分类号： TP309.1 密级：

ＵＤＣ： 编号： 10486

武 汉 大 学

硕士学位论文

基于控制流的Android应用安全分析

研 究 生 姓 名：王庆飞

学 号：2012202110027

指导教师姓名、职称：严飞 副教授

学 科、专 业 名 称：

研 究 方 向：

二〇一五年五月

--

By

Wang Qingfei

Supervised by

Dr. Yan fei

Wuhan University

May 2015

郑 重 声 明

本人的学位论文是在导师指导下独立撰写并完成的，学位论文没有剽窃、抄袭、造假等违反学术道德、学术规范和侵权行为，否则，本人愿意承担由此而产生的法律责任和法律后果，特此郑重声明。

学位论文作者：

年 月 日

# 摘要

由于Android应用的不断发展，它的多样性为人们的生活带来许多便利，但在享受便利的同时用户也承担着隐私信息泄露的风险。如何保护Android隐私数据成为研究者和安全维护者关注的焦点。对于Android平台而言，存在许多保护Android隐私数据的工具，这些隐私数据的保护工具中，TaintDroid备受关注。TaintDroid扩充寄存器为隐私数据附加了污点标记信息，在污点数据传播的过程中，TaintDroid使用污点传播语义对带有污点值的数据进行追踪，通过对sink的写日志操作来记录隐私数据的泄露行为。TaintDroid虽然在一定程度上保护了隐私数据的安全，但也存在着一定的不足：它无法追踪由控制中流隐式传播引起的隐私信息泄露。

本文以TaintDroid动态污点监控系统为例，阐述了控制流对隐私数据的威胁，并且针对TaintDroid引入了几种攻击模型，分析了TaintDroid在控制流污点标记上缺陷。为了能直观表示出控制流引起污点不足问题，本文使用信息流模型中的信息流传播理论表示了污点信息的传播，针对控制流中污点不足问题给出了污点标记不足模型。而控制语句会有多个分支，在动态执行时分为执行分支和未执行分支两种情况，根据控制语句的分支执行情况提出污点标记策略并对污点标记策略可行性进行证明。对于控制流的分析，本文引入了控制流图以及对应的支配树等相关概念，然后对支配关系和控制依赖关系进行分析，并给出支配关系求解的算法以及控制依赖关系算法，对算法复杂度也分别出给了详述；把污点标记策略和控制依赖算法相结合，本文提出污点标记算法。基于静态分析和动态分析的技术分析，本文在TaintDroid基础上设计了隐式控制流监控模块，用于对程序中的隐私数据进行监控。最后利用仿真实验对隐式控制流监控模块进行分析，结果表明该模块对控制流引起的隐式信息传播具有监控的功能。

关键字 ：Android，TaintDroid，静态分析，动态分析，控制依赖

# Abstract

目录

[基于控制流的Android应用安全分析 I](#_Toc418454653)

[摘要 I](#_Toc418454654)

[Abstract II](#_Toc418454655)

[1绪论 1](#_Toc418454656)

[1.1 研究背景和意义 1](#_Toc418454657)

[1.2 本文主要工作 2](#_Toc418454658)

[1.3 本文结构 3](#_Toc418454659)

[2 Android安全分析以及国内外研究进展 4](#_Toc418454660)

[2.1 Android系统框架描述 4](#_Toc418454661)

[2.1.1应用层 4](#_Toc418454662)

[2.1.2系统架构层 4](#_Toc418454663)

[2.1.3内核层 5](#_Toc418454664)

[2.2 Android应用文件解析 6](#_Toc418454665)

[2.3 Android安全机制 7](#_Toc418454666)

[2.3.1 Android系统安全保护 8](#_Toc418454667)

[2.3.2 Android 应用中安全保护 8](#_Toc418454668)

[2.4 Android中存在问题及相关技术分析 9](#_Toc418454669)

[2.4.1 Android信息泄露 9](#_Toc418454670)

[2.4.2 Android安全研究国内外研究状况 10](#_Toc418454671)

[2.4.3 Android静态安全分析 11](#_Toc418454672)

[2.4.4 Android动态安全分析 13](#_Toc418454673)

[2.5 本章小结 15](#_Toc418454674)

[3 TaintDroid分析 16](#_Toc418454675)

[3.1 TaintDroid设计概述 16](#_Toc418454676)

[3.2 TaintDroid的实现 17](#_Toc418454677)

[3.2.1 TaintDroid污点标记存储 17](#_Toc418454678)

[3.2.2 TaintDroid解释型代码污点传播 19](#_Toc418454679)

[3.2.3 TaintDroid程序间通信污点传播 21](#_Toc418454680)

[3.2.4 存储污点传播 21](#_Toc418454681)

[3.3 TaintDroid设计的不足 21](#_Toc418454682)

[3.4本章小结 23](#_Toc418454683)

[4 TaintDroid污点标记不足模型和解决方案设计 24](#_Toc418454684)

[4.1 控制流技术的相关研究 24](#_Toc418454685)

[4.2 由程序控制流引起的TaintDroid攻击模型 25](#_Toc418454686)

[4.3 Under-tainting的形式化描述以及模型 27](#_Toc418454687)

[4.4 Under-tainting问题的解决方案设计 30](#_Toc418454688)

[4.5 本章小结 32](#_Toc418454689)

[5 控制流中的支配关系和控制依赖算法 33](#_Toc418454690)

[5.1 控制流图介绍 33](#_Toc418454691)

[5.2 控制流图中的支配关系 35](#_Toc418454692)

[5.3 控制依赖关系算法设计 40](#_Toc418454693)

[5.4 污点标记算法设计 44](#_Toc418454694)

[5.5 本章小结 44](#_Toc418454695)

[6 系统设计和实验数据 45](#_Toc418454696)

[6.1 TaintDroid控制流监控模块设计 45](#_Toc418454697)

[6.2 CFMM模块实现思路 47](#_Toc418454698)

[6.3 验证分析 50](#_Toc418454699)

[6.4 本章小结 56](#_Toc418454700)

[7 结束语 57](#_Toc418454701)

[8 参考文献 58](#_Toc418454702)

[9 致谢 62](#_Toc418454703)

[11 读研期间研究成果 64](#_Toc418454704)

## 

# 第1章 绪论

## 1.1 研究背景和意义

自从Google在08年推出第一款Android手机后，Android系统和相应产品已经历经七年多的发展历程。据2014年Google I/O大会的数据显示[1]：目前全球Android手机的用户数量已经超过10亿；在全球平板电脑出货量中，使用Android系统的占62%；在国内，Android智能机的用户也超过七成份额；另外，在Android程序的应用市场上，应用的种类和数量都如雨后春笋般的增长，在Google的I/O大会上报告显示：Android的应用安装量以每年236%的速度递增。2014年，市场研究公司App Annie发布一份有关Android应用的使用报告[2]，报告指出Google Play的应用数量已经超过150万款，仅仅在2013年7月至2014年六月这一年内，Google Play的应用总量就增长了近60%；并且在这一年里，应用的下载量是去年的1.5倍。从以上的数据表明，Android的设备使用和应用使用的发展劲头将会非常强劲，尤其在手机价格下降之后，使Android智能机迅速占领原来低端机的市场；随着Android手机使用量的激增，Android应用的爆发式增长在随着Android设备的火爆而继续延续。加上Android系统的开源，使得更多公司、开发者加入到Android开发这一行列，这无疑将更加丰富Android App的种类和功能，为用户带来更多的视觉体验，也加快了Android系统的发展步伐。

在Android 应用不断高速前进的同时， Android应用安全问题也不断浮现在用户眼前，严重阻碍了Android的发展。2011年三月，黑客向多种Android应用中注入”DroidDream”恶意代码，然后把包含恶意代码的程序上传到Google Play。这场攻击最终导致超一万多用户下载并安装了带有恶意代码的应用，给用户带来的损失也是不可估量的。“DroidDream”事件之后，Android应用中恶意代码问题开始引起Google公司的注意；Google开始扫描Google Play中应用，并清除包含“DroidDream”的应用[3]。尽管Google在一定程度上减轻了用户损失，但“DroidDream”恶意代码的变种不断繁衍，在一定程度上威胁了Android的发展。据2014年网秦手机安全报告显示，仅仅在2014年上半年就查出8万多款恶意手机软件，同比增幅度超过60%；受影响的手机超过三千万部，同比增长78.6%；受感染平台96%来自于Android平台。从影响的地区上看，中国大陆地区以18.2%感染比例高居全球第一，远远大于印度的14.2%和沙特9.6%、印度尼的8.2%和美国的7.7%的感染率。从恶意软件类型上看，恶意获取用户隐私信息的应用居多；而在一些非恶意的应用中，也存在一些泄露用户隐私数据的行为，然后从中谋取暴利。由于Android系统的使用不断增加，Android系统及其应用带来的风险也在不断加剧，而且威胁范围也在不断扩大。因此，Android隐私数据安全保护势在必行。

目前，虽然国内外对移动智能终端中隐私信息的安全研究取得一定的进展，但是很多传统的安全分析手段并不适用Android手机。因此，如何确保Android系统中隐私信息的安全已成为一个亟待解决的问题。引入新的研究方法和高效的设计思路，会对Android系统的隐私信息保护研究带来重大影响。

## 1.2 本文主要工作

本文在深入分析了Android的系统框架之后，研究了Android系统和应用的安全防护技术和防护平台，从这些防护平台和技术中又总结了两种技术方法，分别为静态分析和动态分析，并通过阅读大量文献对Android中使用静态和动态分析的技术展开了细致的研究，并对使用这些方法的Android平台保护工具也进行了总结。对于Android系统中隐私数据的防护系统，本文对TaintDroid做了细致分析，并总结出了TaintDroid中存了两大问题，一个是本地方法调用可导致TaintDroid无法检测污点信息泄露；另一个是由控制流引起的TaintDroid无法检测污点信息泄漏，本文都给出了细致的分析。对于在控制分支中的变量是否控制依赖于条件语句，本文主要利用支配关系集和控制依赖算法进行了分析。最后本文把控制依赖和污点传播策略相结合，为TaintDroid设计一个新的监控模块，完成了Android平台下隐私数据污点标记不足问题保护体系设计，最后针对该模块进行仿真实验，实验结果表明该模块对控制流引起的隐式信息传播具有监控的功能。对于本文的创新和贡献，可以归纳为以下几点：

1. 文章先对Android当前的发展现状进行了详细调查，分析了Android保护的重要性和意义，同时对支撑Android平台的防护技术做了研究，并对目前Android的主流防护系统TaintDroid进行了剖析，然后对其中存在的问题也进行了总结。
2. 针对由隐式控制流导致的TaintDroid无法检测隐私信息泄露问题，本文建立了污点标记不足模型；针对该模型，本文又根据控制语句存在分支执行情况进行分析，对于执行分支中的变量和和未执行执行分支中的变量，本文给出了两个污点传播策略，使得两种分支下的变量都能够被污点标记。
3. 本文还对程序的控制流程图以及控制语句中的支配关系进行了深入研究和分析。对于控制流图中的结点间支配关系，本文则把控制流图转化成为了支配树。对支配树中的支配关系、严格支配关系、直接支配关系和后继支配关系等进行细致研究并给出了相应的求解算法，对于求解算法的时间复杂度，本文也给出了分析。对控制依赖关系集求解算法，本文则把通过定义给出算法进行了改进，并对改进后的算法的复杂度进行了分析。
4. 结合控制流中污点传播策略和控制依赖算法，本文提出了控制流污点标记算法；对于污点标记的算法时间复杂度分析，本文也给出了详细的分析。通过污点标记算法，可以对由控制流引起的TaintDroid隐私泄露问题进行有效解决，从而加强了Android系统的防护。

## 1.3 本文结构

第一章 绪论，描述研究背景和研究意义，列举全文主要工作以及全文结构。

第二章 Android安全分析以及国内外研究现状，深入研究了Android系统框架和Android中的安全机制，对Android中存在的安全问题以目前的技术保护手段进行细致分析，并详细总结出静态分析和动态分析以及这两种技术在Android安全分析中的作用以及优缺点，并列举当前使用两种技术的Android保护工具。

第三章 TaintDroid系统分析，对TaintDroid设计以及实现进行深入研究，详细分析TaintDroid系统在数据保护时主要存在的问题。

第四章 建立TaintDroid污点标记不足的模型并提出解决方案，根据控制流引起的TaintDroid中污点标记不足的原因，结合信息流知识建立污点不足模型，针对该模型提出相应污点传播策略，并对传播策略的正确性给出证明。

第五章 控制语句依赖关系计算，对的控制语句依赖关系进行分析，并给出相应的控制依赖算法，分析算法的时间复杂度；结合第四章的污点传播策略提出污点标记算法，对算法的时间复杂度进行阐述。

第六章 控制流监控模块的设计和仿真实验，对TaintDroid系统进行改进，结合静态分析和动态分析为TaintDroid添加监控控制流污点标记模块，对设计思路及其功能进行进行介绍，并利用仿真实验验证该模块的功能。

第七章 总结，概述本文研究工作，展望未来研究方向。

# 第2章 Android安全分析以及国内外研究进展

## 2.1 Android系统框架描述

按照Android层次划分，自上而下的4层可分为：应用层、应用框架层、系统库层、linux内核层。系统框架图如2.1所示[4]，本文将把应用框架层和系统库层合并系统架构层，然后对各层的关键技术进行简要描述。

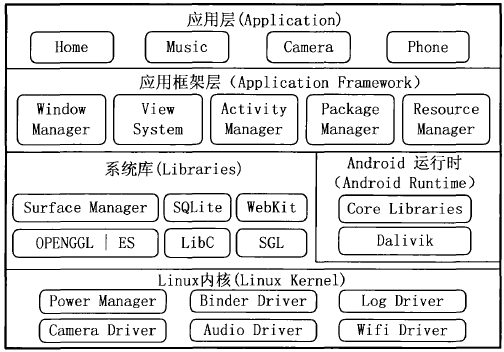


图2.1 Android系统框架图

### 2.1.1应用层

应用层主要是Android系统自带的应用程序[5]：主桌面(Home)、短息(SMS)、浏览器、邮件(Email)、联系人以及google map等；同样也包括开发者为Android操作系统设计开发的应用程序，如游戏、音乐播放器、视频播放器等等。这些程序是通过Java语言编写，通过调用系统框架提供的API完成。目前，Android支持使用java native interface(JNI )方式编写代码，允许Java代码和C/C++代码交互使用，这样不但可以提高程序的性能，同样也使得Java代码平台移植性得到提升。

### 2.1.2系统架构层

系统架构层包含了Android系统的应用框架层、应用的运行环境和系统库层。

应用框架层包含了核心API框架，它能给上层应用提供多种服务和管理工具，其中比较重要的有应用层消息传递、应用包管理、界面管理以及数据访问、电话管理等。

Android应用的运行环境一般是指Dalvik虚拟机。相比java虚拟机，Dalvik是一种轻量级的虚拟机，但与Java虚拟机不同的是，它是基于寄存器的虚拟机，而Java虚拟机是基于栈的虚拟机。因此，Dalvik执行dex字节码而非Java字节码。该格式执行文件是针对Java字节码做了优化，专门应对内存不足的问题，因此Dalvik虚拟机被大量应用在移动设备上。而对于每一个运行的应用程序而言，为了保证程序隔离性，系统都会为它们创建一个单独的虚拟机实例。Dalvik可以同时高效运行多个虚拟机实例，大大提升程序的并行性，也充分利用了系统的资源。

系统库主要是一些由C\C++实现的类库，不仅给系统组件提供服务，也给Android开发者提供接开发口。其中主要的系统库包括：

Bionic：属于 Android的C/C++库。Android没有采用glibc作为C库，而是专门为Android改良了传统的Libc。Bionic是一种轻量C库，能使Android运行带有本地代码的时候更加高效。

浏览器插件:基于Webkit，Android设计了一套供上层网络应用使用浏览器插件。

多媒体系统库：一般是指OpenCore的音频解码库，OpenCore主要包含了两大模块:PVPlayer和PVAuthor，他们分别对应框架层的媒体播放和媒体录制功能。PVPlayer主要提供多媒体播放器的功能，完成不同格式的音频、视频的回放；PVAuthor具有媒体流记录功能，捕获不同格式音频、视频。PVPlayer和PVAuthor通常以软件开发包的形式给开发者开发使用，开发者可以根据这个开发包构建多种应用程序和服务。

数据库插件：Android数据库插件一般是指SQLite。SQLite为轻量级数据库，是一种专门针对嵌入式设备的数据库，具有资源占用低、运行速度快等优点。

### 2.1.3内核层

Android系统内核是基于开源系统linux内核开发，因此它继承了linux内核大多数重要的功能模块。内核层是核心层，为上层提供许多服务，除了沿袭linux内核中多数功能，Android还增加了其他功能模块，例如摄像头驱动、显示驱动和WIFI驱动和Binder通信驱动等。至于进程间通信，Android采用的是Linux上的Binder通信机制，而Binder通信模式是基于C/S模式。在通信时，Client会先通过系统的Service Manager获取Service的代理对象，然后再通过该代理对象调用Service提供的功能接口[5]。Binder驱动会把调用请求发送给Service，同样，Binder驱动也会把Service的处理结果发送给Client。

## 2.2 Android应用文件解析

Android应用程序安装程序是以.apk结尾的压缩文件。在apk的内部包含res和META-INF两个文件夹以及resource.arsc、class.dex和AndroidManifest.xml三个文件，如图2.2所示。

