软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(2):211−230 [doi: 10.13328/j.cnki.jos.005581]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向 Android 应用隐私泄露检测的多源污点分析技术*

王 蕾[1,2], 周 卿[1,2], 何冬杰[1,2], 李 炼[1,2], 冯晓兵[1,2]

[1](计算机体系结构国家重点实验室(中国科学院 计算技术研究所),北京 100190)

[2](中国科学院大学,北京 100190)

通讯作者: 王蕾, E-mail: wanglei2011@ict.ac.cn

摘 要: 当前,静态污点分析检测 Android 应用隐私泄露存在误报率较高的问题,这给检测人员和用户带来很大的不便.针对这一问题,提出了一种多源绑定发生的污点分析技术.该技术可以精确地判断污点分析结果中多组源是否可以在一次执行中绑定发生,用户可以从单一分析 1 条结果转为分析有关联的多组结果,这既缩小了分析范围,又降低了检测的误报率.在精度上,该技术支持上下文敏感、流敏感、域敏感等特性,并可以有效地区分出分支互斥的情况.在效率上,提供了一种高效的实现方法,可以将高复杂度(指数级别)的分析降低为与传统方法时间相近的分析(初始阶段开销为 19.7%,进一步的多源分析平均时间为 0.3s).基于此,实现了一个原型系统 MultiFlow,利用其对 2 116 个良性手机软件和 2 089 个恶意手机软件进行应用,应用结果表明,多源污点分析技术可以有效地降低隐私泄露检测的误报率(减少多源对 41.1%).同时,还提出了一种污点分析结果风险评级标准,评级标准可以进一步帮助用户提高隐私泄露检测的效率.最后探讨了该技术潜在的应用场景.

关键词: 程序分析;污点分析;软件安全;静态分析;Android

中图法分类号: TP311

中文引用格式: 王蕾,周卿,何冬杰,李炼,冯晓兵.面向 Android 应用隐私泄露检测的多源污点分析技术.软件学报,2019,30(2):211−230. http://www.jos.org.cn/1000-9825/5581.htm

英文引用格式: Wang L, Zhou Q, He DJ, Li L, Feng XB. Multi-source taint analysis technique for privacy leak detection of Android Apps. Ruan Jian Xue Bao/Journal of Software, 2019,30(2):211−230 (in Chinese). http://www.jos.org.cn/1000-9825/5581.htm

Multi-source Taint Analysis Technique for Privacy Leak Detection of Android Apps

WANG Lei[1,2], ZHOU Qing[1,2], HE Dong-Jie[1,2], LI Lian[1,2], FENG Xiao-Bing[1,2]

[1](State Key Laboratory of Computer Architecture (Institute of Computing Technology, Chinese Academy of Sciences), Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100190, China)

Abstract: Currently, the results of static taint analysis cannot explain whether the application has privacy leaks directly (high false positives), which causes inconvenience to the detectors or users. Aiming at this problem, this study puts forward a new technique—multi-source binding taint analysis, which can determine whether multiple sets of sources occur in one execution precisely and efficiently. In terms of precision, the technique supports context sensitivity, flow sensitivity, and field sensitivity, and can precisely distinguish exclusive branches. In terms of efficiency, an efficient implementation method is provided to reduce high complexity (exponential level) to an analysis close to traditional method (initial overhead is 19.7%, further multi-analysis stage time is 0.3s). A prototype called MultiFlow is implemented, and it is applied to 2 116 benign Apps and 2 089 malicious Apps. Such results support the feasibility of multi-source

* 基金项目: 国家重点研发计划(2017YFB0202002); 国家自然科学基金(61521092, 61432016)

Foundation item: National Key Research and Development Program of China (2017YFB0202002); National Natural Science Foundation of China (61521092, 61432016)

收稿时间: 2017-07-29; 修改时间: 2017-10-01; 采用时间: 2018-04-04; jos 在线出版时间: 2018-04-27

CNKI 网络优先出版: 2018-04-27 14:58:13, http://kns.cnki.net/kcms/detail/11.2560.TP.20180427.1457.009.html

technique for precision enhancement of privacy leak detection (reducing multi-source pairs by 41.1%). Also, these characteristics are used as a risk rank standard of the Apps to improve detection convenience. Finally, the potential application scenarios of the technology are explored.

Key words: program analysis; taint analysis; software security; static analysis; Android

随着移动应用的广泛普及,其安全问题正受到严重的挑战.移动支付、电子商务、社交网络等活动中存在着大量的用户隐私数据,大量第三方移动应用程序的使用导致隐私数据难以被有效地保护.例如,印度的一家公司设计了一组智能手机应用开发工具包SilverPush,它可以嵌入到一个正常的手机应用中并在后台运行,在用户不知情的情况下收集用户的隐私数据(包括 IMEI ID、位置信息、视频与音频信息、Web 浏览记录等),并将其发送给广告推荐商[1].污点分析技术(taint analysis)[2]是信息流分析技术(information-flow analysis)[3]的一种实践方法.该技术通过对系统中的敏感数据进行标记,继而跟踪标记数据在程序中的传播,检测系统安全问题.据调查分析[4],在 Android 应用的静态安全检测中,污点分析则是最为流行的分析方案.

污点分析可以有效地检测 Android 应用的隐私泄露问题.举例说明,图 1 所示为一段 Android 应用程序代码,运行该段程序会导致用户的密码数据通过发送短信的方式泄露.污点分析首先要识别引入敏感数据的接口(source,污点源)并进行污点标记.具体到程序中,即识别第 4 行的 passwordText 接口为污点源,并对 *pwd* 变量进行标记.如果被标记的变量又通过程序依赖关系传播了给其他变量,那么根据相关传播规则继续标记对应的变量.当被标记的变量到达信息泄露的位置(sink,污点汇聚点)时,则根据对应的安全策略进行检测.图 1 中,第 8 行带污点标记的 *leakedMessage* 变量可以传播到发送信息的 *sendTextMessage* 接口,这就意味着密码数据会被该接口泄露.污点分析又分为静态和动态的污点分析.静态污点分析是指在不运行代码的前提下,通过分析程序变量间的数据依赖关系来进行污点分析.相对于动态运行的污点分析,静态污点分析可以在程序发布之前对应用程序安全问题进行检测,避免了发布之后造成的安全问题.另外,该技术具有分析覆盖率高(不依赖测试集合)、无需程序插桩(而导致的程序运行出现开销)等优势.

Fig.1 An example of taint analysis

图 1 污点分析示例

然而,在检测 Android 应用隐私泄露问题时,静态污点分析存在结果误报率较高的问题.这是由于为了能够发现所有潜在的恶意行为,污点分析需要将可能的敏感信息泄露路径都报告出来,而程序中正常功能也需要选定一些敏感的数据传播到外界.据统计,MudFlow[5]利用静态污点分析工具 FlowDroid 测试的 2 866 个良性软件(benign App)一共产生了 338 610 条污点分析结果,检测人员需要在大量集合中选出具有恶意行为的结果,这将是效率极低的.利用污点分析解决的隐私泄露问题(提高检测精度)是当前 Android 安全的重要研究热点.一些研究工作[5−8]尝试使用统计的方法进行恶意行为区分,例如,MudFlow[5]尝试对比污点分析结果和良性软件结果的差异,利用 SVM 分类模型检测出一个软件是否是恶意软件.然而,检测人员常常需要验证具体的泄露路径,该工具只能给出一个软件整体是否具有恶意行为(而且该工具也会存在误报),无法给出其中一条污点分析路径[*Source*→*Sink*]是否是直接导致隐私泄露的结果,这驱使我们尝试更加有效的分析方案.不同于其他方法,本文选择了另外一种假设作为切入点:我们假设良性软件和恶意软件所使用的敏感数据流之间的相关性(绑定发生特

性)是有差异的(我们将在第 5.1 节来验证该差异确实存在).例如,在良性软件中,地图信息和设备标识信息的验证往往不在一个模块中执行,而恶意软件则将设备信息与定位信息一起(绑定)发送到网络.然而,当前研究工作没有提供有效的多源分析技术.基于上述的分析与假设,本文提出了一种分析污点结果中多源之间是否绑定发生的污点分析技术.该技术可以给出污点分析结果之间是否可以在一次执行中绑定发生.此外,本文还提出了一种高效的实现方法,使得多次的多源分析开销很低.随后,本文将该技术应用到 Android 应用隐私泄露检测中.

总之,本文的主要贡献如下.

(1) 创新地提出了一种解决多源绑定发生问题的污点分析技术.该技术具有上下文敏感、流敏感、域敏感的特性,且可以精确地区分出分支条件路径互斥的情况.

(2) 提出了一种多源绑定发生技术的高效实现方法.该方法将高复杂度(指数级别)的多源问题进行近似,并提出了一种按需的多源污点传播方法.实验结果表明,该方法与传统污点分析相比,开销仅为 19.7%,且多次的多源分析开销很低(进一步分析的平均时间为 0.3s).

(3) 实现了原型系统 MultiFlow,应用其到 2 116 个良性手机软件和 2 089 个恶意手机软件的隐私泄露检测中,发现了良性软件和恶意软件的绑定发生特性有较大的差别,且本文方法和传统方法对比有较大的精度提升(减少多源对 41.1%).最后,我们还提出了一种污点分析结果风险评级标准,应用评级结果可以直接提高检测的效率和精度.

本文第 1 节对本文的研究背景进行介绍,包括静态污点分析的通用解决方案及 Android 隐私泄露检测相关技术.第 2 节使用实际例子演示本文的研究动机并分析该问题的难点.第 3 节介绍多源绑定发生的污点分析技术,包括问题定义以及提高精度和效率两方面的关键技术.第 4 节给出具体实验以验证多源技术的开销.第 5 节将该技术应用到实际隐私泄露检测中.第 6 节为相关工作.第 7 节对本文进行总结,并深入讨论该技术潜在的应用场景.

1 背景知识

1.1 IFDS框架

一般而言,静态污点分析问题都是被转化为数据流分析问题[9]进行求解:首先,根据程序中的函数调用关系构建调用图(call graph,简称 CG);其次,为了提供更细粒度地区分程序不同执行路径的分析,还需要构建控制流图(control flow graph,简称 CFG);最后,在过程内和过程间,根据不同的程序特性进行具体的数据流传播分析(与指针分析).在具体实现中,污点分析常常被设计为一种按需的分析算法:以污点源 source 函数为驱动源,分析其返回值(污点)在程序中的传播,直到汇聚点 sink 函数为止.

