计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2019.20180071
56(3):480-495,2019

基于稀疏框架的静态污点分析优化技术

王 蕾 何冬杰 李 炼 冯晓兵
(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)
(中国科学院大学 北京 100049)
(wanglei2011@ict.ac.cn)

Sparse Framework Based Static Taint Analysis Optimization

Wang Lei, He Dongjie, Li Lian, and Feng Xiaobing
(*State Key Laboratory of Computer Architecture* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)
(*University of Chinese Academy of Sciences*, *Beijing* 100049)

Abstract At present, privacy preserving is an important research challenge of information system security. Privacy leak detection for applications is an effective solution for privacy preserving. Taint analysis can effectively protect the confidentiality and integrity of information in the system, and report the privacy leak risk of applications in advance. However, the existing static taint analysis tool still has the problem of high analysis overhead especially in high sensitive mode. This work first deeply analyzes that there exists a large number of unrelated propagation which leads to unnecessary expenses in current IFDS-based taint analysis, and statistical results show that the proportion of it is up to 85.5%. Aiming at this problem, this paper attempts to use an effective optimization method, sparse optimization in recent years, to eliminate the unrelated propagation in static taint analysis, and achieve the optimization of time and space cost. We have innovatively extended the classic data flow analysis framework (IFDS) into a sparse form, and provide a corresponding taint analysis algorithm. We implemented a tool called FlowDroidSP. Experimental results show that the tool has 4.8 times of time acceleration and 61.5% memory reduction compared with the original FlowDroid under the non-pruning mode. Under pruning mode, it has an average time of 18.1 times speedup and 76.1% memory reduction.

Key words privacy leakage detection; static program analysis; taint analysis; program optimization; Android

摘 要 当前,隐私数据保护是信息系统安全的重要研究挑战,对应用程序进行隐私泄露检测是隐私泄露保护的有效方案.污点分析技术可以有效地对应用程序进行保密性和完整性的安全检测,提前报告出潜在的隐私泄露风险.然而,当前高敏感度的静态污点分析还存在开销过高的问题.通过对目前主流的污点分析工具FlowDroid进行深入分析,发现其污点分析计算中大量无关联污点传播是导致开销过高的重要原因,统计实验表明无关联传播占比高达85.2%.针对这一问题,尝试利用近年来一种有效的程序分析优化手段——稀疏优化——的方法,对静态污点分析中无关联的传播进行消除,达到时间和空间的开销优化.创新地将经典的数据流分析框架扩展成稀疏的形式,在此基础上提供了基于稀疏优化的污

收稿日期:2018-01-31;**修回日期**:2018-12-02
基金项目:国家自然科学基金项目(61521092,61432016);国家重点研发计划项目(2017YFB0202002)
This work was supported by the National Natural Science Foundation of China (61521092, 61432016) and the National Key Research and Development Program of China (2017YFB0202002).

点分析方法.最后实现了工具 FlowDroidSP,实验表明:FlowDroidSP 在非剪枝模式下相比原 FlowDroid 具有平均 4.8 倍的时间加速和 61.5%的内存降低.在剪枝模式下,具有平均 18.1 倍的时间加速和 76.1%的内存降低.

关键词　隐私泄露检测;静态程序分析;污点分析;程序优化;安卓

中图法分类号　TP311

当前,隐私数据保护是信息系统安全的重要研究挑战之一,大量的互联网(电子邮件、社交网络)、移动应用(移动支付、手机定位)、物联网(智能家居、智能医疗等)相关数据需要存放在应用端,如果应用程序因存在漏洞而被攻击者利用(完整性违反)或者第三方应用程序有意地窃取用户系统中本地的数据(保密性违反),则会产生隐私泄露问题[1].对系统应用中潜在的隐私数据泄露风险检测是当前该领域的重要研究方向.污点分析技术[2](taint analysis)是信息流分析技术[3](information-flow analysis)的一种实践方法.该技术通过对系统中敏感数据进行标记,继而跟踪标记数据在程序中的传播,检测系统的保密性和完整性等安全问题.据调查分析[4],在 Android 应用的静态安全检测中,污点分析是最为流行的分析方案.

污点分析又被分为静态污点分析和动态污点分析.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.相比动态运行的污点分析,静态污点分析无需对程序进行插桩而导致运行时出现额外开销,并且不依赖于测试用例输入,可以在程序发布之前对应用程序进行检测,避免发布之后造成的安全问题.然而,当前静态污点分析技术的分析性能仍然不能满足安全检测的需求.例如,MudFlow[5]尝试使用静态污点分析工具 FlowDroid[6]对当前 Android 应用市场的应用进行安全漏洞检测,它只能使用低敏感度的程序分析且在运行内存较大(730 GB)的环境下,一些例子仍然出现检测超时.并且,低敏感度的程序分析会报告出大量的误报,这会给检测人员带来诸多不便,同样降低了检测的效率,类似的问题还存在其他工作中[7-9].因此,对高敏感度的静态污点分析进行空间和时间上的优化是一个值得研究的问题.

最近,在程序分析领域中,大量的研究工作[10-18]尝试使用稀疏的方式对静态程序分析问题进行效率优化.所谓的稀疏程序优化是指数据流值(data flow fact)不再沿着控制流图(control flow graph, CFG)上进行传递,而转移到利用 DEF-USE 链得到的相关数据结构上传递.例如文献[10]利用了半稀疏(semi-sparse)的方法对指针分析进行优化,将指针数据流值从 CFG 上的传播转移到数据流图中进行传播.使用 DEF-USE 链代替 CFG 来传递数据流值可以消除不必要的传播计算从而达到优化的目的.

由于污点分析只是分析污点能否从污点源(source)传播到汇聚点(sink),是一种按需的分析方式,当前的主流静态污点分析(FlowDroid)是基于 IFDS 框架的,而目前的 IFDS 框架是没有对稀疏优化进行支持的.因此,本文的工作内容正是尝试利用稀疏的优化方法对静态污点分析进行优化.

本文的主要贡献包含 4 个方面:

1) 发现并分析当前基于 IFDS(interprocedural, finite, distributive, subset)问题的静态污点工具 FlowDroid 计算中存在大量无关联传播,并通过实验验证其比例平均高达 85.2%.

2) 为了支持稀疏的污点传播,提出了一种稀疏的 IFDS 数据流分析框架,提供面向流敏感和域敏感的变量使用点索引的计算方法.

3) 设计了基于稀疏数据流分析框架的污点分析技术.

4) 实现一个基于本文技术的原型系统 FlowDroidSP,经过实验验证,该工具在提升性能的同时没有带来精度上的损失.在非剪枝模式下,相比原 FlowDroid 具有平均 4.8 倍的时间加速和 61.5%的内存降低,在剪枝模式下,具有平均 18.1 倍的时间加速和 76.1%的内存降低.

1　背景知识

1.1　IFDS 数据流分析框架

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[19]进行求解:首先为每一个过程构建控制流图或者过程间的控制流图,然后根据不同敏感度(上下文敏感、流敏感等)进行具体的数据流传播分析(与指针/别名分析).IFDS 框架[20]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，其全称是过程间(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对并(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[20]中的 Tabulation 算法)，而污点分析正是满足 IFDS 算法要求的一个问题.污点分析的数据流值是污点变量，它表示有哪些污点变量可以到达当前程序点.

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[21].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 1 例子所示).其中数据流值可以被表示成图中的节点，算法扩展了一个特殊数据流值：0 值，用于表示空集合；程序分析中转移函数的计算，即对数据流值的传递计算被转化为图中的边的求解.为了更好地进行描述，算法根据程序的特性将分析分解成 4 种转移函数(边)：1)Call-Flow，即求解函数调用(参数映射)的转移函数.2)Return-Flow，即求解函数返回语句返回值到调用点的转移函数.3)CallToReturn-Flow，即函数调用到函数返回的转移函数.4)Normal-Flow，是指除了上述 3 种函数处理范围之外的语句的转移函数.IFDS 求解算法 Tabulation 是一种动态规划的算法，即求解过的子问题的路径可以被重复地利用，而由于其数据流值满足分配率，因此可以在分支或函数调用将边进行合并.求解算法包括了 2 类路径的计算：路径边集(path edges)和摘要边集(summary edges).路径边集表示的是起点 0 值到当前计算点的可达路径，使用传播函数 propagate 将计算完成的边继续沿 CFG 传播到其下一个节点；摘要边集表示的是函数调用到函数返回的边，其主要特点是如果在不同调用点再次遇到同样的函数调用，可以直接利用其摘要边集信息从而避免了函数内重复的路径边集的计算.

