中国科学技术大学

硕士学位论文

Android 应用间能力泄露漏洞利用的自动化生成

作者姓名： 周明松

学科专业： 计算机应用技术

导师姓名： 曾凡平 副教授

完成时间： 二〇一九年五月三十日

University of Science and Technology of China

A dissertation for master's degree

Automatic Generation of Capability Leaks' Exploits for Android Applications

Author: Mingsong Zhou

Speciality: Technology of Computer Application

Supervisor: Assoc. Prof. Fanping Zeng

Finished time: May 30, 2019

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____ 签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

☑ 公开 ☐ 保密（____ 年）

作者签名：_____ 导师签名：_____

签字日期：_____ 签字日期：_____

摘　要

Android 应用间能力泄露漏洞是一种危害性非常大的 Android 组件间漏洞，其他应用或者攻击者可以利用其非法地使用该应用的特殊能力，因此对 Android 安全造成了巨大的危害。目前 Android 组件间安全工作更多的是关注 Android 组件间漏洞的检测方法，而忽略了漏洞利用的重要性。漏洞利用可以帮助开发者确认 bug，减少漏洞的分析时间。因此本文先后提出了两个 Android 应用间能力泄露漏洞利用的自动化生成工具，两者可以自动化检测 Android 应用间能力泄露漏洞，并生成触发漏洞的利用，帮助开发者确认 bug。

（1）基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成。将符号执行技术引入到 Android 应用间能力泄露漏洞的检测工作中，精确计算每条可能触发漏洞路径的路径条件；并根据能力泄露漏洞检测工作的问题特性，从多个方面对本文的符号执行进行优化，使符号执行适用于实际应用的能力泄露漏洞检测工作；使用符号执行工具检测豌豆荚各类别 611 个 APP，共发现 16 种共 6566 个能力泄露缺陷；根据本文实验评估可知，本文符号执行技术可以产生高精度的测试用例，平均路径条件计算准确率为 85.3%；相比同类型工作 IntentFuzzer，本文符号执行工具漏报率要好 14.97%；符号执行工具的平均 APP 分析时间为 7.16 minutes，满足实际应用场景的时间需求。

（2）基于动态反馈的 Android 应用间能力泄露漏洞利用的自动化生成。通过对 Android 应用插桩，获取程序的动态运行信息产生测试用例。相比静态分析提取获取路径条件，其可以获取更精确有效的路径条件信息；同时使用上下文敏感、流敏感的过程间数据流分析精确找到每一个包含路径条件信息的变量和插桩位置点，保证动态获取路径条件信息的完备性；并针对获取的路径条件信息进行变异，提高代码覆盖率；使用动态反馈工具检测豌豆荚各类别 611 个 APP，共发现 17 种 7143 个能力泄露缺陷；相比同类型工作 IntentFuzzer，漏报率要好 19.39%，相比本文第一个工作漏报率要好 5.30%；动态反馈工具的平均 APP 分析时间为 9.04 minutes，满足实际应用场景的时间需求。

关键词：安卓应用间能力泄露漏洞；漏洞利用；组件间通信；符号执行；动态反馈测试

ABSTRACT

The capability leak of Android applications is one kind of serious vulnerability. It causes other apps or attackers to leverage its capabilities to achieve their illegal goals. So it does great harm to Android security. At present, the security work of Android components is more concerned with the detection of vulnerabilities between Android components, ignoring the importance of exploits to confirm bugs for developers. Therefore, in this dissertation, we propose two tools which can automatically generate capability leaks' exploits of Android applications. They can help developers confirm bugs.

(1) In our first tool, we utilize symbol execution technology to accurately compute the path conditions of each path that may trigger a vulnerability. And we utilize multiple optimization methods to optimize the symbol execution process based on the characteristics of the capability leak for Android applications, which make our tool applicable for practical apps. By applying our tool to 611 popular applications from Wandoujia, our first tool detects 6,566 capability leaks of 16 kinds of permissions. According to our experiment, our symbolic execution can produce high-precision test cases, the average path condition accuracy is 85.3%. Compared with similar work IntentFuzzer, our tool improves 14.97% in false negative rate. And the average analysis time per APP is 7.16 minutes, which can meet the time requirements of the actual application scenario.

(2) In our second tool, we utilize instrumentation technology to get runtime information, and then utilize the runtime information to generate test cases. Compared with static analysis tool, this way can get more valid path conditions. At the same time, we utilize a context-sensitive, flow-sensitive inter-procedural data-flow analysis to accurately find instrumentation points and variables that contain path condition information. It ensures the completeness of path condition information that obtained from dynamic test. We also mutate path conditions to improve code coverage. By applying our tool to 611 popular applications from Wandoujia, our tool detects 7,143 capability leaks of 17 kinds of permissions. Compared with IntentFuzzer and our first tool, our second tool improves 19.39% and 5.30% respectively in false negative rate. And the average analysis time per APP is 9.04 minutes, which can meet the time requirements of the actual application scenario.

Key Words: capability leak, exploit, inter-component communication, symbolic execution, dynamic-feedback test

目　录

插 图 清 单

表 格 清 单

算 法 清 单

第 1 章　绪　　论

本章首先介绍了本文 Android 应用间能力泄露漏洞利用的自动化生成选题背景及研究意义：Android 系统占据了超高的市场份额，以及 Android 应用存在的严重能力泄露问题。然后根据主要相关工作的优势和不足，以及软件工程的实际需求，提出本文工作的主要研究内容，即自动化生成 Android 能力泄露的利用，帮助开发者寻找确认 bug, 缩短开发周期。本章最后列出了论文组织结构。

1.1　选题背景及研究意义

由于移动互联网时代的迅速发展，智能手机越来越普及，智能手机的使用涉及到生活的方方面面，例如：网上购物，在线聊天等。其中 Android 手机占据了智能手机的主要市场份额，根据 IDC 智能手机市场份额最新报告，2018 年第三季度 86.8% 的智能手机配备了 Android 系统[1]。由于 Android 系统的开放性等原因，Android 安全问题日益突出。据 2018 年 360 公司发布的 Android 恶意软件专题报告[2]，360 互联网安全中心共截获移动端新增恶意软件样本约 434.2 万个，平均每天新增约 1.2 万个。据 CVE Details 2017 年度报告[3] 可知, Android 系统在 2017 年共被曝出 842 个安全漏洞，位居产品漏洞榜单第一名。相比 2016 年的 523 个，增长了 60%, 因此研究 Android 安全漏洞很有必要。

Android 权限管理机制是 Android 安全的基础，它要求应用软件使用特定的硬件或者软件资源时必须申请特定的权限，但是同时为了方便应用的功能复用，Android 系统允许开发者调用系统组件或者其他应用组件的接口来实现自己的业务需求。Android 系统对应用间组件间调用并没有进行强制的权限检查，其造成了应用间权限提升的可能，即应用可能存在能力泄露①漏洞，一个特权应用的特权能力可能被其他应用或者攻击者非法利用。即使 Android 系统提供了组件间的权限检测机制，但是根据我们对 2017 年豌豆荚应用市场各类别共 611 个受欢迎的应用的分析结果，发现使用这些检测机制的 APP 并不是很多，只有 35.8 % 的 APP 使用了组件间权限检测机制。Li 等人在文献[4] 中也同样提及发现此类问题。这就导致了一系列安全问题。例如：Go 短信是一个很受欢迎的第三方短信应用，Go 短信曾被曝出短信漏洞。一个不包含任何权限的应用可以利用 Go 短信来发送订购付费服务的短信[5]。三星 Epic 4G 手机系统应用存在 MASTER_CLEAR 能力泄露，只需构造简单的 *intent* 对象 (组件间通信输入) 就可以轻松地删除用户所有的数据[6]。这可能是由于 Android 组件间易混淆的使用规则导致开发者无意间

①本文能力泄露和权限提升一词不做区分

暴露组件或者紧迫的 Android 应用开发周期让开发者忽略了对组件间权限的检查[7]。应用厂商的主要精力放在功能实现和用户体验上，目的是为了尽早推出新版本，占领市场，忽略了对应用安全的关注[8]。因此，研究检测 Android 应用间的能力泄露漏洞具有重要的理论意义和很高的研究价值。

1.2　研究现状

现有针对 Android 组件间漏洞的分析主要分为三类：静态分析，动态分析，动静态结合分析。接下来，本文将从这三个方面来介绍现有关于能力泄露的工作。

1. 静态分析

2011 年 Chin 等人提出 Android 组件间通信漏洞检测工具 ComDroid[9]，检测的 Android 组件间通信漏洞包括组件劫持、Intent 欺骗等。针对 Android 应用间能力泄露漏洞，ComDroid 并没有具体实现对该漏洞的检测，只是提出了可能出现该漏洞的几种组件间通信场景。且对于该工具检测到的其他 Android 组件间通信漏洞，该工具并不能够确定漏洞是否真实存在，需要开发者手动确认。2012 年 Lu 等人提出的自动化检测 Android 应用组件劫持缺陷的静态分析工具 CHEX[10] 同样存在此问题。

2014 年 Li 等人提出了组件间隐私泄露检测工具 PCLeak[4]，PCLeak 支持检测两种隐私泄露方式，分别是 PPCL(Potential Passive Component Leak) 和 PACL(Potential Active Component Leak)。PPCL 是指组件存在从组件 entry-point 到 sink(可发送数据到 Android 应用程序外部的 API) 的数据流，当组件存在 PPCL，恶意应用可通过组件间通信使用该组件泄露隐私数据。PACL 是指组件存在从 source(可获取用户隐私数据的 API) 到组件 exit-point 的数据流，当组件存在 PACL，说明该组件主动发送隐私数据给其他组件，可能是无意或者故意的泄露用户隐私泄露数据。该工具只考虑隐私泄露，并未考虑能力泄露。该工具也考虑了漏洞验证，其漏洞验证的缺点在于测试 APP 的 *intent* 测试用例构造过于简单，漏洞验证的结果缺乏可靠性。可能存在特殊构造的 *intent* 测试用例可以触发漏洞。

2016 年 He 等人提出了一个阻止权限提升的工具[5]，利用静态分析检测出可能存在的权限提升路径，在路径上插桩提醒信息，提醒用户可能存在权限提升。但是由于检测的权限提升路径可能并不能执行 (权限提升漏洞实际并不存在)，会出现误提醒。而且大部分用户专业知识不高，对弹出的提示信息会感到困惑。同样存在此问题的工作还有 2017 年 Xie 等人提出的自动修复 Android 组件间漏洞的补丁工具 AutoPatchDroid[11]。

2017 年 Wu 等人提出了使用静态分析检测 Android 应用间权限提升问题[12]，但是其并没有考虑没有外部数据输入的权限提升。同时，使用他们的工作分析 550 个 APP，共检测出 37 个 APP 存在缺陷，其中手动确认后发现 12 个 APP 为误报，误报率高达 32%。且仅有 10 个 APP 能被确认可被攻击者利用，还有 15 个 APP 认为其是高风险漏洞 (不能确认漏洞是否可被利用)。从中可以看出，静态分析工作可能的高误报率将导致开发者需要手动确认大量漏洞，工作量大，且工作难度高。

2017 年 Liu 等人提出了检测权限提升的工具 MR-Droid[13]，其使用 map-reduce 系统来检测大规模 APP 之间存在的应用间漏洞，将 APP 联合进行分析。这种方式检测出的应用间漏洞结果取决于 APP 的来源选择。不能通过这个结果 APP 来判断当前 APP 是否是安全的。可能只是没有把可能造成应用间漏洞的 APP 没有选择进去。Zhong 提出的工具[14] 以及 COVERT[15] 同样存在这个问题。

现有 Android 组件间漏洞的静态分析相关工作的主要缺陷是不能确定漏洞是否真实存在，开发者需要手动确认 bug。

2. 动态分析

2011 年 Felt 等人提出了一种解决 Android 应用间权限提升问题的方案[16]。其通过动态缩小被调用者 APP 的权限来实现，即通过将被调用 APP 的有效权限减少为调用者 APP 和被调用者 APP 权限集合的交集，并以此实现了 IPC Inspection 工具。虽然该方法有效的解决了权限提升问题，使得调用者不可能通过调用其他 APP 获得自身没有的特权，但是 Android 应用执行被特权保护的 API 需要特定的权限，当没有该权限时会引起 Android 应用崩溃。因此当缩减被调用者 APP 的权限时，可能会造成被调用者 APP 在执行特定的代码时没有权限，导致应用崩溃。

2011 年 Bugiel 等人提出一种运行时检测并阻止权限提升攻击的动态分析工具 XManDroid[17]，XManDroid 监控所有应用程序之间的组件间通信，并获取组件间通信双方的权限输入到策略数据库中，然后通过系统策略检查算法判断能否建立组件间通信连接。策略数据库中包含着预先定义的组件间通信权限策略。例如：含有短信权限的应用不能和含有网络短信权限的应用进行通信；通话功能需要用户确认。XManDroid 通过基于系统策略检查有效地防止了 Android 应用间权限提升的问题。但是既定的策略不能满足实际需求，因为未考虑运行时上下文。对于同一种输入，在不同的上下文，可以是非法的，也可以合法的，例如：通过网络分享短信内容，可以是用户的目的行为，也有可以是恶意的隐私泄露行为。

2017 年 Dai 等人[18] 提出资源虚拟化方法应对组件间权限提升漏洞，对 Android 系统的资源管理和 ActivityManager 等地方进行修改，让资源管理器可以知

道组件间调用的信息，资源管理器通过这个信息决定来决定返回虚拟化的资源还是实际的资源。这种方案虽然解决了之前通过裁剪被调用者权限来预防权限提升的应用崩溃问题。但由于 Android 资源各种各样，虚拟化非数值的资源难度很大，因此可扩展性较差。

动态分析的 Android 应用间权限提升的相关工作主要是针对 Android 应用间权限提升问题提供解决方案。其研究的主要问题是如何判断当前状态是权限提升以及如何阻止权限提升。当前的研究工作主要是通过设定既定的访问策略来检测当前的组件间通信是否合法，因为应用所处的环境是多变的, 所以既定的访问策略无法满足实际需求，必须考虑运行时上下文。例如：用户调用第三方短信应用发送短信是正常行为，非用户确认的发送短信行为为非法。

3. 动静态结合分析

2014 年 Yang 等人[19] 提出了一个动态测试的方式用来测试 APP 是否存在能力泄露漏洞。测试用例 *intent* action 属性是从 APP 的代码中所有字符串中选择，extra 属性值由动态分析反馈得到。其中 *intent* 的 category 属性值并没有考虑，extra 属性值的 key 和 type 能从动态反馈获取，但是 extra 的值是随机生成的，因此其代码覆盖率低。

2016 年 Demissie 等人[20] 提出了一个使用动静态结合分析方法的 Android 应用间权限提升检测工具 AWiDe，对于静态分析检测的权限提升的漏洞，动态产生测试用例来判断漏洞是否可执行。其在构造测试用例时，仅使用 AndroidManifest 文件中暴露组件的 intent-filter 的信息去构造测试用例，并没有使用代码中的信息，例如：intent extra 属性信息不会出现在 intent-filter 中，因此构建的测试用例代码覆盖率低。

现有 Android 应用间能力泄露漏洞的动静态结合分析相关工作在生成测试用例的方式还存在不足，代码覆盖率低。

1.3　主要研究内容

如上分析，Android 应用间能力泄露漏洞是一种危害性非常大的漏洞，其他应用或者攻击者可以利用应用的 Android 应用间能力泄露漏洞非法地使用该应用的特殊能力。如果应用存在这种漏洞，将会带来巨大的安全隐患。现有的静态分析工作存在着不能确定漏洞是否真实存在的问题，检测结果误报率高将极大加重开发者确认修复 bug 的工作量。而市场要求应用开发周期越短越好，因此自动化检测并确认 Android 应用间能力泄露漏洞的工具对于开发者极其重要。而现有动静态结合分析的工作在生成测试用例的方式存在不足，导致代码覆盖率低，漏报率高。为了克服以上问题，本文首先引入符号执行技术，希望通过符号执行

技术生成测试用例，提高测试用例的代码覆盖率。这是本文的第一个工作，基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成。然后在本文的第一个工作研究基础上，总结其存在的不足，本文又重新提出了一个新的 Android 应用间能力泄露利用的自动化生成方法，基于动态反馈的 Android 应用间能力泄露漏洞利用的自动化生成。两个工作都能自动化检测并确认 Android 应用的能力泄露漏洞，同时生成触发漏洞的漏洞利用，帮助开发者快速确认、修复 bug，缩短软件开发周期。本文两个工作的主要研究内容如下：

（1） 基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成：研究使用符号执行技术生成高精度的 Android 应用间通信测试用例，提高代码覆盖率；研究针对 Android 应用间通信的符号执行优化方法，使符号执行适用于实际应用的能力泄露漏洞检测工作。

（2） 基于动态反馈的 Android 应用间能力泄露漏洞利用的自动化生成：研究精确的基于外部 *intent*(来自其他应用组件间调用传入的 *intent* 对象) 的数据流分析技术，用于找到每一个包含路径条件信息的变量和插桩位置点，保证动态获取路径条件信息的完备性；基于动态反馈的测试用例自动化生成方法，充分利用动态反馈信息，生成高代码覆盖率的测试用例集合。

（3） 设计一整套 Android 应用间能力泄露利用自动化生成工具。实现 Android 应用间能力泄露的自动化检测与确认，有效减少开发者手动确认 bug 的工作量，缩短 APP 开发周期。

