Journal of Computer Applications
计算机应用,2019,39(10):3018-3027
ISSN 1001-9081
CODEN JYIIDU
2019-10-10
http://www.joca.cn

文章编号:1001-9081(2019)10-3018-10
DOI:10.11772/j.issn.1001-9081.2019040644

面向 Android 应用的静态污点分析结果的正确性验证

秦　彪[1],郭　帆[1*],涂风涛[2]
(1.江西师范大学 计算机信息工程学院,南昌 330022;　2.豫章师范学院 计算机系,南昌 330103)
(*通信作者电子邮箱 121171528@qq.com)

摘　要:应用静态污点分析检测 Android 应用的隐私泄露漏洞会产生许多虚警,为此提出一种上下文敏感、路径敏感和域敏感的半自动程序分析方法,仅需遍历少量执行路径即可判定漏洞是否虚警。首先,运行插桩后的应用来获得一条覆盖 Source 和 Sink 的种子 Trace。然后,应用基于 Trace 的污点分析方法来验证 Trace 中是否存在污点传播路径,是则表明漏洞真实存在;否则进一步收集 Trace 的条件集合和污点信息,结合活变量分析和基于条件反转的程序变换方法设计约束选择策略,以删除大部分与污点传播无关的可执行路径。最后,遍历剩余执行路径并分析相应 Trace 来验证漏洞是否虚警。基于 FlowDroid 实现原型系统,对 DroidBench 的 75 个应用和 10 个真实应用进行验证,每个应用平均仅需遍历 15.09% 的路径,虚警率平均降低 58.17%。实验结果表明该方法可以较高效地减少静态分析结果的虚警。

关键词:程序验证;污点分析;活变量分析;程序变换;路径敏感
中图分类号:TP311.53　　**文献标志码**:A

Correctness verification of static taint analysis results for Android application

QIN Biao[1], GUO Fan[1*], TU Fengtao[2]
(1. *College of Computer Information Engineering, Jiangxi Normal University, Nanchang Jiangxi* 330022, *China*;
2. *Department of Computer Science, Yuzhang Normal University, Nanchang Jiangxi* 330103, *China*)

Abstract: Many false positives are generated when an Android application is detected by static taint analysis to discover potential privacy-leak bugs. For that, a context-sensitive, path-sensitive and field-sensitive semi-auto analysis method was proposed to verify if a potential bug is a true positive by only traversing a few executable paths. Firstly, a seed Trace covering both Source and Sink was obtained manually by running the instrumented application. Then, a Trace-based taint analysis method was used to verify if there was a taint propagating path in the Trace. If there was a taint propagating path, it meaned a real privacy leak bug existed. If not, the condition set and taint information of the Trace were further collected, and by combining the live-variable analysis and the program transformation approach based on conditional inversion, a constraint selection policy was designed to prune most executable paths irrelevant to taint propagation. Finally, remaining executable paths were traversed and corresponding Traces were analyzed to verify if the bug is a false positive. Seventy-five applications of DroidBench and ten real applications were tested by a prototype system implemented on FlowDroid. Results show that only 15.09% paths traversed averagely in each application, the false positive rate decreases 58.17% averagely. Experimental results demonstrate the analysis can effectively reduce the false positives generated by static taint analysis.

Key words: program verification; taint analysis; live-variable analysis; program transformation; path sensitive

0　引言

随着智能手机的普及,Android 应用的安全性备受关注。隐私泄露是 Android 手机最严重的安全问题之一,它是指应用程序中存在一条从读取隐私数据的 Source 方法调用语句到送出隐私输出的 Sink 方法调用的执行路径,并且未经用户许可。污点分析是检测隐私泄露的主流检测方法之一,它从 Source 开始跟踪外部引入的数据(污点),检查它们是否未经验证就直接传播到 Sink 位置,如果是则可能存在漏洞。

污点分析分为静态分析和动态分析。静态分析不运行代码,直接对代码或者转换后的中间代码扫描,提取其中的词法、语法和语义,结合控制流分析和数据流分析,判定污点是否能从 Source 传递到 Sink[1]。静态分析可靠性高,但是需要耗费大量资源并且时间性能较差,为了实现精度和效率的平衡,往往会对所有分支的数据流信息进行保守的合并,从而产生大量虚警。动态分析指插桩并监控程序运行时行为,动态获取程序的控制流和数据流,实时跟踪污点传播,在 Sink 位置检测是否有污点信息输出[1]。动态分析的精确度高,但是动态分析难以覆盖程序的所有可执行路径,会遗漏许多潜在漏洞,可靠性不高。

为了降低静态分析的虚警率,研究人员提出了不少方案,主要分为基于约束求解[2-4]和基于机器学习[5]两类。约束求

收稿日期:2019-04-17;**修回日期**:2019-06-06;**录用日期**:2019-06-20。
基金项目:国家自然科学基金资助项目(61562040,61762049);江西省教育厅科技项目(GJJ161305,GJJ151330)。
作者简介:秦彪(1993—),男,江西南昌人,硕士研究生,主要研究方向:信息安全、程序验证;　郭帆(1977—),男,江西南昌人,副教授,博士,主要研究方向:网络安全、程序安全;　涂风涛(1976—),男,江西南昌人,讲师,硕士,主要研究方向:网络安全。

解常常与控制流分析和动态符号执行(Dynamic Symbolic Execution, DSE)技术相结合,在收集执行路径约束集合后,使用可满足性模理论(Satisfiability Modulo Theories, SMT)求解器判定路径是否可行,进而验证是否虚警。然而在实际应用中,路径条件复杂多变,约束求解存在约束表达困难和无法获得正确解的问题,导致虚警验证失败。机器学习通过统计方法或神经网络分析真实报警和虚警之间的特征差异,但是存在虚警验证错误的问题。

本文提出一种路径敏感、上下文敏感和域敏感的半自动分析方法,可以高效可靠地验证静态污点分析结果的正确性,在插桩和运行应用获得覆盖 Source 和 Sink 的运行 Trace 后,结合程序插桩、基于 Trace 的污点分析、活变量分析和程序变换方法,对程序的执行路径集合进行剪枝和遍历,进而验证分析结果是否虚警,针对 DroidBench 和真实应用的实验结果表明了该方法的有效性。

1　相关工作

静态分析技术是检查程序漏洞的有效手段,它通过静态扫描程序来找到匹配规则模式的代码从而发现代码中的问题。静态分析往往采用基于近似的分析方法,其分析结果不够精确,所以大多数静态分析工具生成的控制流图存在许多不确定性,如弱类型检查、未定义的行为以及别名指针等。根据 Rice 定理[6],静态分析针对程序的任何非平凡属性(例如程序是否存在数组越界),无法做到既完备又可靠,导致静态分析结果存在误报和漏报[2]。

