论文题目：基于污点分析的安卓安全信息流静态检测方法研究

专　　业：计算机系统结构

硕 士 生：杜茜　　（签名）_____

指导教师：刘晓建　　（签名）_____

摘　要

由于安卓应用程序需要使用大量的用户隐私数据，并且国内安卓应用市场缺乏安全的第三方认证，这会导致包括数据泄漏和侵犯隐私在内的各种问题。维护用户隐私和保护敏感数据不被泄露至关重要。那么在使用安卓应用程序前进行安全检测变得至关重要。由于 Android 应用程序所带来的不同挑战，如组件间通信(ICC)、反射和隐式流等，静态数据流分析方法经常会产生误报。针对 Android 应用程序的组件间通信问题我们提出了一种基于 FlowDroid 的组件间静态检测技术，用于对 Android 应用程序组件内和组件间进行静态数据流分析，以检测敏感数据的泄漏，主要研究内容如下：

(1) 研究当前 Android 应用安全存在的问题,详细分析 Android 系统框架与当前 Android 安全机制。研究静态检测工具 FlowDroid 检测架构与具体分析流程，从底层源码分析 FlowDroid 静态检测工具。并且只能检测组件内的数据流泄露情况，而无法检测组件间的情况；

(2) 基于 IFDS 算法进行优化，通过计算数据事实集合 D 进行更小的等价集合进行替代，减少无关路径的计算。

(3) 基于字符串分析的组件间通信值。所有组件间通信都至少将一个 Intent 作为它们的参数。而 Intent 对象携带的信息指定了会被它启动的组件，以及被启动组件需要执行的操作。所以对 Android 应用程序组件间的信息流泄露检测，我们必须知道组件间的通信关系；

(4) 基于 ICCG 的污点分析。首先提取 Android 软件组件间的通信值，经过一系列的处理，生成 ICCG，基于 ICCG，分析组件间的污点传播路径；

关键词：安卓安全；污点分析；静态分析；组件间通信；字符串分析；

研究类型：理论研究

Subject : Research on static detection method of Android security information flow based on taint analysis

Specialty : Computer Architecture

Name : Du Xi (Signature) _____

Instructor : Liu Xiaojian (Signature) _____

ABSTRACT

Because Android applications need to use a large amount of user privacy data, and the domestic Android application market lacks secure third-party authentication, this will cause various problems including data leakage and privacy violations. It is essential to maintain user privacy and protect sensitive data from being leaked. Then it becomes very important to conduct security checks before using Android apps. Due to the different challenges brought by Android applications, such as inter-component communication (ICC), reflection, and implicit flow, static data flow analysis methods often produce false positives. Aiming at the communication problem between components of Android applications, we propose a static detection technology between components based on FlowDroid, which is used to analyze the static data flow within and between components of Android applications to detect the leakage of sensitive data. The main research content as follows:

(1) Research the current security problems of Android applications, and analyze the Android system framework and current Android security mechanisms in detail. Research the static detection tool FlowDroid detection architecture and specific analysis process, and analyze the FlowDroid static detection tool from the underlying source code. And it can only detect the data flow leakage within the components, but cannot detect the situation between the components;

(2) Optimize based on the IFDS algorithm, and replace it with a smaller equivalent set by calculating the data fact set D, reducing the calculation of irrelevant paths.

(3) Communication value between components based on string analysis. All communication between components takes at least one Intent as their parameter. The information carried by the Intent object specifies the components that will be started by it, and the operations that the started components need to perform. Therefore, for the detection of information flow leakage between Android application components, we must know the communication relationship between the components;

(4) Stain analysis based on ICCG. First, extract the communication value between Android software components, after a series of processing, generate ICCG, based on ICCG, analyze the taint propagation path between components.

Key words: Stain analysis; static analysis; communication between components; machine learning; string analysis;

Thesis : Theoretical Research

目录

1 绪论

1.1 选题背景及意义

在过去的十年里，智能手机已经成为接入互联网的主要方式[1]。智能手机的先进技术和功能允许用户访问和受益于许多在线服务，如浏览网页、交换电子邮件等。因此，用户已经开始在智能手机设备上以前所未有的规模存储他们的私人信息，如信用卡信息、社交号码、密码、位置数据和电子邮件等。如果这些信息落入坏人之手，会给用户带来许多问题。例如，2017 年末，卡巴斯基实验室的研究专家发现了一种新的恶意软件，名为 DfakeToken [2]。一旦用户在受感染的设备上输入信用卡号码，该恶意软件就会窃取用户的信用卡号码和银行发送的验证码。但是如果使用的是公司电话，也会导致一些商业信息泄露[3]。我们应该保护敏感数据，防止任何想要篡改、识别或检索这些数据的攻击者[4]。最近几项研究[5，6]表明，一些敏感数据可能会泄漏，要么是开发商为了广告收入故意泄漏，要么是通过以纯文本形式在公共网络上传输数据而意外泄漏[8]。现在使用的不同智能手机操作系统有很多，如 iOS、Android、Windows Mobile、Blackberry 等。每个智能手机操作系统都存在安全漏洞。根据赛门铁克[10]的一份报告，在 2013 年、2014 年和 2015 年，iOS 平台是移动操作系统漏洞数量最多的平台，其次是 Android、Blackberry 和 Windows 移动操作系统。比如，Android 不能向任何设备推送安全更新。它只能向谷歌制造的设备推送更新，比如 Nexus 手机。另一方面，苹果和微软可以向任何运行其操作系统的设备推送新的更新。NowSecure 在 2016 年的一份报告[7]显示，大约 82%的安卓设备容易受到操作系统漏洞的攻击，因为新的操作系统更新不支持它们。此外，该报告指出，在世界上的 10 部手机中，有 8 部运行安卓操作系统，这使得它成为网络犯罪分子开发恶意应用程序的一个有吸引力的目标。恶意软件，是恶意软件的简称。如今，由于用户设备中的宝贵信息，智能手机恶意软件急剧增加。此外，根据 cAfeeLabs 威胁报告[8]，移动恶意软件从 2015 年增加到 138%,在不同移动平台上检测到超过 200 万种恶意软件。此外，赛门铁克实验室的另一份报告[9]显示，2017 年大多数安卓恶意软件分布在以下类别:生活方式、音乐和音频、书籍和参考、娱乐、工具、房屋和家庭、教育、艺术和设计、摄影和休闲游戏。在这些类别中，27%的安卓恶意软件分布在生活方式类别中。因此，大多数智能手机都面临着许多安全问题，如私人信息泄露、恶意软件和位置泄露。由于恶意软件应用程序的数量不断增长，日常用户的隐私可能会受到威胁。恶意软件活动的一个例子是使用智能手机资源(例如，麦克风、摄像头或设备中的任何其他传感器)来调查用户，这将使他们的手机成为秘密监控设备。此外，用户的

设备不仅是攻击者的目标，也是广告商和政府机构的目标。智能手机继续变得越来越普遍。在不同的平台中，Android 在这一领域处于领先地位，拥有 85%的智能手机用户，将其他平台抛在了后面，比如 iOS，它以 15%的份额远远落后于其他平台。随着智能手机越来越受欢迎，功能越来越多，越来越多的人使用他们的设备进行隐私敏感的活动。这使用户面临移动环境之外可能不存在的隐私风险。

由于恶意软件的猖獗，移动设备中的问题越来越多。随着当前技术的增长，恶意软件成为了移动设备中更大的一个问题。近年来由于用户从有安全隐患的应用市场下载应用程序，导致恶意软件对用户的侵害迅速增加。一旦攻击成功。可能会引发大规模的业务中断。目前的安全解决方案在应对这些网络威方面表现不佳。此外，现有的努力从方便和有效的角度出发。这些解决方案可以归纳为以下三大类。第一，检测恶意软件时依赖使用仿真器的分析技术。然而，有些高级恶意软件变体可以避开这些解决方案，它们能够识别分析环境的存在，并且不表现出任何恶意行为，这类恶意软件就有可能是错误分类。例如 Droiddetector[10]。第二类是需要修改安卓操作系统和研究内核以满足其性能的解决方案。此类解决方案旨在拦截恶意操作，并为安卓平台提供更多增强功能。然而，这一类的解决方案受到厂商发布的频繁更新的影响，这可能会导致重大的可用性问题。每一个版本的发布，开发者都需要确保其解决方案的性能。因此，用户的设备在更新后可能会出现漏洞，除非他们发布同等的版本的解决方案。这种类型的一个显著例子是 TaintDroid[11]。第三类解决办法的问题是，一些现有的方法会产生很高的耗电量，因为这些方法是由一个个不同的人组成的。的 CPU 和内存。这些解决方案对于检测恶意应用程序来说，是无能为力的。因为它们需要大量的硬件资源来分析被检查的应用。这类解决方案的开发者依靠使用非设备分析技术来实现。

基于以上背景，本文提出了一种基于 FlowDroid 的恶意软件静态检测方法，首先通过检测单个组件内的隐私数据泄露，然后结合字符串分析提提取应用程序组件间的通信值，然后生成 ICCG，基于 ICCG 进行组件间的污点分析。

1.2 国内外研究现状及发展方向

1.2.1 静态分析

关于对 Android 安全问题应用静态分析的工作已经有很长一段时间了。最近的工作[32，34]探索了跟踪 Android 应用程序间通信以进行安全审查的各种方法。吕特等人[3]使用一种叫做 CHEX 的静态分析方案来检测安卓系统中的组件劫持问题，该问题被简化为寻找信息流，每个都是从入口点可到达的代码段。然后，它使用 Wala[39]计算数据流摘要。不违反版权系统调用序列和代码结果传递信息流的命令行参数。

这种方法检查被检查的应用程序的语法和结构，属性，而无需运行实际代码。因此，如果被检查的应用程序是恶意软件，这种方法可以在不感染实际设备的情况下检测到它。许多使用静态分析法可以获得一些优势，比如说指向根本的导致安全问题，而不是指向一个具体的问题。同时，它还可以帮助调查人员可以在新的攻击发生之前检测到它们的可能性，它可以检查分析应用程序的所有可能的执行路径。然而，不同的代码变换技术可以欺骗静态分析方法，如代码等。混淆、本机代码、动态加载的代码和反射代码[13]，这些代码往往是出现在 Android 恶意软件中。一个 APK 包含几个重要文件，如 classes.dex 文件，AndroidManifestfile 和 resources。静态分析可以从 APK 包中提取可用于区分恶意活动的信息。因此，大多数安卓静态分析框架的关键组件是(1)AndroidManifest 文件，其中包含了重要的信息(如：应用组件、应用程序、应用系统和权限等)；(2)classes.dex 文件，其中包含了编译后的被检查的应用程序的 java 字节码[47]。classes.dex 文件对于分析人员来说是难以阅读的。因此，它应该被转换为可读的格式[13]。例如，SCanDroid[14]、PSout[15]，以及 PermCheckTool[16]正在将 classes.dex 转换为 Java 源代码，用于分析。另外，其他工具也可以分析 Java 字节码，如 AppIntent[17]、AppAudit[18]和 AndroidLeaks[19]。此外，Android 应用程序的 classes.dex 可以被转换为到 Jimple 代码中进行分析，如 FlowDroid[20]、DroidGuard[21]和 DroidForce[22]。

静态分析框架利用基于文本的信息检索方法或应用程序的代码结构方法来分析所有与安全相关的问题。根据[23]，几乎 50%的 Android 静态分析方法都是采用基于文本的信息检索技术。该技术提取了所有可能的安全特征，如权限、API、应用组件和意图过滤器的特征，以及其他补充技术，如机器学习或自然的语言处理，来评估被检查应用程序的安全性[23]。例如，Derbin[24]从 Android 应用中提取了基于文本的特征（如请求的硬件组件、意图、权限、应用组件、API 和网络地址)，并使用支持向量机(SVM)对分析后的应用程序。类似地，Andrubis[25]反编译一个 Android 应用，并收集了大量的关于被检查应用程序的信息（如权限、服务、广播、接收器和活动等）。以及应用程序的字节码，以检测恶意的应用程序[25]。此外，许多安全分析工具也使用分析应用程序的代码结构，这种技术使用一种众所周知的数据结构，如控制流图(CFG)、调用图(CG)、程序间控制流图(ICFG)，以提取被检查应用程序的底层代码。这可以通过以下方式实现通过将 classes.dex 文件转换为可接受的格式来提取特征（例如 method、API、class 和结构序列）。使用这种技术有很多优点。例如，Android 应用程序可以根据控制流和数据流进行检查，这就意味着为分析者提供了更多关于应用程序使用的数据泄漏和其他危险的信息，例如滥用电话服务。图 1.1 显示了 Android 静态分析的整体流程。

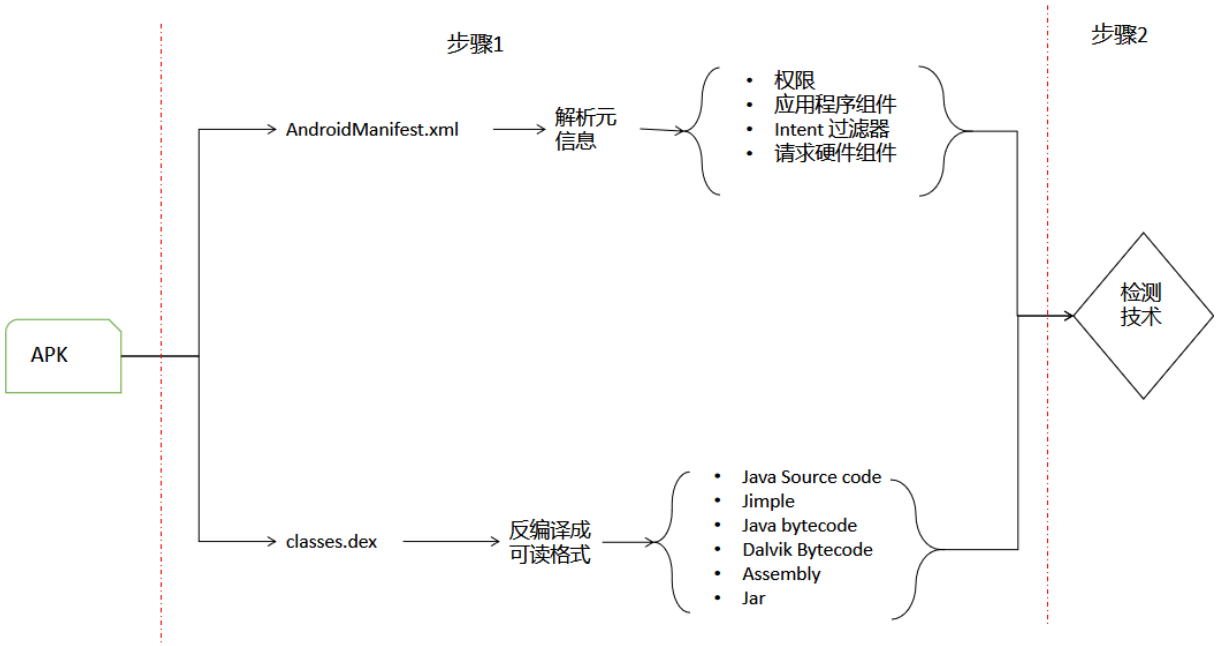

图 1.1　Android 静态分析整体流程

1.2.2 动态分析

与静态分析相反的是，动态分析收集的是关于应用程序在其实际运行时的信息。被检查的应用程序是在一个完整的 Android 系统环境中执行的。在应用程序运行的时候。通过使用检测技术能够监测其行动和活动，并根据训练阶段学到的知识检查异常行为。[44]. 这种类型的分析需要一组输入，如系统事件、用户界面手势和入口点。例如，PuppetDroid[26]记录了用户与一个应用程序接口的交互。然后在动态分析过程中，他们重新执行记录在案的恶意的行为，并检测具有类似行为的应用程序。但是，如果在训练阶段监测到的输入与在训练阶段监测到的输入不同，那么在检测阶段，动态分析可能会漏掉检测到的漏洞或被监控应用程序的恶意活动。主要有两种类型的动态分析，应用程序级分析和内核级分析。分析 Android 操作系统的不同等级增加了特征多样性的数量，这可以提高检测的准确性。

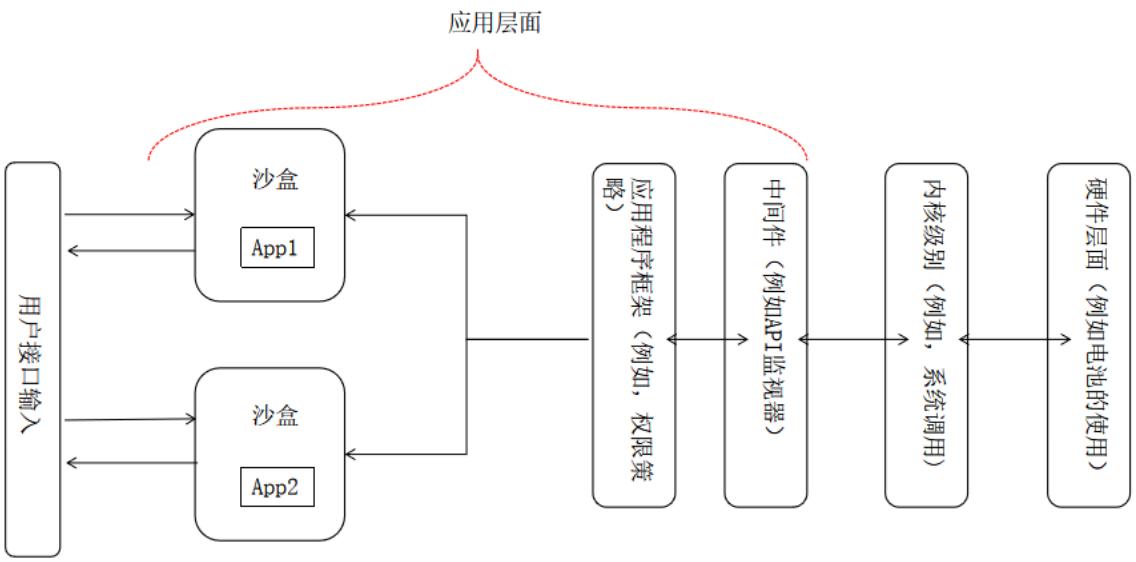

图 1.2 动态分析流程

TaintDroid[11]是一个动态(运行时)污点跟踪和分析系统，用于发现用户个人信息的潜在滥用。DroidScope[27]是一个 Android 应用程序动态分析工具，可以 重建系统级别和虚拟机级别的信息。Droidscope 使用动态数据分析收集详细的本机和硬件指令跟踪、配置文件级别的活动以及跟踪信息泄漏遍历本机和本机组件。Apps 游乐场[12]是一种动态分析方法，通过使用探索程序自动观察和检测 Android 应用程序中的恶意软件。移动沙盒[13]是一种混合方法，结合静态和动态分析来记录应用程序的恶意模式。首先进行静态分析，对源代码进行分析，找出恶意权限。稍后，通过执行应用程序来执行动态分析，以记录所有动作，甚至是从本机 API 调用生成的动作。

1.2.3 动静混合分析

混合分析已经被多种工具使用。例如，DroidDetector [28]使用静态分析从 AndroidManifest 收集请求的权限。然后，它使用 DroidBox [29]来执行动态分析，以在被检查的应用程序运行时提取其他特征，例如系统调用。最后，对提取的特征应用深度信任网络来检测恶意应用。使用 21760 个应用程序(包括 20000 个良性应用程序和 1760 个恶意应用程序)对该方法进行了评估，它们达到了 96.76 %的准确率。DroidDetector 的一个主要限制是通过 DroidBox 动态分析应用程序，高级恶意软件可以避开 DroidTector ，高级恶意软件具有检测分析环境的能力，并且不会表现出恶意行为。ANDRUBIS [30]是一种结合动态和静态分析的自动方法，通过在 Dalvik Virtual Machine 中执行 java 代码和在系统级别执行本机代码来记录事件。