1.4　论文组织结构

本篇论文总共分为六章，各章节主要内容如下：

第一章：绪论。本文首先介绍了 Android 安全问题的现状，从 Android 权限管理机制引出本文的检测对象，Android 应用间能力泄露漏洞。然后通过几个 Android 应用间能力泄露漏洞实例说明 Android 应用间能力泄露漏洞能力泄露的危害性以及研究 Android 应用间能力泄露漏洞的价值。同时对当前 Android 应用间能力泄露漏洞的静态分析、动态分析以及动静态结合分析三个方面相关的工作进行分析介绍以及概括，从而引出本文研究工作的研究目标，最后介绍本文两个研究工作基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成和基于动态反馈的 Android 应用间能力泄露漏洞利用的自动化生成的主要研究内容。

第二章：相关理论知识。本章介绍了本文研究工作的相关背景知识以及 Android 应用间能力泄露漏洞的定义。其中相关背景知识主要包括 Android 应用基础、Android 权利模型、Android 通信机制以及程序分析相关理论。其中 Android 应用基础介绍了 Android 应用程序包格式、AndroidManifest.xml 以及 Android 四

大组件；Android 权利模型介绍了 Android 应用使用权限的规则；Android 通信机制介绍了 Android 组件间通信机制和线程间通信机制；程序分析相关理论介绍了程序分析的概念、程序分析的难度与评价、以及三种程序分析方法与技术，包括数据流分析技术、符号执行以及动态分析。

第三章：基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成。本章介绍了本文第一个工作基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成的具体实现，其中包括 Android 应用的方法调用图和方法控制流图的构建、能力泄露路径查找、*intent* 条件约束提取与约束求解以及能力泄露漏洞自动化测试。其中能力泄露路径查找中介绍了本文根据 Android 应用间能力泄露漏洞问题的特性从方法调用图层次和控制流图层次两个层次对符号执行进行的优化。

第四章：基于动态反馈的 Android 应用间能力泄露漏洞利用的自动化生成。本章介绍了本文第二个工作基于动态反馈的 Android 应用间能力泄露漏洞利用的自动化生成的具体实现，主要包括使用基于外部 *intent* 的上下文敏感、流敏感的过程间数据流分析获取包含 *intent* 路径条件信息的变量和插桩的位置点，自动化生成插桩语句插桩 APP 以及基于动态反馈的测试用例自动化生成技术。

第五章：实验结果与评估。本章首先介绍了实验环境、实验数据集，然后介绍了使用本文两个工具和相同类型工具 IntentFuzzer 分析豌豆荚各类别共 611 个 APP 的实验结果，包括能力泄露检测结果以及分析时间；并从漏报率和时间两个角度对三个工具进行评估；最后介绍了两个严重能力泄露漏洞的利用示例。

第六章：总结与展望。本章先介绍了 Android 应用间能力泄露漏洞的危害性以及目前 Android 应用间能力泄露漏洞检测工作的缺陷，最后总结了本文两个 Android 应用间能力泄露漏洞自动化生成工作的方法与结果，分析了本文工作存在的不足，并针对存在的不足以及对 Android 应用间能力泄露漏洞的理解提出了三点未来的研究工作。

1.5　本章小结

由 360 安全公司和 CVE Details 报告可知 Android 手机安全形势严峻，而 Android 应用间能力泄露漏洞作为危害性极强的漏洞，研究检测它具有重要的理论意义和很高的研究价值。本章分析总结目前已有的 Android 组件间通信安全的相关工作 (包括静态分析、动态分析、动静态结合分析三个方面) 存在的不足，并介绍了当前软件工程漏洞自动化确认的需求，从而引出本文研究工作的研究目标，然后介绍了本文两个研究工作的主要研究内容。最后本章介绍了本篇论文的论文组织结构。

第 2 章　相关理论知识

本章将介绍 Android 应用间能力泄露漏洞利用的自动化生成的相关背景知识，主要分为两个部分：Android 背景知识和程序分析相关知识。Android 背景知识主要介绍 Android 应用基础、Android 权利模型以及 Android 通信机制；程序分析背景知识主要介绍程序分析的概念、程序分析的难度与评价以及本文涉及到的一些程序分析方法与技术。本章最后以一个实际的例子来介绍 Android 应用间能力泄露漏洞，并对 Android 应用间能力泄露问题进行详细定义。

2.1　Android 应用基础

2.1.1　Android 应用程序包格式

Android 应用主要使用 Java 语言、Kotlin 语言等开发。开发者在编译打包 Android 应用时，Java 语言会被编译成 Java 字节码，然后再通过 dx 工具转化为 Davilk/Art 字节码。Davilk/Art 字节码将运行在 Android 的 Davilk/Art 虚拟机上。Android 应用程序以.apk 结尾，其解压之后包括 7 个部分：AndroidManifest.xml 文件、assets 目录、lib 目录、META-INF 目录、res 目录、classes.dex 文件和 resources.arsc 文件。asset 目录、resources.arsc 文件、res 目录包含了应用程序所有的资源，lib 目录包含了应用程序所需要的库文件，META-INF 目录包含应用签名的相关信息，classes.dex 文件是 Android 所有应用程序代码编译后的二进制文件、AndroidManifest.xml 文件是 Android 应用程序的配置文件，包含了应用的一些基本信息。Android 应用程序 (apk 文件) 可以方便地被反编译，获取其中 apk 中的代码信息和 Android 应用程序的配置文件等信息。

2.1.2　AndroidManifest.xml

每一个 Android 应用程序都有一个全局配置文件 AndroidManifest.xml，其是对整个 Android 应用的全局描述，包含了应用的一些基本信息。图2.1是一个应用程序的 AndroidManifest.xml 的简化版本，其声明包括应用程序的名称，图标，包名，拥有的组件，申请的权限等。

组件在 AndroidManifest.xml 中的 *android:exported* 属性决定其是否可被其他应用 (非相同应用签名的应用) 的组件调用，默认情况下 Content Provider 的 *android:exported* 属性为 *true*，表示可被其他应用的组件调用（本文中称可被其他应用调用的组件为“暴露组件”）。其他类型组件默认情况下 *android:exported* 属性值为 *false*，即默认情况下不能被其他应用的组件调用。组件的暴露规则如下：

（1） 组件声明 *android:exported=”true”*, 则该组件为暴露组件。

（2） 组件声明至少含有一个 <intent-filter> 且不包含 *android:exported=”false”*，则该组件为暴露组件。

（3） 不满足以上两条的组件为内部组件

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.superrhino.rarering">
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
    <application android:icon="@drawable/ic_launcher" android:label="@string/app_name">
        <activity android:name="com.superrhino.rare.RouteActivity">
            <intent-filter>
                <action android:name="android.intent.action.VIEW"/>
                <category android:name="android.intent.category.DEFAULT"/>
                <category android:name="android.intent.category.BROWSABLE"/>
                <data android:scheme="inicoapp"/>
            </intent-filter>
        </activity>
        <activity android:name="com.superrhino.rare.account.login.LoginActivity" />
        <activity android:name="com.superrhino.rare.LaunchActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <receiver android:exported="true" android:name="com.superrhino.RongNotificationReceiver">
            <intent-filter>
                <action android:name="io.rong.push.intent.MESSAGE_ARRIVED"/>
                <action android:name="io.rong.push.intent.MI_MESSAGE_ARRIVED"/>
                <action android:name="io.rong.push.intent.MESSAGE_CLICKED"/>
                <action android:name="io.rong.push.intent.MI_MESSAGE_CLICKED"/>
                <action android:name="io.rong.push.intent.THIRD_PARTY_PUSH_STATE"/>
            </intent-filter>
        </receiver>
        <service android:exported="true" android:name="com.xiaomi.mipush.sdk.PushMessageHandler"/>
        <provider android:exported="false" android:name="com.huawei.UpdateProvider"/>
    </application>
</manifest>
```

图 2.1　AndroidManifest.xml

2.1.3　Android 应用组件

Android 应用共有四大组件，其基本上包含了应用程序所有的功能。这四大组件分别是：Activity，Service，Content Provider，Broadcast Receiver。接下来分别介绍这四大组件。

1. Activity

Activity 是提供给用户交互的组件，通常我们使用 Android 应用程序的界面就是 Activity。一个 Android 应用程序一般至少包含一个 Activity。Activity 是由系统的 activity stack 来管理，Activity 拥有生命周期，表示一个 Activity 从创建到销毁时不同的运行状态。在这些状态下，我们可以通过 Android 提供的回调函数(onCreate()、onStart()、onResume()、onPause 和 onStop()s) 实现这些状态下 APP 需要处理的任务。APP 运行时，Android 系统会在不同的 Activity 切换状态下按照一定的顺序调用这些生命周期函数。Android 使用 *Context.startActivity(Intent)*API 启动一个 Activity。

2. Service

Service 是一个可以在后台长时间运行的组件，其不需要和用户进行交互。即使用户切换出当前应用，Service 依然可以保持后台运行，Service 适合做一些耗时操作，比如网络数据传输，音乐播放，文件 IO 等。Service 也有生命周期，其生命周期的回调函数有两种，分别是:onCreate()→onStartCommand()→onDestroy()和 onCreate()→onBind()→onUnbind()→onDestroy()。其分别对应 Service 的两种启动和停止方式：*Context.startService(Intent)* 和 *Context.stopService(Intent)*、*Context.bindService(Intent)* 和 *Context.unbindService(Intent)*。

3. Brocast Receiver

广播表示是一个全局性的事件信息，该组件是为了让 Android APP 可以响应一些全局性的事件。Android 应用可以发送广播和接收广播。Android 系统中定义了各种全局性事件广播，比如：Intent.ACTION_BOOT_COMPLETED 广播表示 Android 系统已经启动，Intent.ACTION_INPUT_METHOD_CHANGED 广播表示当前输入法已改变。Android APP 通过继承 BroadcastReceiver 类覆写 *onReceive(Intent)* 方法实现广播事件的处理，并在 AndroidManifest.xml 中注册监听的广播事件 (监听一些特殊广播需要权限)。当广播事件发生时，该 Android APP 的 Receiver 的 *onReceive(Intent)* 会被 Android 系统调用。Android APP 可通过 *Context.sendBroadcast(Intent)* 来发送广播。

4. Content Provider

Android 每个应用程序都是隔离运行的，每个应用程序除了外部公共存储空间 (应用使用需要申请外部存储权限)，只能访问自己的应用程序内的数据。Content Provider 组件提供了多个 Android APP 之间的数据共享机制以及细粒度的数据访问控制。Content Provider 提供了一系列通用数据访问接口，包括：onCreate(): 初始化 Content Provider，query(Uri): 根据 Uri 在指定 Content Provider 查询数据，insert(Uri)：根据 Uri 向指定 Content Provider 插入数据，update(Uri): 根据 Uri 更新指定 Content Provider 的数据，delete(Uri): 根据 Uri 删除指定 Content Provider 的数据，getType(Uri): 根据根据 Uri 查看指定 Content Provider 的数据类型。

2.2　Android 权利模型

Android 系统要求 Android APP 在使用特定的软件资源或者硬件资源时，需要申请特定的权限。APP 在 AndroidManifest.xml 中使用 <uses-permission> 标签来申请需要申请的权限，如图2.1所示，该 APP 申请了一个监听系统启动广播的权限，即申请自启动的权限。还有一些 special permssions，需要开发者使用代码跳转到特定的系统用户授权页面，让用户授予。另外，Android 系统还支持自定

义权限。

Android 权限被分为四个级别，这些级别将影响这些权限是否需要动态授予。这四个级别分别是：normal、signature、signatureOrSystem、dangerous。对于 signature 级别权限，只有和该 apk(定义了这个权限的 apk) 拥有相同 APK 应用签名的应用才可以申请该权限; 对于 signatureOrSystem 权限，两种应用可以申请该权限:1) 和该 apk(定义了这个权限的 apk) 拥有相同 APK 应用签名的应用 2) /system/app 目录下的 Android 系统应用。在 Android M(Android 6.0) 之前，应用一旦安装了 APP，Android 系统将授予 APP 所有其申请的权限。Android M 之后，Android 系统采取动态授予的机制，APP 在安装完成之后，将授予 APP 其申请的所有 normal 权限，而 dangerous 权限需要 APP 在运行时由用户动态授予，用户可以选择拒绝和同意。Android 系统为了兼容老版本的应用，当 Android APP 将其 targetSdkVersion 设置为 Android 6.0 之前，将依旧采取之前的 Android 权限机制。

Android 支持组件间强制权限，即在 AndroidManifest.xml 的组件声明处声明调用该组件所需要的权限。如果调用者没有该权限，那么调用者的调用方法将会抛出 *SecurityException* 异常。但是该机制只能检查调用者单个权限。Android 还支持通过 *Context.checkCallingPermission()* 来检查组件间调用者的权限，但是根据我们实验发现这个 API 只能在使用 AIDL(Android Interface Definition Language) 的 IPC(inter process communication) 调用时才能工作正常，其他组件间调用并不适用。因此当前 Android 系统官方提供的组件间权限检测机制存在缺陷，这也导致了 Android 应用间能力泄露漏洞问题更加严重。

2.3　Android 通信机制

2.3.1　Android 组件间通信

Android 组件间通信指的是 Android 组件间的相互调用，Android 组件间调用使用 *intent* 对象传递信息。其含有五种属性：Component、Action、Data、Category、Extras、Flags。分别表示要启动的组件名称 (String)、此次执行的操作类型 (String)、此次执行操作的数据类型 (Uri)、处理本 *intent* 对象的组件类型 (Set<String>)、额外的键值对信息 (Set<key,type→value>)、启动组件的方式 (int)。本文称来自其他应用组件间调用传入的 *intent* 对象为外部 *intent*。*Intent* 分为两种类型：显式和隐式。显式 *intent* 指 Component 属性已经确定的 *intent*。隐式 *intent*Component 属性未知，但定义了其他属性值，例如：要执行的操作和要执行的数据类型等。Android 系统将根据隐式 *intent* 的属性值选择匹配合适的目标组件进行启动，如果存在多个匹配合适的目标组件，则显示一个对话框列表，供用户自己选择。*Intent-filter*

是由 Activity 在 AndroidManifest.xml 中声明的其可以接收的 *intent* 类型，包括 Action、Data、Category 属性值。图2.1中 15 行到 18 行为应用 MainActivity 的 *intent-filter*，其接收 Action 值为 *android.intent.action.MAIN*，Category 属性值包含 *android.intent.category.LAUNCHER* 的 *intent*。Android 系统将 *intent* 三个属性值 Action、Data、Category 和组件的 *intent-filter* 匹配，寻找匹配成功的组件启动。*intent* 和 *intent-filter* 具体的匹配规则如下：

（1） 显式 *intent* 不需要匹配 *intent-filter*，直接可调用目标组件。

（2） 隐式 *intent* 匹配组件的任何一个 *intent-filter* 成功，则认为匹配成功。

（3） 隐式 *intent* 匹配组件一个 *intent-filter* 成功，则需要匹配 *intent-filter* 下的所有属性。

（4） 隐式 *intent* 只需匹配组件 *intent-filter* 的一个 Action。

（5） 隐式 *intent* 的 Category 属性集合必须是组件 *intent-filter* Category 属性集合的子集，所以没有 Category 的隐式 *intent* 总是可以匹配。注意：*Context.startActivity(Intent)* 和 *Context.startActivityForResult(Intent)* 中的 *intent* 对象会被默认加入 CATEGORY_DEFAULT 属性。

（6） 隐式 *intent* 只需匹配组件 *intent-filter* 的一个 Data。

2.3.2　线程间通信机制

在 Android 系统中，默认情况下一个应用程序内的各个组件 (例如：Activity、Service、BroadcastReceiver) 都会在一个进程中执行，且由此进程的主线程负责执行。Android 系统规定了 View 组件只能由主线程负责，如果主线程执行耗时任务超过 5s，应用程序会产生应用程序无响应错误 (ANR)。因此 Android 应用程序广泛使用多线程技术。Android 主要支持三种 Android 线程通信方式。分别如下:

（1） 传统 Java 线程通信 Thread，使用 Thread 类时需要实现 Runnable 接口的 run 方法以指定子线程需要执行的任务。使用 Thread.start() 方法后，系统会开启一个子线程执行 run 方法。

（2） Android 异步任务 AsyncTask，使用 AsyncTask 类需要实现 onPreExecute()、doInBackground(Params...)、onProgressUpdate(Progress...)、onPostExecute(Result) 四个方法。这四个方法分别用于任务开始前设置，后台处理任务、任务进度更新、任务结束后收尾工作。其中，除了 doInBackground(Params...) 方法在子线程中调用，其他方法都是在 UI 线程 (主线程) 中被调用。一般情况下，AsyncTask.execute() 方法执行后，onPreExecute()、doInBackground(Params...) 将会依次执行，publishProgress(Progress...) 方法将在 doInBackground(Params...) 中调用发布当前任务执行进度，然后 onProgressUpdate(Progress...) 方法将进度更新到 UI。最

后 onPostExecute(Result) 会被调用，表示任务执行完之后的清理工作。

（3） Android Handler，Android 可通过两个线程之间共享 Handler 对象的方式实现通信。每个 Handler 对象实例都与一个线程的 MessageQueue 关联，其允许其他线程发送 Message(*Handler.sendMessage(Message)*) 和 Runnable 对象 (*Handler.post(Runnable)*) 到当前线程的 MessageQueue。当前线程将在 Handler 对象中的 Handler.Callback(*handleMessage(Message)*) 中处理 Message，或者当前线程执行 post 方法传入的 Runnable 方法。

2.4　程序分析相关理论

程序分析是指通过对程序的分析获取程序相关性质的技术，例如: 程序的安全性、程序的正确性、程序的性能等性质。程序分析技术广泛运用在编译器、漏洞挖掘工具上。程序分析技术传统上可分为静态分析技术和动态分析技术。静态分析技术是指对程序完全静态处理以获取程序相关性质，不运行待检测程序。而动态分析技术是指通过观察程序动态行为获取程序性质，需要运行待检测程序。测试技术、形式化验证技术和程序分析技术密切相关。其中测试技术以触发程序缺陷为目标，其广泛运用在实际软件开发中。测试技术种类各种各样，但是其一般需要人工构建好测试用例，然后使用其作为程序输入运行程序，最后观测程序的运行结果得到测试结论。

2.4.1　程序分析的难度与评价

程序分析的作用目标是多样的，既可以是程序源码，又可以是二进制可执行文件。对于不同的程序源码类型，可能需要不同种类的程序分析技术。例如：JAVA 是面向对象的程序设计语言，其不同于面向过程的 C 语言，分析其需要处理一些面向对象程序语言的性质，例如：多态，JAVA 中允许父类引用指向子类对象，父类调用父类抽象方法时，会调用子类对象的方法。而这种情况在 C 语言中并不存在。同时对于不同的应用环境下，不同类型的程序，程序的性质可能是不同的，且人们关注的性质也是不同的。因此面对各种不同的程序设计语言以及各种不同的程序性质，程序的自动化分析是一个巨大的挑战。由 Rice 定理[21]可知：对于程序行为的任何一个非平凡属性，都不存在能够检查其属性的通用算法。因此存在大量的程序分析问题是无法判定的。例如：程序路径的可行性是不可判定的[22]。如果存在输入数据使得程序沿着某条路径执行，则称该路径是可执行的。没有一种算法可以判断给定计算机程序中的一条路径是可以执行的; 而且一般情况下，也不能判定给定计算机程序是否能够结束。也就是说，没有一种算法可以判断任何给定的计算机程序是否总能终止。

由于程序分析的理论难度以及被分析的计算机程序的复杂度，程序分析在

实际运用中往往不要求完全精确的分析，因此带来了漏报和误报等问题。漏报是指程序分析工具在检测程序 bug 的过程中未检测出的程序 bug。而误报是指程序分析工具检测的缺陷实际上并不存在。评估某个程序分析技术或程序分析工具时，不仅要考虑被检测程序的量级，分析程序的时间效率，发现漏洞的危害程度，以及漏报率和误报率等指标，还要关注用户的需求。

2.4.2　程序分析方法与技术

1. 数据流分析

数据流分析是一种模拟计算机程序数据流动以获取不同程序语句位置点的程序状态信息的技术。编译器中使用了诸多数据流分析技术，例如：部分冗余分析，常量传播等。经典的数据流分析理论[23] 使用格 $< L, \sqcap >$ 表示不同程序语句位置的程序状态集合, 为每一个程序语句设计一个转移函数，转移函数作用于程序状态集合，表示程序语句执行之后，程序状态集合发生的变化。在程序的控制流图中，某个节点的程序状态信息来源于其所有的父节点，因此对于汇聚节点(多个分支语句汇合的节点) 的程序状态信息是所有父节点程序状态集合的并集，其表示执行不同分支语句下所有可能的程序状态集合。数据流分析技术依据是否只考虑当前方法内的数据流动可分为过程内数据流分析和过程间数据流分技术两类，其中过程间数据流分析考虑方法间调用引起的数据流动。对于过程间数据流分析，需要注意的是在不同的方法调用上下文，同一个方法可能会有不同的数据流输入。考虑不同上下文调用的过程间数据流分析称为上下文敏感的过程间数据流分析。上下文敏感的过程间数据流分析主要包括两种方法：

（1） IFDS 数据流分析框架，其是由 Reps 等人在 1995 年提出[24]，IFDS 分析框架将数据流分析中计算状态信息 (需是满足分配性的有限集合) 的问题转化为一个有向图的可达性问题，这使得可以在图上有效地进行上下文敏感的过程间数据流分析。IFDS 框架基于过程间控制流图定义了一个 supergraph，supergraph 上每个节点对应一个静态程序点的程序状态信息；supergraph 上每条边对应数据流分析中的 transfer function。通过遍历从程序入口到某个程序点的路径，我们可以计算出该程序点的程序状态信息。

（2） 基于 value flow graph 的稀疏数据流分析方法，传统的数据流分析计算某个程序点的状态信息时，需要计算从数据流起点到该程序点所有可达路径中所有节点的转移函数，因此在此过程存在较多重复冗余操作，影响数据流分析的效率，尤其是对于过程间数据流分析。Li[25] 和 Sui[26] 等人提出通过构建一个稀疏的 value flow graph 来进行数据流分析。value flow graph 直接表示程序中变量的依赖关系，没有冗余节点，状态信息可以在值流图上进行有效地传播，避免

冗余传播，提高数据流分析效率。

2. 符号执行

符号执行是一种精确度相对较高的程序分析技术。传统的符号执行使用符号化输入代替实际输入以模拟执行被分析程序，其并不实际执行被分析程序，其将程序中的语句操作映射为对符号表达式的操作。针对不同的程序路径，模拟执行该路径上的语句。对于路径中的条件语句，生成分支条件约束加入到当前路径的路径条件中。然后通过约束求解器对路径条件的可满足性进行求解判断。如果约束求解器结果为可满足，则说明此条路径可执行，即存在具体输入使得该路径可执行。如果约束求解结果为不满足，则终止分析当前路径。

符号执行中分析的每条路径都可以映射到实际执行中的一条路径，因此其可以很大程度上模拟计算机程序的执行过程中的行为。所以可通过符号执行的路径信息开展多种软件分析与验证的活动，包括缺陷检查、自动化测试以及部分程序验证等。相比确定程序输入的程序分析方法，在理论上，符号执行可以体现更多的程序行为。符号执行技术强烈依赖于约束求解技术，约束器求解器的发展直接决定了符号执行的发展。符号执行提供了一种遍历程序路径空间的手段，但是程序路径空间大小随着程序的规模增大而指数级增长。例如：对于只有 n 条 2 分支的条件语句的串行程序，其共有 2^n 条路径。这是制约符号执行的关键因素。目前，符号执行技术的实际运用仍面对可行性和可扩展性两方面的挑战。可行性挑战是指如何平衡符号执行的精确性与可靠性的关系；而可扩展性挑战是如何提高符号执行的效率，使其能够在有限的时间、内存等资源下更快地完成分析工作。

在扩展性方面，路径爆炸和约束求解是符号执行的的两大难题，现有的可扩展性研究工作也主要围绕这两方面展开。在缓解路径爆炸方面，现有的工作主要分为两个思路:1) 根据不同程序分析目标问题的特性，针对性的提出高效的路径搜索策略，让符号执行能够快速地到达程序分析目标，例如：探索程序不同版本差异[27-28]、提高程序覆盖率[29-32]；2) 根据程序目标特性减小程序路径空间，包括：等价路径约简[33-36]、条件合并[37]、程序抽象或摘要[38]、程序切片[39]。在约束求解器方面，已有的工作主要是减少约束求解器调用次数以及对复杂程序特征的高效编码。减少约束求解器调用次数主要通过路径条件查询优化，包括查询缓存和重用[40]。

3. 动态分析

动态分析技术是指在预先设定好的测试用例下动态运行程序，并分析程序运行的行为以及程序运行的结果，动态分析常用于程序缺陷检测等。相比静态分析，动态分析能更多地获取静态分析中无法得到的动态属性信息，例如：面向对象语言 (例如：java) 中的继承与多态、指针、线程交替、指针等。因此动态分析

技术一定程度上补充了静态分析技术的不足。

程序的动态分析基本可以分为离线动态分析与在线动态分析。离线动态分析是指记录下程序运行过程中的行为，然后在程序运行结束之后对程序进行分析；在线动态动态分析是指在程序的运行过程中获取程序运行信息 (例如：上下文) 分析当前程序行为。两者的基本思想都是从程序运行过程中获取信息进行分析，从而查找定位缺陷。具体来说，总共包括两方面：1) 通过将程序运行结果与预知结果对比来判断程序中是否存在漏洞 2) 通过插桩技术或者其他监控技术记录程序动态运行信息，以判断是否存在错误的行为。虽然前者很直观，但是前者不能检测出触发了却没有反应在输出中的缺陷；后者可以直接观察出是否存在缺陷触发，即便该次触发并没有导致错误的输出。但是对于后者，我们需要提前定义错误的行为，对于一些非常见的缺陷，我们根本无法定义其错误行为，因此我们无法检测出此类缺陷。

2.5　Android 应用能力泄露定义

能力泄露，又被称为权限再分配[16]。其发生于一个正常应用的特权能力被没有特权的恶意应用非法利用。Android 组件间通信使用广泛，很多 Android 应用开发者通过暴露组件共享自己应用的功能。可能是由于 Android 组件间易混淆的使用规则导致开发者无意间暴露组件或者紧迫的 Android 应用开发周期让开发者忽略了对组件间权限的检查，使得 Android 应用间存在权限提升的可能，因此造成了应用的能力泄露漏洞安全问题。我们以一个实际的能力泄露漏洞为例。如图2.2所示，清理应用存在查杀后台应用 (kill) 能力泄露漏洞, 恶意应用使用精心构造的 *intent* 对象启动清理应用的缺陷组件，使得清理软件杀死某个正常应用。清理应用简化的可利用代码如代码块2.1所示，*LocalService* 是清理应

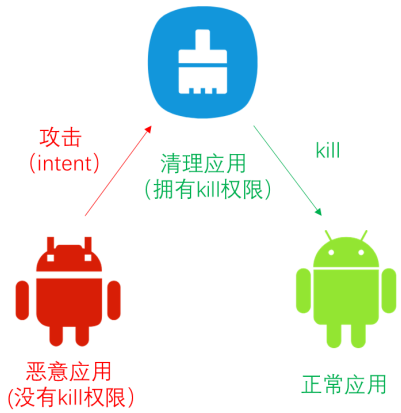

图 2.2　应用间能力泄露漏洞攻击示例

用的一个暴露 service 组件，其存在 KILL_BACKGROUND_PROCESSES 权限能

力泄露漏洞 (能力泄露的位置点是：*killBackgroundProcesses* 语句，其是查杀后台应用的 API，受 KILL_BACKGROUND_PROCESSES 权限保护，后文简述为 *tgtAPI*)。*LocalService* 组件可由其他应用使用 *bindservice* API 调用，*LocalService* 启动时其 *onBind* 方法将会被调用，传入外部 *intent* 对象。*onBind* 方法是暴露组件 *LocalService* 的起始方法，后文简述为 *startPoint*。由代码块2.1可知，执行 *onBind* 方法中的 *killBackgroundProcesses* 语句需要满足一定的路径条件，且路径条件和程序输入 (*intent*) 有关。因此恶意应用通过构建拥有 *key* 值为*"type"*, *value* 值为*"kill"* 的 *java.lang.String extra* 属性和 *key* 值为*"priority"*, *value* 值大于 0 的 *int extra* 属性的 *intent* 能够实现查杀后台应用，不需要 KILL_BACKGROUND_PROCESSES 权限。破坏了 Android 权利模型，因此对用户和手机造成了巨大的安全威胁。

