.
- [47] Ivy: The agile dependency manager. [J]. URL <http://ant.apache.org/ivy/>, 2018.
- [48] Jasmin: an assembler for the Java Virtual Machine. [J]. URL <http://jasmin.sourceforge.net/>, 2018.
- [49] FlowDroid Data Flow Analysis Tool. [J]. URL <https://github.com/secure-software-engineering/FlowDroid>, 2018.
- [50] Jsoup: Java HTML Parser. [J]. URL <https://jsoup.org/>, 2018.
- [51] JSON In Java. [J]. URL <https://mvnrepository.com/artifact/org.json/json>, 2018.

附录 Android 开发中最常使用 API

在第3章的工作中，我们统计了 Android API 在我们所选择的 4,697 个 Android 应用中被累计引用次数，此处附录的是我们人工分析的引用数量超过 100 万次的 API。

| Android API | 引用次数 |
|----------------------------------------------------------------------------------------------|---------|
| <android.os.Parcel: void recycle()> | 7183207 |
| <android.os.Parcel: void writeInt(int)> | 2382740 |
| <android.os.Parcel: void enforceInterface(java.lang.String)> | 1872116 |
| <android.os.IBinder: boolean transact(int,android.os.Parcel,android.os.Parcel,int)> | 1844559 |
| <android.os.Parcel: void writeInterfaceToken(java.lang.String)> | 1844346 |
| <android.os.Parcel: void writeNoException()> | 1756468 |
| <android.os.Parcel: void readException()> | 1731765 |
| <android.os.Parcel: void writeString(java.lang.String)> | 1024844 |
| <android.os.Parcel: void writeStrongBinder(android.os.IBinder)> | 845290 |
| <android.widget.TextView: void setText(java.lang.CharSequence)> | 335285 |
| <android.util.Log: int d(java.lang.String,java.lang.String)> | 281945 |
| <android.os.Bundle: void <init>()> | 279608 |
| <android.util.Log: int e(java.lang.String,java.lang.String)> | 268926 |
| <android.os.Build\$VERSION: int SDK_INT> | 260796 |
| <android.database.Cursor: void close()> | 245250 |
| <android.util.Log: int w(java.lang.String,java.lang.String)> | 232633 |
| <android.content.Intent: android.content.Intent putExtra(java.lang.String,java.lang.String)> | 211503 |
| <android.os.Handler: boolean post(java.lang.Runnable)> | 210300 |
| <android.os.Bundle: void writeToParcel(android.os.Parcel,int)> | 180108 |
| <android.os.Bundle: android.os.Parcelable\$Creator CREATOR> | 172135 |
| <android.app.Activity: void finish()> | 155186 |
| <android.content.Intent: void <init>(java.lang.String)> | 146833 |
| <android.view.View: void setVisibility(int)> | 141904 |
| <android.os.Binder: void <init>()> | 141175 |
| <android.os.Binder: void attachInterface(android.os.IInterface,java.lang.String)> | 138499 |
| <android.content.Intent: void <init>(android.content.Context,java.lang.Class)> | 135474 |
| <android.os.Bundle: void putString(java.lang.String,java.lang.String)> | 134658 |
| <android.util.Log: int i(java.lang.String,java.lang.String)> | 126101 |
| <android.util.Log: int e(java.lang.String,java.lang.String,java.lang.Throwable)> | 125289 |
| <android.app.Activity: void startActivity(android.content.Intent)> | 100248 |
| <org.json.JSONObject: org.json.JSONObject put(java.lang.String,java.lang.Object)> | 493732 |
| <org.json.JSONObject: void <init>()> | 210186 |
| <org.json.JSONObject: void <init>(java.lang.String)> | 109274 |