国内外学者为消除静态分析结果中的误报,提出了不同的解决方法并进行大量的研究工作,设计和实现了各种静态漏洞检测工具。王蕾等[5]认为恶意应用的多个 Source 之间的相关性与正常应用存在差异,提出一种分析结果中的多个 Source 是否绑定触发 Sink 的污点分析技术,利用这种差异可以降低虚警率。赵云山等[2]以静态分析的结果作为输入,逆向搜索可能发生缺陷的约束条件,使用约束求解判断缺陷的可满足性,进而验证结果是否虚警。李筱等[3]对目标程序进行控制流分析,判断警报的可达性并得到制导信息,再利用混合执行测试的方法,跟踪程序运行时内存状态并判断内存是否泄漏,验证漏洞是否虚警。AppIntent[7]从可疑敏感数据泄露路径中抽取事件处理方法集合,形成事件处理方法约束图,并根据约束条件产生用户输入,验证该路径是否是用户许可的路径,从而排除虚警。DyTa[4]是一种自动漏洞检测工具,它的检测过程分为静态和动态两个阶段,静态阶段中利用静态检测技术发现程序的潜在漏洞;动态阶段中通过动态符号执行(DSE)技术生成测试用例,以新的测试用例执行被测程序来验证静态阶段中发现的漏洞是否真实存在。TASMAN[8]基于动态符号执行技术,结合污点分析收集程序控制流图的路径约束,通过 SMT 求解器[9]计算路径约束的可满足性以判断路径是否可行,过滤掉不可行路径中的警报来消除误报。

Fuzzing 测试[10]和动态符号执行是两种对程序安全进行动态测试的主流技术,经典 Fuzzing 测试使用随机产生的程序输入,会导致路径覆盖率较低,无法发现复杂执行路径中潜在的漏洞。Cai 等[11]结合符号执行的优点,搜索更多目标问题的执行路径,从而提高代码覆盖率,同时运用污点分析检测每条路径,并用依赖路径的污点信息指导 Fuzzing 测试生成相关的测试用例,以此发现程序内的问题。T-Fuzz[12]采用程序变换技术删除被测程序中复杂的完整性检查过程,从而暴露目标程序的潜在漏洞,然后跟踪触发漏洞的执行过程信息,收集源程序的路径约束,判断它们的可满足性以消除误报。

动态符号执行技术存在路径爆炸问题,为此业界提出了不同的路径搜索算法,通过选择策略覆盖关键路径,其中最具代表性的是 SAGE 系统[13]。它设计了代搜索(generation search)算法,使用启发式搜索策略,对搜集到的路径条件中的分支约束依次进行取反求解,生成新用例并将它们依次执行然后统计代码覆盖率,依据代码覆盖率为各新用例打分,接着在符号执行过程中选取打分高的用例执行,然后重复上述过程。代搜索虽然有利于提高代码覆盖率、缓解路径爆炸问题,但是打分过程开销大,影响了算法的性能。另外,DyTa 在动态阶段,采用文献[14]方法,根据静态阶段获得的信息指导 DSE 搜索程序的路径,并使用静态发现潜在缺陷的定位技术,对与反转不相关的分支节点进行剪枝,从而使 DSE 过程更高效。

约束求解是动态符号执行的必要过程,但是现有求解器无法求解所有约束,同时求解器的运行效率较低导致动态符号执行的效率低下。本文提出一种验证污点分析结果正确性的半自动方法,首先插桩并手工运行 Android 应用获得一条覆盖 Source 和 Sink 的 Trace。接着对 Trace 进行污点分析判定是否存在从 Source 到 Sink 的污点传播路径,是则验证结果正确;否则收集 Trace 的条件约束和污点信息,结合活变量分析和程序变换[12]的方法设计约束选择策略,对可行路径集合进行剪枝和遍历,判定是否存在污点传播路径,进而验证分析结果是否虚警。该方法没有采用动态符号执行生成测试用例,而是使用程序变换技术将程序中的条件约束逐一取反,生成变换后的程序并按照原有执行动作重复执行,进而获得其他执行路径信息,从而有效提高路径覆盖率。

2　半自动验证方法

本文方法的总体结构如图 1 所示,其执行流程如下:

1)被检测的 APK 静态插桩,生成新的. dex 文件,使转换后的 Android 应用在执行时能够记录程序的执行路径信息(Trace)。

2)将新生成的. dex 文件打包成插桩后的 APK,并安装到 Android 模拟器或真机中。

3)在保证程序执行时能同时覆盖 Source 和 Sink 的前提下,手工执行插桩后的 Android 应用并记录执行时的操作序列(events),得到程序执行结束后的 Trace,即种子 Trace。

4)分析模块对种子 Trace 进行别名分析和污点分析,获得程序执行过程中的运行时信息。

5)根据污点分析结果,如果发现一条从 Source 到 Sink 的污点传播路径,则整个验证过程结束,报告缺陷真实存在;否则,遍历 Source 与 Sink 之间的所有其他可能的执行路径,并对遍历过程中产生的每一条 Trace 进行污点分析,判断其中是否存从 Source 到 Sink 的未经验证的污点传播路径,如果存在则停止遍历,验证结束并报告缺陷真实存在。

6)直到遍历完所有路径后都没有发现一条从 Source 到

Sink 的污点传播路径,则结束验证过程,报告该缺陷是虚警。

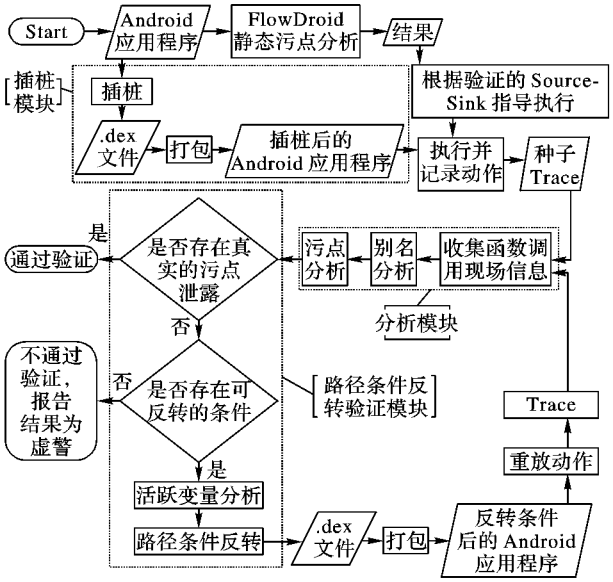


图 1 总体结构

Fig. 1 Overall structure

2.1 污点分析

手工执行经插桩的 Android 应用获得的种子 Trace 本质上是一条顺序的代码序列,污点分析的目标是根据 Trace 中的信息分析它记录的每条指令处的全局污点信息。Java 定义的变量都是以引用的形式标识程序运行时具体的内存位置,因此会有不同变量指向同一块内存空间,即它们互为别名,污点分析必须建立在准确的别名信息基础之上。Android 应用存在大量的方法调用,特别是事件触发的回调方法和注册监听组件事件的处理方法,在进行别名分析之前需要收集 Trace 中的方法调用现场信息,包括发生方法调用的位置信息和实参与形参之间的映射关系。

图 2 是分析模块的内部层次结构,方法调用现场信息收集子模块处在最底层,作为整个分析模块的基石,在别名分析时需要查找方法调用现场信息以确定实参到形参的别名数据流走向,进而别名分析又为污点分析提供支撑。