```
1
2         public class LocalService extends Service
3         {
4           @Override
5           public IBinder onBind(Intent intent)
6           {
7             ...
8             if(intent.getStringExtra("type").equals("kill"))
9             {
10              if(intent.getIntExtra("priority",0)>0)
11                {
12                  ActivityManager activityManager= (ActivityManager)
13                  getSystemService(Context.ACTIVITY_SERVICE);
14                  activityManager.killBackgroundProcesses(intent.
15                  getStringExtra("packageName"));
16                }
17            }
18            ...
19          }
20        }
```

代码块 2.1　可利用代码

现详细定义 Android 应用间能力泄露漏洞如下：

定义 2.1　假设存在 Android 应用 A，其拥有的权限集合设为 *PSet*，权限与其保护的语句 (*unit*) 间的映射关系的集合设为 $PUMap(permission \rightarrow unitSet)$。应用 A 拥有的暴露组件集合为 *ECSet*，暴露组件拥有的根方法 (第一个方法:*root-method*) 集合设为 $ECMethodMap\ (exported\text{-}component \rightarrow methodSet)$。根方法到达权限保护的 *unit* 的可执行路径集合为 $RMUPathMap(root\text{-}method, unit \rightarrow pathSet)$。

$$if \quad PUMap \neq \varnothing, \quad ECMethodMap \neq \varnothing$$

$$\exists \quad intent \neq null, \quad \text{s.t.} \quad RMUPathMap \neq \varnothing \tag{2.1}$$

即当 A 应用 $PUMap$，$ECMethodMap$ 不为空时，存在一个 *intent* 不为空，使得 A 应用的 $RMUPathMap$ 不为空，则说 A 应用存在能力泄露漏洞，其能力泄露类型是 $RMUPathMap$ 中 $unit$ 对应的权限。

因为 Android 应用中存在很多关键功能 API 没有参数，且存在很多 API 即使不能控制数据流入，依然可以导致巨大的危害，因此本文考虑 APP 中所有 *tgtAPI*，即使这些 API 中并没有流入外部 *intent* 数据。需要注意的是：鉴于很多用户交互的越权组件间通信是正常的，因此本文研究能力泄露路径是不考虑 UI 方法的。我们不能认为这些有 UI 交互的能力泄露路径是非法的，因为其是用户知晓的。例如：通过短信分享某个新闻 APP 的内容链接给朋友，这个分享操作中会涉及到用户去点击确认发送短信，我们不能认为存在短信发送能力泄露，因为最终是由用户决定是否发送。而没有 UI 方法的短信发送能力泄露路径就是非法的，其存在严重危害性。

2.6　本章小结

本章主要介绍了本文研究工作的相关背景知识以及 Android 应用间能力泄露漏洞的定义。本文研究工作的相关背景知识主要分为两类，一类是 Android 基础知识，包括 Android 应用基础、Android 权利模型和 Android 通信机制；另一类是程序分析相关理论知识，包括程序分析的概念、程序分析的难度与评价以及本文涉及到的程序分析方法与技术。Android 应用基础主要介绍了 Android 应用的运行原理、Android 四大组件 (Activity, Servive, Brocast Receiver 和 Content Provider) 以及 Android 安装文件包含的的内容，其中重点介绍了 Android 应用的全局性描述文件 AndroidManifest.xml；Android 权利模型主要介绍了 Android 权限级别、Android 权限授予机制以及 Android 组件间强制权限；Android 通信机制主要介绍了 Android 组件间通信和线程间通信机制。程序分析的难度与评价主要介绍了当前程序分析存在的挑战和问题以及程序分析方法评价的一些指标，例如：误报率、漏报率；程序分析方法与技术主要介绍了本文中涉及到的一些程序分析技术，例如：数据流分析技术、符号执行，并对它们的相关分类以及相关研究工作进行介绍。最后，本章对本文要研究的对象 Android 应用间能力泄露进行详细定义。

第 3 章　基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成

本章将介绍基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成工具，工具由方法调用图和方法控制流图构建、能力泄露路径查找、能力泄露 *intent* 路径条件约束求解、能力泄露漏洞自动化测试四个部分组成。本章第一个章节先简要介绍这四个部分的功能和作用，然后分四个章节分别介绍它们的具体实现。

3.1　方法概述

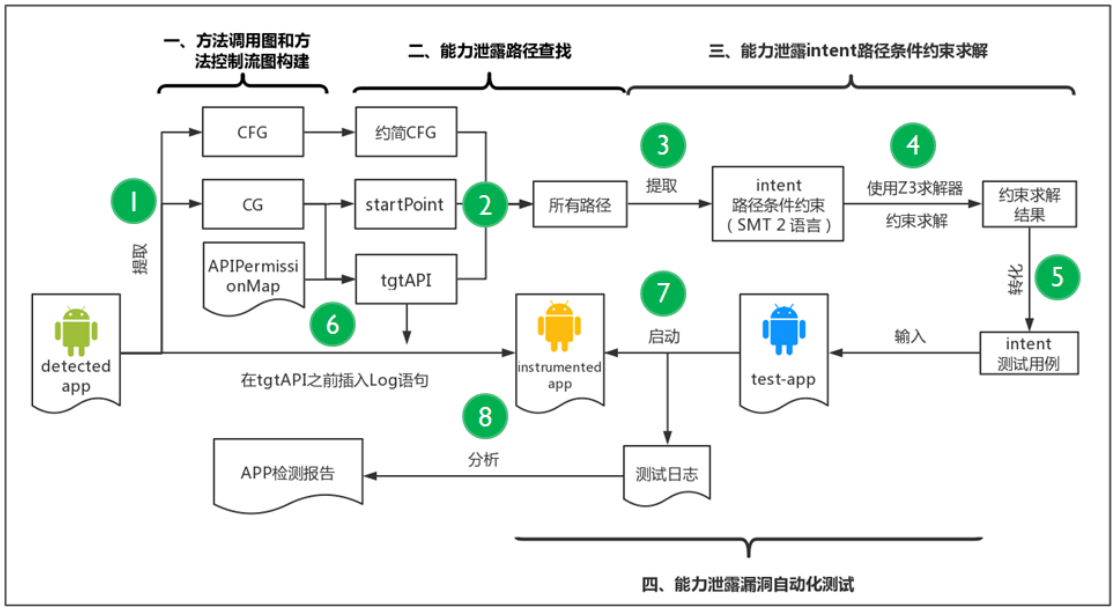

图 3.1　符号执行能力泄露利用生成工具流程图

图3.1是基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成工具的流程图，如图所示，本章提出的工具共分为四个部分: 方法调用图和方法控制流图构建、能力泄露路径查找、能力泄露路径 *intent* 条件约束求解、能力泄露漏洞自动化测试。其中前三个部分为静态分析，静态分析的算法框架如算法3.1 SMain 所示；工具的第四部分为动态测试。

算法 3.1　SMain