IFDS 框架[10]是一个精确高效的上下文敏感和流敏感的数据流分析框架,于 1996 年由 Reps 等人提出.IFDS 的全称是过程间(interprocedural)、有限的(finite)、满足分配率的(distributive)、子集合(subset)问题.该问题作用在有限的数据流域中,且数据值(data flow fact)需要通过并(或交)的集合操作满足分配率.满足上述限定的问题都可以利用 IFDS 算法进行求解(文献[10]中的 Tabulation 算法),而污点分析问题正是满足 IFDS 问题要求的.污点分析的数据流值是污点变量,表示当前污点变量可以到达程序点 Stmt.该算法的最坏的时间复杂度是 $O(ED^3)$,其中,E 是当前控制流图中的边个数,D 是数据流域的大小.

IFDS 问题求解算法的核心思想就是将程序分析问题转化为图可达问题[11].算法的分析过程在一个按照具体问题所构造的超图(exploded supergraph)上进行(如图 2 的例子所示),其中,数据流值可被表示成图中的节点.算法扩展了一个特殊数据流值:0 值,用于表示空集合;程序分析中转移函数的计算,即对数据流值的传递计算,被转化为图中的边的求解.为了更好地进行描述,算法根据程序的特性将分析分解成 4 种转移函数(边).

(1) Call-Flow,即求解函数调用(参数映射)的转移函数.

(2) Return-Flow,即求解函数返回语句返回值到调用点的转移函数.

(3) CallToReturn-Flow,即函数调用到函数返回的转移函数.

(4) Normal-Flow,是指除了上述 3 种函数处理范围之外的语句的转移函数.

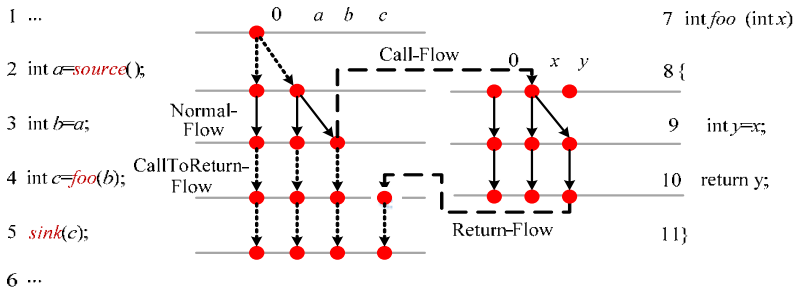


Fig.2 An example of taint analysis solved by IFDS algorithm

图 2 使用 IFDS 算法求解污点分析示例

IFDS 求解算法 Tabulation 是一种动态规划的算法,即求解过的子问题的路径可被重复地利用.由于其数据流值满足分配率,因此可以在分支或函数调用将边进行合并.求解算法包括了两类路径的计算:路径边(path edge)和摘要边(summary edge).

- 路径边表示的是过程内从起点到当前计算点的可达路径.
- 摘要边表示的是函数调用到函数返回的边,其主要特点是,如果在不同调用点再次遇到同样的函数调用,可以直接利用其摘要边信息,从而避免了函数内重复的路径边的计算.

下面以图 2 中的例子来演示 IFDS 算法是如何求解污点分析问题的,其中的数据流值表示由污点源传播来的污点变量.为了简洁,我们简写污点源函数为 *source*(),泄露点为 *sink*(*b*),其中,*b* 为泄露变量(下文同理).如第 2 行所示,此时将产生一条由 0 到 *a* 的边.对于第 3 行的赋值语句[*b*=*a*;]和 *a* 变量,Normal-flow 一般被定义为:如果赋值语句右值是污点变量,那么产生左值为污点变量且保留原值,即产生 $a\rightarrow b$ 和 $a\rightarrow a$ 的边.对于第 4 行的程序函数调用,由于 *b* 是函数 foo 的参数,分析将启动 Call-Flow 函数.Call-Flow 一般定义为:如果实参为污点变量,就会对应产生形参的污点变量,即产生 $b\rightarrow x$ 的边.对于第 10 行的程序函数返回点,Return-Flow 将产生返回语句的值到调用点返回值映射的边,即产生 $y\rightarrow c$ 的边.对于污点汇聚点 sink 函数,如果有数据流值经过该语句,例如图中的变量 *c*,则表示存在一条路径从 0 到达语句 *sink*(*c*),*c* 为最终触发 sink 的污点变量.对于 foo 中已经计算完成的入口到出口的边(图中 $x\rightarrow y$),分析将其保存为摘要边,之后,如果程序其他位置用 foo 时,则直接利用该摘要边即可.

1.2 FlowDroid

FlowDroid[12]是当前最流行的开源 Android 静态污点分析工具,目前被广泛地应用于检测 Android 隐私泄露和其他 Android 安全问题.FlowDroid 接受待分析的 apk 文件作为输入,利用反编译工具 Dexpler 和 Java 分析工具 Soot[13]将 apk 文件转化成 Soot 中间表示 Jimple,随后在 Jimple 上进行静态的污点分析.其分析的结果是多个污点源到污点汇聚点(*Src*→*Sink*)的集合.FlowDroid 提供了完整的 Android 函数调用图的构建,且提供了高精确度和高效率的污点传播分析.Android 框架中存在大量回调函数,导致 Android 程序的分析存在多入口的问题.FlowDroid 尝试对 Android 生命周期函数进行模拟,迭代地加入预先分析的入口函数(异步调用、回调函数等),使用 dummyMain 和虚拟节点连接成完整的调用图.在污点传播分析中,FlowDroid 正是基于 IFDS[10]数据流分析框架(提供了面向 Java 的 Call-Flow、Return-Flow、CallToReturn-Flow、Normal-Flow 的污点传播转移函数).FlowDroid 同时提出一种按需的后向别名分析方法,使用 Access Path 来支持域敏感.因此,FlowDroid 具有流敏感、上下文敏感、域敏感的高精确度.FlowDroid 使用 Susi[14]工具来生成在 Android 隐私泄露应用中使用的源和汇聚点;使用手工书写的摘要来对库函数的语义进行模拟(后续工作提供了 StubDroid[15]来自动生成库函数的摘要).FlowDroid 的初衷是提供一个敏感度高且高效的污点分析工具,然而其没有深入探索污点分析检测 Android 应用隐私泄露的问题.

2 研究动机

正如引言中所述,当前污点分析面临的一个重要研究问题就是:污点分析结果无法回答一个应用程序是否具有隐私泄露(误报率高).本节用一个例子来说明本文的研究动机.我们选择两个应用程序的程序片段来演示当前污点分析检测隐私泄露问题的困难,其中,图 3 是从 Google Play[16]应用市场中下载的一个新闻应用程序的代码片段,它是一个没有隐私泄露行为的应用软件;另一个程序(如图 4 所示)是一个从 Genome 库[17]中得到的恶意软件 DroidKungfu 的变体.具体分析,新闻应用利用了地理位置的信息来提供用户当地的天气状况,图 3 左侧是其代码片段.同时,该应用的另一个常用功能就是提供设备号以便进行网络设备验证,图 3 右侧为其相关代码.然而,如图 4 所示的恶意软件片段在一个特殊的回调函数 *OnStartCommand* 下直接获取程序的地理位置信息和用户的设备号信息,然后传递给网络.它是以盗取用户敏感信息为目的的.

```
1  public void onLocationChanged(Location location){
2      String latLng=updateLocation(location);
3      ...
4  }

5  private String updateLocation(Location location){
6      String latLng;
7      if (location!=null){
8          double lat=location.getLatitude();
9          double lng=location.getLongitude();
10         latLng="Lati:"+lat+"Longi:"+lng;
11     } else {
12         latLng="Can't access your location";
13     }
14     return latLng;
15 }
```

```
1  public String currentDeviceMark(Context context){
2      TelephonyManager tm=(TelephonyManager)
         context.getSystemService(Context.TELEPHONY_SERVICE);
3      deviceId=tm.getDeviceId();
4      WifiManager wm=(WifiManager)context.
         getSystemService(Context.WIFI_SERVICE);
5      String wlanMacAddr=wm.getConnectionInfo.getMacAddress();
6      ...
7      If (deviceId!=null){
8          markId=deviceId;
9      } else if (wifi==State.CONNECTED||wifi==State.CONNECTING){
10         markId=wlanMacAddr;
11     } else if (...){
12     ...
13     }
14     ...
15 }
```

Fig.3 A code snippet of benign App

图 3 良性应用软件代码片段

```
1  protected int onStartCommand(){
2  ...
3  ArrayList list=new ArrayList();
4  List.add(new BasicNameValuePair("pid",tm.getDeviceId()));
5  List.add(new BasicNameValuePair("macAddr",tm.getConnectionInfo().getMacAddress()));
6  String loc="Lat:"+lm.getLatitude()+"Long:"+lm.getLongitude();
7  List.add(new BasicNameValuePair("loc",loc));
8  ...
9  ServerSession.postRequest(urlEncodedFormEntity(list));
10 }
```

Fig.4 A code snippet of malware

图 4 恶意软件代码片段

我们使用 FlowDroid 对两个应用程序进行隐私泄露检测.由于 FlowDroid 只是考虑单一的污点分析结果,分析上述两个应用程序的结果都是:{地理信息(*getLatitude*/*getLongitude*)发送到网络}和{设备信息(*getDeviceId*/*getMacAddress*)发送到网络}.即便我们尝试一些统计分析的方法,例如应用 MudFlow 中的统计结果(一种分类器)来区分,由于两者的结果是完全一样的,分类器也是无法区分两者的恶意行为的.因此,单独地依

据 FlowDroid 的结果判断隐私泄露,会导致该问题检测的误报,这促使我们尝试更加精化的分析方法.

进一步分析我们发现,在新闻应用中,由于发送信息是在不同的模块下,因此在一次执行过程中,必然是只能发送地理位置信息到网络就不会发送用户的设备信息,反之亦然.然而对比恶意软件,由于地理信息和用户设备等信息是恶意软件同时需要的,因此它们往往是绑定在一起发送到网络的.目前的污点分析只能提供单一的污点源和汇聚点的组合结果,并没有考虑多个组合之间的相关性关系.出于上述考虑,本研究的目标就是利用多个污点源到污点汇聚点组合的相关性来降低隐私泄露检测的误报率.更进一步来讲,我们研究在程序的一次执行(一次事件触发下)路径上,多个污点分析组合是否能够绑定发生在这一路径上.我们将该问题命名为多源绑定发生问题,而提供解决此问题的污点分析称为多源绑定发生的污点分析技术.