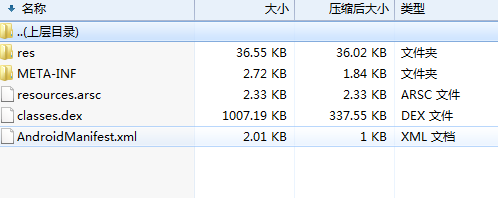


图2.2 apk文件结构示意图

res文件夹下主要存放menu、layout和drawable-\*文件夹，包括了Android所有的资源，像图片资源、颜色资源、尺寸资源等。其中menu中主要定义了应用程序的菜单；layout主要了定义用户界面及其控件的布局，例如控件坐标方向，填充等。对于drawable-\*的文件夹分为三类，-hdpi，- mdpi和-ldpi，而这些文件夹对应的存放图片分辨率分别为高、中、低。因为Android应用上需要使用不同分辨率的图片，因此在开发时针对不同分辨率的图片需要分类保存。

MANIFEST.MF、CERT.SF、CERT.RSA 是META-INF的文件夹下的三个子文件；其中MANIFEST.MF中保存了Android对所有其他文件进行哈希算法后的值，该值被base64编码；而CERT.SF则是Android对所有其他文件前三行进行哈希算法后的值；CERT.RSA存放的是对apk包的签名信息以及公钥等信息。从对META-INF的文件夹下的文件分析不难发现，这些文件是用来保证apk的完整性和和系统安全。一般而在apk安装时，Android的安全管理器会对apk里文件进行校验，然后与META-INF下的内容做对比，一旦发现有差异，就会发出警示。

resources.arsc为程序编译生成的二进制文件，它相当于相当于一个资源索引文件，描述应用中资源ID对应的资源文件。

AndroidManifest.xml是整个Android应用主配置文件，主要包括包括以下信息：

1. 应用名，应用名是应用的唯一标识；
2. 应用中组件信息，例如Service，ContentProvider以及BroadcastRecevier
3. 应用的版本号
4. 应用使用系统的权限声明
5. 访问该程序的其他程序需要的权限声明

classes.dex文件是应用运行在Dalvik虚拟机上的可执行文件。虽然Android应用主要由java代码编写，但Dalvik虚拟机只能运行.dex文件。因此，在程序编译阶段需要借助dx工具把java字节码进行转换，并且把所有的.class文件进行优化重组成一个Dalvik可识别的文件(classes.dex)。转化成.dex文件后，执行文件就会变得更小而且效率更高。

## 2.3 Android安全机制

Android系统是基于Linux开发，它继承了许多Linux安全机制。除了linux上特有的安全机制外，Android系统本身也提供了沙箱、应用程序权限、应用签名、进程间安全通信等安全机制。表2.1所示 Android中安全机制对应要解决的安全问题。宏观上，Android安全安全保护分为：系统保护和应用保护。

表2.1 Android安全机制

|  |  |  |
| --- | --- | --- |
| 安全机制 | 安全机制描述 | 所要解决Android问题 |
| 应用沙箱 | 每一个应用程序都有一个独立用户ID | 防止应用之间相互干扰 |
| 文件系统 | 每一个应用程序都有一个独立的文件目录 | 防止程序文件被其他程序访问 |
| 系统分区 | 把关键系统数据设置成only read模式 | 安全模式一旦启动，就只允许运行核心程序来防干扰 |
| 权限机制 | 应用程序需要在安装时说明所使用的权限 | 防止应用程序随意获取设备隐私信息 |
| 组件封装 | 应用程序中的每一组件可以控制其他程序对该组件的访问 | 防止其他应用程序的恶意干扰或者访问该程序私有API或组件 |
| 签名机制 | 在应用程序发布时，开发者必须使用证书为应用签名，而且还需要相应的验证 | 验证两个应用是否是同一源 |
| Dalvik虚拟机 | 应用程序都会运行在系统为其单独建立的虚拟机上 | 防止远程代码执行和缓冲区溢出、栈溢出 |

### 2.3.1 Android系统安全保护

Android系统安全机制主要包括两部分：应用隔离和系统资源保护。

在Android平台中，系统为应用提供沙箱确保应用程序之间的隔离。在计算机领域，沙箱机制让程序拥有一种独立的运行环境，主要应用于病毒检测和软件安全测试等。而在Android中，主要是借鉴沙箱的思想，保证应用程序之间有独立的运行空间，这样做的好处有：一旦一个应用程序出现崩溃，不影响其他程序运行，也不会导致系统崩溃；另一方面，由于程序间的隔离，应用的数据和资源也被隔离，应用就无法直接窃取其他程序的数据；使用沙箱的另一个好处就是如果一个应用被恶意攻击或远程控制，系统和其他程序也不会受到危害，大大降低了损失。沙箱是在linux内核中实现，内核层上的各层都会受到沙箱的约束。

Android的系统资源包括文件、设备、内存等。文件系统被分为系统分区

SDCard分区用户数据分区。系统区则包含了整个Android的系统框架，而SDCard分区、用户数据分区又同属用户分区。系统区被设置为只读，并且系统区可以读写用户区以及应用。Android文件具有读、写和执行三种权限，并且每一个文件具有用户ID和组ID，而Android系统则根据ID标识和权限标识管理进程对文件的操作。系统还能修改一些参数值来设置文件的全局属性，如全局可读和全局可写。为了进一步提升文件的安全性，3.0以上版本的Android系统都可以选择使用SHA256或者AES128算法对文件进行加密。

在系统设备的保护上，Android系统采用了权限机制。应用程序在安装时，必须要声明相应的权限，得到用户的许可才能申请设备的访问控制权。除此之外，Android系统还对设备使用密码验证机制。应用程序在访问设备时，需要提供相应的密码验证来获得设备使用权。这种密码验证机制可以防止未授权的应用获取对设备访问的权限。

Android系统主要采用地址分布的随机化、堆栈保护、缓冲区和内存的分配优化来保护内存。采用地址随机化机制后，就可使进程组件和对象地址进行随机分配，这对基于内存漏洞的攻击有很好防护作用；堆栈防护主要是限制应用在堆栈上运行的代码；采用缓冲区和内存的分配优化机制可以大大降低存在Android系统中的风险，例如悬浮指针、缓冲区溢出等。

### 2.3.2 Android 应用中安全保护

Android系统为多进程系统，它会为每一个进程创建一个Dalvik虚拟机。应用间的安全是通过进程级标准Linux方式实现[6]，例如对应用分配用户ID和组ID。Android通过权限机制限制应用所执行的动作和对数据的操作，使得安全措施粒度更细。

Android的应用安全主要基于沙箱机制、签名机制和权限机制[7]。应用程序运行对应的Dalvik虚拟机实例中，这就意味着应用的代码在运行时是互相隔离的。因此，一个应用不能直接获取其他应用的数据。Android系统则根据linux通信机制提供了相应进程间通信方式。在沙箱的允许的条件下，应用程序主要采用以下通信方式：

1. Binder机制：Binder是进程间的轻量级通信框架。因为Binder的轻量级并且具有系统化的实现方式和API接口，不仅简化了使用方法，也加适用于移动系统。
2. Intent机制： Android系统把一组数据和一个动作封装成一个Intent消息。根据Intent中的动作，则表明了该消息目的。函数调用和广播都是靠Intent来传递。
3. 应用组件机制：在Android应用中，组件Content Provider应用于应用间的数据共享；应用可以通过Content Provider提供的接口访问其他应用的数据。Content Provider把共享数据写入匿名共享内存中，然后利用Binder通信机制来传输数据。

开发者在发布安装程序时，需要对其进行数字签名。数字签名是应用的重要标志，它同数字证书一起完成对应用的签名。数字签名由软件开发者的私钥完成，而证书则包含开发者的签名公钥。而软件开发商想要获取证书，必须要到公认的权威机构获取，这样才能确保软件来源的可靠程度。如果两个安装程序使用同一证书签名，则两个应用程序在系统中拥有相同的用户ID。拥有同一用户ID的应用会运行在同一Linux进程，分享同一虚拟机。开发者也可以通过分享ID来达到应用间数据分享的目的。

## 2.4 Android中存在问题及相关技术分析

2.4.1 Android信息泄露

由于Android平台涉及大量用户隐秘信息，使得有关Android隐秘信息泄露问题层出不穷。例如2010年的Tech Credit Union，该应用通过把用户银行账户连接到客户端，然后盗取用户的银行卡信息。在2010年的RSA大会上，Derek和Daniel两位研究者指出用恶意程序可以轻松获取用户的隐私数据。同年的8月，Android系统上第一个木马病毒出现，该病毒伪装成了无害的播放程序。一旦用户把它安装到手机中，他就利用受害用户手机发送收费短信。在2011年3月，攻击者通过攻击Android市场，然后利用恶意软件获取用户设备信息。2011年12月则又爆发了Carrier IQ事件[8]。Carrier IQ是手机制造商预制在Android系统中的软件，并且一般用户不能卸载。但一些研究人员发现，Carrier IQ会对搜集和上传用户的短息、键盘和上网记录。三星和HTC等一些著名手机制造商受到影响，而且受此事件影响的用户量更是惊人。复旦大学在2012年做了一份研究报告，报告称国内超过半数的应用会在用户不知情的情况下获取用户隐私信息，并将这些信息上传至应用开发商的服务器以获取牟利。

2.4.2 Android安全研究国内外研究状况

自从Android问世以来，它的平台开放性和功能丰富性让很多研究人员产生了兴趣，针对Android系统的研究也越来越多，而Android的研究主要分为Android的架构研究、内核研究、应用的权限研究和Android进程通信研究等。由于Android系统的使用范围已经遍布全球，其安全问题被每一个国家列为重点研究对象，对Android安全问题关注程度一直是居高不下。Android的安全性研究宏观上可以分为系统研究和应用程序研究，而在更细粒度级别上又可以分为诸多小类，如表1.1所示。

表1.1 Android的安全性研究

|  |  |
| --- | --- |
| Android安全研究 | 研究内容描述 |
| Android系统的安全 | Android系统的安全模型 |
| 进程的安全性 |
| 设备通信协议安全性 |
| Android系统的安全漏洞等 |
| Android应用程序安全 | Android的文件格式 |
| 恶意代码，病毒和木马程序 |
| 用户密码保护 |
| Android程序的数字签名 |
| Android数据加解密 |
| 应用程序的权限使用 |
| 隐私数据的安全性等 |

由于Android平台独特的权限系统涉及到了用户大量的隐私信息，使得目前针对Android权限研究逐渐增加。在2009年，Ongtang等人[9]对Android权限模型提出了质疑，指出当时Android权限模型无法满足一些安全性需求，并对提出的问题进行扩展和增强。然而，这些扩展性后的权限模型比较复杂，不适合大规模推广。Enck等人[10]认为Android这些权限本身不会存在风险，但是多个权限组合后可能会增大风险的可能性，他们研究出Kirin检测工具，该工具可对同时使用多个权限的程序进行风险提示。例如，一个应用程序如果同时申请打电话、录音和链接Network的权限，这样Kirin就会进行风险提示，因为这样程序会可能会把通话录音传至互联网的。为了防止应用在动态运行时权限提升， Quire[11]、XMAndroid[12]等防止权限提升系统工具出现在大众面前。Barrera等人[13]利用数据挖掘的方式分析权限间的相关度，找出一些相关度比较高的权限相关集，这对今后Android权限模型研究有很大意义。

在Android中的隐私数据保护中，Enck在2010年提出了TaintDroid[14]动态检测机制。TaintDroid主要是针对Android移动平台的隐私信息监控系统。通过修改Android平台中Dalivk虚拟机堆栈结构，然后结合传统的污点技术，TaintDroid能在运行态时对系统中敏感信息的传播进行监控，会把导致隐私信息的传播写到相应的日志文件中[15]。针对TaintDroid技术，本文会在第三章进行详细介绍。Hornyack在TaintDroid的基础上，设计了AppFence[16]，在动态运行时监控敏感数据的传播，将发生泄露的敏感数据伪造成Shadow Date，这样不仅可以不影响程序运行，而且还能从最大程度上保护用户的隐私数据，同样对已经泄露的隐私信息也是一种保护。与AppFence相似的系统还有TISSA[17]和MockDroid[18]，为了防止隐私数据的泄露，他们会伪造一些隐私信息数据，这样供攻击者就无法区分获得的数据是否为正在的隐私数据。

面对Android系统中隐私信息泄露问题，国内外的研究机构以及学者都做出了努力，提出了许多有意义的理论模型并设计了大量检测工具。通过研读这些理论和原型工具的文献资料，本文总结了Android隐私数据的防护方法主要分为两种：Android静态分析和动态分析。2.4.2和2.4.3会详细介绍这两种方法。

2.4.3 Android静态安全分析

静态分析方法[19]是一种通过对源代码分析来发现漏洞的方法，静态分析方法不需要运行代码。静态分析的方法把程序的源码用一种抽象的表示描述出来，例如树或者图，然后用数据结构的方法对这些树或者图进行具体的语言表示，然后根据定制好的规则或者漏洞检测模型生成分析算法对其进行分析。静态分析中常见的分析方法有词法分析、模型匹配分析、约束分析和类型推断。静态分析的优点是能对代码进行全面覆盖分析，可以实现代码分析的自动化，效率高等。相反的一面是，静态分析需要获取源码、模式库和安全规则。而静态分析的效果会受到安全规则和模式库的影响；虽然静态分析对代码覆盖面较广，但是需要分析所有执行路径，这就会导致大量的虚假信息，是误报率大大提升。

静态分析技术已经在C语言的漏洞检测上使用。Evan的Splint的静态分析器[20]和Cqual[21]，它们把C的源代码作为输入，然后使用”tainted”和”untainted”作为标注来查找程序中是否有字符串格式化以及缓冲区溢出等漏洞。但基于Splint和Cqual的静态分析方法受到不确定性问题约束比较多。Shankar等人[22]使用类似的方法检测漏洞，但是他们加入一个新的限定器，可以标记来源不可信的数据。Denning[23][24]定义一种理论证明机制，在程序给定的安全类别下确定数据流和流关系的一致性。为了构造这种机制，程序的编译阶段则使用了晶格模型。JFlow编译器[25]通过使用信息流标记和形式化规则来阻止存储信道中的信息泄露。尽管目前Android的绝大多数应用程序用Java编写，但是目前比较先进的Java静态分析工具并不适合Android应用程序。不适用的原因主要由三方面组成：

1. Android应用主要由组件组成。而组件又分为Intent Filter 和Intent。Intent主要应用于Android应用间的通信与交互，可以对应用的操作进行描述。而Intent又分为显示intent和隐式intent。显式Intent主要针对应用内部组件通信，它在构造Intent对象时就指定接受者；而隐式Intent在构造对象时则不关心接收者，这样做的好处是降低两者间的耦合度。Intent Filter根据Intent的属性信息，如动作(action)，数据(data)和优先级(priority)等可以对Intent进行过滤。当系统启动组件时，需要运行态时才能对Intent和Intent Filter进行匹配。这种动态交互的会对Android静态分析带来一定困难。
2. Android应用大多数以用户为中心，用户通过触屏与系统交互。Android系统往往采用特定的回调函数方法对用户的输入进行处理，例如onClick方法。而静态分析的方法就无法分析一些由回调函数引起数据泄露问题[26]**。**。
3. Android系统中的组件存在生命周期。在Android系统中，会通过onStart, onResume 或onCreate回调函数来转换组件的生命周期的状态，这也给静态分析技术带来巨大挑战。

通过对传统静态分析方法的改进，许多研究人员针对Android平台设计出了许多基于静态分析技术的Android保护系统或者模型，如表2.2所示。

表 2.2基于静态分析的Android防护系统/模型

|  |  |  |
| --- | --- | --- |
| 系统/模型名称 | 系统/模型保护机制描述 | 要解决的Android安全问题 |
| ComDroid[27] | 利用逆向工程和静态分析技术分析代码 | 查找存在Android组件中的漏洞 |
| FlowDroid[28] | 获取Android应用程序的manifest 等xml文件，提取信息，构建隐式信息流调用图 | 能够探测无意或者恶意的intent导致的数据流泄漏 |
| DroidChecker[29]，CHEX[30] | 分析组件间的信息流 | 挖掘由信息流造成组件劫持和功能暴露等安全漏洞 |
| ICCTA[31] | 静态分析应用间的数据流 | 检测程序应用间组件通信时是否有信息泄露 |
| EdgeMiner[26] | 静态分析整个Android框架，并生成所有回调函数的的接口集合 | 对解决由回调函数导致的信息泄露问题分析提供帮助 |

2.4.4 Android动态安全分析

动态分析是在不知道源码的情况下通过运行程序来分析的程序安全性，属于黑盒测试方法。常见的动态分析方法有输入追踪、故障注入分析和堆栈比较等。动态分析具有针对性强、准确率较高等优点。但是由于没有源码，动态分析也存在代码覆盖率低和漏报的情况。动态分析可以通过异常数据输入或者对数据污点标记来分析程序安全。像Fuzzing[32]漏洞分析技术就是通过对程序不断输入异常的数据或者文件来分析程序安全性。

污点分析起源于对程序数据声明周期的追踪[33]。该方法主要通过对要分析的变量进行污点标记，然后利用信息流分析的方法来对这些数据分析。污点分析系统对这些带有标记的变量以及其派生数据跟踪分析直至其生命周期结束。动态污点分析是在污点分析上基础上进行扩充，主要是在程序执行时实时跟踪数据。这些动态污点分析系统会在在污点数据到达特定的程序模块时做出相应动作，用来检测程序执行路径或者分析数据是否泄露。

污点标记机制已经被多种编程语言实现，并且应用到了多种用途。许多流行的语言都已经可以进行污点检测。这些应用可以帮助用户在编写程序时进行程序测试，来发现程序错误或者漏洞。一个最常见的例子就是验证输入， 当外部信息第一次输入系统的时候被标记成污点数据来跟踪信息，确保污点数据在验证前的安全性。如图2.2所示， X作为外部输入数据被标记成污点数据，然后在程序执行的过程中，经过函数F1和F2的转化，X被演化成数据Y和Z，因此Y和Z也被作为污点数据来跟踪。当数据Z到达一个不安全的函数F3时，污点跟踪系统就会发出警报。