```
Data: cg, startPoint, tgtAPI, flowInUnitDataMap(unit → flowInDataSet)
Result: intentSet
Function SMain(cg, startPoint, tgtAPI, flowInUnitDataMap):
    reducedCG ← reduceCG(cg, startPoint, tgtAPI);
    allNodes ← reducedCG.getAllNodes();
    SE ← ∅;
    for n in allNodes do
        for s in successors(n) do
            SE.add(Pair(n, s));
        end
    end
    intent_path_condition_set ← ∅;
    for n,s in SE do
        reducedCFG = reduceCFG(n.cfg, s, flowInUnitDataMap);
        for path in getPaths(reducedCFG,s) do
            intent_path_condition =
              symbol_execution(n, path, flowInUnitDataMap);
            intent_path_condition_set.add(intent_path_condition);
        end
    end
    actionSet,categorySet,extraSet,dataSet← ∅;
    for i in intent_path_condition_set do
        actionSet,categorySet,extraSet,dataSet.addAll(i.value);
    end
    selectCategorySet.add(categorySet);
    selectExtraSet ← combinationWithDifferentKeyAndType(extraSet);
    intentSet ← ∅;
    for action in actionSet do
        for data in dataSet do
            for categorySet in selectCategorySet do
                for extraSet in selectExtraSet do
                    intent = newIntent(action, data, categorySet, extraSet);
                    intentSet.add(intent);
                end
            end
        end
    end
End Function
```

第一部分方法调用图和方法控制流图构建是整个工具的基础，首先构建待检测 APP(图3.1中 *detected-app*) 的方法调用图 (Call Graph, 简述为 CG) 和每个方法的控制流图 (Control Flow Graph, 简述为 CFG)，找到 Android 应用中所有受权限保护的语句 (*tgtAPI*) 以及所有暴露组件方法 (*startPoint*)；在第二部分能力泄露查找，寻找 *startPoint* 和 *tgtAPI* 之间所有需要求解路径条件的路径；第三部分能力泄露 *intent* 路径条件约束求解，提取第二部分中每条路径的 *intent* 路径条件约束 (使用约束求解器输入语言 SMT 2 描述), 然后使用 Z3[41] 约束求解器求解出每条路径的 *intent* 路径条件，最后根据这些 *intent* 路径条件生成每个 <*startPoint*, *tgtAPI*> 的 *intent* 测试用例集合；第四部分能力泄露漏洞自动化测试，首先在待检测 APP 的 *tgtAPI* 之前插入 Log 语句记录当前即将要执行的 *tgtAPI* 的信息，并重新打包签名生成新的 APP(图*3.1*中 *instrumented-app*)。然后使用没有任何权限的 *test-app* 读取第三部分生成的 *intent* 测试用例自动化测试 *instrumented-app*。最后分析测试生成的测试日志，生成 *detected-app* 的检测报告，其包含能力泄露漏洞的情况以及对应的能力泄露利用。本章接下来的四个章节将详细介绍这四个部分。

3.2　方法调用图和方法控制流图构建

如算法3.1 SMain 所示，算法的输入是 *cg*, *startPoint*, *tgtAPI*, *flowInUnitDataMap (unit → flowInDataSet)*。本节将介绍方法调用图 (*cg*) 和每个方法控制流图的构建，以及 *startPoint*, *tgtAPI* 的识别。*flowInUnitDataMap* 中存储了 Android 应用中和外部 *intent* 数据相关的语句，其计算过程将在本章3.3.2控制流图层次优化中介绍。

本文基于 Soot[42] 工具构建方法调用图和每个方法的控制流图。Soot 是一个 Java 字节码[43] 分析和优化框架，其支持将 Java 字节码转化成多种中间语言，例如：Jimple[44]，方便开发者对于 Java 程序进行程序分析。Jimple 语言为三地址代码，非常规整，其在转化过程中去除了 Java 代码中没有使用的变量和赋值，且相比 Java 字节码的 200 多种指令，其仅有 15 种语句。在 Jimple 代码上进行程序分析可以极大地降低程序分析的难度和复杂度。Dexpler[45] 是一个基于 Soot 的 Android Dalvik 字节码[46] 转化框架，其可以将 Android Dalvik 字节码转化为 Jimple 中间代码。因此通过 Dexpler，我们可以使用 Soot 将 Android 应用程序转化为 Jimple 语言来分析。

Android 应用程序中并没有 Java 语言中 Main 函数的概念，一个 Android 应用程序会包含多个程序入口，例如：各个应用组件的生命周期方法。为了满足 Java 字节码分析优化框架 Soot 对 Main 函数的需求，本文采取和 FlowDroid[47]

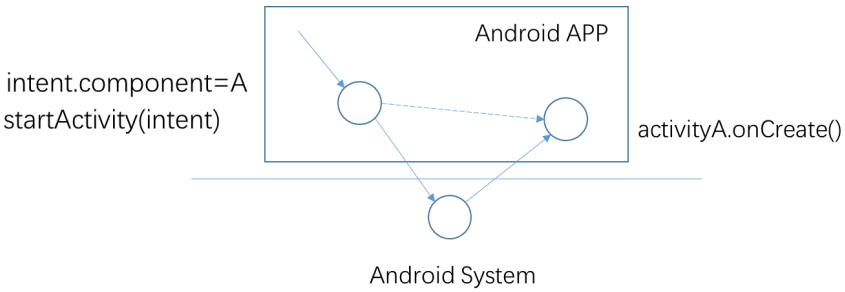

图 3.2　隐式调用示例

相同方法，构建 Android 应用程序的 DummyMain 函数，DummyMain 函数中调用了 Android 应用程序的各个程序入口。Android 应用程序中还存在很多隐式调用，例如：如图3.2所示，Android APP 中存在两个方法 *startActivity(intent)* 和 *startActivity(activityA.onCreate())* 方法，在 Android APP 中，它们不存在调用的边。但是在 Android APP 运行中，*startActivity(intent)* 方法会调用 Android System 的方法，最终 Android System 方法会调用 *startActivity(activityA.onCreate())* 方法。因此如果单纯地分析 Android APP，那么构建的 Android 应用程序的方法调用图将不完整。但是为了提高分析 APP 的效率，本文并不考虑将 Android 源码纳入我们的分析当中。本文采用与 FlowDroid[47] 与 IccTa[48] 相似的方法来解决 Android 应用程序方法调用图不完整的问题。即不断搜索识别可能存在的回调方法调用，添加到当前 Android 应用的方法调用图上，并继续搜索识别可能存在的回调方法调用，直到当前 Android 应用的方法调用图不再变化。本文搜索识别 Android 中主要的存在 Android 应用程序回调的方法，这些方法主要共分为两类：组件间通信方法和多线程方法。组件间通信方法主要有：*startActivity(Intent)* 、*startService(Intent)* 、*sendBroadcast(Intent)* 等。多线程方法主要有：*Thread.start()*、*AsyncTask.execute()*、*Handler.post(Runnable)* 等。

安卓恶意软件分析工具 Androguard[49] 提供了 *APIPermissionMap*[50]，其存储了 API 签名与权限的映射。本文使用其来识别 Android 应用程序中受权限保护的语句 (*tgtAPI*)。

Android APP 的 AndroidManifest.xml 中描述了 Android 应用的组件信息。通过解析 AndroidManifest.xml，并根据章节2.1.2AndroidManifest.xml 中所述的暴露组件规则，可以找到 Android APP 中所有的暴露组件，从而找到所有暴露组件方法 (*startPoint*)。

在 Android 应用程序方法调用图构建完成之后，开始 Android 方法调用图的第一步约简。如算法3.1 SMain 第 2 行所示，使用 *reduceCG* 函数对 Android 应用方法调用图约简。*reduceCG* 函数使用了以 *startPoint* 和 *tgtAPI* 为起点的双向 BFS 算法找到所有在 *startPoint* 和 *tgtAPI* 两点之间路径上的方法，然后从方法调用图

中去除那些路径不相关的节点，以减少搜索两点之间所有路径时的搜索空间，最后得到 *reducedCG*。

3.3　能力泄露路径查找

Android 应用的 CG 和 CFG 都是有向有环图，对于循环，我们仅展开处理一次。本文搜索的 *startPoint* 和 *tgtAPI* 的路径是欧拉路径，即路径中的每一条边都不相同。因为求两之点间所有路径是一个 np-hard 问题，且应用方法调用图可能非常巨大，因此会导致路径爆炸。这也一直是符号执行的一个重大挑战，我们必须根据我们问题的特性，进行优化近似，以缓解能力泄露漏洞利用生成过程中的路径爆炸问题。本章从方法调用图和控制流图两个层次来优化我们的符号执行。

3.3.1　方法调用图层次优化

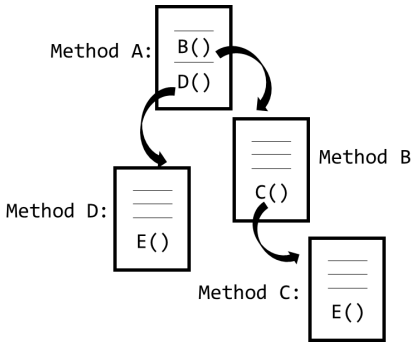

图 3.3　简化的应用方法调用图

如图3.3所示，其是简化的应用方法调用图，方法 A 是暴露组件方法，方法 E 是 *tgtAPI*。方法 A 中调用了方法 B 和方法 D，方法 B 调用了方法 C，方法 C 和方法 D 调用了方法 E。直接求取 A 到 E 的所有路径，然后符号执行求解每条路径的路径条件是不明智的。因为 A 到 E 的所有路径很多，存在路径爆炸问题。设方法 A 到方法 E 的路径集合为 $Path(A,E)$, 则 $Path(A,E) = Path(A,B) \times Path(B,C) \times Path(C,E) \cup Path(A,D) \times Path(D,E)$。我们可以分段求取 $Path(A,B)$、$Path(B,C)$、$Path(C,E)$、$Path(A,D)$ 和 $Path(D,E)$ 集合中路径的 *tgtAPI* 路径条件，然后进行组合。由于 Android 应用程序大部分方法是和外部 *intent* 数据无关，因此大部分方法无需计算 *intent* 路径条件。且获取的 *intent* 路径条件属性值的个数有限，因此组合这些 *intent* 属性值得到的 *intent* 测试用例数数量并不会很大。这样有效避免了求取所有路径时的路径爆炸问题。

如算法3.1 SMain 第 4 行到第 9 行所示，遍历 *reducedCG* 中所有方法节点，求取 *reducedCG* 中的所有分段 (调用方法与被调用方法对)，存储在 *SE* 中。然后对于每一分段，使用 *reduceCFG* 函数约简方法的控制流图，然后在约简的

方法控制流图上求解 *n* 和 *s* 之间的所有路径 (算法3.1第 12 行到第 13 行)。*reduceCFG* 函数详细约简过程将在章节3.3.2控制流图层次优化中详细介绍。然后对于每一条路径使用符号执行获取 *intent* 路径条件，并将 *intent* 路径条件存储在 *intent_path_condition_set* 中 (算法3.1中第 14 行到第 15 行)。符号执行获取 *intent* 路径条件将在章节3.4能力泄露 *intent* 路径条件约束求解中详细介绍。

如算法3.1 SMain 第 18 行到 27 行所示，将所有的 *intent* 路径条件的属性组合生成测试用例。*Intent* 各属性类型如下：action 为 String 类型，data 为 Uri 类型，category 为 Set<String> 类型，extra 为 Set<key,type→value> 类型。组合方式如下：将整个 categorySet 集合作为 *intent* 测试用例中 category 属性 (算法3.1第 22 行)。理论上应该选取 $2^{categorySet}$ 中元素作为 *intent* 测试用例中 category 属性，但是通常字符串类型不为空且不等于"" 是有意义的程序输入，因此将整个 categorySet 集合作为 *intent* 测试用例中 category 属性。这样近似也避免了因偶然性的 categorySet 数量过大而导致的测试用例数量爆炸的问题；将 extraSet 中的 extra 元素按 key 和 type 划分成多个集合，并从每个集合中选取一个 extra 属性构成的集合作为 *intent* 测试用例的 extra 属性 (算法3.1第 23 行)；每次从 actionSet 中选取一个 action 值作为 *intent* 测试用例的 action 属性；每次从 dataSet 中选取一个 data 值作为 *intent* 测试用例的 data 属性。最后将所有生成的测试用例 *intent* 保存在 *intentSet* 中 (算法3.1中第 30 行), 用于后续的能力泄露漏洞的自动化测试。

3.3.2　控制流图层次优化

组件间通信的唯一输入为 *intent* 对象，因此本文只关心和外部 *intent* 相关的语句，其他语句我们是无法控制的。而绝大多数应用的外部 *intent* 数据相关的语句占 Android 应用程序代码的分量很小，因此本文根据这个特性，使用 *flowInUnitDataMap*(Android 应用中和外部 *intent* 数据相关的语句集合) 对 Android 应用程序方法的控制流图进行约简 (算法3.1 SMain 中第 12 行)。并在这个约简的控制流图上获取调用方法与被调用方法对之间的所有路径，可以大大减少相同 *intent* 路径条件的路径条件求解。本节将详细介绍 *flowInUnitDataMap* 的计算过程以及控制流图的约简细节。

表 3.1　数据流分析起点

组件	外部 *intent* 数据流起点
Activity	*<android.content.Intent getIntent()>*
Service	*onBind(Intent)*、*onStartCommand(Intent)*、*onUnbind(Intent)*
Receiver	*onReceive(Intent)*

本文使用基于外部 *intent* 的上下文敏感、流敏感的过程间数据流分析获取 *flowInUnitDataMap*。首先，本文找到 Android 应用程序的外部 *intent* 的数据流起点，结果如表3.1 所示。Android 应用组件 Activity、Service 和 Receiver 分别可从 *<android.content.Intent getIntent()>* 方法，*onBind(Intent)* & *onStartCommand(Intent)* & *onUnbind(Intent)* 和 *onReceive(Intent)* 中获取外部 *intent*。过程间数据流分析主要由算法3.2到达定义和算法3.3转移函数组成，其主要基于到达定义数据流分析技术[51] 与 DFS 算法实现。

算法 3.2　过程间数据流分析算法——到达定义

Data: $method, inData$

Result: $flowInUnitDataMap(unit \rightarrow flowInDataSet), returnData$

```
1  Function inter-procedure-data-flow(method, inData):
2      cfgNodes ← method.cfgNodes();
3      for n in cfgNodes do
4          OUT[n] = ∅;
5      end
6      f ← cfgNodes.getFirstNode();
7      IN[f] ← IN[f] ∪ inData;
8      changed ← cfgNodes;
9      while changed ≠ ∅ do
10         choose a node n in changed;
11         changed = changed − n;
12         for all nodes p in predecessors(n) do
13             IN[n] ← IN[n] ∪ OUT[p];
14         end
15         OldOUT ← OUT[n];
16         OUT[n] ← transfer_function(IN, n, flowInUnitDataMap);
17         if OldOUT ≠ OUT[n] then
18             for node s in successors(n) do
19                 changed ← changed ∪ s;
20             end
21         end
22     end
23     l ← cfgNodes.getReurnNode();
24     if l.returnLocal in IN[l] then
25         returnData ← l.returnLocal.data;
26     end
27 End Function
```

算法3.2到达定义的输入是被分析的方法 *method* 和其与 *intent* 数据相关的参数 *inData*。算法中每个被分析的方法对应一个控制流图，方法中每条语句对应控

制流图上的一个节点。每个节点 n 拥有有 IN 和 OUT 集合，分别表示节点 n 执行之前 *intent* 数据相关的变量集合和节点 n 执行之后的 *intent* 数据相关的变量集合。如算法3.2到达定义第 3 行到第 7 行和第 12 行到第 14 行所示，每个节点的 OUT 集合初始为空；每个节点的 IN 集合初化为该节点所有父节点的 OUT 集合的并集，且控制流图的第一个节点的 IN 集合为 {*inData*}。每个节点实际执行之后，和 *intent* 数据相关的变量集合会发生变化，对于每一个节点使用 transfer_function 函数计算其 OUT 集合变化 (算法3.2第 16 行)。如果在 transfer_function 函数执行后，节点的 OUT 集合发生变化，则将其子节点加入到待处理节点集合 (*changed*)(算法3.2第 17 行到第 21 行)。当控制流图中所有节点的 IN 集合不再变化时，当前算法结束，即此时已经完成每个程序点 (语句)IN 集合的计算。算法返回值 *returnData* 用于记录当前被分析方法 *method* 返回值是否和和 *intent* 数据相关。当 *method* 返回值和 *intent* 数据相关时，*returnData* 为 *method* 返回值的 *intent* 数据，否则 *returnData* 为空；算法返回值 *flowInUnitDataMap* 中存储了每条语句的 IN 集合。

$OUT[n]$ 变化具体由算法3.3转移函数实现。其变化是算法3.3依据以下公式计算得到: $OUT[n] = GEN[n] \cup (IN[n] - KILL[n])$，其中 $GEN[n]$ 是执行这个节点之后新增的和 *intent* 数据相关的变量的集合，$KILL[n]$ 是执行这个节点之后被重新赋值且重新赋值之后与 *intent* 数据不相关的变量集合。如算法3.3第 6 行到 8 行所示，如果节点使用了和 *intent* 数据相关变量，则认为节点中被定义的变量 *defLocal* 和 *intent* 数据相关，将其加入到 $GEN[n]$ 中。并将每个节点和流入每个节点的 *intent* 数据记录在 *flowInUnitDataMap* 中，其中记录的 *intent* 数据包含数据的类型信息和数据来源位置 (由什么语句产生)。可能的数据类型有 *intent* 对象，*intent action* 属性，*intent data* 属性，*intent category* 属性和 *intent extra* 属性。其中 *intent extra* 数据类型信息还包含 extra 属性可能的 *key* 值。如算法3.3第 13 到第 21 行、第 23 行到 28 行所示，如果节点包含方法调用，则递归调用算法3.2到达定义进行分析。如果该方法调用的返回值与 *intent* 数据相关，则将 *defLocal* 加入到 $GEN[n]$ 中 (算法3.3第 16 行到 18 行)。对于每个方法，在不同的上下文调用可能会有不同的数据输入。因此对每个方法进行数据流分析时，应该考虑不同上下文的数据流输入 (即上下文敏感的过程间数据流分析)。针对某个方法的不同的数据流输入，在调用处创建原函数的一份复制以考虑不同类型数据流输入, 即本文中过程间数据分析分析的单位是 *MethodSummary*(*Pair(method,inData)*)。由于本文只考虑与外部 *intent* 相关的数据流，且方法的参数输入类型确定以及 *intent* 数据类型有限（Intent、Action、Data、Category、Extra）, 所以创建的方法复制数量有限。因此基于克隆的上下文敏感的过程间数据流分析可以在保证数据流分析的精确度的同时，不会引起很大的性能开销。

算法 3.3　过程间数据流分析算法——转移函数