| | |
|-----------------------------------------------------------------------------------|---------|
| <java.io.File: void <init>(java.lang.String)> | 149928 |
| <java.io.InputStream: void close()> | 116894 |
| <java.io.IOException: void <init>(java.lang.String)> | 202081 |
| <java.lang.AssertionError: void <init>()> | 122774 |
| <java.lang.Enum: void <init>(java.lang.String,int)> | 200053 |
| <java.lang.IllegalArgumentException: void <init>(java.lang.String)> | 844560 |
| <java.lang.IllegalStateException: void <init>(java.lang.String)> | 517677 |
| <java.lang.NullPointerException: void <init>(java.lang.String)> | 116946 |
| <java.lang.Object: void <init>()> | 6534525 |
| <java.lang.RuntimeException: void <init>(java.lang.String)> | 191600 |
| <java.lang.StringBuffer: java.lang.StringBuffer append(java.lang.String)> | 256711 |
| <java.lang.StringBuffer: void <init>()> | 118430 |
| <java.lang.StringBuilder: java.lang.StringBuilder append(char)> | 266744 |
| <java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)> | 1125183 |
| <java.lang.StringBuilder: void <init>()> | 3978428 |
| <java.lang.StringBuilder: void <init>(int)> | 264384 |
| <java.lang.StringBuilder: void <init>(java.lang.String)> | 892139 |
| <java.lang.System: void arraycopy(java.lang.Object,int,java.lang.Object,int,int)> | 344508 |
| <java.lang.Throwable: void printStackTrace()> | 186074 |
| <java.lang.UnsupportedOperationException: void <init>()> | 112997 |
| <java.lang.UnsupportedOperationException: void <init>(java.lang.String)> | 132520 |
| <java.util.ArrayList: boolean add(java.lang.Object)> | 535847 |
| <java.util.ArrayList: void <init>()> | 825019 |
| <java.util.ArrayList: void <init>(int)> | 174437 |
| <java.util.ArrayList: void <init>(java.util.Collection)> | 115450 |
| <java.util.concurrent.locks.Lock: void unlock()> | 180329 |
| <java.util.HashMap: java.lang.Object put(java.lang.Object,java.lang.Object)> | 881172 |
| <java.util.HashMap: void <init>()> | 532329 |
| <java.util.HashSet: void <init>()> | 186610 |
| <java.util.List: boolean add(java.lang.Object)> | 829726 |
| <java.util.Locale: java.util.Locale US> | 101739 |
| <java.util.Map: java.lang.Object put(java.lang.Object,java.lang.Object)> | 1198618 |
| <java.util.Set: boolean add(java.lang.Object)> | 498780 |

附录 研究技术报告

我们在研究过程中，做了大量基础性的工作，使用了许多有用的开发工具，我们详细记录于此，供以后回忆或他人借鉴之用。

编译 AOSP 项目

AOSP^[42] 是 Android Open-Source Project 的缩写，它包含 Android 系统的源码，应用框架，开发工具等几乎 Android 生态下所有的东西。编译 Android SDK，获取应用框架及其支持库以及提取 Android 原生系统应用等都需要编译 AOSP。

下载 AOSP 项目需要使用 repo 脚本，该脚本实际上是 Android 官方提供的一个 shell 脚本，它对代码版本管理工具 Git 进行封装，从而有效管理 AOSP 项目。从本质上来说，一个版本的 AOSP 包含上百个独立的由 Git 管理的子项目构成。这些子项目的地址和对应版本信息存储在一个叫 default.xml 的文件中。repo 命令通过解析该 xml 文件就可以批量管理这上百个子项目。

Android 官网有详细的指导说明如何一步一步下载和编译 AOSP 项目。然而，不同版本的 AOSP 项目编译的环境和编译方法都有一些差异。这些差异给我们的编译工作带去了特别多的困难。为了在下一次编译 AOSP 项目时，能够让我们的编译工作不再那么麻烦，我们创建了一个叫 SDKsBuilder^① 的项目。在该项目中，我们编写了许多 shell 脚本程序，如图B.1，它能够从 Android 官网或者指定的 AOSP 镜像处自动下载 AOSP 源码，并自动对其进行编译，最后提取 SDK 和 framework 以及系统内置应用程序的 APK 文件等我们研究中需要的材料。稍后，你将可以从 Bitbucket 网站下载和使用该工具。

使用 Doop 构建项目基本输入

在我们的研究工作中，无论前期的经验性研究，对 API 演化导致的兼容性检测研究还是最后对 API 演化识别研究都离不开对 Android SDK 中 API 信息的提取和分析。为了快速获得 SDK 中含有的 API 信息，我们使用了 Doop