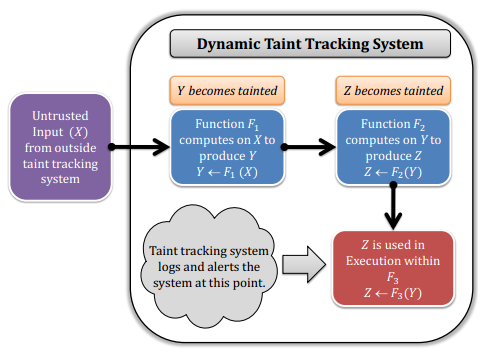


图2.2污点标记系统的工作模式

编程语言Perl[34]的数据标记模式是经典的污点分析工作之一，它能显式的标记任何来自程序外的数据，并能阻止数据在一些隐私函数中的声明，避免污点数据影响本地命令的运行以及文件的创建、读写或者数据的网络传输。与Perl类似，Ruby编程语言也使用了污点检测机制。相比Perl而言，它的粒度更细，它对安全级别也有划分。Perl设置了0-4五个安全级，并且对每一个级别都设置了不同的安全检测机制。但是，Perl和Ruby语言的污点检测方法共同限制是它们只能对特定代码的脆弱性进行保护。不仅仅是编程语言，其他程序保护工具也使用污点标记模式。TaintCheck[35]在程序的二进制级别使用Valgrind[36]来进行程序的动态污点分析。基于DynamoRIO[37]， TaintTrace[38]对程序做了优化，相比TaintCheck，它大大降低了系统损耗。同样，LIFT[39]在软件级别上对信息流追踪的分析系统损耗也比较低。Haldar等人[40]实现了基于Java动态污点分析，他们通过标记跟踪用户在程序运行时输入的数据，来阻止那些恶意影响程序执行的污点数据传播。

随着Android平台的推广和大面积使用，研究人员发现，Android系统也可以使用类似的动态监控方法能很好保护Android平台的安全。动态污点分析技术在Android平台上的分析也愈来愈多，如表2.3所示。

表 2.3基于动态分析的Android防护系统/模型

|  |  |  |
| --- | --- | --- |
| TaintDroid[14] | 修改Dalvik虚拟机，对获取的隐私数据进行污点标记，并进行跟踪 | 解决用户无法控制Android中隐私数据流问题 |
| DroidScope[41] | 对TaintDroid进行扩展，使用qemu仿真，对指令级别信息进行监控 | 检测Android中的恶意应用 |
| Droidbox[42] | 对Android应用程序和linux之间的交互进行监控 | 对Android应用和底层交互带来的数据泄露问题进行检测 |
| NDroid[43] | 使用了QEMU仿真系统，对JNI中的隐私信息流进行污点标记并监控 | 针对TaintDroid对JNI环境中的隐私信息控制不足问题进行改进 |
| CopperDroid[44] | 对Android应用的行为进行分析，使用qemu仿真，对指令级别信息进行监控 | 检测Android中恶意应用程序 |
| AppFence[16]，TISSA[17]，MockDroid[18] | 动态监控系统中隐私数据的传播，对将要泄露的信息进行伪装 | 对Android系统中外泄的隐私数据进行保护 |
| VetDroid[45] | 通过显式和隐式两种方法分析了Android应用程序的内部逻辑，并分析了应用程序内部如何使用敏感资源；并对应用程序的行为和callback进行了分析 | 监控引用安全行为保护隐私数据 |
| TaintEraser[46] | 主要对应用层的商业应用进行分析 | 监控Android上商业应用引起的信息泄露 |

## 2.5 本章小结

本章首先从对Android系统进行划分层次，根据划分的层次对Android整个系统框架的进行了详细分析，并对Android安装应用内部文件的作用进行了细致研究；其次从系统和应用两方面出发介绍了Android的安全机制，对目前引起Android系统的信息泄露问题的原因进行了总结性分析。然后本章对于目前程序分析主要的方法分析并总结了两种程序保护技术，一种是静态分析，另一种为动态分析，并对两种技术的优缺点深入介绍。最后本章针对这两种技术以及对应的Android保护工具进行了总结。

# 3 TaintDroid分析

Android系统中涉及了大量用户隐私的信息，这些信息包括用户的设备号，短息，联系人和位置等信息。而一些生活消费商家为了获取手机用户一些隐私信息，往往与应用开发商合作获取用户手机号、地理位置等信息。由于用户无法了解并控制应用程序获取隐私信息的方式，使得多数Android应用程序在用户不知情的情况下获取手机的GPS、IMEI、联系人等隐私信息，这无疑给用户带来一定的利益损失。而在目前的Android应用中，大多数应用会有获取隐私数据的权限。而这些应用程序中很多属于蓄意泄露用户的隐私；这些应用程序往往会通过动态获取的方式得到这些隐私信息，因此Android的静态程序分析工具就无法保护数据的安全性或者检测隐私信息的泄露。而TaintDroid则刚好解决上述的弊端，与静态分析技术相比，TaintDroid首先对敏感信息进行污点标记，然后通过运行指定的程序，监控污点数据在程序中的流向，最终在sink（例如网络、写文件等）处进行拦截，获取完整的隐私数据动态流向。在带有污点的敏感信息被发送到sink时，TaintDroid对其进行写日志的操作，这些行为被统一记录在日志文件中，供研究者进行分析。图3.1描述了TaintDroid监控隐私信息的传输整个过程。

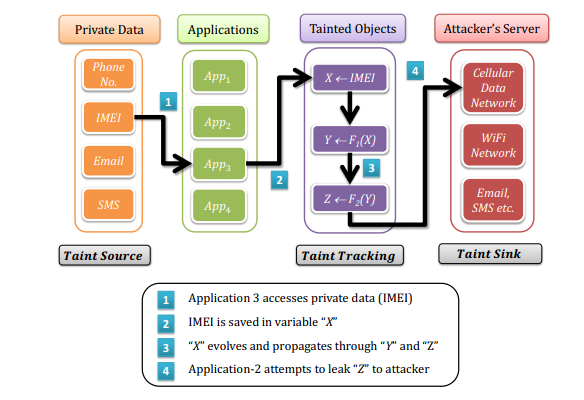


图3.1TaintDroid监控信息流架构图

## 3.1 TaintDroid设计概述

由于多数Android的应用程序多数都是未开源的，而且隐私数据的泄露一般需要动态完成。因此，利用静态分析技术污点标记数据和监控数据的流向显然是不合适的。在源代码可用的情况下，一些信息的使用往往由运行事件支配决定，而且只有在实时的环境下才能确定一些信息的依赖关系，所以要依靠实时监控才能确定这些信息的流向。TaintDroid设计目标旨在让Android手机用户实时监控应用程序对隐私数据的操作。通过对Android系统的扩展，TaintDroid成为首个实时跟踪手机隐私信息流的平台。动态污点分析技术在监控数据的传播时，往往会受到精确度和性能的影响。如果监控的力度过细，不仅会对造成系统的开销过大，也会带来污点爆炸的问题；如果监控力度过粗，无疑精确度就会变得过低，漏报率就会增加。因此，为了在精确度和系统性能间找出一个平衡点，TaintDroid则从变量级别、方法级别、消息级别以及存储级别这四种粒度上考虑数据的污点传播，使得TaintDroid性能和精确度都得到体现。

指令级别的动态污点分析会对系统资源的消耗和性能都带来巨大影响，因为对于每个受监控的指令而言，污点监控系统必须首先保存指令的上下文，然后执行污点传播，最后还需要恢复指令上下文，这势必会提升系统开销。对Android系统来说，有限的系统资源不能为这种污点分析方式提供足够的CPU、内存等硬件资源。TaintDroid通过修改解释器执行污点传播，这样的好处就是不需要对指令进行上下文保护和恢复，降低了系统资源使用。除此之外，TaintDroid还对文件操作以及数组操作等污点传播进行了定义，文件的污点数据读写传播在3.2.4节会有详细说明。

## 3.2 TaintDroid的实现

3.2.1 TaintDroid污点标记存储

污点标记存储方法对系统的性能以及存储效率都会带来一定的影响，因此选择合适的污点存储方式是TaintDroid设计的首要问题。传统的污点跟踪平台往往会对每一个字节或者字附加一个标记，而该标记通常为一个比特位。这些平台会采用标记图或者非相邻影子内存技术[47]，然后只对污点字或者字节的标记维护，这样可以降低内存的存储开销。由于TaintDroid能区分解释器中的变量，TaintDroid把污点标记存储在变量相邻的内存。这样，TaintDroid就可以仅通过持续跟踪变量的污点状态来缩小要跟踪的内存范围，这会为污点的访问带来空间局部性，明显缩短了污点访问时间。对于Android手机而言，IMEI，地理位置信息，摄像头等均为敏感信息，而敏感信息类型各有不同。TaintDroid允许对每个变量存储一个32位的位向量，这样能够标记出32种不同的隐私信息种类，隐私信息类型如图表3.1所示。表3.1只给出了16种污点类型，对于另外类型的污点标记，TaintDroid则是通过污点传播语义把不同隐私信息的污点类型进行了结合操作。

表3.1 TaintDroid中隐私数据污点类型

|  |  |  |
| --- | --- | --- |
| 污点类型 | 宏定义 | 污点值 |
| 清除污点标记 | TAINT\_CLEAR | 0x00000000 |
| 位置信息 | TAINT\_LOCATION | 0x00000001 |
| 联系人信息 | TAINT\_CONTACTS | 0x00000002 |
| 话筒 | TAINT\_MIC | 0x00000004 |
| 手机号码 | TAINT\_PHONE\_NUMBER | 0x00000008 |
| GPS位置信息 | TAINT\_LOCATION\_GPS | 0x00000010 |
| 网络位置信息 | TAINT\_LOCATION\_NET | 0x00000020 |
| 当前位置信息 | TAINT\_LOCATION\_LAST | 0x00000040 |
| 设备摄像头 | TAINT\_CAMERA | 0x00000080 |
| 加速传感器 | TAINT\_ACCELEROMETER | 0x00000100 |
| 短信息 | TAINT\_SMS | 0x00000200 |
| 设备识别ID | TAINT\_IMEI | 0x00000400 |
| 移动用户ID | TAINT\_IMSI | 0x00000800 |
| SIM卡的编号 | TAINT\_ICCID | 0x00001000 |
| 设备序列号 | TAINT\_DEVICE\_SN | 0x00002000 |
| 账户信息 | TAINT\_ACCOUNT | 0x00004000 |
| 信息记录 | TAINT\_HISTORY | 0x00008000 |

Android应用使用的方法分为两种，一种虚拟机内部方法，另一种是JNI的方法。当方法被调用时，Dalvik虚拟机会为每个局部变量分配一个新的堆栈帧，对于这些变量和引用的污点标记存储，方法中的变量和对象引用变量都会另外分配32位存储空间，在对寄存器取值时，假如变量vi存储在寄存器相对偏移量为fp[i]的位置，增加污点空间后就要在fp[2\*i]处取值；而对于64位的变量，Dalvik虚拟机需要两个32位的寄存器对其存储，而解释器在管理寄存器时都是单独处理，遇到64位值直接取两个寄存器值；TaintDroid则采用类似的方法存储污点标记，在对64位变量污点值存储和读取时，也是用两个32位寄存器。图3.2表示，TaintDroid修改了Dalvik虚拟机的栈结构来增加栈的大小存储和寄存器相关污点标记。对于虚拟机内部方法的调用，TaintDroid把参数的的污点标记存储在当前栈帧结构的taint tag处，然后为被调用的函数局部变量分配栈slot，用栈帧指针指向被调用方法的第一个局部变量。之后，TaintDroid为保存调用函数信息，在栈的顶部分配一个Stacksavearea；当被调用的方法返回类型不为空时，TaintDroid会把这些返回值的污点标签保存到当前线程的InterpSaveState处。如果调用的是一个本地方法，TaintDroid同时存储传递参数和返回值的标签。对于数组和引用字段值的污点标记，TaintDroid对数组和引用中的字段数据结构都进行了修改，扩充了32位的存储空间。数组和引用的污点标记就直接存储在他们的内部结构中。这样修改的优势是数组和引用仅仅需要存储一个污点标记值，大大降低了存储开销。当然，数组的污点标记也会带来一定的误报例率，如一个未被污点标记变量x存储在数组A下标为0的位置，即A[0] = x；现在有一个被污点标记的变量变量y存放在A[1]处，即A[1] = y；现在要对A[0]进行读取，那么读出来的值也会被污点标记。不过在实际使用中，java往往使用较多的是引用，而非数组；而对对象的污点标记并没有想象中那么频繁，大大减少了误报率。

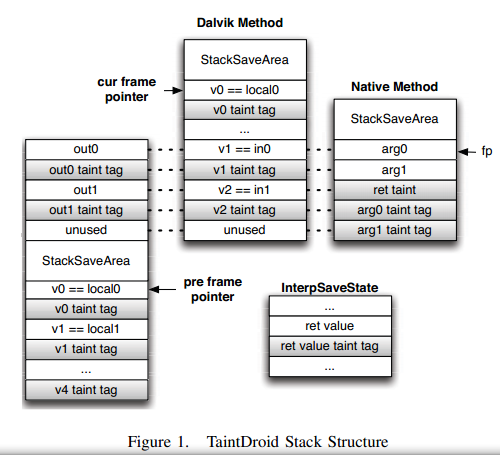


图3.2 TaintDroid对栈修改

3.2.2 TaintDroid解释型代码污点传播

Dalvik虚拟机使用dex字节码形式运算，这为TaintDroid标记变量时带来很多优势：首先，dex字节码的运算都有明确的语义，方便污点标记的跟踪；其次，非方法的局部变量可保留明确类型的污点标记。针对污点在dex解释代码中的传播行为，TaintDroid定义了不同的传播语义，共为16种，如表3.1所示。τ()表示污点映射函数，τ(v)表示取出变量v的污点标记值，τ(u) ←τ(v)则表示把v的污点标记值赋给变量u；寄存器变量和类中的类中字段值分别用vX和fX表示，vX[.]为数组，假如vY为一个实例对象，则vY(fX)则表示fX为vY中的一个字段值；R和E分别表示解释器中的返回值和异常值；A、B和C表示字节码中的常量。

在多数情况下，污点标记传播是预期进行的，指令会始终覆盖目标寄存器。因此，一元运算会把目标寄存器的污点标记设为原来寄存器的污点标记；而对于二元运算，会把目标寄存器的污点标记设置为两个源寄存器的并集，如a = b + c ， τ(vA) ← τ(vB) ∪ τ(vC)。在一些特殊情况下，数据的下标以及对象的引用等污点标记并不是直接把目标寄存器污点标记设置为源寄存器的污点标记。

表3.1 TaintDroid的传播语义

|  |  |  |  |
| --- | --- | --- | --- |
| 操作格式 | 操作语义 | 污点传播 | 描述 |
| const-op vA C | vA ← C | τ(vA) ← ∅ | 清除变量vA的污点标记 |
| move-op vA vB | vA ← vB | τ(vA) ← τ(vB) | 把vA设置成vB的污点格式 |
| move-op-R vA | vA ← R | τ(vA) ← τ(R） | 把vA设置成返回值的污点格式 |
| return-op vA | R ← vA | τ(R) ← τ(vA) | 设置返回值的污点格式(函数类型为void则为∅) |
| move-op-E vA | vA ← E | τ(vA) ← τ(E) | 把vA设置成异常值的污点格式 |
| throw-op vA | E ← vA | τ(E) ← τ(vA) | 设置异常值的污点格式 |
| unary-op vA vB | vA ← vB | τ(vA) ← τ(vB) | 把vA设置成vB的污点格式 |
| binary-op vA vB vC | vA ←  vB vC | τ(vA) ←  τ(vB) ∪ τ(vC) | 把vA污点格式设置成vB和vC的并集 |
| binary-op vA vB | vA ←  vA vB | τ(vA) ←  τ(vA) ∪ τ(vB) | 把vA污点格式更新成vB和vA污点格式的并 |
| binary-op vA vB C | vA ← vB C | τ(vA) ← τ(vB) | 把vA设置成vB的污点格式 |
| aput-op vA vB vC | vB[vC] ← vA | τ(vB[·]) ←  τ(vB[·]) ∪ τ(vA) | 把数组vB的污点格式更新成vB和vA的并集 |
| aget-op vA vB vC | vA ← vB[vC] | τ(vA) ←  τ(vB[·]) ∪ τ(vC) | 把vA的污点格式设置成数组和数组下标的并集 |
| sput-op vA fB | fB ← vA | τ(fB) ← τ(vA) | 把字段值fb设置成vA的污点格式 |
| sget-op vA fB | vA ← fB | τ(vA) ← τ(fB) | 把vA设置成字段fB的污点格式 |
| iput-op vA vB fC | vB(fC) ← vA | τ(vB(fC)) ← τ(vA) | 把字段值fC的污点格式设置成vA的污点格式 |
| iget-op vA vB fC | vA ← vB(fC) | τ(vA) ←  τ(vB(fC)) ∪τ(vB) | 把vA的污点格式设置成引用对象vB和fC的并集 |

在污点信息传播语义中，当为带有污点的敏感信息赋一个常数值，TaintDroid就会清除变量的污点标记信息(const-op)；在函数返回值不为空，若带有污点的变量被返回，获取返回值的变量将被打上污点标记(return-op)；在进行二元操作时，要看操作变量是否被全部污点标记，如果只有一个操作数被污点标记，就把该污点信息标记赋值到目标寄存器；如果操作数都被污点标记，则对污点标记进行合并操作，然后目标寄存器的污点类型为这些污点标记的并集(binary-op vA vB vC 和binary-op vA vB C)。