1.3 研究内容

本课题通过研究现有 Android 恶意程序的行为特征，一种基于 FlowDroid 的组件间静态检测方法，最后再结合机器学习进行恶意软件判定。首先通过对 FlowDroid 深入研究，析其框架、检测流程、污点追踪算法等关键技术，以部分组件的检测模式，减少不必要的检测时间和内存损耗，定义了一个组件间通信图，并解决了组件间的污点分析，最后采用机器学习算法训练效果良好的分类器。具体内容包括以下四部分

（1）基于 FlowDroid 进行 Android 应用程序组件内分析，并设计了一个部分组件搜索模式；

（2）基于字符串分析解析组件间的 Intent 通信关系，构造了一个 ICCG 来表示组件之间的关系，并在 ICCG 的基础上进行数据流分析；

（3）基于 IFDS 算法进行优化，对于需要计算的 data fact 集合 D 进行替代，找到最小等价集合

1.4 论文结构安排

本课题的论文章节安排如下：

第一章为绪论，通过介绍目前 Android 系统的普遍性和移动安全面临的严峻性，从而引入本课题所具有的现实意义。并介绍了目前国内外的相关研究现状，对传统工作进行总结分析，从而引入本课题的主要研究内容。

第二章进行了相关知识介绍。对 Android 系统的体系结构，安全机制进行了详细阐述，这是进行研究的理论基础，并对本课题在研究过程中用到的相关工具和知识进行了介绍。

第三章进行了基于 FlowDroid 的 Android 应用程序组件内污点分析。首先对本文所 Flow Droid 工作原理进行了详细介绍；然后分步讲述了污点分析的过程以及提出的部分组件搜索模式。并对 IFDS 算法进行优化，对于需要计算的 data fact 集合 D 进行替代，找到最小等价集合

第四章介绍了基于字符串分析的 Android 应用程序组件间通信解析过程。通过对相关知识的介绍，以及具体的解析过程，并对结果进行进一步处理，生成 ICC 链接，构造 ICCG，最后再通过 ICCG 来判定组件间的污点传播。对第三章提取出来的程序特征进行特征变换，将其嵌入到特征向量，形成特征空间，采用多种机器学习算法进行分类器训练，并通过多组对比实验验证了该方法的有效性。

第五章对本课题的工作进行了总结，并提出了本课题仍存在的不足之处，介绍了未来的工作计划。

2 相关知识介绍

本节概述了 Android 操作系统，包括 Android 架构、应用程序框架和 Android 内置安全性。然后，概述了机器学习算法。最后，在本章的最后给出了本文的组织结构。Android 是由谷歌和开放手机联盟(OHA) [31]创建的一个平台，该联盟是不同移动技术公司之间的一个协会。因此，谷歌不允许任何制造商将其设备品牌化为安卓设备，除非其设备符合谷歌的要求。开发者下载或上传应用程序的官方市场是 Google Play，它拥有不同类别，如书籍、游戏、电影等。图 2.1 是 Google Play 网站的首页。

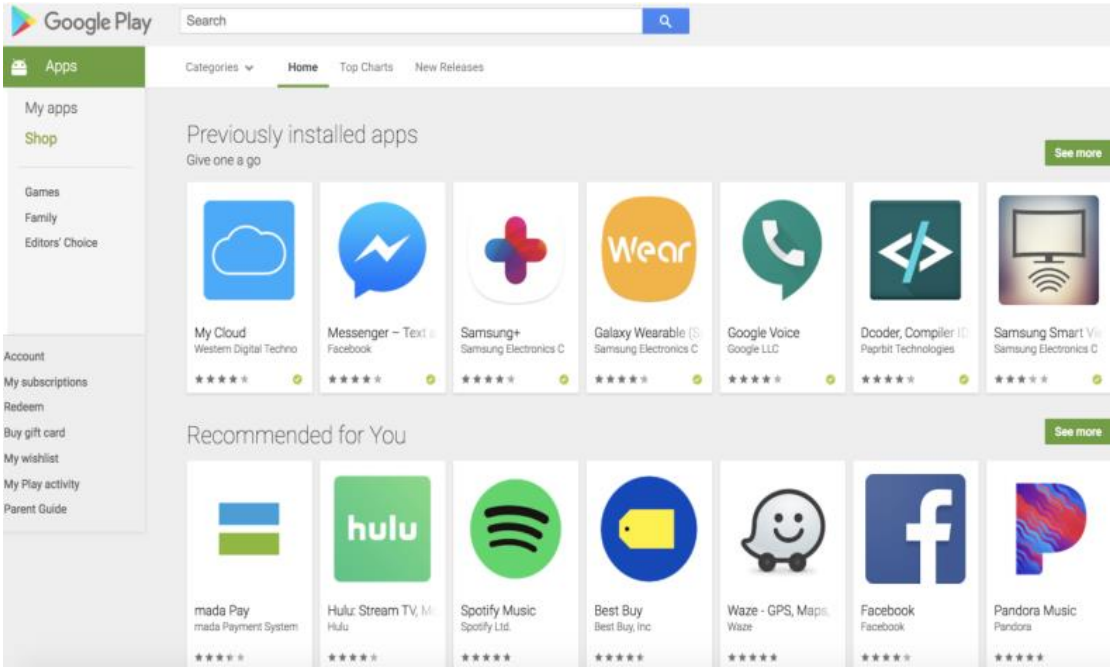

图 2.1 Google Play 网站的首页

除了官方市场，Android 允许用户从第三方商店下载应用程序没有任何限制。Google 负责运行和管理通过自己的数据中心提供的所有服务的物理服务器。Android 是开源的，也就是说开发者可以看到源代码，不需要任何费用和许可就可以修改[32]。开发人员可以通过使用免费且可访问的软件开发工具包(SDK)来构建他们的应用程序，该工具包提供了软件库、应用程序接口(API)、取证工具和仿真器。此外，Android 还提供了另一个开发工具包，称为本地开发工具包(NDK)，可以帮助游戏开发人员提高游戏性能。此外，NDK 允许应用程序开发人员在其应用程序中包含并运行一些本机代码。因此，Android 开发人员在开发应用程序时，可以将 SDK 和 NDK 结合起来。

Android 由四个主要层组成，如图 1.2 所示。这些层的基础是 Linux 内核，它是众所周知的操作系统之一，具有许多特性，如可移植性、安全性和开源。在 Linux 内核上面，我们可以看到 Native 库和 Android 运行时。然后，在此之上将是应用程序框架(Java API 框架)。用户安装或预装在设备上的所有应用程序都在系统应用层上运行，该层位于所有先前层的顶部。

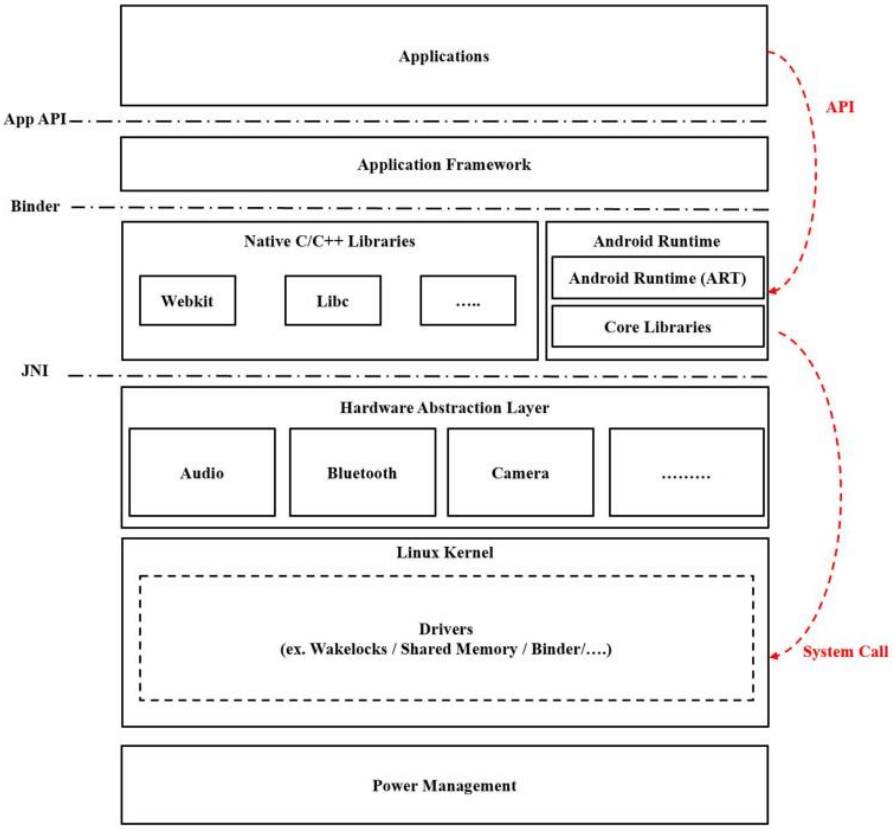

图 2.2　Android 架构

(1)Linux 内核:Android Linux 内核已经被修改，以克服开发者在 Android 操作系统早期版本中发现的一些限制。一些应用的修改包括内存绑定、匿名共享内存、进程内存分配器和偏执网络[33]。Binder 是 Android 的 Linux 内核最关键的变化之一。它是一种进程间通信(IPC)机制，使用进程标识(PID)和用户标识(UID)信息来识别调用进程。此外，它允许进程相互通信或与系统服务交互。

(2)Android 虚拟机(Android):如图 1.2 所示，Android 运行时(ART)与原生库共享同一层。它允许开发人员运行从应用程序生成的 Dex 字节码，并为应用程序提供所需的钩子、环

境、本机库等。与系统互动。ART 是运行 Android 5.0 及更高版本的设备的默认运行时。ART 通过引入不同特性，如提前编译(AOT)、改进的垃圾收集以及开发和调试改进，提高了 Android 操作系统的性能和流畅度[34]。此外，Java 不需要开发人员跟踪每个创建的对象。Java 让垃圾收集器删除所有没有活动代码引用的对象[35]。

(3)应用程序框架（Application Framework）:这是开发人员通过应用程序接口直接交互的主要层。这一层为开发人员提供了应用程序所需的重要资源[36]。通过使用这些 API，开发人员可以访问该层提供的一些功能，如表 1.1 [33]所示。其中一个重要的特性是内容提供者，它封装了应用程序的数据，并为 8 个表 1.1 框架层特性示例提供了策略.框架层的描述功能活动管理器管理应用程序的生命周期，如意图、活动和目的地等。查看开发人员用来构建不同组件(如列表、文本框、按钮等)的系统。资源管理器提供对图形、用户界面布局等的访问。这就是所谓的非代码应用程序资源。内容提供者封装应用程序的数据，并提供访问这些数据的策略。

表 2.1 框架层特性

框架层的功能	描述
活动经理	管理应用程序生命周期，例如意图，活动和目的地等
查看系统	开发人员用来构建不同的组件，例如列表。 文本框。 按钮等
资源经理	提供对图形化的访问。 uI 布局等。这称为非代码应用程序资源
内容提供者	封装应用程序数据并提供访问此数据的策略
通知管理员	允许所有应用程序在状态栏中显示警报

(4)应用层(Application):这是顶层。如图 1.2 所示，该层中的应用程序分为两种类型。第一种是重新安装的应用程序，由谷歌、制造商或移动运营商安装在设备上。这些应用程序安装在系统分区中，并使用 Android 平台密钥进行签名，这赋予了它们在系统分区中运行的权限。第二种是用户安装的应用程序，由用户从任何市场安装。这些应用程序使用开发人员创建的特殊密钥进行签名。因此，制造商和开发人员可以使用他们的特殊键为他们的应用程序推送任何更新[33]。

2.1 安卓组件

一个 Android app 不是封闭系统；安卓系统提供了一个应用运行的环境。应用程序生命周期中可能执行的代码并不都存在于应用程序的包中。安卓系统(包括安卓运行时)除了应用程序的代码之外，还做了大量的工作。

任何安卓应用程序都是使用一些松散连接的组件构建的。这些组件允许用户从一个屏幕切换到另一个屏幕。因此，开发人员可以根据用户的一些输入，灵活地选择应用程序启动时应该首先启动哪个组件，或者应该触发哪个事件。Android 有以下四个主要组件：

（1）活动（Activity）：活动是允许用户与应用程序交互的用户界面。它由窗口和其他用户界面组件组成，如按钮、文本框、消息显示等。任何应用程序都可以有一个或多个活动和用户可以根据用户事件从一个移动到另一个(向前或向后)。用户可以从另一个应用程序启动应用程序。例如，当用户正在写消息时，用户可以运行另一个活动(如照相机)来拍摄照片并将其附加到消息中。对于开发人员来说，所有应用程序的活动都应该在清单文件中定义。

（2）服务（Service）：当其他进程正在运行时，服务负责在后台运行长时间的操作。例如，一些应用程序可以使用另一个应用程序的核心功能之一作为服务。服务在没有任何用户界面的情况下运行应用程序的组件，例如在后台播放音乐。

（3）广播接收器（Broadcast Receiver）：广播接收器使用发布/订阅方式，当订阅事件被激活时，应用程序被触发。例如，开发人员可以定义一个广播接收器，以便在接收到新的文本消息时，在正在运行的应用程序的屏幕上显示文本消息。它可以在设备上的同一应用程序或不同的应用程序中使用，而无需使用任何视觉表示。如果开发人员想为他们自己的应用程序创建一个广播接收器，Android 给了他们定义自己权限的灵活性，以指定谁可以或不能访问所需的广播。

（4）内容提供商（Content Provider）：安卓操作系统的另一个重要特征是内容提供商。它是一个集中的单元，将相同的数据放在一个位置，并允许授权的应用程序访问它。例如，用户的联系人可以与其他应用程序共享，如拨号器、短信和联系人。只要这些应用程序有权限访问所请求的数据。因此，每个 Android 设备都有一个 ContentProvider 应用程序，它充当应用程序之间的结构化接口，以访问彼此的数据，应用程序的提供者之间共享的数据由 SQLite 数据库或直接文件系统返回。

2.2 安卓生命周期

在 Java 中，一个程序有一个唯一的条目来启动一个应用程序，比如 main()。在 Android

中，一个应用程序有一个或多个组件，如服务、广播和内容提供商，以实现这一功能。用户不负责管理应用程序之间的活动切换。例如，当用户想从一个应用程序切换到另一个应用程序时，用户只需按下主页按钮，启动一个新的应用程序，甚至恢复对先前应用程序的工作。

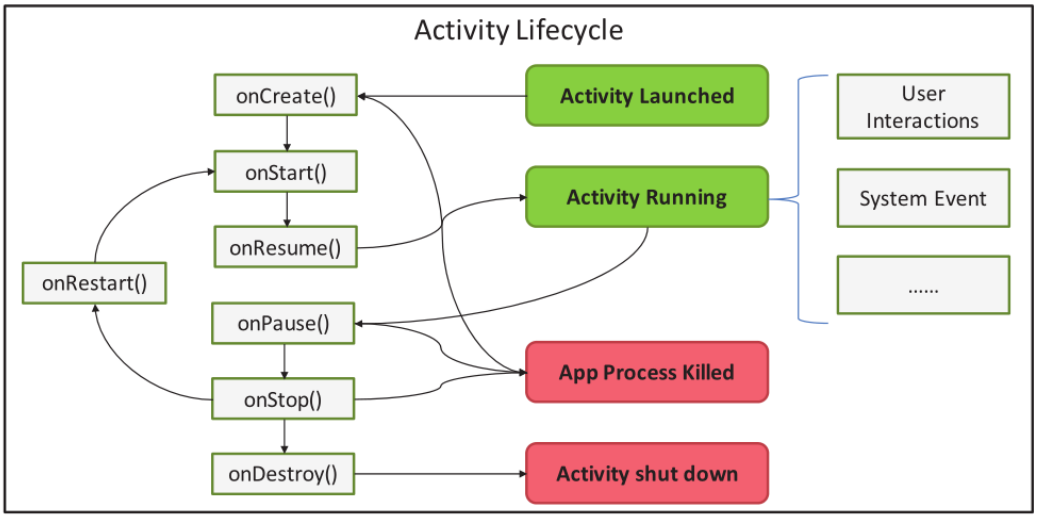

图 2.3 Android 生命周期图

2.3 安卓安全机制

2.3.1 Android 内置安全性

如前所述，Android OS 的基础是 Linux 内核，它为 Android 提供了权限、进程隔离、用户资源相互隔离等几个安全特性。和任何操作系统一样，Android 有两个不同空间：内核空间和用户空间。这些空间对双方都有不同的信任。例如，来自内核空间的任何操作都被信任与低级组件通信，并被允许访问虚拟和物理内存。另一方面，由于中央处理器的限制策略，用户空间不能执行这样的操作。本节讨论 Android 中的两个主要安全特性；应用程序沙箱和 Android 权限模型如下

（1）Android 的沙盒(Sandbox):Android 利用 Linux 内核，将系统资源与应用隔离，应用与应用相互隔离。Android 上的每个应用程序都被分配了一个 Uid，有时称为(应用程序 ID)，并作为一个独立的进程运行。此外，每个应用程序都有一个特定的数据目录，这是该应用程序唯一有权读写数据的目录，除非有另一个应用程序配置了相同的 UID。因此，应用程序在进程级和文件级都是沙箱化的。这种方法可以防止应用程序使用相同的内存空间以及读取或写入其他应用程序拥有的数据。但是，Android 允许由同一开发人员创

建并使用同一证书签名的应用程序在同一沙箱中运行。这种方法在 Android 操作系统中引入了一个称为应用共谋攻击的漏洞。例如，攻击者创建两个应用程序，每个应用程序通常分别执行，但是当这两个应用程序安装在同一个设备上时，可能会执行共谋攻击。

（2）安卓权限模式:安卓是基于 Linux 桌面的系统。前者为每个应用程序分配一个唯一的标识，而后者为每个用户分配一个唯一的标识。因此，在基于桌面的系统上以相同用户帐户运行的所有应用程序都具有相同的权限，这意味着没有针对每个应用程序的隔离机制[25]。但是在 Android 中，每个运行的应用都使用自己的 UID，将应用相互隔离，保证每个应用只能访问自己的数据。Android 不仅将应用程序相互隔离开来，还引入了一个权限系统来控制已安装应用程序使用的服务的使用。该系统控制可能包含用户敏感或危险信息的服务的使用。例如，开发人员可以使用电话管理员应用编程接口，该接口为他们提供敏感信息，如用户的电话号码、国际移动设备标识(IMEI)和国际移动用户标识(IMSI)。因此，Android 使用一种机制，在安装时向用户请求权限，以使用系统提供的这些服务之一。

2.3.2 权限机制

安卓系统中最关键的安全机制之一是权限系统。每个 Android 应用都在自己的沙箱中运行，这意味着它不能正常访问和利用设备的外部功能(如传感器、网络连接等)。然而，由于有了权限，应用程序可以克服这一限制并增加其功能。例如，如果应用程序希望访问系统摄像头来拍照，或者如果它需要从 SIM 卡读取联系人，它必须要求用户获得特定的权限。很明显，授予权限的过程意味着用户信任应用程序，从而导致安全隐患。所以，并非所有权限都以相同的方式处理，它们被分类为:

（1）正常权限：当应用程序第一次安装时，它们由系统自动授予。它们被认为对用户隐私和安全没有太大影响，因此用户无法从应用程序中撤销这些权限之一。例如振动(允许应用程序使用设备振动器)，蓝牙(允许应用程序访问设备蓝牙服务)和互联网(允许应用程序使用网络)；