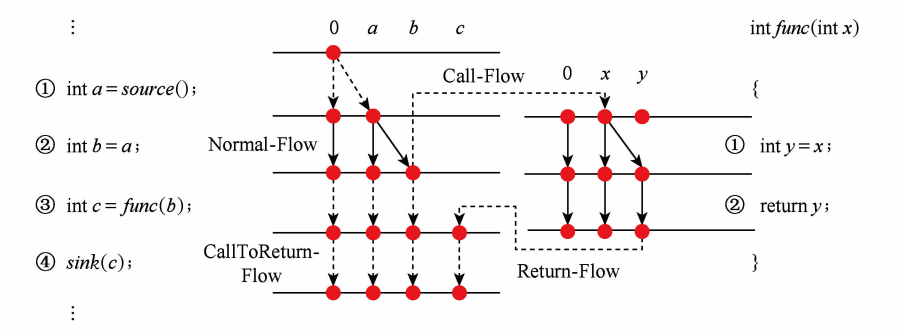


Fig. 1 An example of taint analysis by using IFDS algorithm

图 1 使用 IFDS 算法求解污点分析示例

1.2 FlowDroid 与别名分析

FlowDroid[6]是当前最为流行的面向 Java，Android 应用程序的开源静态污点分析工具，被广泛应用于 Android App 的隐私泄露检测和 Java 程序相关安全漏洞挖掘.FlowDroid 基于 Java 分析工具 Soot[22]，将待分析程序字节码转化成 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(source→sink)的集合.在污点传播分析中，FlowDroid 正是基于 IFDS 数据流分析框架.同时，FlowDroid 使用访问路径(access path，即形如$v.f.g$)来支持域敏感，如果访问路径长度超过预定值时，FlowDroid 将对其进行保守的裁剪(例如裁剪成 $v.f.*$ 的形式)，在具体实现中访问路径的长度一般设置为 5.因此，FlowDroid 具有流敏感、上下文敏感、域敏感的高精确度.

FlowDroid 的另一贡献就是提供了一种按需的面向别名的污点传播方法，该方法同样是基于 IFDS 框架的.具体来讲，FlowDroid 在遇到一个堆变量(heap object)的域进行污点赋值时，会启动一个后向的 IFDS 求解器求解其别名相关的污点值，如果生成与该变量别名的污点变量，则又启动一个前向的 IFDS 求解器进行前向传播该值.以图 2 中的伪代码为例，程序的行④污点源(函数 *source*()的返回值)赋值到了一个堆对象的域 $a.f$ 中，此时将启动一个后向的分析来寻找与 $a.f$ 别名的污点值(过程 1)，当后向求解器分析到 a 使用点行②时，将产生与 a 别名的对象 b 的对应污点值，即 $b.f$(过程 2)，同时启动一个前向分析，继续找 $b.f$ 的使用位置(过程 3).同理，过程 4/5/6/7 是对 $b.f$ 和 $c.f$ 的进一步传播.为了解决流敏感的分析，FlowDroid 引入激活点的概念，激活点的作用是保证未被激活的变量只有

通过了激活点之后才是一个真正的污点值,否则即便到达泄露点也不会触发泄露.例如在过程 6 中的 $c.f$ 变量只有再次经过了行④的激活点到达行⑤触发了 *sink*,才是一个真正的泄露.而对于过程 3 中的 $b.f$,虽然也到达了函数 *sink*,但是它没有经过激活点,因此行③的 *sink* 不能被触发.

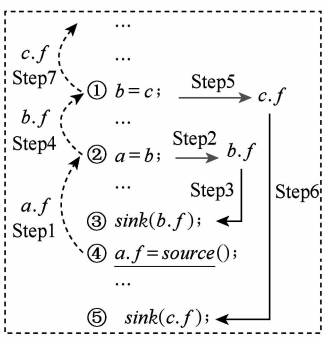


Fig. 2　An example of FlowDroid-style alias based taint analysis

图 2　FlowDroid 方式的别名间污点传播示例

本文的工作正是基于 FlowDroid 进行扩展优化的,目前 FlowDroid 中污点传播所使用的数据流分析无论是前向求解还是后向求解(计算别名)都是基于 IFDS 框架的.IFDS 框架的计算规模直接影响了污点分析的开销.然而,当前 IFDS 框架仍然是基于 CFG 或过程间 CFG 的,这会导致数据流分析中存在无关联的传播.所谓的无关联传播是指数据流值传播到某语句,但是该语句不会利用其生成其他数据流值或将其杀死,例如图 1 中变量 a 在行①被使用之后还会继续传递到行②～④直到程序结束,而行③～④中没有使用 a 产生其他数据流值,a 在行③～④上的传播正是无关联传播.因此本文的目标正是利用稀疏的优化方法来消除这种无关联的传播,从而达到效率优化的目的.

2　研究动机

在面向高精确(流敏感和域敏感)的污点分析的计算中,无关联的数据流传播还将被扩大,进而产生更大的性能开销.具体来讲:

1) 现代编译器利用中间表示对程序进行分析或者优化,基于 3 地址的中间表示被各大编译器采用(如 Soot 的 Jimple 表示).例如图 3(a)程序为一段程序的源码,图 3(b)程序是其 3 地址表示,这种转换过程必然会引入大量的临时变量,对于行②中的 $b.f.h=a.f$ 语句将被转化为程序图 3(b)中的行②～④的程序,其中利用了 2 个临时变量 $tmp1$,$tmp2$.这种临时变量的使用会增加无关联传播的规模.

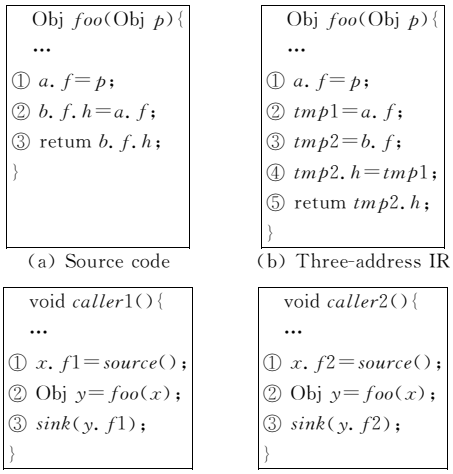


Fig. 3　An example of unrelated taint propagation

图 3　无关联污点传播示例

2) 为了支持域敏感的分析,FlowDroid 使用访问路径来表示污点变量,由于相同对象的不同域需要不同的区分,这会导致同一份代码因不同域的使用而被执行多次、无关联传播将会被放大.例如,图 3(c)和图 3(d)中程序都调用了函数 *foo*(),但是它们所传递的参数对应的域是不同的,分别是 $x.f1$ 和 $x.f2$.对于同一个对象但访问不同参数的情况,IFDS 认为它们是不同的变量而不会对其进行摘要优化.此时,参数 $p.f1$ 传递到 $tmp2.h$ 等过程产生的变量,如 $tmp1.f.f1$ 和 $tmp2.h.f1$ 等,那么对于 $p.f2$ 的计算,也同样需要再次生成,如变量 $tmp1.f.f2$ 和 $tmp2.h.f2$ 等,可见无关联传播增加了 1 倍.

3) 为了支持流敏感,FlowDroid 使用了激活点的方法,每一个未被激活的数据流都会存在一个激活点,即使数据流值中的访问路径相同但因激活点不同,数据流值仍然是不同的,同理 2)这也同样导致摘要优化失效.不同激活点的数据流值通过相同的代码片段将会导致这种无关联传播的增加.

为了验证上述无关联传播确实存在,以及计算其规模和所占比例,我们针对 FlowDroid 进行实验验证.

具体的评测对象是最新版本的 FlowDroid(2017 年 11 月),针对其 IFDS 计算边进行插桩统计,统计的规则有 3 项:

1) 对于 Normal Flow 和 CallToReturn Flow,通过该函数生成了其他值或者被杀死,那么置为有关联的,否则视为无关联传播.

2) 对于 Call Flow,如果存在被调用函数(callee)且当前数据流值可以通过被调用函数生成相关形参,那么这个变量是有关联的传播.

3) 对于 Return Flow,如果数据流值可以通过退出语句生成调用者(caller)的相关返回值,那么将设置其为有关联的传播.

评测的实验机器配置是:64 核 Intel® Xeon® CPU E7-4809(2.0 GHz)和 128 GB RAM,为每一个 Java 虚拟机分配 32 GB 的内存.配置选项选择打开流敏感性并且设置访问路径的长度为 5.具体的评测用例是在安智网[23]随机下载的 25 个 Android 应用程序.由于某些软件是不存在恶意行为的且对计算规模较小的程序评测没有意义,这里去除掉了没有产生泄露结果或分析结果过快(小于 10 s)的 6 个应用和 3 个运行超出内存的应用,最终评测用例集合包含 16 个应用程序,它们的基本信息如表 1 所示,本文后续评测用例所用标号是与该表标号一一对应的.随后使用 FlowDroid 对其运行 3 次检测,提取结果并求平均值,其最终的统计运行结果如表 2 所示.

Table 1 The Information of Evaluation Dataset

表 1 评测用例基本信息

No.	Application Name	Version	App Size/MB
1	Alarm/Clock	5.3.7	2.3
2	NingBoBus	1.0.1	3.8
3	Voicechanger	9.8.0	8.5
4	WeiboContact	2.0.2	5.4
5	RootVerifier	1.0	0.2
6	APVPDFViewer	0.4.7	3.3
7	Dingshidaren	2.1.1	1.7
8	LCDDisplay	1.0	5.5
9	FeeQuery	1.0	0.7
10	HuozhongContacts	3.0.0	1.5
11	MyAccount	0.8.7	1.5
12	SuningLottery	1.9.0	2
13	YoushengAlbum	1.0	5.7
14	iWencai	2.42.01	4.1
15	Gushiheima	2.2	1.5
16	Caike	2.0.23	4

表 2 中第 2 列(Running Time)和第 3 列(Memory Size)是程序的运行时间和内存开销.其中编号 11 的程序,尽管程序的规模只有 1.5 MB,但是程序运行时间却长达 1 922 s,这正是由于流敏感和上下文敏感的较高时间复杂度导致的.第 5 列(♯IFDS)是总的 IFDS 计算边个数,而第 6 列是无关联的 IFDS 计算边(♯UR-IFDS)个数,其中总 IFDS 计算边个数平均高达 1.1×10^7,而无关联 IFDS 计算边个数平均为 1.01×10^7.可见无关联的计算边占总计算边的比例是很高的,其中最低比例是 79.1%,最高比例为 92.9%,平均比例为 85.2%,这也充分验证了前面的分析.所以得出结论是:由于域敏感和流敏感的特性使得 IFDS 算法的摘要优化失效,当前无关联的计算占总计算的比例较高且规模较大.本文的研究目标正是尝试将数据流值直接传递到其使用点的位置,跨过这些无关联的传播(稀疏的方式),来消除这种无关联的计算.