更加特殊的情况如图 3 右侧程序的分支 7 和分支 9 下,在设备验证模块中,由于某些情况(缺少 SIM 卡),getDeviceId 函数将返回空,这时需要使用设备的 Wifi 物理地址(*getMacAddress*)作为设备标识.FlowDroid 得到的结果是 *getDeviceId*→*NETWORK* 和 *getMacAddress*→*NETWORK*.然而在一次执行过程中,它们是不可能都被执行的,原因是 IF 分支的真分支和假分支是互斥的情况.由此可见,多源绑定发生的污点分析技术需要区分不同路径下的分析结果;其次,虽然在上述例子中没有出现,但分析还需要保证如 FlowDroid 一样的精度(上下文敏感、流敏感、域敏感等)以及可用范围内的开销.这些都给发掘污点分析结果之间的相关性带来了困难.因此,本研究的关键技术是提出一种精确高效的多源绑定发生的污点分析技术.

3 多源绑定发生的污点分析技术

3.1 多源绑定发生问题定义

首先,本文给出了多源绑定发生问题的形式化描述.

定义 1(程序执行的轨迹流). 在程序的一次执行中,从程序入口到程序出口的一系列被执行的语句序列 P:

$$P=ENTRY\rightarrow S_1\rightarrow S_2\rightarrow\ldots\rightarrow S_{n-1}\rightarrow S_n\rightarrow EXIT,$$

其中,S_i 表示程序执行过程中执行的指令语句.如果 P_1 是 P 的子序列,那么 $P_1\subseteq P$,其含义表示 P_1 出现在 P 的执行过程中.

定义 2(污点分析轨迹流). 从污点源传播到污点汇聚点的一系列被执行的语句序列 F:

$$F=SRC\rightarrow S_1\rightarrow S_2\rightarrow\ldots\rightarrow S_{n-1}\rightarrow S_n\rightarrow SINK.$$

如果 $F\subseteq P$,则说明一条污点分析轨迹流 F 出现在程序执行 P 中或程序的一次执行 P 可以触发污点分析结果 F.

定义 3(多源绑定发生). 令 $\overline{F}$ 表示一组污点分析轨迹流,那么在一次执行轨迹流 P 中,该组轨迹流能够绑定发生,当且仅当满足如下公式:

$$\overline{F}(P)=\{F_i \mid F_i\in\overline{F} \text{ and } F_i\subseteq P\}.$$

我们简称多源绑定发生问题为多源问题.令 $\overline{F}$ 的大小(即 $\overline{F}$ 中污点结果的个数)为 k,我们称大小为 k 的 $\overline{F}$ 的绑定发生问题为 k-多源绑定发生问题,简称 k-多源问题.传统的产生独立污点结果的污点分析则是 1-多源问题.

目前,使用静态分析解决多源绑定发生问题具有很大的挑战.

- 首先,由于程序中分支和循环的存在,程序可能的执行路径数量会根据分支的数量呈指数级上升.为了完成有效的分析,静态(污点)分析常用的方法是在分支交汇处合并数据流值,以表示出当前可能发生的数据流值.例如,图 3 例子中,分支结束点第 13 行得到的最终数据流值包含的源是{*getDeviceId*, *getMacAddress*},表示两个源都可能传播到网络.然而,同样的问题在多源绑定发生问题中并不适用,因为如果在程序分支处进行合并就无法有效地区分出不同路径下的污点传播.
- 其次,一次污点分析往往会产生多组污点分析结果,记其个数为 k.对 k-多源问题求解的复杂度会是 k 的指数级别(将在后文进行论证),即便我们仅对结果中的两对源进行分析(2-多源问题),k 对结果之间两两组合的个数是 C_k^2,此时,我们可能的查询次数最多会有 C_k^2 次数.如果多源问题的开销较大的话,多次

的查询会导致系统难以使用.

本文创新地提出了解决上述难点的方法,我们将在第 3.2 节介绍多源绑定发生的污点分析技术(保证精度),在第 3.3 节介绍一种高效的实现方法(保证效率).

3.2 多源绑定发生的污点分析

正如第 1 节所述,静态污点分析的数据流值表示的是当前的污点变量可能到达该语句 Stmt.为了能够保证多源绑定发生的分析,我们创新地提出了将数据流值扩展以污点变量组成的向量的形式,如公式(1)所示.

令 $v_1,v_2,\ldots,v_{n-1},v_n$ 为污点变量,其绑定发生的数据流值为

$$DataFlow\ Fact=\langle v_1,v_2,\ldots,v_{n-1},v_n\rangle \tag{1}$$

数据流值是一个污点变量组成的向量,而整个数据流值表示向量中所有的变量在当前的语句 Stmt 上是可以绑定到达的.以图 5 中的程序为例,如果数据流值$\langle b1,b2\rangle$分别传播到 *sink*(*b*1)和 *sink*(*b*2),则表示污点变量 *b*1 和 *b*2 的源是可以绑定发生的.传统的污点分析方法(如 FlowDroid 中的方法)正是该方法的一个特例,即数据流值为一元向量的情况.

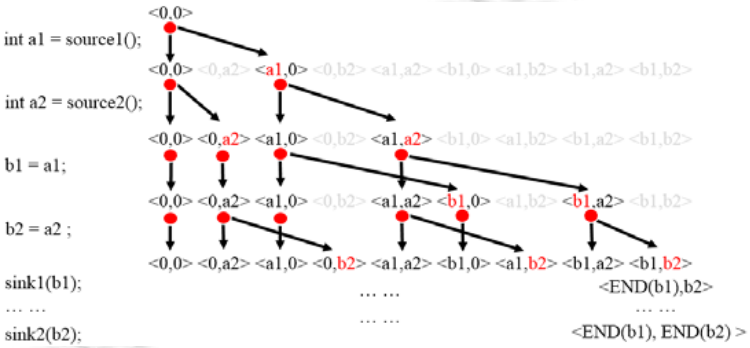


Fig.5 An example of multi-source binding data flow analysis (1)

图 5 多源绑定发生数据流传播示例(1)

上述数据流值形式保证其分析可以满足 IFDS 问题,从而对该问题的求解可以支持上下文敏感和流敏感的性质.具体来讲,我们提供了 IFDS 问题的扩展数据流方程相关的转移函数,即 Normal-Flow 函数、Call-Flow 函数、Return-Flow 函数、CallToReturn-Flow 函数,表 1 给出了算法详细的伪代码.解释如下:令 $\bar{v}$ 表示数据流值向量,用 v 表示 $\bar{v}$ 中对应维度的元素,算法重用了传统的污点分析的转移函数(简称为 solo 转移函数,算法中的 soloNormal/soloCall/soloReturn/soloCallToReturn 函数)用来表示对单一的 v 的污点变量独立的传播规则.由于 solo 转移函数已经在文献[18]中提供,这里省略.在 Normal-Flow 函数中,对 $\bar{v}$ 中的每一维度 v 进行遍历,如果 v 在当前语句 n 的计算,即 *Normal*(*v*,*n*)的结果为 soloRes 集合,那么对于 soloRes 中每一个元素 *t*,用 *t* 替换原 $\bar{v}$ 中 v 生成新值,即 $\bar{v}$.*replace*(*v*,*t*)来表示保证其他维度不变用新值 *t* 替换 *v* 生成新值.例如在图 5 右侧的传播:对于语句[*b*1=*a*1;],数据流值$\langle a1,a2\rangle$中的 *a*1 会根据 soloNormal 函数产生 *a*1 和 *b*1,所以最终的函数会生成$\langle a1,a2\rangle$和$\langle b1,a2\rangle$.在 Call-Flow 函数中,需要对 $\bar{v}$ 中所有在函数调用中使用的变量都进行污点传播;对 $\bar{v}$ 中每一维度元素,查找其是否在函数调用中使用,如果有使用,则用 soloCall 生成的新值进行替换.Return 函数与 Call 函数类似,只是将参数映射转化为返回值的映射.CallToReturn 与 Normal 类似,一种特殊的 CallToReturn 函数是对污点源和汇聚点的处理函数,例如,$\langle 0,0\rangle$表示 IFDS 框架的初始值(0 值),污点源 *source*1 和 *source*2 为两个不同的源.对于污点源的计算,算法将从 0 产生对应位置的污点变量,如语句[int *a*1=*source*1();]将从$\langle 0,0\rangle$产生$\langle a1,0\rangle$.算法的初始输入是待分析的多个源的集合,而其他不在这个集合的污点源并不会进行分析.图 5 右侧演示了左侧程序的传播过程.可见,算法能够有效地生成$\langle b1,b2\rangle$向量,也就是之后可以绑定地触发 *sink*(*b*1)和 *sink*(*b*2)的污点变量.

Table 1 Data flow transfer functions of multi-source binding analysis

表 1 多源绑定发生的数据流转移函数

Normal-flow function	CallToReturn-flow function

for each v in $\bar{v}$:
  Set soloRes = *soloNormal*(v,n)
  for each t in *soloRes* :
  $T = T \cup \bar{v}.replace(v,t)$
  end
end
	for each v in $\bar{v}$:
	  Set soloRes = *soloCallToReturn*(v,n)
	  for each t in *soloRes* :
	  $T = T \cup \bar{v}.replace(v,t)$
	  end
	end

Call-flow function	**Return-flow function**

Set T = *new Set*($\bar{v}$)
for i in [0 : $\bar{v}$.*len* −1]:
  for each $\bar{v}$ in T :
  $v = \bar{v}.get(i)$;
  Set soloRes = *soloCall*(v,n)
  for each t in *soloRes* :
  $T = T \cup \bar{v}.replace(v,t)$
  end
  end
end
	Set T = *new Set*($\bar{v}$)
	for i in [0 : $\bar{v}$.*len* −1]:
	  for each $\bar{v}$ *in* T :
	  $v = \bar{v}.get(i)$;
	  Set soloRes = *soloReturn*(v,n)
	  for each t in *soloRes* :
	  $T = T \cup \bar{v}.replace(v,t)$
	  end
	  end
	end

该方法的主要特点就是能够区分出分支(IF)互斥的路径中污点变量的传播,如图 6 所示程序为例,*source*1()源的污点变量 *c*1 的传播路径是 *source*1→*a*1→*b*1→*c*1.其中,如果变量无法传递到中间变量 *b*1,自然无法传递到 *c*1.如果考虑到分支的情况,分支 *Brb*1 和 *Brc*1 控制了污点源 *source*1()进行传递的变量 *b*1 和 *c*1,而分支 *Brb*2 和 *Brc*2 则控制对 *source*2()传播的 *b*2 和 *c*2.不难发现,*Brb*1 和 *Brb*2 是互斥的.这会导致 *a*1 污点传播到 *b*1 而 *a*2 就不会传播到 *b*2,反之亦然.所以对于可以触发 *sink*(*b*1)的 *b*1 和触发 *sink*(*b*2)的 *b*2 对应的源是不能绑定发生的.我们的方法可以有效地区分出此类情况.如图 6 右侧的具体数据流传递所示,传播算法可以正确地区分出具有互斥关系的分支.由最后的结果可见:结果向量集合中不会存在具有互斥(不会再一次执行中绑定发生)的污点变量,⟨*b*1,*b*2⟩,⟨*c*1,*c*2⟩,⟨*b*1,*c*2⟩,⟨*c*1,*b*2⟩是不会出现在结果中的.

