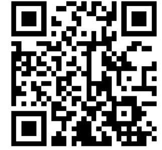
软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(6):1701−1716 [doi: 10.13328/j.cnki.jos.006245]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于污染变量关系图的 Android 应用污点分析工具*



张 捷, 田 聪, 段振华

(西安电子科技大学 计算机科学与技术学院,陕西 西安 710071)

通讯作者: 田聪, E-mail: ctian@mail.xidian.edu.cn

摘 要: 污点分析技术是检测 Android 智能手机隐私数据泄露的有效方法,目前主流的 Android 应用污点分析工具主要关注分析的精度,常常忽略运行效率的提升.在分析一些复杂应用时,过大的开销可能造成超时或程序崩溃等问题,影响工具的广泛使用.为了减少分析时间、提高效率,提出一种基于污染变量关系图的污点分析方法.该方法定义了污染变量关系图用于描述程序中污染变量及其关系,摒弃了传统数据流分析框架,将污点分析和别名分析进行结合,从程序中抽象出污染变量关系图和潜在污染流,并在控制流图上对潜在污染流进行验证以提高精度.详细描述了基于该方法所实现的工具 FastDroid 的架构、模块及算法细节.实验使用了 3 个不同的测试集,分别为 DroidBench-2.0,MalGenome 以及 Google Play 上随机下载的 1517 个应用.实验结果表明:FastDroid 在 DroidBench-2.0 测试集上的查准率和查全率分别达到 93.3%和 85.8%,比目前主流工具 FlowDroid 更高,并且在 3 个测试集上所用的分析时间更少且更稳定.

关键词: 静态分析;污点分析;软件安全;隐私保护;Android 应用

中图法分类号: TP311

中文引用格式: 张捷,田聪,段振华.基于污染变量关系图的 Android 应用污点分析工具.软件学报,2021,32(6):1701−1716. http://www.jos.org.cn/1000-9825/6245.htm

英文引用格式: Zhang J, Tian C, Duan ZH. Taint analysis tool of Android applications based on tainted value graph. Ruan Jian Xue Bao/Journal of Software, 2021,32(6):1701−1716 (in Chinese). http://www.jos.org.cn/1000-9825/6245.htm

Taint Analysis Tool of Android Applications Based on Tainted Value Graph

ZHANG Jie, TIAN Cong, DUAN Zhen-Hua

(School of Computer Science and Technology, Xidian University, Xi’an 710071, China)

Abstract: The taint analysis technology is an effective method to detect the privacy data leakage of Android smart phones. However, the state-of-the-art tools of taint analysis for Android applications mainly focus on the accuracy with few of them addressing the importance of the efficiency and time cost. Actually, the high cost may cause problems such as timeouts or program crashes when the tools analyze some complex applications, which block them from wide usage. This study proposes a novel taint analysis approach based on the tainted value graph, which reduces the time cost and improves the efficiency. The tainted value graph is formalized to describe the tainted values and their relationships and the taint analysis and alias analysis are combined together without using the traditional data flow analysis framework. In addition, the taint flows are verified on the control flow graph to improve accuracy. The architecture, modules, and algorithmic details of the proposed tool FastDroid are also described in this paper. The tool is evaluated on three test suites: DroidBench-2.0, MalGenome, and 1517 apps randomly downloaded from Google Play. The experimental results show that, compared

* 基金项目: 科技部重点研发计划(2018AAA0103202); 国家自然科学基金(61732013, 61751207); 陕西省科技创新团队(2019TD-001)

Foundation item: Major Program of the Ministry of Science and Technology of China (2018AAA0103202); National Natural Science Foundation of China (61732013, 61751207); Key Science and Technology Innovation Team of Shaanxi Provience (2019TD-001)

本文由“形式化方法与应用”专题特约编辑邓玉欣教授推荐.

收稿时间: 2020-08-29; 修改时间: 2020-10-26, 2020-12-19; 采用时间: 2021-01-18; jos 在线出版时间: 2021-02-07

with the tool FlowDroid, FastDroid has a higher precision of 93.3% and a higher recall of 85.8% on DroidBench-2.0, and the time cost for analysis is less and more stable on all the test suites.

Key words: static analysis; taint analysis; software security; privacy protection; Android applications

近年来,智能手机得到了广泛的普及和应用,智能手机成为人类生活工作中不可或缺的重要工具.在智能手机的操作系统中,Android 系统占据主要份额,据统计,在全球手机操作系统中,Android 系统的市场份额达到74.1%[1].智能手机的普及,离不开丰富多样的应用,用户可以通过安装在智能手机上的各种应用,实现其社交、娱乐、管理、支付等多种需求.然而,根据 360 安全公司 2019 年手机安全状况报告的数据,2019 年新增的恶意程序样本约 180.9 万个,其中,隐私窃取类的恶意程序占 41.9%[2].因此,智能手机应用的安全问题备受工业界和学术界的关注,特别是隐私数据保护问题成为安全问题的研究重点和热点.Android 应用由于功能的需要,经常会存储和访问用户的隐私数据,例如通讯录、短消息、地理位置、账户密码等等.应用在获取用户授予的相关权限后,可以访问隐私数据并提供相应服务,比如:导航应用会实时获取用户的地理位置,从而计算道路规划并进行导航.然而,一些应用可能会过度申请权限或滥用权限,违规违法泄露和使用用户的隐私数据并造成严重的问题.比如:用户的银行账户密码泄露可能造成财产的损失,通讯录的泄露会严重侵犯用户的隐私.因此,智能手机的隐私数据保护具有十分重要的意义.

对于应用本身造成的隐私数据泄露(不包括用户使用不当、操作系统漏洞或网络攻击等),有效地检测应用中代码包含的隐私泄露行为,是保护隐私数据的前提.污点分析(taint analysis)作为一种信息流分析技术,近年来被广泛应用于隐私数据泄露的检测[3].污点分析能够追踪应用程序中隐私数据从获取到泄露的整个传播过程,其中涉及诸多方面的研究,包括应用程序反编译、危险权限使用、隐私数据获取与泄露方式、污染变量传播方式等等.虽然污点分析结果不能直接判定某项隐私数据的泄露是否是恶意的,但可以从中了解应用程序使用和泄露隐私数据的相关细节,从而帮助判定是否构成恶意行为.另外,恶意行为的判定,还需要根据应用功能及其行为发生的上下文(context)等具体情况进行具体分析[4].例如:地图类型的应用访问用户地理位置这一隐私数据是合理的,而要求获取用户通讯录通常是不正常的.总之,污点分析提供了应用获取和使用隐私数据的全貌,可为进一步深入分析应用、保护隐私打下基础.

目前已有的面向 Android 应用的精确的静态污点分析工具[5−8]往往关注分析的精度,忽略了分析效率和运行时间.在分析大规模复杂应用时,效率问题可能造成分析超时甚至程序崩溃[9,10]等问题,影响工具在现实中的广泛应用.本文提出一种基于污染变量关系图的 Android 应用静态污点分析方法,该方法摒弃了传统数据流分析框架,将污点分析和别名分析进行结合,从程序中抽象出污染变量关系图和潜在污染流,并在控制流图上对潜在污染流进行验证.实验表明:基于该方法实现的工具 FastDroid 在保证较高精度的同时,大大减少了分析时间,并且分析时间更加稳定.

本文的主要贡献包括以下 3 点:

(1) 定义了用于描述污染变量及其关系的污染变量关系图;

(2) 提出一种新的基于污染变量关系图的高效准确的静态污点分析方法;

(3) 基于该方法实现了工具 FastDroid,实验表明:FastDroid 能够准确检测 Android 应用中的隐私数据泄露,并且在运行时间上优于当前主流的静态污点分析工具.

本文第 1 节简要介绍 Android 应用污点分析的相关背景.第 2 节阐述基于污染变量关系图的污点分析方法的研究动机,给出污染变量关系图和污染传播规则的定义,并介绍方法的实施步骤.第 3 节详细介绍基于该方法的工具实现.第 4 节给出工具的实验和分析结果.第 5 节介绍相关工作.第 6 节总结全文并展望未来工作.

1 Android 应用污点分析的相关背景

1.1 污点分析简介

污点分析是一种信息流分析技术,通过插桩、数据流分析等方法追踪程序中带污点标记的数据的传播以及

泄露情况[3,10].污点分析目前广泛应用于 Android 应用的隐私泄露检测,总体上可分为动态和静态两种方法[3]:动态方法主要在程序运行过程中对程序的行为进行监控,及时发现隐私数据的泄露,典型的面向 Android 应用的动态方法有 TaintDroid[11],TaintEraser[12]等;静态方法是在不运行软件的前提下,对隐私数据泄露行为进行分析,分析的对象可以是源码、中间代码或目标代码等[13],目前比较流行的面向 Android 应用的静态方法有 FlowDroid[6],Amandroid[8]等.本文主要研究静态污点分析方法,相比动态方法,静态方法拥有如下优势:(1) 覆盖全部代码,分析更加全面、准确;(2) 某些恶意应用能够检测出动态分析的运行环境,之后隐藏恶意行为从而逃避检测,而在静态分析中则不会出现此情况.