```
Data: IN, n, flowInUnitDataMap
Result: OUT[n]
Function transfer_function(IN, n, flowInUnitDataMap):
    KILL[n] ← ∅;
    GEN[n] ← ∅;
    useLocals ← n.getUsedLocals();
    defLocal ← n.getDefLocal();
    if useLocals ∩ IN[n] ≠ ∅ then
        GEN[n] = GEN[n] ∪ defLocal;
        flowInUnitDataMap.put(n.unit, intentData);
    else
        KILL[n] = KILL[n] ∪ defLocal
    end
    if defLocal ≠ null then
        if m=getMethodCall(n) ≠ null then
            if Pair(m,arg) not in hasProcessedMethodSummarySet then
                returnData
                  ←inter-procedure-data-flow(m, arg.data).returnData;
                if returnData ≠ null then
                    GEN[n] ← GEN[n] ∪ defLocal
                end
                hasProcessedMethodSummarySet.add(Pair(m, arg));
            end
        end
    else
        if (m=getMethodCall(n)) ≠ null then
            if m not in hasProcessedMethodSummarySet then
                inter-procedure-data-flow(m, arg.data).returnData;
                hasProcessedMethodSummarySet.add(Pair(m, arg));
            end
        end
    end
    OUT[n] = GEN[n] ∪ (IN[n] − KILL[n]);
End Function
```

通过上述外部 *intent* 过程间数据流分析，我们得到 Android 应用程序中和外部 *intent* 数据相关的语句。本文通过其去除控制流图中和外部 *intent* 数据不相关的语句，以此约简控制流图，减小调用方法与被调用方法对之间的路径数量。本章以一个示例介绍本章控制流图的约简。如代码块3.1所示，方法 *doTask1* 为 CFG

约简前，从第一条语句 *int pid = 643* 到 *killProcess(pName,pid)*(*tgtAPI*) 共有 6 条路径。通过数据流分析之后，我们知道 *if y< 6* 和 *if z >7* 条件语句块都和外部 *intent*

```
// before reduction
    public void doTask1(String pName) {
        int pid=643;
        if (x > 5)
        {
            if (intent.getAction.equals("kill"))
            {   //branch1
                String key="pid";
                pid=intent.getIntExtra(key);
                ...
            } else {...}//branch 2
        }
        else
            {
                if (y < 6) {...}//branch 3
                else {...}//branch 4
                if (z > 7) {...}//branch 5
                else {...}//branch 6
            }
        killProcess(pName,pid);// tgtAPI
    }
// after reduction
    public void doTask2(String pName) {
        int pid = 643;
        if(x>5)
        {
            if (intent.getAction.equals("kill"))
            {   //branch 1
                String key="pid";
                pid=intent.getIntExtra(key);
            }else {}//branch 2
        }
        else {}//branch 3
        killProcess(pName,pid);//tgtAPI
    }
```

代码块 3.1　CFG 约简示例

不相关，因此删掉 *if y< 6* 和 *if z>7* 条件语句块，最后得到约简后的方法 *doTask2*，方法 *doTask2* 中语句 *int pid = 643* 和语句 *killProcess(pName,pid)*(*tgtAPI*) 只有 3 条路径。因为 Android 代码中和外部 *intent* 数据语句相关语句较少，因此可以大大减小调用方法与被调用方法对之间的路径数量。需要注意的是即使 branch2 中的

语句和 *intent* 不相关，被全部删除，但是 branch2 的路径仍然存在，因为 branch1 中存在与 *intent* 相关语句，导致 *if x> 5* 不可以删除。

3.4　能力泄露 *intent* 路径条件约束求解

计算路径的 *intent* 路径条件过程如算法3.4所示，其输入是路径 *path* 和 *flowInUnitDataMap*。*flowInUnitDataMap* 记录了 Android 应用程序中所有和外部 *intent* 数据相关的语句以及和流入这些语句的 *intent* 数据 (*intentData*)，*intentData* 中存储的内容包括数据的数据来源类型、数据产生的位置。如算法3.4第 3 行到第 16 行所示，分析路径中每一条语句，如果这条语句和 *intent* 相关 (*intentData* 不为空)，则继续分析该语句。否则，则分析下一条语句。分别根据语句的类型对语句进行处理，生成相应的约束表达式，并添加到 *pathExprSet*。本章工具支持 android.content.Intent 类、android.os.Bundle 类主要方法调用语句的处理和字符串类型、数值类型数据比较语句的处理，前者会产生 *intent* 数据，后者会影响 *intent* 路径条件。

算法 3.4　符号执行求解路径条件

Data: $path, flowInUnitDataMap$
Result: $intent\_path\_condition$
1 **Function** `symbol_execution`(*path, flowInUnitDataMap*) **:**
2 　$pathExprSet \leftarrow \varnothing$;
3 　**for** *unit* ***in*** *path* **do**
4 　　$intentData \leftarrow flowInUnitDataMap.get(unit)$;
5 　　**if** $intentData \neq null$ **then**
6 　　　**if** *unit is form of if(op1.equals(op2))* **then**
7 　　　　$valueAssignExpr \leftarrow genValueAssignExpr(intentData, op1, op2)$;
8 　　　　$pathExprSet \leftarrow pathExprSet \cup valueAssignExpr$;
9 　　　**end**
10 　　　**if** *unit is form of if(op1 op op2)* **then**
11 　　　　$valueCmpExpr \leftarrow genValueCmpExpr(intentData, op1, op2)$;
12 　　　　$pathExprSet \leftarrow pathExprSet \cup valueCmpExpr$;
13 　　　**end**
14 　　　...
15 　　**end**
16 　**end**
17 　$intent\_path\_condition \leftarrow Z3Solve(pathExprSet)$;
18 **End Function**

例如：如果路径中一条语句是

$$if(!str.equals("success"))$$

且当前路径为 true 分支，str 来源于 *intent* 的 action，则 *valueAssignExpr* 是

$$(declare-const\quad str\quad String)$$

$$(assert\quad (not\quad (=\quad str\quad "success")))$$

$$(assert\quad (=\quad action\quad str))$$

如果语句是

$$if(x > 5)$$

且当前路径为 true 分支，x 来源于 *intent* 的 extra 属性，extra 属性 *key* 值为 *fromPush*，则 *valueCmpExpr* 是

$$(declare-const\quad x\quad int)$$

$$(declare-const\quad extra-id\quad int)$$

$$(assert\quad (=\quad (keyIs\quad extra-id\quad "fromPush")\quad true))$$

$$(assert\quad (>\quad x\quad 5))$$

$$(assert\quad (=\quad extra-id\quad x))$$

其中 *id* 唯一标识一个 extra，extra 属性的 *key* 和 *id* 均来源于 *intentData*，都是由过程间数据流分析产生。将上述 *pathExpr* 输入到 Z3 约束求解器中 (算法3.4第 17 行)，求解将得到 *key* 为 *fromPush*，*type* 为 *int*、*value* 为 *6* 的 extra 属性，以及值为*!success* 的 action 属性。实际 APP 生成的路径条件约束如图3.4所示。

3.5　能力泄露漏洞自动化测试

本文插桩通过 Soot 工具实现，Soot 支持 Jimple 中间代码和 Davilk 字节码的相互转化以及 Jimple 中间代码的修改，因此可通过对 Jimple 中间代码的修改映射为对 Android 应用程序的修改，从而实现对 Android 应用程序的插桩。通过3.2节方法调用图和方法控制流图构建可知 Android 应用程序所有的 *tgtAPI*, 本文在所有 *tgtAPI* 之前插入 Log 语句记录 *tgtAPI* 语句的信息以及 APK 信息。插桩完成之后重新签名形成新的 Android APP(即图3.1中 *instrumented-app*)。图3.1中 *test-app* 为测试发起者，其没有任何权限。它的功能是读取测试用例文件，并根据测试用例文件内容生成 *intent* 对象, 发起组件间通信测试 *instrumented-app*。如果 *tgtAPI* 被触发，则说明出现越权操作 (即应用存在能力泄露漏洞)，*instrumented-app* 会

```
(declare-const $z0_boolean_h_com.baidu.android.pushservice.i_141 Bool )
(declare-const $z0_boolean_h_com.baidu.android.pushservice.i_133 Bool )
(declare-const $r2_pushservice.i_147 String )
(declare-const $r4_pushservice.i_147 String )
(declare-const $r1_pushservice.i Object )
(declare-const $z0_boolean_h_com.baidu.android.pushservice.i_157 Bool )
(declare-const $z0_boolean_h_com.baidu.android.pushservice.i_12 Bool )
(declare-const $r2_pushservice.i_163 String )
(declare-const $r3_pushservice.i_163 String )
(assert (not (= $z0_boolean_h_com.baidu.android.pushservice.i_157 true)))
(assert (not (= $z0_boolean_h_com.baidu.android.pushservice.i_141 true)))
(assert (not (= $z0_boolean_h_com.baidu.android.pushservice.i_12 true)))
(assert (= (containsKey $r1_pushservice.i "app_id") true))
(assert (= $r2_pushservice.i_163 $r3_pushservice.i_163))
(assert (= $r2_pushservice.i_163 "null" ))
(assert (= $r2_pushservice.i_147 (str.++ "ZMS!" $r4_pushservice.i_147)))
(assert (not (= $z0_boolean_h_com.baidu.android.pushservice.i_133 true)))
(assert (= (containsKey $r1_pushservice.i "user_id") true))
(assert (= (containsKey $r1_pushservice.i "package_name") true))
(assert (= $r2_pushservice.i_147 "null" ))
(check-sat-using (then qe smt))
(get-model)
```

图 3.4　实际 APP 路径条件约束示例

产生测试日志，测试日志中记录了能力泄露漏洞的详情。由章节2.2 Android 权利模型可知，Android M 之后，Android 应用的危险权限是由用户动态授予。为了保障测试自动化和检测出所有可能触发的能力泄露漏洞，本文使用的动态测试的模拟器为 Android4.4，每一个被安装的 APP，系统将授予其申请的全部权限，保证 APP 拥有其所申请的全部权限。图3.5为实际测试产生的部分测试日志示例，*ZMSInstrument* 是日志 tag，用来标识日志为本文工具所插桩 Log 语句产生。从该日志可以得到 A.apk 的 *<com.igexin.a.a.b.a.a.e: void a_()>* 方法的 *34* 行的 *virtualinvoke r2. < android.os.PowerManagerWakeLock: void release()>()* 语句产生了能力泄露，泄露类型为权限 *android.permission.WAKE_LOCK*。本文编写自动化测试 python 脚本，不断使用 *test-app* 测试 *instrumented-app*。最后，分析测试日志，生成待检测 APP 的检测报告，检测报告中包含待检测 APP 的能力泄露漏洞情况以及触发能力泄露漏洞的测试用例 (利用)。

```
ZMSInstrument(4458):
#Instrument# A.apk
#method# <com.igexin.a.a.b.a.a.e: void a_()>
#unitPoint_before# virtualinvoke $r2.<android.os.PowerManager$WakeLock: void release()>()
#lineNumber# 34
#unitPoint_permission# android.permission.WAKE_LOCK
```

图 3.5　测试日志示例

3.6　本章小结

本章介绍了本文第一个工作基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成的具体实现过程。本章首先使用 Soot 构建 Android 应用的方法调用图和每个方法的控制流图，由于 Android 应用广泛存在隐式调用，因此构建的 Android 应用方法调用图不完整。本章不断搜索识别出可能存在的的回调方法调用，添加到当前方法调用图中，以构建完整的方法调用图。在方法调用图的基础上，本章使用 *APIPermissionMap* 文件和 AndroidManifest.xml 找到所有可能的能力泄露点 (*tgtAPI*) 和暴露组件方法起点 (*startPoint*)。然后本章使用基于外部 *intent* 上下文敏感，流敏感的过程间数据流分析找到应用中每一个和外部 *intent* 数据相关的语句以及该语句的数据来源存储在 *flowInUnitDataMap*。使用 *flowInUnitDataMap* 对每个方法的控制流图进行约简，去除那些和外部 *intent* 不相关的语句。然后分阶段使用符号执行求解每个方法内的执行路径的条件，最后组合各阶段符号执行的路径条件结果生成最后的测试用例，然后使用测试用例测试 *instrumented-app*。最后分析 *instrumented-app* 的测试日志，生成待检测 APP 的分析报告。分析报告中记录了待检测 APP 的能力泄露情况，以及触发能力泄露漏洞的利用。

第 4 章　基于动态反馈的 Android 应用间能力泄露漏洞利用的自动化生成

基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成工具中使用了静态符号执行，尽管本文在其实现过程中处理了绝大数的语句，保证了一定的准确率，但是无法从根本上解决静态分析的缺陷，因此本章试图从动态分析角度解决这个问题。我们提出了基于动态反馈的 Android 应用间能力泄露漏洞利用的自动化生成工具，希望通过 Android 应用的动态运行信息生成有效的测试用例，触发 Android 应用间能力泄露漏洞。

4.1　方法概述

如图4.1所示, 基于动态反馈的 Android 应用间能力泄露漏洞利用的自动化生成工具共分为静态分析和动态测试两个部分。首先对待检测 APP(图4.1中 *detected-app*) 进行静态分析找到需要插桩打印的包含 *intent* 路径条件信息的变量(关键变量) 以及插桩的位置点，自动化生成插桩语句，并对待检测 APP 进行插桩，重新打包签名后得到新的 APP(图4.1中 *instrumented-app*)。然后使用没有任何权限的 *test-app* 发送 *intent* 测试用例动态测试 *instrumented-app*，*instrumented-app* 在测试过程中会产生测试日志。从日志中获取 APP 的动态运行信息重新生成新的 *intent* 测试用例测试 *instrumented-app*，直到日志中没有新的动态运行信息产生。测试日志中还包含能力泄露漏洞信息，最后分析测试日志，得到待测 APP 的能力泄露漏洞报告。接下来，本章将从两个部分详细介绍。

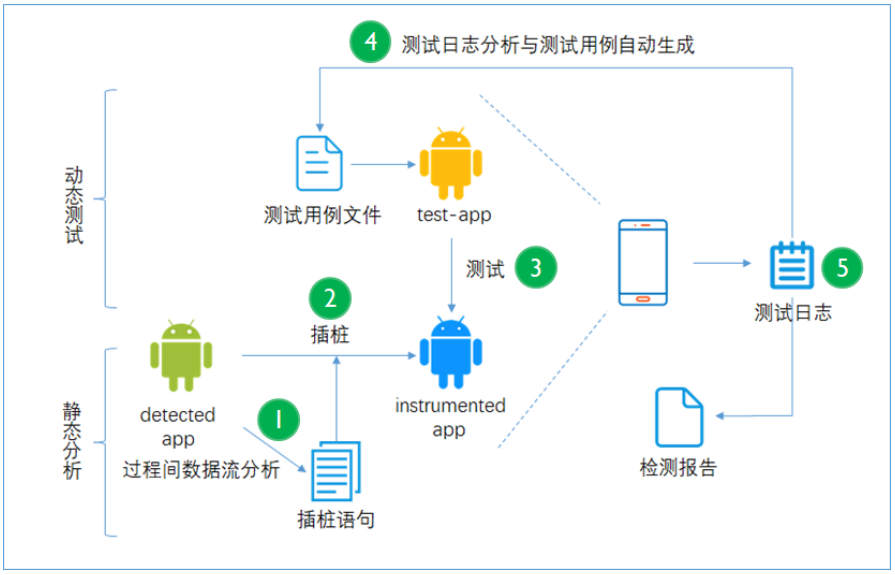

图 4.1　动态反馈能力泄露利用生成工具流程图

4.2　静态分析

本工具在构建 APP 方法调用图和方法和控制流图上，采取和符号执行能力泄露利用生成工具相同的技术和方法 (章节3.2 方法调用图和方法控制流图构建)。使用 Androguard 提供的 APIPermissionMap 文件识别 Android 应用程序中所有受权限保护的语句 (*tgtAPI*)，存储在集合 *tgtAPISet* 中。本工具中采用上下文敏感、流敏感的过程间数据流分析技术获取 Android 应用程序中各个程序位置点 (语句) 的外部 *intent* 数据流情况，以此获取关键变量以及插桩位置点。本工具中使用的过程间数据流分析算法和章节3.3.2中的过程间数据流分析算法相同，其以获取外部 *intent* 的方法为数据流分析的起点，主要使用了到达定义数据流分析技术和 DFS 算法。该过程间数据流分析算法的结果为 *flowInUnitDataMap (unit → flowInDataSet)*，*flowInUnitDataMap* 中存储了所有和外部 *intent* 数据相关的语句集合，以及流入每条语句的 *intentData*。*intentData* 中记录了 *intentData* 的类型 (intent 对象、action、category、data、extra) 以及 *intentData* 产生的位置 (由什么语句产生)。迭代 *flowInUnitDataMap*，找到其中所有的条件语句 (*if*)，以及流入条件语句的 *intentData* 存储在 *ifControlDataMap* 中。

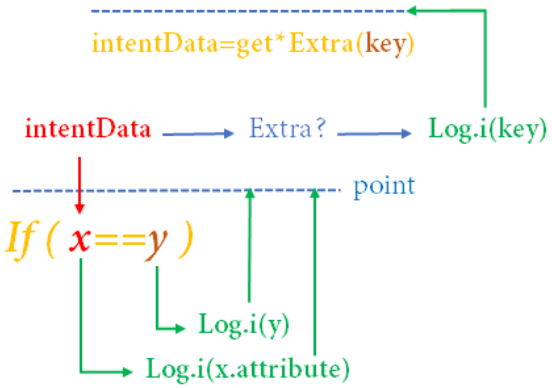

图 4.2　插桩示例

迭代 *tgtAPISet* 集合，对于每一个 *tgtAPI* 生成打印 *tgtAPI* 信息的 Log 语句，并插桩在 *tgtAPI* 之前，打印的 *tgtAPI* 信息包括 *tgtAPI*、*tgtAPI* 对应权限以及当前 APP 等信息; 迭代 *ifControlDataMap*，对于每一个条件语句生成打印关键变量的 Log 语句和记录流入的 *intentData* 属性的 Log 语句，并插桩在条件语句之前。关键变量是条件语句中包含 *intent* 路径条件信息的变量，其决定能否通过条件语句。如图4.2所示, 条件语句 $if(x == y)$ 中 x 为 *intentData*, 则 y 为关键变量。生成记录关键变量 y 值的 Log 语句为 $Log.i(y)$，生成记录 x 变量中数据来源属性的 Log 语句 $Log.i(x.attribute)$。通过动态运行，可从运行日志信息中得到关键变量 y 的值, 假设为 v；以及 x 的数据来源信息，假设为 *intent.action*。根据这些动态运行信息重新生成 *intent* 测试用例，并令 $intent.action = v$，则该测试用例能通过

条件语句 $if(x == y)$, 触发更多代码。如果图4.2中 *intentData* 是 extra，则需要在 *intentData* 产生位置 $get*Extra(key)$ 前插入打印 key 的 Log 语句，从而从日志获取 extra 属性的 key 值。插桩完成之后重新打包签名 APP 得到 *instrumented-app*，然后使用 *test-app* 动态测试 *instrumented-app*。

4.3　动态测试

算法4.1是本章测试用例生成算法，本章最初使用不包含任何数据的 *intent* 对象测试 APP（算法4.1的第 7 行到 9 行），然后分析产生的测试日志，将和 *intent* 路径条件相关的测试信息存储在 intent-test-info。如果 intent-test-info 为空，即 intent-test-info 的 actionSet、categorySet、dataSet 和 extraSet 为空, 则停止当前暴露组件的测试，继续下一个暴露组件的测试。否则，则将新的反馈信息加入到 intent 属性集合中（算法4.1 第 29 行和第 30 行），继续生成针对该暴露组件的测试用例。

一个条件语句存在多个分支，为了让本章工具的测试用例可以尽量覆盖更多的分支，本文对获取的反馈信息进行变异 (算法4.1 第 29 行和第 30 行)。*Intent* 对象存储数据的主要数据类型为字符串类型和数值类型，因此本章工具只针对这两种数据类型数据的变量进行变异。针对所有字符串数据类型的变量，本文添加两种变异值，分别是 *null* 和""；针对数值类型数据，本文通过一个示例说明其变异方式：新得到 extra 反馈属性信息：key:“fromPush”, type: int ,value: 0 则生成可能满足条件语句某个分支的另外两个 extra 属性为：1) key:“fromPush”, type: int , value: 1 ; 2) key:“fromPush”, type: int , value: -1; 并将这两个属性加入到 extraSet，用于下一次的测试用例生成。

如算法4.1 的第 11 行到 25 行所示，本章根据新的反馈信息重新生成新的测试用例测试 APP。测试用例生成方式与本文第一个工作的测试用例生成方法相同 (算法3.1)。*intent* 对象中 category 属性类型是 Set<String>，截止上一次测试得到的 category 值的集合为 categorySet，将整个 categorySet 集合作为 *intent* 对象中 category 属性（算法4.1第 11 行）；intent extra 属性类型是 Set<key,type→value>，截止上一次测试得到的 extra 值的集合为 extraSet，本章将 extraSet 按 key、type 划分成不同集合，每次从这些不同集合中挑选一个值组合成一个 *intent* 测试用例的 extra 属性（算法4.1 第 12 行）。如算法4.1 第 17 行到 20 行所示，每次生成一个测试用例去测试 APP，同时记录下已经测试过的测试用例，保证测试的测试用例不重复。不断根据测试日志的 intent-test-info 生成新的测试用例测试，直到 intent-test-info 为空。待检测 APP 的能力泄露漏洞结果存储在 *capabilityLeakSet* 中。

算法 4.1　基于动态反馈的测试用例生成方法