（2）危险权限：这类权限必须由用户明确授予，因为它们被认为具有安全和隐私影响。出于这个原因，从 API 级别 23 开始，开发人员必须遵循特定的实现要求，以便通过对话框窗口正确提示用户，请求允许或拒绝授予他们的应用程序使用的最危险的权限。相反，对于较低的应用编程接口级别，该对话框仅在应用程序安装时显示。例如日历(允许应用程序读写日历事件)、照相机(允许应用程序启动系统照相机并拍照)；

（3）签名权限：相对于正常和危险级别的权限，它们提供了更强的安全性。事实上，这类权限是由系统自动授予的，但前提是请求该权限的应用程序与定义该权限的应用程序

具有相同的签名。这意味着，如果系统应用程序(如设置应用程序)定义了签名级别权限，则只有具有相同签名的应用程序才能被授予该权限，从而将其使用仅限于系统应用程序。如下表所示，是对常见的 android 应用程序高危权限的一个总结。

表 2.2 高危权限及功能描述

权限名称	权限功能描述
INTERNET	开启移动数据
READ_CONTACT	访问手机通讯录
READ_PHONE_STATE	读取手机状态
SEND_SMS	发送短信
RECEIVE_SMS	接收短信
READ_SMS	读取短信
CALL_PHONE	拨打电话
ACCESS_FINE_LOCATION	精确的位置信息

2.3.3 安卓恶意软件种类

安卓的不安全性主要分为三类

（1）预装应用攻击

在这种模式下，攻击者可以利用一些预加载应用程序的漏洞。如浏览器、邮件和天气应用。例如，攻击者可以使用钓鱼攻击，以获取一些敏感信息，如用户名和密码。这可以通过预装的电子邮件应用程序来实现。根据[37]跟电脑用户相比，移动设备用户更有可能将自己的敏感信息提交给钓鱼网站。这是由很多因素造成的，比如更多的人在手机上浏览互联网网页缩减了完整 URL 链接、警告信息等。这使得一些用户很难识别出不可信的网站。尽管如此，攻击者使用的另一种技术是利用一些预装的使用框架的应用程序，这是一种在 iFrame 中显示 Web/WAP 网站的方法。这种方法可能会导致点击劫持攻击，即用户通过单击的东西与他们打算点击的东西完全不同。此外，点击劫持已经不仅出现在应用程序上，而且出现在 Facebook 等网站上，将用户重定向到恶意网站[37]。由于移动设备上有很多敏感信息，攻击者会比点击劫持攻击更进一步。在安卓系统上观察到的"驾车式下载 "攻击比其他移动操作系统更多[37]。在这种攻击中，用户可能会访问其中一个被感染的网站，会在用户不知情的情况下下载恶意应用。这就是通过使用扫描设备安全漏洞的漏洞来实现。这已经是在安卓操作系统中报道较多，因为出现了严重的安全漏洞。在安卓 4.0 至 4.3 版本上，让攻击者可以通过 root 访问脆弱的手机，其中之一是

Towelroot[38]，这种攻击利用了一个漏洞在安卓系统的内核中，它使物理硬件和操作系统相互对话。最后，还有利用短信应用的 "手机中的人 "攻击。来嗅探所有发送到用户设备上的短信。例如，它将发送一条银行验证码被送到用户的设备上，在用户不知情的情况下以确认他们对第三方的身份。

（2）电话/短信攻击

这类攻击的主要功能是拨打电话或发送短信。大量信息不停的送到由专人监察的优质号码或收费服务。这为那些操作这些网络犯罪的攻击者带来了可观的收入。这种类型的攻击在 2014 年影响了 300000 台 Android 设备[39]。这种攻击是通过四个应用程序来实现，这些应用程序可以在官方应用商店中免费找到。一旦用户安装其中一个受影响的应用程序，并接受条件，该应用将受害者的电话号码发送给服务器，并将其签约为高级服务。它通过阅读由服务器发送的确认文本消息来确认受害者的订阅。然后，应用程序获取收到的代码，并确认受害者订阅服务。此外，还有一种攻击方式可以通过短信进行，称为 SMiShing。这种攻击类似于钓鱼攻击，在这种攻击中，用户被欺骗通过短信，攻击者向用户发送信息，让用户访问一个恶意的网站。

（3）安卓恶意软件

恶意软件是一种恶意软件，旨在进行有害活动，如禁用有效设备和窃取敏感信息。恶意软件有两种传播方式。第一个选项是在向市场发布之前，先创建一个有恶意活动的应用程序。第二种选择是，攻击者将恶意代码插入合法应用程序，然后将其重新引入市场。最近一个恶意软件的例子是 Stagefight，它引用了大多数应用程序使用的 StageSharedLibrarY。这种恶意软件允许网络罪犯通过向受害者的设备发送特殊的彩信来执行代码。根据 McAfee lab 的技术报告，这种恶意软件可以攻击运行 Android 1.5 到 Android 5.1 的有效设备，这些设备大约有 10 亿台。此外，超过 60 个安卓游戏被(安卓)感染。Xiny.19.origin，这是一种从用户设备收集敏感信息的特洛伊木马。表 2.1 显示了安卓反病毒软件报告的不同恶意软件类型。然而，一些应用程序允许在不安全的网络上传输数据，这使得数据容易受到许多攻击。此外，根据 nowsecure[39]的一份报告，83%的移动应用程序存储数据不安全。因此，这为许多恶意软件攻击用户的隐私信息打开了大门。

表 2.3 安卓恶意软件类型

类型	特征	例子
Spyware	间谍软件通常作为游戏安装，其主要功能是删除或传输数据。	Beaver Gang Counter

Worm	它创建自己的副本，并通过蓝牙将其发送到其他配对的设备。	Gazon Amazon Rewards
Botnet	使用僵尸程序控制受影响的设备，该程序使攻击者能够执行恶意行为。	Spam Soldier
Trojan	作为良性应用程序安装。一旦安装，它将在用户不知情的情况下启动恶意活动。	Fake Lookout
Backdoor	允许其他恶意程序绕过安全程序并进入设备。	Kmin
Adware	未经用户同意显示不需要的广告并发送敏感信息。	Hummingbad
Ransomware	使用户的设备无法访问，直到他们支付了一定数量的钱。	X bot

2.3.4 安卓组件间通信

Android 应用程序组件通过 ICC（Inter-Component Communication）交互——主要对象是 Intent。Intent 用于链接组件，并且可以在活动、服务和广播接收器之间发送。Intents 的主要功能是启动活动、启动和停止服务，以及向广播接收器发送广播信息。例如，开发人员可能希望启动组件在地图上显示用户的当前位置。她可以创建一个包含用户位置的 Intent，并将其发送到呈现地图的组件。开发人员可以通过两种方式指定 Intent 的目标组件：（1）显式：通过指定目标的应用程序包和类名显式地指定；（2）隐式：通过设置 Intent 的 action、category 或 data 字段隐式地指定。为了使组件能够接收隐式意图，必须在应用程序的清单文件中为其指定 Intent 过滤器。常用的 ICC 方法如表 2 所示。

定义 2　安卓应用程序组件之间的通信需要依靠 Intent 对象进行传递，我们将 Intent 对象定义为

$$Intent=\{Action,Data, Category\}$$

表 2.4 常用的 ICC 方法

ICC 方法	**作用**

startActivity	启动一个新的 Activity，但不会收到有关 Activity 何时退出的任何信息。
startActivityForResult	启动一个 Activity。退出此 Activity 后，将使用给定的 onActivityResult 方法
startService	请求启动给定的应用程序服务
sendBroadcast	将指定的广播发送到用户
query	查询
insert	插入
bindService	连接到应用程序服务
delete	删除

应用程序的每个组件都可以在 AndroidManifest.xml 中声明自己的 Intent 过滤器，该过滤器包含一个 Intent 必须持有的值。当发送隐式意图时，系统进程会将其字段与每个已注册的 Intent 过滤器的字段进行比较。如果找到精确匹配，操作系统会立即将意图传递给匹配的组件，否则用户将被提示一个对话框，从能够处理该 Intent 的应用程序列表中选择一个。根据意图目标，其字段可以填充预定义的常数值(Intent.ACTION_SEND)常量用于针对能够共享某些数据的组件的 Intent)，或者用于用户定义的组件。组件是通过各种回调方法(包括生命周期方法)调用的。根据事件的不同，系统调用组件的生命周期方法。它还会记住最近发送的 Intent 并传递它们，这些 Intent 可以在组件级环境中进行抽象。应用程序间通信(Inter-Application Communication)与 ICC 类似，IAC 研究安装在同一设备上的不同应用程序的组件之间如何传输敏感数据。ApkCombiner [18]等工具旨在通过将不同的应用程序组合成一个单一的 APK，现有工具可以在其上执行应用程序间分析，从而将 IAC 问题简化为 ICC 问题。Android 组件间通信示例如图 4 所示。

如下是一个示例 App 的部分代码。代码 1 是 FooActivity 方法的部分代码，而代码 2 是 BarActivity 方法的部分源码，代码 3 是 FooActivity 在 AndroidManifest.xml 文件中的声明，代码 4 是 BarActivity 所对应的声明。

代码 2.1 FooActivity

```
publicclass FooActivity extends Activity{
@Override
protected void onCreat(@Nullable Bundles avedInstanceState){
```

```
    super.onCreate(savedInstanceState);
    String imei2 = data.getStringExtra("key");
    SmsManager smsManager=SmsManager.getDefault();
    smsManager.sendTextMessage("xxx",null,imei2,null,null)//sink,;泄露设备 id

}
```

代码 2.2 BarActivity

```
publicclass BarActivity extends Activity{

    @Override
    protected void onCreat(@Nullable Bundle savedInstanceState){
    TelephonyManagerm=(TelephonyManager getSystemService(TELEPHONY_SERVIECE);
    this.imei = m.getDeviceId();//source 泄露源点
                String y = Extend(imei);
    Intent i3 = new Intent();
    i3.setAction("com.icc.ACTION");
    i3.putExtra("key",y);
    startActivity(i3);
    finish();
    }
Public static String Extend(String s){
        String a=getAddress();
        String g=a+s;
        return g;
}
```

代码 2.3 FooActivity 对应声明

```
 <activity
            android:name="com.icc.FooActivity"
            android:label="@string/app_name" >
            <intent-filter>
```

```
                <action android:name="com.icc.ACTION" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
```

代码 2.4 BarActivity 对应声明

```
<activity
            android:name="com.icc.BarActivity"
            android:label="@string/app_name" >
    <intent-filter>
      <action android:name="com.icc.EDIT" />
      <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
      </activity>
```

一般通过跟踪组件内部的控制和数据流来对整个 Android 应用程序进行分析，在上述代码中，BarActivity 通过 startActivity()启动了 FooActivity。当 BarActivity 通过 startActivity()发送了一个 intent i3，它的 Action 设置为 "com.icc.ACTION"表示发送的组件 Intent-Fileter 设置为这个参数。 Android 系统以 i3（即 data=i3）为参数调用的 FooActivity 中的方法，将 imei 的敏感信息返回给 FooActivity。所以示例中的整个泄露源头是 BarActivity 中参数通过 getDeviceId()获取了设备号，而泄露 的汇聚点就是 FooActivity 中通过 sendTextMessage()方法将隐私数据发送出去。

2.4 静态数据流分析方法

2.4.1 简介

静态污点分析是一种流行的信息流分析技术，它跟踪从一组敏感源到敏感接收器的敏感信息流。在我们的上下文中，源定义了我们希望在移动设备上保护的信息(例如，电话号码、联系人、位置和唯一的设备标识符)，而汇定义了不需要的信息发布点(例如，与互联网和短信传输相关的方法)。如果来自敏感源的数据到达接收器，污点跟踪会将从源到接收器的路径识别为数据泄漏的实例。污点分析可以静态和动态实现；本文主要关注静态污点分析技术，即那些通过分析 Android 应用程序的代码而不运行它来跟踪污点传播的技术。实现静态污点分析的工具包括 FlowDroid[19]、Amandroid[20]和

DroidSafe[21]。这些工具在设计决策上有很大的不同，它们采取的设计决策是为了使分析同时准确和可扩展。

2.4.2 分析的灵敏度

静态分析技术使用的不同算法的分析灵敏度不同，使得分析精度和可扩展性之间的权衡。分别对三个敏感度展示如图 2.4 所示。

- 流敏感。技术考虑语句的顺序，并为每个语句计算单独的信息。
- 上下文敏感。方法跟踪方法调用的调用上下文，并为同一过程的不同调用计算单独的信息。
- 路径敏感。分析将执行路径考虑在内，并区分从不同路径获得的信息。

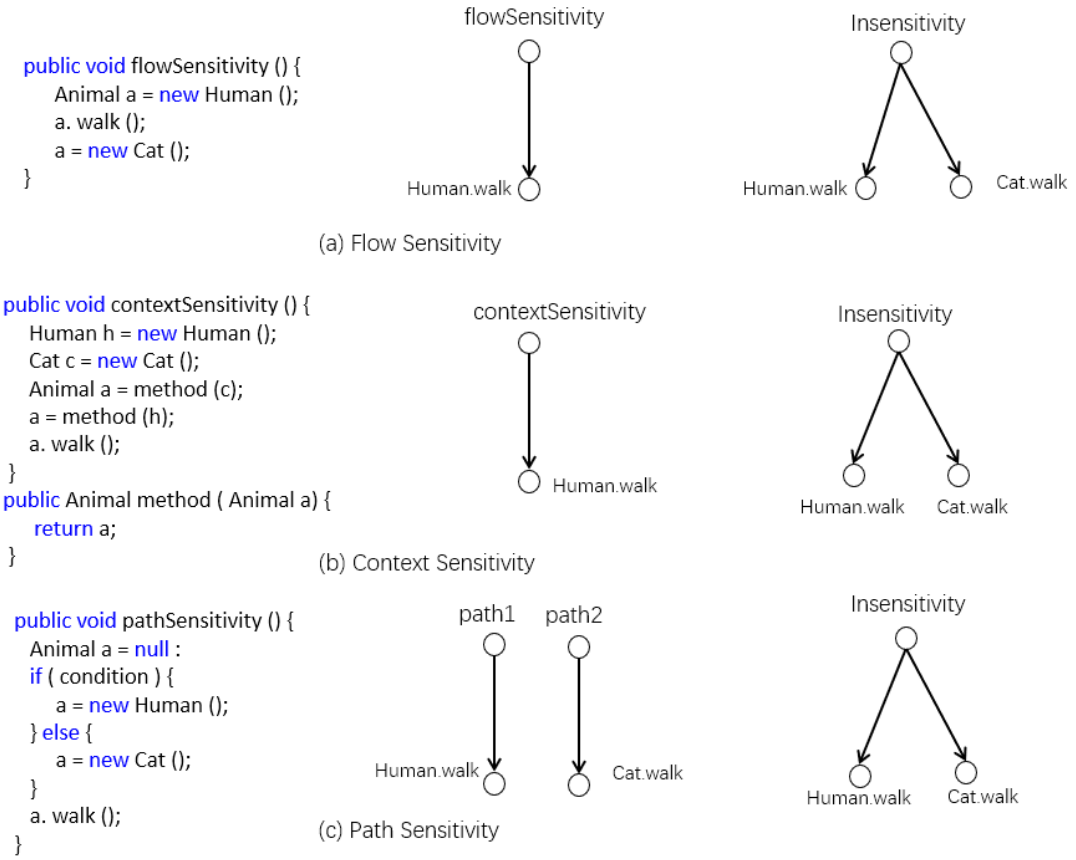

图 2.4　不同敏感度表示图

2.4.3 jimple 中间表示

每一种语言在编译期都会做一定的优化工作，但这些优化工作大多都是为了执行期的性能而做的。Java 字节码有超过 200 种不同的操作码，对其进行数据流分析必须模

拟 200 多种数据操作。为了降低分析的复杂性，静态数据流分析般不会直接在 Java 语言层面或者 Dalvik 虚拟机字节码层面上做分析工作，而是在一个易于识别的中间表示层进行分析。因此为了简化分析，诸如 FlowDroid 的静态数据流分析都是基于 Soot 编译器框架[22]基础上进行的。Soot 编译器框架提供了一种称为 JIR（Jimple Intermediate Representation）的中间表示。这种中间表示 JIR 只有 15 中不同的语句，用这 15 种不同的语句就能包含 29 种不同的语义表达。在程序编译阶段通过将 Android 应用程序源码转换成中间表示（Intermediate Representation），然后基于 IR 进一步进行静态分析。静态分析算法和方法通常在现成的框架上实现，这些框架对它们自己的程序代码的中间表示(IR)进行分析。将常用的 IR 作以下分类：

- Java 源代码，因为 Android 应用程序大多是用 Java 语言编写的。然而，这一假设限制了分析对开源应用或应用开发者的适用性。
- Java 字节码，与 Java 源代码相比，这大大拓宽了方法的适用性。与 Java 不同，Android 有自己的 Dalvik 字节码格式，称为 Dex，可由 Android 虚拟机执行。因此，这类工具需要在分析之前将 Dalvik 重定向到 Java 字节码，使用 APK-JAR 转换器，如 dex2jar [17]，ded[23]及它的扩展 Dare[24]。
- Jimple 是 Java 字节码的简化版本，每个语句最多有三个变量。它被流行的静态分析框架 Soot 所使用。Dexpler 是 Soot 框架的一个插件，它将 Dalvik 字节码翻译成 Jimple。
- Smali 是另一种中间表示，它由流行的 Android 逆向工程工具 Apktool 所使用[25]。

2.4.3 source and sink

所谓隐私泄露，是通过数据流分析方法分析出的合理的、有可能发生的一条从源(Source)和到泄漏点（Sink）之间的链接路径。检测源的位置通常为敏感数据的源头(Source)，一般是从某些域中读取的有关用户的隐私信息的函数，例如 getDeviceId()等。泄漏点一般是将获取到的隐私数据发送出去的函数，例如 sendSMS()等。

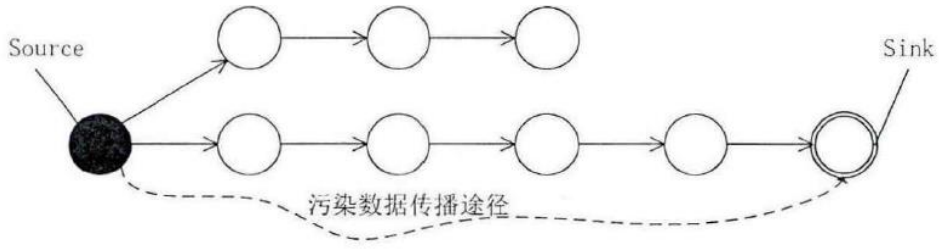

图 2.5 污点传播分析

一条隐私泄露表示从一个敏感 API 函数的调用开始，经过程序代码中的数据传播链路，传播到了另一种敏感 API 函数的调用位置，而在该函数中操纵了用户的隐私数

据，而且该函数试图将该隐私数据以 socket 读写等方式发送到网络中或其它应用，这便是一个泄漏点（Sink）。Arzt 等人[24]和 Li[26]等人使用了 SuSi 项目产生的源和汇列表[23]。表 2.5 列出了一些常用的 source,sink 的 API 以及他们对应的行为。

表 2.5 常用的 source，sink 的 API

Source	行为
getLongitude（）	获取经度
getLatitude（）	获取纬度
getDeviceId（）	获取设备号码 IMEI
getLastKnownLocation (java.lang.String)	获取位置信息
getLine1Number（）	获取电话号码
Sink	行为
SharedPreferences（）	放入字符串发送
sendRequestHeader(org.apache.http.HttpRequest)	通过网络发送
Log (java.lang.String)	传入日志
write (int)	写入文件
sendTextMessage(org.springframework.web.socket.TextMessage)	通过 message 传送 txt 文本