3.2.3 TaintDroid程序间通信污点传播

当Android进程之间进行通信时，它们会绑定进程的通信接口来发送数据。Android进程间通信是以数据包的形式进行交互。TaintDroid就是通过对进程之间发送的数据包标记来监控进程间的通信；但是数据包中包含了不同的数据类型，标记的粒度的大小会直接影响系统的性能和内存开销。TaintDroid则采用信息级的污点跟踪。当应用进程对另一个经常发送数据包时，TaintDroid为此数据包分配一个污点标记，而不是对数据包中的变量或者字节进行标记；如果采用变量粒度数据包进行污点标记，接收器可以通过打包的方式移除污点标记，从而逃过污点TaintDroid的监控。除此之外，消息级的污点标记大大降低了内存开销，使污点传播的性能变得更高。

3.2.4 存储污点传播

当带有污点标记的数据写入文件时，该文件就会包含这一隐私内容。TaintDroid为每一个文件都分配一个存储标记，以便保存在文件中的污点数据在被读取时污点标记得以恢复。这样的做法是考虑到在更细粒度时，对文件的频繁操作会对系统造成重大开销。但是这样做也会存在一定的弊端，当写入文件的污点标记类型较多时，误报率也随之上升。为了存储污点标记，TaintDroid对文件系统的属性进行扩展。最初，YAFFS2文件系统不带有xattr支持，TaintDroid把YAFFS2文件系统加入正式的xattr支持。除YAFFS2外，Android的扩展卡使用的是FAT文件系统，该系统也不支持xattr。TaintDroid将扩展卡格式化为ext2格式并对文件写入的API进行修补，这样就可以使用与FAT文件系统使用相同的文件权限，确保应用程序的兼容性。

## 3.3 TaintDroid设计的不足

TaintDroid在用户隐私数据的监控上有革新性的突破，但仍存在一些设计上的不足。TaintDroid在对隐私数据进行监控时，仅仅对信息的显式流进行跟踪。对于JNI和控制流带来的信息传播，TaintDroid不能监控其传播。如果一个信息流从敏感源到一个能泄露信息的sink而未被监控，信息泄露就会发生。本文把能敏感信息的出处看做来源(source)，来源和sink可以是java 环境也可以是JNI环境。因为从信息来源到sink会经过不同的环境，本文把从source到sink所经过的环境成为intermediate(简写为int)。表3.2列出可能的通过JNI的信息流的三元组组合{source,intermediate,sink }。如果source， int和sink处于同样的环境，本文用N/A表示。当source和sink全部是java环境，int是本地代码环境，那么敏感信息的传播经过的环境如表中的case1 和case1’ 所述，其中Taintdroid只能检测出case1。

Case1：获取敏感数据后，java代码调用本地方法处理这些数据并把它发送到sink。如图三a所示，java代码首先调用本地方法，并将带有敏感数据的变量以参数的形式传给本地方法接口；数据经过处理后，经过返回值形式把处理后的数据送出至java环境。Taintdroid能检测这类的信息传播，因为它标记传入的参数带有污点标记，因此本地方法的返回值也会被标记。

Case1’：如图中的3.3 b所示，java代码调用本地方法，并带有敏感数据的变量以参数的形式传给本地方法，而本地方法却不把返回值发出去。然而，另一段java代码通过本地方法的调用可以获取带有敏感信息的变量(step 2’’)；或者该本地代码调用java代码，并把敏感数据变量以参数的形式从本地环境传到java环境。最后java代码最终把隐私数据泄露(step 3)。因为TaintdDroid没有标记来自本地方法的隐私数据，因此无法检测这种类型的数据泄露。

Case2：如图3.3 b所示，本地方法在收到java代码发来的带有污点标记的数据后，会直接通过本地方法，把数据发送至设备外或者写到文件中(step2)。Taintdroid无法检测出这种类型的数据泄露，因为taintdroid没有追踪本地环境中隐私数据和本地方法的sink。

Case3：如图3.3 c所示，本地方法主动调用java环境来获取隐私数据(step1)，本地代码获取敏感数据后，再调用java环境中代码以参数形式传播把隐私数据传至java环境(step3)，或者本地方法获取隐私数据后，等待java环境中的方法调用它(step 3’)。最后通过java代码环境把隐私数据传输至设备外。taintdroid没有标记该隐私数据，因为该隐私数据来源为本地方法。

Case4：如图3.3 c所示,本地方法首先通过java环境中获取敏感信息（step 1），然后从本地方法中直接泄露隐私信息(step 2)，taintdroid无法检测该类泄露，因为它没有标记来自本地方法的数据。

表3.2 {source,intermediate,sink }组合

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sink | | Java | | | | Native | |
| Intermediate | | Java | | Native | | Java | Native |
| Source | Java | N/A | Case 1 | | Case 1’ | Case2 | |
| Native | Case 3 | | | | Case 4 | N/A |



图 3.3 JNI泄露隐私数据方式

除了JNI中的信息传播外，由控制流带来的隐式传播TaintDroid也无法监控，本文将会在第四章详细介绍。

## 3.4本章小结

本章针对Android系统隐私信息监控TaintDroid进行了分析，首先对TaintDroid设计进行了全局的描述；其次从污点标记在变量和类字段等中的存储，污点数据解释代码中的传播以及进程间通信等污点传播方式分析了TaintDroid的实现；最后真对引起TaintDroid污点标记不足的原因进行了总结，并详细分析了由JNI调用引起的污点标记不足导致信息泄露的问题。

# 第4章 污点标记不足模型和解决方案设计

在程序中的数据传播时，一般会有两种传播方式，一种是显式传播，另一种为隐式传播。显式传播为一般的变量赋值，例如x=y+z，这就是一个显示传播的过程，而隐式传播则是通过一些条件语句引起，例如for、while循环if和switch条件分支等。显式传播的分析主要采用的方法是数据流分析，通过到达-定值分析等方法可以对引起变量变化的条件进行分析。变量y和变量z通过加法运算把得到的结果显式赋值给x，这也意味着变量x对y和z形成了数据依赖。而TaintDroid主要针对数据显示传播进行污点标记，即变量y或者变量z如果被污点标记，对y和z数据依赖的变量x也进行污点标记。在程序程序控制流中，一些变量往往对条件语句不会形成直接的数据依赖关系。表4.1的描述一段if语句形成的隐式传播，变量z通过Get\_sensitive()函数获取一个整型隐私数据，变量x和y分别赋值为true，if语句主要判断z获取的值是否为零，然后对x和y进行

表4.1 if语句代码段

|  |  |
| --- | --- |
| line | code |
| 1 | int    z = Get\_sensitive() ; |
| 2 | boolen x = true ; |
| 3 | boolen y = true ; |
| 4 | if (z == 0) |
| 5 | x = false ; |
| 6 | else |
| 7 | y =false ; |
| 8 | output(x,y); |

相应的false赋值操作。通过output对变量x和y进行输出结果可以判断z是否为0，这就对z造成一定的信息泄露。对于控制流的相关工作会在第5章进行介绍。

## 4.1 控制流技术的相关研究

在早期就有很多学者对由控制流引起的程序安全做了研究。Fenton[48]提出了一个抽象的Data Mark Machine模型，该模型会把每一个程序计数器p关联一个安全级s。当程序进入一个条件分支时，condition c：s1,s2,……sn，程序计数器p会把当前安全级设置成ps ⊕ cs；如果一个语句的依赖多个控制语句c1,c2……cn，那么它的程序计数器的安全级会被设置成为c1s⊕……⊕cns；如果信息从对象a1,a2……an传至对象b，那么对象b的计数器安全级被设置为c1s⊕……⊕cns⊕ps。该模型被证实在保护控制流安全上非常有效，但是对程序整体安全保证上效果就变得不太理想。Gat和Saal等人[49]对Data Mark Machine进行了改进，但是改进后的模型太过理想化而无法实现。Brown [50]认为依赖条件语句中的执行语句s的安全级别如果小于条件语句c中的安全级别，再把s的安全级别设置成为c的安全级，这就会引起一定隐私信息泄露问题。他们则严格按照安全级别的高低来制定信息传播策略。而Beres和Dalton[51]认为如果采用Brown的方法，则会影响程序的正常执行，并且会通过一些隐性信道泄露信息。Denning[52]把条件分支中的执行分支和未执行分支中的执行语句都进行安全级别的更新，更全面的显示出控制语句的信息传播。BitBlaze[53]把静态分析和动态污点分析进行创新性的结合，既可以监控程序的显式流也可以监控隐式流。基于BitBlaze，DTA++[54]对BitBlaze中的动态污点分析进行了加强，取得了比较理想的效果。RIFLE模型[55]通过结合硬件在程序运行时对信息流进行跟踪，但是移植性并不理想。Chandra[56]通过扩展Java结构，对信息流的污点进行跟踪，但是他们污点标记使用的是整数，性能受到很大的影响。Trishful[57]使用一个比特位来当做污点标记位，并设计了一套控制流中的污点跟踪的方案；针对被标记的污点，Trishful还指定了污点传播的策略。虽然这些模型或者工具可以对控制流中的隐式信息监控有一定的效果，但是它们无法解决Android手机中存在的隐私传播信息泄露问题。

## 4.2 由程序控制流引起的TaintDroid攻击模型

对于一般的Android应用程序动态防护平台而言，它们监控的主要是程序中的显示数据流，却无法监控对控制流的传播。在动态污点分析技术中，控制流往往会引起信息的污点信息标记不足(under-tainting)的问题。

Sarwar[34]和等人专门针对TaintDroid等Android动态污点监控系统做了攻击实验，实验结果表面程序控制流可以造成一些基于污点标记的防护系统的信息泄露。实验中，研究者们把带有污点标记的信息放入控制语句，通过条件语句中的执行语句对带有污点信息进行隐式传播。类似TaintDroid这样的信息防护系统并未检测这一类的信息传播。而这一特点往往会背攻击者利用，他们通过代码注入等方式，在应用中加入一些混淆代码，利用隐式传播把隐私数据赋值到常规变量，并利用写文件或者互联网上传等操作达到攻击的目的；不仅如此，Android多数应用程序为了反逆向等在代码中使用了大量的判断语句和循环语句，这在无意中对Android隐私信息带来一定的泄露。针对这些控制流造成的信息泄露，本文对TaintDroid使用以下攻击模型。

表4.1 攻击模型1

|  |
| --- |
| 攻击模型1 |
| X ←Get\_Sensitive\_Data() |
| for each x ∈ X do |
| for each s ∈ AsciiTable do |
| if ( s == x) then |
| Y ←Y + s |
| end if |
| end for |
| end for |
| SendtoNetwork ( Y ) |

表4.1描述了第一类攻击模型。变量X通过Get\_Sensitive\_Data()函数获取了一个隐私数据。攻击者利用循环操作把隐私数据X中的字符一一读取后赋值到变量x，利用字母表查找的方式找出x的值，最后通过if语句把X中的每一个字符赋值到变量Y中。等到循环结束，X的值就会完全赋值给Y，并且Y不会被污点标记。因为TaintDroid不会在控制语句中传播污点，因此Y通过联网传值，把隐私信息X传播到设备外部。

表4.2 攻击模型2

|  |
| --- |
| X ←Get\_Sensitive\_Data() |
| for each x ∈ X do |
| n ←CharToInt( x) |
| y ←0 |
| for i =0 to n do |
| y ←y +1 |
| end for |
| Y ←Y + IntToChar( y ) |
| end for |
| SendtoNetwork ( Y ) |

表4.2描述了第二类攻击模型。攻击者可以把隐私信息存储在变量X中，然后利用循环操作读取变量X中的每一个字符x，并把字符x转换成一个整数。通过下一个循环，攻击者可以不断增加y的值来得到整数n值。最后，攻击者通过字符转换，把y转换成字符然后通过字符连接操作，把这些字符拼接到变量Y中。Y中包含了隐私信息值，但不会被TaintDroid标记。因为在CharToInt( x)函数中，n会被污点标记；而对变量n的for循环操作，变量y虽然获得了n的值，但n并未对y进行直接赋值，因此y不会被标记。最后攻击者利用网络传输函数把Y传出设备，而且不会引起设备的警告。

表4.3 攻击模型3

|  |
| --- |
| X ←Get\_Sensitive\_Data() |
| for each x ∈ X do |
| n ←CharToInt( x) |
| y ←0 |
| While y < n do |
| Try { |
| Throw NewException() } |
| Catch( Exception e) { |
| y ←y +1} |
| end while |
| Y ←Y + IntToChar( y ) |
| end for |
| SendtoNetwork ( Y ) |

攻击模型3属于一种基于异常的攻击。攻击者把隐私数据X所包含的字符x进行整数转化，利用CharToInt(x)函数把字符x转化成整数n。然后用while循环对n进行循环并引发n次异常操作。攻击者在抛出异常的的捕获catch块中不断增加y的值。最后使用字符连接操作，隐私数据X中所有的字符全部赋值到变量Y中。TaintDroid也不会检测到Y中包含了隐私信息，因为在异常操作中存在这控制流传播，而TaintDroid对于控制流信息无法监控。由第三个攻击模型知，攻击者可以用异常操作方式获取隐私信息。

通过上述三个攻击模型不难发现，控制流能把隐私信息的值隐式赋值到其他变量。攻击者可以利用if和while等语句把隐私信息进行隐式传播，窃取一些隐私信息，而这些隐式的数据传播不会被TaintDroid监控。

## 4.3 Under-tainting的形式化描述以及模型

在实际的理论分析中，研究人员一般会用信息流或者控制流的形式描述数据在程序中的传递和依赖关系，然后通建立程序的模型来或者数据传播策略找出程序中的不安全因素。本文主要借助Denning[23]的信息流模型来研究under-tainting问题，Denning的信息流模型表示如下：

FM = <N, P, SC, , > 4-1

在信息流模型中，包含了五个符号，其中N是客体的有限集合，P是有限的进程集，SC是有限的安全类别集，N中的客体默认有一个较低的安全级别L；类结合运算符“”是一个二进制运算符，它满足结合律和交换律。对任意两个操作数的类，“”指定了他们之间任意二进制运算产生的结果属于操作数类。信息流关系符号“”是指两个安全类之间的信息流向。信息流关系符号“”为：当且仅当安全类型A中信息允许流入B时，记为AB。

信息流安全模型FM的安全策略可以表述为：任何操作序列都不会违反关系所规定的信息流时，则认为模型FM是安全的。下面本文就借助FM模型和Mariem Graa[57]中的关系符和信息流符进行under-tainting理论分析，并提出under-tainting的污点标记规则。

与FM模型不同的是，本文把结合运算符“”作为对象污点传播运算的操作符，连接操作符{⇒, , , }的语法定义如下，这里A和B为逻辑公式，x和y为两个变量，Taint()对变量的污点标记操作：

1. A⇒B ：if A then B
2. xy ：从变量x流向y的信息
3. xy ：y赋值到x
4. Taint( x) Taint( y )：污点结合操作

连接符{, , }的满足条件如下：

1. 具有自反性：x为一个变量，x x
2. 具有传递性 ：x，y，z为三个变量，if ( x y ) ( y z ) then x z
3. 具有自反性 ：x为一个变量，xx
4. 具有传递性：x，y，z为三个变量，if ( x y ) ( y z ) then x z
5. 连接符和为非对称的，xy xy
6. 满足交换律：Taint( x) Taint( y )= Taint( y ) Taint( x)
7. 满足结合律：Taint( x) ( Taint( y ) Taint( z )) = ( Taint( x) Taint( y )) Taint( z )

**定义4.1** under-tainting

当x控制依赖条件语句时，x在条件分支中的值改变并且控制语句被污点标记，但x未被污点标记。

在动态污点分析中，引起under-tainting的主要问题是由控制语句导致。当条件语句中存在被标记为污点数据，而依赖控制语句的变量在信息传播过程中没有被标记而造成了一定的信息泄露。针对这一问题的分析，under-tainting的模型可以表示为：

Ismodified(x)ControlDependency( x, condition) Tainted( condition) Tainted( x) 4-2

**定义4.2** Ismodified(x)

Ismodified(x) ≡ (x DataDependency)

其中Is modified(x)表示变量x的值发生了改变， DataDependency赋值给变量x而引起x变化，即x数据依赖于DataDependency。

**定义4.3** ControlDependency( x, condition)

ControlDependency( x, condition) ( condition x)

ControlDependency( x, condition) 可以表示为当x控制依赖控制条件语句时，条件语句的信息流向变量x。对于控制依赖的相关分析本文会在第五章给出详细说明。

**公理 4.1** 如果变量x的信息流向y，x如果被污点标记，y被污点标记的结果为Taint(x)

(x y) ⇒ (Taint(y) Taint(x))

**公理 4.2** 如果变量y赋值到x，则说明有信息流从y到x

(x y) ⇒ (yx)

**公理 4.3** 变量y和z都能引起变量x污点标记，则经过“⊕”运算后的结果成为x污点标记结果

(Taint(x)Taint(y))(Taint(x) Taint(z)) ⇒ (Taint(x) Taint(y) ⊕Taint(z))

**公理4.4** 如果对一个带有污点标记的变量x赋一个常量的值C，则把x的污点标记设置为0x00

x C ⇒ Taint(x) 0x00

从公理4可以看出，该公式主要使用了表3.2中的第一条污点传播规则，对带有污点标记的变量进行赋常量的操作，会清除污点标记，即把污点标记位赋常量值0。

**公理 4.5** 如果一个带有污点标记的变量y和一个常量的经过运算符“⊕”运算后赋值到变量x，那么x的污点标记就是y的污点标记

Taint(x) Taint(y) ⊕Taint(C) Taint(x) Taint(y)