① <https://bitbucket.org/DongjieHe/sdksbuilder>

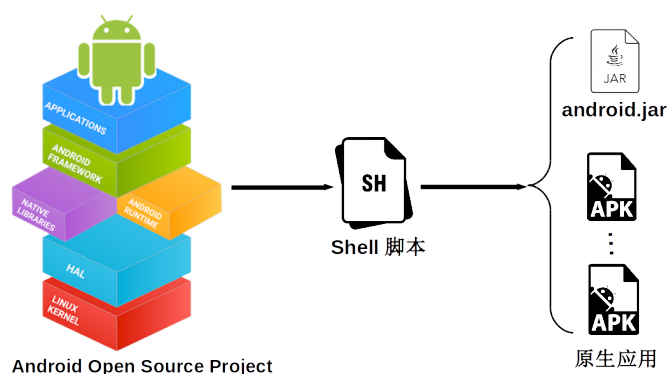


图 B.1 自动下载并编译 AOSP 项目

工具。如图B.2，借助 Doop 工具可以对已经编译好的 SDK 包和 APK 文件中的 API 信息进行提取，最后将提取得到的信息生成到 Datalog 数据库中，使用 Datalog 引擎工具可以非常便利的从数据库中直接读取我们需要的 API 信息。

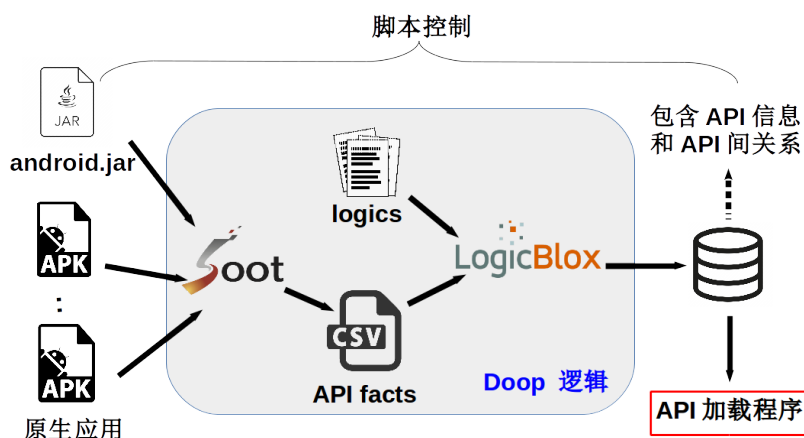


图 B.2 使用 Doop 提取 API 信息

Doop^[30] 是一个优秀的 Java 程序分析框架。在它的内部，它使用 Soot^[24] 工具从 jar 包中提取程序信息，并使用 Datalog^[31] 语言编写程序分析逻辑，最后它使用 Datalog 引擎（实际上是基于 LogicBlox^[32] v3 引擎修改的可用于学术研究使用的一个 Datalog 引擎，叫做 PA-Datalog^①）对程序进行分析，程序分析结果最终保存在普通文件或数据库中。

我们需要对不同版本的 Android 项目都使用 Doop 提取 API 信息，或者从 AOSP 源码中获得其他有用信息（例如 Frameworks 源码等）来构建我们研究项目的输入。为此，我们创建了 Formatter^② 项目，该项目中包含许多 shell 脚本工具，可以自动化的完成对编译 AOSP 后提取得到的 SDK 等进行格式化工作，使

① <http://snf-705535.vm.okeanos.grnet.gr/agreement.html>

② <https://bitbucket.org/DongjieHe/formatter>

得最终结果能够为我们的研究项目直接使用。不久，你也将可以从 Bitbucket 网站下载和使用该工具。

在构建 `Formatter` 项目时，我们使用了一些非常有用的工具，在此有必要介绍其中几个。首先要介绍的一个工具是 `dex2jar`^[43]。顾名思义，该工具的功能是将 `dex` 格式的文件转换成 `jar` 格式的文件。`Dex` 文件是 `Dalvik Executable files` 的缩写形式，它是 `Android` 虚拟机上可执行程序，包含应用程序的所有执行指令。在 `Android` 系统上除了系统应用程序之外，很多文件如 `Frameworks` 中很多包最终都将被编译成 `dex` 文件格式，但这种文件格式一般很难直接看到里边有哪些 `class` 文件，这给研究工作带来不便，使用 `dex2jar` 工具将其转换成 `jar` 包便可以查看包中都含有哪些 `class` 文件了。