对source,sink按照类别进行分类总结，这里应该有一个大表

一条从 Source 到 Sink 的路径我们可以理解为一条隐私泄露路径。数据流分析的目标是为了检测出那些标记的源头（Source）所操作的数据是否能流向并且到达所定义的汇聚点（Sink）位置。对于检测隐私泄露来说，通常情况下我们关心的是用户的隐私敏感信息是否会被泄露到潜在的不可信的第三方应用程序中去。最简单的例子就是在一个安卓应用中读取用户通讯录的读取函数作为 Source，而所读取的通讯录数据在程序中进行传播，直到通过某一些 Socket 读写函数或者发送函数将此数据发送到其它远程服务或应用中。

2.5 FlowDroid

FlowDroid 是一个针对 Android 应用程序的流、上下文、字段和对象敏感的静态分析工具，它建立在 Soot 和 Dexpler 的基础上。它精确地模拟了安卓的生命周期，并通过用户界面对象的回调来处理数据传播。它还提供了最常见的 Android 框架方法的精确模型。对于剩下的，它应用了一个保守的策略，用至少一个被污染的参数污染所有的参数和方

法的返回值。FlowDroid 只能解析参数为常量字符串的反射调用；它将其异常处理机制建立在 Soot 的基础上。此外，FlowDroid 进行程序间数据流分析，因此无法处理 ICC 或 IAC。FlowDroid 的架构如下图 2.6 所示。

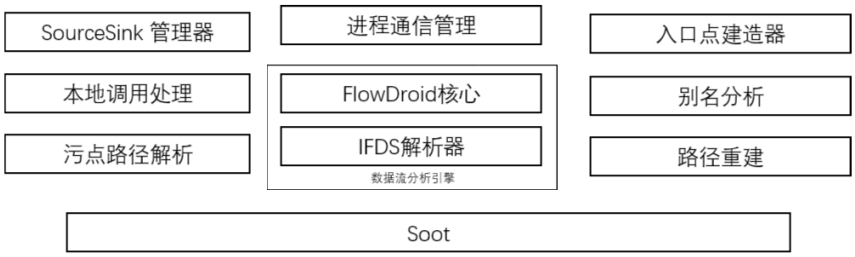

图 2.6 FlowDroid 架构

FlowDroid 的数据流分析是基于 IFDS 框架，它是数据流分析的精确而有效的算法，用于解决各种数据流分析问题[27,28]。IFDS 框架是数据流分析功能方法的一个实例[27]，因为它构建了被调用过程效果的总结。对于问题的求解 IFDS 问题求解算法的核心思想就是将程序分析问题转化为图可达问题。IFDS 框架适用于过程间数据流问题，其域由有限集合 D 的子集组成，并且其数据流函数是分布的。大多数其他过程间分析算法要么(1)由于无效路径而不精确，(2)通用但不在多项式时间内运行，要么处理一组非常具体的问题，IFDS 算法的输出是所有有效路径相交解。Reps 等[15]的 IFDS 算法是一种动态编程算法，它可以计算出程序间、有限、分布、子集的合并过所有有效路径的解。

格式不对

问题，合并是在有效路径上进行的，过程调用和返回是正确的。

匹配（即分析对上下文敏感）。该算法要求域由有限集 D 的子集组成，其数据流函数是分布式的。

数据流函数在集 union 上是分布式的条件为

$$f(x1 \sqcap x2)=f(x1) \sqcap f(x2)$$

一个 IFDS 问题的建模可以用实例 IP（Instance Problem）表示，其表现形式是一个九元组：

$$IP = (G^*, D, F, B, T, M, Tr, C, \prod)$$

其中：G^*是一个超图，如定义 2.2；D 是有限集的数据流事实（facts）的集合，代表数据流元素总集；F 是一个分布式数据流函数的集合，表示函数内部数据流传递关系，形式是 2^D->2^D,集合中的每一个元素都对应着一条 E_0 边上的数据流传递规则；B 是一个正数，在数据流传递中限制数据“带宽”；T 是一个满足分配率的函数集合，表示函数间数据流传递关系；M 是一个映射，将超图中的 E_0，E_1 集合里的边与边上的数据流函数 F 进行映射 M(m,n)->F(m,n)；Tr 是一个映射，将超图中的 E_1 集合里

的边与边上的数据流函数 *T* 进行映射映射 *Tr*（caller，callee）->*T*(caller,callee)，其中 caller 是主调函数，callee 是被调函数； *C* 是一个和超图 *G**中的 Smain 联系的特殊点；∏操作符是联合操作符。相遇操作符将始终是并集，但是当相遇是交集时，所有的结果都是双重应用的。

对于超图中的每个节点 *n*，IFDS 算法的输出是所有有效路径相交解

$$\mathrm{MVP}_F(n) = \prod_{q \in \mathrm{VP}(n)} M_F(q)(\top),$$

IFDS 核心思想是将任意分布式函数（distributive function）$f=2^D \rightarrow 2^D$ 转换为表示关系(representation relations)$R_f \subseteq (D \cup \{0\}) \times (D \cup \{0\})$。而表示关系 R_f 规则如下

- $0 \rightarrow 0$ 始终有一条边；
- $0 \rightarrow d_1$ $y \in f(\Phi)$ 若没有任何输入也能得到 y，则加上该边；
- $d_1 \rightarrow d_2$ y 可以从 x 得到，但不能从 0 得到，则加上该边；
- 还有一条，$d_i \rightarrow d_i$，与 d_i 无关时自己连自己，保持可达性。

通过以上的描述给出如下图 2.7 的示例。给定集合 D={u,v,w}和分布式函数 f(S)=S\{v}∪{u},那么图中的表示关系为 R_f={(0,0),(0,u),(w, w)} 。表示关系将流函数分别分解成每个数据流事实(fact)的函数(边)。而两个分布式函数的相遇操作可以简单的看作他们表示关系的联合 $R_{f \sqcap f'} = R_f \cup R_{f'}$ 。

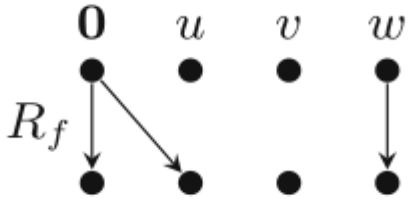

图 2.7 R_f 关系

原始的 IFDS 的算法如下图 2.8 所示。

```
declare PathEdge, WorkList, SummaryEdge: global edge set
algorithm Tabulate(G#_IP)
begin
[1]  Let (N#, E#) = G#_IP
[2]  PathEdge := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
[3]  WorkList := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
[4]  SummaryEdge := ∅
[5]  ForwardTabulateSLRPs()
[6]  for each n ∈ N* do
[7]    X_n := { d2 ∈ D | ∃ d1 ∈ (D ∪ {0}) such that ⟨s_procOf(n), d1⟩ → ⟨n, d2⟩ ∈ PathEdge }
[8]  od
end
procedure Propagate(e)
begin
[9]  if e ∉ PathEdge then Insert e into PathEdge; Insert e into WorkList fi
end
procedure ForwardTabulateSLRPs()
begin
[10] while WorkList ≠ ∅ do
[11]   Select and remove an edge ⟨s_p, d1⟩ → ⟨n, d2⟩ from WorkList
[12]   switch n
[13]     case n ∈ Call_p :
[14]       for each d3 such that ⟨n, d2⟩ → ⟨s_calledProc(n), d3⟩ ∈ E# do
[15]         Propagate(⟨s_calledProc(n), d3⟩ → ⟨s_calledProc(n), d3⟩)
[16]       od
[17]       for each d3 such that ⟨n, d2⟩ → ⟨returnSite(n), d3⟩ ∈ (E# ∪ SummaryEdge) do
[18]         Propagate(⟨s_p, d1⟩ → ⟨returnSite(n), d3⟩)
[19]       od
[20]     end case
[21]     case n = e_p :
[22]       for each c ∈ callers(p) do
[23]         for each d4, d5 such that ⟨c, d4⟩ → ⟨s_p, d1⟩ ∈ E# and ⟨e_p, d2⟩ → ⟨returnSite(c), d5⟩ ∈ E# do
[24]           if ⟨c, d4⟩ → ⟨returnSite(c), d5⟩ ∉ SummaryEdge then
[25]             Insert ⟨c, d4⟩ → ⟨returnSite(c), d5⟩ into SummaryEdge
[26]             for each d3 such that ⟨s_procOf(c), d3⟩ → ⟨c, d4⟩ ∈ PathEdge do
[27]               Propagate(⟨s_procOf(c), d3⟩ → ⟨returnSite(c), d5⟩)
[28]             od
[29]           fi
[30]         od
[31]       od
[32]     end case
[33]     case n ∈ (N_p − Call_p − {e_p}) :
[34]       for each ⟨m, d3⟩ such that ⟨n, d2⟩ → ⟨m, d3⟩ ∈ E# do
[35]         Propagate(⟨s_p, d1⟩ → ⟨m, d3⟩)
[36]       od
[37]     end case
[38]   end switch
[39] od
end
```

不能copy，要自己编辑。不一定按照原文中的算法，可以简化一些，对算法关键部分要做说明。

图 2.8 原始的 IFDS 算法

算法的输入是一个分解的超图，它既代表了被分析的程序也是数据流函数。超图是由程序的程序间控制流图(ICFG)构建的，将超图的顶点是一个对<l,d>，其中 l 是程序中的标签，d 是数据流事实并且 d∈D∪{0}。如果在 ICFG 中含有一个边 l->l'且 d0∈f({d})，其中 f 是指令语句 l 的分布式函数，那么在对应的超图中就会包含一个<l,d>→<l',d'> 的边。对于在 ICFG 中的程序间调用边或返回边，那么超图中就包含一组边，表示与该 call 或 return 相关的流函数。

对于算法中的数据结构，其中 PathEdge 是能够相连的两个节点的边，Worklist 是待处理的边的集合，SummaryEdge 是调用点到返回点的边，可以避免重复探索。Xn 则储存了该节点能到达的所有 data fact。而向前分析的过程通过遍历 Worklist 处理调用边，

返回节点以及其他节点的边。

2.6 IC3

安卓应用程序由组件组成，这些组件构成了应用程序的基本构件，并提供应用程序的功能。一些特殊的安卓方法被用来触发 ICC。这些方法被称为 ICC 方法，它使用一个独立的对象作为特定的参数，该参数保存发送方组件需要与之交互的接收方组件的信息。所有 ICC 方法都至少将一个 Intent 作为它们的参数。该意图可以被定义为在运行时将应用程序组件相互连接的异步消息。

2.6.1 格模型

Denning 提出了一种安全级别的格模型[19]：一个信息流安全策略被定义为一个“格”（Lattice）代数结构：

$$(SC, \leq),$$

其中 SC 为安全级别的有限集，“≤”为 SC 上的一个偏序关系。对于我们要讨论的 Android 安全信息流检测问题，安全级别可分为 High 和 Low 两个级别，即 SC={High, Low}，Low≤High。High 代表敏感信息的安全级别，Low 代表可以被外部观察到的公开信息的安全级别；如果信息从具有 High 级别的对象流向具有 Low 级别的对象，那么就会造成信息泄露。

2.6.2 COAL 语言

一个 Intetn 对象中有多个变量，包括 action、categories、data、minitype 几个属性。IC3 需要解决的问题是确定组件通信时参数 Intent 对象中所有可能的值。Octeau 等人[29]提出了 COAL 语言。COAL 是为解决这种多值复合常量传播问题设计的一种声明性语言,可以用来说明和查询多值复合常量传播问题。它说明了需要被推断值的变量的类型和这些值是通过什么方式被程序的语句修改的。它在 API 方法的语义上实现抽象推理。一个类的 COAL 模型的一般结构如下:

```
class android.content.Intent {
<field declarations>
<modifier method declarations>
<query declarations>
<source declarations>
<constant declarations>
}
```

field declarations 为建模的类指定字段名和字段类型。modifier method declarations 描述了修改字段的方法。这些查询指定求解程序应该在哪些程序位置计算对象值。最后，source declarations 字段获取器和 constant declarations 对存在模型化对象常数的情况进行建模。

（1）字段声明（Field Declarations），字段声明指定应该建模的字段的名称和类型。字段名不必与原始类中的字段名相同，只要命名方案与修饰符声明一致。例如，String action; Set<int> flags; Set<String> categories;

修饰符声明（Modifier Declarations），修饰符声明有以下一般形式：

mod <method signature> {
<modifier argument 1>
<modifier argument 2>
}

其中 modifier 参数是方法参数索引，操作以及字段名。例如一个 modifier 如下所示：

mod <android.content.Intent: android.content.Intent addFlags(int)> {
0: add flags;
}

大多数修改器只使用 mod 关键字。可以对其他类型的 Modifier 进行建模。比如 copy 关键字，gen 关键字。

（2）查询声明（Query Declarations），查询声明一般有以下形式：

query <method signature> {
<query argument 1>
<query argument 2>
}

它的参数声明表明了应该推断的方法参数的索引和类型。一个具体的查询声明如下所示

query context : void sendBroadcast(android.content.Intent,java.lang.String) {
0: type android.content.Intent;
1: type String;
}

（3）源声明（Source Declarations），它是在对字段获取器进行建模。当以 COAL 为模型的对象的字段值流向以 COAL 为模型的另一个对象的字段值时，它们很有用。源声明的一般形式如下:

source <method signature> {

<field name>

}

（4）常数声明（Constant Declarations），通过将常量声明为 COAL 规范的一部分，常数随后会在整个程序中传播，例如一个 uri 字段的定义如下所示：

staticfield uri <android.provider.Browser: android.net.Uri BOOKMARKS_URI> =

"content://browser/bookmarks";

对于要分析的 Intent 对象，需要根据对象的属性书写几个对应的 COAL 语言声明，先将 Intent 的对象进行声明，然后提出相应的声明，如下代码 2.5 所示。

代码 2.5 COAL 语言声明部分代码

```
String action;
Set<String> categories;
String package;
String clazz;
Set<int> flags;
Set<String> extras;
String dataType;
mod <android.content.Intent: void <init>(android.content.Context,java.lang.Class)> {
    0: replace package, type context;
    1: replace clazz, type Class;
}
mod <android.content.Intent: android.content.Intent
    putExtra(java.lang.String,double[])> {
    argument addExtra;
}

mod gen <android.content.Intent: android.content.Intent
    makeMainActivity(android.content.ComponentName)> {
    0: replace package, type android.content.ComponentName:package;
```

```
    0: replace clazz, type android.content.ComponentName:clazz;
    replace action "android.intent.action.MAIN";
    replace categories "android.intent.category.LAUNCHER";
  }
source <android.content.Intent: java.lang.String getAction()> {
    action;
  }
query context : void startActivity(android.content.Intent,android.os.Bundle) {
    argument intentActivity0;
  }
```

2.6.3 IC3 工作流程

IC3 工具通过基于 COAL 语言解决安卓应用程序组件间的通信值。通过程序的过程间控制流图，数据流函数以及格模型作为输入,输出复合常量传播问题的结果。具体步骤如图 2.9 所示。

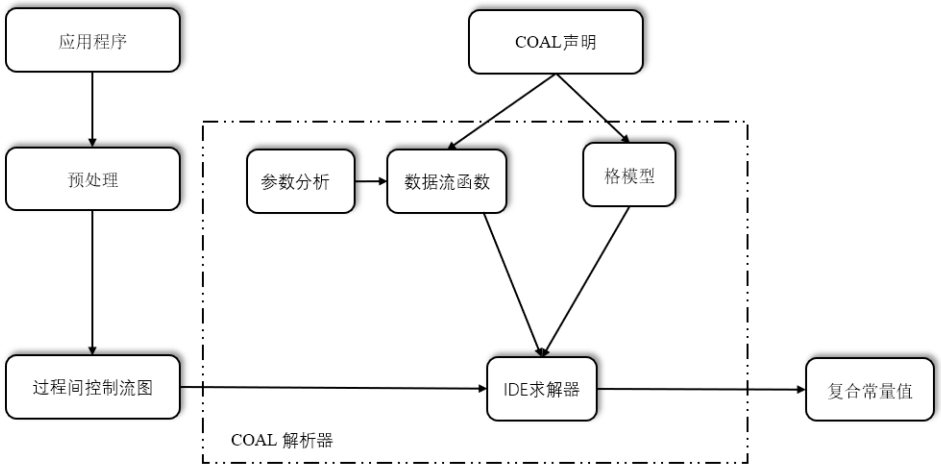

图 2.9　IC3 工具具体步骤

2.7 本章小结

本章介绍了安卓系统的一些基础知识，包括它的底层架构，体系结构以及安全机制。然后介绍了静态污点分析技术，以及相关的概念。同时还介绍了 Flowdroid 分析工具以及它的底层原理，最后介绍了组件分析工具 IC3 以及它的工作原理。

3 单个组件内信息流泄露检测

3.1 FlowDroid 工作流程

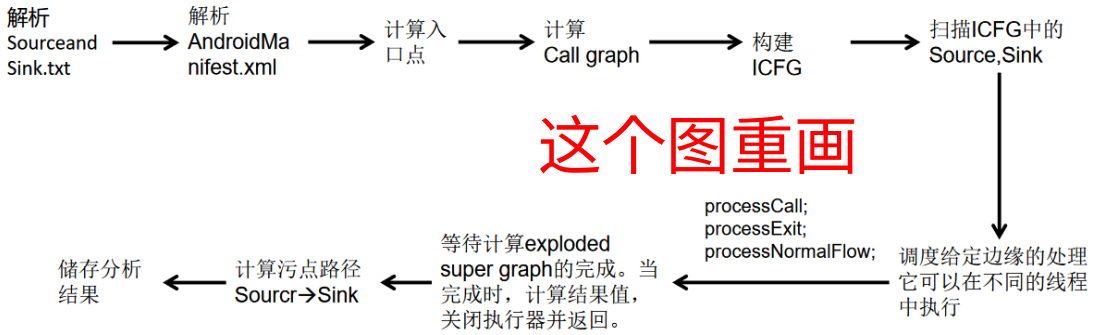

图 3.1 FlowDroid 详细工作流程

（1）解析 SourceandSink.txt

SUSI 等人通过机器学习的方法将 Android 应用程序中敏感的 API 进行区别，通过 SourceandSink.txt 文件将他们引入到程序当中。通过 java 里 IO 包提供的 FileReader 对 txt 文件进行提取。

代码 3.1 解析 SourceandSink.txt 部分源码

```
private void readFile(String fileName) throws IOException {
        FileReader fr = null;
        try {
            fr = new FileReader(fileName);
            readReader(fr);
        } finally {
            if (fr != null)
                fr.close();
        }
```

行距太宽

```
}
```

（2）解析 AndroidManifest.xml

AndroidManifest 文件，其中包含了重要的信息(如：应用组件、应用程序、应用系统和权限等信息)。AndroidManifest 文件，其中包含了重要的信息(如：应用组件、应用程序、应用系统和权限等)。通过对 AndroidMainifest 文件进行解析，了解这个应用程序声明得权限，组件等内容，如下图 3.2 所示。