```
Data: detected-app
Result: capabilityLeakSet
Function Main(detected-app):
    ECSet ← getEC(detected-app.AndroidManifestXml);
    capabilityLeakSet ← ∅;
    for exported-component in ECSet do
        actionSet, dataSet, categorySet, extraSet ← ∅;
        while true do
            if isFirstTest then
                initial-intent= newIntent(exported-component);
                logFile ← testApp(detected-app,initial-intent);
            else
                selectCategorySet.add(categorySet);
                selectExtraSet ← combineWithDiffKeyAndType(extraSet);
                for a in actionSet do
                    for d in dataSet do
                        for c in selectCategorySet do
                            for e in selectExtraSet do
                                intent = newIntent(a,d,c,e,exported-component);
                                if hasNotTested(intent) then
                                    logFile ← testApp(detected-app,intent);
                                end
                            end
                        end
                    end
                end
            end
            oneTestCLSet,intent-test-info= analyseLog(logFile);
            capabilityLeakSet ← capabilityLeakSet ∪ oneTestCLSet;
            if intent-test-info ≠ ∅ then
                actionSet,dataSet,categorySet,extraSet
                    .addAll(intent-test-info,mutation(intent-test-info));
            else
                break;
            end
        end
    end
End Function
```

4.4　本章小结

本章介绍了本文第二个工作基于动态反馈的 Android 应用间能力泄露漏洞利用的自动化生成，其从获取程序运行动态信息的角度生成 *intent* 测试用例，旨在缓解上章工作中存在的缺陷。其通过基于外部 *intent* 的上下文敏感、流敏感的过程间数据流分析找到插桩的位置点和包含 *intent* 路径条件信息的变量，自动化生成插桩语句插桩待检测 APP，得到 *instrumented-app*。然后使用不包含任何权限的 *test-app* 读取测试用例动态测试 *instrumented-app*，并不断根据 *instrumented-app* 动态运行的结果，生成新的测试用例测试 *instrumented-app*，直到 *instrumented-app* 没有新的动态运行信息产生。最后，从 *instrumented-app* 的测试日志获取待检测 APP 的能力泄露漏洞情况，包括能力泄露漏洞类型以及能力泄露漏洞的利用。

第 5 章　实验结果与评估

5.1　实验准备

5.1.1　实验环境

本文所有实验都是在处理器为 3.6GHz Intel Core i7，运行内存为 32GB，操作系统为 ubuntu 18.04 x64 LTS 的计算机下完成。本文工具依赖的编程环境为 java 1.8，python 3.6。工具测试部分在 Android SDK 提供的 Nexus 5X API 19 模拟器下完成，自动化测试脚本由 python 和 shell 构成。

5.1.2　实验数据集

本文实验数据集由 2017 年豌豆荚应用市场选取的 18 个类别应用组成。每个类别选取排名靠前的 45 个应用，共 810 个应用，然后剔除使用加固技术[52]、Soot 分析或插桩失败的应用[45]，最后剩下 611 个应用①。这 611 个应用的大小和代码量 (SLOC:source lines of code) 如图所示5.1。应用平均大小为 19.76MB，平均代码量为 39.7 万行。

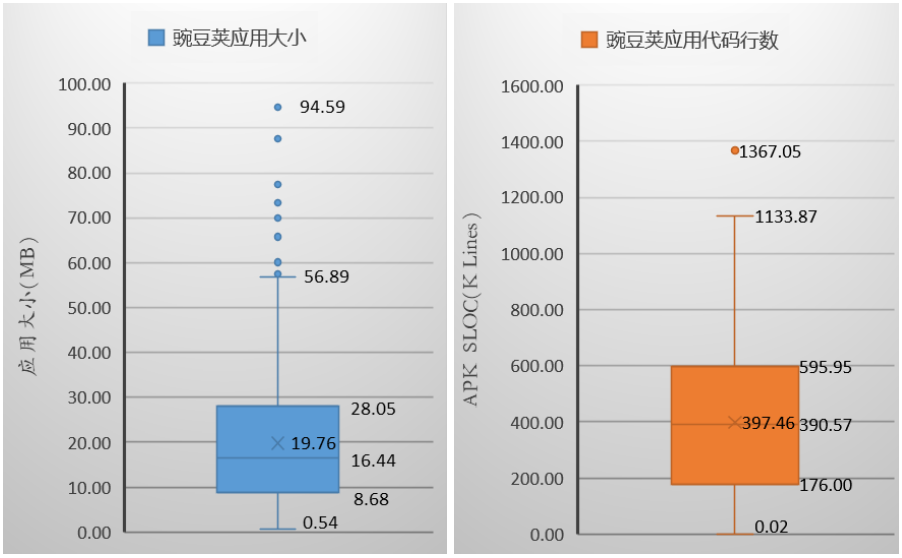

图 5.1　数据集应用大小和代码量分布

①需要解释的是：为了防止应用被反编译和破解，加固技术和防重打包技术越来越盛行。加固技术导致静态分析工具获得的并不是应用真正的源码，从而导致应用分析失败。而防重打包技术导致插桩重新签名之后的 APP 不能运行。因为本文工具是为开发者设计，开发者可以在应用发布之前 (使用加固技术和防重打包技术之前) 使用本文工具，因此本文工具仍然有效。

5.2　基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成

为了更好地评估此工具，本文将从以下几个问题的角度进行阐述：

（1） 符号执行工具计算 intent 路径条件的准确率如何？

（2） 符号执行工具可以用于实际应用的检测吗？能检测出能力泄露问题吗？

（3） 自动化确认漏洞对开发者有多重要？本文工具能减少多少误报？

（4） 工具使用了符号执行技术，其执行效率怎么样？

5.2.1　Intent 路径条件计算准确率

从章节3.4可知，基于符号执行的 Android 应用间能力泄露利用的自动化生成工具中需要着对路径中语句进行处理生成路径条件约束，但是相同含义的程序代码的表达形式可能是多种多样的，因此不可能处理所有的情况。本文选取绝大多数 APP 的语句使用方式作为我们工具处理的目标，为了评估工具生成路径条件约束的有效性，本文使用以下公式评价 *intent* 路径条件计算的准确率:

$$AC = \frac{size(allStatementsSet) - size(unhandledSet) - size(incorrectSet)}{size(allstatementsSet)}$$

其中 *allStatementsSet* 代表所有需要处理的语句集合，*unhandledSet* 代表没有处理的语句集合，*incorrectSet* 代表处理了但是没有获取正确值的语句集合。例如：*unhandledSet* 中包括一些不常见的 API 语句，或者无法得出有效信息的 API 语句，符号执行工具不对其处理。例如：Intent.hasFileDescriptors() 用于判断 intent 的 extra 属性是否包含文件描述符，此方法被用到的很少，且无法从此语句推测出 extra 属性。*incorrectSet* 中包括一些无法获取变量值的语句，例如：Intent.getExtra(key) 语句中 key 变量可能来源于复杂计算或者网络。

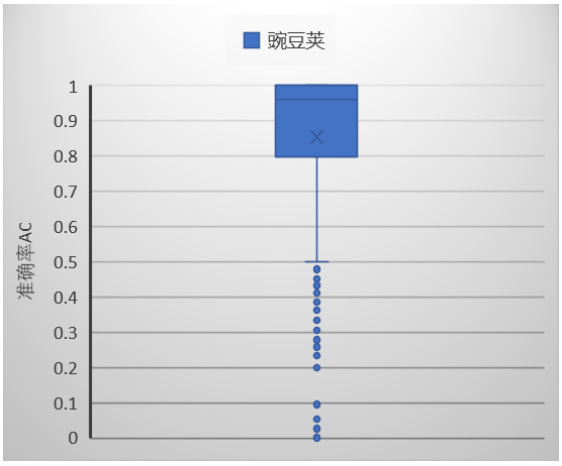

图 5.2　数据集 Intent 路径条件计算准确率分布

使用本文工具分析数据集 611 个 APP 时，分别记录 *allStatementsSet*、*unhan-*

表 5.1　数据集 Intent 路径条件计算准确率箱型图主要参数

非异常范围最小值	0.500
下四分数	0.799
中位数	0.961
上四分位数	1
非异常范围最大值	1
平均值	0.853

dledSet 和 *incorrectSet*。对于每一个 APP 使用公式计算 *AC*，最后将 611 个 APP 的 intent 路径条件计算准确率结果绘制成箱型图，箱型图如图5.2所示。箱型图5.2主要参数如表5.1所示，从中可以看出 611 个 APP intent 路径条件计算准确率非异常范围最小值为 50%，下四分位数为 79.9%, 中位数为 96.1%, 上四分位数为 100% ，平均值为 85.3%。因此可以得出：611 个 APP 的平均 intent 路径条件计算准确率为 85.3%；总共有 3/4 的 APP Intent 路径条件计算准确率在 79.9% 以上，且准确率低于 50% 的 APP 属于异常点，属于极少数；一半的 APP 路径条件计算准确率在 96.1% 以上，且有 1/4 的 APP 路径条件计算准确率为 100%。因此本文工具可以准确计算绝大多数 APP 的 intent 路径条件。

5.2.2　能力泄露检测结果

符号执行工具检测实验数据集的能力泄露缺陷结果如表5.2所示，工具共发现 16 种共 6566 个能力泄露类型缺陷。表5.2第一栏为能力泄露缺陷类型，第二栏 AppUseCount 为出现该能力泄露类型 APP 的数量，第三栏 AllCount 为该能力泄露类型出现在所有 APP 中的能力泄露点 (测试用例可达的 *tgtAPI* 称为一个能力泄露点) 的数量。

实验数据集 611 APP 中共有 545 个 APP 存在能力泄露缺陷，占 89%，其中存在一些严重能力泄露漏洞，例如：DISABLE_KEYGUARD 权限能力泄露，DISABLE_KEYGUARD 权限是实现锁屏功能的主要权限；KILL_BACKGROUND_PROCESSES 能力泄露，KILL_BACKGROUND_PROCESSES 是查杀后台进程的权限。也存在一些危害性较弱的能力泄露，例如：BROADCAST_STICKY 能力泄露，其被利用将导致应用的广播不能正常工作；ACCESS_FINE_LOCATION 能力泄露，其被利用可导致应用耗电问题；CHANGE_WIFI_STATE 能力泄露，其被利用将导致手机 wifi 可以任意打开或者关闭。同时，我们注意到在动态测试部分，本文构建的测试用例使得很多 APP 崩溃，这说明当前 APP 处理外部 *intent* 的代码鲁棒性较差，因此可被利用于恶性竞争。能力泄露漏洞检测的意义在于发现那些影响应用内

部状态，或者执行特权行为的外部越权路径，能力泄露缺陷的危害性主要取决于能力泄露的权限类型以及应用如何使用这些能力。能力是改变某项状态或者执行某项行为的能力泄露缺陷是很容易被利用的，本文将在后序章节5.5介绍两个实际能力泄露利用示例。

表 5.2　符号执行方法能力泄露漏洞检测结果

Permission	AppUseCount	AllCount
DISABLE_KEYGUARD	8	9
CHANGE_WIFI_MULTICAST_STATE	2	2
RECEIVE_BOOT_COMPLETED	1	3
SET_WALLPAPER_HINTS	2	2
BROADCAST_STICKY	153	225
ACCESS_FINE_LOCATION	158	482
KILL_BACKGROUND_PROCESSES	2	3
ACCESS_COARSE_LOCATION	148	323
CHANGE_WIFI_STATE	4	5
ACCESS_NETWORK_STATE	470	3426
GET_TASKS	311	771
WAKE_LOCK	107	211
ACCESS_WIFI_STATE	321	1022
SET_WALLPAPER	1	1
BLUETOOTH	9	11
READ_PHONE_STATE	55	70

5.2.3　误报约简能力评估

即使静态工作也可以有效地检测 Android 应用间能力泄露漏洞，但是其不能确定漏洞的真实存在。为了说明漏洞利用对于开发者的重大意义，本文使用本文静态分析的漏洞检测结果，和实际本文工具产生漏洞利用的结果对比。本文采用以下公式计算误报率:

$$FPR = \frac{ALL - SE}{ALL}$$

其中 ALL 是本文静态分析检测的结果，SE 为本文工具检测的结果 (已经确认可触发的漏洞)。由于缺乏标准数据集，本文在这里近似认为本文静态分析的漏洞检测结果包含程序实际所有的漏洞。由章节5.2.1可知，本文符号执行可以精确计算 *intent* 路径条件，因此近似认为 SE 为应用程序实际可触发的全部漏洞。根据

以上公式，分别计算本文静态分析检测在 611 个 APP 上的误报率，结果如图5.3所示。从图中可知，静态分析检测在绝大多数 APP 上的检测结果的误报率都非常高，平均误报率为 88.50%。因此漏洞确认对于开发者非常重要，其很大程度可以减少开发者确认 bug 的工作量。

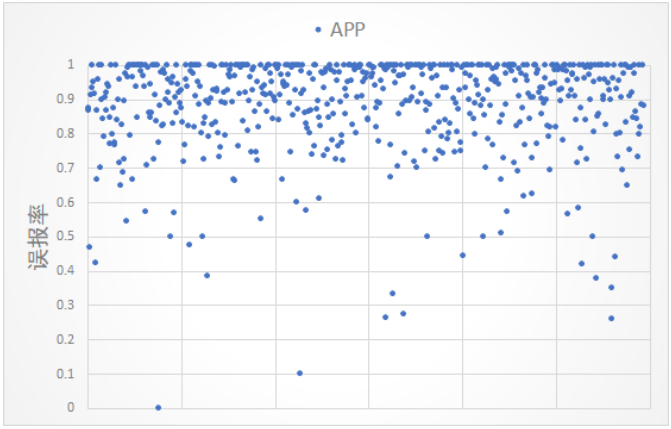

图 5.3　静态分析检测误报率散点图

5.2.4　工具执行效率

使用符号执行工具分析实验数据集的运行时间如表5.3所示，分别包括符号执行，插桩和动态测试三个阶段的时间。其中符号执行阶段最短时间为 0.19s，最长时间为 3.73h, 平均时间为 4.01 minutes; 插桩阶段最短时间为 0.15s，最长时间为 1.44 minutes，平均时间为 19.42s；动态测试阶段最短时间为 6.14s，最长时间为 26.86 minutes，平均时间为 2.83 minutes。因此平均每个 APP 的检测时间为 7.16 minutes，其满足实际应用场景的时间需求。尽管符号执行阶段分析时间最长为 3.73h，但是实际中出现这种分析时间的 APP 概率很低，且对于高精度的符号执行这个时间仍然是可以接受的。APP 符号执行阶段的时间主要取决于 APP 对外部 *intent* 的处理，主要包括两个方面：APP 中外部 *intent* 数据相关的方法数量和 APP 方法中外部 *intent* 相关的分支语句数量。当 APP 中存在较多外部 *intent* 数据相关的方法以及方法中存在较多外部 *intent* 相关的分支语句时，符号执行阶段需要求解更多路径的路径条件。因此，不同 APP 的符号执行阶段的分析时间可能差别很大。而 APP 插桩阶段和动态测试阶段的执行时间分别取决于 APP 中 *tgtAPI* 的数量和符号执行阶段产生的测试用例数量。

表 5.3　符号执行应用间能力泄露生成方法运行时间

阶段	执行时间		
	Minimum	**Maximum**	**Average**
符号执行	0.19s	13442.52s(3.73h)	240.43s(4.01minutes)
插桩	0.15s	86.41s(1.44minutes)	19.42s
动态测试	6.14s	1611.43s(26.86minutes)	169.91s(2.83minutes)

5.3　基于动态反馈的 Android 应用间能力泄露漏洞利用的自动化生成

5.3.1　能力泄露检测结果

动态反馈方法能力泄露漏洞检测结果如表5.4所示，工具共发现 17 种 7143 个能力泄露缺陷。表5.4第一栏为能力泄露缺陷类型，第二栏 AppUseCount 为出现该能力泄露类型 APP 的数量，第三栏 AllCount 为该能力泄露类型出现在所有 APP 中的能力泄露点的数量。

5.3.2　工具执行效率

表5.5记录了动态反馈工具分析实验数据集 APP 的时间。分别包括三个阶段的执行时间：过程间数据流分析，插桩和动态测试。其中过程间数据流分析最短时间最短为 0.06s, 最长时间为 2.70 minutes，平均每个 APP 的过程间数据流分析时间为 25.40s；插桩最短时间为 0.15s, 最长时间为 1.54 minutes，平均插桩时间为 20.21s；动态测试最短时间为 14.00s，最长时间为 69.37 minutes，平均每个 APP 动态测试的时间为 8.28 minutes。因此，平均每个 APP 的检测时间为 9.04 minutes，其满足实际应用场景的时间需求。对于一些个别 APP 动态测试的时间很大，达到了 69.37 minutes，这个时间仍然是可以接受的。APP 动态测试的时间主要取决于测试用例的数量。因为不同 APP 动态测试获取的信息是不同的 (每个 APP 对外部 *intent* 处理不同)，所以动态测试产生的测试用例数量不同。且不同 APP 拥有的暴露组件数量可能不相同，因此不同 APP 的动态测试的时间差异可能很大。过程间数据流分析阶段和插桩阶段执行时间主要取决于 APP 的大小、APP 对外部 *intent* 处理以及 APP 中 *tgtAPI* 的数量。

表 5.4　动态反馈方法能力泄露漏洞检测结果

Permission	AppUseCount	AllCount
DISABLE_KEYGUARD	9	10
CHANGE_WIFI_MULTICAST_STATE	2	2
RECEIVE_BOOT_COMPLETED	1	2
SET_WALLPAPER_HINTS	5	5
BROADCAST_STICKY	196	278
ACCESS_FINE_LOCATION	167	518
KILL_BACKGROUND_PROCESSES	2	3
ACCESS_COARSE_LOCATION	157	340
CHANGE_WIFI_STATE	4	5
ACCESS_NETWORK_STATE	488	3793
GET_TASKS	306	732
WAKE_LOCK	130	258
ACCESS_WIFI_STATE	340	1102
SET_WALLPAPER	4	4
MODIFY_AUDIO_SETTINGS	3	3
BLUETOOTH	7	10
READ_PHONE_STATE	59	78

表 5.5　动态反馈应用间能力泄露生成方法运行时间

阶段	执行时间		
	Minimum	**Maximum**	**Average**
过程间数据流分析	0.06s	161.70s(2.70minutes)	25.40s
插桩	0.15s	92.32s(1.54minutes)	20.21s
动态测试	14.00s	4162.40s(69.37minutes)	496.52s(8.28minutes)

5.4　工具比较

5.4.1　实验评估指标

由章节2.4.1程序分析的难度与评价可知，误报率、漏报率、时间是程序分析和测试工作的重要评价指标。本文的两个工具本质上都为测试工具，根据章节2.5的能力泄露定义可知，两者的误报率为 0，这也是本文选择使用动态测试的重要原因。因此本文着重考察工具的漏报率指标。本文采取以下公式作为评价工

具漏报率的指标。设待检测 APP 集合为 *AppSet*，大小为 n。对于 APP A_i, 设其能力泄露的权限集合为 $CLSet_i$, 大小为 s_i。设使用工具 t 检测 APP A_i，检测的权限 P_j 能力泄露的能力泄露点集合为 PS_{ij}^{t}。对于 APP A_i 的 P_j 权限的能力泄露，工具 t_1 比工具 t_2 的漏报优势比例为：

$$G_{t_1t_2}(A_i,P_j)=\frac{((PS_{ij}^{t_1}-PS_{ij}^{t_1}\cap PS_{ij}^{t_2})-(PS_{ij}^{t_2}-PS_{ij}^{t_1}\cap PS_{ij}^{t_2}))}{PS_{ij}^{t_1}+PS_{ij}^{t_2}-PS_{ij}^{t_1}\cap PS_{ij}^{t_2}}$$

即工具 t_1 和工具 t_2 检测的 P_j 权限能力泄露结果的差值占两个工具检测 APP A_i 结果总和的比例。工具 t_1 比工具 t_2 平均漏报优势比例为：

$$\bar{G}_{t_1t_2}=\frac{\sum_{i=0}^{n}\sum_{j=0}^{s_i}G_{t_1t_2}(A_i,P_j)}{n*s_i}$$

5.4.2　实验对比对象

本文选取相似动态测试工具 IntentFuzzer[19] 作为本文工具的对比。由于没有联系上作者，本文按照其论文实现其工具。IntentFuzzer 的 *intent* 测试用例的四个属性构建分别如下：

（1） IntentFuzzer 的 *intent* action 属性候选集合由以下三个部分组成，分别是：暴露组件所有 intent-filter 中的 action 值构成的集合；APP 中所有以应用包名为前缀的字符串集合 (APP dex 文件的字符串常量池中存储了 APP 中定义的所有字符串常量)；Android 系统已经定义的标准 action 集合。如果暴露组件含有 intent-filter 的组件，则使用第一部分的 action 值集合构建测试用例的 action 属性；如果暴露组件不含有 intent-filter，则使用第二、第三部分的 action 值集合构建测试用例的 action 属性。

（2） IntentFuzzer 预先定义常用 data 类型的 URI 构成 data 属性候选集合。测试 APP 时，使用 data 属性候选集合中的每个元素去匹配暴露组件的 intent-filter，如果匹配成功，则使用该 URI 作为 *intent* 测试用例的 data 属性。

（3） IntentFuzzer 不考虑 *intent* category 属性，*intent* category 属性候选集合为空。

（4） IntentFuzzer 通过修改 Android 系统源码实现动态测试时获取 extra 属性的 key、type, 并随机产生 extra 属性的 value 值构成 *intent* extra 属性。

IntentFuzzer 组合 *intent* 不同属性的候选集合元素构建 *intent* 测试用例测试 APP，并在测试过程中不断获取 extra 属性加入到 extra 属性候选集合。当组合完所有测试用例且没有新的 extra 属性产生时，测试结束。

IntentFuzzer 能力泄露漏洞检测结果如表5.6所示，IntentFuzzer 共发现 14 种 4216 个能力泄露缺陷。

表 5.6　IntentFuzzer 能力泄露漏洞检测结果

Permission	AppUseCount	AllCount
DISABLE_KEYGUARD	4	6
CHANGE_WIFI_MULTICAST_STATE	2	2
SET_WALLPAPER_HINTS	3	3
BROADCAST_STICKY	134	222
ACCESS_FINE_LOCATION	108	328
KILL_BACKGROUND_PROCESSES	3	4
ACCESS_COARSE_LOCATION	124	303
CHANGE_WIFI_STATE	3	4
GET_TASKS	201	433
ACCESS_NETWORK_STATE	358	2007
WAKE_LOCK	96	170
ACCESS_WIFI_STATE	180	657
BLUETOOTH	4	7
READ_PHONE_STATE	44	70

5.4.3　漏报率比较

使用章节5.4.1的平均漏报优势比例公式分别计算 $\bar{G}_{t_1t_3}$, $\bar{G}_{t_2t_3}$, $\bar{G}_{t_1t_2}$ (符号执行工具 (t_1), 动态反馈工具 (t_2),IntentFuzzer(t_3)), 计算结果如表5.7所示。从表可知，符号执行工具比 IntentFuzzer 漏报率要好 14.97%，动态反馈工具比 IntentFuzzer 工具漏报率要好 19.39%。相比本文工具，IntentFuzzer 采用模糊测试的方法，其构造测试的方式使用更加粗略。虽然 IntentFuzzer 通过修改系统源码的方法获取 intent extra 属性的 key 和 type, 但是其无法获取 value 值，因此其使用随机生成的值作为 value。而本文符号执行工具从代码中获取 extra 属性，而动态反馈通过插桩的方式获取 extra 属性，因此本文工具更加优越。同时本文工具还考虑了 category 属性。本文动态反馈工具比符号执行工具好 5.30%，相比符号执行工具通过静态分析获取 *intent* 路径条件，动态反馈工具通过插桩可以获取更精确的 *intent* 路径条件，因此其漏报率更低。

表 5.7　工具漏报率比较

$\bar{G}_{t_1t_3}$	$\bar{G}_{t_2t_3}$	$\bar{G}_{t_1t_2}$
14.97%	19.39%	-5.30%

5.4.4　时间比较

由于 IntenFuzzer 是我们按照其论文实现，我们并不知道其具体实现细节，因此本文仅从理论上分析其执行时间 (构造测试用例数量的角度)。对于没有 intent-filter 的暴露组件，IntentFuzzer 使用 APP 中所有以应用包名为前缀的字符串集合和 Android 系统已经定义的标准的 action 集合 (177 个) 作为 *intent* 测试用例的 action 值候选集合，此 action 值候选集合数量较大，再和 *intent* 测试用例的其他属性组合将导致 IntentFuzzer 测试用例数量很大。平均情况下，本文两个工具的测试用例数量要比 IntentFuzzer 更少，因此理论上本文工具的时间效率更好。

根据符号执行时间表5.3和动态反馈时间表5.5可得，符号执行工具的平均每个 APP 分析时间为 7.16 minutes，动态反馈工具平均每个 APP 分析时间为 9.04 minutes，因此平均时间效率符号执行工具更好。但是符号执行工具符号执行阶段时间最高为 3.73h，因此最坏时间上动态反馈工具更好。

5.5　能力泄露利用示例

本文提出的两个 Android 应用间能力泄露漏洞的检测工具都能自动化生成能力泄露利用，旨在帮助开发者确认 bug。本节以本文工具产生的两个 APP 的能力泄露利用为例，介绍能力泄露漏洞利用。

应用 A 是一个非常受欢迎的锁屏应用，已经被下载超过一千万次，本文的工具检测到其有 DISABLE_KEYGUARD 权限能力泄露漏洞，存在 DISABLE_KEYGUARD 能力泄露漏洞的暴露组件是 *com.*.LockService*，其在 AndroidManifest.xml 的声明如图5.4所示。