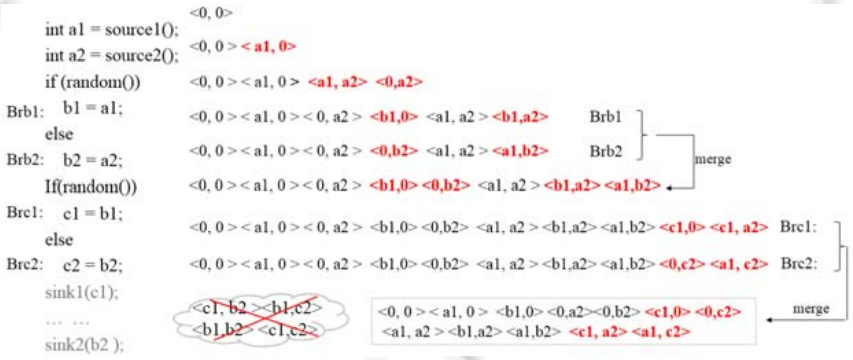


Fig.6 An example of multi-source binding data flow analysis (2)

图 6 多源绑定发生数据流传播示例(2)

此外,我们还扩展了一个特殊的污点变量:*END* 值(如图 5 所示),*END*(*S*)表示从 *S* 的源出发的污点变量已经流到 *sink*().这是出于考虑:有些情况虽然一个源已经被 sink 触发了,但是为了整体的分析,还需要继续告知其他源该源已经发生的信息.例如图 7 的情况,源 *src*1 和源 *src*2 是可以同时发生的,但是在 *src*1 的污点变量还未能到达 *sink*1()时,*src*2 的污点变量就已经传播到 *sink*2,且在 *func*1 内部完成了该传播.通过使用 *END* 数据流,可以有效地解决该问题,即在 *sink*2 之后,在 *src*2 源对应维度增加 *END* 变量继续传递.*END* 值不同于触发 sink 但又可以

继续传播的污点变量,因为 *END* 是可以跨越过程的.

```
Main:                          func1:
    int a=src1();                  int a=src2();
    if(random()){                  ...
      b=a;                         sink2(a);
      func1();
    } else{                        func2:
      func2();                     int a=src3();
    }                              ...
    sink1(b);                      sink3(a);
```

Fig.7 An example of multi-source binding data flow analysis (3)

图 7 多源绑定发生数据流传播示例(3)

3.3 多源问题的高效实现方法

由于 IFDS 求解算法的最坏时间复杂度是 $O(ED^3)$,算法的效率与数据流域 D 的大小(即数据流值的个数)有直接关系.我们令传统方法(1-多源问题)的污点变量个数是 D_{solo},多源问题中源的个数(向量维度)是 k,那么最坏情况 $D=D_{solo}^k$, 代入得知,整个问题的复杂度是 $O(ED_{solo}^{3\times k})$. 由此可见,多源绑定发生问题算法的复杂度是 k 指数级别的.

为了能够高效地实现多源问题的分析,我们将该问题近似为:将 k-多源问题转化为 C_k^2 次的 2-多源问题求解.具体来讲,令 k-多源问题中的所有源进行两两组合形成的集合为 S,如果 S 中所有的 2-多源问题都能满足(绑定发生),我们就近似认为 k-多源问题能够满足.例如,对于结果为{*getDeviceId*,*getMacAddr*,*getLineNum*}的 3-多源问题,需要保证{*getDeviceId*,*getMacAddr*}、{*getDeviceId*,*getLineNum*}和{*getLineNum*,*getMacAddr*}这 3 个 2-多源问题都满足.我们提出上述近似方法的原因是:首先,如果 k-多源问题满足,那么 S 中的 2-多源问题是一定都满足的(充分条件),所以该近似不会产生漏报.虽然反之不成立(不满足必要条件),即会产生误报,但是我们认为在隐私泄露检测问题中,k-多源问题绑定发生和其所有两两组合的 2-多源绑定都发生的危害是相近的.同时,我们在实验中发现,一般能够满足上述近似的 k-多源问题,实际上有很大概率是完全满足 k-多源绑定发生的,误报率很低.我们将在实验部分进一步阐述相关验证.

上述的近似方法导致数据流域个数 D 的值降低为 $C_k^2\times D_{solo}^{3\times 2}$, 算法的时间复杂度是 $O(k^2ED_{solo}^6)$. 一般情况下,污点源的个数 k 的值并不会太大,且为了保证完整的分析结果,控制流的边的个数往往不变,因此,$k^2\times E$ 的数量往往无法被降低,此时,减少 D_{solo} 的个数将是本技术提高效率的关键.本节将提供一种减少 D_{solo} 的个数的高效实现方法.

由于污点分析的本质是一个按需的分析问题,即污点传播只在污点源到汇聚点之间的路径传播,直观上,污点分析应该只针对该路径进行分析即可.传统的污点分析在未遇到污点汇聚点时并不知道最终的传播路径,因此需要保守地传播所有污点传播的变量.当扩展到多源绑定发生问题时,这种方法将产生大量的无用传播.如图 8 中的例子所示.来自 *source*1 的返回值 *a*1 会在第 3 行传递给 *f*1、在第 4 行传递给 *t*1,*t*1 会在第 11 行传递给 *t*2,*t*2 在第 12 行触发 sink.这条完整的路径为 $a1\rightarrow t1\rightarrow t2\rightarrow$sink,在图 8 右侧路径中的红色节点即为该传播路径.然而对于 *f*1 来讲,他可以传递给 *f*2、*b*、*f*3 变量,但这些变量并不会触发 sink,如图 8 右图中白色节点之间的路径.然而在多源绑定问题分析时,即与 *source*2 的返回值 *a*2 形成向量,会根据上节的算法生成⟨*f*2,*a*2⟩,⟨*b*,*a*2⟩,⟨*f*3,*a*2⟩的值.同理,这些值也不会到达 sink 进行触发.可见,这些值的传播实际上是没有意义的.当 *a*2 进一步在程序中传播时,则会带来更多的无用数据流值.基于上述观察,本节方法的核心思想就是消减这些无用的数据流值,只在可以触发 sink 的路径上进行多源分析,即在图中只传播红色节点变量(对于 *source*1)和蓝色节点变量(对于 *source*2).

为了维护污点源和汇聚点的路径信息,我们需要利用一种重要的数据结构[19]——数据流值的前驱节点(predecessor)和邻居节点(neighbor).所谓一个数据流值的前驱节点,是指直接生成该节点的数据流值.例如,图 8

中,数据流值 $t1$ 的前驱节点是 $a1$.数据流值的邻居节点是指,如果两个数据流值可在分支(循环)出口进行合并(merge),但它们分别来自不同分支,那么它们之间互为邻居关系.例如图 8 中第 6 行处的分支,第 7 行和第 9 行都会产生数据流值 b.为了区分不同分支的值,令第 7 行处为 b',第 9 行处为 b,那么在分支结束处会将 b 的邻居节点设置成 b'.为了维护正确的污点分析路径,邻居信息是不可以被省略的,b'的前驱是 $f1$,b 的前驱是 $f2$,它们分别代表不同的路径.有了上述的数据结构,就可以利用触发 sink 的变量进行回溯,求得完整的污点分析路径,进而在多源分析中重用.另外,为了在多源分析中使用这些路径信息,我们还需为数据流值扩展一个结构用于存储由⟨*useStmt*,*nextDatafact*⟩组成的集合.*useStmt* 表示当前数据流值将要被使用的位置;*nextDatafact* 表示在 *useStmt* 处产生的新数据流值.例如,$t1$ 在第 11 行生成 $t2$,那么 *useStmt* 为 *line*11,*nextDatafact* 是 $t2$.

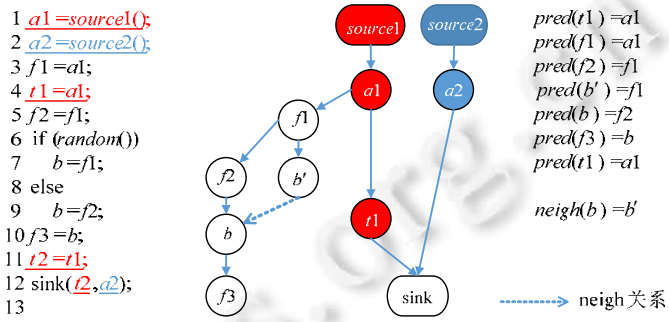


Fig.8 Efficient implementation method and data flow's predecessor and neighbor

图 8 高效实现方法以及数据流值前驱和邻居示例

基于上述结构,多源绑定发生的高效实现方法具体流程如下.

(1) 运行传统(1-多源问题)的污点分析,并通过 FlowTwist[19]中的算法将数据流值的前驱和邻居进行记录;并得到触发 sink 的污点数据流值集合 *sinkSet*.我们记该过程为 SoloFlow.

(2) 将 *sinkSet* 中的值加入工作集中;对于工作集中的数据流值 D,得到 D 的前驱 *pred*(通过 *getPredecessor* 方法),将 D 和产生 D 值的位置语句(通过 *getCurrentStmt* 得到)记录到 *pred* 中的⟨*useStmt*,*nextDatafact*⟩集合域中(使用方法 *SetUseStmt*()).将 D 的 *neigh* 和 *pred* 加入工作集,迭代地对工作集中的数据流值遍历直到结束.具体算法如算法 1 所示.我们记该过程为 PreProcess.

(3) 执行多源绑定的污点分析:由于每一个数据流值已经在 PreProcess 过程中记录了其后继的产生位置 *useStmt* 和后继的值 *nextDatafact*,我们修改原 Solo 转移函数为只有在 *useStmt* 的位置产生新的数据流值 *nextDatafact* 即可.记该过程为 MultiFlow.