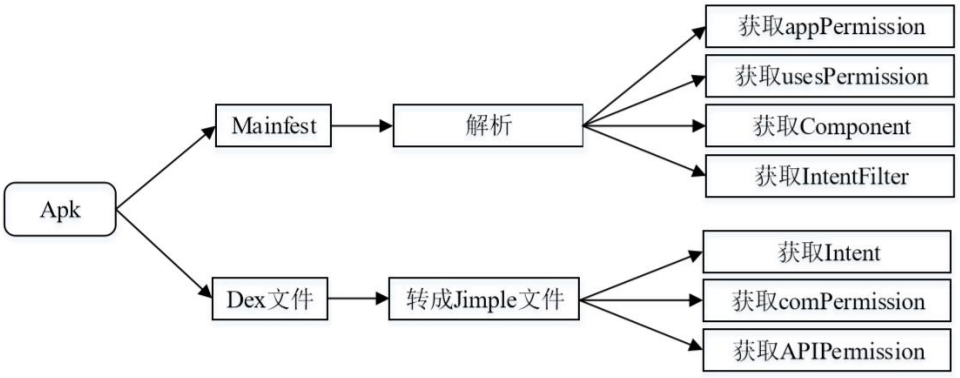

图 3.2 解析 APK 文件结果

对于具体代码实现如下所示。

代码 3.2 解析 Manifest 文件部分源码

```
final File targetAPK = new File(config.getAnalysisFileConfig().getTargetAPKFile());
        if (!targetAPK.exists())
            throw new RuntimeException(
                    String.format("Target APK file %s does not exist",
targetAPK.getCanonicalPath()));
        // Parse the resource file
        long beforeARSC = System.nanoTime();
        this.resources = new ARSCFileParser();
```

```
        this.resources.parse(targetAPK.getAbsolutePath());
        logger.info("ARSC file parsing took " + (System.nanoTime() - beforeARSC) / 1E9
+ " seconds");

        // To look for callbacks, we need to start somewhere. We use the Android
        // lifecycle methods for this purpose.
        this.manifest = new ProcessManifest(targetAPK, resources);

    SystemClassHandler.v().setExcludeSystemComponents(config.getIgnoreFlowsInSyste
mPackages());
        Set<String> entryPoints = manifest.getEntryPointClasses();
        this.entrypoints = new HashSet<>(entryPoints.size());
        for (String className : entryPoints) {
            SootClass sc = Scene.v().getSootClassUnsafe(className);
            if (sc != null)
                this.entrypoints.add(sc);
```

（3）计算入口点

与普通 Java 应用程序相比，Android 应用程序的一个主要区别是，Android 应用程序没有一个主方法作为它的入口点。一个 android 应用程序有多个入口点。由于移动计算环境的性质和 Android 操作系统的设计，Android 应用程序与我们所熟悉的桌面应用程序相比有一个非常独特的执行模型。也就是他的生命周期，每个组件都有相同的标准生命周期。要实现特定的组件，开发人员必须扩展该组件的基类并覆盖一组 Android 框架回调函数，如 onCreate、onStart、onStop 和 onDestroy。然后组件可以响应感兴趣的事件，如内存满、高优先级应用程序启动或用户导航回到之前的 activity，在某种意义上，Android 框架是对组件粒度的“调度”，一个应用程序可以从没有被 AndroidManifest.xml 禁用的任何组件开始。为了使用现有的静态分析框架和算法来分析 Android 应用程序，我们需要生成一个虚拟的 main 方法来建模 Android 框架的调用

行为和每个组件的生命周期。更具体地说，生成的虚拟 main 方法将连接到调用图上每个组件的所有可能的系统回调。这个虚拟的主方法作为整个调用图的根，我们的静态分析将从这个虚拟的入口点开始。一个 LeakageApp 的 main 方法如下图 3.3 所示。

Flowdroid 通过 Set<String> entryPoints = manifest.getEntryPointClasses();这个方法从 AndroidManifest.XML 文件中获取程序的入口点。

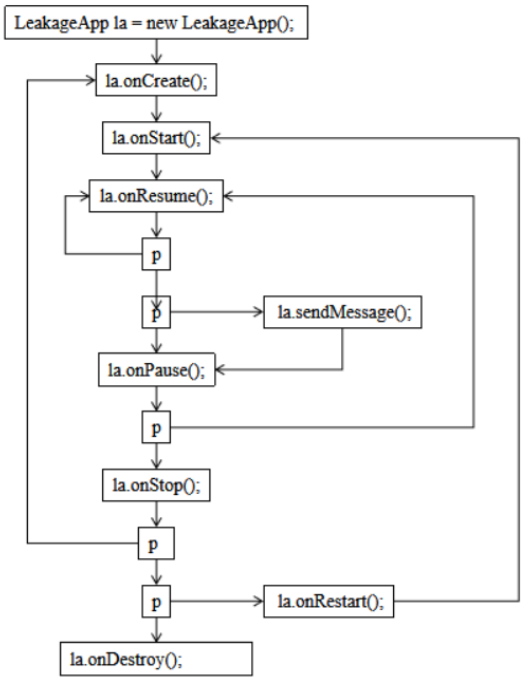

图 3.3 App 的 main 方法

（4）计算 Call graph

静态分析需要构建目标 app 的调用图（Call graph），指导即将到来的动态分析。不同于用包含一个主函数作为执行入口点的(桌面)程序，安卓应用程序不包含一个单独的主方法。相反，它们由多个组件组成，其中每个活动服务组件都是一个 Java 类，有自己的生命周期和事件侦听器。生命周期模型对组件状态之间的转换进行建模，如创建、暂停、恢复和终止。事件侦听器允许应用程序响应各种类型的运行时事件，例如用户界面交互或接收短信。生命周期和事件侦听器是由相应的回调方法构建的，每个回调都可以被视为一个入口点，因为它们是由 Android 框架隐式调用的。为了构建应用程序的调用图，FlowDroid [6]为整个应用程序创建了一个虚拟 main，所有组件共享该 main。一个示例 Call graph 如下图 3.4 所示。

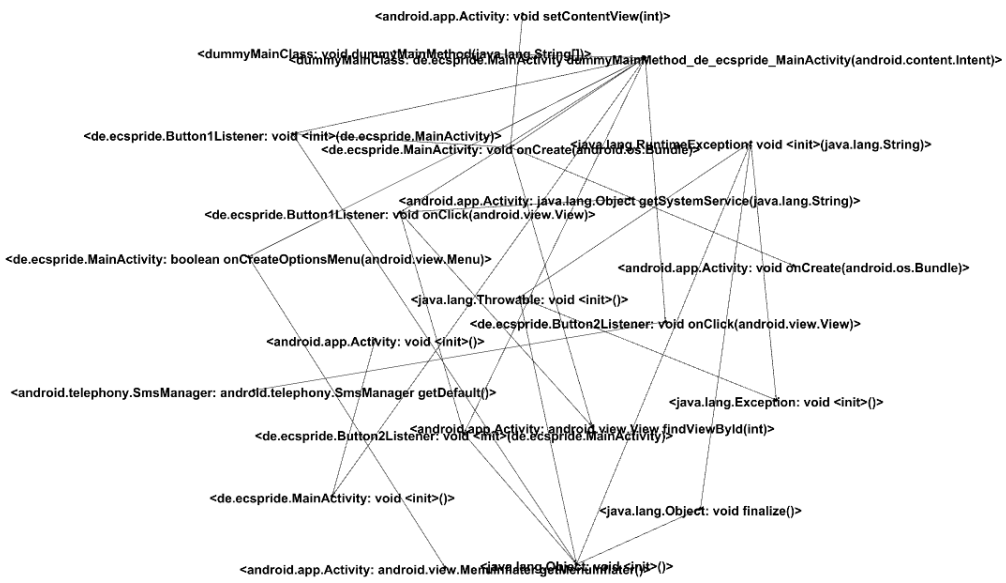

图 3.4 示例 Call Graph

Flowdroid 是基于 soot 工具生成安卓应用程序的调用图，代码中通过申请 soot 工具中“wjpp”和“cg”两个包（Pack）来实现，其中“wjpp”表示“the whole-jimple transformation pac”的简称，意思先将要分析的源代码转换成 jinmple 中间表示；“cg”表示“the call graph pack”,意思是 Soot 工具提供的生成调用图的功能包

代码 3.3 基于 Soot 生成 Call Graph 部分源码

```
PackManager.v().getPack("wjpp").apply();

PackManager.v().getPack("cg").apply();
```

（5）构建 ICFG

定义 3.2(Control-Flow Graph) 一个类方法 m 的控制流图 CFG 定义为一个有向图：

$$CFG =< N, E, head, tail >$$

其中：N 为一个方法内程序语句的有限集{s1, s2,…, sn}，程序语句包括赋值、条件判断、函数调用、调用返回等；$E \subseteq N \times N$ 是边的有限集，每条边 $e = (n_i, n_j) \in E$ 表示控制流从节点 ni 转移到节点 nj；$head \in N$ 是唯一的入口节点；$tail \in N$ 是唯一的出口节点。

由于程序中的类方法可以在其方法体的任意位置调用其他类方法(也可以调用自身以形成递归调用)，因此本文需构造一个图来描述方法调用和调用返回点的控制流转移，这种图称为过程间 CFG，或简称为 ICFG。ICFG 是将多个单独的 CFG 以及调用边和返回边集成在一起的超图。

而 IFDS 框架分析则需要基于整个安卓应用程序的 ICFG 它的数据结构定义为 DirectedGraph={Units, Map<Unit,List<Unit>> Preds, Map<Unit,List<Unit>> Succs, List<Unit> Heads,List<Unit> Tails}，其中，Units 为节点集，方法中的每条语句对应图中的一个节点，节点的类型为 Unit；Size 为节点的数量；Preds 为节点集中每个节点对应的前驱节点的集合；Succs 为节点集中每个节点对应的后继节点的集合；Heads 和 Tails 分别为方法的入口点集合和出口点集合。

（6）扫描 ICFG 进行数据流分析

FlowDroid 基于 ICFG，使用流和上下文敏感的 IFDS[36,37]进行污染和随需应变别名分析。调用图构造中的流不敏感性可能会引入假调用边(假阳性)，从而影响后续 IFDS 分析的分析精度。

执行源代码如下所示，通过对线程的 run 方法进行设计，使得它可以在不同的线程中执行。processCall()方法是分析调用边，processExit()方法是分析退出边，而 processNormalFlow()方法是分析其他普通的边。

代码 3.4 IFDS 线程调度的 run 方法源码

```
public void run() {
        final N target = edge.getTarget();
        if (icfg.isCallStmt(target)) {
            processCall(edge);
        } else {
            // note that some statements, such as "throw" may be
            // both an exit statement and a "normal" statement
            if (icfg.isExitStmt(target))
                processExit(edge);
            if (!icfg.getSuccsOf(target).isEmpty())
                processNormalFlow(edge);
```

```
            }
        }
```

（7）等待计算 exploded super graph 的完成。当完成时，计算结果值，关闭执行器并返回。

（8）计算可达污点路径

基于 ICFG 图通过 IFDS 算法进行可达性分析，最后将可达的具体 Source 和 Sink 方法计算出来，通过 SourceContextAndPath 类的对象来进行储存得到的结果。

代码 3.5 计算污点可达路径部分源码

```
protected Runnable getTaintPathTask(final AbstractionAtSink abs) {
        SourceContextAndPath scap = new
SourceContextAndPath(abs.getSinkDefinition(),
                abs.getAbstraction().getAccessPath(), abs.getSinkStmt());
        scap = scap.extendPath(abs.getAbstraction(), pathConfig);

        if (pathCache.put(abs.getAbstraction(), scap))
            if (!checkForSource(abs.getAbstraction(), scap))
                return new SourceFindingTask(abs.getAbstraction());
        return null;
    }
```

3.2 组件搜索模式

本文的污点分析主要基于 FlowDroid 这个工具，它的主要工作流程 ：构建虚拟入口点，控制流分析，数据流分析，结果输出处理。其中构建虚拟入口点就是对 Android 应用程序进行建模，包括文件解析、组件搜索、回调搜索等。控制流分析包括过程内控制流分析、生成 CFG（Control Flow Graph）、过程间控制流分析、生成 ICFG

（Interprocedural Control Flow Graph）等；数据流分析包括污点初始化，Source 和 Sink 的搜索，IFDS 问题求解，基于超图的一个图可达分析等。而对于 Android 生命周期进行建模时需要考虑两个问题。一个是入口点的处理，另一个是回调的处理。

组件搜索是对 Android App 进行初步解析，获取 Android App 中的组件列表和回调列表。在对 FlowDroid 进行测试实验的时候我们发现，它是一个重复迭代的过程，每当发现一个新的组件都会重新扫描整个 dummymain 函数，比较耗费时间。而我们每次只收集部分组件的回调信息，不以整个 App 为单位进行搜集。首先分析 AndroidMainfest 文件中静态组件的组件列表，然后每次只分析 N 个组件，在分析这 N 个组件的回调时，如果发现这些相关代码中注册了动态组件，则将新组件添加了到待分析列表中，等待下轮再进行分析，具体过程如下：

代码 3.6　部分组件搜索模式方法

```
输入 APK 文件
输出 <组件>集合、<回调>集合
通过 Androidmanifest.xml 文件搜集到组件集合
  foreach node in xml 文件
    if(node== activities)    checkAndAddComponent;
    if(node==providers) checkAndAddComponent;
    if(node==services)     checkAndAddComponent;
    if(node==receivers)    checkAndAddComponent;
  end
  for N components in < components >
    foreach component in components
      收集每个组件的 callback，并加入到集合当中;
      收集每个组件的动态组件，并加入到集合当中;
    end
  end
end
```

其中输入是安卓应用程序源文件，输出的是扫描到的组件的集合和回调函数的集合。首先通过提取 APK 文件中的 Androidmanifest.xml 进行扫描，如果属于 activities，providers，services，receivers 四大组件中的某一个，进行检查并将它添加到组件集合当中，直到将 Androidmanifest.xml 文件中所声明的组件全部扫描完成为止。然后基于已经扫描完成的组件集合，对于每个组件分析它的回调函数，以及相关的动态组件，将得到的回调函数添加到回调集合，将动态组件添加到组件集合，直到整个组件集合里的每个组件都被扫描之后分析结束。

3.3 污点传播规则

(1) Normal flow function：无论调用或者返回操作，分析时都要有 Normal 流函数。在本文采用的安全检测方法中，只有方法调用被视为污点源。因此, Normal 流函数不会产生新的污点，只能转化、保持或者杀死已有的污点。同时,本文采用的安全检测系统对数组切片也是不敏感的，一旦数组中的单个元素被视为污染,则整个数组都会被标记为污染。即使是被污染的单个元素后来又被未污染的元素值覆盖，整个数组仍然被标记为污染。

(2) Call flow function:调用流函数主要处理对从调用者到被调用者之间上下文环境变化进行建模，本文采用的安全检测系统将虚参替换为实参;并以调用者污染集为基础构造污染集合。在调用函数中，如果一个变量被污染,本文的安全检测方法就用 this 替换 s，使污染变量转换到被调用函数中。静态变量也会被拷贝到被调用函数中,但是本地变量则不会被传递到被调函数中。这样就能实现从调用函数的上下文转换到被调函数的上下文。

(3) Return flow function:在函数返回时,返回流函数将来自被调用函数的上下文的污点映射回调用函数,主要处理不可变的函数值。根据其特性，这些值无法改变自身的污点状态。因此，如果一个不可变类型的参数,比如说 string 和 int，在调用前没有被污染,那么在经过被调用函数时也不会被污染。本地变量对于被调用函数是封闭的，因此也不会被映射到调用函薮中。在其他情况下，所有的污点访问路径都会被秧射回调用函数的上下文中。

(4)Call- to-return flow function:对于每一次函数调用,都存在过程内边,这些过程内边独立地传播污点数据,与被调用函数无关。在 Call-to-returT 流函数中，通过种简单的模式匹配在 Source 处创建新的污点,本地方法调用也是在这里进行处理如果以参变量或基于对象开始的访问路径被标记为污染,那么就检测该方法是否被分析方法检测过,本文的分析方法主要分析污点是否允许流过该路径。由于在被调用函数中，污点可以被删除,这就

提高了精度。然而,如果不检测该方法,就需要假定对污点就行保留。所有其他的污点也需要继续进行传递。

对于整个污点分析，最主要的就是程序当中所存在的污点路径，可以直观的看到整个安卓应用程序的隐私泄露情况是如何。而通过控制流分析、数据流分析之后，对于分析的结果我们可以得到一对或者多对 source 到 sink 的泄露路径。本文给出以下的定义

定义 3.1 对于每一对 source 到 sink 的路径都可以认为是一个泄露路径对

$$Leakage\ Path=Pair(src,sin,Com)$$

其中 *src* 是这条泄露路径的 Source，而 *sin* 是这条泄露路径的 Sink，*Com* 则是这条泄露路径所存在的组件。*Leakage Path* 也意味着存在一条 Source→Sink 的路径。

3.4 样例分析

FlowDroid 对安卓单个组件内的核心检测方法是通过对程序生成过程间控制流图，然后进行污点分析。而它的污点分析主要是基于 IFDS 框架，它的核心思想是将程序分析转换为图可达问题。而图 3.5 是转换为 IFDS 问题后的图可达分析，通过将 BarActivity 中中数据流值转换为图中的节点。其中 getDeviceId 为 source，也就是隐私泄露点，imei 被标记为污点变量。StartActivity(i3)为泄露方法也是就 sink。当 imei = getDeviceId 时，会产生一条 0 到变量 imei 的边，然后对于函数调用，因为 imei 是函数 Extend 的参数，那么调用函数后，因为它的实参为污点变量，那么形参 s 也会变为污点变量，并产生一条 imei 到 s 的边。当传播到 Extend 函数返回时，因为所返回的参数 g 已经被污染，那么接受返回值的变量 y 也变成污点变量，并产生一条 g 到 y 的边。最终被标记的污点变量传到泄露方法，将隐私数据传播出去。

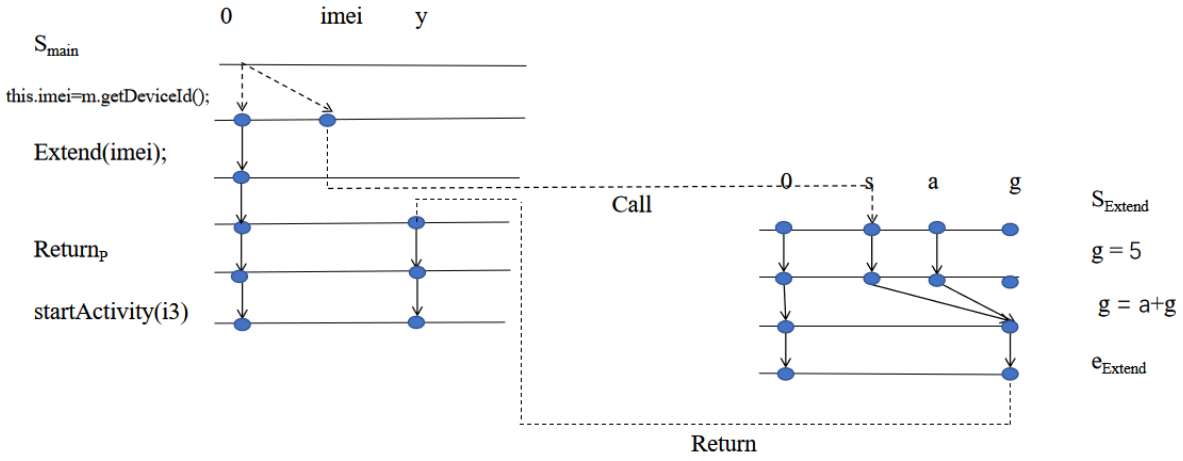

图 3.5　FlowDroid 污点分析求解示例　跨页

3.5 IFDS 优化

IFDS(过程间、有限、分配、子集)数据流问题解决了广泛的应用领域，包括模型检查、程序验证、切片、指针分析、动态生成、错误检测、安全分析和线程分析。数据流事实的集合被定义为 2^D->2^D 的数据流函数。

IFDS 问题求解算法的核心思想就是将程序分析问题转化为图可达问题。最终目的是求得所有可达路径 MRP（Meet-Over-All-Realizable-Paths)