下面对污点分析用于隐私泄露检测中的常见概念进行解释.程序中,隐私数据的获取接口被称为污点源(source).隐私数据的泄露接口被称为泄露点(sink).程序在获取隐私数据之后,可以通过赋值、计算等方式在变量中对其进行传播,这些变量被称为污染变量(tainted value).污染变量可以是局部变量、静态变量或者对象的域等.隐私数据从污点源经过污染变量到达泄露点的传播过程形成的数据流可称为污染流(taint flow).图 1 是一个应用中两个函数的代码片段,图 1(a)中的 *onStart* 生命周期函数调用了一个污点源 *getDeviceID* 用于获取设备 ID 号,该隐私数据随后经过 *a*,*b*,*x*,*y* 等污染变量的传播,到达图 1(b)中 *Send* 函数的一个泄露点 *sendTextMessage*,该 API 可将隐私数据通过短信方式进行发送.图中,污染变量用红圈标记,污染流用蓝色实线示意.这条污染流跨越两个函数,通过赋值、函数参数传递、字符串计算等不同方式进行传播.这些传播方式在污点分析方法中,一般需要抽象成为污染传播规则用于检测污染变量.

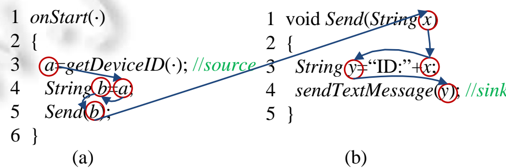


Fig.1 An example of inter-procedural taint analysis

图 1 过程间的污点分析示意图

1.2 Android应用污点分析工具实现的关键问题

(1) Android 系统环境的模拟

Android 应用具有事件驱动、多入口、多组件等特点,严格来说,应用不是一个能够独立运行的程序,甚至可以看作是 Android 操作系统的插件[14].应用中没有类似 Java 程序的 *main* 函数,各组件的生命周期函数或者回调函数根据用户互动或设备事件触发而由系统环境直接调用,在代码中缺少这些函数之间明确的调用关系.因此,传统 Java 程序分析工具不能直接使用在 Android 应用上.为了准确分析 Android 应用,我们需要对 Android 系统环境进行模拟,生成完整准确的函数调用图、控制流图等程序结构.另外,一个 Android 应用可由多个组件构成,隐私数据可在不同组件间传播,通常将隐私数据封装在用于组件间通信的专用对象 Intent 中,显式或隐式地传送到另一个组件.因此,Android 应用的污点分析需要获得所有组件间通信信息用于支持组件间分析.目前,研究人员已经提出一些分析组件间通信的工具,例如 IC3[15],Epicc[16]等.

(2) APK 文件的反编译

Android 应用的安装文件是 Android 应用程序包(Android application package),文件格式是 APK 文件.该文件由多个文件组合压缩而成,包含 dex 文件、文件资源、证书、manifest 文件等.其中,与源代码最为相关的是 dex 文件,它是在 Android 的 Dalvik 虚拟机上运行的可执行文件.因为 dex 文件中的字节码不利于直接分析,工具往往首先将该文件反编译为中间代码(IR)或者源码方便操作.常见工具和中间代码有 Soot[17]生成的 Jimple、APKtool 生成的 Smali[18]、Androguard[19]生成的 DEX_ASSEMBLER 等等.比较成熟的反编译工具不仅可以生成中间代码,而且提供在中间代码层面上进行程序分析的操作接口,用于获取程序中的语句、变量等详细信息,甚至可以直接生成函数调用关系和控制流图等信息.

(3) 污点源和泄露点的识别

Android应用中存在许多污点源和泄露点的方法,污点分析需要准确识别出程序中的污点源和泄露点.常见的污点源可通过调用 Android SDK 中相关 API 获取隐私数据,有些隐私数据可通过 Android 组件内容提供器(content provider)获取,例如通讯录数据,但最终访问数据时仍需调用相应的 API.研究初期的检测方法主要从Android的SDK中手工筛选出与污点源和泄露点相关的API,但这种方式收集的API并不完整.Rasthofer等人[20]针对这个问题提出了一个较全面的方案,基于对污点源和泄露点的规范定义开发了工具 SUSI,利用机器学习的方法找出 Android 所有 API 中存在的符合定义的大量污点源和泄露点并进行归类.

(4) 污染传播规则的定义

污点分析需要根据污染变量传播的特征归纳出污染传播规则,并依据规则设计相应算法追踪污染变量.一般来说,污染传播规则包括污点源和泄露点的识别规则、污染变量的数据流规则和别名规则等.例如:FlowDroid工具[21]定义了基于污染传播规则的流方程,并为别名分析专门设计了独立的流方程.在 IFDS 框架下,流方程实际上实现了分析语句时所采用的规则.同样,Apposcopy 工具[5]的静态污点分析模块专门定义了污点源、泄露点和污染传播的谓词和规则,并基于此规则进行污点分析.如果从污染变量的传播关系上看,隐私数据在污染变量之间的常见传播方式可包括赋值、参数传递、计算、别名等.需要注意的是:别名分析是污点分析中需要解决的重要问题,当某变量被污染后(即被传入隐私数据),它的别名变量同样会被污染,因为在 Java 中,堆变量的所有别名都指向同一块内存.另外,工具的实现需要考虑 Android 或 Java 平台的一些特殊污染方式,并单独设计对应的污染传播规则,比如应用中存在涉及线程、隐式污染流、native 代码、组件间通信、反射机制等特殊的传播方式,这些方式无法归纳到通常的规则当中,需要根据具体特征来定义规则.总之,为了识别不同污染传播方式所产生的污染变量,污点分析应尽可能全面地定义污染传播规则以提高分析精度.

(5) 敏感性的支持

静态分析方法,对不同敏感性的支持直接影响分析的精度和速度,例如流敏感、对象敏感、域敏感、上下文敏感、路径敏感等敏感性[10].具体来说,支持流敏感需要在分析中考虑各语句的执行顺序,一般要求工具获得准确的控制流图.支持对象敏感的方法能够准确识别某个对象,在分析该对象的域或者方法时,不会与相同类的其他对象混淆,即工具需要区分同一类的不同对象.支持域敏感的方法在分析中能够区分对象中的每个域,工具需记录域的信息.支持上下文敏感需要分析函数在不同位置调用所产生的影响,工具需记录函数调用点相关信息.支持路径敏感需要分析程序各执行路径不同所导致的不同结果,工具需考虑程序分支等细节.静态分析方法对敏感性的支持在设计和实现中的精度和复杂性差异很大,一般来说,敏感性支持得越多,分析精度越高,但随之,工具实现越复杂、分析成本越大、速度越慢.精度与速度之间往往存在矛盾,需要在二者间进行平衡[13].

2 基于污染变量关系图的污点分析方法

2.1 研究动机

目前,精确的静态污点分析方法大多基于数据流分析框架,在控制流图上对隐私数据进行追踪和分析,另外单独对污染变量进行别名分析.例如:Amandroid[8]提前对所有变量进行大规模的别名分析之后,再进行污点分析;而 FlowDroid[6]则是在发现污染变量是堆变量后,才按需进行反向的别名搜索.虽然 FlowDroid 仅针对污染变量进行别名分析,开销相对 Amandroid 小一些,但在实验中我们发现:FlowDroid 在分析大规模复杂的应用时,一个应用花费的时间常常达到几分钟至几十分钟,甚至出现超时或运行失败的情况.所以我们认为,目前的方法在运行效率上仍存在提升空间:一方面,当程序中污染变量的别名较多时,独立运行的别名分析将不可避免地产生很大的开销;另一方面,传统的数据流分析框架不是专门为污点分析定制的,框架本身的运行开销很大.

为了提高工具效率,我们提出一种新的污点分析方法,该方法摒弃了传统数据流分析框架,采用一种轻量级的算法构建出污染变量关系图,将别名分析和污点分析同步进行,也就是在追踪污染变量的同时识别污染变量的别名.方法的大致步骤如下:首先,从一个应用的程序中抽象出一个树形结构,扫描所有树形结构的叶子节点(即语句),基于污染传播规则检测所有可能的污染变量及其别名变量;然后,根据变量关系生成污染变量关系图,

确定潜在污染流;最后,在控制流图上对污染流进行验证,得到程序中真实存在的污染流.为此,我们形式化定义了污染变量关系图,设计了污染传播规则和验证污染流的算法.