算法 1. PreProcess 过程算法.

输入:*sinkSet*.

输出:*sourceSet*.

```
worklist←sinkSet;
while (!worklist.isEmpty())
  D=worklist.pop();
  if (isSource(D))
    sourceSet←D;
  pred=D.getPredecessor();
  pred.setUseStmt(D.getCurrentStmt(),D);
  worklist←pred;
  foreach neigh:D.getNeighbours()
```

worklist←*neigh*;
end
end

举例说明,对于 *a*1 变量,经过 PreProcess 过程之后,*a*1 中⟨*useStmt*,*nextDatafact*⟩域的值是⟨*line*4,*t*1⟩.因此,该变量只在第 4 行处生成 *t*1 而不会在第 3 行处生成 *f*1.同理,*t*1 也只会在第 11 行处产生 *t*2.如图 8 右侧红色部分是 *source*1 的传播路径,而白色部分则不会计算.可见,该方法可将大量的无用变量的传播进行消除,而对于过程间的数据流值,效率提高的效果将更加明显.此外,我们还利用了 SoloFlow 阶段产生的摘要边信息应用到上述方法进行优化,即当只有一个维度的污点变量被函数调用使用,并且当前该调用已经具有该值得摘要边时,我们直接使用其摘要信息产生其函数返回值.摘要边也是在 SoloFlow 中自然产生的,因此不会引入额外的开销.

上述方法的主要特点是:我们只需要执行 1 次污点传播以及回溯地完成路径信息的记录(soloFlow 和 PreProcess 过程),但是可以执行多次的多源绑定发生(MultiFlow 过程)的分析.所以,对于一次污点分析的结果进行 *n* 次的绑定发生分析,其时间为 *T*=*time*(SoloFlow)+*time*(PreProcess)+*n*×*time*(MultiFlow).对于一个 *k*-多源问题,其近似计算方法的时间是 *T*=*time*(SoloFlow)+*time*(PreProcess)+ C_k^2 ×*time*(2-MultiFlow).

4 实 验

4.1 实 现

为了验证本文技术的有效性,我们实现了一个原型系统 MultiFlow.MultiFlow 的实现是基于 Soot 和 FlowDroid 的:重用了 FlowDroid 面向 Android 特性构建的函数调用图,使用访问路径(access path)来保证域敏感的分析,在 Soot 的中间表示 Jimple 上完成多源污点分析.MultiFlow 重用了 FlowDroid 的后向传播求解的别名分析,将一个污点变量别名的数据流值的前驱设置成该变量的前驱.由于 Android 系统是事件驱动的,MultiFlow 设置一次多源绑定分析的结束为一次回调函数的结束.

在下文的所有实验以及第 5 节的应用中,我们统一使用表 2 所示的污点源和汇聚点.

Table 2 Sources and sinks in MultiFlow provided by Susi

表 2 MultiFlow 使用的 Susi 提供的源和汇聚点

源(source)			
名字	标签	名字	标签
getLatitude	位置信息	(BluetoothAdapter)getAddress	蓝牙地址
getLongitude	位置信息	(WifiInfo)getMacAddress	Wifi 信息
getDeviceId	设备信息	(WifiInfo)getSSID	Wifi 信息
getSubscriberId	设备信息	(Locale)getCountry	地区代码
getSimSerialNumber	设备信息	getCid/getLac	位置信息
getLine1Number	设备信息	(AccountManager)getAccounts	账户信息
(URLConnection)getOutputStream	网络信息	(Calendar)getTimeZone	时区信息
(URLConnection)getInputStream	网络信息	(Browser)getAllBookmarks	浏览器书签
(http.HttpResponse)get	网络信息	(Browser)getAllVisitedUrls	历史记录
(http.util.EntityUtils)toString	网络信息	(java.net.URL)openConnection	网络信息
android.media.AudioRecord	媒体信息	(ContentResolver)query	ContentProvider
getLastKnownLocation	位置信息	(database.Cursor)getString	数据库信息
getDefaultSharedPreferences	配置数据	(SQLiteDatabase)query	数据库信息
汇聚点(sink)			
(BasicNameValuePair)⟨init⟩	网络输出	(java.net.URL)set/init	网络输出
(OutputStream)write	字符流输出	java.net.URLConnection	网络输出
(FileOutputStream)write	文件流输出	(SmsManager)sendTextMessage	短信输出
(java.io.Writer)write	字符流输出	(DefaultHttpClient)execute	网络输出
(OutputStreamWriter)append	字符流输出	(HttpClient)execute	网络输出
(SharedPreferences) put*	配置数据	(ContentResolver)insert/update	ContentProvider

表 2 所示的污点源和汇聚点由 SuSi 工具提供,由于排版受限,我们将源和汇聚点缩写成其函数名和方法名

的形式,省略了参数和返回值类型,完整的名字请参考 https://blogs.uni-paderborn.de/sse/tools/susi/.

4.2 效 率

本节将验证多源绑定技术的效率.本文的技术为传统污点分析技术精度的提升提供了可能,但是该技术也会引入开销的增加.因此,该实验的目的就是验证多源绑定发生技术与传统技术相比有多少开销.实验对比对象是传统的污点分析,此处我们使用 FlowDroid 的污点分析阶段.实验机器的配置是:64 核 Intel(R) Xeon(R) CPU E7-4809(2.0GHz)和 128G RAM,为每一个 Java 虚拟机分配 32G 的内存.我们随机地从 Google Play 应用市场下载验证程序,并尽量选择其中可运行规模较大的程序.这里,我们选择 IFDS 边的个数作为计算规模的衡量标准,最终的验证程序包含 16 个 Android 应用程序.由于我们将 *k*-多源问题转化为 2-多源问题,我们将验证多次的 2-多源问题.此时,我们在 1-多源问题(SoloFlow 阶段)的结果中两两组合形成的集合中随机选择 10 对组合进行 2-多源问题计算(如果少于 10 次,则将所有源进行两两组合计算).我们对验证集合的程序运行 3 次,随后记录其执行时间的平均值,表 3 是相关的实验结果.

Table 3 Time overhead of multi-source binding taint analysis technique

表 3 多源绑定发生污点分析技术的时间开销

编号	程序名称	程序大小(M)	IFDS 边个数(百万)	结果数量	FlowDroid 执行时间(s)	MultiFlow 执行时间(s)
1	com.facebook.katana.apk	14.6	7.4	22	67.3	74.2
2	com.pandora.android.apk	5.0	19.1	27	213.5	259.3
3	com.nhn.android.nbooks.apk	19.3	1.2	27	15.5	16.7
4	com.alt12.babybumpfree.apk	8.5	2.8	39	33.1	39.5
5	com.onelouder.baconreader.apk	6.3	4.7	7	53.6	64.1
6	org.limb.boral.pack.over.apk	1.2	1.7	49	16.4	18.6
7	com.zennia.app.sms.apk	1.5	14.2	419	198.1	219.7
8	com.yoka.yueting.apk	3.2	3.5	23	52.3	75.4
9	cn.kaoshi100.view.apk	3.1	5.8	94	60.7	63.1
10	cn.com.y2m.apk	12.8	3.9	228	40.1	43.3
11	com.siyami.apps.cr.apk	1.1	17.1	74	165.4	199.5
12	cn.Ffcs.Cartoonplayer.apk	3.3	13.2	17	217.6	261.7
13	com.firebear.androil.apk	11.3	2.5	56	25.7	28.4
14	cn.wps.moffice_eng.apk	13.2	24.2	12	178.6	209.3
15	com.accesslane.livewallpaper.apk	5.3	1.8	57	21.2	35.1
16	com.yskj.djp.activity.apk	1.9	23.7	79	253.3	309.6

由实验结果得出,我们的方法与传统方法相比不会有较大开销.此外,由于通常多源分析会进行多次 MultiFlow 阶段的查询,我们验证了多源问题高效实现方法的各个阶段时间所占比例.如图 9 所示,由于 SoloFlow 阶段的时间即为 1-多源问题的时间,所以我们的主要开销是 *time*(PreProcess)+*n*×*time*(2-MultiFlow).

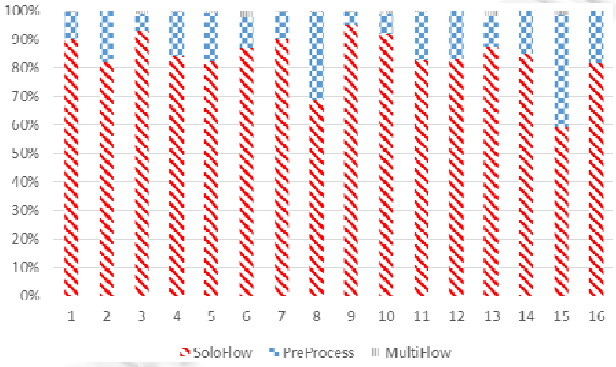


Fig.9 Percentage of each phase of the efficient implementation method

图 9 高效实现方法的各个阶段所占比例

从图 9 中可以发现,2-MultiFlow 所用时间所占比例仅为 1%(平均 0.3s).这正说明了多源高效实现方法的有

效性.因此,我们的开销主要消耗在 PreProcess 阶段.也就是说,进一步对 2-MultiFlow 进行多次查询对系统的查询开销影响并不大.综合统计,本文技术的平均时间开销(PreProcess 阶段开销)为 19.7%,这在静态分析中往往是可以接受的.

4.3 *k*-多源问题近似方法验证

本节将验证将 *k*-多源问题转化为 C_k^2 次的 2-多源问题的可行性.由于此方法并不会产生漏报,而可能产生误报,我们这里主要验证其误报率的大小,也就是说,验证 C_k^2 次 2-多源问题满足的条件下,*k*-多源问题的满足情况.这里,我们选择在恶意软件库中挑选程序进行验证.我们在 Genome 库[17]中随机挑选 50 个具有隐私泄露问题的程序,利用 MultiFlow 对其所有的污点分析结果两两配对进行 2-多源问题分析.我们使用反编译工具将其转化为可读的中间表示代码,对其结果进行人工验证.我们发现,这 50 个程序中,所有通过近似方法满足的 *k*-多源问题就是 *k*-多源绑定发生的.进一步分析,如果近似方法不满足,则说明具有多个 2-多源问题分别出现在了不同的分支下,如图 10 所示的分支,多源对⟨*src*1,*src*2⟩,⟨*src*1,*src*3⟩,⟨*src*2,*src*3⟩分别出现在不同的分支下,而我们选择的这 50 个例子是没有这种复杂情况的.这也充分地验证了 *k*-多源问题近似方法的可行性.

```
if (...) {                foo1:                 foo2:                 foo3:
   foo1();                  int p = src1();       int p = src1();       int p = src2();
} else if (...) {           ...                   ...                   ...
   foo2();                  sink1(p) ;            sink1(p) ;            sink2(p) ;
} else {                    ...                   ...                   ...
   foo3();                  int q = src2();       int q = src3();       int q = src3();
}                           ...                   ...                   ...
                            sink2(q) ;            sink3(q) ;            sink3(q) ;
```

Fig.10 An example of the *k*-multi approximation method violation

图 10 *k*-多源问题近似方法违反示例

5 Android 应用隐私泄露检测应用

本节我们将探索多源技术在 Android 应用隐私泄露检测中的应用.首先,本节验证了当前良性软件和恶意软件在多源绑定发生特性上的差异,从而提供区分恶意行为的依据;其次,本节验证多源技术相比简单方法的精确度提升;最后,本节提出了一种污点分析结果风险评级方法,检测人员即可从结果中恶意行为级别由高到低地进行检测,从而缩小了检测范围,提高了分析的效率.