Table 2 The Results on Evaluation of FlowDroid

表 2 使用 FlowDroid 对数据集进行评测结果

No.	Running Time/s	Memory Size/GB	♯RES	♯IFDS /10^6	♯UR-IFDS /10^6	PP /%
1	73.4	3.3	41	3.8	3.5	92.9
2	672	15.8	27	17.1	14.6	85.3
3	30.8	3.8	32	1.0	0.9	89.8
4	491.7	7.2	18	14.1	11.3	80.5
5	245	8.8	111	1.1	0.91	79.8
6	172	5.1	46	7.9	6.9	87.1
7	253	11.6	99	11.0	9.5	86.0
8	760	15.9	19	22.2	17.5	79.1
9	93.3	7.3	37	3.9	3.2	83.7
10	855	12.6	46	37.1	33.2	89.4
11	1922	13.1	184	36.1	29.3	81.2
12	246	8.1	33	7.5	6.3	84.1
13	43.2	2.6	38	1.2	1.1	90.1
14	109	4.6	14	3.1	2.6	86.0
15	104	5.7	28	2.7	2.2	83.5
16	419	6.2	39	11.0	9.3	84.4

为了解决该问题,本文的方法将 DEF-USE 链的思想引入到 IFDS 计算框架中,在数据流传播时,直接将数据流值从其产生的位置传递到其真正使用点.

3 稀疏的 IFDS 框架

提供稀疏的污点分析的前提正是提供稀疏的数据流分析框架.本节将介绍如何将经典的数据流分析框架 IFDS 扩展成稀疏的形式.本文的方法同样

是尝试使用 DEF-USE 链来代替 CFG 进行数据流值传播.然而与传统方法不同的是:为了支持域敏感和流敏感,本文使用了一种特殊的数据结构来存储变量的使用点,并提供了快速的构建算法(3.3 节和 3.4 节介绍),最后我们在 3.5 节修改了原 IFDS 的传播函数,将得到的数据流值直接传播到其使用点.

3.1　指令定义

首先定义本框架中用到的指令规则,将程序中所有语句转化为 3 地址的形式(Jimple 中间表示形式),保证程序中只存在 6 种与数据流传播相关的指令.为了方便后续规则的表示,这里将赋值语句[COPY]根据目标为左值或者右值的情况分成[READ]和[WRITE]形式.对于堆变量的域访问,如形如 $v.f$ 变量的访问,我们命名其堆所在对象为基对象(base),即变量 v,f 则是基对象中的域(field).对堆对象域的读和写被定义为[LOAD]和[STORE]规则.如表 3 所示为所有的 6 种指令规则.

Table 3　The Rules of the Instruction

表 3　指令规则

Rule	Instruction
ALLOC	$v=\text{new } V()$
READ	$w=v$
WRITE	$v=w$
LOAD	$w=v.f$
STORE	$v.f=w$
CALL	$r=c.m(v)$

3.2　变量使用点索引

为了维护域敏感和流敏感性,这里使用一种特殊的数据结构来表示变量的 DEF-USE 关系.

定义 1. 变量使用点索引.变量使用点索引是一个表(map)结构,它的键值(key)是变量的域,它的值(value)是该变量对应域的所有使用点集合.

这里使用一个特殊域 null 来表示变量中域为空的情况,使用函数 $setUses[f]=[stmt]$ 表示将语句[$stmt$]加入到变量的域 f 的使用点集合中,使用 $v.getUseStmts(f)$ 来表示获得的 f 域对应的使用点集合,使用 $v.getUseStmts(*)$ 表示获得 v 的索引表中所有域对应的使用点集合.

3.3　自治数据流图 ADFG

定义 2. 自治数据流图(autonomous data flow graph, ADFG).自治数据流图是一个有向图 $G=\langle N,E\rangle$,其中节点 N 表示程序中的变量且所有变量的基对象相同,边 E 为表示节点所在语句之间的控制流.

如图 4(b)正是基对象为 a 的自治数据流图,对于基对象 a 的所有相关变量均在该图中,且这些变量所在的语句(图 4(b)用行号表示)之间的控制流是节点之间相连的边.ADFG 的优势就是图中变量的基对象相同且保存了完整的控制流信息,之后对每一个独立的变量计算其 DEF-USE(变量使用点索引)时,即可在 ADFG 中进行,这既提高了分析效率,又保证了流敏感.

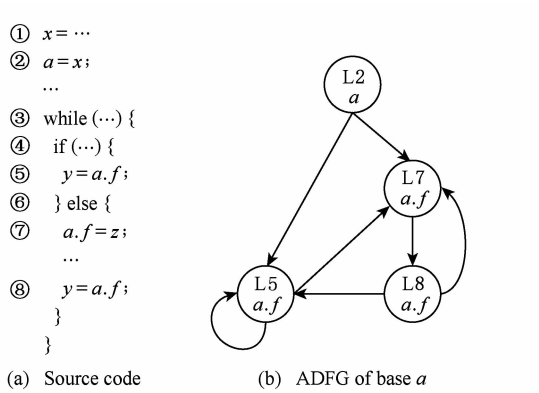


Fig. 4　An example of ADFG

图 4　自治数据流图(基对象为 a)示例

基于 ADFG 的定义,我们提供了一种快速的构建 ADFG 的算法.该算法将按照变量的基对象进行分区,然后对于每一个分区中的变量分别计算其独立的数据流图.具体算法如算法 1 所示:

算法 1. 构建自治数据流图算法.

输入:程序的 $allCFGs$;

输出:程序的 $allADFGs$.

① $collectVariables(\cdots)$;

② $partitionByBasicBlock(\cdots)$;

③ $buildADFG(\cdots)$.

子过程 1. $collectVariables$.

输入:程序的 $allCFGs$.

① for each $oneCFG$ in $allCFGs$ do

/* 对于每一个方法的 CFG 进行遍历 */

②　for each var in each $stmt$ of $oneCFG$ do

③　$varInfoSetMap[base]\leftarrow[var,bb,idx]$;

/* 将程序中所有变量按照其基对象进行分区,存储到 $varInfoSetMap$ Hash 表中 */

④　end for

⑤ end for

子过程 2. $partitionByBasicBlock$.

输入:$varInfoSetMap$.

① for each $varInfoSet$ in $varInfoSetMap$ do

② for each *var* in *varInfoSet* do
③ *bb*=*var*.*getBasicBlock*();
/*获取当前变量的基本块*/
④ *bbToVarInfoMap*[*bb*]←*var*;
/*将变量按照其基本块进行分区,存储到 *bbToVarInfoMap* Hash 表中*/
⑤ end for
⑥ end for

子过程 3. *buildADFG*.

输入:*bbToVarInfoMap*.

① for each *bb* in *bbToVarInfoMap* do
② *ret*=*innerBBSolver*(*bb*,*bb*.*varSet*);
/* *innerBBSolver* 将构建基本块内部节点的控制流关系,并返回其头尾节点〈*head*,*tail*〉*/
③ varinfo *tail*=*ret*.*getTail*;
④ for *next* in *bb*.*getSuccs* do
⑤ *traverseEachBB*(*next*,*tail*);
⑥ end for
⑦ end for

子过程 4. *traverseEachBB*.

输入:后继基本块 *next*,和前驱基本块的尾节点 *preTail*.

① *ret*=*innerBBSolver*(*next*,*next*.*varSet*);
② *varinfo head*=*ret*.*getHead*;
③ *preTail*.*setSucc*(*head*);
④ for *next* in *bb*.*getSuccs* do
⑤ *traverseEachBB*(*next*,*ret*.*getTail*);
⑥ end for

函数 *collectVariables* 将变量按照其基对象进行分区,存储到 Hash 表 *varInfoSetMap* 中,对于一个变量,这里存储的信息包括变量的值 *var* 和其基本块信息 *bb* 以及其所在基本块内部的索引号 *idx*. 函数 *partitionByBasicBlock* 再通过它所在的基本块分区并存储到 Hash 表 *bbToVarInfoMap* 中. 之后,函数 *buildADFG* 对基对象相同的变量集合按每一个基本块进行遍历,在基本块内部,*buildADFG* 调用 *innerBBSolver* 方法顺序的将后继语句加入到前驱语句的后继集合中. 在基本块之间,*buildADFG* 通过 *innerBBSolver* 的返回值得到每一个基本块的头节点和尾节点,为上一个基本块的尾节点设置下一个基本块的头节点为其后继.