2.2 污染变量关系图

我们提出污染变量关系图(tainted value graph,简称 TVG)用于描述程序中所有可能存在的污染变量、别名变量及变量间的污染传播关系.直观来看,所有污染变量都是由污点源传播得到的,污染变量及其别名变量通过各种污染传播方式不断扩散,最终传播到一个泄露点,形成一条泄露隐私数据的污染流.为准确表达这样的污染传播过程,我们使用图的结构记录变量及其传播关系.下面给出污染变量关系图的定义:污染变量关系图是一个简单有向连通图 $TVG=(V,E)$,其中,顶点 $v\in V$ 表示一个污染变量、污点源或泄露点,边 $e\in E$ 表示两个变量的污染传播关系、变量被污点源污染或者污染变量传播给泄露点的关系.另外,我们要求 TVG 满足以下性质.

- 有且仅有一个入度为零的顶点,即污点源顶点 SC_i;存在零或若干泄露点顶点 SK_j;
- 除了 SC_i 以外的所有顶点都至少存在一条从 SC_i 到该顶点的路径;
- 图中不含回路.

从性质来看,每个 TVG 对应一个污点源在程序中的传播过程.当一个应用中存在多个污点源时,则可生成相同数量的 TVG,而污染流能否形成取决于图中是否有污染变量被泄露.例如,图 2 显示了图 1 程序中的 TVG,整个程序中存在许多变量,用空心圆圈表示,其中,红色空心圆圈是污染变量,红色实心圈表示污点源,蓝色实心圈表示泄露点.该 TVG 的顶点集 $V=\{SC_1,a,b,x,y,SK_1\}$,边集 $E=\{(SC_1,a),(a,b),(b,x),(x,y),(y,SK_1)\}$.在现实中,污染变量之间的关系可能更加复杂,存在一对一、一对多和多对一的情况.举例来说:当某污染变量分别污染多个不同变量,或者作为参数传入到多个函数,此时,该变量的顶点将连接多条边到其他顶点;当两个不同的污染变量都作为同一参数调用某函数时,此时,两个污染变量的顶点将各自连接一条边到该函数形参的顶点上.

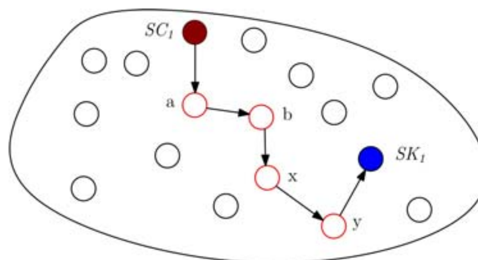


Fig.2 An example of tainted value graph

图 2 污染变量关系图示例

2.3 污染传播规则

我们定义了一系列污染传播规则用于发现污点源、污染变量及泄露点,并创新性地将别名规则也作为一类污染传播方式加入到规则中.这样,在分析中可同时寻找污染变量及其别名,并一同构成 TVG.污染传播规则采用类似 Datalog 的语言来定义,Datalog[22]是一种应用于数据库查询的语言,由于其较强的逻辑性,被广泛应用于各类研究领域,例如程序分析、信息提取等.一段 Datalog 程序由一套规则和一组事实组成,事实由谓词描述.每条规则由左右两部分构成,中间用:-符号分隔开.当右部所有谓词都成立时,左部的谓词为真.对于污染传播规则来说,当语句及其包含的变量满足某项规则右部的所有谓词时,左部的变量污染传播关系存在或者某个描述变量或方法性质的谓词成立.

我们定义了 *S*,*IN*,*Param*,*Return*,*ReturnP*,*Clean*,*hp* 等谓词用于描述程序中语句或者变量的性质.*S*(*stmt*)谓词描述了语句 *stmt* 的类型,当满足特定类型的语句时,谓词为真;*IN*(*x*)为真,表示变量 *x* 是污染变量;*Param*(*f*,*x*,*i*)为真,表示函数 *f* 第 *i* 个参数被传入污染变量 *x*;*Return*(*f*,*x*)为真,表示函数 f 返回污染变量 *x*;*ReturnP*(*f*,*x*,*i*)为真,表示函数 *f* 第 *i* 个实参在函数内被变量 *x* 污染;*Clean*(*x*)为真,表示污染变量 *x* 被清除;*hp*(*x*)表示 *x* 为堆变量.规则中的 *Source*,*Sink* 分别表示污点源和泄露点方法的集合,*TW* 表示污染封装方法集合.污染封装方法是指在 Java 和

Android SDK 中存在一些特殊 API,如果向这些 API 传入一个污染变量作为参数,运算后的返回值同样是一个污染变量,例如方法 *java.lang.Integer toString*(*int*).为了节省开销,我们对于污染封装方法无需深入内部分析,而是将其作为污染传播的捷径.当识别出污染封装方法传入污染变量作为参数时,直接将其返回值判定为污染变量.

表 1 展示了定义的污染传播规则:

- 污染源规则归纳了污染源的检测方式,生成类似$\langle SC,x.*\rangle$的关系;
- 污染变量传播规则归纳了污染变量之间传播的各种方式,包括赋值、计算、参数传递、函数返回等,生成类似$\langle x.*,y.*\rangle$的污染传播关系;
- 别名规则归纳了污染变量之间别名关系,生成类似$\langle x.*,y.*\rangle$的污染传播关系;
- 方法调用和方法返回规则描述了污染变量跨函数传播的情况,得到 *Param*,*Return*,*ReturnP* 等谓词.这些谓词可以出现在污染变量传播规则的右部,作为污染传播关系产生的依据;
- 污染泄露规则总结了泄露点的检测方式,生成类似$\langle x.*,SK\rangle$的关系;
- 清除规则表示某污染变量被赋值为空值、非污染变量或者常量时,生成 *Clean* 谓词.

Table 1 Definition of our taint rules

表 1 污染传播规则的定义

规则类型	规则定义
污染源规则	R1:$\langle SC,x.*\rangle$:-$S(x.*=f(\cdot)),f\in Source$
污染变量传播规则	R2:$\langle x.*,y.*\rangle$:-$S(y.*=x.*),\neg hp(x.*),IN(x.*)$
	R3:$\langle x.*,y.*\rangle$:-$S(y=f(\cdot)),Return(f,x.*)$
	R4:$\langle x.*,p_i.*\rangle$:-$S(p_i=f.param_i),Param(f,i,x_i.*)$
	R5:$\langle p_i.+,x_i.+\rangle$:-$S(f(x_0,\ldots,x_n)),hp(x_i),ReturnP(f,i,p_i.+)$
	R6:$\langle p_i.+,x_i.+\rangle$:-$S(y=f(x_0,\ldots,x_n)),hp(x_i),ReturnP(f,i,p_i.+)$
	R7:$\langle x.*,y\rangle$:-$S(y.*=f(x.*)),f\in TW,IN(x.*)$
别名规则	R8:$\langle x.*,y.*\rangle$:-$S(y.*=x.*),hp(x.*),IN(x.*)$
	R9:$\langle x.*,y.*\rangle$:-$S(x.*=y.*),hp(x.*),IN(x.*)$
方法调用规则	R10:$Param(f,i,x_i.*)$:-$S(f(x_0.*,\ldots,x_n.*)),IN(x_i.*)$
	R11:$Param(f,i,x_i.*)$:-$S(y=f(x_0.*,\ldots,x_n.*)),IN(x_i.*)$
方法返回规则	R12:$Return(f,x.*)$:-$S(return\ x),IN(x.*)$
	R13:$ReturnP(f,i,p_i.+)$:-$S(p_i=f.param_i),hp(p_i),IN(p_i.+)$
污染泄露规则	R14:$\langle y.*,SK\rangle$:-$S(f(\ldots,y.*,\ldots)),IN(y.*),f\in Sink$
	R15:$\langle y.*,SK\rangle$:-$S(x=f(\ldots,y.*,\ldots)),IN(y.*),f\in Sink$
清除规则	R16:$Clean(x.*)$:-$S(x.*=y.*),IN(x.*),\neg IN(y.*)$
	R17:$Clean(x.*)$:-$S(x.*=null),IN(x.*)$

另外存在一些未被定义的特殊规则针对一些特殊情况下的污染传播方式而定义,生成相应的特殊谓词或者污染传播关系.例如:为了方便分析,我们在特殊规则中定义了涉及数组类型变量的污染方式,当出现数组中一个元素被污染时,认定该数组的所有元素都被污染.这样的规则可能会造成一些误报(false positive),但是可以减少分析成本.以规则:$\langle x.*,y.*\rangle$:-$S(y.*=x.*),\neg hp(x.*),IN(x.*)$.为例,当存在一个赋值语句 $y.*=x.*$,且 $x.*$是污染变量且不是堆变量时,则存在序偶关系$\langle x.*,y.*\rangle$表示 $x.*$到 $y.*$有污染传播关系.$x.*$和 $y.*$是变量的访问路径,我们使用变量的访问路径(access path)来明确污染变量[6].访问路径的格式如 $x.m.n$,其中,x 表示局部变量名、函数参数或者类名,m 和 n 表示域名.访问路径的长度根据变量具体情况可变,因此,我们在规则中使用 $x.*$或 $x.+$的形式表示.*表示零个及以上的域,+表示大于零个域.当为零个域时,访问路径表示一个简单的局部变量或参数.