5.1 良性软件和恶意软件多源绑定发生特性差异

本节将验证文章开始部分提出的假设:良性软件和恶意软件的多源绑定特性具有较大差异,从而可以利用其提高检测精度.实验使用统计方法进行验证,选择具有较大数量的良性软件库和恶意软件库作为验证集合.我们在应用市场中随机选择了 2 116 个程序作为良性软件库测试集合,其中,1 148 个来源于 Google play 应用市场[16],968 个来源于安智网应用市场[20].另外选择了 2 089 个程序作为恶意软件库集合,其中,1 089 个来自 Genome 库[17],1 000 个来自 VirusShare 库[21].最后,使用 MultiFlow 对这些软件进行多源绑定的分析,将所有 SoloFlow 阶段产生的结果之间的所有组合进行了分析,如果两个源可以绑定发生,我们就停止对这两个源(其他调用点)进行分析.

在良性软件库运行结果中,337 个(占 15.9%)应用程序(由于超时、反编译错误等问题)无法正常分析,443 个(占 20.9%)程序产生小于 2 条污点分析结果(绑定发生至少需要 2 个源).我们将这两类程序去除,最终可以进行绑定发生分析的集合包括 1 336 个应用程序,其中有 382 个(占能多源分析的 28.6%)应用程序没有绑定发生的污点分析结果(虽然有大于 1 条的传统污点分析结果,但都不是绑定发生的).对剩下的具有绑定发生结果的 954 个应用程序进行统计分析,实验使用 Apriori 算法[22]抽取频繁项集,其中,选择最小支持度(support)为 10%,即多

源组合中出现次数大于整个集合的 10%(95 个)的多源组合.最后统计结果按照支持度进行排序,见表 4.在恶意软件库运行结果中,除去 103 个(占 4.9%)应用程序无法正常分析和 711 个(占 34%)程序产生小于 2 条的污点分析结果(恶意软库中还包含很多非隐私泄露问题的恶意问题[17],这种情况并不会产生污点分析结果),最终可以进行绑定分析的集合有 1 275 个.另外发现有 237 个(占能多源分析的 18.7%)应用程序没有绑定发生的污点分析结果.最后对剩下的产生结果的 1 038 个应用程序进行统计分析,同样使用 Apriori 算法抽取频繁项集(最小支持度 10%),统计结果见表 5.同时,我们还对比良性软件库中使用频繁的组合在恶意软件库中所占的比例(反之亦然),结果为表 4 和表 5 的最后一列.

从表中的结果可以发现,恶意软件和良性软件在多源绑定发生特性上具有明显的差异.具体来讲:

- 首先,在使用频繁度上,良性软件和恶意软件的统计结果差别很大.良性软件统计结果包含 16 个组合,且其中有 8 个组合出现在恶意软件统计结果中;但恶意软件统计结果产生了 38 个组合,而只有 6 个组合也出现在良性软件的统计结果中.针对排名次序差异的比较结果更为明显,良性软件结果中出现频率最多的是{getLongitude,getLatitude}(47%),其次是{getDeviceId,getLongitude/getLongitude}(39%)和{getDeviceId,(java.net.URL)openConnection}(34%),这也说明了良性软件使用的都是用户常用的功能,即地理信息(经度/纬度)、地理信息和设备号、设备号和网络输入信息.我们发现,这些信息在恶意软件统计结果中也会出现,这充分说明了当前恶意软件经常需要伪装成一个正常的软件,所以同样会使用这些常用的功能组合.然而对比恶意软件结果,{getDeviceId,getSubscriberId}出现的次数是最多的(43%),而该对组合在良性结果中仅占 0.4%(4 个),这反映出良性软件往往不需要 getDeviceId 和 getSubscriberId 绑定发生,而恶意软件则出于盗取更多设备信息的目的将两者绑定发送.表 5 中的大部分组合都是此类的信息,它们是组合 1、组合 5、组合 6、组合 9、组合 10、组合 12~组合 38,一共 32 组,它们出现在良性软件中的概率最大仅为 2.5%.
- 其次,我们发现,恶意软件结果平均每个组合中源的个数更多,良性软件结果中只有组合 5 和组合 10 包含 3 个源(这里,getLongitude、getLatitude 被视为 1 个源).而在恶意软件结果中,有 18 个组合包含 3 组以上的源,有 5 个组合包含 4 个以上的源,更特殊地,组合 36 包含了 5 个源,这也充分说明了恶意软件是需要更多的信息来完成恶意行为的.
- 最后,针对某些源,两个库之间的配对差异也很明显.例如,在恶意软件库中,对于(ContentResolver)query 主要和 getDeviceId 绑定发生,(ContentResolver)query 出现时,getDeviceId 绑定发生概率为 95%.而在良性软件库中,(ContentResolver)query 经常和(database.Cursor)getString(数据库查询)源绑定出现,(database.Cursor)getString 出现概率为 99%.这也说明恶意软件经常是发送用户账户信息结合设备信息来完成用户账户信息的盗取,而良性软件则是正常的数据库功能.

Table 4 High frequency results in benign App set

表 4 良性应用软件测试集合中出现频率高的结果

编号	多源组合	支持度(%)	与恶意软件对比(%)
1	getLongitude,getLatitude	47	22
2	getDeviceId,getLongitude/getLongitude	39	20
3	getDeviceId,(java.net.URL)openConnection	36	29
4	(java.net.URL)openConnection,getLongitude/getLongitude	34	8.1
5	getDeviceId,(java.net.URL)openConnection,getLongitude/getLongitude	31	6.3
6	getLine1Number,getLongitude/getLongitude	31	30
7	getDeviceId,getMacAddress	29	2.1
8	getDeviceId,getLine1Number	20	26
9	getLongitude/getLongitude,getMacAddress	19	0.5
10	getDeviceId,getLongitude/getLongitude,getMacAddress	17	0.4
11	(ContentResolver)query,(database.Cursor)getString	13	13
12	getDeviceId,(database.Cursor) getString	13	28
13	getLastKnownLocation,getLongitude/getLongitude	13	1.5
14	getDeviceId,getLastKnownLocation	11	0.9
15	getMacAddress,getLine1Number	10	0.1
16	(java.net.URL)openConnection,(database.Cursor)getString	10	18

Table 5 High frequency results in malicious App set

表 5 恶意软件测试集合中出现频率高的结果

编号	多源组合	支持度(%)	与良性软件对比(%)
1	getDeviceId,getSubscriberId	40	0.4
2	getDeviceId,(java.net.URL)openConnection	29	36
3	getDeviceId,(database.Cursor)getString	28	13
4	getDeviceId,getLine1Number	26	20
5	getSubscriberId,(java.net.URL)openConnection	24	1.5
6	getDeviceId,getSubscriberId,(java.net.URL)openConnection	24	1.3
7	getLongitude,getLatitude	22	47
8	getDeviceId,getLongitude/getLatitude	20	39
9	getDeviceId,getSimSerialNumber	19	1.1
10	(HttpResponse)get,(java.net.URL)openConnection	19	1.7
11	(java.net.URL)openConnection,(database.Cursor)getString	18	10
12	getDeviceId,(java.net.URL)openConnection,(database.Cursor)getString	17	2.5
13	getSubscriberId,getSimSerialNumber	16	1.7
14	getDeviceId,(ContentResolver)query	16	1.7
15	getDeviceId,getSubscriberId,getSimSerialNumber	16	0.7
16	getSubscriberId,(ContentResolver)query	15	0.7
17	(java.net.URL)openConnection,(ContentResolver)query	15	1.9
18	getDeviceId,(java.net.URL)openConnection,(ContentResolver)query	15	0
19	getDeviceId,getSubscriberId,(ContentResolver)query	15	0.8
20	getSubscriberId,(database.Cursor)getString	15	2.1
21	getDeviceId,getSubscriberId,(database.Cursor)getString	14	2.1
22	getSubscriberId,(java.net.URL)openConnection,(ContentResolver)query	14	0
23	getDeviceId,getSubscriberId,(java.net.URL)openConnection,(ContentResolver)query	14	0
24	getSubscriberId,getLine1Number	14	0.7
25	getDeviceId,getSubscriberId,getLine1Number	14	0.7
26	getDeviceId,(ContentResolver)query,(database.Cursor)getString	13	2.3
27	getSimSerialNumber,getLine1Number	13	0.1
28	getDeviceId,getSimSerialNumber,getLine1Number	12	0.1
29	getSubscriberId,(java.net.URL)openConnection,(database.Cursor)getString	12	0.4
30	getDeviceId,getSubscriberId,(java.net.URL)openConnection,(database.Cursor)getString	12	0.4
31	getSubscriberId,(ContentResolver)query,(database.Cursor)getString	12	0.7
32	(java.net.URL)openConnection,(ContentResolver)query,(database.Cursor)getString	12	0
33	getDeviceId,getSubscriberId,(ContentResolver)query,(database.Cursor)getString	12	0
34	getSubscriberId,(java.net.URL)openConnection, (ContentResolver)query,(database.Cursor)getString	11	0
35	getDeviceId,(java.net.URL)openConnection, (ContentResolver)query,(database.Cursor)getString	11	0
36	getDeviceId,getSubscriberId,(java.net.URL)openConnection, (ContentResolver)query,(database.Cursor)getString	11	0
37	getSubscriberId,getSimSerialNumber,getLine1Number	11	0.1
38	(URLConnection)getOutputStream,(java.net.URL)openConnection	10	0.1

5.2 Android应用隐私泄露检测的精确度提升验证

本节所验证精确度的衡量标准是对良性软件的检查结果中 2-多源对的个数,2-多源对个数越少,该工具的精确度就越高.之所以使用良性软件的结果,是因为工具报告的良性软件中的污点路径并不是以隐私泄露窃取为目的的,在隐私泄露检测上,该工具产生了误报.当分析多源问题结果时,用户必须手工验证每个多源对结果是否是真实的泄露.多源对的数目越少,用户所检测的范围就越小,说明检测工具的误报就越少.而且当前没有准确标注恶意软件中污点分析路径的恶意行为的基准集合(benchmark),而良性软件中的路径是非恶意行为则是一个确定的事实.我们会在下一节介绍风险评级方法用以提高人工检测恶意软件中恶意行为路径效率的方法.