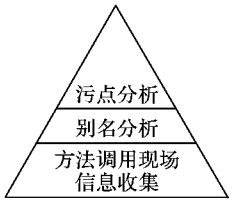


图 2 分析模块的层次结构

Fig. 2 Hierarchical structure of analysis module

本文方法没有对底层系统调用库、JDK 和 SDK 库方法的内部数据流进行污点分析,而是采用建模的方式定义库方法的污点传播摘要,根据摘要来记录调用库方法前后内存中污点信息的变化,同时,也对库方法中的验证方法(Sanitizer)建模定义无害化处理的污点传播摘要。方法采用白名单结合正则匹配的策略对自定义验证方法进行识别,主要基于方法名称、传递参数类型和返回值类型。例如,对于名字中包含“validate”“encrypt”或“check”等子串的方法调用语句,如果参数类型和返回值类型的签名满足预定义规则,就使用预定义的污点传播摘要直接生成方法调用后的内存污点信息。

2.2 路径条件反转

在遍历 Source 到 Sink 之间的所有可执行路径时,为缓解路径爆炸的问题,方法设计了一种路径条件的选择策略,以程序变换的方式反转选取的路径条件,重新生成新转换的 Android 应用,然后将原执行动作序列(events)重放于转换后的 Android 应用,进而获得包含其他分支路径信息的 Trace。选择策略是选取同时满足以下两点的条件语句:

1)在程序的反向跨方法控制流图(Interprocedural Reverse Control Flow Graph, IRCFG)中,剪去位于 Sink 前方的子图,在剩下的子图中,以 Sink 为起点进行跨方法的活变量分析,要求条件语句的活变量集合中必须至少有一个污点变量。

2)如果一条条件语句与 Sink 属于同一个方法体,那么在这个方法体对应的控制流图(Control Flow Graph, CFG)中,从这条条件语句的另一个分支出发的路径集合中至少有一条路径会经过 Sink 节点。

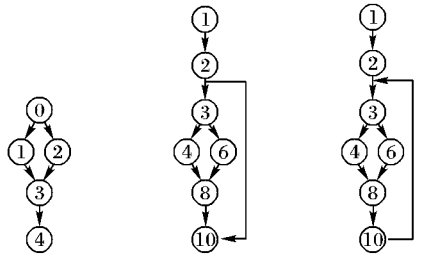
如果在某条条件语句处已经不存在任何活的污点变量,说明 Source 在该条件语句之前已经被验证过或后续没有任何与 Source 相关的数据流传播,那么在该条件语句后面的所有分支路径就不可能从 Source 传播到 Sink 处,即不存在从 Source 到 Sink 的污点传播路径,因此没有必要反转此条件。而与 Sink 属于同一方法体的条件语句,必须满足从该条件语句的另一个分支出发的路径集合中至少有一条路径会经过 Sink 语句;否则反转后产生的新 Trace 不会经过 Sink,更不可能存在 Source 到 Sink 的污点传播路径。

条件语句处的活变量信息通过对 Android 应用进行跨方法的活变量分析得到。活变量分析问题是一种典型的数据流分析问题,FlowDroid[15]将跨方法的数据流分析问题统一转换为程序间的有限分配子集(Interprocedural, Finite, Distributive Subset, IFDS)[16]问题,按照框架抽取的“exploded super graph”中的不同流边定义相应的流处理方法,操作具体数据事实的传播动作即可实现活变量分析。Android 程序在执行过程会大量调用系统回调方法,例如 Activity 的 onCreate、onResume, Button 按钮注册的点击事件方法等,这些方法没有显示调用。在 FlowDroid 构建的跨方法控制流图(Interprocedural CFG, ICFG)中,这些回调方法可能会成为孤立节点,也就是说,从 ICFG 中的入口节点无法到达这些孤立点。因此在进行静态分析时,通过它们传递的方法间数据流事实将会丢失,导致分析结果不精确。因此,方法在实现活变量分析时作了保守处理,认为这类回调方法将出口处的活变量数据流事实传递给了 ICFG 中所有其他节点,但是不包括调用方法内部的节点。

为判定从条件语句的另一个分支出发的路径集合中是否至少存在一条路径经过 Sink,方法引入必经节点(dominator)的概念,如果每一条从流图的入口节点到节点 n 的路径都经过节点 d,则认为 d 支配(dominate)n,称作 d 是 n 的必经节点,记为“d dom n”。例如图 3(a),从节点 0 出发,3 号节点是 4 号节点的必经节点;并且每个节点都是自己的必经节点。

通过反转路径条件来遍历 Source 到 Sink 每条可行路径时,需要反转的每条条件语句的分支汇聚点必须在 Sink 之前。也就是说,沿着其在 CFG 中不同分支路径的汇聚点开始深度遍历 CFG 产生的节点序列必须包含 Sink 节点。因为

Trace 中出现的每条条件语句已经有一条分支路径经过了 Sink 节点,所以如果从该条件语句的两个不同分支出发的两个路径集合都至少存在一条经过 Sink 节点的路径,显然该条件语句的另一个分支出发的路径集合满足至少存在一条经过 Sink 的路径。



(a) Dominator示意图　(b) CFG　(c) RCFG

图 3　程序流图实例

Fig. 3　Example of program flow graph

根据上述分析,在判断条件语句是否满足第 2)条选择策略的方法时,生成方法体的反向控制流图(Reverse Control Flow Fraph, RCFG),选择同时满足以下条件的条件语句进行反转:

1)以 Sink 节点作为起点,Sink 节点是条件语句节点的必经节点(dominator);

2)从 Sink 节点开始深度遍历 RCFG 得到的节点序列,包含条件语句节点的两个直接前继节点。

Java 代码片段如下:

```
public static void main(String[ ] args) {
  1)  String line = source( );
  2)  if (line. equals("save") ) {
  3)    if (Math. random( ) > 10) {
  4)      System. out. println("Inner branch1. ");
  5)    } else {
  6)      System. out. println("Inner branch2. ");
  7)    }
  8)    sink(line);
  9)  }
  10) System. out. println("Finish!");}
```

图 3(b)中每个节点的标号对应代码行号。以这个 CFG 为例,置反后就得到图 3(c)中的 RCFG。在 RCFG 子图中,以节点 8 作为起点,它是节点 3 的必经节点,满足条件 1)。然后从节点 8 开始深度遍历,获得的节点序列是 8→4→3→2→1→6,其中包含节点 3 的直接前继节点 4 和节点 6,满足条件 2),所以可以选择反转节点 3 处的条件。然而,前面得到的深度遍历序列中,只包含了节点 2 的直接前继 3,没有包含另一个前继节点 10,所以不满足条件 2),因此对节点 2 处的条件语句不能进行反转。