至此,ADFG 上每一个变量都包含其控制流中的后继信息,且不包含其他基对象的变量. 函数 *collectVariables* 和 *partitionByBasicBlock* 线性地扫描程序中的变量,所以其算法时间复杂度是线性的. 而 *buildADFG* 将对每一个基本块进行遍历,在基本块内部则顺序访问每一个变量,所以 *buildADFG* 算法的时间复杂度也是线性的,整个构建 ADFG 算法的时间复杂度是线性的.

3.4 变量使用点索引的计算

在 ADFG 上,所有变量的基对象均相同且仍包含其控制流信息,整个 CFG 按照基对象的不同被分成多个 ADFG. 分析器将分别遍历每一个 ADFG,使用数据流分析的方法计算其变量的使用位置,将其存储到使用点索引中. 如表 4 所示为计算使用点索引的数据流转移函数,图 5 为对应示例. 我们将在 4.2 节讨论如何使用该索引为污点分析的访问路径生成具体的使用点以及对使用点集合进行剪枝. 该分析使用了传统的工作集(worklist)算法,初始化集合是所有语句中的变量 $v/v.f$ 与当前语句[*stmt*](即变量的定义语句)组成的对〈[*stmt*],$v/v.f$〉.

Table 4 Data Flow Transfer Function for Computing Variable Using Map

表 4 计算变量使用点索引的数据流转移方程

Rule	Instruction	DEF Value	Set USE Rule	IsKill
ALLOC	v=new V()	$v/v.f$		True
WRITE	$v=v'$	$v/v.f$		True
READ	$v'=v$	v	*setUses*[null]=[*stmt*]	False
		$v.f$	*setUses*[f]=[*stmt*]	False
LOAD	$v'=v.f$	$v/v.f$	*setUses*[f]=[*stmt*]	False
STORE	$v.f=v'$	v	*setUses*[null]=[*stmt*]	True
		$v.f$		True
CALL	$v.m(v')$/	v	*setUses*[null]=[*stmt*]	True
	$v'.m(v)$	$v.f$	*setUses*[f]=[*stmt*]	True

表 4 中,第 3 列定义值(DEF value)是转移函数的输入,即变量的定义位置传递过来的数据流值,转移函数的功能是决定是否将当前语句设置为该值的使用点. 根据 3.1 节中的指令定义,定义值只能为 v 或者 $v.f$ 的形式,例如图 5 中被方格括住的变量. 第 4 列 Set USE Rule 表示对定义值设置使用点索引的规则. 为了区分域敏感特性,需要为不同的域存储不同的使用点. 如 3.2 节定义,使用 *setUses*[f]=[Stmt]来表示将语句[*stmt*]加入到变量 v 的域 f 的使用点集合中,而 *setUses*[null]表示定义值域是空的情况. 第 5 列 IsKill 列表示定义值是否被杀死(True)或者继续寻找其使用位置(False).

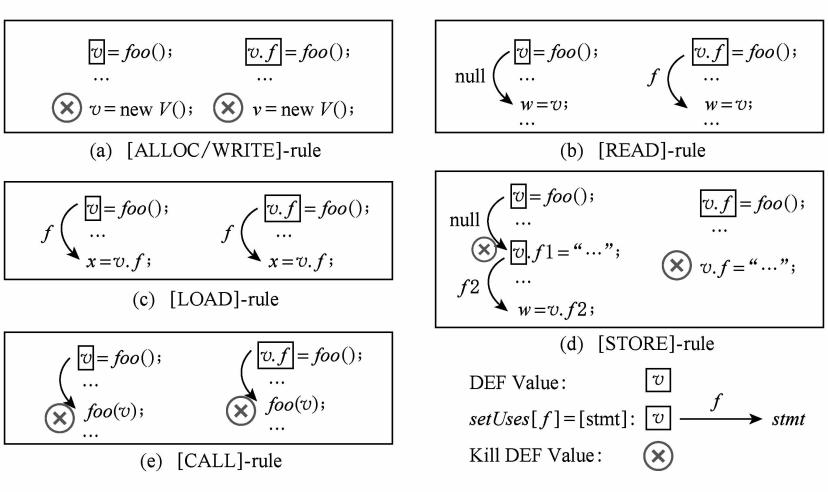


Fig. 5　Examples of variable using map computing

图 5　计算变量使用点索引示例

对于[ALLOC]和[WRITE]规则,如图 5(a)中例子所示,无论定义值是 v 还是 $v.f$,都会将对 v 变量进行重新赋值,传来的定义值不再有效.因此设置 IsKill 为 True 且不会产生使用关系.

对于[READ]规则,如果定义值是 v,如图 5(b)左侧程序,说明任何的形如 $v.*$的访问路径都会传递到使用点,所以设置空域的使用点集合(setUses[null]=[*stmt*]).如果定义值是 $v.f$,如图 5(b)右侧程序,说明只有形如 $v.f.*$的访问路径会传递到使用点,则设置域 f 的使用点集合(*setUses*[f]=[*stmt*]).

对于[LOAD]规则,说明访问路径第 1 个域满足和 f 相等时才会被使用.所以无论访问路径是形如 $v.*$或 $v.f.*$的输入,都将设置域 f 的使用点集合(*setUses*[f]=[*stmt*]).

对于[STORE]规则,如果定义值是 $v.f$,如图 5(d)规则右侧程序,同[WRITE]规则,传来的定义值不再有效.如果定义值是 v,说明形如 $v.*$的访问路径可能传播到该语句,且需要杀死其中第 1 个域为 f 的值.由于当前访问路径是无法表示形如 $v.\{*-f\}$ 的数据流值,为了不丢失信息,这里使用一种延迟处理的方法:首先将该语句加入到定义值空域的使用点集合中(*setUses*[null]=[*stmt*]),然后保守地认为该定义值全部失效(将其杀死),最后在初始化集合中增加一个虚拟的〈[*stmt*],v〉的定义值,增加新的虚拟定义值的目的是为了继续传播除 f 域之外的值.以图 5(d)左侧程序为例,假设函数 *foo* 的返回值得到的访问路径是 $v.f2$,$v.f2$ 根据空域的使用点可以传播到[$v.f1$="…";]处,由于域不同,分析规则不会杀死该值,则会继续启动对 $v.f2$ 使用点的查询,此时将使用虚拟定义值 v 进行索引查询,即可找到域 $f2$ 的使用点[$w=v.f2$;].如果函数 *foo* 的返回值是 $v.f1$ 的话,该值传播到[$v.f1$="…"]会根据分析规则被杀死.这种延迟处理的方式既不会损失精度又可以正确地进行更新.

对于[CALL]规则,如图 5(e)所示规则.这里设置使用点的规则同[WRITE],这是因为在 IFDS 框架中函数的参数是否能够通过函数调用继续向下传递是由 Call 和 CallToReturn 规则决定的,所以这里直接将定义值置为失效,待 Call 和 CallToReturn 产生其返回值再进一步查找与传播.

3.5　传播函数

框架还需修改原 IFDS 框架的传播函数(文献[20]中算法行 35 的函数 *propagate*),这里为传播函数增加一个包装函数 *propagateWrapper*,如算法 2 所示.

算法 2. 稀疏的 IFDS 传播函数 *propagateWrapper*.

输入:定义语句 *src*、目标数据流值 *dtarget*;

输出:目标语句 *use*、目标数据流值 *dtarget*.

① *useSet*=*defValue*.*getUseStmts*();

② for each *useStmt* in *useSet* do:

③　　*propagate*(*useStmt*,*dtarget*)

④ end for

包装函数将通过当前变量及其使用点索引获得其使用点集合,然后将数据流值直接传递到其使用点.本算法不会影响 IDFS 产生新的数据流值的计算,而只是修改其传播的目的语句,即达到稀疏传播而又不影响正确性.例如图 6 中 a 在行②处将仍然

会产生 b 值,但会将 b 值直接传播到其使用位置,即行③.

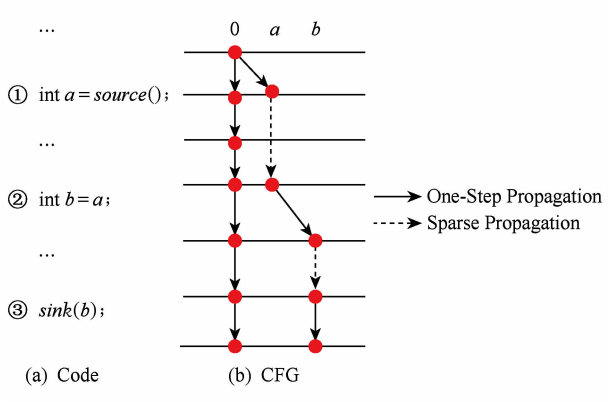


Fig. 6 An example of sparse IFDS framework

图 6 稀疏的 IFDS 框架示例

4 基于稀疏框架的污点分析

4.1 总体框架

基于第 3 节的稀疏框架,本节设计实现了与其适配的污点分析技术,图 7 所示为其整体框架. 相对于传统的污点分析(FlowDroid),该框架增加一个预先分析(preAnalysis)的过程,即构建 ADFG 和计算变量使用点索引的过程. 与 FlowDroid 一样,分析的第 1 部分仍是待分析程序的 ICFG(图 7 中 Build ICFG 过程),以此作为预先分析的输入. 预先分析过程生成的使用点索引作为污点传播前向分析和后向分析(别名分析)的输入,最后是输出结果部分.