基于上述标准,我们统计了第 5.1 节中能够产生大于 1 条污点分析结果的良性软件,一共 1 336 个.分别使用 MultiFlow 和 FlowDroid 分析其产生的结果数目,对比图如图 11 所示.对于不考虑多源(图 11 中 SOLO 列)的情况,即将 FlowDroid 中所有的源进行统计(这里,我们假设同一个源出现多个调用点等同于用出现一次源),此时 FlowDroid 得到 3 935 个源.此时,MultiFlow 会产生 3 472 个 2-多源对,其中,一共有 2 827 个非重复的源.对于多

源的情况,由于当前 FlowDroid 没有多源分析的功能,我们假设其产生的结果中所有源的两两组合等同于 2-多源分析的结果,此时,FlowDroid 产生了 5 890 个 2-多源的对,对比 MultiFlow 产生的 3 472 个对,可见 MultiFlow 与 FlowDroid 相比在多源计算减少了 2 418 个 2-多源对,提升了 41.1%的精度.

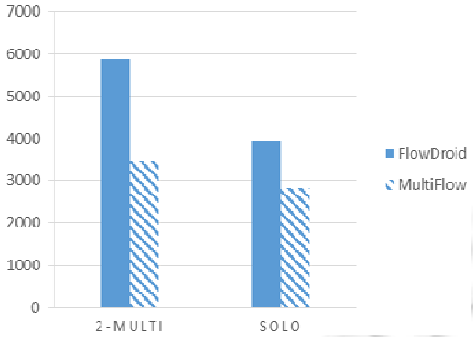


Fig.11 Result number comparision of MultiFlow and FlowDroid on multi-source analysis

图 11 MultiFlow 和 FlowDroid 多源结果数量对比

5.3 污点分析结果风险评级

对污点分析的结果进行风险评级具有较大的意义.当前的工具[5−8]不能保证 100%检测精度的恶意软件分类,这难免使得检测人员需要手工地检查每一条污点分析结果.即便上述工具可以精确报告该软件具有恶意行为,检测人员也可能想知道具体是哪条路径泄露的信息,或产生什么类型的泄露行为等.如果对污点分析结果进行评级,程序员就可以从危害级别较高的组合到级别较低的组合分别进行检测,一旦确定高危险度的路径是恶意路径则不必继续分析,这使得待分析的范围缩小而提高了检测的效率.由于第 5.1 节的实验证实了多源特性在恶意和良性软件间的差异,本节尝试利用该特性对污点分析结果进行风险评级.

本文的风险评级规则建立在 3 个启发式规则上.

启发式规则 1. 如果一个污点源不与其他任何的源绑定发生,我们则认为该源的污点分析路径几乎没有恶意行为.我们相信,恶意软件在盗取了用户的敏感数据之后是需要进行恶意行为的,然而单独地盗取一种敏感数据往往是没有意义的,例如只盗取用户的密码信息而不知道用户名等.而对于良性软件来讲,它们往往针对一类固定的信息进行正常使用,例如手机设备号验证或者单一的物理地址使用等.

启发式规则 2. 在大量的良性软件分析中,如果某些源的组合经常绑定发生,那么很可能这些组合是良性的.例如,地理信息中,经度信息往往和纬度信息绑定发生,如果只有经度和纬度的绑定发生,则认为其只是获取地理信息这一单一的行为.

启发式规则 3. 在大量的数据分析中,如果某些多源组合经常出现在恶意软件中,却很少出现在良性软件中,那么这些多源组合的绑定发生是具有恶意性的.在表 5 中,一些组合正好满足这样的特性.例如组合 18 和组合 22,它们出现在恶意结果中的频率分别为 15%和 14%,然而在良性软件统计结果中出现次数为 0.这也说明了在上一节的库中,只要包含组合 18 和组合 22 的都是恶意软件,而产生泄露的信息正是组合 18 和组合 22.

基于此,我们将污点分析的组合分成 4 类级别,由高到低依次是级别 IV、级别 III、级别 II、级别 I.其中,级别 IV 是最严重的危险级别,其组合是根据规则 3 产生的,在本应用中,我们使用表 5 中的组合 1、组合 5、组合 6、组合 9、组合 10、组合 12~组合 38 作为级别 IV 的组合,这些组合出现在恶意结果中的频率高于 10%,但是出现在良性结果中的频率低于 2.5%.级别 II 的组合是由规则 2 产生的,在本应用中我们认为,如果一个应用程序产生的多源组合集合中只出现在表 4,那么这些组合的危险度一般不高.级别 I 的组合是由规则 1 产生的,如果一个污点源不与其他任何的源绑定发生,则是级别 I.最后,如果组合不是上述 3 种级别,则认为它是级别 III.

进一步地,我们将上述的评级规则对实际手机软件进行应用,验证该规则是否可以有效地区分出不同类别的软件及其污点分析结果.我们直接将软件中包含的评级规则最高的级别作为其标签,利用其标签将所有验证

集合的软件进行分类.我们的验证集合是第 5.1 节中所有能够产生 1 条以上污点结果的软件,其中包含 1 336 个良性软件和 1 275 个恶意软件.如果使用表 5 中的组合 1、组合 5、组合 6、组合 9、组合 10、组合 12~组合 38 对集合中软件进行检测,可以直接检测出 685 个具有级别 IV 组合的软件.使用规则 2 可以检测出 953 个包含级别 II 及其以下级别组合的软件.使用规则 1 可以检测出 621 个只包含级别 I 污点分析结果的软件,剩下的 352 个软件是包含级别 III 及其以下级别组合的软件.图 12 为该结果的分布图,可见,包含级别 IV 组合的软件占有 26%,这也说明了我们的方法可以直接给出占较大规模的恶意结果(占实际恶意软件的 66%),检测人员可以直接识别出 26%个恶意软件中的具体泄露路径而不必分析其他路径.而级别 II 和级别 I 的结果分别占 37%和 24%,由此可见,该方法可以直接筛选出超过大半的良性的软件,一般情况下,此类软件的检测优先级较低.而对于包含级别 III 及其以下危险的软件只包含 13%,且检测人员可以优先检查具有级别 II 的污点路径.上述的应用数据也直接说明了使用该评级标准可以区分出不同级别的软件,间接地帮助检测人员提高检测效率和精度.

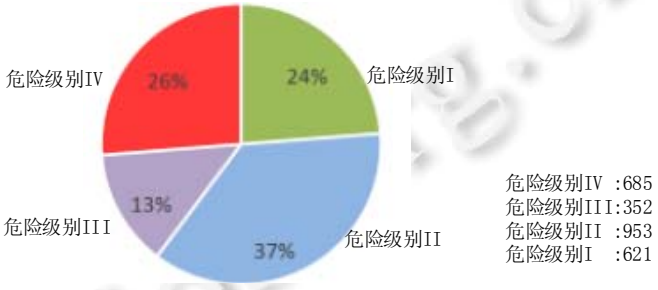


Fig.12 Distribution of App's taint result risk ranking

图 12 污点结果评级分布

6 相关工作对比

污点分析技术是一类重要的程序分析技术,如今,大量的研究工作应用其解决计算机系统信息的保密性和完整性问题.Mino[23]在硬件级别上扩展了寄存器的标志位来实现污点分析,以进行一些基础漏洞挖掘,如缓冲区溢出.Privacy Scope[24]、Dytan[25]利用插桩工具 Pin[26]实现了针对 x86 程序的动态污点分析.TAJ[27]工具利用混合切片的污点分析提供 Java Web 安全漏洞的检测.在面向 Android 安全的应用中,TaintDroid[28]是当前最流行的动态检测隐私泄露的工具,它使用多层次的插桩来完成污点传播.然而,TaintDroid 必须在运行时对 APK 文件进行检测,这依赖于 APK 输入测试集合来触发敏感数据流,且 TaintDroid 会对 Android 系统本身带来一定的开销,每次 Android 版本更新之后,TaintDroid 都需要重新进行定制.CHEX[29]尝试使用静态的污点分析方法检测智能手机组件劫持漏洞.FlowDroid[12]则提供了更精确的静态污点传播.DroidSafe[30]结合 Android Open Source Porject(AOSP)实现了与原 Android 接口语义等价的分析模型,并使用精确分析存根(accurate analysis stub)将 AOSP 代码之外的函数加入到模型,在此基础上进行污点分析.IccTA[31]利用静态程序插桩和 IC3 工具[32]提供了敏感信息通过组件间进行泄露的检测方法.正如前文所述,上述工具虽然提供了基础污点分析方法,但是没有关注分析结果之间的相关性,也没有解决结果之间多源绑定发生的技术.

此外,利用污点分析对 Android 应用市场中恶意软件分类的相关工作也与本文类似.MudFlow[5]利用数据挖掘的方法直接使用 FlowDroid 的输出结果进行恶意软件分类,该工具针对 2 866 个良性软件的污点结果进行统计训练,其核心思想是,通过对良性软件产生的污点结果与当前待检测的 Apk 文件的差异来判断其恶意性.Apposcopy[6]尝试利用污点分析结合组件的语义信息提供应用签名,之后,尝试利用签名匹配来检测恶意行为.Dark Hazard[7]使用特殊的分支条件下的污点分析触发路径作为机器学习的特征,从而检测恶意的隐藏敏感操作.虽然上述工具都是以提高检测精度为出发点,但是我们发现,这些方法都是直接使用污点分析的结果,并不会提高污点分析本身的功能.我们如果假设污点分析是一个黑盒,那么这些方法使用黑盒产生结果,然后在外部进行分析;而我们的方法是在黑盒内部提供更细粒度的分析.所以我们相信,多源污点分析是可以直接应用到上

述方法中完成更进一步精度提升的.我们将在后续的工作中,将多源技术结合机器学习技术及更多语义信息来探索高精度的恶意软件分类方法.