2.4 TVG构建方法

为了构建 TVG,我们根据污染传播规则分析程序中每条语句,从而判定是否产生新的污染变量或谓词.首先将程序的语句按照方法和类的归属关系抽象成一个树形结构,每个方法中的语句按默认顺序组成树的叶子节点并成为该方法的子节点,各个类则将所属的成员方法作为子节点并将应用的根节点作为父节点,从而得到一个树形结构.为了提升效率,TVG 构建算法被设计得尽可能简单.构建算法的过程如下:启动迭代检测污染变量,即逐一分析树形结构的所有叶子节点(语句),当语句满足任何规则的右部条件时,如果生成新的序偶关系,则构成新的污染变量和 TVG 边;如果生成新的谓词,则进行记录.另外,已加入 TVG 的污染变量节点不会重复添加,

www.jos.org.cn

以避免形成回路.当一轮迭代结束时,如果此轮迭代生成新的 TVG 边或谓词,则启动新一轮迭代继续检测;否则,算法结束.需要说明的是:启动新一轮迭代主要是因为在前一轮迭代中,新生成的污染变量或谓词可能在下一轮迭代产生新的污染变量或谓词,需要进一步检测.当程序中存在较复杂的污染流时,例如跨越多个方法或者包含较多别名,算法可能需要更多的迭代,从而找出所有污染变量.

我们以图 1 程序为例,算法在分析图 1(a)方法中语句 3~语句 5 时,分别使用 R1,R8,R10 生成序偶$\langle SC_1,a\rangle$,$\langle a,b\rangle$和谓词 *Param*(*Send*,1,*b*);在分析图 1(b)方法时,语句 1、语句 3、语句 4 分别使用 R4,R7,R14(语句 1 在 Jimple 代码中对应一条参数声明语句),生成序偶$\langle b,x\rangle$,$\langle x,y\rangle$,$\langle y,SK_1\rangle$,由此在一次迭代中可得到 TVG 边集 $E=\{(SC_1,a),(a,b),(b,x),(x,y),(y,SK_1)\}$.在下一次迭代中,算法没有更新 TVG 或生成谓词,因此算法结束.值得说明的是:一些赋值语句既可满足别名规则 R9 生成污染变量,也可满足清除规则 R16 生成 *Clean* 谓词.因为在现实程序中,一些别名关系的形成本身伴随清除污染变量的效果.例如:*x* 是污染变量且是堆变量,如果赋值语句 *x*=*y* 发生在 *x* 被污染之前,则仅仅是别名关系而没有清除作用;但如果发生在 *x* 被污染之后,则 *x* 被清除并指向 *y* 的内存,形成与 *y* 的别名关系.因此,我们在使用规则时,对被清除的变量使用清除标记表明该变量被清除,在 TVG 中并不删除该节点,并同样可以传播新的污染变量.而在之后的污染流验证过程,算法会根据具体语句执行顺序进行判定.

2.5 污染流提取和验证

根据 TVG 的性质,如果图中存在 *SK* 节点,则一定存在从 *SC* 到 *SK* 的可达路径.该路径上的顶点形成一条从污点源经过污染变量到泄露点的变量序列,这条变量序列所表示的污染流,我们称为潜在污染流.路径上的边实际上对应程序中发生污染传播过程的语句,路径上的边序列能够确定一条语句序列,我们称之为潜在污染路径.因为 TVG 是根据污染变量的关系构建的,潜在污染流的生成没有考虑语句执行顺序和函数调用点情况,所以是不准确的.为了提高精度,我们需要在函数调用图和控制流图找出一条合理的可执行路径,验证潜在污染流所对应的潜在污染路径能够成功得到执行,从而证明潜在污染流的可行性.

一般来说,验证潜在污染路径就是要寻找一条可执行路径,使其完全符合潜在污染路径的语句顺序.图 3 中展示了 3 个不同程序片段的控制流图.图 3(a)中的程序在构建 TVG 后,可以得到从 *SC* 到 *SK* 的可达路径为(*SC*,*b*,*c*,*SK*),对应的潜在污染路径是(2,3,4),我们在验证此潜在污染路径时,只需沿着控制流图的执行顺序一一确认,即可验证该潜在污染流成立,如图中蓝色曲线所示.但是,当涉及别名规则生成的污染变量和被清除的污染变量时,我们需要根据情况判断潜在污染路径.在图 3(b)中的程序,TVG 的可达路径是(*SC*,*b.str*,*c.str*,*SK*),对应的潜在污染路径是(3,2,4),不存在可执行路径满足该潜在污染路径.由于变量 *b.str* 对 *c.str* 的污染传播是在 *b* 被污染前的别名声明中发生的,我们对由别名规则产生的污染变量的语句增加一条反向的搜索,如图中红色曲线所示,之后可找到一条执行路径(2,3,4)满足条件,由此可验证污染流成立.需要说明:假如别名声明语句 *b*=*c* 出现在语句 3 之后,虽然别名关系可以生成,但语句同时满足变量 *b* 的清除规则,所以无法形成污染流.假如别名声明语句 2 改为 *c*=*b*,该语句无论出现在语句 3 之前还是之后都可以形成污染流.在图 3(c)中,污染变量 *a* 在语句 3 中被赋值为 *null*,此时污染变量 *a* 被清除,而在 TVG 中仍会得到可达路径(*SC*,*a*,*SK*)和潜在污染路径(2,4),但我们定义的清除规则中会标记变量 *a* 并记录语句 3,在搜索可执行路径时,如果在语句 2 后面存在语句 3,则终止此条路径的搜索并验证此污染流不成立.验证潜在污染流的相关算法和实现细节我们会在下一节详细解释.

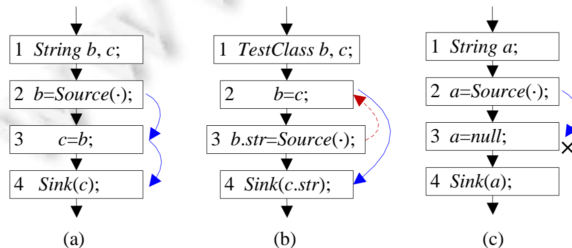


Fig.3 Control flow graph of code snippets with taint flows

图 3 含有污染流的代码片段的控制流图

2.6 方法对各敏感性的支持情况

我们提出的污点分析方法的主要环节在于 TVG 的构建和污染流提取及验证.构建 TVG 时,根据规则分析每条语句并生成污染变量作为新的 TVG 顶点.如前所述,我们使用访问路径表示污染变量,访问路径的生成需解析 Jimple 语句中的变量对象,从而确定污染变量具体的变量名或类名和域名,所以此阶段的分析可以支持对象敏感和域敏感.然而,TVG 构建过程中没有考虑语句的执行顺序和函数调用点信息,所以属于流不敏感和上下文不敏感的分析.为了提高分析精度,我们对污染流进行提取及验证,在控制流图上搜索一条符合条件的可执行路径,此阶段不仅考虑语句的执行顺序,而且在污染流跨越多个函数时,根据污染变量的上下文信息分析函数调用情况,所以经过该阶段的验证后,工具可以支持流敏感和上下文敏感.综上,FastDroid 工具的污点分析能够支持流敏感、上下文敏感、域敏感及对象敏感.

3 FastDroid 工具的实现

3.1 工具架构

根据基于污染变量关系图的污点分析方法,我们开发了污点分析工具 FastDroid 用于检测 Android 应用中隐私数据的泄露.工具基于 Eclipse 平台开发,由大约 9 000 行 Java 代码组成,使用 Maven 进行项目管理.工具使用了一些公开项目,包括 FlowDroid-2.5[23]、IC3[15]、JGraphT 图形库[24]等.其中:FlowDriod-2.5 是 FlowDroid 1.0 工具的升级版本,它基于 Soot 框架和 IFDS 数据流框架[25]实现了精确的污点分析,并集成了 IccTA 模块用以支持组件间通信分析,FastDroid 主要使用 FlowDroid-2.5 的 app 解析和系统环境建模相关模块;IC3 是一个独立的分析应用中组件间通信的工具,可获取应用中显式和隐式的组件调用;JGraphT 库是一个免费的图形运算库,提供了图论相关的常用数据结构和算法,FastDroid 基于 JGraphT 库进行图、路径的相关运算.图 4 展示了 FastDroid 的整体架构.工具从功能上可以分为 4 个模块:配置文件解析模块、App 解析及建模模块、TVG 构建模块、污染流提取及验证模块.工具的输入是待分析应用的 APK 文件及一些配置文件,输出为污染流信息,即隐私泄露信息.目前,我们已经公开了 FastDroid 工具,地址为:https://github.com/zhangjie-xd/FastDroid.