**推论 4.1** 变量x控制依赖于被污点标记条件语句condition，即Taint(conditon)为真，且有Ismodified(x)为真，不管引起x变化的DataDependency有没有被污点标记，变量x都能被污点标记

证明：由控制依赖定义ControlDependency( x, condition) ( condition x)，由公理4.1知

condition x⇒ (Taint(x) Taint(condition))；

由于

Taint(condition) = Taint(condition) (Tainted(DataDependency )Taint(DataDependency))

=(Taint(condition)Tainted(DataDependency))(Tainted(DataDependency )Taint(condition) )

⇒ Taint(x) (Tainted(DataDependency )Taint(condition) ) （公理4.3）

⇒Taint(x)

**推论4.2** 变量x控制依赖于条件语句condition，x的值发生了变化而且Tainted(DataDependency)为真，不管条件语句有没有被污点标记，变量x都能被污点标记

Taint(DataDependency) ⇒Taint(x)

推论2的证明方法同推论1。从TaintDroid中传播语义不难发现，推论2属于一般的赋值操作等污点传播。

**推论3** 污点标记操作Taint()具有自反性

Taint(x) ⇒ Taint(x)

**推论4**变量y和z都能引起变量x污点标记，只要y和z有一个被污点标记，那么变量x也一定被污点标记

(Taint(z)Taint(y)) (Taint(x)Taint(y)) (Taint(x) Taint(z)) ⇒ Taint(x)

证明：已知Taint(x)Taint(y)和Taint(x) Taint(z)为真，那么假设其中Taint(z)为真，那么Taint(z)Taint(y)也一定为真，对于y的情况也一样。

**定理4.1** 满足公理4.1-4.5组成的污点标记系统不会处于under-tainting的状态

证明：利用反证法来证明。假设处于under-tainting情况下的必要条件都满足，即Ismodified(x)为真，ControlDependency( x, condition)为真，Tainted( condition)为真，但是Taint(x)则为假，那么模型4-2得出结果为真。因为ControlDependency(x，condition)为真，由ControlDependency(x，condition)定义知，可以推出condition x。由已知条件和推论4.1知，可以得出Taint(condition) ⇒Taint(x)，因此x被标记。这与Taint(x)为假相矛盾。

## 4.4 Under-tainting问题的解决方案设计

4.3节给出了引起under-tainting的原因，并给出了相应的under-tainting的理论分析，为污点标记的传播规则的指定提供了依据。本节就利用under-tainting理论分析执行控制流中变量x的传播规则。本文使用ContextTaint表示条件语句中存在污点标记的数据，BranchTaken表示在控制语句中的分支语句被执行。而在一般的动态污点标记系统中，变量的污点标记往往是在程序运行时进行污点标记，而遇到像if和while这样的判断语句后，每次只能执行两条分支中的其中一条，而未执行的分支中也存在泄露隐私信息的可能性。例如表4.1中的代码，加入变量z获取的隐私信息值为0，而else分支中的赋值语句未执行，但在对y输出时，仍然可以判断出z一定为0的信息，因此为未执行分支中变化的变量污点标记也极为重要。下面本文就介绍在控制流中变量污点标记的具体规则。这里表示在控制语句引起了变量x值发生了变化；ControlDependency(x,condition)表示x控制依赖条件变量condition；BranchTaken表示得到执行的条件分支。

规则1 如果变量x的值被更改，x控制依赖控制条件并且变量x所在分支语句得到执行，本文用下面如下规则污点标记x。

规则1的说明：变量x控制依赖条件语句condition，并且在执行控制语句中x的值发生了变化，对condition污点标记并且对引起x变化的DataDependence进行污点标记后，经过运算符“”的操作结果是变量x污点标记值。

规则2 如果变量x的值被更改，x控制依赖控制条件并但分支语句未得到执行，本文使用如下规则污点标记变量x。

规则2的说明：变量x控制依赖条件语句condition，并且在未执行的控制语句中x的值发生了变化，对condition污点标记并且对x进行污点标记后的结果是变量x污点标记值。

污点标记规则1证明：首先，本文先假定ControlDependency(x , condition)为真，Ismodified(x) 和BranchTaken也为真。下面来证明Taint(x) (ContextTaint ⊕Taint(DataDependency)成立。由已知条件知，ControlDependency(x , condition)为真，那么condition x，由公理4.1可知，Taint(x) Taint(condition)。由于ContextTaint = Taint(condition)，因此Taint(x) ContextTaint。由于Ismodified(x)为真，可知x DataDependency。使用公理4.1，可以得到Taint(x) Taint( DataDependency )。最后使用公理4.3，可以最终推出Taint(x) (ContextTaint ⊕Taint(DataDependency)为真。因此定义污点标记传播规则1成立。

污点标记传播规则2证明：首先，本文先假定ControlDependency(x , condition)为真，Ismodified(x) 也为真，但BranchTaken为假。由于关系式“”具有自反性，因此有x x，由公理4.1可以得出Taint(x) Taint(x)。由已知条件知，ControlDependency(x , condition)为真，那么condition x，由公理4.1可知，Taint(x) Taint(condition)。由于ContextTaint = Taint(condition)，因此Taint(x) ContextTaint。从公理4.3可以推出，Taint(x) (Taint(x) ⊕ContextTaint)。因此定义污点标记传播规则2成立。

从两个传播规则上可以看出，只要变量x的值在控制分支中发生变化，不管变量x在执行分支还是在未执行分支中都得到标记。当x出现在执行分支中时，则说明x是在显式流中；当x所在分支未得到执行，则说明x在隐式分支中，但变量x的值一旦有所变化并出现在条件分支中，也会导致信息的泄露。根据两条污点标记规则两种情况下的变量x也都会被污点标记。根据制定的污点传播规则，本文对表4.1中的代码段进行分析。从第一行代码知，变量z获取了一个隐私数据，此时z被污点标记。经过第4行的if判断语句后，执行变为两个分支，对于两个分支中的变量x和y分别执行不同的污点传播策略。假如变量z获取的隐私值为0，那么变量x执行的污点传播策略为Taint(x) Taint(z) ⨁Taint(C)，y执行的污点传播策略为Taint( y ) Taint(z) ⨁ Taint(y)，如表4.2所示，从该表中本文可以发现，不管if语句的分支有没有执行，但是依赖控制语句的变量都能得到污点标记。

表4.2 表4.1代码中污点传播示例

|  |  |  |  |
| --- | --- | --- | --- |
| 行号 | 是否为分支 | 分支是否执行 | 执行传播策略后的结果 |
| 4-5 | 是 | 是 | Taint(x) ← Taint(z) ⨁Taint(C) |
| 6-7 | 是 | 否 | Taint(y) ← Taint(z) ⨁Taint(y) |

## 4.5 本章小结

本章首先对TaintDroid建立了几种控制流引起攻击模型，并对这些模型的攻击方式进行详细分析；其次对目前控制流的相关技术进行了详细研究，为解决under-tainting问题打下基础；其次根据信息流模型的相关知识对under-tainting问题建立了模型；最后根据under-tainting模型提出了两个污点传播规则，两条规则分别对应条件语句两条分支，一种是执行分支，另一种是未执行分支。

# 第5章 控制流中的支配关系和控制依赖算法

支配关系由Prosser首先提出：从入口到到箱子j的所有路径都经过箱子i，就认为箱子i支配j[59]。支配关系对程序分析和优化的作用非常大，如程序中的循环分析，消除不可达结点，控制流分析，数据流分析，消除循环冗余，依赖关系计算，以及静态单赋值分析和函数调用等等。针对支配关系的求解方法，主要是结合图论的知识，利用图中的可达性来判断结点之间的支配关系；除可达性分析外，迭代法求解支配关系的方法也比较多。

可达性算法是通过不断移除入口结点到某个结点v上路径的一些点，如果在移除某一点后v变得不可达，就说明该结点支配v结点。Lowry和Medlock[60]提出了基于图的可达性算法。而Purdom和Moore[61]把图转化为树，然后通过移除结点计算可达性，从一定程度上降低了算法复杂度。

迭代法主要根据程中的基本块之间的传递关系，利用控制流图求解支配关系。Allen[62]利用数据流等式描述支配关系的问题，利用迭代法求解支配关系。Hecht和Ullman[63]对图进行逆向分析，采用后续迭代求解支配关系。效率比较高的支配关系求解算法还有Tarjan[64]的基于深度优先搜索的联合查找算法和Bunchsbaum[65]分治法，时间复杂度都是O(NlogN)。

相关研究表明，控制流图上的支配和依赖关系是程序静态分析不可或缺的基本信息。本章主要对程序的支配关系和依赖关系分析求解，为系统设计和实现打下理论基础。

## 5.1 控制流图介绍

根据程序执行语句的支配关系，Jeanne[66]又提出了程序依赖图的概念。程序依赖图分为两个部分，一是数据依赖图，另一个为控制依赖图。程序数据流仅仅描述程序中数据之间的依赖关系；控制流则描述程序中在条件分支中控制依赖依赖关系。

控制流图是描述程序所有带有控制语句或者循环的执行路径，它能直观的描述出程序的跳转分支。在介绍控制流图前，本文先对有向图进行定义。

**定义5.1 有向图**

在图中，结点之间的边有方向区分，表示结点之间的可达性。

有向图G=<V,E>，其中V表示图中的结点集合，E表示图中边的集合，且图中各边都有方向，对任意两个结点v,w∈V，且v≠w，都有<v , w>∈E且<v , w> ≠ <w , v>。

**定义5.2 控制流图（Control Flow Graph，CFG）**

控制流图是以基本块(basic block)为结点的有向图。CFG=<V , E , entry ,exit>为控制流图，满足以下条件：entry为其入口结点，即程序开始；exit表示出口结点，即程序结束； vV(v表示程序基本块)； e = (ni,nj)E，其中ni,njN， e表示基本块ni至nj的边。同时控制流图中任何一个结点v，都至少有一条路径从entry(程序入口)到n也至少存在一条路径从n到exit(程序出口)。

**定义5.3 基本块（BB，Basic Block）**

基本块是指程序中具有唯一入口和唯一出口的指令序列，入口为指令的首指令，出口表示指令序列的最后一条指令。

在控制流图中，每个基本块都是程序语句集合，基本块之间的有向边表示一条执行路径。程序控制流图只能有唯一的入口和出口，分别用entry和exit表示。程序中存在一条从entry直接到exit路径，而其他有向边表示控制流图基本块之间的控制和传递关系。本文CFGV表示控制流图的所有结点集，用p(v,w)表示控制流图中结点v到结点w的路径。e(x,y)表示结点x到结点y的边，在边(x,y)上出现的结点y是x结点的后继，用succ(x)表示结点x的后继结点集合；在边e(x,y)上出现的结点x是y结点的前驱，前驱结点集可以类似表示为pred(y)。如果一个结点出现多个后继结点，则该结点是一个分支结点；如果一个结点出现了多个先驱，则认为这个结点是一个聚合结点。图5.1表示控制一个程序的控制流图。



图5.1 程序控制流图

从图5.1中得知，有结点1至结点2的边e(1,2)，其中2是1的后继，1是2的前驱；而结点1有两个后继，说明结点1为分支结点；结点4有两个前驱，分别为2、3和9，说明4为一个聚合结点。

## 5.2 控制流图中的支配关系

**定义5.4 支配关系(Dominator)**

如果所有从入口到结点v的路径都包含结点w，就说结点w支配结点v。对于控制流图上任意两个结点w,v CFGV，如果w支配v，记作 w dom v，如果w !dom v，表示结点w不支配结点v。用dom(v)表示支配v结点的集合，v的支配结点公式表示如5-1：

dom(v) = { v } ( w∈pred(v) dom(w) ) 5-1

任何结点都支配结点本身, v dom (v)，满足自反性。

对于三个结点x，y，z如果x dom y并且y dom z知 x dom z，如果x dom z 并且y dom z知x dom y 或者y dom x，满足传递性。

对于两个结点x，y且x ≠ y，如果x dom y则y !dom x，满足反对称性

entry结点支配控制图中的每一个结点，如公式5-2所示：

∀ v CFGV : entry dom(v) 5-2

根据公式5-1和5-2本文可以得出支配关系的求解算法：入口结点entry的支配集合dom(entry)为其自身，可以把入口结点的支配集作为迭代的初始条件。其他非entry结点支配结点集合的初始支配集为全部结点，然后根据公式5-2，,利用自底而上的方法不断迭代求解，如果在某次迭代中结点的支配点集没有发生变化，则求解完成。深度优先对控制流图的支配关系求解有很大意义，它可以求出所有结点的后继和前驱结点(前驱结点可以逆序访问控制流图)。算法5.1描述控制流图的深度优先算法。

**算法5.1 深度优先遍历**

for each v CFGV

mark v as “unvisited”

Dv { entry }

v = entry //访问结点从第一个结点开始

DFS(v) {

mark v as “visited”

for each w succ( v) do

if w is “unvisited” then

Dv dv { w}

call DFS(w) //对未访问的结点继续深度遍历

end if

end for

}

根据支配深度优先，求解支配结点算法描述如下，

**算法5.2** 支配结点求解算法

dom( entry ) { entry } //对entry的初始结点赋值

Dv DFS(CFG) //深度优先遍历控制流图，Dv表示遍历的结点集合

for each v Dv and v ≠ entry

dom(v) CFGV //对非入口结点的支配结点集合赋值，全部赋值为CFGV

do {

changed false

for each v Dv // 深度优先生成结点集合求解

if v = entry then continue ;

olddom dom(v) // 保存v结点当前支配集合

dom(v) = { v } ( w∈pred(v) dom(w) ) // 对这些结点的支配结点不断迭代

changed dom(v) ≠ olddom // 直至迭代结果不发生变化

end for

}while ( changed ) //查看changed的值是否发生变化

复杂度分析：在最好情况下，控制流图中的结点不存在环形连接，则两次迭代就可以得到结点的支配集合，此时则为线性复杂度。而在在最坏的条件下，每次迭代只有一个结点收敛，此时复杂度为O(N2)[59]。因此，程序中的循环语句对求解支配结点算法影响非常大。

表5.1 图5.1中结点的支配点求解过程

|  |  |  |
| --- | --- | --- |
| 结点 | 迭代过程 | 结点支配集 |
| entry | dom(entry) = dom(entry) | entry |
| 1 | dom(1) = {1}∪ dom(entry） | entry,1 |
| 2 | dom(2) = {2} ∪ dom(1) | entry,1,2 |
| 3 | dom(3) = {3} ∪ dom(1) | entry,1,3 |
| 4 | dom(4) = {4} ∪ (dom(2)∩dom(3)∩dom(9)) | entry,1,4 |
| 5 | dom(5) = {5} ∪ (dom(4)∩dom(10)) | entry,1,4,5 |
| 6 | dom(6) = {6}∪ (dom(5)∩dom(7)) | entry,1,4,5,6 |
| 7 | dom(7) = {7}∪ dom(5) | entry,1,4,5,7 |
| 8 | dom(8) = {8}∪ (dom(6)∩dom(10)) | entry,1,4,5,6,8 |
| 9 | dom(9) = {9} ∪ dom(8) | entry,1,4,5,6,8,9 |
| 10 | dom(10) ={10} ∪dom(8) | entry,1,4,5,6,8,10 |

**定理5.1** 在深度优先生成树上，第n层结点会在n次迭代后收敛。

证明：下面用归纳法证明这一定理，根据支配关系定义知，在控制流图的深度优先的生成树上，入口结点在第一次迭代时即可收敛，入口结点的后继结点支配集一定只包含自身和入口结点，因此入口结点的后继结点在第二次迭代时即可收敛[67]。假设在深度优先生成树上，k层结点会在k次收敛。根据支配关系定义知，在第k+1次迭代后，k+1层结点的所有前驱的支配结点集合中都包含了收敛的k层结点的支配关系。因此在第k+1次迭代后，k+1层结点也一定收敛。定理得证。

因此，在深度遍历优先生成树上求解支配结点的最坏条件下，一次只收敛一个结点，而每次迭代需要时间复杂度为O(N)，最坏的时间复杂度O(N2)。

**定义5.5 严格支配关系(Strict Dominators)**

如果所有从入口到结点v的路径都包含结点w但是不包含v本身，就说结点w支配结点v。本文用dom!表示严格支配关系，w dom! v表示结点w严格支配结点v，w !dom! v表示结点w不严格支配结点v。结点v的严格支配集用dom!(v)表示，公式表示如下：

dom!(v) = dom (v) − { v} 5-3

对于三个两两不同结点x，y，z如果x dom! y并且y dom! z知 x dom! z，如果x dom! z 并且y dom! z知x dom! y 或者y dom! x，满足传递性。

对于两个结点x，y，如果x dom! y则y !dom! x，满足反对称性。

因为，dom!(v)=dom(v)-v,求解严格支配结点的算法在求解支配结点的基础上移除结点本身，然后该支配结点集为v的严格支配结点，该算法就不赘述。

**定义5 直接支配关系(Immediate Dominator)**

结点v的直接支配结点是从入口结点至自身的路径上的最靠近自身的一个严格支配结点。本文用idom表示直接支配关系符，v idom w表示结点v直接支配w；v !idom w表示结点v不直接支配w，结点v的直接支配关系结点如5-4表示

d idom v d dom! v ∧ (∀ w | w dom! v)[w dom d ] 5-4

对于两个结点x，y，如果x idom y则y !idom x，满足反对称性。

求解直接支配结点的算法思想：对于某个结点v，遍历该结点的严格支配集中的所有结点，如果某个结点w的支配集满足dom(w) = dom(v)-{v}，说明结点w就是结点v的直接支配关系结点，即w idom v。根据该思想求解结点的直接支配结点算法如下：