7 总结与未来的工作

本文以当前污点分析检测 Android 应用隐私泄露的误报率高为出发点,针对恶意软件和良性软件之间的多源绑定发生特性差异来提高检测精度.我们提出了一种多源绑定发生的静态污点分析技术:在精度上,该技术具有上下文敏感、流敏感、域敏感等特性,并且可以有效地区分出分支互斥路径的情况;在效率上,提出一种按需的高效实现方法,降低了多次的多源问题计算的开销.我们的实验数据表明:即使在一次污点分析结果上进行多次多源问题分析,我们的执行时间也在可以接受的范围之内(初始阶段开销为 19.7%,进一步的多源分析时间平均为 0.3s).随后验证了多源绑定发生技术在隐私泄露检测问题中的应用,发现当前良性软件和恶性软件的多源绑定发生特性确实具有较大的差异.随后,我们验证了多源技术与简单方法相比的精度提升(减少多源对 41.1%).最后提出了一种智能手机隐私泄露结果风险评级标准,应用此评级结果,可以进一步改善用户的检测的效率.

在未来的工作中,我们将把该技术与其他应用技术相结合,以提供更高的恶意软件检测精度.

- 首先,该技术可以结合更多语义信息,基于语义信息的恶意软件检测方法可以避免代码混淆技术带来的威胁,而多源绑定发生技术为更多语义信息支持提供可能性,一些有效的语义信息包括 Android 生命周期、特殊的回调函数、关键的 API 使用、GUI 信息特征、特殊的分支触发条件等.例如,我们可以提取污点分析路径中的关键 API 作为语义信息基础,探索多个绑定发生源之间的 API 调用序列之间的关系.又如,我们可以探索在一个特殊的 Button 触发下,有哪些多源被绑定发生.如果这些源绑定发生行为与 Button 本身语义违反,则可以报告相关问题.
- 其次,我们将探索更有效的统计分析方法.目前,利用该方法在检测 Android 安全问题上有很大的应用空间.例如,MudFlow 尝试将待分析程序与良性软件间的差异作为特征,进一步利用 SVM 分类器进行检测.DroidADDMiner[8]利用了 FlowDroid 提取 API 数据之间的依赖关系作为特征向量进行恶意软件检测.多源方法在上述方法中的应用需要更有效的统计方法支持.另外一个值得研究方向是探索应用市场中不同的目录类别下的应用程序中多源问题的使用情况,因为相同类别下的 APK 往往会有类似的使用结果,探索相关的多源特性违反情况可以提供保证软件质量和软件安全的特征.

此外,多源绑定的污点分析技术还可以用来对别名分析进行优化.非别名的污点传播是跟随程序的执行而传播的,然而对于别名的传播往往不能通过直接的计算可以得到.例如,FlowDroid 尝试在遇到堆变量赋值计算别名时启动一个后向的别名分析求解器.此时,如果后向的别名传播和正向的变量在分支互斥的路径下,则会产生误报.例如,如图 13 所示,*a* 和 *b* 别名的前提是 IF 语句执行了第 5 行的分支,而实际的污点传播则是通过第 7 行进行的.此时,我们可以利用多源绑定的技术对该问题进行精度提升(设置 *a* 为另一个污点源,判断 *a* 是否与 source 绑定发生).我们坚信多源技术是一类重要基础性的技术,未来将被更多程序分析领域应用.

```
1  void foo(){
2   A a = new A();  A b = ...
3    String p= source();  String q= ...
4   if (random()){
5      b = a;
6   } else{
7      q = p;
8   }
9    b. f = q;
10   sink(a.f);
11}
```

Fig.13 An example of alias analysis optimization

图 13 别名分析优化示例

References:

[1] McAfee. Mobile threat report. 2016. http://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf

[2] Livshits VB, Lam MS. Finding security vulnerabilities in Java applications with static analysis. In: Proc. of the Conf. on Usenix Security Symp. USENIX Association, 2005. 262−266. https://www.usenix.org/legacy/event/sec05/tech/full_papers/livshits/livshits_html/

[3] Sabelfeld A, Myers AC. Language-based information-flow security. IEEE Journal on Selected Areas in Communications, 2003, 21(1):5−19. [doi: 10.1109/JSAC.2002.806121]

[4] Li L, Bissyandé TF, Papadakis M, Rasthofer S, Bartel A, Octeau D. Static analysis of Android apps: A systematic literature review. In: Proc. of the Information & Software Technology. 2017. 67−95. http://orbilu.uni.lu/handle/10993/26879

[5] Avdiienko V, Kuznetsov K, Gorla A, Zeller A, Arzt S, Rasthofer S, Bodden E. Mining apps for abnormal usage of sensitive data. In: Proc. of the 37th Int'l Conf. on Software Engineering (ICSE), Vol.1. IEEE Press, 2015. 426−436. [doi: 10.1109/ICSE.2015.61]

[6] Feng Y, Anand S, Dillig I, Aiken A. 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. ACM Press, 2014. 576−587. [doi: 10.1145/2635868.2635869]

[7] Pan X, Wang X, Duan Y, Wang X, Yin H. Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in Android apps. In: Proc. of the NDSS. 2017. http://www.cs.ucr.edu/~heng/pubs/ndss2017.pdf

[8] Li Y, Shen T, Sun X, Pan X, Mao B. Detection, classification and characterization of Android malware using API data dependency. In: Proc. of the Int'l Conf. on Security and Privacy in Communication Systems. Cham: Springer-Verlag, 2015. 23−40. [doi: 10.1007/978-3-319-28865-92]

[9] Aho AV, Sethi R, Ullman JD. Compilers, Principles, Techniques. Boston: Addison Wesley, 1986.

[10] 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. ACM Press, 1995. 49−61. [doi: 10.1145/199448.199462]

[11] Reps T. Program analysis via graph reachability. Information and Software Technology, 1998,40(11):701−726. [doi: 10.1016/S0950-5849(98)00093-7]

[12] 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]

[13] Lam P, Bodden E, Lhoták O, Hendren L. The Soot framework for Java program analysis: A retrospective. In: Proc. of the Cetus Users and Compiler Infastructure Workshop (CETUS 2011), Vol.15. 2011. [doi: 10.1.1.221.5311]

[14] Rasthofer S, Arzt S, Bodden E. A machine-learning approach for classifying and categorizing Android sources and sinks. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2014. [doi: 10.14722/ndss.2014.23039]

[15] Arzt S, Bodden E. StubDroid: Automatic inference of precise data-flow summaries for the Android framework. In: Proc. of the 38th Int'l Conf. on Software Engineering. ACM Press, 2016. 725−735. [doi: 10.1145/2884781.2884816]

[16] Google play. https://play.google.com/store

[17] Zhou Y, Jiang X. Dissecting Android malware: Characterization and evolution. In: Proc. of the 2012 IEEE Symp. on Security and Privacy (SP). IEEE, 2012. 95−109. [doi: 10.1109/SP.2012.16]

[18] 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

[19] Lerch J, Hermann B, Bodden E, Mezini M. FlowTwist: Efficient context-sensitive inside-out taint analysis for large codebases. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. ACM Press, 2014. 98−108. [doi: 10.1145/2635868.2635878]

[20] http://www.anzhi.com/applist.html

[21] http://virusshare.com

[22] Agrawal R, Srikant R. Fast algorithms for mining association rules. In: Proc. of the 20th Int'l Conf. on Very Large Data Bases (VLDB'94), Vol.1215. 1994. 487−499. [doi: 10.1.1.100.247]

[23] Crandall JR, Chong FT. Minos: Control data attack prevention orthogonal to memory model. In: Proc. of the 37th Int'l Symp. on Microarchitecture (MICRO-37). IEEE, 2004. 221−232. [doi: 10.1109/MICRO.2004.26]

[24] Zhu Y, Jung J, Song D, Kohno T, Wetherall D. Privacy scope: A precise information flow tracking system for finding application leaks. Technical Report, EECS-2009-145, Berkeley: University of California, 2009.

[25] Clause J, Li W, Orso A. DYTAN: A generic dynamic taint analysis framework. In: Proc. of the 2007 Int'l Symp. on Software Testing and Analysis. ACM Press, 2007. 196−206. [doi: 10.1145/1273463.1273490]

[26] Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Notices, 2005,40(6):190−200. [doi: 10.1145/1064978.1065034]

[27] Tripp O, Pistoia M, Fink SJ, Sridharan M, Weisman O. TAJ: Effective taint analysis of Web applications. ACM SIGPLAN Notices, 2009,44(6):87−97. [doi: 10.1145/1542476.1542486]

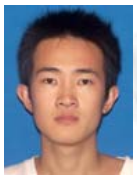
[28] Enck W, Gilbert P, Han S, Tendulkar V, Chun BG, Cox LP, Jung J, McDaniel P, Sheth AN. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. on Computer Systems, 2014,32(2):393−407. [doi: 10.1145/2619091]

[29] Lu L, Li Z, Wu Z, Lee W, Jiang G. Chex: Statically vetting Android apps for component hijacking vulnerabilities. In: Proc. of the 2012 ACM Conf. on Computer and Communications Security. ACM Press, 2012. 229−240. [doi: 10.1145/2382196.2382223]

[30] Gordon MI, Kim D, Perkins JH, Gilham L, Nguyen N, Rinard MC. Information flow analysis of Android applications in DroidSafe. In: Proc. of the NDSS 2015. 2015. [doi: 10.14722/ndss.2015.23089]

[31] 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 37th Int'l Conf. on Software Engineering, Vol.1. IEEE Press, 2015. 280−291. [doi: 10.1109/ICSE.2015.48]

[32] Octeau D, Luchaup D, Dering M, Jha S, McDaniel P. Composite constant propagation: Application to Android inter-component communication analysis. In: Proc. of the 37th Int'l Conf. on Software Engineering, Vol.1. IEEE Press, 2015. 77−88. [doi: 10.1109/ICSE.2015.30]



王蕾(1989－),男,吉林白山人,博士,主要研究领域为程序分析,软件安全.

李炼(1977－),男,博士,研究员,博士生导师,CCF 专业会员,主要研究领域为程序分析,编译,自动测试,软件安全.

周卿(1987－),男,博士,主要研究领域为静态多线程程序分析.

冯晓兵(1969－),男,博士,研究员,博士生导师,CCF 杰出会员,主要研究领域为编程模型,编译优化.

何冬杰(1992－),男,硕士,主要研究领域为 Android 安全,静态分析,经验研究.