3　原型实现

原型系统由插桩模块、别名分析模块、污点分析模块和路径条件反转验证模块组成,如图 4 所示。

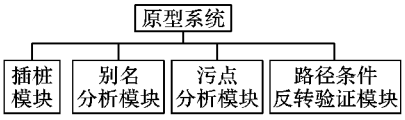


图 4　原型系统模块

Fig. 4　Diagram of prototype system modules

3.1　别名分析模块

别名分析的基础是方法调用现场信息,重点是方法调用过程中的实参与形参的映射关系。定义数据结构“Stack〈HashMap〈String, Object〉〉”记录方法调用现场信息,每个现场元素以 HashMap〈String, Object〉键-值对的形式存储,包含两种信息:一是“position”,表示方法调用语句在 Trace 中的位置信息,直接从 Trace 中记录的语句信息获得。二是“actual_formal_map”,使用“LinkedList〈Pair〈Object, Object〉〉”类型,根据方法签名存储实参与形参之间的映射关系,Pair 的第一个元素表示实参,第二个元素表示形参;LinkedList 中最后一个元素用于记录实例方法的 this 引用的传递信息,如果不是实例方法调用语句则不记录。

别名分析模块按 Trace 中的指令顺序模拟实际运行时动态分配的内存空间,在每块内存空间中记录所有指向该内存空间的别名引用,即别名集合。根据不同语句类型判断别名信息的传递,进而跟踪内存空间的别名信息的变化。内存空间的数据结构定义如下:

LinkedList〈Pair〈HashSet〈HashMap〈String, Object〉〉,
HashSet〈HashMap〈String, Object〉〉〉〉

以链表的形式存储程序申请的所有内存块,其中每一个 Pair 代表一个内存块,在每个内存块中记录了两种信息:别名信息和内存块信息。它们各映射成一个集合(HashSet),分别是 PointsToSet 和 BlocksSet。PointsToSet 记录所有指向该内存块的变量,集合中的所有变量它们之间都互为别名。BlocksSet 记录的是内存块集合。集合中的元素类型是 HashMap〈String, Object〉,每个元素记录了申请内存块的位置信息、内存块的子空间位置信息和内存块的污点信息,具体字段记录的内容如表 1 所示。

表 1　内存块记录的信息

Tab. 1　Recorded information of memory block

字段名	记录内容含义
block_id	内存块在内存块链中的编号
block_position	在 Trace 中分配内存块的位置
block_sub_instances	内存块的所有子域对象指向的内存块位置信息
block_sub_arrayItems	内存块的所有数组元素对象指向的内存块位置信息
block_tainted_state	内存块的污点状态信息

依照表 1 的定义,顺序遍历 Trace 中的语句信息,分析每条指令并跟踪别名信息的传递过程。对 Android 应用执行时的别名信息产生影响的语句类型共有四种,分别是参数传递语句(IdentityStmt)、赋值语句(AssignStmt)、方法调用语句(InvokeStmt)和方法返回语句(ReturnStmt)。

3.1.1　参数传递语句(IdentityStmt)

在分析参数传递语句时,查找记录的方法调用现场信息中是否包含实参与形参之间的映射关系,如果包含则将形参指针信息添加到实参所指向的内存块的别名指针集合(PointsToSet)中。例如图 5 中第 13)行发生的自定义方法调用,后面紧跟着参数的传递过程,实参$r6、“Jordan”和 26 分别传递给下面的 $r1、$i0和 $i0(图 5 中箭头①②③),即它们两两互为别名。

由于 Android 程序中存在大量的底层系统回调方法,并

且 Trace 仅包含 APK 的应用程序代码，不包含 Android 框架代码，所以有时会无法匹配方法调用时实参和形参的映射关系。在这种情形下需要按参数传递语句右值的不同类型分别记录数据流传递：

```
1) $r0 := @this: com.example.showdemo.MainActivity
2) specialinvoke $r0.<android.app.Activity: void <init>()>()
3) return
4) $r9 := @this: com.example.showdemo.MainActivity
5) $r8 := @parameter0: android.os.Bundle
6) specialinvoke $r9.<android.app.Activity:
     void onCreate(android.os.Bundle)>($r8)
7) virtualinvoke $r9.<com.example.showdemo.MainActivity:
     void setContentView(int)>(2130903040)
8) $r5=virtualinvoke $r9.<com.example.showdemo.
     MainActivity:java.io.FileOutputStream
     openFileOutput(java.lang.String,int)>("file.txt", 32768)
9) virtualinvoke $r5.<java.io.FileOutputStream:
     void close()>()
10) $r6 = new com.example.showdemo.MainActivity$Person
11) <com.example.showdemo.MainActivity$Person:
      java.lang.String type> = "Person"
12) return
13) specialinvoke $r6.<com.example.showdemo.MainActivity
      $Person: void <init>(java.lang.String,int)>("Jordan", 26)
14) $r1 := @this: com.example.showdemo.MainActivity$
      Person
15) $r0 := @parameter0: java.lang.String
16) $i0 := @parameter1: int
17) specialinvoke $r1.<java.lang.Object: void <init>()>()
18) $r1.<com.example.showdemo.MainActivity$Person:
      java.lang.String name> = $r0
19) $r1.<com.example.showdemo.MainActivity$Person:
      int age> = $i0
20) return
21) $r7 = newarray (java.lang.Object)[2]
22) $r10 = virtualinvoke $r6.<com.example.showdemo.
      MainActivity$Person: java.lang.String getName()>()
23) $r1 := @this: com.example.showdemo.MainActivity$
      Person
24) r0 = $r1.<com.example.showdemo.MainActivity$Person:
      java.lang.String name>
25) return r0
26) $r7[0] = $r10
    ⋮
32) $r7[1] = $r11
33) $r12 = $r7[1]
34) $z0 = $r12 instanceof java.lang.Integer
35) if $z0 == 0 goto $r16 = <java.lang.System:
      java.io.PrintStream out>
36) $r13 = $r7[1]
37) $r14 = (java.lang.Integer) $r13
38) $i1 = virtualinvoke $r14.<java.lang.Integer:
      int intValue()>()
39) $i2 = $i1 + 1
40) $r15 = <java.lang.System: java.io.PrintStream out>
41) virtualinvoke $r15.<java.io.PrintStream:
      void println(int)>($i2)
42) $r16 = <java.lang.System: java.io.PrintStream out>
43) $i3 = lengthof $r7
    ⋮
51) $r18 = <com.example.showdemo.MainActivity$Person:
      java.lang.String type>
    ⋮
58) $i4 = $r6.<com.example.showdemo.MainActivity$Person:
      int age>
    ⋮
62) $r26 = new java.lang.RuntimeException
63) specialinvoke $r26.<java.lang.RuntimeException:
      void <init>(java.lang.String)>("Here is an exception!")
64) throw $r26
65) $r27 := @caughtexception
66) virtualinvoke $r27.<java.lang.RuntimeException:
      void printStackTrace()>()
67) return
```

图 5 Trace 执行语句片段示例

Fig. 5 Example of Trace snippet

1）右值类型是 ThisRef（图 5 第 4）行语句），代表这是 this 引用的传递。在记录的内存块链中反向查询与方法 this 引用类型一致的内存块，如果找到则近似认为左值引用是指向该内存块，否则直接视为在当前语句位置为左值分配新的内存空间；在图 5 第 4）行语句处没有与$r9匹配的实参，所以反向查找类型一致的内存块，定位到第 1）行的$r0所指向的内存块，即$r0与$r9互为别名。