**算法5.3** 直接支配结点算法

for ∀ x CFGV // 控制流图中的任意结点x

if x ==entry // 判断x是否为入口结点

then continue;

dom(x)dom(x)-{ x };//在dom[x]中移除自身结点

for ∀ y dom(x) //x除自身外任意支配结点

if dom(y) == dom(x)

idom(x) idom(x) { y };//把结点y并入idom

break

end if

end for

dom(x) dom(x) { x };

end for

该算法是在求解支配结点基础上求解，时间复杂度依赖支配结点算法，因此复杂度为O(N2)。根据算法2可以得出流程图5.1的直接支配结点如表5.2所示

表5.2 控制流图5.1的直接支配结点

|  |  |  |
| --- | --- | --- |
| 结点 | 迭代过程 | 直接支配结点集 |
| 1 | idom(1) = dom(entry) - {1} | entry |
| 2 | idom(2) = dom(1) -{2} | 1 |
| 3 | idom(3) = dom(1) - {3} | 1 |
| 4 | idom(4) = dom(1) - {4} | 1 |
| 5 | idom(5) = dom(4) - {5} | 4 |
| 6 | idom(6) = dom(5) - {6} | 5 |
| 7 | idom(7) = dom(5) - {7} | 5 |
| 8 | idom(8) = dom(6) - {8} | 6 |
| 9 | idom(9) = dom(8) - {9} | 8 |
| 10 | idom(10) = dom(8) - {10} | 8 |

**定理5.2** 结点v的直接支配结点唯一

证明：利用反证法证明。假设c idom v并且d idom v，因此有c ≠ v 并且 d≠ v，c dom! v 且d dom! v。由idom的定义知：d dom ! v ∧ (∀ w | w dom! v)[w dom d ]，由此可以得出c dom d且d dom c，但是由于dom是反对称的，除非c=d，因此可以得出结论结点v的直接支配结点是唯一的。定理得证。

**定义5.7 支配树 (dominator tree)**

除了入口结点，把控制流图中的每一个结点v和v的直接直接支配结点idom(v)连接成为一条边，然后以entry为根结点形成树叫支配树。

支配树可以把控制流图上的支配和直接支配关系更加简洁的表示出来，它直接支配孩子结点，支配所有子孙结点。由表5.2知，图5.1的支配树如下



图 5.3 图5.1的支配树

**定义5.8** **回边(back edge)**

在控制流图中，结点x到结点y存在一条路径p(x , y)，同时也存在一条结点y到结点x的边e(y , x)并且满足x dom y，称e(y , x)为一条的回边。

从回边的定义不难发现，回边可以构成控制流图中的循环结构。因此，判断控制流图中是否存在循环结构，可以通过查看结点是否同时存满足以下两个条件：

1. 有唯一的入口结点，该入口结点支配循环中所有的结点
2. 存在一条进入入口结点的回边

给出一个回边e(y,x)，可以通过自底向上的递归算法求出循环中所有结点，求解式如下：

Ev( loop(e(y,x)) ) = { x } + {z | p(z , y) x p(z , y)} 5-5

其中Ev( loop(e(y,x)) )表示由回边e(y,x)构成的循环结构中的结点集合，{z | p(z , y) x p(z , y)}存在结点z到达结点y路径，并且结点x不在该路径上。根据5-5式的描述，本文可以求出一条回边上循环上的所有结点。算法描述如下：

**算法5.4** 回边中结点求解算法

//给定一个回边p(D,N)，求出循环中结点

Stack nul //定义一个空栈

Loop {D} // Loop表示为循环中的结点集合，初始化Loop为结点D

Insert(m) // 定义insert函数

if m Loop then // 判断结点m是否在Loop中

Loop Loop {m} //把结点m加入Loop中

push m onto Stack //把m压入栈中

end if

end Insert //函数结束

Insert(N) //结点N进栈，并进行压栈

While Stack nul do //如果栈非空，进行以下操作

pop m // 让栈顶结点m出栈

for each p pred(m) do //对结点m的所有前驱p

Insert(p) //结点p进栈，并进行压栈

end for

end while

由回边的定义知，图5.1中p(9 , 4)就是一个回边(存在路径p(4,9)且结点4 dom 9)， 根据算法回边结点算法，可以求出回边p(9,4)中所有结点，求解过程以及结果如表5.3所示

表5.3图5.1中回边p(9,4)中包含的结点

|  |  |  |
| --- | --- | --- |
| 步骤 | Stack中包含的结点 | Loop中的结点 |
| 1 | nul | 4 |
| 2 | 9 | 4,9 |
| 3 | 8 | 4,9,8 |
| 4 | 6 | 4,9,8,6 |
| 5 | 7 | 4,9,8,6,7 |
| 6 | 5 | 4,9,8,6,7,5 |
| 7 | 4 | 4,9,8,6,7,5 |
| 8 | Nul | 4,9,8,6,7,5 |

**定义5.9 后继支配关系(post Dominator )**

如果结点v到exit的所有路径都包含结点u，则v被u后继支配，即u pdom v。同支配关系类似，后支配关系也具有自反性和传递性。如果x后继支配y并且有y x，此时x严格后支配y(x pdom! y)。

如果结点m和结点n满足下面条件，就说明m是结点n的直接后继支配

m ≠ n; m  pdom n;  ∀d ≠ n.    d  pdom n  ⇒ d  pdom m

同直接支配关系相同，每一个结点仅有一个直接后继支配结点。在求解后继支配关系时，求解方式等同于控制流的支配关系求解，仅仅是把控制流图中的所有边进行变向根据后继支配关系。

**定义5.10 后继支配关系树**(pdtree)

后继支配关系树是以exit为根结点，树中任意一结点都被它的祖先后继支配，被父结点直接后继支配。

根据支配关系求解算法，得出图5.1的直接后继支配关系为exit ipdom entry，4 ipdom 1，4 ipdom 2，4 ipdom 3，exit ipdom 4，6 ipdom 5，8 ipdom 6，6 ipdom 7，9 ipdom 8，4 ipdom 9，8 ipdom 10。根据直接支配关系得到的后继支配树如图5.4所示。



图 5.4 图5.1对应的后继支配树

## 5.3 控制依赖关系算法设计

Jeanne [66]首先提出了程序依赖图。程序依赖图分为两个部分，一是数据依赖图，另一个为控制依赖图。程序数据流仅仅描述程序中数据之间的依赖关系；控制流则描述程序中在条件分支中控制依赖依赖关系，结点的控制依赖关系往往从控制流图中求得。控制依赖关系能描述条件语句与执行语句之间的依赖关系。它是程序依赖图的一部分。控制依赖关系可以很直观的描述那些依赖控制语句的执行语句，对依赖条件语句的隐私信息标记是本文设计系统的前提条件之一。下面本文将会对控制依赖关系进行分析求解。

**定义5.11** 控制依赖

假设CFG表示一个控制流程图，x和y是CFG上的结点，y控制依赖于x当且仅当：

1. 存在一条x到y的执行路径p(x,y)；
2. 路径p(x,y)上的任意结点n(除了结点x外)，n到出口结点exit的任意路径一定经过y，也就是说结点y后继支配结点n(y pdom n)；
3. y一定不严格后继支配结点x(y !pdom x)(y ≠ x)

从定义可知，如果结点y控制依赖结点x，x一定会存在两个出口，进而使得从一个出口出发后要么导致y结点中语句执行，要么导致y结点语句不执行。从直观上说，在x到y之间一定是一条控制流程的语句，进而使得除了x到y之外还会跳转到其他地方的执行路径；但x和y之间一定不存在其他的控制语句，因为x和y之间一旦还存在其他控制语句就会改变到y的执行路径。例如图5.5中码段，语句5控制依赖于条件语句4，但是语句6和4之间没有直接的控制依赖关系，因为4和6之间存在语句5，而5是一条控制语句。但另一方面，语句6、7、8都控制依赖语句5，语句7严格后继支配语句6，语句8严格后继支配语句7。

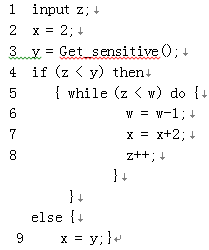


图5.5程序代码段图

控制依赖关系对程序静态分析以及动态执行时的优化都有很重要的意义，因此，对控制依赖图的生成以及求解控制流图上控制依赖结点的算法复杂度要求都比较高。然而，直接根据控制流图和控制依赖定义生成控制依赖算法复杂度非常高。根据定义可生成控制依赖关系算法，假设CFGV表示控制流所有结点集，p代表结点之间的有向路径，cd(x)代表结点x的控制依赖的结点，算法介绍如下：

**算法5.4** 根据定义生成控制依赖关系算法

for each x CFGV

for each y CFGV //找出控制流图中任意两个结点组合

if x pdom!( y ) // 判断x是否严格后继支配于y

break; // 执行条件为true ，跳出

else

for each p p(x , y)

// p(x , y)表示结点x至y的所有路径，p是其中一条

for each v p and v x and v y

//找出路径p上不同于x和y的结点

if v pdom!( y )

//判断v是否为y的严格后继支配

cd(x) cd(x) { y }

end if

end for

end for

end if

end for

end for

此算法需要控制流图的任意两个结点组合，这就需要N2次循环，而求解后继支配算法复杂度为O(N2)(参考直接支配结点算法)；因为结点x至y存在多条路径，因此最坏情况下时间复杂度为O(N)，每条路径又存在多个结点，而且还要判断结点的后继支配关系，时间复杂度就为O(N)\*(O(N)\*O(N2))；综上所述，根据控制依赖定义生成控制依赖的算法复杂度为O(N2)\*(O(N2)+(O(N)\*(O(N)\*O(N2)))=O(N6)。由此可见，这种算法复杂度非常高。 为了能有效降低生成控制依赖关系图的时间复杂度，本文先给出以下定理，该定理为控制依赖算法的基础。

**定理5.3** 对于控制流图上任意一条边( x , y )，w是x，y在后继支配关系树上最近的共同祖先结点，则w为x，或者为x在后继支配关系树中的父结点。

证明： 假设w是x在后继支配关系树上的父结点。由此可知，w肯定不是y结点。首先本文来证明w后继支配y结点。本文采用反证法证明，证明方法如下：假设结点y不被w后继支配，那么存在一条从y到exit的路径p1，且p1不包含结点w；如果把边( x , y )加入到p1，则就存在一条从x到exit且不经过w的路径p2；这与x被w后继支配相矛盾。因此，结点w后继支配y结点。因为结点y被w后继支配，在后继支配关系树上，结点w必是y结点的祖先。因此，结点x被w后继支配，结点y也被w后继支配。由此可以得出，x、y在后继支配树中的最近的共同祖先点要么是x,要么是w。定理得证。

根据后继支配关系树算法和定理3，可以得到结点的控制依赖关系生成算法。其中变量x、y、w分别表示控制流图上的结点，e( x , y )表示一条由结点x、y形成的边；CFG表示控制流图，cd(x)表示v结点的控制依赖关系集合。控制依赖关系的算法如下：

**算法5.5** 优化后的控制依赖关系算法

E nul

cd(x) nul

for each ( x , y ) CFG // 找出控制流图上的边

if (y !pdom x ) in pdtree

//在后继支配树中,结点y的后继不是x的前驱

E E { ( x , y ) } //满足以上条件的边并入集合E中

end if

end for

for each ( x , y ) E // 对集合E中的路径进行循环操作

find w and w=closest( pred(( x , y ) in pdtree)

//找出后继支配关系树中x和y的最近小公共祖先

if w=parent (x)

for each v p(w , y) and v w in pdtree

//如果w为x的父结点，对于所有在路径p(w , y)上的结点(不包括w)都控制依赖于x

CD(x) CD(x) { v }

end for

end if

if w = x

for each v p(w , y) and v w in pdtree

//如果w为x本身，对于所有在路径p(w , y)上的结点都控制依赖于x

CD(x) CD(x) { v }

end for

end if

end for

针对该控制依赖算法，算法复杂度最高的步骤为对边进行循环操作，算法复杂度为O(n)。然后找出路径( x , y )的后继支配关系树上最近公共祖先，而后继支配关系树的高度不会超过N2，因此该算法的时间复杂度为O(N2)。根据控制依赖算法，可以得出图5.1中结点的依赖关系，如表5.4所示。

表5.4 图5.1中点的依赖集合

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V  E | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| (1,2) |  | **1** |  |  |  |  |  |  |  |  |
| (1,3) |  |  | **1** |  |  |  |  |  |  |  |
| (4,5) |  |  |  | **1** | **1** | **1** |  | **1** | **1** |  |
| (5,7) |  |  |  |  |  |  | **1** |  |  |  |
| (8,10) |  |  |  |  |  |  |  | **1** |  | **1** |
| (10,5) |  |  |  |  | **1** | **1** |  |  |  |  |

## 5.4 污点标记算法设计

结合第四章的两条污点传播策略和本章的控制依赖算法，本文提出了污点标记算法此算法主要用来解决控制语句中的under-tainting问题。污点标记算法用二进制代码的控制流图作为输入，该流程图的结点是用指令组成的基本块。使用控制依赖算法计算出依赖控制语句的结点，用Depend\_ List表示存储具有依赖关系结点的((x,y) ∈Depend\_ List表示结点x控制依赖于y)数组。对Depend\_ List数组中的元素进行条件语句判断，然后使用污点的传播规则对依赖条件语句的变量进行污点标记，算法的表示如下。

**算法5.6**  污点标记算法

输入：二进制代码的控制流图

输出：被污点标记的变量

for each x CFGV

or each y CFGV

Dependency\_List ← 控制依赖算法(CFG)

While (x, y) Dependency\_List **do //** 对属于控制依赖的条件边循环

if y.tflag // 判断y中是否为污点标记情况  
Set\_Taint(x) //对结点x执行污点标记操作

end while

算法复杂度分析：本算法主要依赖控制依赖算法，控制依赖算法的时间复杂度为O(N2)，而对于污点标记操作中，主要是对Dependency\_Lis进行循环，循环次数小于N次，因此该算法时间复杂度不超过O(N2)。由表5.4知，CD(1)={2,3}，根据污点标记算法，e(1,2)和e(1,3)都属于Dependency\_List，如果判断结点1里存在污点变量，则Set\_Taint(2,3)。

## 5.5 本章小结

本章首先介绍控制流图概念，分析了控制流图中结点间的支配关系；其次根据结点支配关系的定义生成了一些支配关系算法，并对算法复杂度进行了分析；然后对控制流图中结点控制依赖关系进行了分析，并优化了由定义生成的控制依赖算法；最后结合第四章中的污点传播策略提供了污点标记算法，对算法复杂度也进行了相应的分析。

# 第6章 系统设计和实验

## 6.1 TaintDroid控制流监控模块设计

由第三章的TaintDroid系统分析知，TaintDroid主要分为四个模块：(1) 虚拟机层的变量跟踪；(2) 进程通信级的消息跟踪；(3)存储级别的文件模块信息；(4)本地代码级别的消息跟踪。为解决TaintDroid中控制流引起的under-tainting问题，本文主要对Dalvik虚拟机层解释代码的污点传播语义进行扩展。在Dalvik虚拟机层，添加一个控制流信息流监控模块(Control Flow Monitoring Module，CFMM)，如图6.1所示。



图6.1 TaintDroid改进设计

根据对Android应用程序的解析，CFMM的功能主要分为装载时的静态分析和运行时的动态分析。

装载时的静态分析：

1. 为每一个方法创建一个控制流程图，分析控制流图中的控制分支结构；
2. 根据控制流图， 计算出控制依赖条件语句的基本块

运行时动态分析主要进行如下操作：

1. 创建一个数组，包含了所有带有污点标记条件语句基本块
2. 通过条件语句依赖关系集合，找出依赖控制语句的变化变量
3. 使用污点标记规则对存在于控制语句中的变化变量进行污点标记
4. 通过设置隐私信息的传播提示，提醒用户隐私信息的传播情况，并对该信息进行日志记录

根据CFMM模块在静态和动态时的功能，图6.2详细描述了CFMM模块中各个子模块。



图6.2 CFMM模块框架

## 6.2 CFMM模块实现思路

在静态分析中，需要对dex文件进行反编译，本文使用ddx工具，它可以把应用的dex文件反编译一种中间语言ddx，通过对ddx分析，然后对依赖于带有污点标记的控制语句的变量进行分析，找出应该被标记的变量。因此本文利用ddx指令来构造基本块，再利用基本块语句信息生成控制流图。ddx指令基本块的划分要确定指令首指令，也就是基本块的入口，确定规则如下:

1. 指令序列的第一条指令为首指令；
2. 若指令为跳转语句，则该跳转指令的跳转地址所在的指令是一个首指令，该跳转指令之后的指令也是首指令。
3. 方法入口指令为首指令。

针对CFMM模块的整体实现思路，如图6.3所示。



图6.3 CFMM模块实现思路

根据CFMM实现思路，本文使用python工具对ddx反编译后的指令进行分析，并利用pydot工具生成基本块的控制流图，核心代码如(部分)下：

for line in infile:

if line.startswith('.end method'): #读取每一个方法的指令，直到方法结束

break

lines.append(line)

def findExit(lines): #找出基本块所在行号

#省略部分#

return return\_blocks

exitBlocks = findExit(lines)#找到基本块出口所在的行

def switchBlock(to): #求下一个结点

global block

if not was\_goto:

graph.add\_edge(pydot.Edge(block, to))

#把当前结点后继形成一条边

exitBlock()

block = to

def exitBlock():

node.set\_label("Node\_"+"%d"%Node\_i)