定义 3.2 对于路径 p 的路径函数 表示为pf_p，对于 p 上的所有边（有时是节点）的流函数的组合

$$pf_p = f_n \circ \ldots \circ f_2 \circ f_1$$

定义 3.3 表示 CFG 中从起始节点到 n 的路径集表示为

Paths(start，n)

定义 3.4 对于每个节点 n，满足的所有路径定义为 MOPn (meet-over-all-paths)

$$MOP_n = \bigcup_{p \in Path(start,n)} pf_p(\bot)$$

定义 3.5 对于每个节点 n，满足所有可实现的路径定义为 MRPn(meet-over-all-realizable-paths)

$$MRP_n = \bigcup_{P \in RPath(start,n)} pf_p(\bot)$$

其中 *Rpaths*(*start*，*n*)表示可实现路径的集合从起始节点到 n。

根据定义，可以知道 MRP_n 和 MOP_n 的关系如下

$$MRP_n \sqsubseteq MOP_n$$

IFDS 问题实际上是识别（可达）Realizable path，沿着可达路径来传播数据分析。

定义 3.6 Supergraph：G*=(N*, E*)。

G*包含所有的流图 G1, G2, ... （每个函数对应一个流图，本例对应 Gmain 和 Gp）；

每个流图 Gp 都有个开始节点 sp 和退出节点 ep；

每个函数调用包含调用节点 Callp+返回节点 Retp。

一个示例程序如下所示，它的超图如下 3.6 所示

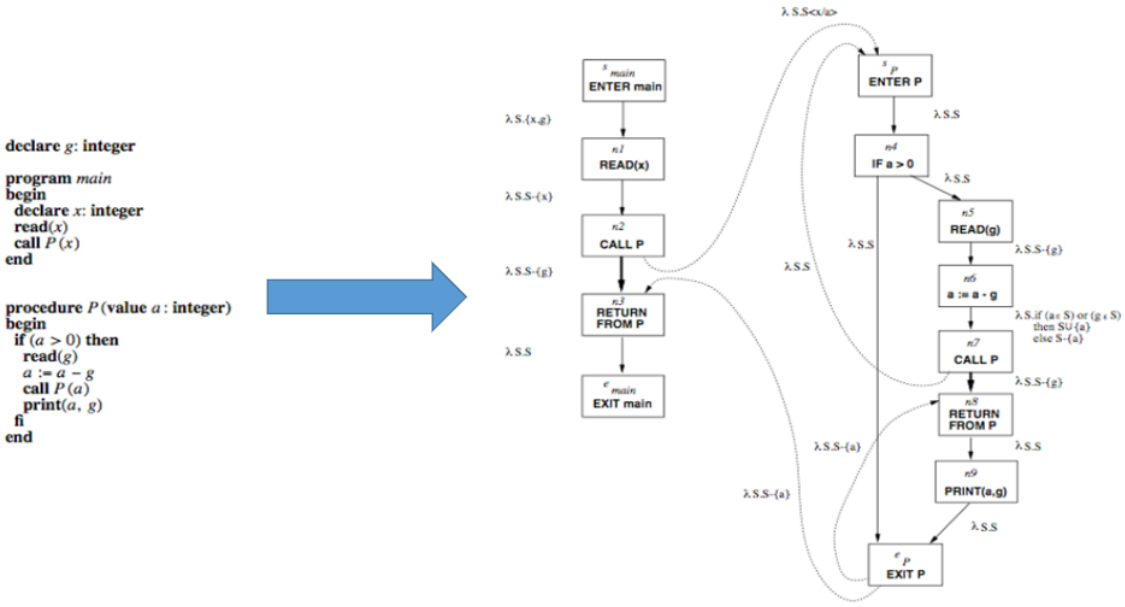

图 3.6 示例程序转化超图

IFDS 算法的分析过程在一个按照具体问题所构造的超图（exploded supergraph)上进行（如图 3.5 例子所示）.其中数据流值可以被表示成图中的节点，算法扩展了一个特殊数据流值：0 值，用于表示空集合；程序分析中转移函数的计算，即对数据流值的传递计算被转化为图中的边的求解.为了更好地进行描述，算法根据程序的特性将分析分解成 4 种转移函数（边）：1）Call-Flow，即求解函数调用（参数映射)的转移函数.2)Return-Flow, 即求解函数返回语句返回值到调用点的转移函数.3）CallToReturn-Flow，即函数调用到函数返回的转移函数.4）Normal-Flow，是指除了上述 3 种函数处理范围之外的语句的转移函数.IFDS 求解算法 Tabulation 是一种动态规划的算法，即求解过的子问题的路径可以被重复地利用，而由于其数据流值满足分配率，因此可以在分支或函数调用将边进行合并.求解算法包括了 2 类路径的计算：路径边集（path edges)和摘要边集（summary edges). 路径边集表示的是起点 0 值到当前计算点的可达路径，使用传播函数 propagate 将计算完成的边继续沿 CFG 传播到其下一个节点；摘要边集表示的是函数调用到函数返回的边，其主要特点是如果在不同调用点再次遇到同样的函数调用，可以直接利用其摘要边集信息从而避免了函数内重复的路径边集的计算。

如下是一个具体示例，它有类 A 包含方法 foo 和 bar，还有类 B 继承了类 A 也有方法 foo 和 bar 方法。

```
1  class A {
2    String foo { return secret(); }
3    void bar(String s) {}
4  }
5  class B extends A {
6    String foo {
7      return "not secret";
8    }
9    void bar(String s) {
10     System.out.println(s);
11   }
12 }
13
14 class Main {
15   static void main(String[] args) {
16     A a = (args == null)
17         ? new A() : new B();
18     String v = a.foo();
19     a.bar(v);
20   }
21 }
```

图 3.7　示例程序源码

在示例中，控制流与每个方法直接相关，变量 a 可以指向一个 A 对象或一个 B 对象，在代码第 18 行的调用可以分派给具体方法 A.foo()或 B.foo()，如从节点标记为 $call_{foo}$ 的边缘流出来的传到标记为 $start_{A.foo()}$和 $start_{B.foo()}$的节点上，并从节点的边缘传递到标记为 $end_{A.foo}$ 和 $end_{B.foo}$ 的节点然后再到 $return_{foo}$。同样，也有一个从标记为 $call_{bar}$ 的节点到 $start_{A.bar()}$和 $start_{B.bar()}$，然后通过路径传播到标记为 $end_{A.bar}$ 和 $end_{B.bar}$ 的节点然后返回到 $return_{bar}$ 节点。通过 IFDS 算法进行污点传播分析，它可以识别到通过一个 callsite 调用方法然后再从目标方法传回 return 节点。 就是对 foo()和 bar()的调用可能会分派给类 A 和 B 中的实现，因为接收者变量可能是绑定到运行时对象。但是不管生成 A 的对象还是 B 的对象，方法 foo()和 bar()都会在在同一对象上调用。因此，如果对 foo()的调用通过调用 A.foo()，则对 bar()的调用必须分派到 A.bar()，并类似地如果分派给 B.foo()则也是 B.bar() 所以，如图 3.8 中以粗体显示的路径，IFDS 算法会报告出调用调度到 A.foo()和 B.bar()，这个是不可行的。

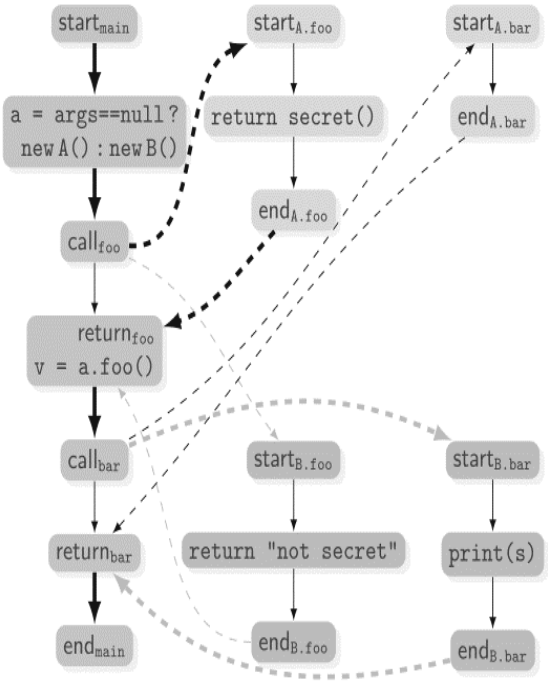

图 3.8 示例程序的超图（Supergraph）

所以设计了一个方法对 IFDS 算法优化方法，通过对 dataflow fact 集合 D 进行分析，寻找它的更小等价类进行替换。减少无效边的计算。在示例类型分析中，fact<x，Circle>包含 fact<x，Shape>，因为知道 x 指向类型为 Circle 对象意味着它的类型也是 Shape 的子类型。因此，如果分析针对某些程序点计算集合{<x，Circle>，<x，Shape>}，这表示 x 指向 Circle 的子类型或者指向 Shape 的子类型，与计算更小的集合{<x，Shape>}相比，计算{<x，Circle>}没有得到任何其他有用的信息。

定义 一个任意的分析， 如果把 a 包含 b，那么可以写为

$$a \leq b$$

例如上面的例子，<x，Circle>≤<x，Shape>。

我们将所有数据流函数按部分顺序成为：

$$a \leq b \Rightarrow \text{flow(a)} \leq \text{flow(b)}$$

定义 3.7 通过分析计算出的两个集合是等效的，并且每个集合的每个元素都被另一个集合的某个元素所包含可以写作

$$D_1 \leq D_2 \Leftrightarrow \forall d_1 \in D_1 \exists d_2 \in D_2 \; s.t. \; d_1 \leq d_2$$

$$D_1 \sim D_2 \Leftrightarrow D_1 \leq D_2 \wedge D_2 \leq D_1$$

原始 IFDS 算法只会计算{<x，Circle>，<x，Shape>}而不是更小的集合{<x，Shape>}。通过扩展算法使得在计算 D 时使用更小的等价集合。

3.6 本章小结

本章对安卓单个组件内的污点分析做了详细的介绍，以及示例分析，并提出了一个新的组件搜索模式，改善在对组件进行分析时在组件以及回调搜索模块上的消耗。并对 IFDS 算法提出优化，减少无效边的计算。

4 组件间信息流泄露检测

4.1 检测组件间的通信值

安卓应用程序由组件组成，这些组件构成了应用程序的基本构件，并提供应用程序的功能。一些特殊的安卓方法被用来触发组件间通信（ Inter-Component Communication ）。这些方法被称为 ICC 方法，它使用一个独立的对象(Intent)作为特定的参数，该参数保存发送方组件需要与之交互的接收方组件的信息。所有 ICC 方法都至少将一个 Intent 作为它们的参数。该 Intent 可以被定义为在运行时将应用程序组件相互连接的异步消息。 而 Intent 对象携带的信息指定了会被它启动的组件，以及被启动组件需要执行的操作。所以对 Android 应用程序组件间的信息流泄露检测，我们必须知道组件间的通信关系。

示例 Android 应用程的 AndroidManifest.XML 文件如下图 4.1 所示。其中第 2 行表示包名，第 4 行是这个应用程序所申请的权限，11 行，21 行和 30 行代表了不同的组件名称，13-17 行，23-26 行，32-35 行代表了这三个组件分别声明的 Intent 过滤器内容。

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android" package="edu.mit.icc_concat_action_
3      android:versionName="1.0" >
4     <uses-permission android:name="android.permission.READ_PHONE_STATE" />
5      <application
6          android:allowBackup="true"
7          android:icon="@drawable/ic_launcher"
8          android:label="@string/app_name"
9          android:theme="@style/AppTheme" >
10         <activity
11             android:name="edu.mit.icc_concat_action_string.OutFlowActivity"
12             android:label="@string/app_name" >
13             <intent-filter>
14                 <action android:name="android.intent.action.MAIN" />
15
16                 <category android:name="android.intent.category.LAUNCHER" />
17             </intent-filter>
18         </activity>
19
20         <activity
21             android:name="edu.mit.icc_concat_action_string.InFlowActivity"
22             android:label="@string/app_name" >
23             <intent-filter>
24                 <action android:name="edu.mit.icc_concat_action_string.ACTION" />
25                 <category android:name="android.intent.category.DEFAULT" />
26             </intent-filter>
27         </activity>
28
29         <activity
30             android:name="edu.mit.icc_concat_action_string.IsolateActivity"
31             android:label="@string/app_name" >
32       <intent-filter>
33         <action android:name="edu.mit.icc_concat_action_string.EDIT" />
34         <category android:name="android.intent.category.DEFAULT" />
35       </intent-filter>
36         </activity>
37     </application>
38 </manifest>
```

图 4.1 示例 XML 文件

基于 IC3 获取示例应用程序的组件之间的通信值，然后进行提取并进一步的分析。对于示例 IC3 分析产生的部分结果如下图 4.2 所示。我们需要根据 Intent 和 Intent 过滤器匹配规则对组件进行匹配，即可产生组件间的链接。