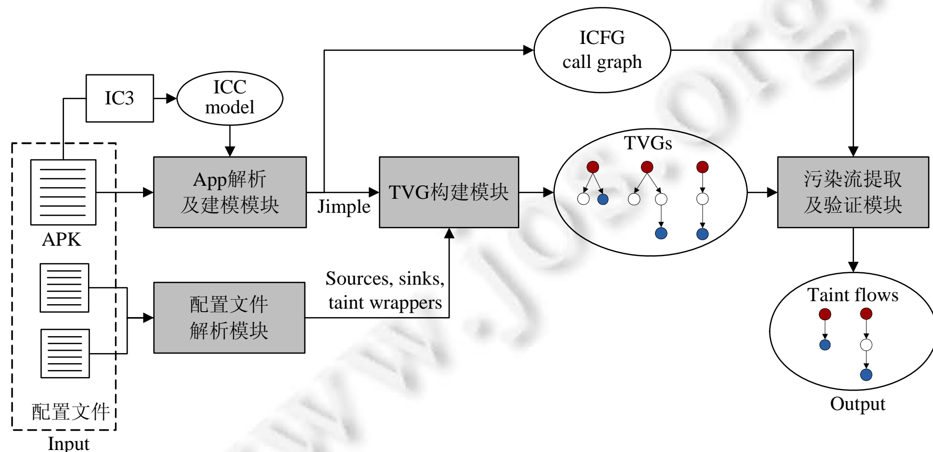


Fig.4 Framework of FastDroid

图 4 FastDroid 工具框架

3.2 配置文件解析模块

FastDroid 的配置文件包括污点源和泄露点 API 列表文件和污染封装 API 列表文件.污点源和泄露点 API 列表文件存储了 Android SDK 中获取和泄露隐私数据的所有 API 集合.FastDroid 使用 SUSI[20]提供的经过分类的污点源和泄露点列表,每个 API 由其所属类、方法名、参数值、返回值这 4 项信息进行描述.另外,我们收集

了大约 600 个污染封装 API 形成列表文件,每个 API 同样以所属类、方法名、参数值、返回值这 4 项信息进行描述.配置文件读取模块将这两个 API 列表文件读入程序中,解析 API 的描述信息并生成唯一的方法特征签名.

3.3 App解析及建模模块

该模块主要利用 FlowDroid 的 app 解析和建模相关模块对输入的 APK 文件进行解析和反编译,生成 Jimple 中间表示[17],并进行 Android 应用系统环境的模拟.同时,我们需要运行 IC3 工具分析应用中的组件间通信信息,生成 ICC-model 文件作为模块的另一个输入.整个模块运行步骤如下:首先,使用 Dexpler[26]插件将 app 中的 dex 文件转换为 Jimple 码;然后,调用 FlowDroid 的 constructCallgraph 接口为每一个组件生成 dummyMain 虚拟主函数;接着,根据 ICC-model 文件的组件间通信信息,运行集成的 IccTA 模块生成分析组件间通信的辅助对象和函数,用于模拟组件间显式和隐式通信的函数调用关系.至此,我们就可以根据以上信息生成完整的函数调用图(call graph)及过程间控制流图(inter-procedural control-flow graph,简称 ICFG).

在对 app 解析后,工具通过 Soot 提供的接口在 Jimple 中间表示上获取程序分析所需的各类信息.Jimple 语言作为 Soot 最主要的中间表示,是一种基于语句的类型化和三地址的中间表示[17],相比 Java 字节码的 200 多种语句,Jimple 仅定义了 15 种核心语句,使得分析更加简便,包括赋值语句 AssignStmt、声明语句 IdentityStmt、方法调用语句 InvokeStmt 等等.Jimple 的每条语句最多涉及 3 个变量或常量,所以 Java 源码中的一条语句可能对应多条 Jimple 语句.Soot 作为一个成熟的 Java 分析框架,为程序静态分析提供了多种数据结构和接口.常见的类包括 Scene,SootClass,SootMethod,SootField,Stmt,Value 等:通过 Scene 可获取整个程序的基本信息;SootClass 包含类的基本信息,从中可以获得类的域和方法,分别用 SootField,SootMethod 封装;SootMethod 描述类的方法,并可获取方法中 Jimple 语句集合,每条 Jimple 语句由 Stmt 描述,Stmt 是语句类型的父类,根据类型的不同可分成多种子类;而 Stmt 语句操作的常量、变量等信息封装在 Value 中.如图 5 所示:FastDroid 工具基于 Soot 提供的数据结构,将整个应用抽象为一个树型结构.树的叶子节点为 Stmt 类型的对象用于表示语句,其父节点为 SootMethod 类型的对象用于表示成员方法.方法的父节点为 SootClass 类型的对象表示类,类的父节点为整个应用.在分析各语句时,我们可以通过向上追溯父节点来获取分析需要的方法、类及应用的相关信息.

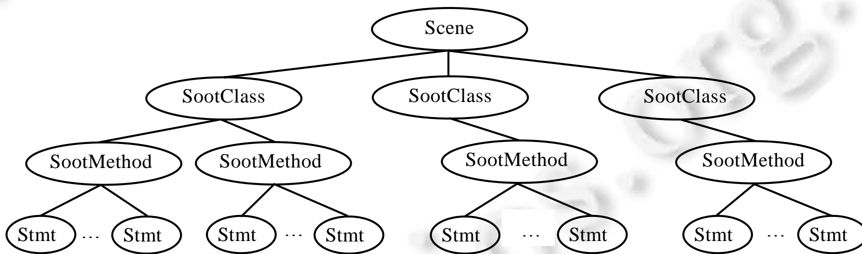


Fig.5 Tree structure of the app

图 5 应用的树形结构

3.4 TVG构建模块

TVG 构建模块主要功能是根据污染传播规则分析所有语句,当发现符合规则的语句时生成污染变量或谓词,最后构建出 TVG.为了提高分析速度,FastDroid 遍历树形结构中的叶子节点(语句),在此过程中不依赖于函数调用图和控制流图,不考虑语句的执行顺序,仅根据污染传播规则对语句及包含的变量进行轻量级分析,分析时没有类似数据流框架中的复杂控制和流方程的运算.当语句满足规则并发现新的污染变量时,工具生成新的顶点和边,并将其加入到 TVG 中,TVG 在各语句的分析过程中不断扩充.当语句满足规则并发现新的谓词时,算法根据谓词情况生成相应的方法或者变量的标记.如前所述,一个应用可能包含多个污点源,并生成多个 TVG,在构建时,每个 TVG 中涉及的污染变量和谓词等数据都是独立的,而分析一条语句时会考虑每一个 TVG 的更新和扩展,所以多个 TVG 的构建实际上是同步进行的,这区别于 FlowDroid 使用多线程来追踪不同污染流.另外,构建 TVG 可能需要经过多次迭代才能完整.在实现中,当本轮迭代得到新的污染变量加入 TVG 或新的标记时,

算法将启动下一次迭代,直到 TVG 不再更新或不产生新的标记时,算法终止.

我们设计了一些重要的类用于构建 TVG,包括 TVG 类、TaintedValue 类和 Context 类.如图 6 所示:TaintedValue 类描述一个具体的污染变量,其成员分别表示名称、类型、所属类、所属方法、变量种类(静态变量、动态变量或者成员变量等)、上下文信息、污染传播方式、清除标记及位置等,TaintedValue 类的名称为变量的访问路径;Context 类记录了污染变量产生语句的上下文信息,包含 SootClass 对象、SootMethod 对象以及用于唯一标识语句位置的 StmtTag 类型的成员;TVG 类用于存储一个 TVG,该类包含了唯一的 ID 值、一个有向图 graph 和一个 TaintedValue 对象的集合.有向图 graph 是以 TaintedValue 对象为顶点,以污染传播关系为边的有向图结构,是 TVG 图形的具体实现.