#当前基本块建立完成，为基本块进行标记

def addEdgeTo(to,label='jmp'): #添加边标记，边的默认参数为’jmp’

if to in exitBlocks:

tmp = gensym()

node = pydot.Node(tmp, label=as\_html(exitBlocks[to]))

graph.add\_node(node)

to = tmp

graph.add\_edge(pydot.Edge(block, to, taillabel=label))

#画出block至to的边

switchBlock('entry')#从entry结点开始画图

graph.write(out, format=extn) #画流程图

生成控制流图后，为了方便对结点间支配集合的计算，本文把用python代码求出的基本块用数字编号标记，然后把基本块的编号和对应的指令保存到文件中，利用编号求解支配关系以及控制依赖关系集合。这样仅仅需要在开始计算时建立基本块指令内容数组，然后利用整数数组表示结点间边，用一个整数编号对应整个指令块，大大简化了程序运行时开辟的内存空间。指令基本块和求解控制依赖相关核心代码如下：

tyedef  struct Basic\_Block //定义基本块

{

        int num ; //基本块编号

        bool cflag;//是否为控制块

        bool Tflag;//是否带有污点标记

char \*\*instr;//保存指令的内容

}BB;

BB \*array ; //维护指令基本块的数组

class DominatorGraph { //支配结点类

private:

int n; //控制流图结点数

int narcs; //控制流图边数

int onarcs; //控制流图初始边数

int source;

typedef struct {

union {int \*label2post; int \*label2pre;};

union {int \*post2label; int \*pre2label;};

int next;

int \*parent;

} DFSParams;//定义深度优先访问

typedef union { //结点类型

int value;

int \*ptr;

} intptr;

intptr \*first\_in; //first\_in[v]: 结点v的第一个出边

intptr \*first\_out; //first\_out[v]:结点v的第一个出边

int \*in\_arcs; //入边数

int \*out\_arcs; //出边树

inline void getOutBounds (int v, int \* &start, int \* &stop) const {}

inline void getInBounds (int v, int \*&start, int \*&stop) const {}

inline int \*getFirstIn(int v) const {return first\_in[v].ptr;} //前驱

inline int \*getBoundIn(int v) const {return first\_in[v+1].ptr;} //后继

inline int intersect (int v1, int v2, int \*dom) { }

//后继支配树中最近祖先

void buildGraph (int \_nvertices, int \_narcs, int \_source, int \*arclist, bool simplify); //构造控制流图

void readDimacs (const char \*filename, bool reverse, bool simplify);

//从文件中读取结点

~DominatorGraph() {deleteAll();}

int postDFS (int v, int \*label2post, int \*post2label);

void idfs (int r, int \*idom); //深度遍历求解直接支配结点

void dependence\_list(int v , int \*cd);// 求结点v的控制依赖集合

void detect(BB bb, char \* var[]); //根据基本块内容查找变化变量

};

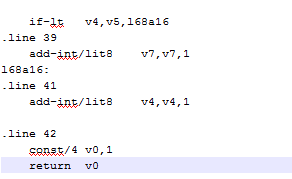


图 6.4 程序的字节码

图6.4为用ddx反编译后的一段if语句字节码。if语句中对寄存器v4和v5的值进行比较，如果比较结果为真，那么.line 39的分支就会执行，因此v7的值就加1。如果比较结果为假，则v4寄存器中的值就会加1。图6.5是本文使用控制流图算法控制流效果图。由图知，控制块中的cflag和Tflag初始化为真。控制流图中的结点集合CFGV={BB(1),BB(2), BB(3) , BB(4)}；控制流图中所有的边为



图 6.5 图6.4的控制流图

{(BB(1),BB(2)) ,(BB(1),BB(3)) , (BB(2),BB(4)) , (BB(3),BB(4))}，通过控制依赖算法生成一个控制依赖的列表DepList(BB(1)) = { BB(2) , BB(3) }；根据污点标记算法，对基本块2和3的Tflag进行进行设置，BB(2) .Tflag = BB(3).Tflag = ture。

对于基本块里可能出现多个变量，而污点标记算法仅仅只对发生变化的变量进行污点标记操作。本文对变化变量的检测主要根据执行指令的类型以及寄存器的位置进行判断，例如move，move-result，add等指令涉及到很寄存器，则把出现在指令里的目标寄存器设置为变化变量，如move v2 v3操作，v2为目标寄存器，那么v2对应的变量就会被设置成污点。

## 6.3 验证分析

根据污点传播策略以及污点标记算法的分析，本节就使用攻击模型中代码仿真验证CFMM模块的可行性。其中Android应用代码见图6.6，其中Get\_IMEI()函数表示获取设备的IMEI号码，而getY()函数则是把IMEI号经过控制语句赋值到其他变量。sendByHttp()函数表示建立了一个通信连接，然后发送信息到百度输入框。

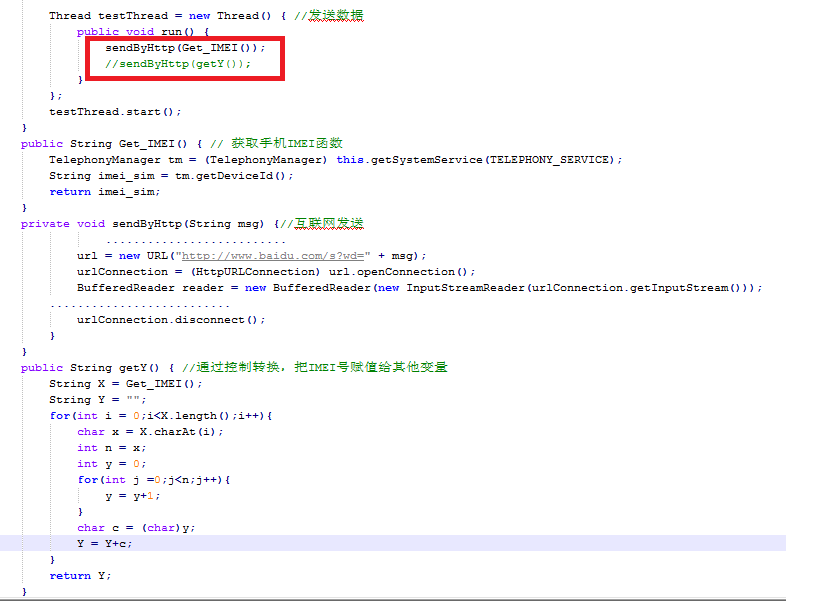


图6.6 Android应用程序源代码

示例代码在Android模拟器上编译后运行如图6.7，红色区域显示为IMEI号，模拟器的IMEI号为000000000000000。使用TaintDroid对应用程序进行监控，监控分两次进行， sendByHttp()函数以函数Get\_IMEI()和getY()的返回值作为参数。通过两次执行应用程序，记录得到的监控结果。如图6.8的区域1和2分别两次执行的监控结果。1区域表示TaintDroid只对sendByHttp( Get\_IMEI() )

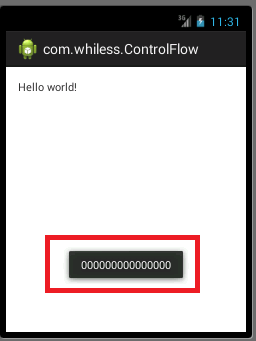


图 6.7代码运行图

操作监控，得到Taintlog值为“ received data with tag 0x400 ”；而在只运行sendByHttp( getY)函数下，TaintDroid监控日志为空，图6.8中的区域2所示。从该实验得出，TaintDroid的确存在污点标记不足的问题。

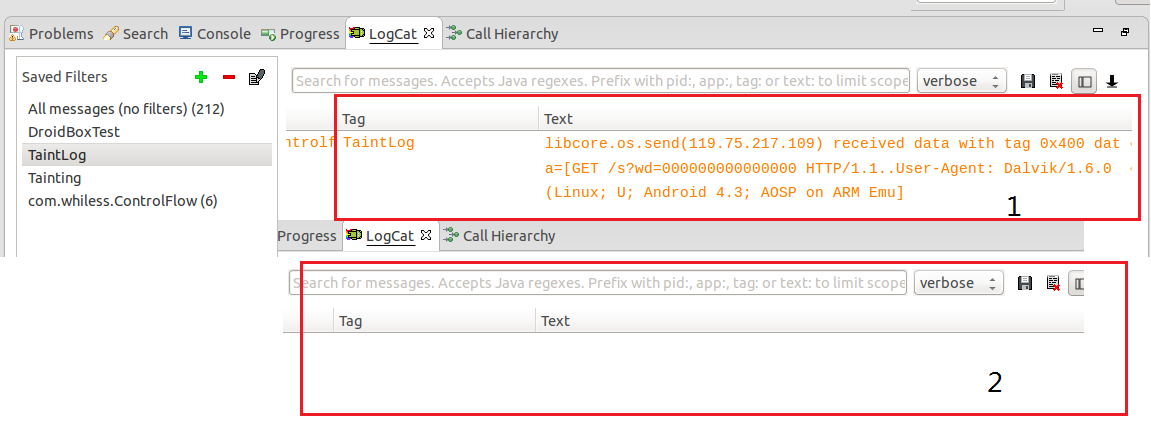


图 6.8 TaintDroid监控污点日志

为了验证CFMM模块的可行性，本节对CFMM模块进行仿真验证。以图6.9的源代码生成的应用为例，然后对应用中的控制流进行分析。首先，通过ddx反编译，得到应用dex二进制文件的反汇编代码，反汇编后的getY()方法的指令如图6.9所示。在反编译后的文件中，可以找出getY()方法中存在两个控制分支结点，分别为” if-lt v3,v8,l68a48”和” if-lt v4,v5,l68a84 ”，而这两个控制语句对应两个for循环。通过对反编译后的指令基本块进行划分，然后生成控制流图，如图6.10所示。

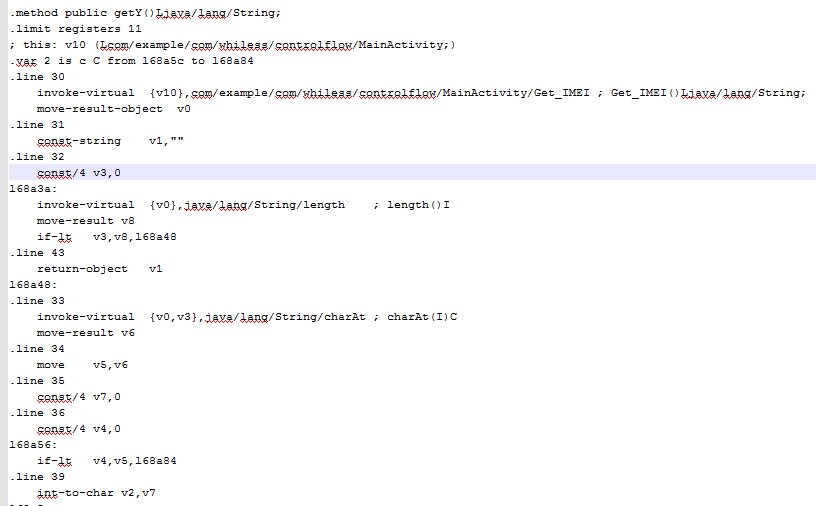


图6.9 Android应用反汇编代码

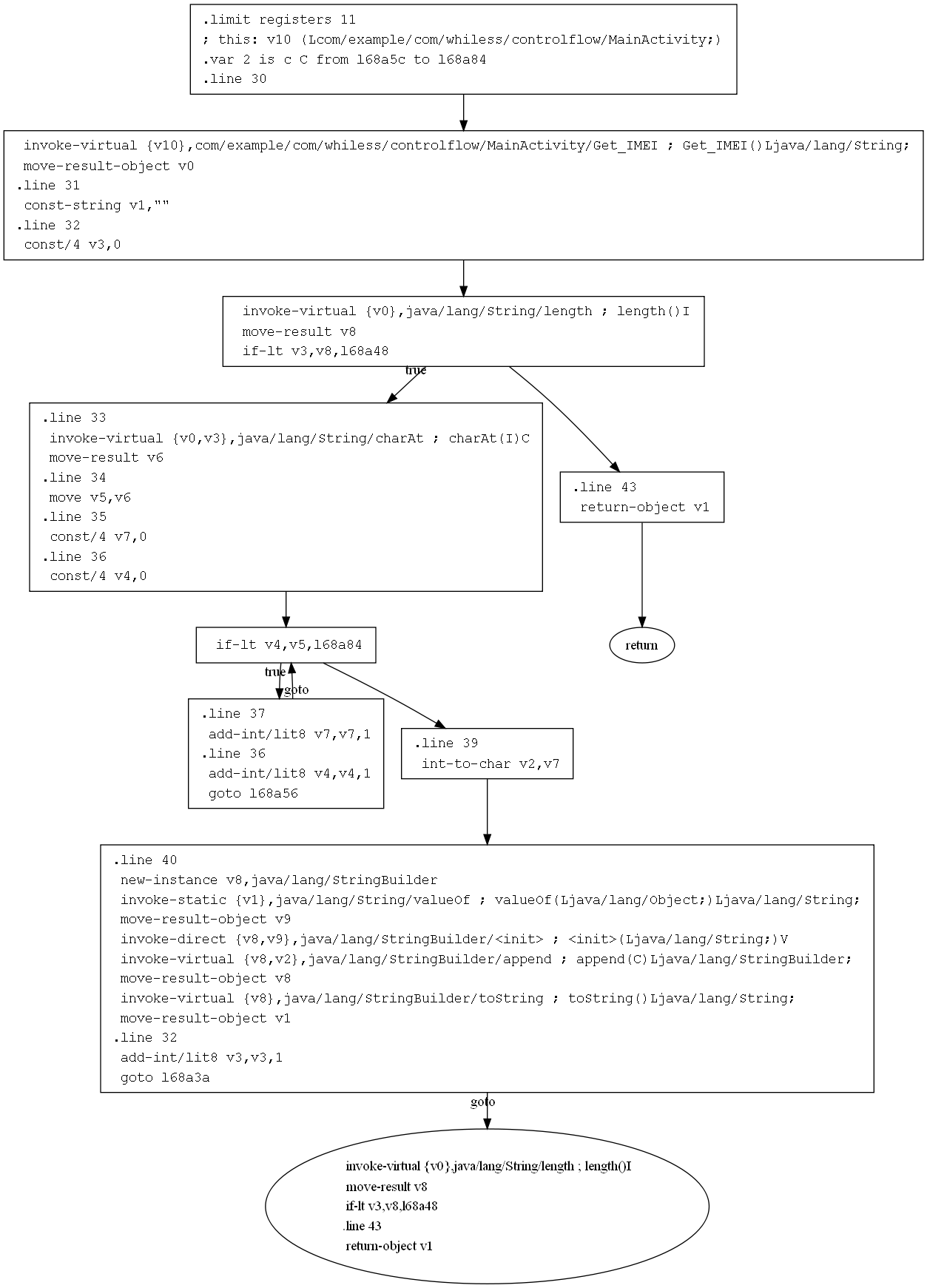


图6.10 反编译后getY()方法指令控制流图

基本块划分后，对每一个基本块都进行编号，建立基本块与编号的一一对应关系，然后保存在文件中。图6.11和6.12表示基本块建立结点号后的控制流图以及对应的结点关系。

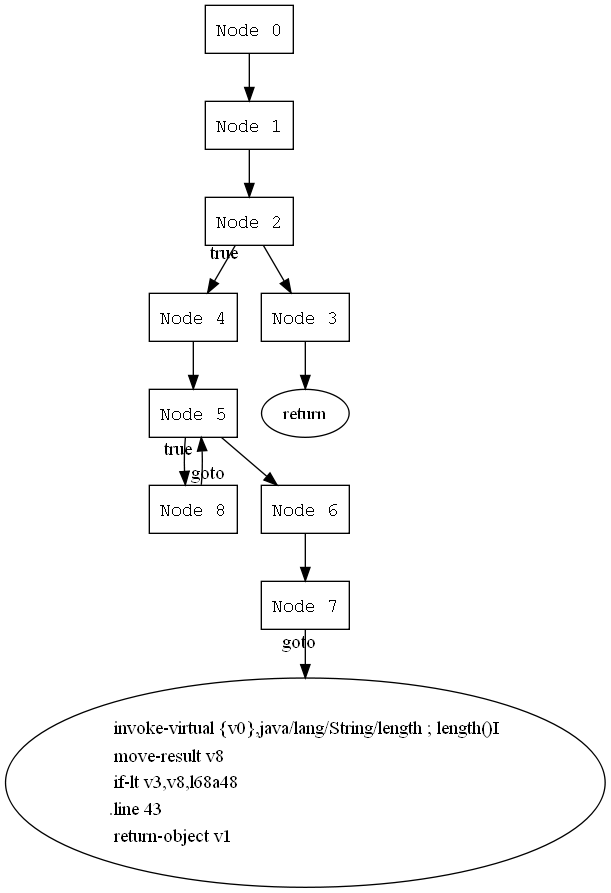


图6.11图6.10对应的结点图

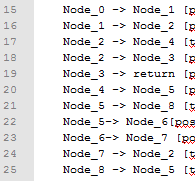


图6.12 基本块转化为结点写入文件

得到基本块之间的对应编号后，可以直接用结点号对支配关系和后继支配关系以及控制依赖等进行求解。根据TaintDroid的传播语义可知，Get\_IMEI()的返回函数要被污点标记，在对变量X进行赋值时，X也应该被污点标记，因此在初始化Node 1的时候，Tflag被设置为1，但Node 1不是条件控制语句，cflag被设置为0；而在初始化Node 2时，存在被污点标记的信息，而Node 2又包含条件语句，因此在设置污点标记Tflag为1，控制标识cflag为1，同样初始化的还有Node 5。初始化后的结点的信息后，利用控制依赖求解结果如图6.13所示，其中Node 9为添加的exit结点。从图中不难发现，控制依赖于基本块2的结点有4、5、6、7、2，控制依赖于5的基本块有5和8，而根据污点标记算法，需要对结点2,4,5,6,7,8中的变化变量进行污点标记。在初始化的时候结点2和结点5的cflag已经被设置为1，而4,6,7,8的变量污点设置如表6. 1所示。表中的第1