```
1  name: "edu.mit.icc_concat_action_string"
2  version: 1
3  used_permissions: "android.permission.READ_PHONE_STATE"
4  components {
5    name: "edu.mit.icc_concat_action_string.OutFlowActivity"
6    kind: ACTIVITY
7    exported: true
8    intent_filters {
9      attributes {
10       kind: ACTION
11       value: "android.intent.action.MAIN"
12     }
13     attributes {
14       kind: CATEGORY
15       value: "android.intent.category.LAUNCHER"
16     }
17   }
18   exit_points {
19     instruction {
20       statement: "virtualinvoke r0.<edu.mit.icc_concat_action_string.OutFlowActivity: void startActivi
21       class_name: "edu.mit.icc_concat_action_string.OutFlowActivity"
22       method: "<edu.mit.icc_concat_action_string.OutFlowActivity: void onCreate(android.os.Bundle)>"
23       id: 10
24     }
25     kind: ACTIVITY
26     intents {
27       attributes {
28         kind: ACTION
29         value: "edu.mit.icc_concat_action_string.ACTION"
30       }
31       attributes {
32         kind: EXTRA
33         value: "DroidBench"
34       }
35     }
```

图 4.2　IC3 部分结果图

其中第 3 行就是提取到的该应用程序所声明的权限，第 5 行为组件的名称，第 8-17 行为 Intent 过滤器里的内容，第 26-35 行表示的是这个组件中包含的两个 Intent 的内容。第 29 行的 Value 值表示将要把 Intent 发送给类别是“Action”值为“edu.mit.icc_concat_action_string.ACTION”的 Intent 过滤器。并且还携带有额外内容“DroidBench”。

4.2 组件匹配

安卓应用程序由组件组成，这些组件构成了应用程序的基本构件，并提供应用程序的功能。一些特殊的安卓方法被用来触发组件间通信（ Inter-Component Communication ）。这些方法被称为 ICC 方法，它使用一个独立的对象(Intent)作为特定的参数，该参数保存发送方组件需要与之交互的接收方组件的信息。所有 ICC 方法都至少将一个 Intent 作为它们的参数。该 Intent 可以被定义为在运行时将应用程序组件

相互连接的异步消息。而 Intent 对象携带的信息指定了会被它启动的组件，以及被启动组件需要执行的操作。Intent 中包含的主要信息如下表 4.1 所示。

表 4.1 Intent 对象主要信息

Intent 属性	内容
Component Name	它将会放入被该 Intent 启动的组件的名称。同时表明这是一个显式 Intent。创建显式 Intent 必须定义组件名，其他信息为可选
Action	指定被启动组件要执行的操作
Data	发送给目标程序的信息
Category	应处理 Intent 组件类型的附加信息

系统收到显式 Intent 后会在 AndroidManifest.xml 文件中寻找与 Intent 携带的信息中的组件名相同的组件，如果有，则启动该组件。系统收到隐式 Intent 时，将按照如下规则匹配 Intent 和 Intent 过滤器：

（1）匹配 Action：要指定接受的操作，Intent 过滤器可以指定零个或者多个 action。如果 Intent 中的 action 与 Intent 过滤器的 aciton 集合中的某一个相同，或者 Intent 和 Intent 过滤器均未指定 action，则两者的操作匹配；

（2）匹配 Category：要指定接受的类别，Intent 过滤器可以指定零个或者多个 Category 。如果 Intent 中的 Category 集合是 Intent 过滤器中 Category 集合的子集，则两者的类别匹配；

（3）匹配 URI：URI 包含四个元素：架构、主机、端口和路径，其中主机和端口合称为权限。Intent 和 Intent 过滤器的 URI 在以下情况匹配：a）如果过滤器只指定了架构，Intent 的架构和过滤器的架构相同；b)如果过滤器只指定了架构和权限，Intent 的架构和权限与 Intent 过滤器的相同；c)如果过滤器指定了架构、权限和路径，Intent 的架构、权限和路径与过滤器相同；

（4）匹配 Data：要指定接受的数据，Intent 过滤器可以指定零个或者多个 Data。Data 分为 URI 和 type，Intent 和 Intent 过滤器的 Data 在以下情况匹配：a）Intent 和 Intent 过滤器均未指定 data 和 type；b）Intent 和 Intent 过滤器都只指定了 URI，且两者的 URI 匹配；c）Intent 和 Intent 过滤器都只指定了 type，且两者的 type 相同；d）Intent 和 Intent 过滤器的 type 相同，且 Intent 同时包含 URI 和 type，在这种情况下，Intent 的 type 和 Intent 过滤器的才匹配。如果 Intent 的架构为 content 或 file 且过滤器未指定 URI，则

Intent 的与过滤器的 URI 匹配。URI 和 type 都匹配，数据才匹配。

只有操作、类别和数据均匹配，Intent 和 Intent 过滤器才匹配。具体的 ICC 匹配流程如下图 4.3 所示。

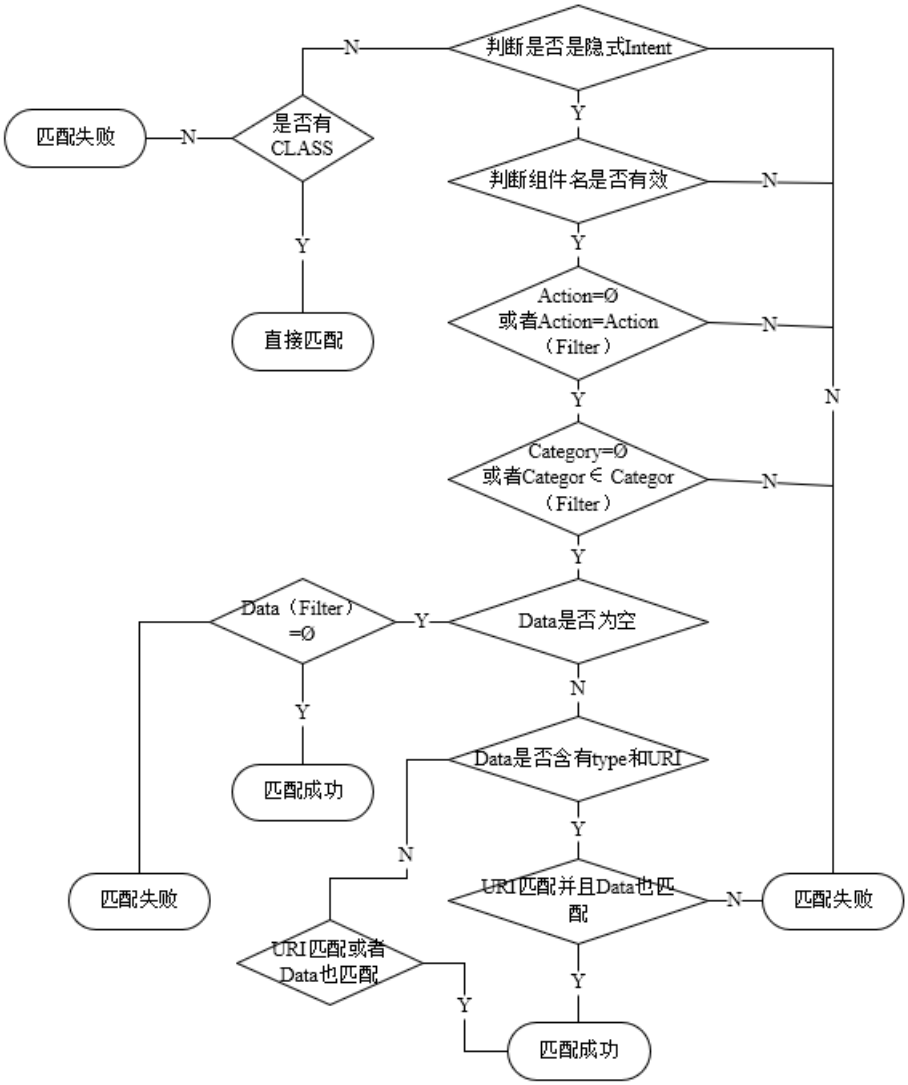

图 4.3 ICC 匹配流程图

对于能够匹配成功的 Intent 和 Intent 过滤器，我们有如下定义

定义 4 当一个组件 C1 和组件 C2 之间通过 Intent 通信时，我们可以定义 ICC 链接如下

Intent Link=I(C1,C2,Intent)

其中 C1 是发出 Intent 的组件，而 C2 是接收 Intent 的组件，Intent 是组件之间所传递的对象，其中包括动作、数据等信息。对于显示 Intent，我们直接通过类名进行匹配。对于隐式 Intent，我们需要先通过 Action 在 AndroidManifest.xml 文件中寻找对应的 Intent-Filter 从中找出要发送的组件名称，当符合上述组件间的匹配规则后，这两个组件之间

便会生成一个 Intent Link。

4.3 ICCG 图

对于组件间通信的结果，我们用一种新的数据结构来保存它。定义如下

定义 5 如果安卓应用程序组件间存在互相通信，我们可以对他们的调用关系作一定义如下：

$$ICCG=(N,E,I,L)$$

其中 N 是是 Android 应用程序中的组件的集合，E⊆($N\times N$)，I 是定义 4 中的 Intent Link ,L 是边 E 的映射。通过将 IC3 分析结果进行链接后，对每个 App 构造一个的组件间调用图（ICCG）。基于 ICCG 的定义，我们给出了一个 ICCG 快速构造的算法，通过将 IC3 分析结果进行链接后，对每个 App 构造一个的组件间调用图（ICCG）。

如下图 12 所示，是一个安卓应用程序组件间的控制流图，其中图中每个节点是各个组件的名称，边为传递的 Intent 对象包含的数据。

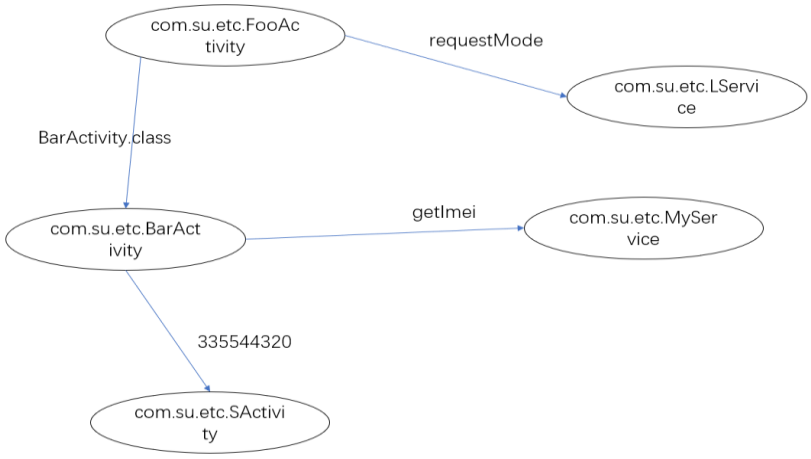

图 4.4 一个示例的 ICCG

4.4 组件间的数据流分析

Flowdroid 只能检测到单个组件内的隐私信息泄露，所以我们需要通过检测组件间通信值来分析组件间的隐私信息泄露基于组件间通信值生成的 ICCG 图，可以进行数据流分析，判断它的跨组件的污点路径。

对于一个 ICCG，他表示了这个安卓应用程序所有组件之间的通信关系，它是由许多的 ICC Links 组成。如果存在某 ICC Links L(C1,C2,Action,Data)，则表示从一个 Pair(src1，

sin1，C1)到另一个 Pair(src2，sin2，C2)的转换，而对于跨组件的污点传播路径便由 src1->sin1 和 src2->sin2 变成 sin1->src2。基于 ICCG 将跨组件的检测转换为全新的 source→sink 的污点路径跟踪。转换过程如下图 4.5 所示

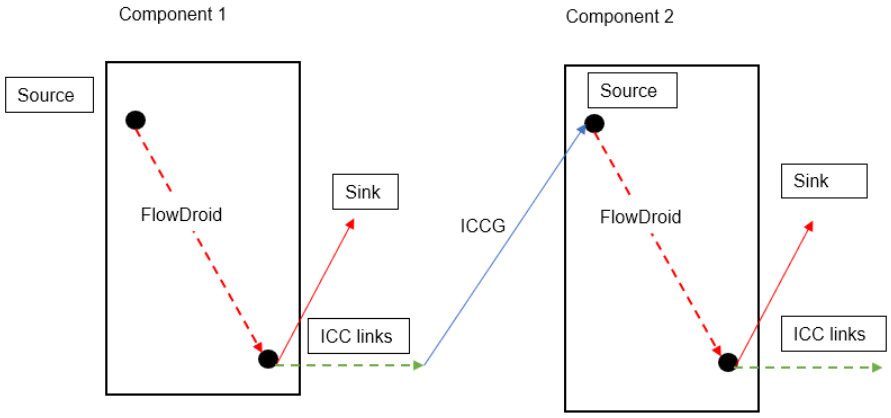

图 4.5 组件间数据流传递

组件间的污点路径整合后，组件间的污点路径如下图 4.6 所示，组件 Ci 调用 ICC API 方法从 srci 发送数据 Intent Ii 给组件 C2，组件 C2 从 Intent Ii 读取数据，并通过调用返回类的 API 方法将数据发送回组件 Ci，或者组件 C2 直接将隐私数据通过 sink2 发送出去，组件 Ci 通过 onActivityResult 方法从返回结果中读取数据并将其写入 sinki。那么这三个组件之间最后得到的污点路径便是 src1->sink1，src1->sink2,src3->sink2 以及 src3->sink3 这四条条路径

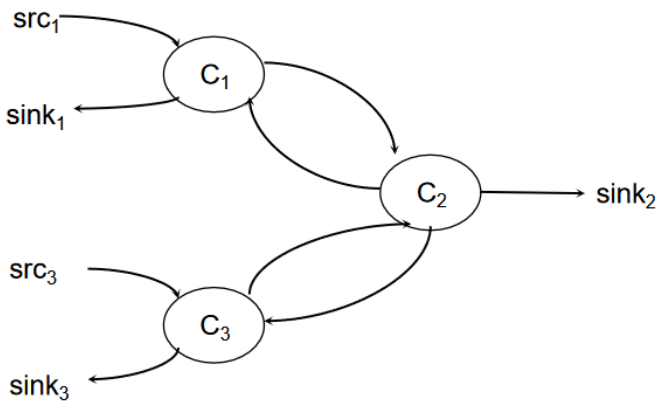

图 4.6 组件间的污点示例图

而上述提到的 ICC 方法，通过对许多应用程序分析，总结了一些常见的 ICC 方法，如下图所示。其中最常用的就是 startActivity 和 startActivityForResult，他们分别代表着发送 Intent 值给另一个 Activity 组件和返回 Intent 值给之前的 Activity 组件。

表 4.2 新增加的 Source 和 Sink

Source	Sink
getIntent(java.lang.String)	startActivity(android.content.Intent)
getAction()	startActivityForResult(android.content.Intent, int)
getCategories()	setResult(int,android.content.Intent)
getComponent()	setAction(java.lang.String)
getData()	setClassNameandroid.content.Context,java.lang.Class)
getExtras()	putExtra(java.lang.String,int)
queryBroadcastReceivers(android.content.Intent,int)	startService(android.content.Intent)
onActivityResult(int,int,android.content.Intent)	sendBroadcast(android.content.Intent,java.lang.String)

如果存在某个 *L*(*C1,C2,Intent*)则需要从一个 *Pair*(*C1，src，sin*)到另一个 *Pair*(*C2，src，sin*)的转换。基于 ICCG 将跨组件的检测转换为全新的 Source→Sink 的污点路径跟踪。

基于 ICCG，对于某个把包含污点的 Intent 对象传递给其他组件，那么这条边也会变为受污染的边，边链接的两个组件都是被污染组件，那么他们所使用的 ICC 方法已经变为 Source 方法或者 Sink 方法。这样就可以在单个组件内进行污点分析，判断是否有污点被泄露。而对于代码 1 和代码 2 中示例的组件间通信过程，可以转为如下图 4.7 所示的污点分析过程。

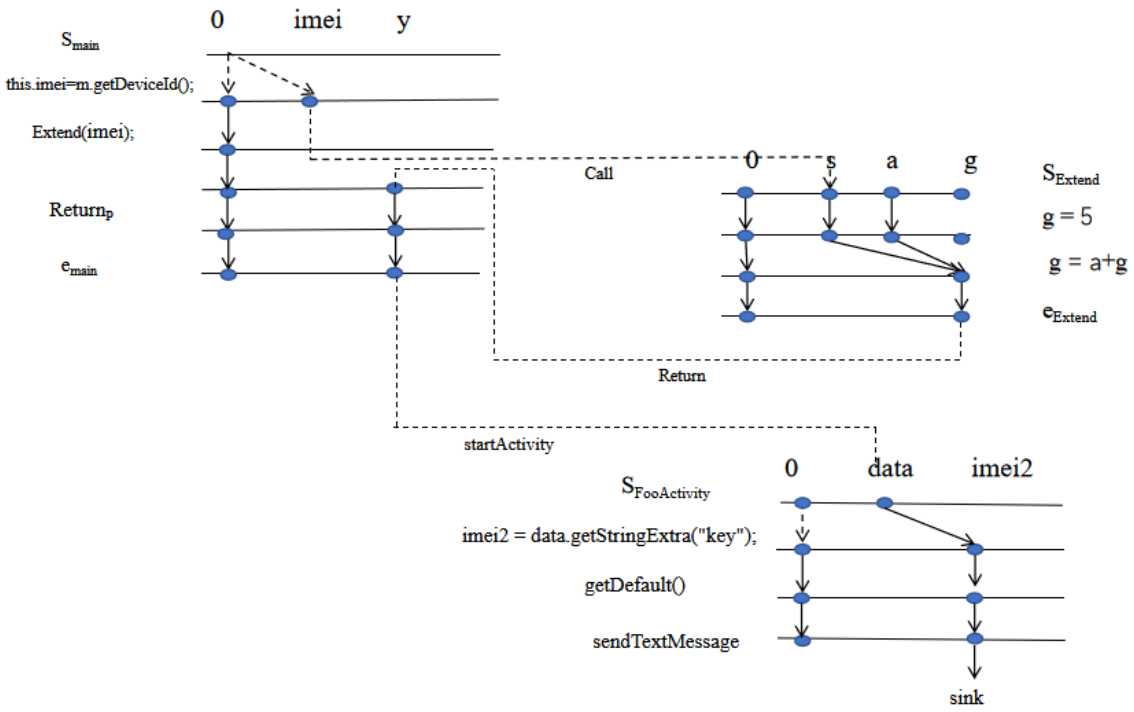

图 4.7 组件间的污点分析示例

5 实验与分析

5.1 实验配置

5.1.1 FlowDroid 关键配置

- -ImplicitFlowMode.NoImplicitFlows；不跟踪隐性流。
- -FlowSensitiveAliasing；别名搜索对流的敏感性。
- -SequentialPathProcessing;；执行顺序路径重构
- -LayoutMatchingMode.NoMatch；不使用 Android 布局组件作为来源
- -StaticFieldTrackingMode. ContextFlowSensitive；作为普通污点抽象，跟踪静态字段上的污点。这种方法是 对上下文和流程敏感。

FlowDroid 提供了一些不同配置参数，以上是本文测试所采取的一些关键设置及介绍。详细介绍

ImplicitFlowMode 参数表示在数据流分析时支持对隐式流处理所能采用的模式，ImplicitFlowMode.NoImplicitFlows 表示将不会去处理隐式流，ImplicitFlowMode.ArrayAccesses 表示使用污染索引访问无污染数组时创建新的污染，但不传递其他控制流依赖项，ImplicitFlowMode.AllImplicitFlows 表示追踪所有的污点数据的控制流依赖。

FlowSensitiveAliasing 设置是否使用别名搜索时对流的敏感性，他的选项有 ture 和 false，我们设置为 ture。

SequentialPathProcessing 设置 FlowDroid 是否应执行顺序路径重构，而不是同时运行所有重构任务。这样可以减少内存消耗，但是当内存不是问题时，可能会花费更长的时间。

LayoutMatchingMode 将布局组件匹配为数据流分析中 Source 的可选项，其中 LayoutMatchingMode.NoMatch 表示不将 Android 布局组件用作 Source，LayoutMatchingMode.MatchAll 表示使用所有布局组件作为 Source，LayoutMatchingMode.MatchSensitiveOnly 表示仅使用敏感的布局组件（例如密码字段等）作为 Source 进行数据流分析。

StaticFieldTrackingMode 跟踪静态字段上的污点方法。

StaticFieldTrackingMode.ContextFlowSensitive 表示对上下文和流程敏感，StaticFieldTrackingMode.ContextFlowInsensitive 表示跟踪静态字段上的污点，既不是上

下文相关的，也不是流量敏感的，这样做可以减少时间的浪费，
StaticFieldTrackingMode.None 表示不追踪任何静态字段。

5.1.2 实验环境

系统为 windows10 版本，FlowDroid 版本为 2.5.1，IC3 版本为 0.2.0。CPU 为 Intel Core i7，1.8kHz，内存大小 16GB。

5.2 测试集

我们所采用的是测试集是 DroidBench，是最广泛的基准测试套件，由一组 Android 应用程序组成，用于评估静态数据流分析工具的准确性和精度。在我们的评估中，我们使用了 DroidBench3.0 版本，并且我们在 Google Play 中下载了排名较高的 15 个应用进行测试。

5.3 实验结果

我们通过对 Google Play 中随机选取下载量较高的 15 个 APP 上比较了我们的工具的性能和原始 FlowDroid 的性能。表 6 给采取得应用程序情况，包括他们每个应用程序的大小和类别以及通过我们的工具对他们检测的一个结果。

5.3.1 IC3 能力分析

安卓应用程序组件间通讯需要依靠Intent对象，用于执行或运行另一个组件的操。Android中有两种意图:显式和隐式。在显式意图中，开发人员指定哪个组件接收意图。此类型仅适用于同一应用程序中的组件。另一方面，隐式意图指定所需组件的类型，并让用户决定如何继续。例如，如果用户在设备上安装了两个互联网浏览器，并且有打开链接的意图，则用户可以从这两个浏览器中进行选择。那我们在分析安卓应用程序组件间的通信时首先需要分析出传送的Intent对象中字符串的值。我们采取现有的IC3 和dialdroid进行测试对Intent对象中的字符串解析的能力。其中daildroid是基于IC3 工具优化的工具。通过以下三个案例进行对比说明。
案例一通过字符串的形式将值传递给 Intent 对象，其中图中 a 是两个常量字符串”com.icc”和”.ACTION”直接相加，并直接传递给 Intent 对象。

```java
public class OutFlowActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        TelephonyManager telephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
        String imei = telephonyManager.getDeviceId(); //source

        Intent i = new Intent("com.icc"+ ".ACTION");
        i.putExtra("test", imei);

        startActivity(i);
    }

}
```

(a) 程序部分源代码

```java
public class OutFlowActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        TelephonyManager telephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
        String imei = telephonyManager.getDeviceId(); //source

        String str = "com.icc";
        Intent i = new Intent(str + imei);


        startActivity(i);
    }

}
```

(b) 程序部分源代码

图 5.1 案例一部分源码

如下图 5.2 为案例二的部分源码其中 a 通过使用 String 类提供的 concat()方法将两个字符串 str1 和 str2 拼接为一个新的字符串对象然后将这个对象赋值给 Intent 对象，而(b)通过将常量拼接变量的方式将值传送给 Intent 对象。

```
public class OutFlowActivity extends Activity {


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        TelephonyManager telephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
        String imei = telephonyManager.getDeviceId(); //source

        String str1="com.icc";
        String str2=".ACTION";
        Intent i = new Intent(str1.concat(str2));
        i.putExtra("test", imei);

        startActivity(i);
    }

}
```

(a) 程序部分源代码

```
public class OutFlowActivity extends Activity {


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        TelephonyManager telephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
        String imei = telephonyManager.getDeviceId(); //source

        String str = "com.icc";
        Intent i = new Intent(str.concat(imei));


        startActivity(i);
    }

}
```

(b) 程序部分源代码

图 5.2 案例二部分源码

如下图 5.3 为案例三的部分源码其中 (a) 通过 StrinBuilder 类提供 append 方法将一串字符串类型拼接在另一段字符串的后边，然后将值直接传给 Intent 对象，而(b) 通过将一个变量添加到一个常量的后边，然后将这个对象赋值给 Intent 对象。

```java
public class OutFlowActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        TelephonyManager telephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
        String imei = telephonyManager.getDeviceId(); //source

        StringBuilder str=new StringBuilder("com.icc.") ;
        StringBuilder result=str.append("ACTION");
        String newstr=result.toString();

        Intent i = new Intent(newstr);

        i.putExtra("text",imei);

        startActivity(i);
    }

}
```

(a) 程序部分源代码

```java
public class OutFlowActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        TelephonyManager telephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
        String imei = telephonyManager.getDeviceId(); //source