2）右值类型是 ParameterRef（图 5 第 5）行语句），代表这是方法调用的参数传递，可直接认为左值在该语句处分配新的内存空间。

3）此外，参数传递语句中右值还有一种类型：CaughtExceptionRef，表示传递抛出异常变量的信息，如图 5 中第 65）行的语句；它实际上是接收上面第 64）行抛出的异常变量 $r26，别名信息的传递如图中④号箭头方向所示。为记录异常抛出时参数的传递，方法在遍历 Trace 的过程中定义一个栈（throw _ value _ stack），每遇到一条异常抛出语句（ThrowStmt）抛出异常变量时，就压栈记录抛出的异常变量所指向的内存块的位置信息。当遇到接收抛出异常变量的参数传递语句时，可直接将语句的左值引用指向 throw_value_stack 的栈顶元素代表的内存块。

3.1.2 赋值语句（AssignStmt）

赋值语句的特点是将左值引用指向右值标识的内存空间，记录赋值语句的别名数据流事实传递分为三步：

1）在别名指针集合中清除已记录的与左值相关的指针信息，同时清除包括记录相关内存块之间关系（父子域或数组元素域）的信息；

2）具体定位右值引用指向的内存块，将左值指针信息加入该内存块的别名指针集合中；

3）调整与左值相关的别名信息，如访问路径中父域记录的子域信息。

第 2）~3）需要根据赋值语句的右值和左值的具体类型进行不同的操作。

第 2）步定位右值引用指向的内存块，分为几种情况：

①右值是 Local（局部变量）或 CastExpr（强制类型转换表达式）时，在记录的内存块链中查找右值指向的内存块，然后直接将左值指针添加到内存块的别名指针集合中。例如图 5 中第 32）行的$r7[1]指向$r11 原来的内存块，第 37）行语句执行后$r13 和$r14 互为别名。

② 右值类型是 InvokeExpr、NewExpr、BinopExpr、InstanceOfExpr、UnopExpr 或 Constant 时，分别对应图 5 第 8）、10）、39）、34）、43）和 11）行的语句，即认为在执行语句处为左值分配新的内存空间。

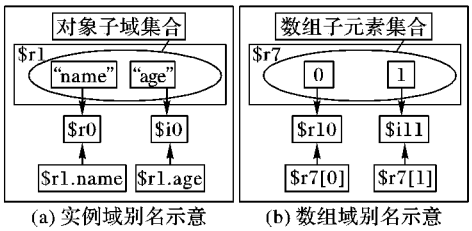
③ 右值类型属于 StaticFieldRef、InstanceFieldRef 或 ArrayRef 时，分别对应图 5 第 51）、58）和 33）行，如果不能找到右值对应的内存块，可以在满足污点传播一致性约束的前提下，根据父域的污点状态分配新的内存块。例如 a 已经是完全污染的，当第一次使用 a. f 对象时为其分配新的内存块，新内存块也标记为完全污染；但如果 a 是部分污染或可信时，新分配的 a. f 的内存块应该标记为可信。定位好右值指向的内存块后，直接将左值引用添加到内存块的别名集合中即可。

第 3）步调整与左值相关的别名信息，主要是对左值类型是静态域（StaticFieldRef）、实例域（InstanceFieldRef）和数组元素（ArrayRef）三种情形做调整：

1）左值是静态域时，找出所有类型是静态域所在类的类型的内存块，将这些内存块中记录的相应静态子域空间的位置标识修改为右值所指向的位置。例如图 5 第 11）行的静态

域 type 所在类的类型是 Person,所以查找所有类型是 Person 的内存块,并将这些内存块的 type 子域的内存空间位置标识成第 11)行右值所指向的内存位置。

2)左值是实例域或数组元素时,分别找到实例域的父对象或数组对象所指向的内存块,将记录子域内存块位置的标识修改为相应的右值的内存块位置。例如图 5 第 18)和 19)行$r1 的两个子域对象 name 和 age 分别指向$r0 和$i0,那么父对象$r1 中记录的子域集合中的信息也需要调整,如图 6(a)所示。再有图 5 第 26)和 32)行,分别将$r10 和$r11 赋值给数组元素$r7[0]和$r7[1],那么对应的数组对象$r7 中记录的数组下标 0 和 1 的元素分别指向$r10 和$r11 指向的内存块,如图 6(b)所示。



(a) 实例域别名示意　　(b) 数组域别名示意

图 6　别名示意图

Fig. 6　Diagram of alias

3.1.3　方法调用语句(InvokeStmt)

在 Java 中,方法调用语句不会对已记录的内存块链和别名集合造成太大的影响,加上之前已经对参数传递语句进行过分析(确保实参-形参互为别名),所以在方法调用语句处无需过多的操作。但有一种情况例外,就是调用对象实例初始化方法(〈init〉),该方法表示在被调用位置给方法调用的 this 变量分配内存空间;因此需要将 this 变量所对应的内存块分配位置(position)修改为〈init〉方法的调用位置。如图 5 中第 2)行的变量$r0,它指向的内存块的分配位置应该修改成这条语句所在的位置。一般调用〈init〉方法都是紧跟在实例对象 New 完之后,所以在调用〈init〉方法之前在内存块链中就已经记录了实例对象的内存块。

3.1.4　方法返回语句(ReturnStmt)

如果在方法调用现场接收方法的返回值,即方法调用现场是赋值语句,那么接收变量就与方法的返回值变量互为别名,如图 5 中⑤号箭头标识的$r1 接收返回语句的返回值 r0,它们互为别名。随后具体别名信息传递的操作与前面赋值语句的处理过程类似,相当于把赋值语句中的右值替换成方法返回语句的返回变量。

3.2　污点分析模块

根据获得的别名分析结果,将污点状态信息标记到别名变量指向的内存块上,以此来跟踪污点传播过程。污点分析中传递的数据流事实是被污染变量的污染状态集合,称为污点状态集合。在污点状态集合中的每个元素以二元组的形式定义(var, taint_level),其中:var 表示变量的访问路径(Access Path);taint_level 表示被污染变量的受污染程度。方法规定三种污染程度:部分污染(pa)、完全污染(ta)和可信(trust)。影响污点数据流传播的执行语句包括方法调用语句和赋值语句。

方法调用语句分为调用库方法和调用自定义方法。分析库方法调用时,以污点传播摘要的方式对库方法执行产生的污点信息流建模,根据具体的摘要调整并记录方法执行后各相关内存块的污点状态信息。分析自定义方法调用时需进一步递归分析方法体内部每条语句的污点传播语义来实现跨方法污点传播过程。

分析赋值语句时,首先按赋值语句右值的不同类型定义污点传播语义规则,依照规则记录污点传播过程;然后再按左值的不同类型,调整相关变量的污点状态信息。对内存块的污点状态信息的修改或调整都必须满足污点传播的一致性约束,避免错误记录污点传播信息。

在污点分析过程中,如果变量 a.f 的污点状态改变,那么 a 的污点状态应该作相应的调整。同时,对所有与 a 互为别名的实例域、数组元素和静态域,需要对与它们相关的内存块(父域内存块、数组对象内存块)的污点信息作进一步调整。此时存在一个向上递归调整相关变量的污点状态信息的过程,方法将它定义为 up_transmit_taint(var, tainted_level, deepth),其中:参数 var 表示发生污点状态改变的变量,tainted_level 表示改变的污染程度。递归深度变量 deepth 记录每次递归的深度,用于控制向上递归调整污点状态信息的最大递归层数(一般不超过 5 层)。up_transmit_taint 方法根据已发生污点状态改变的变量类型调整污点状态信息,分为如下三种情形(MustAlias(a)表示所有肯定与 a 互为别名的变量集合)。