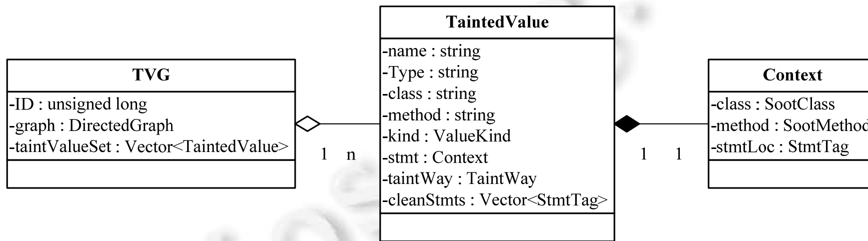


Fig.6 Class diagram of the TVG

图 6 TVG 的类图

下面介绍构建 TVG 时分析语句的一些基本操作:1) 发现污染源时,创建新的 TVG 并进行初始化,生成一个特殊的 TaintedValue 对象 Source 加入到 TVG 中;2) 当判定某变量为污染变量时,生成一个新的 TaintedValue 对象,将其加入所属的 TVG 中,并添加连接该对象与产生它的已知污染变量的有向边;3) 发现污染变量被泄露时,生成一个特殊的 TaintedValue 对象 Sink,并添加连接该污染变量和 Sink 的有向边;4) 当标记某方法时,生成方法的标记对象(包括 Param,Retrun,ReturnP 这 3 种类型的标记),标记对象包含方法调用点、污染变量和参数或者返回值等信息;5) 当污染变量被清除时,生成清除污染变量的标记,标记对象为 stmtTag 类型描述的语句位置,添加标记对象到该污染变量的 TaintedValue 对象中的 cleanStmts 成员中.下面具体说明不同污染传播规则下基于基本操作执行的详细操作.

- 污染源规则:对于赋值语句,如果其右值是方法调用表达式且方法满足污点源方法特征签名,则发现污染源并判定左值为污染变量;
- 污染变量传播规则:
  - 对于赋值语句,当出现以下情况时,判定左值为污染变量:1) 右值是污染变量且不是堆变量;2) 右值是带 Return 标记的方法调用表达式;3) 右值是满足污染封装方法特征签名的方法调用表达式且参数是污染变量;
  - 对于声明语句,当声明的变量是带 Param 标记方法的相应形参时,该变量被判定是污染变量;
  - 对于赋值语句或者方法调用语句,如果语句中包含方法调用表达式且方法带 ReturnP 标记,则相应实参被判定为污染变量;
- 别名规则:对于赋值语句,当左值是堆变量且左值是污染变量时,则右值被判定为污染变量;当右值是堆变量且右值是污染变量时,则左值被判定为污染变量;
- 方法调用规则:对于赋值语句或方法调用语句,如果语句中包含方法调用表达式且方法的实参为污染变量时,生成该方法的 Param 标记;
- 方法返回规则:对于返回语句,如果返回值是污染变量,生成该方法的 Return 标记;对于声明语句,如果该方法声明的形参为污染变量且是堆变量时,生成该方法的 ReturnP 标记;
- 污染泄露规则:对于赋值语句或者方法调用语句,如果语句中包含方法调用表达式,方法满足泄露点方法特征签名且相应参数是污染变量,则该变量被泄露;

- 清除规则:对于赋值语句,若左值是污染变量,右值为常量、空值或非污染变量,则该污染变量被清除;
- 特殊规则:当识别出特殊规则规定的污染方式时,执行对应的特殊操作.

3.5 污染流提取及验证模块

在 TVG 构建完成之后,污染流提取及验证模块首先使用 GraphT 图形库计算 TVG 中所有从顶点 *SC* 到顶点 *SK* 的可达路径,每条可达路径代表一条潜在污染流.路径由以 TaintedValue 对象为元素的数组($tv_1,tv_2,\ldots,tv_n$)表示,tv_1 和 tv_n 分别表示顶点 *SC* 和 *SK*.然后,工具根据函数调用图和控制流图中的函数调用点和语句执行顺序相关信息验证潜在污染流的可行性,TaintedValue 对象中包含的 Context 类型的 stmt 成员提供了语句位置信息,我们按照该语句位置在控制流图上定位语句,并分段搜索是否存在符合要求的可达路径,即,每次确认相邻两个 TaintedValue 对象的 stmt 语句之间是否存在可达路径.最后,工具将删除虚假的潜在污染流,保留并输出可行的污染流.算法 1 给出了验证潜在污染流可行性的具体步骤,算法输入为 TVG 上的一条潜在污染流以及 app 的函数调用图和控制流图,输出为表示潜在污染流是否可行的布尔值.算法的大致思路是:分段搜索符合要求的可执行路径,在搜索中遇到因别名规则产生的变量时会启动额外的反向路径搜索;另外,每段符合要求的路径上不能存在当前变量或者其别名的清除语句.算法的具体解释如下.

第 1 行、第 2 行进行变量的初始化,其中:*tv_c*,*tv_d* 分别表示污染流每段的当前污染变量和目标污染变量,*cleanSet* 存储清除当前污染变量及其别名变量的语句集合.

第 3 行启动循环,对潜在污染流分段进行验证,在确定所有污染变量后循环结束.

第 4 行~第 11 行:当 *tv_d* 是由别名规则生成的污染变量时,首先搜索从 *tv_d* 到 *tv_c* 的反向路径,如果存在反向路径且路径不包含别名变量 *tv_d* 的清除语句,则在 *cleanSet* 中添加别名污染变量 *tv_d* 的清除语句,将 *tv_d* 更新为下一个污染变量,*tv_c* 不改变,并跳过当前循环;如果反向路径包含别名变量 *tv_d* 的清除语句使得别名关系不成立,则返回 false.

第 12 行~第 19 行:如果不存在从 *tv_d* 到 *tv_c* 的反向路径,则搜索从 *tv_c* 到 *tv_d* 的前向路径,如果存在前向路径且路径不包含 *cleanSet* 中的清除语句,则在 *cleanSet* 中添加别名污染变量 *tv_d* 的清除语句,并且将 *tv_c* 更新为 *tv_d*,将 *tv_d* 更新为下一个污染变量;如果不存在符合条件的路径,则返回 false.

第 20 行~第 27 行:当 *tv_d* 不是通过别名规则生成的污染变量时,搜索从 *tv_c* 到 *tv_d* 的前向路径.如果存在路径且路径不包含 *cleanSet* 中的清除语句,将 *cleanSet* 更新为 *tv_d* 的清除语句,并且将 *tv_c* 更新为 *tv_d*,将 *tv_d* 更新为下一个污染变量;如果不存在符合条件的路径,则返回 false.

第 28 行、第 29 行:如果循环正常结束,则潜在污染流中的每一段都被验证,返回 true.

算法 1. 污染流验证算法.

输入:*Flow*=($tv_1,tv_2,\ldots,tv_n$),*CG*,*CFG*;

输出:true or false /*污染流是否可行*/.

```
1.  tv_c=tv1; i=2; tv_d=tvi;
2.  cleanSet=null;
3.  while i≤n do
4.    if tv_d.taintWay==TaintWay.ALIAS
5.      path=findPath(tv_d.stmt,tv_c.stmt);   //反向路径搜索
6.      if path!=null
7.        if checkOnPath(path,tv_d.cleanStmts)==false   //检查 path 上是否存在 tv_d 的清除语句
8.          cleanSet.addAll(tv_d.cleanStmts);   //添加清除语句到 cleanSet
9.          i++; tv_d=tvi; continue;
10.       else return false;
11.       end if
12.     else
```

```
13.         path=findPath(tv_c.stmt,tv_d.stmt);   //前向路径搜索
14.         if path!=null && checkOnPath(path,cleanSet)==false
15.            cleanSet.addAll(tv_d.cleanStmts);   //添加清除语句到 cleanSet
16.            tv_c=tv_d; i++; tv_d=tv_i;
17.         else return false;
18.         end if
19.      end if
20.   else
21.      path=findPath(tv_c.stmt,tv_d.stmt);   //前向路径搜索
22.      if path!=null && checkOnPath(path,cleanSet)==false
23.         cleanSet.updateTo(tv_d.cleanStmts);   //更新 cleanSet 的清除语句
24.         tv_c=tv_d; i++; tv_d=tv_i;
25.      else return false;
26.      end if
27.   end if
28.   end while
29.   return true;
```

4 实验及分析

为了验证工具 FastDroid 的精度和效率,我们在多个测试集上分别对 FlowDroid-2.5[22]和 FastDroid 进行了实验及结果分析.实验使用了 FlowDroid-2.5 工具默认的参数设置,采用的机器配置是 Intel i7 CPU(4.0GHz)和 16GB RAM,操作系统采用 Ubuntu 16.04 系统.实验工具 FlowDroid-2.5 和 FastDroid 都使用项目管理工具 Maven 打包生成 Jar 文件,然后用 Shell 脚本运行 Jar 文件,逐一对测试集中的应用进行批处理并记录实验结果.