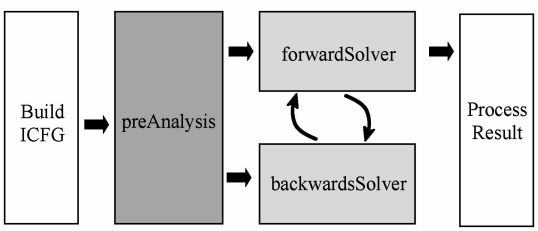


Fig. 7 The overview of the our taint analysis framework

图 7 基于稀疏的污点分析总体框架

4.2 污点传播中使用点的维护与剪枝

由于在 3.4 节中提供了面向域敏感和流敏感的变量使用点索引,本节将根据具体的输入使用这些索引产生污点分析的使用点,并达到对使用点集合剪枝的效果. 与 FlowDroid 类似,我们同样使用访问路径来表示数据流值. 具体如算法 3 所示:

算法 3. 污点传播使用点集合剪枝算法.

输入:当前语句点 $stmt$、数据流值访问路径 $apath$;输出:数据流值的使用点集合 $useSet$.

① if $apath.getFirstField \neq$ null do

/* 形如 $v.f.*$ 的访问路径输入 */

② $useSet = \bigcup getUseStmts(apath.getFirstField) + getUseStmts(\text{null})$;

③ else /* 形如 $v.*$ 的访问路径输入 */

④ if $isStore(stmt)$ do:

/* 形如 $v.f=$"…"的语句 */

⑤ $useSet = \bigcup(getUseStmts(*) - getUseStmts(f))$;

/* 从索引中所有域的使用点集合中减去域 f 的使用点 */

⑥ else

⑦ $useSet \bigcup = getUseStmts(*)$;

⑧ end if

⑨ end if

算法 3 的输入是当前语句 $stmt$ 和产生的数据流值的访问路径,这里用 $apath$ 表示. 如果当前 $apath$ 是形如 $v.f.*$的形式,即访问路径的第一个域是 f,那么使用域 null 和 f 产生其使用点集合,即算法 3 中行②. 这种方法可以过滤掉和 f 域不同的且不为空的域的使用点. 如果 $apath$ 是形如 $v.*$的形式,说明当前 v 的子域不确定,因此需要传播到其所有域的使用点,即算法 3 的行⑦. 一种特殊的情况,如果语句是[STORE]形式,根据表 4 的[STORE]规则,当前存在一个 v 的虚拟定义变量的使用点索引,那么查找并使用该索引:除了需要产生所有域的使用点之外,还需要从中减去当前语句被赋值域的使用点,即算法 3 中行⑤,这是由于当前[STORE]已经将该域更新为其他值,无需再为该域计算使用点. 最终算法的输出是 $useSet$,即根据当前访问路径得到的使用点集合.

本文的方法虽然没有改变算法的复杂度(IFDS 算法的时间复杂度为 $O(ED^3)$,空间复杂度为 $O(ED^2)$,其中 E 是当前控制流图中的边个数,D 是数据流域的大小). 但是由于我们的方法对当前控制流图转化成稀疏的形式(自治数据流图),所以 E 的值将会明显的缩小,此外 D 的值也由于无关联传播被消除而变小,这也是本文方法能够提高效率的原因.

4.3 面向别名传播的变量使用点索引

在反向的 IFDS 计算中,后向的别名间污点传播使用的变量使用点索引与前向传播稍有不同:由于需要在使用点计算其别名的污点,所以同样将语句加入其使用点集合中. 表 5 是对应的转移函数,其中与

正向的转移函数不同的是:[ALLOC]和[WRITE]和[LOAD]规则,即需要保留它们的使用点.

Table 5　Data Flow Transfer Function for Computing Backwards Variable Using Map

表 5　后向分析中计算变量使用点索引的数据流转移方程

Rule	Instruction	DEF Value	Set USE Rule	IsKill
ALLOC	v=new V()	v	$setUses$[null]=[$stmt$]	True
		$v.f$	$setUses$[f]=[$stmt$]	True
WRITE	$v=v'$	v	$setUses$[null]=[$stmt$]	True
		$v.f$	$setUses$[f]=[$stmt$]	True
READ	$v'=v$	v	$setUses$[null]=[$stmt$]	False
		$v.f$	$setUses$[f]=[$stmt$]	False
LOAD	$v'=v.f$	$v/v.f$	$setUses$[f]=[$stmt$]	False
STORE	$v.f=v'$	v	$setUses$[null]=[$stmt$]	True
		$v.f$	$setUses$[f]=[$stmt$]	True
CALL	$v.m(v')$/ $v'.m(v)$	v	$setUses$[null]=[$stmt$]	True
		$v.f$	$setUses$[f]=[$stmt$]	True

4.4　流敏感的别名传播维护

为了维护流敏感的别名传播,基于 IFDS 的污点分析引入了激活点的概念,稀疏的方法同样需要考虑这个问题,由于稀疏的方法不能沿着 CFG 上的每一条语句进行传播,这给维护激活点带来了挑战.

为了解决该问题,我们利用了可能可达性的概念,所谓的可能可达(may reach)是一种偏序关系 $\mapsto$:如果程序语句 S_1 和 S_2 存在偏序关系 $S_1 \mapsto S_2$,则说明可能存在一条执行路径使得变量从 S_1 到 EXIT 流经 S_2.相反,如果 $S_1 \mapsto S_2$ 不被满足,则说明一定不存在一条执行路径从 S_1 到 EXIT 流经 S_2.

那么,定义当前污点变量的定义点是 $defStmt$,污点变量的激活点是 $activeStmt$,污点变量的使用点是 $useStmt$,如图 8 中所示.则污点值被激活的必要条件是:

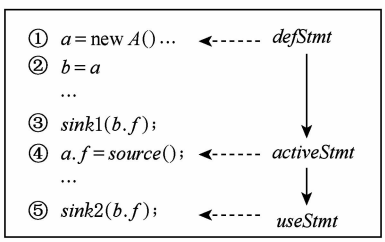


Fig. 8　An example of active stmt maintenance

图 8　FlowDroid 的激活点维护示例

约束 1. $defStmt \mapsto activeStmt$ 且 $activeStmt \mapsto useStmt$.

证明. 必要性证明.如果 $defStmt \mapsto activeStmt$ 不满足,那么污点变量不会流经激活点,污点一定不会被激活.如果 $activeStmt \mapsto useStmt$ 不满足,则说明到达 $activeStmt$ 的值一定不会到达 $useStmt$,也就不会存在被激活的变量到达 $useStmt$.　证毕.

至于偏序关系 $\mapsto$ 是可以在预处理阶段计算出的哈希映射进行查询的.比较过程内 2 点 S_1 和 S_2 的偏序关系 $\mapsto$ 的方法:如果它们在一个基本块中且该基本块不在循环中,那么直接比较其基本块内部索引号.否则比较基本块之间的可能可达关系.而基本块的可能可达关系可以在预处理中很快的得到一个 Hash 映射.所以整个偏序关系比较的时间复杂度是 $O(1)$的.

不过满足约束 1 的情况并不是满足污点激活的充分条件,这是因为会存在数据流值被杀死的情况.为了保证精度,我们为满足约束 1 的数据流值再启动一个独立的更新(update)分析,更新分析是沿着 CFG 进行的,直接使用传统数据流方程求解(与 FlowDroid 方法一致).如果数据流值既满足约束 1 又不被更新阶段所杀死,则说明是被激活的.

4.5　剪枝优化讨论

至此,本文的算法既保证了稀疏性又保证相比原 FlowDroid 不会损失任何的精确度.在具体实现中,基于变量使用点索引的剪枝算法还能够消除 FlowDroid 的误报,提高检测的精度.具体分析:

如果当前分析器不确定具体的访问路径的子域时,FlowDroid 会用一种保守的方式为其所有的子域均进行污点标记,即使用形如 $v.*$的访问路径表示 v 以及 v 的所有子域均为污点标记.这里命名该处理方式为保守子域处理.保守子域处理方式虽然不会产生漏报,但是显然是会产生误报的.除了分析器不知道具体是哪个子域被标记的之外,还会存在分析器无法对子域进行正确更新的情况.如图 9 所示,行①的 $source$ 正是产生了 $v.*$的数据流值,是一种保守子域的处理.当 $v.*$传播到行②时,由于不能

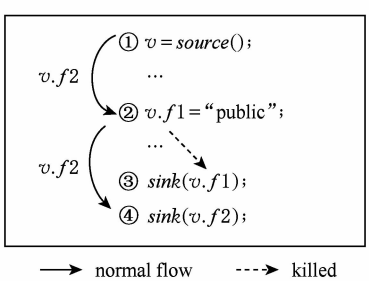


Fig. 9　An example of FlowDroid conservative sub-field process

图 9　FlowDroid 的保守子域处理示例

确定是哪个子域被更新的(当前数据流值无法表示形如 $v.\{*-f\}$),此时只能保守地不更新而继续传播 $v.*$值.因此 $v.*$到达行③时则会报告出泄露.而实际上行③的 $v.f1$ 值已经在行②被设置成非敏感的数据,所以该报告是误报.