1)发生污点状态变化的变量是实例域对象。根据实例域和它的父域的污点状态信息的不同,分别对污点状态信息做不同的调整操作,共存在三种情况:

① $(a.f, pa) \notin X \wedge (a.f, ta) \notin X \wedge (a, ta) \in X \rightarrow (b, pa) \in Y \wedge Y \cup up\_taint\_transmit(b, pa, --deepth);\ \forall b.c \in MustAlias(a)$

② $(a.f, pa) \in X \rightarrow (a, pa) \in Y \wedge (b, pa) \in Y \wedge Y \cup up\_taint\_transmit(b, pa, --deepth);\ \forall b.c \in MustAlias(a)$

③ $(a.f, ta) \in X \wedge (a, ta) \notin X \rightarrow (a, pa) \in Y \wedge Y \cup up\_taint\_transmit(b, pa, --deepth);\ \forall b.c \in MustAlias(a)$

第①种情况表示如果实例域是可信的并且父域是完全污染,那么将父域调整为部分污染,接着对父域的别名继续向上递归调整;第②种情况表示实例域是部分污染,那么直接将父域标记成部分污染,然后对父域的别名向上递归调整;第③种情况表示如果实例域完全污染,并且父域不是完全污染,那么父域应该调整为部分污染,接着对父域的别名递归向上调整。

2)发生污点状态改变的变量是数组元素时。为保证数组对象和各元素的污点状态信息的一致性,方法保守地规定数组中只要有一个元素不可信,就将整个数组标记为被完全污染,其中所有元素都标记为不可信,而且数组中所有元素的污点状态都保持一致,即数组元素要么可信要么完全污染。

3)发生污点状态改变的变量是静态域对象。调整相关污点状态信息的操作与 1)相同,但需要对其他与静态域所属类的类型相同的所有变量都进行调整,因为静态域属性被所有实例对象共享。

3.3　路径条件反转验证模块

为遍历 Source 到 Sink 的所有可执行路径,方法通过收集 Source 到 Sink 之间的条件语句,结合程序变换方法,反转路径条件来覆盖所有路径。为缓解路径爆炸问题,设计了反转条件的选择策略,即选择满足 2.2 节中的两条选择策略。

第 1)条选择策略判断是否存在活的被污染变量,通过静态的跨方法分析,收集每条语句处的活变量信息。方法基于 FlowDroid 中提供的 IFDS 框架,定义四种相应的流方法,完成对活变量数据流事实的传递,实现跨方法的活变量分析。算

法 1 用于判断第 2)条选择策略,结合反向 Dominator 和深度遍历方法。Global 声明全局变量,共定义了 4 个函数:Is_reverse、Domination、DFS 和 Contain。其中 DFS 是经典深度遍历算法,Domination 判断在图 graph 中以 start 为起点的图中各节点之间的必经节点(dominator)关系。Contain 分别从条件语句的两个分支开始遍历路径,判断遍历的节点序列是否都经过参数 stmt 的语句节点,结果记录在全局变量 contains 中。Is_reverse 函数判断条件语句是否可以加入反转条件的集合中。

算法 1 判断从条件语句的另一个分支出发的所有路径中至少有一条路径会经过 Sink。

输入 方法体的反向控制流图 RCFG;条件语句 IfStmt;污点汇聚节点 Sink。

输出 是否反转输入的条件语句。

```
Global: N, adjacent_matrix, domination, visited, contains,
nodeToId;
function Is_reverse(sink, IfStmt, RCFG)
  Domination(sink, RCFG);
  if domination[nodeToId.get(sink)][nodeToId.get(IfStmt)] then
    foreach pre in graph.getPredsOf(IfStmt) do
      /*getPredsOf 是在 RCFG 中得到 IfStmt 的所有前继节点*/
      contains ← false;
      Contains(IfStmt, sink, RCFG, set);
      if (contains = false) then
        return false;
      end if
    end foreach
  end if
  return true;
end function
function Domination(graph, start)
  N ← graph.size();
  count ← 0;
  foreach node in graph do
    nodeToId.put(node,++count);
  end foreach
  foreach node in graph do
    foreach succ in graph.getSuccsOf(node) do
      /* getSuccsOf 是在 graph 中得到 start 的所有后继节点 */
      adjacent_matrix[nodeToId.get(node)][nodeToId.get(succ)]
      ← true
    end foreach
  end foreach
  DFS(nodeToId.get(start), -1);
  reachable ← visited;
  for i ← 0 to N do
    DFS(nodeToId.get(start), i);
    for j ← 0 to N do
      if reachable[j] = true and visited[j] = false then
        domination[i][j] = true;
      end if
    end for
  end for
end function
function DFS(start, ignore)
  if start = ignore then
    return;
  end if
  visited[start] ← true;
  for i ← 0 to N - 1 do
    if (adjacent_matrix[start][i] = true and visited[i] ≠ true)
      then
      DFS(i, ignore);
    end if
  end for
end function
function Contain(stmt, start, graph, set)
  put start into set;
  if (start = stmt) then
    contains ← true;
  end if
  foreach succ in graph.getSuccsOf(start) do
    if (set not contains succ) then
      Contain(stmt, succ, graph, set);
    end if
  end foreach
End function
```

4 实验分析

原型系统基于 Soot-trunk 3.0 和 FlowDroid 2.0 框架实现,使用 JDK 1.8 开发,总计 8 000 余行代码,其中插桩模块 1 400 余行,分析模块 4 300 余行,路径条件反转验证模块 2 200 余行。实验环境为 Genymotion 搭建的模拟器,运行系统版本为 Android 4.4,操作系统版本是 Ubuntu 18.04.1 LTS,处理器 i5-3230M,CPU 2.6 GHz,内存 8 GB。

实验测试选取 DroidBench 2.0 作为测试数据集,它包含了 13 类共 119 个 Android 应用。剔除跨组件通信、应用间通信和多线程等 3 类测试样本共 34 个,原型系统目前还不支持这三类的应用程序。另外还剔除了实验环境无法模拟的 10 个样本,并在 Android 开源软件仓库 F-Droid 和 Github 上采集 10 个真实的 Android 应用,最后,对 85 个 Android 应用样本进行实验。