另一个值得推荐的工具是 `jd-gui`^[44]。该图形工具可以对 `Java` 字节码进行反编译，对我们阅读和理解 `jar` 包有非常大的帮助作用。

使用 `Gradle` 构建研究项目

在我们的研究工作过程中，我们使用了 `Gradle`^[33] 作为我们的项目构建工具。`Gradle` 是一个基于 `Apache Ant`^[45] 和 `Apache Maven`^[46] 概念的项目自动化构建工具，兼具二者之优势。它使用一种基于 `Groovy`^[34] 的特定领域语言来声明项目设置。通过 `Gradle`，可以方便的管理项目依赖，测试，打包，部署，发布等等各种项目管理中的问题，极大便利了我们的程序开发过程。

`Gradle` 可以构建单个项目也可以构建具有多个子项目的大项目。每个项目称为一个 `Project`，对应一个 `build.gradle` 文件。每个 `Project` 通常由多个 `Task` 构成，用户可以自己在 `build.gradle` 文件中自定义一些自己需要使用的 `Task`。`Task` 其实是一组被顺序执行的 `Action` 对象构成，类似 `Java` 中的方法。

`Gradle` 构建项目的过程通常可以分为三个阶段，第一个是初始化阶段 (`initialization`)。在该阶段，构建工具首先会分析根目录下的 `settings.gradle` 文件，确定哪些项目参与构建过程，然后为这些要参与构建的项目创建 `Project` 对象实例。这些 `Project` 对象实例构成一棵层级关系的树结构。第二个阶段是配置阶段 (`configuration`)。在该阶段，`Gradle` 会去分析每个参与构建的 `Project` 项目对应的 `build.gradle` 文件，解析依赖，创建并配置所有的 `Task`，并且分析这些 `Task` 之间的依赖关系。`Task` 之间的依赖关系必须是一个有向无环图。

第三个阶段是执行阶段 (Execution)。对于每一个需要执行的 Task，Gradle 会分析其依赖，然后按照依赖拓扑序依次执行每一个 Task。这些 Task 可以是编译，打包，测试等，也可以是用户自己定义的 Task。

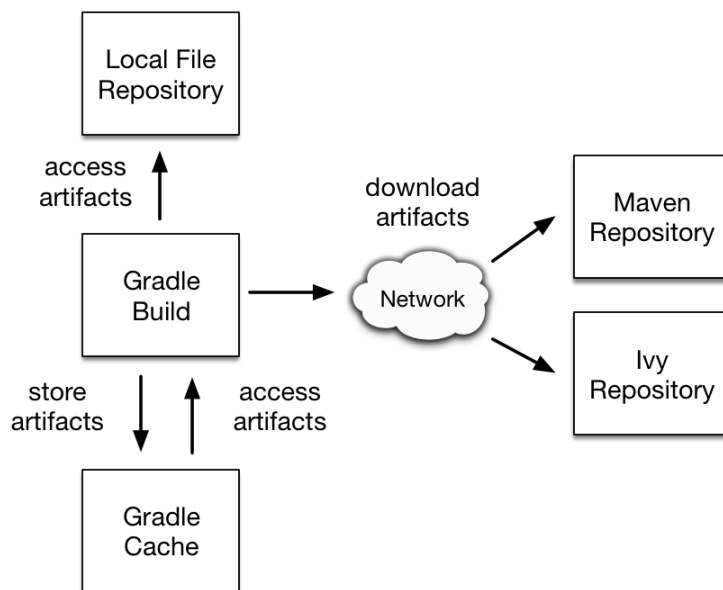


图 B.3 Gradle 依赖管理

个人觉得 Gradle 最大的特色是依赖管理。通常一个 Project 会依赖许多外部库或者其他工程中的文件，被依赖的文件很有可能又会依赖许多其他库中的文件。人工管理这些项目之间的依赖关系非常复杂且容易出错。在 Gradle 构建工具中，它提供了非常强大的依赖管理功能，能够自动帮助项目管理所有的依赖。如图B.3所示，在项目的 `build.gradle` 中只要配置好项目依赖哪些模块或者外部库以及从哪些地方去搜索这些依赖，Gradle 就会自动帮助项目解析依赖。这些存储模块或外部库的地方称为仓库，比较著名的仓库有 Apache Maven 仓库^[46] 以及 Apache Ivy 仓库^[47]。用户也可以构建本地仓库。Gradle 将下载的这些依赖模块或库存储在 Gradle Cache 中，以便下次构建项目时直接使用。在构建项目的配置阶段，这些依赖被包装成 Configuration 对象，放置在 Project 对象中。