4.1 测试集

本文实验使用了 3 个测试集:(1) DroidBench-2.0[27],一个专门检测 Android 应用污点分析工具精度的测试集,该测试集是 DroidBench-1.0[6]的升级版,包含 119 个开源应用,应用中包含的污染流种类多样,能够有效测试工具能否全面地识别各类污染流;(2) MalGenome[28],该测试集包含 49 种不同类型的 1 260 个 Android 恶意应用样本;(3) 从 Google Play 应用商店随机下载了 1 517 个较为热门的应用,应用类型包括游戏类、社交类、视频音频类、管理类等等,应用大小从 15KB~51MB 不等.

4.2 实验结果及分析

(1) 查准率(precision)和查全率(recall)

DroidBench-2.0 测试集的应用中一共存在 113 个污染流,FlowDroid 和 FastDroid 两个工具检测污染流的正确数、漏报数和误报数的统计结果如表 2 所示.FlowDroid 的查准率和查全率分别为 84.7%和 73.5%,而 FastDroid 查准率和查全率均高于 FlowDroid,分别为 93.3%和 85.8%.综合来看,FastDroid 的统计量 F-measure 为 0.89,高于 FlowDroid 的 0.79.FastDroid 实现高精度的主要原因如下:首先,FastDroid 工具与 FlowDroid 工具一样能够支持流敏感、上下文敏感、域敏感及对象敏感的分析;其次,我们对 DroidBench 测试集的源码进行了深入研究,对 FlowDroid 无法检测的一些复杂污染流的污染方式进行分析和抽象,制定了专门的特殊规则并加入到分析算法中.例如:FastDroid 设计了针对隐式污染流(implicit flow)的特殊污染传播规则,对于隐式污染流类型的污点分析 FastDroid 结果明显好于 FlowDroid.由于测试集 DroidBench 的测试用例都是开源的,我们可以判断工具的检测结果是否准确;而对于 Google Play 和 MalGenome 两个测试集,我们很难获取应用中准确的污染流信息,因此没有统计这两个测试集上的查准率和查全率.

Table 2 Comparison of FlowDroid and FastDroid on DroidBench

表 2 FlowDroid 和 FastDroid 在 DroidBench 上的对比

	FlowDroid	FastDroid
正确数(c)	83	97
漏报数(m)	30	16
误报数(f)	15	7
查准率:$p=c/(c+f)$	84.7%	93.3%
查全率:$r=c/(c+m)$	73.5%	85.8%
F-measure: $2pr/(p+r)$	0.79	0.89

(2) 运行时间

表 3 统计了 FlowDroid 和 FastDroid 在 3 个测试集上检测的污染流数量和运行时间.因为 FlowDroid 和 FastDroid 使用了相同的模块对应用进行反编译和建模,所以统计的运行时间是除去工具反编译和建模的开销后花费在污点分析的时间.实验结果表明:FastDroid 在 3 个测试集上花费的平均时间、最大时间和最小时间均小于 FlowDroid.特别是在 MalGenome 上,FastDroid 的最大时间为 17.61s,远远小于 FlowDroid 的 396.96s.

Table 3 Comparison of runtime of FlowDroid and FastDroid

表 3 FlowDroid 和 FastDroid 的运行时间对比

工具	FlowDroid			FastDroid		
测试集	DroidBench	Google play	MalG	DroidBench	Google play	MalG
应用数量	119	1 517	1 260	119	1 517	1 260
污染流数	83	26 644	11 981	97	17 908	15 072
平均时间(s)	0.45	67.99	15.80	0.05	6.20	1.59
最大时间(s)	1.17	458.83	396.96	0.14	292.34	17.61
最小时间(s)	0.01	0.09	0.19	0.003	0.01	0.01

为了反映运行时间与程序存储大小的关系,我们将工具在 Google play 测试集上的实验结果绘制成散点图.图 7 所示的 FlowDroid 运行时间较分散,大部分应用的运行时间在 50s 以下,但不少应用花费超过 200s.相比而言,图 8 中 FastDroid 的运行时间绝大多数小于 50s,超过 50s 的应用不超过 10 个.由此可得出,FastDroid 的运行时间更稳定.

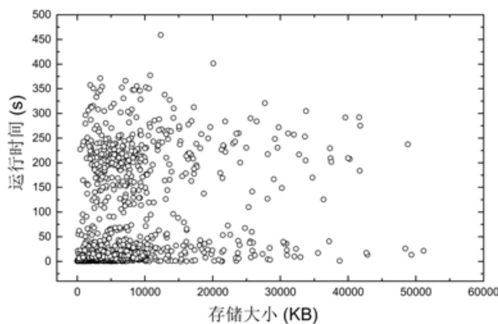


Fig.7 Runtime of FlowDroid on Google play

图 7 FlowDroid 在 Google play 上的运行时间

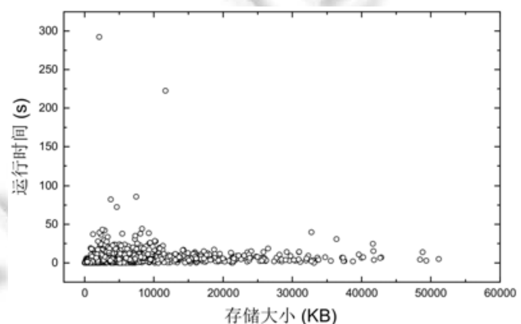


Fig.8 Runtime of FastDroid on Google play

图 8 FastDroid 在 Google Play 上的运行时间

5 相关工作

5.1 静态污点分析方法

近年来,研究人员已经提出了若干面向 Android 应用的精确的静态污点分析方法.首先,我们重点介绍与 FastDroid 最相关的 FlowDroid 工具,其次对其他方法进行简要介绍.

FlowDroid[6]是目前十分流行的开源静态污点分析工具,自该工具提出以来,其性能不断得到改进且版本持

续更新.FlowDroid 基于 Java 程序分析框架 Soot,采用 Jimple 语言作为中间表示.该工具主要具有以下创新点.

- 第一,实现了对 Android 系统环境的建模.FlowDroid 根据 Android 应用的特点建立了一个虚拟主函数 *dummyMain* 模拟系统环境,虚拟主函数直接调用了各组件中的生命周期函数和回调函数,因此,FlowDroid 可生成完整的函数调用图,并解决了 Android 应用多入口的问题;
- 第二,创新性地设计了按需反向别名分析,使得污染变量的别名可根据需要及时得到追踪,并在确定新的别名后启动新的线程对别名变量进行前向污点分析;
- 第三,设计了专门用于评价污点分析工具的测试集 DroidBench[27],该测试集中的应用包含各种类型的污染流且近年来不断得到扩充和升级.IccTA[7]工具对 FlowDroid 工具进行了扩充,该工具使用程序静态插桩的方法增加了组件间分析,并为每个组件都单独创建一个虚拟主函数以模拟该组件的生命周期,使得分析的查准率和查全率都得到提高.

Amandroid[8]是一个通用的 Android 应用静态分析平台,可用于检测隐私数据泄露、数据注入及 API 误用滥用等多种问题.该工具采用完全不同于 FlowDroid 的技术框架,使用 Scala 语言实现.Amandroid 对 Android 系统环境同样进行了模拟,在组件层面建立一个虚拟主函数 *ENV_C*,并为每个组件都建立了数据流图 DFG 和数据依赖图 DDG.Amandroid 与 FlowDroid 在别名分析上有很大不同,Amandroid 在污点分析前首先进行大规模的别名分析,计算出所有对象(包括非污染变量)的别名信息.与之相比,FlowDroid 仅对污染变量进行按需的反向别名分析从而减少开销,而我们提出的 FastDroid 工具将别名分析与污点分析结合,进一步提升了效率.DroidSafe[9]是一个开源的面向 Android 应用的支持组件间分析的污点分析工具,该工具基于 Android 开源项目(AOSP)的分析对 Android 应用环境建模,并基于所建立的 ADI 模型进行信息流分析,但该工具不支持流敏感的分析.在 Pauck 等人[29]的论文实验中,DroidSafe 的精度和运行效率都不如 FlowDroid.Apposcopy 工具[5]是一种基于语义的分析,可以检测泄露隐私的 Android 恶意应用.该工具基于 Soot 框架并使用 Jimple 中间表示,定义了一系列谓词用于描述应用中的某些特征,并生成恶意软件家族的特征签名;同样也定义了相应谓词描述污染流,并在构建的组件间函数调用图(ICCG)上进行污点分析,但该工具无法从网络上公开获取.