```
<service android:exported="true" android:name="com.*.LockService"/>
```

图 5.4　*com.*.LockService* 声明

从该暴露组件方法出发共有两个能力泄露点，如图5.5所示，分别为方法 *reenableKeyguard()* 和方法 *disableKeyguard()*。两者受 DISABLE_KEYGUARD 权限保护，通过查阅 Android 开发文档可得，*disableKeyguard()* 方法用于解除锁屏，*reenableKeyguard()* 方法用于反解除锁屏。

根据本文工具生成的能力泄露漏洞利用，构造如下测试用例:

#service#com..LockService*

android.os.Parcelable&data&Null

boolean&return-data&false

即 targetComponent 属性为 *com.*.LockService*，extra 属性为 {*android.os.Parcelable&data&Null,boolean&return-data&false* } 的显式 *intent* 对

```
#method#
<com.*.lock.service.d: void b()>
#unitPoint#
$r1.<android.app.KeyguardManager$KeyguardLock: void reenableKeyguard()>()
#lineNumber#35

#method#
<com.*.lock.service.d: void c()>
#unitPoint#
$r1.<android.app.KeyguardManager$KeyguardLock: void disableKeyguard()>()
#lineNumber#43
```

图 5.5　DISABLE_KEYGUARD 能力泄露点

象。其可触发 *disableKeyguard()* 方法，解除当前手机锁屏状态，无需密码。详细攻击演示视频见优酷①。应用 A 的 *com.*.LockService* 组件中并未检查调用者的身份或者无意间暴露了该组件，从而导致了能力泄露漏洞的问题。我们已经通知应用 A 开发者修复此问题。

应用 B 是一个清理软件，其存在 KILL_BACKGROUND_PROCESSES 权限能力泄露漏洞。存在该能力泄露缺陷的暴露组件是 *com.*.QuickBoostActivity*, 该暴露组件在 AndroidManifest.xml 中的声明如图5.6所示。由图可知该组件接收 action