在我们的项目研究过程中，我们使用了 Soot^[24]，Jasmin^[48]，Heros^[29]，soot-inflow^[49]，jsoup^[50]，JDT Core^[40] 以及 JSON^[51] 等依赖模块。通过使用 Gradle 构建我们的项目，这些依赖管理起来特别方便。此外，著名的 Java 开发 IDE Eclipse 已经集成了 Gradle 构建项目的功能，两者结合极大的促进了我们的开发效率。

致 谢

论文每写及此，内心总是既兴奋又激动。因为写到这里意味着论文工作将要完成了，忙碌又辛苦的生活可以暂时告一段落了。回想整个论文工作过程，许许多多片段在脑海回放，太多感情一时间不知如何表达。有论文踌躇不前的焦灼，有被老师们否定后内心的痛苦，有熬夜半天最后还是解决不了 bug 的失落，当然也有新 idea 出现在脑子里时的兴奋。

完成本论文工作非一人之力。在此，我要一一感谢对我论文工作有过帮助的人们。

首先，衷心感谢我的导师李炼老师。从调研，选题，论文研究到最后撰写，李老师给了我非常多的指导和帮助，包括引导我从哪些方面去探索去尝试，安排师弟们帮我一起做这个项目的的工作等等。在我论文工作一筹莫展的时候，李老师给予我更多的是他的耐心，和对我的鼓励和肯定。在此，再次向李老师表示感谢。

其次，感谢郑恒杰师弟和王蕾大师兄。郑恒杰是武汉大学本科大四学生并且准备来咱组读研。李老师让他跟着我一起在做他的本科毕业设计。在我的整个论文工作过程中有许多苦力活都让他在帮我做，包括第3章的实验，第4章缺陷的人工分析以及第5章人工计算演化识别的召回率和准确率等。如果没有他的帮助，论文中很多工作可能都没有时间开展，因此非常感谢他的帮助。关于检测 Android 应用中 API 不兼容使用部分的工作，在有了 idea 之后，我是在王蕾大师兄的帮助和指导下才得以快速的完成工具原型的实现。王蕾师兄即将博士毕业，他的研究方向主要是使用静态程序分析方法检测 Android 中隐私泄漏问题，因此他对 IFDS 算法以及 infoflow 框架等都非常熟悉，在他的帮助下，我那部分的工作进展非常顺利，非常感谢大师兄的帮助。

还要特别感谢的是冯晓兵老师。他在我的论文开题和中期报告上给了我特别多的提问和建议。冯老师一直关心的一个问题是我的研究工作到底有什么用。是的，一个没有意义的工作又有什么研究价值呢？我一直觉得论文中期报告时的评价结果是给我的工作把把脉，打打预防针，这比鲜花和掌声要有用的多。在此后的研究工作中，我一直在思考如何让自己的研究工作对别人有意

义。我想这是我能够想到去做 API 兼容性检测的一个重要原因。

此外，要感谢李昊峰和刘晨，他们曾短暂参与过我的研究工作中。感谢李广威和陆杰，我们时常在去食堂吃饭的路上交流论文研究工作上的问题，时则会碰撞出思想的小火花来，在服务器维护方面，他俩也给了我不少帮助。也要感谢组里乃至所里其他同学，他们都特别优秀，在他们身上，我学到了也获得了很多东西。我想，这些东西才是未来我离开计算所之后最大的人生财富。

最后的最后，我要感谢我的父母和家人，感谢我的女朋友。感谢他们的关爱，理解，支持和陪伴。在以后的生活中，我会一如既往的努力，不辜负他们对我的期许和厚爱。

何冬杰，2018 年 4 月于北京