FlowDroid 对其中 77 个样本报告 71 个泄露缺陷,每个缺陷都只存在一条可执行路径。经过别名分析和污点分析后,方法准确地验证其中 7 个泄露是虚警。

下面以实验中的一段 Android 程序源码片段作为例,简述在实验过程中,验证只存在一条可执行路径的泄露缺陷是否虚警的流程。

```
…
3)  protected void onCreate(Bundle savedInstanceState) {
      …
7)    button1.setOnClickListener(new View.OnClickListener() {
8)      public void onClick(View view) {
9)        SmsManager sms = SmsManager.getDefault();
          …
11)       sms.sendTextMessage(number, null, imei, null, null);
                                          // sink, potential leak;
12)       Log.i("TAG", "sendIMEI:" + imei);
                                          // sink potential leak
13)       imei = null;
14)     }
15)   });
17)   button2.setOnClickListener(new View.OnClickListener() {
18)     public void onClick(View v) {
19)       imei = null;
20)       Log.i("TAG", "Button2:" + imei);     // sink, leak
21)     }
```

```
22)    });
23)  }
24)  public void clickOnButton3(View view) {
25)    TelephonyManager telephonyManager = (TelephonyManager)
       getSystemService(Context.TELEPHONY_SERVICE);
26)    imei = telephonyManager.getDeviceId();          // source
27)    Log.i("TAG", "Button3:" + imei);          // sink, leak
28)  }
     ...
```

FlowDroid 报告其中第 26)行 Source 分别传播到第 11)、12)、20)和 27)行 Sink 的 4 条泄露。但其中 button2 按钮点击事件的处理方法在第 20)行 Sink 调用 Source 变量 imei 之前,imei 变量已经被置空,即此时 imei 变为可信的,同时,由于这对 Source-Sink 之间只存在一条路径,因此方法判断这条(Source,Sink)泄露是虚警。

针对其余的 8 个样本程序的实验结果如表 2 所示。

表 2　采用本文方法验证 FlowDroid 告警的实验结果

Tab. 2　Experimental results of validation for alarm in FlowDroid by the proposed method

编号	名称	代码行数	验证条目数	反转路径数	总路径数	验证结果	平均减少反转路径占比/%
1	Merge1	83	1	2	2	虚警	—
2	VirtualDispath1	68	1	0	2	泄露	—
			1	4	4	虚警	0.00
3	PrivateDataLeak1	94	1	0	8	泄露	—
4	Encrypt	100	1	4	32	虚警	87.50
			1	0	1	泄露	—
5	LocalLogin	220	6	0	1	泄露	—
			3	0	8	泄露	—
			2	0	4	泄露	—
			2	4	64	虚警	93.75
			2	4	8	虚警	50.00
6	SendMail	554	3	0	8	泄露	—
			2	0	32	泄露	—
			2	0	8	虚警	87.50
			1	0	2 048	虚警	99.90
			1	0	4 096	虚警	99.90
			2	4	1 024	虚警	99.60
			1	4	64	虚警	93.75
7	KnonWeather	119	1	0	4	虚警	75.00
8	通讯管家	2 279	2	0	256	泄露	—
			1	0	128	泄露	—
			4	0	1	虚警	—
			6	0	1	泄露	—
			5	0	2	泄露	—
			5	0	8	泄露	—
			2	0	16	泄露	—
			4	0	64	泄露	—
			1	0	128	虚警	99.90
			1	0	4	虚警	75.00
			3	0	4	泄露	—
			1	0	8	虚警	87.50
			1	0	2	虚警	—

表 2 中第 2 组实验获得的种子 Trace 片段结构如下:

```
1)  $r0 = source();
    ...
2)  if $i3 != 0 goto r0 = new de.ecspride.DataLeak
    ...
3)  sink1($r0);
    ...
4)  if $i3 != 0 goto r0 = new de.ecspride.DataLeak
    ...
5)  sink2("no leak");
    ...
6)  use($r0)
```

FlowDroid 报告了 2 条泄露,分别是从第 1)行 source 执行到第 3)和 5)行的 sink1 和 sink2。对于从 source 到 sink1 的隐私泄露,在对种子 Trace 作污点分析时可以发现从 source 到 sink1 的污点传播路径,因此直接报告隐私泄露真实存在,无需反转。

而验证 source 到 sink2 的隐私泄露,分析它们之间总共存在 4 条路径,路径遍历过程收集的反转路径条件语句是第 2)和 4)行的条件语句。经反转,包括种子 Trace 在内共遍历 4 条路径,没有发现任何从 source 到 sink2 的污点传播路径,验证结果是虚警。

表 2 中第 4 组实验过程中记录的种子 Trace 片段如下:

```
1)  $r1 = source();
    ...
```

```
2) if $z1 == 0 goto $r9 = $r4. < com. example. encrypt.
   MainActivity: java. lang. String selected_algorithm >
   …
3) if $z3 == 0 goto $r10 = $r4. < com. example. encrypt.
   MainActivity: java. lang. String selected_algorithm >
   …
4) encrypt($r1);
   …
5) if $i0 < $i2 goto $r10 = new java. lang. StringBuilder
   …
6) if $z2 == 0 goto $r11 = $r4. < com. example. encrypt.
   MainActivity: java. lang. String selected_length >
   …
7) if $z0 == 0 goto virtualinvoke $r0. < android. widget.
   TextView: void setText(java. lang. CharSequence) > ($r6)
   …
8) sink($r1);
```

虽然在第 8)行的 sink 处引用了第 1)行的污点变量$r1,但是在第 4)行调用的 encrypt 方法对$r1 作了无害化处理,所以从第 4)行往下的变量$r1 变成可信。根据路径条件反转策略,虽然$r1 在第 5)~7)行都是活变量,但已经不是污点变量,因此不满足第①条选择策略,只需将第 2)和 3)行的条件语句加入到路径条件反转集合中。

在第 5 组实验里获得的种子 Trace 片段如下所示:

```
1) $r6 = source();
   …
2) if $r3 != null goto $r5 = < com. example. testtargetsample.
   DBHelper: com. example. testtargetsample. DBHelper instance
   >
   …
3) if $z0 == 0 goto $r43 = new java. lang. StringBuilder
   …
4) sqlEscapeString($r6);
   …
5) if $z2 == 0 goto $z3 = virtualinvoke $r8. <java. lang. String:
   Boolean equals(java. lang. Object) > ("administrator")
   …
6) if $z3 == 0 goto (branch)
   …
7) if $z4 == 0 goto (branch)
   …
8) if $z5 != 0 goto $i2 = interfaceinvoke $r37. < android.
   database. Cursor: int getInt(int) > (1)
   …
9) sink($r6);                    //(sink 反转在 6、7、8 的分支内部)
```

在第 4)行位置调用 sqlEscapeString 方法对污点变量$r6 作了无害化处理,所以第 4)行之后的语句已经没有活的污点变量。并且第 6)~8)行的条件语句不满足第②条路径条件反转的选择策略。虽然存在 64 条执行路径,但只需要反转执行 3 条路径即可。

第 8 组实验报告总共存在 128 条路径,但是无需反转即可验证结果是虚警,获得的种子 Trace 如下所示:

```
1) $r0 = source();
   …(中间存在多条条件语句)
2) sink("no leak");
   …(没有使用到污点变量$r0)
```

Source 到 Sink 之间的所有条件语句处的污点变量$r0 都已经死了,即后续根本没有使用污点变量,所以这些条件语句都不用反转。

MultiFlow 通过检测是否组合绑定的多对 Source 能同时触发 Sink 来降低虚警率,包括两个分析阶段:1)单源分析,检测每一个 Source 是否传播到 Sink;2)多源分析,以组合的方式检测多个 Source 能否绑定同时触发 Sink。