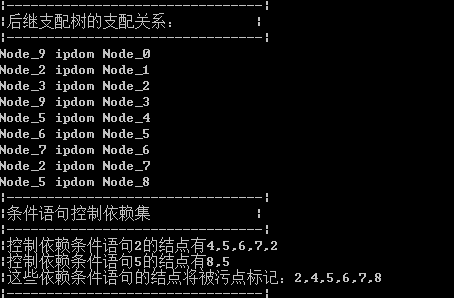


图6.13 依赖控制语句的基本块

1列和第2列分别表示基本块编号和源代码，第三列表示反编译后变量对应的寄存器。第四列表示使用TaintDroid标记的变量，第五列表示使用CFMM模块仿真的污点标记结果。根据TaintDroid污点传播语义，在Get\_IMEI()返回的返回值中被附加了一个0x400类型的污点标记，进入getY()方法后，Get\_IMEI()返回值赋值给变量X，因此变量X被污点标记。进入第一个循环后，X.charAt(i)赋值给变量n，此时n被污点标记。而对定义的变量y，TaintDroid没有标记，而根据

表6.1 基本块对应的源代码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Node | 源代码 | 反编译后  对应变量 | TaintDroid  标记变量 | CFMM模块  标记变量 |
| 4 | char x = X.charAt(i); int n = x; int y = 0; | v6 | 1 | 1 |
| v5 | 1 | 1 |
| v7 | 0 | 1 |
| 6 | char c = (char)y; | v2 | 0 | 1 |
| 7 | Y = Y+c; i++； | v1  v3 | 0  0 | 1  1 |
| 8 | y=y+1;  j++ | v7  v4 | 0  0 | 1  1 |

污点传播策略，y的值有所改变且控制依赖于i<X.length()(cflag=1)，因此Taint(y) = Taint(X.length())；同样对于变量Y也会被污点被标记。从表6.1的最后两列可以发现，TaintDroid对变量i,j,c,y,Y没有标记，而这些变量都会导致隐私信息IMEI的泄露，而CFMM模块对这些变量都能污点标记。

## 6.4 本章小结

根据第四章的污点传播策略以及第五章的控制依赖算法，本章首先为TaintDroid系统进行了改进设计，为TaintDroid添加了CFMM模块。针对CFMM模块的功能，本章使用仿真实验进行分析。通过对Android应用的dex二进制代码分析反编译，使用反编译后的指令构造基本块，并运用控制流图中支配关系对基本块的控制依赖集合进行求解，并对求解出的基本块中变量进行污点标记分析。最后根据仿真的结果与TaintDroid标记结果进行对比分析，结果表明CFMM模块可以标记控制流引起的隐私信息泄露。

# 第7章 结束语

Android应用的多样性为人们的生活带来许多便利，但在享受娱乐的同时也承担着隐私信息泄露的风险。由于Android应用的不断发展，其极大的开放性带来的风险也随之剧增。如何保护Android隐私数据成为Android研究者和安全维护者一个巨大难题。TaintDroid虽极大程度监控了隐私数据，但也存在着一定的不足，它无法监控由控制流引起的隐式信息传播。本文就TaintDroid在控制流上存在的污点标记不足的缺陷加以改进，使用污点标记不足的模型来表示TaintDroid的缺陷，并根据该模型提出污点标记策略并对污点标记策略的可行性进行了验证，为改进TaintDroid打下理论基础。对于控制流的分析，本文又引入了控制流图以及对应的支配树等相关概念，然后对支配关系和控制依赖关系的求解给出相应的算法，并对这些算法的算法复杂度进行了详细分析；然后把污点标记策略和控制依赖算法相结合提出污点标记算法，并对算法复杂度进行了分析。最后本文对TaintDroid进行改进设计，在TaintDroid上添加CFMM模块，模块由静态分析和动态分析结合，在静态分析上对Android应用反编译，然后使用指令构造基本块，并生产控制流图；动态分析中利用支配关系和控制依赖等相关算法对控制依赖于条件语句的基本块进行求解，然后用污点标记算法对这些依赖基本块进行污点标记。最后对CFMM模块的监控功能进行了仿真实验分析，通过仿真实验验证CFMM模块可以标记控制流中的隐式流信息。

由于研究时间和涉及知识跨度较广等多方面方面原因，本文内容仍有一些可以改进和完善的地方。因此在下一阶段的研究工作中，目标主要定在以下两个方面：

1. 本文使用仿真实验来验证CFMM模块的功能，因此后面的首要目标就是要实现CFMM模块，把CFMM模块移植到TaintDroid上；把设计变为实现，可以近一步完善TaintDroid的功能，增加TaintDroid的实用性。
2. 本文中CFMM模块仅仅对应用中控制流的隐式传播进行监控，对于应用框架层的控制流隐式传播问题还未予考虑；在实现CFMM模块的同时，下一步就对应用框架层进行分析，研究信息在在框架控制语句中的传播方式，然后建立更完善的保护模型，对Android隐私数据形成更全面防护。

# 参考文献

1. 2014谷歌I/O大会. [EB/OL]. http://tech.163.com/14/0626/03/9VKS4619000915BF.html
2. App Annie 2014年第一季度移动应用市场指数报告. [EB/OL].[2014-04-21]. http://www.cnbeta.com/articles/285041.htm
3. 张嘉宾. Android 应用的安全性研究[D]. 北京邮电大学, 2013.
4. 彭智俊. 采用静态分析检测 Android 应用信息泄露的研究[D]. 复旦大学, 2012.
5. 张玉清 王 凯 杨 欢 方喆君 王志强 曹 琛. Android 安全综述[J]. 计算机研究与发展, 2014, 51(7): 1385-1396.
6. 杨博. Android系统下应用程序的安全性研究[D]. 上海交通大学, 2013.
7. Au K W Y, Zhou Y F, Huang Z, et al. PScout: analyzing the Android permission specification[J]. Acm Conference on Computer & Communications Security, 2012.
8. 路程. Android 平台恶意软件检测系统的设计与实现[D].北京邮电大学,2012
9. Ongtang M, McLaughlin S, Enck W, et al. Semantically rich application-centric security in Android[C]. //Computer Security Applications Conference. 2009:658–673.
10. Enck W, Octeau D, McDaniel P, et al. A study of Android application security[J]. Proc Usenix Security Symposium, 2011.
11. Dietz M, Shekhar S, Pisetsky Y, et al. Quire: Lightweight Provenance for Smart Phone Operating Systems[J]. Computer Science, 2011:23-23.
12. Bugiel S, Davi L, Dmitrienko A, et al. Xmandroid: A new android evolution to mitigate privilege escalation attacks[J].Technische University Darmstadt, Technical Report TR-2011-04, 2011
13. Barrera D, Kayacik H G, ne&#, et al. A methodology for empirical analysis of permission-based security models and its application to android[C]. //Ccs Proceedings of Acm Conference on Computer & Communications Security. ACM, 2010.
14. Enck W, Gilbert P, Chun B, et al. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones[J]. Proceedings of Usenix Conference on Operating Systems Design & Implementation Osdi, 2010, 57(3):99-106.
15. 杨珉, 王晓阳, 张涛等. 国内Android应用商城中程序隐私泄露分析[J]. 清华大学学报：自然科学版, 2012, (10):1420-1426.
16. Hornyack P, Han S, Jung J, et al. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications.[J]. In Acm Ccs, 2011.
17. Yajin Zhou, Xinwen Zhang, Xuxian Jiang, et al. Taming Information-Stealing Smartphone Applications (on Android)[J]. Lecture Notes in Computer Science, 2011:93-107.
18. Beresford A R, Rice A, Skehin N, et al. MockDroid: trading privacy for application functionality on smartphones[C]. //Proceedings of the Twelfth International Workshop on Mobile Computing Systems & Applications Hotmobile’11 Acm. ACM, 2011:49-54.
19. Batyuk L, Herpich M, Camtepe S A, et al. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications[C]. //International Conference on Malicious & Unwanted Software. IEEE, 2011:66-72.
20. Evans D, Larochelle D. Improving Security Using Extensible Lightweight Static Analysis[J]. IEEE Software, 2002, 19(1):42-51.
21. Zhang X, Edwards A, Jaeger T. Using CQUAL for Static Analysis of Authorization Hook Placement.[J]. Usenix Security Symposium, 2002.
22. Shankar U, Talwar K, Foster J S, et al. Detecting format string vulnerabilities with type qualifiers[J]. Proceedings of Usenix Security Symposium, 2001, 10:201--220.
23. Denning D E. A lattice model of secure information flow[J]. Communications of the Acm, 1976, 19(5):236-243.
24. Denning D E, Denning P J. Certification of programs for secure information flow[J]. Communications of the Acm, 1977, 20(7):504-513.
25. Myers A C. JFlow: Practical Mostly-Static Information Flow Control[J]. In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL, 1970:228-241.
26. Cao Y, Fratantonio Y, Bianchi A, et al. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework[J]. 2015.
27. Chin E, Felt A P, Greenwood K, et al. Analyzing inter-application communication in Android[C]//Proceedings of the 9th international conference on Mobile systems, applications, and services. ACM, 2011: 239-252.
28. Arzt S, Rasthofer S, Fritz C, et al. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps[J]. Acm Sigplan Notices, 2014, 49.
29. Chan P P F, Hui L C K, M. Yiu S. DroidChecker: analyzing Android applications for capability leak[J]. Proc of Wisec’, 2012:125-136.
30. Lu L, Li Z, Wu Z, et al. CHEX: statically vetting Android apps for component hijacking vulnerabilities[C]. //Proc of the Acm Conference on Computer & Communications Security Ccs ’12 Acm. ACM, 2012:229--240.
31. IccTA: Detecting IccTA: Detecting Inter-Component Privacy Leaks in Android Apps Inter-Component Privacy Leaks in Android Apps
32. 李伟明, 张爱芳, 刘建财等. 网络协议的自动化模糊测试漏洞挖掘方法[J]. 计算机学报, 2011, 34(2):242-255. DOI:10.3724/SP.J.1016.2011.00242.
33. Chow J, Pfaff B, Garfinkel T, et al. Understanding data lifetime via whole system simulation[C]. //Usenix Security Symposium. USENIX Association, 2004.
34. Sarwar G, Mehani O, Boreli R, et al. On the Effectiveness of Dynamic Taint Analysis for Protecting against Private Information Leaks on Android-based Devices[C]//SECRYPT. 2013: 461-468.
35. Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software[J]. Network & Distributed Systems Security Symposium, 2005.
36. Nethercote N, Seward J. Valgrind: A program supervision framework[J]. Electronic Notes in Theoretical Computer Science, 2003, 89(4):44–66.
37. Graa M, Cuppens-Boulahia N, Cuppens F, et al. Detecting control flow in smarphones: Combining static and dynamic analyses[M]//Cyberspace Safety and Security. Springer Berlin Heidelberg, 2012: 33-47.
38. Hiroshige S, Yu B, Zhao Q, et al. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting.[J]. Proceedings- International Symposium on Computers and Communications, 2006:749--754.
39. Qin F, Wang C, Li Z, et al. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks[C]. //Null. IEEE Computer Society, 2006:135-148.
40. Haldar V, Chandra D, Franz M. Dynamic taint propagation for Java[J]. Computer Security Applications Conference, 2006:9 pp. - 311.
41. Kwong L, Yin Y H. DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis[J]. Proceedings of Usenix Security Symposium, 2012.
42. Lantz P, Desnos A, Yang K. DroidBox: Android application sandbox[J]. 2012.
43. Qian C, Luo X, Shao Y, et al. On Tracking Information Flows through JNI in Android Applications[C]. //IEEE/IFIP International Conference on Dependable Systems & Networks. IEEE Computer Society, 2014:180-191.
44. A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In Proceedings of the sixth European Workshop on Systems Security (EuroSec), 2013.
45. Zhang Y, Yang M, Xu B, et al. Vetting undesirable behaviors in android apps with permission use analysis[C]. //IEEE Transactions on Information Forensics & Security. 2013.
46. Zhu D (, Jung J, Song D, et al. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking[C]. //Sigops Operating Systems Review. 2011.
47. Yin H, Song D, Egele M, et al. Panorama: Capturing system-wide information flow for malware detection and analysis[J]. In Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS’07, 2007:116-127.
48. Fenton J S. Information Protection Systems.[J]. University of Cambridge, 1973.
49. Gat I, Saal H J. Memoryless execution: A programmer's viewpoint[J]. Software Practice & Experience, 1976, 6(4):463–471.
50. Brown J, al. E. A Minimal Trusted Computing Base for Dynamically Ensuring Secure Information Flow[J]. Aries Memo, 2001.
51. Beres Y, Dalton C I. Dynamic label binding at run-time[J]. Proc Workshop on New Security Paradigms, 2003:39--46.
52. Denning D E R. Secure information flow in computer systems.[J]. Phd Th, Comptr Sci Dep, Purdue, 1975.
53. Song D, Brumley D, Yin H, et al. BitBlaze: A new approach to computer security via binary analysis[M]//Information systems security. Springer Berlin Heidelberg, 2008: 1-25.
54. Stephen M, Dawnsong M P. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation[J]. Ndss, 2011.
55. Ottoni G, Rangan R, Chang J, et al. RIFLE: An Architectural Framework for User-Centric Information-Flow Security[C]. //International Symposium on Microarchitecture. IEEE Computer Society, 2004:243-254.
56. Chandra D. Information flow analysis and enforcement in java bytecode[M]. University of California at Irvine, 2006.
57. Nair S K, Simpson P N D, Crispo B, et al. A virtual machine based information flow control system for policy enforcement[J]. Electronic Notes in Theoretical Computer Science, 2008, 197(1): 3-16.
58. Graa M, Cuppens-Boulahia N, Cuppens F, et al. Formal Characterization of Illegal Control Flow in Android System[C]. //International Conference on Signal-image Technology & Internet-based Systems. IEEE Computer Society, 2013:293-300.
59. 马红途. OpenMP程序分析及优化技术研究[D]. 解放军信息工程大学, 2009.
60. Lowry E S, Medlock C W. Object code optimization.[J]. Communications of the Acm, 1969, 12(1):13-22.
61. Jr P W P, Moore E F. Immediate predominators in a directed graph [h] (algorithm 430[C]. //Communications of the Acm. 1972.
62. Allen F E. Control Flow Analysis[J]. Acm Sigplan Notices, 1970, 5:1 - 19.
63. Ullman M S H A J D. A Simple Algorithm for Global Data Flow Analysis Problems.[J]. Siam Journal on Computing, 1975, (4):519-532.
64. Endre Tarjan R. Testing flow graph reducibility[J]. J. Comput. System Sci, 1974, 9(3):355–365.
65. Buchsbaum A L, Kaplan H, Rogers A, et al. Linear-Time Pointer-Machine Algorithms for Least Common Ancestors, MST Verification, and Dominators[C]. //Proceedings of the Thirtieth Annual Acm Symposium on Theory of Computing. 1998:279--288.
66. Jeanne Ferrante K J O A J D W. The program dependence graph and its use in optimization[J]. Acm Transactions on Programming Languages & Systems, 1983, 9(4):319--349.
67. 马红途, 赵荣彩, 苏彦兵. 控制流图上支配关系计算方法的分析与实现[J]. 计算机科学, 2009, 36(3): 54-57.

# 致谢

研究生学习生涯即将结束，回顾三年的岁月，这段宝贵的求学经历使我终生难忘。在此，感谢读研期间帮助我的老师、同学和亲人。

首先要感谢我的导师严飞副教授。因为我是跨专业读研，在开始相关专业知识的学习上略有吃力。是严老师不厌其烦的耐心指导，让我逐步掌握了相关专业技能。严老师不仅学术上对我细心指导，而且在在生活的关怀也是无微不至。严老师严谨的治学精神、循循善诱的教导方式、不拘一格的治学思想、渊博的知识和博大的胸怀都让我受益匪浅。

感谢我们实验室的张焕国老师，赵波老师、王鹃老师以及唐明老师对我的关怀和指导，让我感受到实验室如家庭般的温暖。

感谢与我一起进行该课题研究的王涛、伍湘奇。本课题的很多工作受益于他们的帮助。

感谢与我一同走过研究生生活的杨永娇、李志华、谢旭徽、周思珺、宋蕊霞、张乐艺、王兰、于钊、曾进、黄凡、彭慧荣等同学，他们让我在这个集体里身心愉悦地进行学习和研究，度过了一段终身难忘的岁月。

感谢室友周考、姚琪、黄訸在生活上给我的帮助。

感谢我的家人给予的理解、鼓励和支持。养育之恩，无以回报，愿他们永远健康快乐。

最后，再一次向所有帮助、关心过我的人说一声谢谢！

# 读研期间研究成果

发表论文

1. 王庆飞, 严飞, 王鹃, 等. IaaS 下虚拟机的安全存储和可信启动[J]. 武汉大学学报: 理学版, 2014 (3): 231-236. 第一作者
2. Wang J, Xie X, Wang Q, et al. Towards a Trusted Launch Mechanism for Virtual Machines in Cloud Computing[M]//Cloud Computing. Springer International Publishing, 2014: 90-101.第三作者
3. 王涛, 严飞, 王庆飞, 等. 基于软件定义网络的非集中式信息流控制系统——S-DIFC[J]. 计算机应用, 2015, 35(1): 62-67. 第三作者