而这种误报是可以利用本文的方法进行消除的.具体分析:对于行②的左值 $v.f1$,根据 3.4 节算法计算得到的使用点索引是($setUses[f1]=[L3]$),而当数据流值 $v.*$到达行②时,分析器执行 4.2 节算法 3 的行⑤分支,即使用规则[$useStmt = \bigcup getUseStmt(*) - getUseStmt(f)$]来计算其使用点集合.此时,Hash 表中 $f1$ 域对应的使用点将会从总使用集合中删除.因此,此时使用点集合中只包含行④而不会包含行③,继而误报被消除.

另外,据实验观察,这种保守子域的处理方式存在的情况是比较多的,这是因为不但直接引入的变量会产生保守子域处理,如果访问路径的长度超过预先设置的值时,FlowDroid 会将该访问路径进行裁剪(cut off),裁剪后的变量也是一种保守子域处理的方式.例如预先设置的访问路径阈值为 3 时,当生成的访问路径为 $v.f.g.h$ 时,则会保守地转化成 $v.f.g.*$的情况.如果程序中存在循环的情况(使得访问路径长度不断增加),这种裁剪而导致的保守子域处理的情况尤为突出,我们在实验评测部分统计验证了经过裁剪的数据流值占总数据流值所占比例是较大的(5.4 节).

5 实现与实验

基于本文方法,我们在 FlowDroid 和 Soot 的基础上实现了原型系统 FlowDroidSP,其中重用了 FlowDroid 创建 ICFG、访问路径、提取结果部分以及计算污点的转移函数(4 种 IFDS 转移函数).为了保证函数调用和函数返回的过程能够找到参数和返回值对应的使用点,这里在创建 ADFG 时,为函数参数和返回值增加了虚拟的定义值.最后,我们使用了 100 多个单元测试用例测试其健壮性.

为了验证本文方法的精确度,FlowDroidSP 实现了 2 种模式:剪枝模式和非剪枝模式.剪枝模式(简称 FlowDroidSP-P)正是利用了具有域敏感和流敏感的变量使用点索引进行剪枝的实现版本.如在 4.5 节所讨论,由于 FlowDroid 的保守子域处理,导致剪枝的模式会对 FlowDroid 的精度进行提升,然而这不利于验证其是否会损失精度.因此,我们实现了非剪枝的模式(简称 FlowDroidSP-NP)专用于验证精确度.在非剪枝模式的实现中,我们将 4.2 节的算法 3 中所有分支都执行[$useSet = \bigcup getUseStmt(*)$;]规则,即使用其所有域的使用点,不进行剪枝处理.非剪枝算法理论上是和 FlowDroid 原算法精度完全一致的.

所有实验所使用的机器的配置是:64 核 Intel® Xeon® CPU E7-4809(2.0 GHz)和 128 GB RAM,为每一个 Java 虚拟机分配 32 GB 的内存.同 FlowDroid 一样,设置访问路径的长度为 5.这里所使用的测试集合同第 2 节中的 16 个应用程序.使用 FlowDroidSP 对每一个测试用例独立运行 3 次,提取结果并求平均值.

5.1 评测问题

本节我们将试图利用实验验证 3 个评测问题:

1) 相比 FlowDroid,本文的方法是否会损失精度.

2) 预处理阶段和独立更新算法的开销是多少.

3) FlowDroidSP 相比 FlowDroid 的性能提升效果是多少.

5.2 问题 1:精确度验证

验证精确度需要使用非剪枝版本.这里的评价标准是产生的结果数目是相同的.表 6 是 FlowDroid 和 FlowDroidSP 的对比运行结果.表中F-Res 列为原 FlowDroid 产生的结果个数而 NP-Res 列为非剪枝模式 FlowDroidSP 产生的结果个数,可见非剪枝版本的 FlowDroidSP 与 FlowDroid 产生的结果个数是完全一致的,总共产生 341 个泄露结果.这进一步验证了本文的方法相比 FlowDroid 方法是没有精度损失的.

5.3 问题 2:预处理和更新阶段开销

如表 6 所示,表中 PRE-T 列是 FlowDroidSP 非剪枝模式下的预处理过程的运行时间,其中标号 16 的程序最高为 2.6 s,标号 9 的程序最低为 0.2 s,平均为 0.9 s.对于内存开销(PRE-Mem 列),其中标号 16 开销最高为 429 MB,标号 9 开销最低为 11 MB,平均为 126 MB.预处理的平均消耗时间占平均总分析时间的百分比小于 1%,而内存消耗平均占比为 4%.由此可见,预处理的时间和空间开销是很低的.时间上,相对于分析的平均时间 91.1 s,这种开销几乎可以忽略不计.预处理开销低的原因是由于线性复杂度的 ADFG 构建算法和在此基础上的过程内的 DEF-USE 分析.同样,对于独立更新阶段的时间消耗(表 6 中 SP-T 列),最高时间消耗是 4.1 s,最低是 0.07 s,平均时间消耗是 1.3 s.可见,虽然独立更

新阶段算法不是线性复杂度的,但是由于我们首先使用了约束 1 进行过滤,导致需要独立更新的数据值已经很少了.为此,本实验单独的统计了独立更新的值(Active-SP)与需要激活但不满足约束 1 的值(Active)之间所占百分比的对比,结果如图 11 所示.可见,独立更新部分所占比例是较小的(平均 4.2%).

Table 6　The Results on Evaluation of FlowDroidSP

表 6　FlowDroidSP 评测数据

No.	F-Time /s	F-Mem /GB	F-Res	NP-Time /s	NP-Mem /GB	NP-Res	PRE-T /s	PRE-Mem /MB	SP-T /s	P-Time /s	P-Mem /GB	P-Res
1	73.4	3.3	41	16	0.8	41	1.5	230	0.7	15	2.3	41
2	672	15.8	27	172	6.3	27	0.6	79	3.2	50	3.7	9
3	30.8	3.8	32	6.4	1.3	32	1	123	0.07	4.6	0.8	32
4	491.7	7.2	18	73.2	1.8	18	0.5	39	0.47	11.3	1.4	18
5	245	8.8	111	69.3	3.2	111	0.3	37	2	11	1	21
6	172	5.1	46	46	1.9	46	0.2	17	0.2	21	1.1	46
7	253	11.6	99	73	3.5	99	1.3	190	2	19	2.2	99
8	760	15.9	19	153	4.7	19	0.6	71	2.2	14	0.7	16
9	93.3	7.3	37	22	1.1	37	0.2	11	0.3	22	1	37
10	855	12.6	46	256	5.4	46	1	141	3	146	2.7	40
11	1922	13.1	184	400	12	184	1.7	306	4	60	5.3	180
12	246	8.1	33	43	3.3	33	0.8	129	0.9	19	1	31
13	43.2	2.6	38	5	0.8	38	0.6	80	0.1	3.3	0.6	26
14	109	4.6	14	18.9	1.1	14	0.5	64	0.18	7.4	0.9	13
15	104	5.7	28	22.9	2.4	28	0.6	78	0.2	3.1	0.4	14
16	419	6.2	39	82	4.4	39	2.6	429	1.9	57	3.5	35