对比实验使用 MultiFlow 对相同的应用测试集进行分析,表 3 给出了 MultiFlow 与本文分析方法的实验结果对比。

表 3 MultiFlow 实验结果

Tab. 3 Experimental results of Multiflow

编号	名称	单源分析数	多源分析数	减少虚警率对比	
				本文方法结果	MultiFlow 分析结果
1	Merge1	1	0	100.00	0.00
2	VirtualDispatch1	1	0	50.00	50.00
3	PrivateDataLeakage1	1	0	—	—
4	Encrypt	2	0	50.00	0.00
5	LocalLogin	10	3	26.67	33.33
6	SendMail	8	0	58.33	33.33
7	KnowWeather	1	0	100.00	0.00
8	通讯管家	32	0	22.22	11.11

第 1 组和第 7 组应用只存在一个 Source,MultiFlow 的多源分析阶段被阻断,导致 MultiFlow 未能正确验证虚警。虽然 MutiFlow 对第 5 和第 8 组应用正确验证了不少虚警,但是经过进一步人工分析源码,MultiFlow 没有精确跟踪回调方法间的污点传播,导致将发生在回调方法间传递的污点传播路径误判为虚警。

在 MultiFlow 对 DroidBench 的验证结果中,有 3 组应用的真实报警被误判为虚警。两组发生在回调方法中:一组是回调方法修改 SharedPreference 过程中的污点传播;另一组是回调方法构造 Fragment 过程中的污点传播;第三组是在多源分析阶段将 2 条泄露报警错误地合并成一条泄露报警。因此,本文方法相比 MultiFlow 的验证结果更为可靠。

在 85 个样本应用中,平均遍历路径比为 15.09%,虚警率平均降低 58.17%,MultiFow 将虚警率平均降低了18.25%。

实验结果表明,本文方法能高效可靠地验证静态分析结果的正确性,在验证过程中裁剪不必要的路径遍历,极大缓解路径爆炸问题,提升验证效率。

5 结语

本文提出一种半自动验证污点分析结果的正确性的方法,实现基于 Trace 的别名分析和污点分析,判定 Trace 中是否有 Source 到 Sink 的污点传播。设计一种路径剪枝方法,提出结合活变量分析的路径约束选择策略,以程序变换的方式搜索路径,极大缓解路径爆炸问题,基于 FlowDroid 实现的原型系统对真实的 Android 应用分析表明了方法的有效性。

本文方法的不足主要包括:1)不支持跨组件、跨应用程序之间的数据通信过程的污点分析,对多线程并发的处理也不够完善;2)不支持 Java 语言的某些特性如反射,另外,对 Android 库方法的建模不完备,可能导致污点分析不精确;3)需要手工执行 Android 应用获得同时覆盖 Source 和 Sink 的初始种子 Trace,当程序规模较大时,该项工作费时费力。

未来工作主要包括:1)研究跨组件污点数据流事实的传播;2)对 Android 库方法的污点传播语义进行更为完备的建模;3)研究结合 Fuzzing 测试和动态符号执行来获得初始种子 Trace。

参考文献(References)

[1] 王蕾, 李丰, 李炼, 等. 污点分析技术的原理和实践应用[J]. 软件学报, 2017, 28(4): 860 - 882. (WANG L, LI F, LI L, et al. Principle and practice of taint analysis[J]. Journal of Software, 2017, 28(4): 860 - 882.)

[2] 赵云山, 宫云战, 王前, 等. 静态缺陷检测中的误报消除技术研究[J]. 计算机研究与发展, 2012, 49(9): 1822 - 1831. (ZHAO Y S, GONG Y Z, WANG Q, et al. False positive elimination in static defect detection[J]. Journal of Computer Research and Development, 2012, 49(9): 1822 - 1831.)

[3] 李筱, 周严, 李孟宸, 等. C/C++程序静态内存泄漏警报自动确认方法[J]. 软件学报, 2017, 28(4): 827 - 844. (LI X, ZHOU Y, LI M C, et al. Automatically validating static memory leak warnings for C/C++ programs[J]. Journal of Software, 2017, 28(4): 827 - 844.)

[4] GE X, TANEJA K, XIE T, et al. DyTa: dynamic symbolic execution guided with static verification results[C]// Proceedings of the 33rd International Conference on Software Engineering. New York: ACM, 2011: 992 - 994.

[5] 王蕾, 周卿, 何东杰, 等. 面向 Android 应用隐私泄露检测的多源污点分析技术[J]. 软件学报, 2019, 30(2): 211 - 230. (WANG L, ZHOU Q, HE D J, et al. Multi-sources taint analysis technique for privacy leak detection of Android apps[J]. Journal of Software, 2019, 30(2): 211 - 230.)

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

[7] YANG Z, YANG M, ZHANG Y, et al. AppIntent: analyzing sensitive data transmission in Android for privacy leakage detection[C]// Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. New York: ACM, 2013: 1043 - 1054.

[8] ARZT S, RASTHOFER S, HAHN R, et al. Using targeted symbolic execution for reducing false-positives in dataflow analysis[C]// Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis. New York: ACM, 2015: 1 - 6.

[9] JUNKER M, HUUCK R, FEHNKER A, et al. SMT-based false positive elimination in static program analysis[C]// Proceedings of the 2012 International Conference on Formal Engineering Methods, LNCS 7635. Berlin: Springer, 2012: 316 - 331.

[10] ZHANG L, THING V L L. A hybrid symbolic execution assisted fuzzing method[C]// Proceedings of the 2017 IEEE Region 10 Conference. Piscataway: IEEE, 2017: 822 - 825.

[11] CAI J, YANG S, MEN J, et al. Automatic software vulnerability detection based on guided deep fuzzing[C]// Proceedings of the IEEE 5th International Conference on Software Engineering and Service Science. Piscataway: IEEE, 2014: 231 - 234.

[12] PENG H, SHOSHITAISHVILI Y, PAYER M. T-Fuzz: fuzzing by program transformation[C]// Proceedings of the 2018 IEEE Symposium on Security and Privacy. Piscataway: IEEE, 2018: 697 - 710.

[13] GODEFROID P, LEVIN M Y, MOLNAR D. SAGE: whitebox fuzzing for security testing[J]. Communications of the ACM, 2012, 55(3): 40 - 44.

[14] TANEJA K, XIE T, TILLMANN N, et al. Guided path exploration for regression test generation[C]// Proceedings of the 31st International Conference on Software Engineering. Piscataway: IEEE, 2009: 311 - 314.

[15] 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(6): 259 - 269.

[16] BODDEN E. Inter-procedural data-flow analysis with IFDS/IDE and soot[C]// Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis. New York: ACM, 2012: 3 - 8.

This work is partially supported by the National Natural Science Foundation of China (61562040, 61762049), the Science-Technology Project of Education Bureau of Jiangxi Province(GJJ161305, GJJ151330).

QIN Biao, born in 1993, M. S. candidate. His research interests include information security, program verification.

GUO Fan, born in 1977, Ph. D., associate professor. His research interests include network security, application security.

TU Fengtao, born in 1976, M. S., lecturer. His research interests include network security.