```
<activity android:exported="true" android:name="com.*.QuickBoostActivity">
    <intent-filter>
        <action android:name="com.*.shortcut"/>
        <category android:name="android.intent.category.DEFAULT"/>
    </intent-filter>
</activity>
```

图 5.6　*com.*.QuickBoostActivity* 声明

属性值为 *com.*.shortcut* 的 *intent*, 由于该 action 为自定义 action(不属于 Android 系统定义的标准 Action 值集合的 action 值)，因此我们推测该 Activity 为内部使用 (应用内的其他组件使用或同一个开发者的其他应用使用)。开发者可能没有意识到该组件暴露可能会引起外部攻击，因此导致了能力泄露漏洞的出现。该能力泄露漏洞被利用后可清除后台进程，攻击演示视频详见优酷②。

5.6　本章小结

本章使用本文的两个工具基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成和基于动态反馈的 Android 应用间能力泄露漏洞利用的自动化生成分别对豌豆荚各类别 611 个 APP 进行分析。分别检测出 16 种共 6566 个能

①https://v.youku.com/v_show/id_XNDExOTg1ODA3Mg

②https://v.youku.com/v_show/id_XNDExOTg2MDAxMg

力泄露类型缺陷和 117 种 7143 个能力泄露缺陷，其中包括一些非常严重的能力泄露漏洞。例如：DISABLE_KEYGUARD 权限能力泄露漏洞，本文工具发现有 9 个 APP 存在这种缺陷。由能力泄露示例章节可知，这个能力泄露漏洞被利用可无密码越过键盘锁，因此能力泄露漏洞可能会对用户和手机安全造成巨大伤害。同时，对比同类型工具 IntentFuzzer，本文符号执行和动态反馈工具相比 IntentFuzzer 漏报率分别好 14.97% 和 19.39%，因此本文工具能检测出更多能力泄露漏洞。本文两个工具的 APP 平均分析时间分别为 7.16min 和 9.04min，其满足实际应用场景的时间使用需求。

第 6 章　总结与展望

6.1　本文工作总结

Android 应用间能力泄露漏洞是一种危害性非常大的漏洞，其他应用或者攻击者可以利用应用的 Android 应用间能力泄露漏洞非法地使用该应用的特殊能力，因此对 Android 安全造成了巨大的危害。

目前 Android 组件间通信漏洞的主要工作分为静态分析、动态分析和动静态结合分析三个方面。静态分析工作的主要缺陷是不能确定漏洞的真实存在，而自动化漏洞确认对于开发者极其重要，可以避免让开发者花费大量时间确认 bug；而现有的动静态结合分析工作的测试用例生成技术存在不足，导致代码覆盖率低，因此漏报率较高。针对这两个问题，本文先后提出了两个 Android 应用间能力泄露漏洞利用的自动化生成工作，并设计了一整套自动化检测工具，可以自动化检测 Android 应用间能力泄露漏洞，生成漏洞利用，帮助开发者确认 bug，缩短软件开发周期。本文的主要工作总结如下：

（1） 基于符号执行的 Android 应用间能力泄露漏洞利用的自动化生成。将符号执行技术引入到 Android 应用间能力泄露漏洞的检测工作中，使用其产生高精度的测试用例，提高代码覆盖率；并根据 Android 应用间能力泄露漏洞检测工作问题的特性从方法调用图层次和控制流图层次对符号执行进行优化，缓解路径爆炸问题，使符号执行适用于实际应用的能力泄露漏洞检测工作；精确计算出每条路径的 *intent* 路径条件。根据豌豆荚各类别 611 个 APP 的实验结果，本文符号执行技术的平均路径条件计算准确率为 85.3%；相比同类型工作 IntentFuzzer，本文符号执行工具漏报率要好 14.97%；符号执行工具的平均 APP 分析时间为 7.16 minutes，满足实际应用场景的时间需求。

（2） 基于动态反馈的 Android 应用间能力泄露漏洞利用的自动化生成。通过对 Android 应用插桩，获取程序的动态运行信息产生测试用例。相比静态分析提取获取 *intent* 路径条件，其可扩展性更强，能解决静态分析中复杂变量值难计算和动态变量值无法获取的问题；同时使用上下文敏感、流敏感的过程间数据流分析精确找到每一个包含 *intent* 路径条件信息的变量和插桩位置点，保证动态获取路径条件信息的完备性；针对获取的 *intent* 路径条件信息进行变异提高分支覆盖率。相比同类型工作 IntentFuzzer，漏报率要好 19.39%，相比本文第一个工作漏报率要好 5.30%；动态反馈工具的平均 APP 分析时间为 9.04 minutes，满足实

际应用场景的时间需求。

6.2　未来展望

根据我们研究工作中存在的不足以及对 Android 应用间能力泄露漏洞的理解，我们未来将从以下三个方面继续开展 Android 应用间能力泄露的研究工作：

（1） 本文研究的 Android 应用间能力泄露漏洞是基于 *intent* 对象，因为 Content Provider 并不支持使用 *intent* 查询和更改，所以本文工具并不支持 Content Provider 的能力泄露漏洞检测。本文支持 Android 组件只包括 Activity、Service 和 Brocast Receiver，我们将在未来研究工作加入对 Content Provider 能力泄露漏洞检测的支持。

（2） 从本文检测出的 Android 应用间能力泄露漏洞的结果来看，豌豆荚各类别 611 APP 中存在一些非常严重的 Android 应用间能力泄露漏洞，也存在一些危害性较小的 Android 应用间能力泄露漏洞。Android 应用间能力泄露漏洞的危害性主要取决于 Android 应用间能力泄露类型以及 Android APP 如何使用这些能力。对于开发者而言，往往希望优先修复危害性更大的 Android 应用间能力泄露类型，因此我们将 Android 应用间能力泄露漏洞危害性评级作为我们未来工作之一。

（3） 本文中关注的能力是 Android 系统定义的权限，对于 Android 应用而言，能力可以不仅仅可以是权限，也可以是某项功能，例如：智能家居伙伴 APP 中打开门锁的功能。功能泄露也会导致重大的安全事故，因此功能泄露研究具有重大的研究意义，其主要挑战是 Android 应用的功能语义分析，我们将其作为我们未来的研究工作之一。

同时，根据章节2.2可知，当前 Android 组件间权限机制还存在缺陷与不足：通过在 AndroidManifest.xml 中声明一个 Android 应用组件的组件间强制权限只支持检查调用者的一种权限；通过 *Context.checkCallingPermission()* API 检测调用者权限只支持在使用了 AIDL(Android Interface Definition Language) 的 IPC(inter process communication) 中使用。因此希望 Google 继续完善 Android 系统组件间权限检测机制，帮助开发者规避 Android 应用间能力泄露漏洞。

参 考 文 献

[1] IDC. Smartphone market share[EB/OL]. 2019. https://www.idc.com/promo/smartphone-market-share/os.

[2] 360. 2018 年 Android 恶意软件专题报告[EB/OL]. 2019. http://zt.360.cn/1101061855.php?dtid=1101061451&did=610100815.

[3] DETAILS C. 2017 年度 CVE Details 报告[EB/OL]. 2018. https://www.cvedetails.com.

[4] LI L, BARTEL A, KLEIN J, et al. Automatically exploiting potential component leaks in android applications[C/OL]//13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014. 2014: 388-397. https://doi.org/10.1109/TrustCom.2014.50.

[5] HE Y, LI Q. Detecting and defending against inter-app permission leaks in android apps [C/OL]//35th IEEE International Performance Computing and Communications Conference, IPCCC 2016, Las Vegas, NV, USA, December 9-11, 2016. 2016: 1-7. https://doi.org/10.1109/PCCC.2016.7820624.

[6] GRACE M C, ZHOU Y, WANG Z, et al. Systematic detection of capability leaks in stock android smartphones[C/OL]//19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012. 2012. https://www.ndss-symposium.org/ndss2012/systematic-detection-capability-leaks-stock-android-smartphones.

[7] YAN J, DENG X, WANG P, et al. Characterizing and identifying misexposed activities in android applications[C/OL]//Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. 2018: 691-701. https://doi.org/10.1145/3238147.3238164.

[8] ENCK W, OCTEAU D, MCDANIEL P D, et al. A study of android application security [C/OL]//20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings. 2011. http://static.usenix.org/events/sec11/tech/full_papers/Enck.pdf.

[9] CHIN E, FELT A P, GREENWOOD K, et al. Analyzing inter-application communication in android[C/OL]//Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, MD, USA, June 28 - July 01, 2011. 2011: 239-252. http://doi.acm.org/10.1145/1999995.2000018.

[10] LU L, LI Z, WU Z, et al. CHEX: statically vetting android apps for component hijacking vulnerabilities[C/OL]//the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012. 2012: 229-240. http://doi.acm.org/10.1145/2382196.2382223.

[11] XIE J, FU X, DU X, et al. Autopatchdroid: A framework for patching inter-app vulnerabilities in android application[C/OL]//IEEE International Conference on Communications, ICC 2017, Paris, France, May 21-25, 2017. 2017: 1-6. https://doi.org/10.1109/ICC.2017.7996682.

[12] WU S, ZHANG Y, JIN B, et al. Practical static analysis of detecting intent-based permission leakage in android application[C]//2017 IEEE 17th International Conference on Communication Technology (ICCT). IEEE, 2017: 1953-1957.

[13] LIU F, CAI H, WANG G, et al. Mr-droid: A scalable and prioritized analysis of inter-app communication risks[C/OL]//2017 IEEE Security and Privacy Workshops, SP Workshops 2017, San Jose, CA, USA, May 25, 2017. 2017: 189-198. https://doi.org/10.1109/SPW.2017.12.

[14] ZHONG X, ZENG F, CHENG Z, et al. Privilege escalation detecting in android applications [C/OL]//3rd International Conference on Big Data Computing and Communications, BIGCOM 2017, Chengdu, China, August 10-11, 2017. 2017: 39-44. https://doi.org/10.1109/BIGCOM.2017.21.

[15] BAGHERI H, SADEGHI A, GARCIA J, et al. COVERT: compositional analysis of android inter-app permission leakage[J/OL]. IEEE Trans. Software Eng., 2015, 41(9):866-886. https://doi.org/10.1109/TSE.2015.2419611.

[16] FELT A P, WANG H J, MOSHCHUK A, et al. Permission re-delegation: Attacks and defenses [C/OL]//20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings. 2011. http://static.usenix.org/events/sec11/tech/full_papers/Felt.pdf.

[17] BUGIEL S, DAVI L, DMITRIENKO A, et al. Xmandroid: A new android evolution to mitigate privilege escalation attacks[J]. Technische Universität Darmstadt, Technical Report TR-2011-04, 2011.

[18] DAI T, LI X, HASSANSHAHI B, et al. Roppdroid: Robust permission re-delegation prevention in android inter-component communication[J/OL]. Computers & Security, 2017, 68: 98-111. https://doi.org/10.1016/j.cose.2017.04.002.

[19] YANG K, ZHUGE J, WANG Y, et al. Intentfuzzer: detecting capability leaks of android applications[C/OL]//9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014. 2014: 531-536. http://doi.acm.org/10.1145/2590296.2590316.

[20] DEMISSIE B F, GHIO D, CECCATO M, et al. Identifying android inter app communication vulnerabilities using static and dynamic analysis[C/OL]//Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, Austin, Texas, USA, May 14-22, 2016. 2016: 255-266. http://doi.acm.org/10.1145/2897073.2897082.

[21] RICE H G. Classes of recursively enumerable sets and their decision problems[J]. Transactions of the American Mathematical Society, 1953, 74(2):358-366.

[22] ZHANG J, WANG X. A constraint solver and its application to path feasibility analysis[J]. International Journal of Software Engineering and Knowledge Engineering, 2001, 11(02):139-156.

[23] AHO A V, LAM M S, SETHI R, et al. Compilers: Principles, techniques, and tools, pearson education[M]. Inc, 2006.

[24] REPS T W, HORWITZ S, SAGIV S. Precise interprocedural dataflow analysis via graph reachability[C/OL]//Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995. 1995: 49-61. https://doi.org/10.1145/199448.199462.

[25] LI L, CIFUENTES C, KEYNES N. Boosting the performance of flow-sensitive points-to analysis using value flow[C/OL]//SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011. 2011: 343-353. https://doi.org/10.1145/2025113.2025160.

[26] SUI Y, XUE J. On-demand strong update analysis via value-flow refinement[C/OL]// Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. 2016: 460-473. https://doi.org/10.1145/2950290.2950296.

[27] MARINESCU P D, CADAR C. make test-zesti: A symbolic execution solution for improving regression testing[C/OL]//34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. 2012: 716-726. https://doi.org/10.1109/ICSE.2012.6227146.

[28] PERSON S, YANG G, RUNGTA N, et al. Directed incremental symbolic execution[C/OL]// Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. 2011: 504-515. https://doi.org/10.1145/1993498.1993558.

[29] CADAR C, DUNBAR D, ENGLER D R. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs[C/OL]//8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. 2008: 209-224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

[30] XIE T, TILLMANN N, DE HALLEUX J, et al. Fitness-guided path exploration in dynamic symbolic execution[C/OL]//Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. 2009: 359-368. https://doi.org/10.1109/DSN.2009.5270315.

[31] LI Y, SU Z, WANG L, et al. Steering symbolic execution to less traveled paths[C/OL]// Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013. 2013: 19-32. https://doi.org/10.1145/2509136.2509553.

[32] SEO H, KIM S. How we get there: a context-guided search strategy in concolic testing[C/OL]// Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. 2014: 413-424. https://doi.org/10.1145/2635868.2635872.

[33] YU H, CHEN Z, WANG J, et al. Symbolic verification of regular properties[C/OL]// Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. 2018: 871-881. https://doi.org/10.1145/3180155.3180227.

[34] QI D, NGUYEN H D T, ROYCHOUDHURY A. Path exploration based on symbolic output [J/OL]. ACM Trans. Softw. Eng. Methodol., 2013, 22(4):32:1-32:41. https://doi.org/10.1145/2522920.2522925.

[35] GUO S, KUSANO M, WANG C, et al. Assertion guided symbolic execution of multithreaded programs[C/OL]//Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015. 2015: 854-865. https://doi.org/10.1145/2786805.2786841.

[36] WANG H, LIU T, GUAN X, et al. Dependence guided symbolic execution[J/OL]. IEEE Trans. Software Eng., 2017, 43(3):252-271. https://doi.org/10.1109/TSE.2016.2584063.

[37] KUZNETSOV V, KINDER J, BUCUR S, et al. Efficient state merging in symbolic execution [C/OL]//ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. 2012: 193-204. https://doi.org/10.1145/2254064.2254088.

[38] YI Q, YANG Z, GUO S, et al. Eliminating path redundancy via postconditioned symbolic execution[J/OL]. IEEE Trans. Software Eng., 2018, 44(1):25-43. https://doi.org/10.1109/TSE.2017.2659751.

[39] SLABY J, STREJCEK J, TRTÍK M. Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution[C/OL]//Formal Methods for Industrial Critical Systems - 17th International Workshop, FMICS 2012, Paris, France, August 27-28, 2012. Proceedings. 2012: 207-221. https://doi.org/10.1007/978-3-642-32469-7_14.

[40] VISSER W, GELDENHUYS J, DWYER M B. Green: reducing, reusing and recycling constraints in program analysis[C/OL]//20th ACM SIGSOFT Symposium on the Foundations of

Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012. 2012: 58. https://doi.org/10.1145/2393596.2393665.

[41] MICROSOFT. z3[EB/OL]. 2019. https://github.com/Z3Prover/z3/wiki.

[42] SABLE. Soot[EB/OL]. 2019. https://sable.github.io/soot/.

[43] WIKI. Java bytecode[EB/OL]. 2019. https://en.wikipedia.org/wiki/Java_bytecode.

[44] VALLEE-RAI R, HENDREN L J. Jimple: Simplifying java bytecode for analyses and transformations[J]. 1998.

[45] BARTEL A, KLEIN J, TRAON Y L, et al. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot[C/OL]//Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012. 2012: 27-38. https://doi.org/10.1145/2259051.2259056.

[46] GOOGLE. Dalvik bytecode[EB/OL]. 2019. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.

[47] ARZT S, RASTHOFER S, FRITZ C, et al. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps[C/OL]//ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. 2014: 259-269. http://doi.acm.org/10.1145/2594291.2594299.

[48] LI L, BARTEL A, BISSYANDÉ T F, et al. Iccta: Detecting inter-component privacy leaks in android apps[C/OL]//37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1. 2015: 280-291. https://doi.org/10.1109/ICSE.2015.48.

[49] DESNOS A, et al. Androguard: Reverse engineering, malware and goodware analysis of android applications[J]. URL code. google. com/p/androguard, 2013:153.

[50] ANDROGUARD. Android permission api mappings[EB/OL]. 2019. https://github.com/androguard/androguard/blob/master/androguard/core/api_specific_resources/api_permission_mappings/permissions_25.json.

[51] WIKI. Reaching definition wiki[EB/OL]. 2019. https://en.wikipedia.org/wiki/Reaching_definition.

[52] GUARDSQUARE. dexguard[EB/OL]. 2019. https://www.guardsquare.com/en/products/dexguard.

致　谢

突然时间来到毕业的这一刻，仿佛昨日才来到科大。蓦然回首，发现自己在科大已经整整生活了三年。这三年，我收获很大。在这里，我要表达研究生生活中关心爱护我的人的感谢，感谢你们一路相陪，帮助我茁长成长。

首先，我要感谢的是曾凡平老师。做为我的研究生导师，曾老师给予了我们很大的学习空间，让我们自由地选择自己感兴趣的研究方向，当我们科研受挫时，他时常鼓励我们不放弃。曾老师在科研的道路上给予了我诸多宝贵的意见，使我在科研路上少走了许多弯路。曾老师在修改我们论文时会逐字逐句的反复检阅，其认真严谨的治学态度让我深受感染。曾老师还努力培养我们各方面的能力。感谢老师辛勤的付出！

其次，我要感谢科大的老师们。从他们的课程和工作汇报中，我了解学习到了计算机学科各个领域顶尖的研究工作，极大的开拓了我的眼界，他们的科研能力让我着实钦佩。我还要感谢实验室的师兄师姐们以及同届同学吕成成。每当我遇到科研难题，他们总能在我的前方点亮一盏明灯，给我照亮前方的路。仲星球师姐、谢念念师姐他们为为人处事的方式，程志超师兄对技术追求的态度让我收获颇多。同时也要感谢我的师弟师妹们：陈钊，陈国柱，束文娟，鲁厅厅，他们也帮助了我很多。尤其是陈钊师弟，在我研究课题初期帮助我做了很多调研工作。

此外，我还要感谢科大，是您给我提供这么好的学习环境，让我茁壮成长！我会牢记校训：红专并进，理实交融。在以后的工作中，我会加倍努力，追求卓越，不负曾为科大人！

最后，我要感谢本文的评审老师，感谢您们从繁忙的科研工作中抽出时间评阅我的论文，您们辛苦了!

在读期间发表的学术论文与取得的研究成果

已发表论文

1. Zhichao Cheng, Fanping Zeng, Xingqiu Zhong, **Mingsong Zhou**, Chengcheng Lv, Shuli Guo. Resolving Reflection Methods in Android Applications [C]. 2017 IEEE International Conference on Intelligence and Security Informatics. IEEEISI2017 (July 22-24, 2017, Beijing, China). 143-145
2. Niannian Xie, Fanping Zeng, Xiaoxia Qin, Yu Zhang, **Mingsong Zhou** and Chengcheng Lv. RepassDroid: Automatic Detection of Android Malware Based on Essential Permissions and Semantic Features of Sensitive APIs. The 12th International Symposium on Theoretical Aspects of Software Engineering. TASE 2018 (August 29-31, 2018, Guangzhou, Guangdong, China), 52-59.
3. 谢念念, 曾凡平, **周明松**, 秦晓霞, 吕成成, 陈钊. 多维敏感特征的 Android 恶意应用检测 [J], 计算机科学, 2019, 46(2): 95-101.

已录用论文

1. **Mingsong Zhou**, Fanping Zeng, Yu Zhang etc. Automatic Generation of Capability Leaks’ Exploits for Android Applications[C]. ICST workshop 2019

参与项目

1. 国家自然科学基金，2017“移动和云平台软件的系统资源使用行为分析与改进”
2. 中科院软件所合作课题，2016“基于多重对应分析的 Android 应用安全评估”