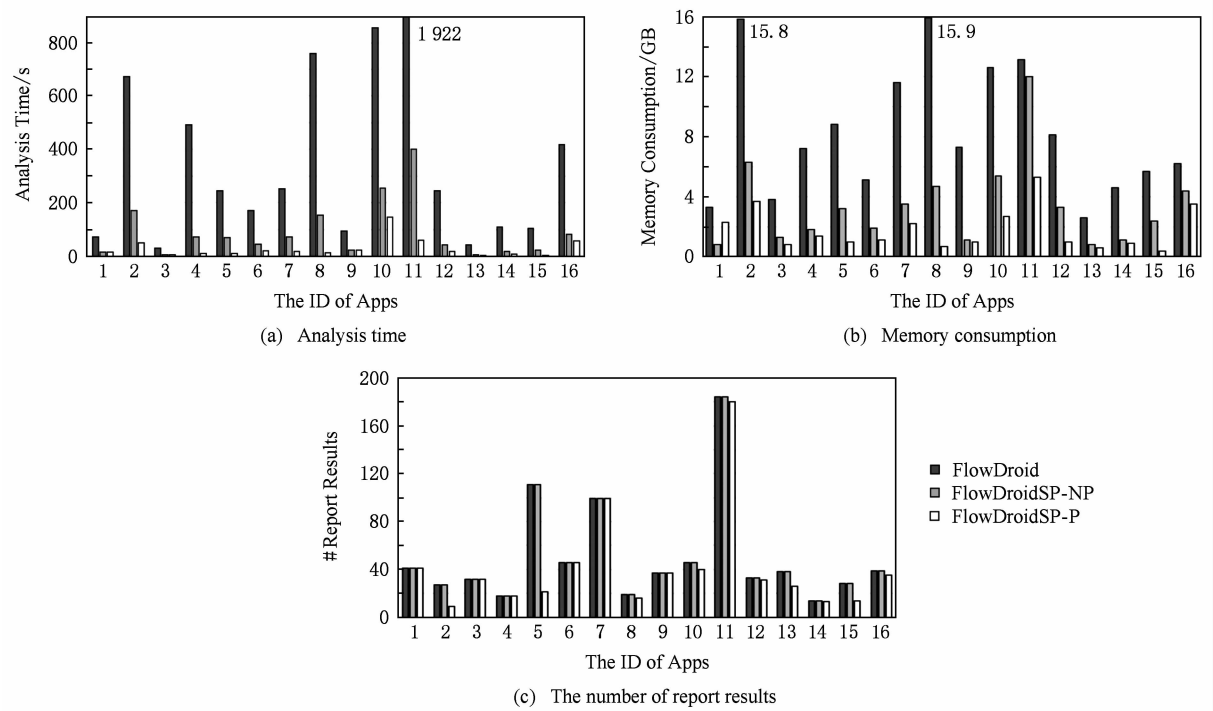


Fig. 10　Comparison on FlowDroid and FlowDoidSP beyond different mode

图 10　FlowDroid 和 FlowDroidSP 不同模式下的结果对比

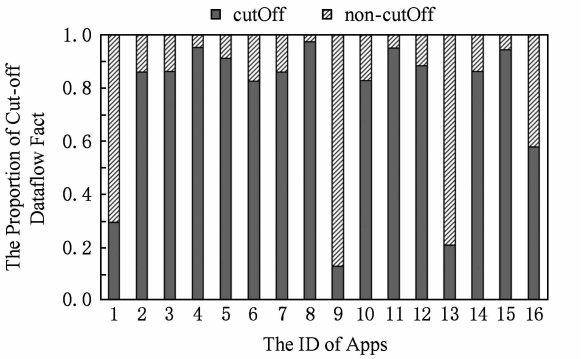


Fig. 11 Proportion diagram of independent updating

图 11 独立更新计算值占可激活值的比例图

5.4 问题 3:性能提升效果

本节将对打开剪枝优化进行评测,如表 6 中列 P-Time,P-Mem,P-Res 为剪枝模式下 FlowDroidSP 的运行时间、内存消耗和产生泄露的结果数量. 最终图 10 给出了 3 种评测模式(原 FlowDroid,非剪枝模式,剪枝模式的 FlowDroidSP)的在运行时间,内存消耗和产生结果数量的对比柱状图. 在运行时间上,非剪枝版本平均加速比为 4.8 倍,在剪枝版本中,平均加速比为 18.1 倍. 非剪枝版本的时间消耗加速比波动范围正常,最低 3.3 倍,最高 8.6 倍. 而剪枝版本的加速比波动范围是非常大的,最低提升 4.2 倍,最高提升高达 54 倍,这是一方面由于剪枝方法本质决定,如果在程序传播初始阶段就能够有效地杀死不该传播的数据流值,会避免后续污点大量的扩散,另一方面我们猜测剪枝方法的加速比也和程序中保守子域的规模有关,保守子域处理的情况越多,可能剪枝的情况就会越多. 为此,我们统计了具有被裁剪标识(当前访问路径被裁剪后会得到一个裁剪标识)的数据流的数量,如图 12 所示为该数目占当前所有数据流值的比例. 对于编号 1、编号 9、编号 13 的例子,其所占比例为 25%,12.9%,20.9%. 所以剪枝优化对其提升比例不高(分别为 4.8,4.2,13.9 倍). 而对于其他例子中该规模是相对较高,例如编号 8 的比例高达 97.4%,所以剪枝算法对其优化的加速比可以高达 54 倍. 对于内存消耗上,在非剪枝模式下,FlowDroidSP 降低内存消耗平均为 61.5%. 在剪枝模式下,FlowDroidSP 降低内存消耗平均为 76.1%. 另外对于产生泄露结果数目的影响,在剪枝模式下,FlowDroidSP 是有一定精确度提升的,一共消减了 154 个误报结果,占总数的 17.7%.

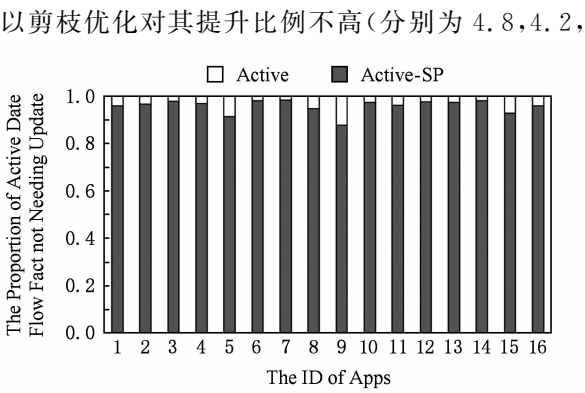


Fig. 12 Proportion diagram of cutting off date flow facts

图 12 具有裁剪标识的数据流占总数比例图

6 相关工作对比

数据流分析是静态程序分析的经典分析方法,IFDS 框架[20]率先将流敏感和上下文敏感的分析问题转化成图可达问题进行求解,提高了分析的效率,该算法的最坏时间复杂度是 $O(ED^3)$,其中 E 是当前控制流图中的边个数,D 是数据流域的大小. 文献[24]在 2010 年对 IFDS 进行了效率提升,它利用一种按需的方法对 ICFG 进行构建,为调用函数和被调用函数建立一个映射关系,当调用退出时只有存在于映射里面的调用者才会为其计算返回值. 后来,该方法被实现到开源工具 Heros[25] 中,FlowDroid 正是基于这个版本的 IFDS 进行实现的. FlowTwist[7] 利用了祖先和邻居的数据结构将数据值之间的传播关系进行连接,以保证来自不同源的数据流值可以进行合并,增加了分析的扩展性. 然而,当前没有工作探索对 IFDS 进行稀疏化的支持. 同样,本文的方法本质是修改 IFDS 的传播函数,因此也适用于上述 2 个工作.

针对 FlowDroid 进行利用以检测安全问题的工作有很多[4]. 例如,达姆施塔特大学的安全软件工程实验室围绕了 FlowDroid 构建了一整套的安全检测生态系统[26],这包括使用 Susi[27] 来提供污点分析所使用的源和汇聚点;使用 IccTA[28] 提供敏感信息通过组件间进行泄露的检测方法;使用 StubDroid[29] 利用 FlowDroid 对库函数进行摘要提取;Harvester[30] 利用 FlowDroid 提取关键路径信息,然后动态的验证这些信息是否是泄露等工作;静态污点分析的另一大挑战就是流敏感的别名分析,FlowDroid 的流敏感的别名分析最初思想来源于 Andromeda[31],其本质是基于 CFL-图可达问题的变体;Sridharan 等人[32] 尝试将对 Java 的别名分析转化为 CFL-图可达问题;随后又在文献[33]中提出了基于精简(refinement)的优化,然而,基于 CFL 的 Java 指针分析都是流不敏感的. 为了解决流敏感问题,FlowDroid 引

入了激活点的方法;Boomerang[34]是最近提出的一个面向 Java 的按需的别名分析工具,它也是基于 IFDS 的,可以支持流敏感和上下文敏感.同样 Boomerang 也会面临类似的效率问题,我们认为本文的方法同样适用于 Boomerang 来提高其分析效率.

稀疏程序优化方法是目前指针分析领域的重点优化方案.Hardekopf 和 Lin[10]在 2009 年首次将流敏感的指针分析应用到百万行代码量的程序分析中,提出了半稀疏的流敏感指针分析方法,该方法利用部分 SSA(partial SSA)使得本地变量的分析转移到该表示中,构建了 SEG(sparse evaluation graph)进行具体指针分析;随后 Hardekopf[11]又利用了流不敏感的分析做为预处理将其扩展成全稀疏的形式;Lhotak 和 Chung[13]的 SUPT 同样利用全稀疏的 SSA 表示并提供了强更新(strong update)的支持;Li 等人[14]将图可达的思想引入到值流图(value flow graph)中进行按需的流敏感的指针分析优化.然而,上述的工作都是针对于 C/C++语言进行开展的且这些分析往往需要一定的预处理(SSA 或者流不敏感的分析支持).

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[35]在硬件级别上扩展了寄存器的标志位来实现污点分析用来进行一些基础漏洞挖掘,如缓冲区溢出;TAJ[36]工具利用了混合切片的静态分析技术提供 Java Web 安全漏洞的检测;TaintDroid[37]是当前最流行的动态 Android 应用隐私泄露检测工具,它使用的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 系统本身带来一定开销,每次 Android 版本更新之后,TaintDroid 都需要重新进行定制.CHEX[38]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;DroidSafe[39]结合 Android Open Source Project 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.

7　总结与未来工作

本文针对检测移动应用隐私泄露问题的污点分析技术进行性能提升.基于稀疏优化的思想,本文将传统数据流分析框架扩展成稀疏的形式,即将数据流值直接传播到其使用点,避免其在定义点和使用点之间的无关联传播.我们提出了一种特殊的数据结构-变量使用点索引来存储流敏感和域敏感的 DEF-USE 关系,并提供快速的构建算法.以此结构为输入,提供了对应的污点分析技术.另外,本文还发现基于流敏感和域敏感的 DEF-USE 关系可以对污点分析中保守子域处理的情况进行有效的剪枝,可以同时提升其效率和精度.

最后本文实现了工具 FlowDroidSP 并进行评测.实验表明,FlowDroidSP 在提升性能的同时没有带来精度上的损失.在非剪枝模式下,可以带来时间加速比 4.8 倍,内存消耗降低 61.5%,在剪枝模式下,可以带来时间加速比 18.1 倍,内存消耗降低 76.1%,并且消减了 17.7%的误报结果.