5.2 提升污点分析效率的方法

目前,研究人员提出了一些用于改进静态污点分析效率的方法.Cai 等人[30]设计了一种可以减少污点分析搜索空间的方法,该方法基于对旧版本应用的分析结果,在分析该应用升级后的新版本时,仅对有差异的代码进行分析.这种方法可以有效提高效率,但对于全新的应用并没有优势;而 FastDroid 可以检测全新的应用.SparseDroid 工具[31]对 FlowDroid 中的传统 IFDS 框架算法进行稀疏化的改进,从而提升了运行效率,其沿用了 FlowDroid 其他组件,没有提升检测的精度.

6 总结与未来工作

本文针对精确污点分析技术存在的效率低、运行慢的问题,提出了一种基于污染变量关系图的污点分析方法.该方法支持流敏感、上下文敏感、域敏感、对象敏感的分析,在保证精度的基础上提高了分析效率,降低了分析时间.本文阐述了方法涉及到的污染变量关系图、污染传播规则等概念,详细叙述了基于该方法实现的 FastDroid 工具的架构、模块及各模块细节.最后,在 3 个不同测试集上,与主流工具 FlowDroid 进行了实验对比,结果表明:FastDroid 的查准率和查全率达到 93.3%和 85.8%,高于目前主流工具 FlowDroid;同时,FastDroid 的运行时间更少且更稳定.

本文方法和工具仍然存在一些不足:工具不能支持利用反射机制进行的隐私泄露、工具在个别情况下无法正确检测或存在误报、工具的运行效率仍存在提升空间等.在未来工作中,我们将继续完善污染传播规则,优化 TVG 构建和污染流验证的算法,进一步提高工具精度和效率.

References:

[1] Mobile operating system market share worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide

[2] Mobile phone security report in 2019 (in Chinese). http://zt.360.cn/1101061855.php?dtid=1101061451&did=610435085

[3] Wang L, Li F, Li L, Feng XB. Principle and practice of taint analysis. Ruan Jian Xue Bao/Journal of Software, 2017,28(4):860–882 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5190.htm [doi: 10.13328/j.cnki.jos.005190]

[4] Yang W, Xiao X, Andow B, Li S, Xie T, Enck W. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In: Proc. of the 2015 IEEE/ACM 37th IEEE Int'l Conf. on Software Engineering. 2015. 303–313. [doi: 10.1109/ICSE.2015.50]

[5] Feng Y, Anand S, Dillig I, Alex Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. 2014. 576–587. [doi: 10.1145/2635868.2635869]

[6] Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. ACM Sigplan Notices, 2014,49(6):259–269. [doi: 10.1145/2594291.2594299]

[7] Li L, Bartel A, Bissyandé TF, Klein J, Le Traon Y, Arzt S, Rasthofer S, Bodden E, Octeau D, McDaniel P. Iccta: Detecting inter-component privacy leaks in Android apps. In: Proc. of the 2015 IEEE/ACM 37th IEEE Int'l Conf. on Software Engineering. 2015. 280–291. [doi:10.1109/ICSE.2015.48]

[8] Wei F, Roy S, Ou X. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. ACM Transactions on Privacy & Security, 2018,21(3):1–32. [doi: 10.1145/3183575]

[9] Gordon MI, Kim D, Perkins J, Gilham L, Nguyen N, Rinard M. Information-Flow analysis of Android applications in droidsafe. In: Proc. of the Network and Distributed System Security Symposium. 2015,15(201):110. [doi: 10.14722/ndss.2015.23089]

[10] Li L, Bissyandé TF, Papadakis M, Rasthofer S, Bartel A, Octeau D, Klein J, Le Traon Y. Static analysis of Android apps: A systematic literature review. Information & Software Technology, 2017,88:67–95. [doi: 10.1016/j.infsof.2017.04.001]

[11] Enck W, Gilbert P, Han S, Tendulkar V, Chun BG, Cox LP, Jung J, McDaniel P, Sheth AN. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. on Computer System, 2014,32(2):393–407. [doi: 10.1145/2619091]

[12] Zhu DY, Jung J, Song D, Kohno T, Wetherall D. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. ACM SIGOPS Operating Systems Review, 2011,45(1):142–154. [doi: 10.1145/1945023.1945039]

[13] Mei H, Wang QX, Zhang L, Wang J. Soft analysis: A road map. Chinese Journal of Computers, 2009,32(9):1697–1710 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2009.01697]

[14] Arzt S. Static data flow analysis for Android applications [Ph.D. Thesis]. Darmstadt: Technische Universität Darmstadt, 2017.

[15] Octeau D, Luchaup D, Dering M, Jha S, Mcdaniel P. Composite constant propagation: application to Android inter-component communication analysis. In: Proc. of the IEEE/ACM Int'l Conf. on Software Engineering. IEEE, 2015. 77–88. [doi: 10.1109/ICSE.2015.30]

[16] Octeau D, McDaniel P, Jha S, Bartel A, Bodden E, Klein J, Le Traon Y. Effective inter-component communication mapping in Android with epicc: An essential step towards holistic security analysis. In: Proc. of the 22nd USENIX Security Symp. 2013. 543–558.

[17] Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot: A Java bytecode optimization framework. In: Proc. of the CASCON 1st Decade High Impact Papers. 2010. 214–224.

[18] Rami K, Desai V. Performance base static analysis of malware on Android. Int'l Journal of Computer Science & Mobile Computing, 2013,2(9):247–255.

[19] Desnos A, Gueguen G. Android: From reversing to decompilation. In: Proc. of the Black Hat Abu Dhabi. 2011. 77–101.

[20] Rasthofer S, Arzt S, Bodden E. A machine-learning approach for classifying and categorizing Android sources and sinks. Network and Distributed System Security Symp., 2014,14:1125. [doi: 10.14722/ndss.2014.23039]

[21] Fritz C, Arzt S, Rasthofer S, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P. Highly precise taint analysis for Android applications. Technical Report TUD-CS-2013-0113. EC SPRIDE, 2013. http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf

[22] Huang SS, Green TJ, Loo BT. Datalog and emerging applications: An interactive tutorial. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. 2011. 1213−1216. [doi: 10.1145/1989323.1989456]

[23] FlowDroid data flow analysis tool. https://github.com/secure-software-engineering/FlowDroid

[24] Jgrapht. https://github.com/jgrapht/jgrapht/wiki

[25] Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In: Proc. of the 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. 1995. 49−61. [doi: 10.1145/199448.199462]

[26] Bartel A, Klein J, Le Traon Y, Monperrus M. Dexpler: Converting Android dalvik bytecode to jimple for static analysis with soot. In: Proc. of the ACM SIGPLAN Int'l Workshop on State of the Art in Java Program Analysis. 2012. 27−38. [doi: 10.1145/2259051.2259056]

[27] DroidBench 2.0. https://github.com/secure-software-engineering/DroidBench

[28] Zhou YJ, Jiang XX. Dissecting Android malware: Characterization and evolution. In: Proc. of the 2012 IEEE Symp. on Security and Privacy. IEEE, 2012. 95−109. [doi: 10.1109/SP.2012.16]

[29] Pauck F, Bodden E, Wehrheim H. Do Android taint analysistools keep their promises? In: Proc. of the 2018 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2018. 331−341. [doi: 10.1145/3236024.3236029]

[30] Cai H, Jenkins J. Leveraging historical versions of android apps for efficient and precise taint analysis. In: Proc. of the 15th ACM/IEEE Int'l Conf. on Mining Software Repositories. 2018. 265−269. [doi: 10.1145/3196398.3196433]

[31] He D, Li H, Wang L, Meng H, Zheng H, Liu J, Hu S, Li L, Xue J. Performance-Boosting sparsification of the ifds algorithm with applications to taint analysis. In: Proc. of the 2019 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2019. 267−279. [doi: 10.1109/ASE.2019.00034]

附中文参考文献:

[2] 2019年中国手机安全状况报告. http://zt.360.cn/1101061855.php?dtid=1101061451&did=610435085

[3] 王蕾,李丰,李炼,冯晓兵.污点分析技术的原理和实践应用.软件学报,2017,28(4):860−882. http://www.jos.org.cn/1000-9825/5190.htm [doi: 10.13328/j.cnki.jos.005190]

[13] 梅宏,王千祥,张路,王戟.软件分析技术进展.计算机学报,2009,32(9):1697−1710. [doi: 10.3724/SP.J.1016.2009.01697]

张捷(1986－),男,讲师,CCF 专业会员,主要研究领域为软件安全,恶意代码分析.

段振华(1948－),男,博士,教授,博士生导师,CCF 会士,主要研究领域为形式化方法,可信软件基础理论与方法.

田聪(1981－),女,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为软件安全,智能软件开发方法,可信软件基础理论与方法.