        StringBuilder str=new StringBuilder("com.icc.");
        StringBuilder result=str.append(imei);
        String newstr=result.toString();
        Intent i = new Intent(newstr);


        startActivity(i);
    }

}
```

(b) 程序部分源代码

图 5.3 案例三部分代码

通过以上案例进行测试分析，发现 IC3 无法检测到一些使用 Java API 中 String 的一些处理字符串的方法，而进一步分析得出，实际上 IC3 无法检测到在程序运行时传入的未知参数以及 java 中部分函数，对于无法检测到的结果，我们则认为这个 Intent 有可能与应用程序的任何组件的 Intent 过滤器相匹配。

表 5.1　IC3 工具检测结果统计

String 类	字符串相加	可以检测到

	字符串加实时获取变量	(.*)
	str1+str2	可以检测到
	substring()	(.*)
	concat()	(.*)
	valueOf()	(.*)
	toLowerCase()	(.*).ACTION
	replace()	(.*)
StringBuilder	append()	可以检测到
	replace()	(.*)
	delete()	(.*)
	insert()	(.*)
join	if	分叉路结果都输出
	try/catch	可以检测到
	Linklist	(.*)

而通过相同的案例对比，dialdroid 的提取精度基本相同，对于实时的参数或者调用一些 java api 依旧无法识别。最终我们采用 IC3 作为 Intent 对象字符串提取工具。

5.3.2 构建 ICCG 的效率

我们通过组件间的 Intent 值来进行构建 ICCG，而对于 ICCG 具体的数据如何，我们通过对 8 个 dowgin 恶意家族的恶意软件分析识别，他们的情况如下所示。首先我们统计了所有组件的 link 数目，如下表 5.2 所示。

行距不合适

表 5.2 组件间 links 相关信息

编号	APK 名称	大小	Intent 数目	links 数目
1	4b3b5b7d8d36301ac099b1de94f97c11.apk	10.0 MB	27	11
2	37b993b5f59bbed7538265885429e4c4.apk	3.77 MB	34	6
3	4b3206b49960db5937ff9ab83eade925.apk	16.2 MB	13	12
4	41172f215cdf0086a58d6fd023b3bc3b.apk	616 KB	10	10
5	5b47d801ad308fc5a766970fdbd713fe.apk	5.33 MB	6	5

6	1d15765ffee294f27da4865356a994bd.apk	4.96 MB	30	15
7	3e30f2644a2e9f1b81f7f5a810e5f6ce.apk	1.92 MB	26	8
8	5c12d7911a2544a7edfe69ee9bbc9b89.apk	6.62 MB	26	17

然后生成 ICCG 后，对图中的节点和边进行统计，具体数据如下表 5.3 所示。

表 5.3　ICCG 图中相关信息

APK 名称	节点个数	边
4b3b5b7d8d36301ac099b1de94f97c11.apk	12	11
37b993b5f59bbed7538265885429e4c4.apk	12	6
4b3206b49960db5937ff9ab83eade925.apk	11	12
41172f215cdf0086a58d6fd023b3bc3b.apk	11	10
5b47d801ad308fc5a766970fdbd713fe.apk	6	5
1d15765ffee294f27da4865356a994bd.apk	18	15
3e30f2644a2e9f1b81f7f5a810e5f6ce.apk	8	8
5c12d7911a2544a7edfe69ee9bbc9b89.apk	17	17

5.3.3 组件内污点分析

我们通过对 Google Play 中随机选取下载量较高的 15 个 APP 上比较了我们的工具的性能和原始 FlowDroid 的性能。表 5.4 给采取得应用程序情况，包括他们每个应用程序的大小和类别以及通过我们的工具对他们检测的一个结果。

表 5.4 检测结果　行距不合适

ID	名称	大小	类别	组件个数	回调个数	总时间	泄露个数
1	AmazonShopping.apk	49.4 MB	购物类	171	31	11m5s	15
2	Carrefour.apk	47.4 MB	购物类	53	220	11m19s	9
3	Coursera.apk	21.7MB	学习类	121	49	20s	5

4	BBCLearningEnglish.apk	6.91 MB	学习类	44	141	11m15s	3
5	TED.apk	18.4 MB	学习类	42	21	5s	3
6	JustDating.apk	32.6 MB	社交类	119	630	4m58s	24
7	iPair.apk	76.9 MB	社交类	187	34	20s	7
8	Seeking.com.apk	62.1 MB	社交类	111	423	26s	1
9	Telegram.apk	39.9 MB	社交类	55	101	1h36m48s	14
10	McDonald.apk	34.4 MB	饮食类	139	295	34m1s	23
11	Coinbase.apk	32.4 MB	金融类	28	62	4m31s	3
12	GooglePay.apk	16.5 MB	金融类	10	21	9m31s	2
13	gongshangbank.apk	119 MB	银行类	354	3	2s	0
14	jianshebank.apk	142 MB	银行类	1780	8	6s	0
15	zhongguoyinhang.apk	92.1 MB	银行类	263	78	31s	12

通过结果我们发现了无论是组件数还是回调个数在优化的基础上基本保持不变或者有所增加，而总的检测时间有所减少。下图 6.4 展现出了这两种工具在精度和运行时间以及内存消耗的比较。实验结果可以看到，通过优化后的 FlowDroid 对检测时间和所需要的内存有一定的效果，通过对组件的搜索模式进行改进从而提升了一些效率。

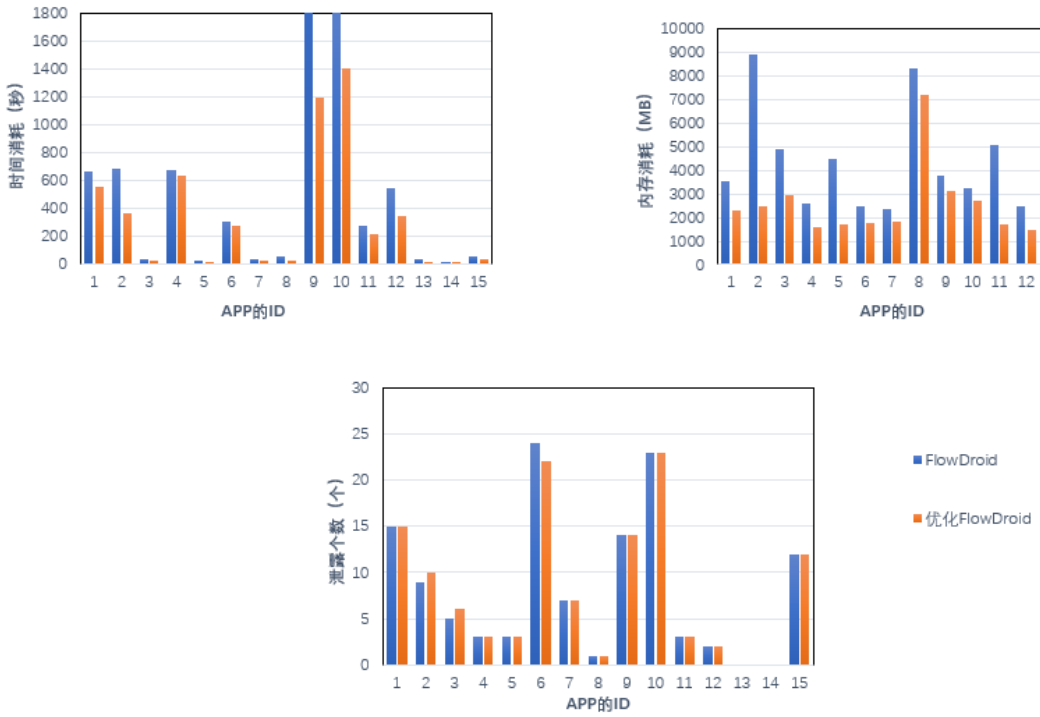

图 5.4 FlowDroid 优化前后结果对比

5.3.4 组件间污点分析

通过对 5.3.2 的部分恶意家族 app 进行组件间的测试，具体结果如下表 5.5 所示。其中 Intent 表示传播对象 Intent 的个数，Intent 过滤器表示该 APP 里包含的所有 Intent-Filter 的个数，消耗时间是整个污点分析过程中该应用程序消耗的时间。

表 5.5 组件间污点分析结果

名称	大小	Intent	Intent 过滤器	消耗时间	消耗内存	泄露个数
1c4e357a8ec5f13de4ffd57cc2711afe.apk	3.55 MB	415	55	1 分 29 秒	139 MB	40
37b993b5f59bbed7538265885429e4c4.apk	3.7 MB	34	37	5 秒	50 MB	12
1d15765ffee294f27da4865356a994bd.apk	4.96 MB	30	24	5 秒	48 MB	4
5b47d801ad308fc5a766970fdbd713fe.apk	5.33 MB	6	41	38 秒	108 MB	27
3e30f2644a2e9f1b81f7f5a810e5f6ce.apk	1.92 MB	26	33	7 秒	50 MB	10
03e8e080616cc50e06ddfea24aed0623.apk	2.86 MB	191	12	3 秒	42 MB	3

为了进一步比较分析，我们进行了文献分析，最后选择了 Amandroid[20]和 DroidSafe[21]，因为这两个工具是最受欢迎的关于组件间静态数据流分析方法，用来检测数据泄漏和高精度的应用程序，我们使用了 Amadroid 的版本是 3.1.2， DroidSafe 的版本是 2016 年 6 月 22 的版本。由于 DroidBench 这个基准测试中的数据泄漏是众所周知的，这些应用程序被用作基本事实，允许作为性能指标，例如精度，召回率和 F 分数。精度（Precision）是实际数据流与总数之比，由工具和可计算为(TP/(TP+FP))，其中 TP 和 FP 分别是真阳性和假阳性。召回率(recall)是指正确报告给工具的数据流之比。通过预期数据流的总数，可以计算为(TP/(TP+FN))，其中 FN 是假阴性。F 分数(F-score)用于衡量测试的准确性。它是精确度和召回率的加权平均值，可以计算为(2*精度*召回/(精度+召回))。它在 1 处达到最佳值，这意味着完美的精确度和召回率。通过对基准应用程序将本文方法与他们相比，结果如下表 5.6 所示。

表 5.6　基准测试应用测试结果

	DroidSafe	**Amandroid**	**Our proach**
TP	107	71	124
FP	13	16	2
FN	18	56	4
Precision (%)	89.2	81.6	98.4
Recall (%)	85.6	55.9	96.9
F-score	0.87	0.66	0.98

6 总结与展望

6.1 工作总结

智能手机和平板电脑等移动设备无处不在。这些设备上的应用程序环境实现了一个市场模型，在这个模型中，应用程序开发人员发布应用程序，用户可以方便地从应用程序商店下载和安装这些应用程序。这些应用程序可能会访问各种敏感信息，如用户的位置、联系人和手机的唯一标识符(IMEI)。应用程序，如关联网络和银行应用程序，可以额外收集大量敏感数据。这种设置中很重要的一点是过滤敏感数据，这可能会侵犯用户的隐私，并允许不希望的跟踪用户的行为。儿 FlowDroid 作为安卓恶意程序静态检测方法中最先进的工具，它建立在 Soot 的基础上，并且精确地模拟了安卓的生命周期，并通过用户界面对象的回调来处理数据传播。本文基于 Flowdroid 研究安卓恶意程序静态检测方法，主要工作有以下几点：

（1）研究分析安卓系统放的整体框架以及安卓系统中的安全机制基础理论，指出当前 Android 应用安全研究领域静态检测和动态检测方法存在的优点与不足，阐述本文的研究背景和研究意义。

（2）详细研究安卓静态检测工具 FlowDroid 的工作原理以及源码分析，分析其框架、检测流程、污点追踪算法等关键技术。使用 FlowDroid 工具对来自谷歌官方应用商店的常用应用进行了测试分析，测试应用中数据流和控制流以判定泄露路径。从理论和实验数据两方面表明，大量的冗余检测路径带来巨大的时间和内存消耗，而假阳性的检测路径严重影响检测准确性，并且对于组件之间产生的信息流无法追踪到。

（3）针对以上问题，本文提出了相应的静态检测改进方案。

设计并实现了基于 flowdroid 的静态检测方法，以部分组件的检测模式，减少不必要的检测时间和内存损耗；设计并实现了部分组件搜索计算算法，通过每次只搜索部分组件的回调方法动态组件等来减少内存的消耗；设计了 ICC 链接算法，通过 IC3 工具提取出相应的组件间通信值然后计算组件之间的链接生成 ICC 链接；并在 ICCG 的基础上进行数据流分析。

6.2 未来期望

在本课题设计了一种基于 FlowDroid 的安卓恶意程序静态检测方法，通过对安卓应用程序单个组件内和组件间进行污点分析判断是否有隐私信息泄露。但本文方法仍存在

一些不足之处，将在后续的工作中不断进行完善，未来的工作计划安排如下：

(1)虽然检测到组件间的通信值，但是工具还是不够完美，分析结果准确率不够高，分析时间也比较长，有待提升。

(2)FlowDroid 污点分析阶段消耗过大，计算量暴增，有很大的优化空间。

致 谢

时光如梭，如歌。转眼间，三年的研究生求学生活即将结束，站在毕业的门槛上，回首往昔，奋斗和辛劳成为丝丝的记忆，甜美与欢笑也都尘埃落定。交通大学以其优良的学习风气、严谨的科研氛围教我求学，以其博大包容的情怀胸襟、浪漫充实的校园生活育我成人。值此毕业论文完成之际，我谨向所有关心、爱护、帮助我的人们表示最诚挚的感谢与最美好的祝愿。

首先，我衷心的感谢敬爱的老师刘晓建老师，感谢老师对我的指导，栽培，给我认识 Android 软件安全世界的机会。刘老师为人正直踏实，细心仔细，要求严格，文质彬彬，满腹经纶。从看论文、书籍学习，到自己做实验验证，期间出现了很多想不到的问题，是刘老师悉心地指导，帮我指明方向。感谢老师在我停步不前时一直开导我。感谢刘老师在日常琐碎的生活中教会我踏实认真的道理。

其次，非常感谢郭泽敏师妹、南姿师妹。感谢大家在 Android 逆向学习和安卓恶意应用程序分析方面的探讨学习。感谢师妹们对我的支持和鼓励，因为有他们，我想成为更好地自己，想要不断的学习进步。感谢实验室的同学，让我有一个良好的学习氛围，感谢他们的持之以恒，让我逐渐进步。感谢大家的互相帮助和互相鼓励，让过去的每天有所收获，未来的每一天有所期待，今天有所坚持。感谢西安科技大学计算机学院 2017 级全班同学陪我度过这段有苦有甜有意义的时光。

感谢家人、亲人，感谢他们总是站在我身边，总是默默爱护我，给予我最大的关怀及感动。感谢他们无条件的理解和不问理由的支持。感谢他们一直将最好的一切给我，用汗水和皱纹换我无忧无虑的生活，因为他们，我才是我。

最后，感谢各位审阅本论文的老师。感谢出席本次论文答辩的老师。感谢你们对我的指导

参考文献

[1] “Number of smartphones sold to end users worldwide from 2007 to 2016 (in Million units).” https://www.statista.com/statistics/263437/ global-smartphone-sales-to-end-users-since-2007/, 2017.[Accessed on 03/20/2017].

[2] A. Drozhzhin, “Taxi Trojans are on the way.” https://www.kaspersky.com/blog/faketoken-trojan-taxi/18002/,2017. [Accessed on 05/26/2018].

[3] A. Malhotra and P. P. Singh, “Android Malware: Study and Analysis for Privacy Leak in Ad-Hoc Network,” International Journal of Computer Science and Network Security, vol. 12, no. 3, pp. 39–43, 2013.

[4] Y. Song and U. Hengartner, “PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices,” in Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices,pp. 15–26, ACM, 2015.

[5] M. Pistoia, O. Tripp, P. Centonze, and J. W. Ligman, “Labyrinth: Visually Configurable Data-Leakage Detection in Mobile Applications,” in 2015 16th IEEE International Conference on Mobile Data Management, vol. 1, pp. 279–286, IEEE,2015.

[6] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, “Effective Real-Time Android Application Auditing,” in 2015 IEEE Symposium on Security and Privacy,pp. 899–914, IEEE, 2015.

[7] “Mobile Security Report.”https://info.nowsecure.com/rs/201-XEW-873/images/ 2016-NowSecure-mobile-security-report.pdf, 2016. [Accessed on11/16/2017].

[8] “ McAfee Labs Threats Report: December 2016. ” https://www.mcafee.com/us/resources/reports/ rp-quarterly-threats-dec-2016.pdf, 2016. [Accessed on 11/16/2017].

[9] “ Internet Security Threat Report. ” https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf[Accessed on 10/17/2018].

[10] Z. Yuan, Y. Lu, and Y. Xue, “Droiddetector: android malware characterization and detection using deep learning,” Tsinghua Science and Technology, vol. 21, no. 1,pp. 114–123, 2016.

[11]W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and

A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," ACM Transactions on Computer Systems, vol. 32, no. 2, pp. 5:1–5:29, 2014.

[12]T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, pp. 785–794, ACM, 2016.

[13]A. Sadeghi, H. Bagheri, J. Garcia, et al., "A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software," IEEE Transactions on Software Engineering, vol. 43, no. 6, pp. 492–530, 2017.

[14] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, " Scandroid: Automated security certifification of android," tech. rep., 2009. [Accessed on 11/16/2017].

[15] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specifification," in Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 217–228, ACM, 2012.

[16] T. Vidas, N. Christin, and L. Cranor, "Curbing Android Permission Creep," in Proceedings of the Web 2.0 Security & Privacy 2011, vol. 2, pp. 1–5, 2011.

[17] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection," in Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 1043–1054, ACM, 2013.

[18] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effffective Real-Time Android Application Auditing," in 2015 IEEE Symposium on Security and Privacy, pp. 899–914, IEEE, 2015.

[19] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale," in Proceedings of the Trust and Trustworthy Computing: 5th International Conference, pp. 291–307, Springer Berlin Heidelberg, 2012.

[20] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," Acm Sigplan Notices, vol. 49, no. 6, pp. 259–269, 2014.

[21] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek, "Automated Dynamic Enforcement of Synthesized Security Policies in Android," 2015.

[22] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden, "DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android," in 2014 Ninth International Conference on

Availability, Reliability and Security, pp. 40–49, IEEE, 2014.

[23] A. Sadeghi, H. Bagheri, J. Garcia, et al., "A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software," IEEE Transactions on Software Engineering, vol. 43, no. 6, pp. 492–530, 2017.

[24] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effffective and Explainable Detection of Android Malware in Your Pocket," in Proceedings of the 21st Annual Network and Distributed System Security Symposium, The Internet Society, 2014.

[25] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer, "ANDRUBIS - 1,000,000 Apps Later: A View on Current Android Malware Behaviors," in 2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, pp. 3–17, IEEE, 2014.

[26] A. Gianazza, F. Maggi, A. Fattori, L. Cavallaro, and S. Zanero, "PuppetDroid: A User-Centric UI Exerciser for Automatic Dynamic Analysis of Similar Android Applications," eprint arXiv:1402.4826, 2014.

[27]McClurg J, Friedman J, Ng W. Android privacy leak detection via dynamic taint analysis. Electric. Eng. Comput. Sci. 2013;450.

[28] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," Tsinghua Science and Technology, vol. 21, no. 1, pp. 114–123, 2016.

[29]"DroidBox: An Android Application Sandbox for Dynamic Analysis." https://www.honeynet.org/gsoc2011/slot5, 2011. [Accessed on 03/20/2017].

[30] Weichselbaum L, Neugschwandtner M, Lindorfer M, Fratantonio Y, van der Veen V, Platzer C. Andrubis. Andrubis, 1. Vienna University of Technology; 2014. Tech. Rep. TRISECLAB-0414.

[31]Google: Permissions overview | Android Developers. https://developer.android.com/guide/topics/permissions/overview#normal_permissions.

[32]Google: Request App Permissions | Android Developers. https://developer.android.com/training/permissions/requesting.

[33]J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski, Android Hacker's Handbook. Wiley Publishing, 1st ed., 2014.

[34]https://developer.android.com/guide/platform/index.html, 2016. [Accessed on 11/16/2017].

[35] K. Yaghmour, Embedded Android: Porting, Extending, and Customizing. O'Reilly Media, Inc., 1st ed., 2013.

[36] A. Hoog, Android Forensics: Investigation, Analysis and Mobile Security for Google Android. Syngress Publishing, 1st ed., 2011.

[37] “Derivation: Error Backpropagation & Gradient Descent for Neural Networks.” https://goo.gl/RSwfcT. [Accessed on 03/25/2017].

[38] D. Goosin, “Active Drive by Exploits Critical Android Bugs, Care of Hacking Team.” https://goo.gl/5zuGGX, 2016. [Accessed on 03/20/2017].

[39] S. Khandelwal, “300000 Android Devices infected by Premium SMS-Sending Malware.” https://goo.gl/k9rNGx, 2014. [Accessed on 03/10/2017].

附 录

硕士期间发表的论文:

[1] X Liu, X Du, Q Lei , et al. Multifamily Classification of Android Malware With a Fuzzy Strategy to Resist Polymorphic Familial Variants[J]. IEEE Access, 2020, PP(99):1-1.