由于 IFDS 框架分配率的限制,当前污点分析还不能支持别名之间的强更新,后续我们将探索如何利用预处理信息提供别名间的强更新.此外,根据 5.4 节的实验结果不难得出的结论是当前保守子域问题也是限制污点分析扩展性的瓶颈,尽管稀疏框架可以对其进行优化剪枝,但仍然不能完全消除其误报,未来对保守子域的精确计算也是一个值得研究的方向.

参　考　文　献

[1] Zhang Yuqing, Zhou Wei, Peng Anni. Survey of Internet of things security [J]. Journal of Computer Research and Development, 2017, 54(10): 2130-2143 (in Chinese)
(张玉清,周威,彭安妮. 物联网安全综述[J]. 计算机研究与发展, 2017, 54(10): 2130-2143)

[2] Livshits V, Lam M. Finding security vulnerabilities in Java applications with static analysis [C] //Proc of the 14th USENIX Security Symposium. Berkeley, CA: USENIX Association, 2006: 262-266

[3] Sabelfeld A, Myers A C. Language-based information-flow security [J]. IEEE Journal on Selected Areas in Communications, 2003, 21(1): 5-19

[4] Li Li, Bissyandé T F, Papadakis M, et al. Static analysis of android apps: A systematic literature review [J]. Information & Software Technology, 2017, 88: 67-95

[5] Avdiienko V, Kuznetsov K, Gorla A, et al. Mining apps for abnormal usage of sensitive data [C] //Proc of the 37th Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2015: 426-436

[6] Arzt S, Rasthofer S, Fritz C, et al. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps [C] //Proc of the 35th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2014: 259-269

[7] Lerch J, Hermann B, Bodden E, et al. FlowTwist: Efficient context-sensitive inside-out taint analysis for large codebases [C] //Proc of the 22nd ACM SIGSOFT Int Symp on Foundations of Software Engineering. New York: ACM, 2014: 98-108

[8] Mirzaei O, Suarez-Tangil G, Tapiador J, et al. TriFlow: Triaging Android applications using speculative information flows [C] //Proc of the 2017 ACM on Asia Conf on Computer and Communications Security. New York: ACM, 2017: 640-651

[9] Yang Wei, Xiao Xusheng, Andow B, et al. AppContext: Differentiating malicious and benign mobile app behaviors using context [C] //Proc of the 37th IEEE/ACM Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2015: 303-313

[10] Hardekopf B, Lin C. Semi-sparse flow-sensitive pointer analysis [C] //Proc of the 36th ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages. New York: ACM, 2009: 226-238

[11] Hardekopf B. Pointer Analysis: Building a Foundation for Effective Program Analysis [M]. Austin: The University of Texas, 2009

[12] Hardekopf B, Lin C. Flow-sensitive pointer analysis for millions of lines of code [C] //Proc of the 9th Annual IEEE/ACM Int Symp on Code Generation and Optimization. Piscataway, NJ: IEEE, 2011: 289-298

[13] Chung K C A. Points-to analysis with efficient strong updates [C] //Proc of the 38th ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages. New York: ACM, 2011: 3-16

[14] Li Lian, Cifuentes C, Keynes N. Boosting the performance of flow-sensitive points-to analysis using value flow [C] //Proc of the 6th Joint Meeting of the European Software Engineering Conf and the ACM SIGSOFT Int Symp on Foundations of Software Engineering. New York: ACM, 2011: 343-353

[15] Oh H, Heo K, Lee W, et al. Design and implementation of sparse global analyses for C-like languages [J]. ACM SIGPLAN Notices, 2012, 47(6): 229-238

[16] Sui Yulei, Su Yu, Xue Jingling. Region-based selective flow-sensitive pointer analysis [C] //Proc of the 21st Int Static Analysis Symp. Berlin: Springer, 2014: 319-336

[17] Yu Hongtao, Xue Jingling, Huo Wei, et al. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code [C] //Proc of the 8th Annual IEEE/ACM Int Symp on Code Generation and Optimization. New York: ACM, 2010: 218-229

[18] Sui Yulei, Di Peng, Xue Jingling. Sparse flow-sensitive pointer analysis for multithreaded programs [C] //Proc of the 14th Annual IEEE/ACM Int Symp on Code Generation and Optimization. New York: ACM, 2016: 160-170

[19] Aho A V, Sethi R, Ullman J D. Compilers, Principles, Techniques [M]. New York: Addison Wesley, 1986

[20] Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability [C] //Proc of the 23rd ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages. New York: ACM, 1994: 49-61

[21] Maluszyski J. Program analysis via graph reachability [C] //Proc of the 14th Int Symp on Logic Programming. Cambridge: MIT Press, 1998: 5-19

[22] Lam P, Bodden E, Lhoták O, et al. The Soot framework for Java program analysis: A retrospective [C] //Proc of Cetus Users and Compiler Infrastructure Workshop. New York: ACM, 2011

[23] Litianwuxian Network Technology Co., Ltd. Anzhi App Store [OL]. [2017-10-01]. http://www.anzhi.com/applist.html (in Chinese)
(北京力天无限网络技术有限公司. 安智网应用市场[OL]. [2017-10-01]. http://www.anzhi.com/applist.html)

[24] Naeem N A, Lhoták O, Rodriguez J. Practical extensions to the IFDS algorithm [C] //Proc of the 19th Annual Int Conf on Compiler Construction. Berlin: Springer, 2010: 124-144

[25] Bodden E. Inter-procedural data-flow analysis with IFDS/IDE and Soot [C] //Proc of the ACM SIGPLAN Int Workshop on State of the Art in Java Program analysis. New York: ACM, 2012: 3-8

[26] Arzt S, Rasthofer S, Bodden E. The soot-based toolchain for analyzing android apps [C] //Proc of the 4th Int Conf on Mobile Software Engineering and Systems. Piscataway, NJ: IEEE, 2017: 13-24

[27] Rasthofer S, Arzt S, Bodden E. A Machine-learning approach for classifying and categorizing Android sources and sinks [C/OL] //Proc of the 21st Annual Network and Distributed System Security Symp. 2014 [2017-09-01]. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.432.7534

[28] Li Li, Bartel A, Klein J, et al. IccTA: Detecting inter-component privacy leaks in Android apps [C] //Proc of the 37th Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2015: 280-291

[29] Arzt S, Bodden E. StubDroid: Automatic inference of precise data-flow summaries for the android framework [C] //Proc of the 38th Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2016: 725-735

[30] Rasthofer S, Arzt S, Miltenberger M, et al. Harvesting runtime values in Android applications that feature anti-analysis techniques [C/OL] //Proc of the 23rd Annual Network and Distributed System Security Symp. 2016 [2017-09-01]. http://www.bodden.de/pubs/ssme16harvesting.pdf

[31] Tripp O, Pistoia M, Cousot P, et al. Andromeda: Accurate and scalable security analysis of web applications [C] / /Proc of the 2013 Int Conf on Fundamental Approaches to Software Engineering. Berlin: Springer, 2013: 210-225

[32] Sridharan M, Gopan D, Shan L. Demand-driven points-to analysis for Java [J]. ACM SIGPLAN Notices, 2005, 40 (10): 59-76

[33] Sridharan M. Refinement-based context-sensitive points-to analysis for Java [J]. ACM SIGPLAN Notices, 2006, 41 (6): 387-400

[34] Späth J, Do LN, Ali K, Bodden E. Boomerang: Demand-driven flow-and context-sensitive pointer analysis forJava [C] //Proc of the 30th European Conf on Object-Oriented Programming. Dagstuhl: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016: 22-26

[35] Crandall J R, Chong F T. Minos: Control data attack prevention orthogonal to memory model [C] //Proc of the 37th Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2004: 221-232

[36] Tripp O, Pistoia M, Fink S J, et al. TAJ: Effective taint analysis of web applications [C] //Proc of the 30th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2009: 87-97

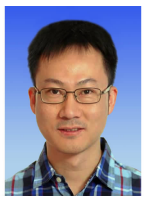
[37] Enck W, Gilbert P, Chun B G, et al. TaintDroid: An information flow tracking system for real-time privacy monitoring on smartphones [J]. ACM Transactions on Computer Systems, 2010, 32(2): 1-29

[38] Lu Long, Li Zhichun, Wu Zhenyu, et al. CHEX: Statically vetting Android apps for component hijacking vulnerabilities [C] //Proc of the 19th ACM Conf on Computer and Communications Security. New York: ACM, 2012: 229-240

[39] Gordon M I, Kim D, Perkins J, et al. Information-flow analysis of Android applications in DroidSafe [C/OL] //Proc of the 22nd Annual Network and Distributed System Security Symp. 2015 [2017-09-01]. https://www. ndss-symposium. org/ndss2015/ndss-2015-programme/information-flow-analysis-android-applications-droidsafe

Wang Lei, born in 1989. PhD. His main research interests include program analysis, Android security.

He Dongjie, born in 1992. Master. His main research interests include program analysis, Android security.



Li Lian, born in 1977. PhD, professor. PhD supervisor at the Institute of Computing Technology, Chinese Academy of Sciences. Member of CCF. His main research interests include program analysis, compiler, software security.

Feng Xiaobing, born in 1969. PhD. Professor and PhD supervisor at the Institute of Computing Technology, Chinese Academy of Sciences. Member of CCF. His main research interests include compiler optimization, binary translation.