

基于通信顺序进程的 Android 程序复杂信息流分析方法

袁占慧¹, 杨智¹, 张红旗¹, 金舒原², 杜学绘¹

(1. 信息工程大学, 河南 郑州 450001;

2. 中山大学, 广东 广州 510006)

摘要: Android 隐私泄露问题日益严重, 信息流分析是发现隐私泄露的一种主要方法。传统信息流分析方法以单一可达性分析为主, 难以分析复杂信息流。提出一种基于通信顺序进程的信息流分析方法, 建立应用程序行为的形式化模型, 从而全面刻画程序信息流行为。基于进程迹等价分析方法能够自动化验证信息流关联、信息流约束等复杂信息流问题, 进而判断是否存在敏感信息泄露。实验表明, 所提方法能够达到 90.99% 的准确率。

关键词: 安卓; 信息流分析; 隐私防护; 形式化分析; 通信顺序进程

中图分类号: TP309

文献标识码: A

DOI: 10.11959/j.issn.2096-109x.2021086

Android complex information flow analysis method based on communicating sequential process

YUAN Zhanhui¹, YANG Zhi¹, ZHANG Hongqi¹, JIN Shuyuan², DU Xuehui¹

1. Information Engineering University, Zhengzhou 450001, China

2. Sun Yat-sen University, Guangzhou 510006, China

Abstract: Android privacy leak problem is becoming more and more serious. Information flow analysis is a main method to find privacy leak. Traditional information flow analysis methods mainly focus on single accessibility analysis, which is difficult to analyze complex information flow. An information flow analysis method based on communication sequence process was proposed. The formal model of application behavior was established, which can fully describe the information flow of program. The process trace equivalence analysis method could automatically verify complex information flow problems such as information flow association and information flow constraints. This method could detect whether the application program leaks sensitive information. Experimental results show that the accuracy of the proposed method can reach 90.99%.

Keywords: Android, information flow analysis, privacy protection, formal analysis, communicating sequential process

收稿日期: 2021-08-19; 修回日期: 2021-09-23

通信作者: 杨智, zynoah@163.com

基金项目: 国家自然科学基金 (62176265, 61972040)

Foundation Item: The National Natural Science Foundation of China (62176265, 61972040)

论文引用格式: 袁占慧, 杨智, 张红旗, 等. 基于通信顺序进程的 Android 程序复杂信息流分析方法[J]. 网络与信息安全学报, 2021, 7(5): 156-168.

YUAN Z H, YANG Z, ZHANG H Q, et al. Android complex information flow analysis method based on communicating sequential process[J]. Chinese Journal of Network and Information Security, 2021, 7(5): 156-168.

1 引言

智能手机逐渐融入人们的日常生活，成为生活中不可或缺的一部分，越来越多的隐私信息、私人数据等被存储在智能手机中。Android 系统在智能手机的操作系统中占有极高份额，然而由于其具有开放性，大量不可信组件和第三方应用程序威胁着用户的隐私信息安全。针对 Android 系统的隐私敏感信息流分析具有很高的应用价值，是当前 Android 应用程序安全和用户隐私信息保护的迫切需要。

当前比较常见的信息流分析（IFA，information flow analysis）方法，主要采用污点跟踪的方式，分别从静态、动态或动态与静态混合的方向入手，将信息流分析问题抽象为可达性问题，最终获得可达性信息流路径结果。只对信息流处理可达性问题具有很大的局限性，无法处理复杂信息流。例如，在针对多污点源的信息流分析，以及存在反射的信息流分析问题时，以 FlowDroid 为代表的一类信息流分析方法无法给出复杂的信息流分析结果。本文采用基于通信顺序进程（CSP，communicating sequential process）^[1]的形式化分析方法全面捕捉信息流行为，完成复杂情况下的信息流分析。

CSP 是由 Hoare^[1]提出的一种著名的进程代数模型，能够很好地描述并发进程之间的相互作用。CSP 可以通过代数推演对具有并发性、不确定性的分布式软件系统进行分析，通过定义进程的迹，将进程从开始运行到某一时刻内所有顺序发生的事件进行建模，通过事件序列刻画进程行为。CSP 的特性很好地契合了复杂信息流分析的要求。基于 CSP 对 Android 程序进行信息流分析，是形式化方法在信息流分析领域的创新性应用。

信息流分析是一种重要的软件分析方法，在 Android 应用程序的安全性分析以及隐私防护方面有广泛的应用。TaintDroid^[2]是一个动态的全系统信息流跟踪工具，可以同时跟踪多个敏感数据源，通过集成变量级、消息级、方法级和文件级粒度的污点传播实现跟踪分析。这种动态的信息流分析方法能够获得足够的上下文信息且有效减少误报，已被应用于信息流控制^[3-6]、漏洞发

现^[7-10]、安全攻击^[10-12]、恶意软件检测^[13-14]、隐私泄露分析^[15-17]等领域。FlowDroid^[18]是著名的静态信息流分析工具，在 Android 应用程序的单个组件中执行静态污染分析。FlowDroid 充分模拟了 Android 特有的应用程序生命周期和回调方法，用于减少漏报或误报。Amandroid^[19]和 IccTA^[20]执行组件间通信污点分析，以检测基于组件间通信（ICC，inter-component communication）的隐私泄露。Bianchi 等^[21]采用反向数据流分析方法跟踪 API 的信息流，针对触摸屏界面存在的安全风险及带有视图覆盖替换的钓鱼攻击，深度挖掘具有风险的系统调用。InputScope^[22]使用静态污点分析来标记用户输入并监视其传播，以识别用户输入验证，这些验证会暴露隐私信息，如后门和黑名单。传统信息流分析方法往往缺乏复杂信息流分析能力。

形式化方法在 Android 信息流分析检测中的应用研究相对有限，当前研究主要对程序进行形式化建模，以进行模型检验^[23-28]。Song 等^[23]利用下推系统对 Android 应用程序的 Smali 代码进行建模，并通过计算树逻辑或线性时态逻辑来表达恶意行为。Bai 等^[24]构建了一个通用框架 DROIDPF，用以验证 Android 应用程序的安全性，由于 DROIDPF 侧重于应用程序中常见的安全漏洞，因此需要手动建立每个漏洞的状态机模型。Mercaldo 等^[25]围绕恶意软件家族的检测和分类发表了一系列研究成果。他们在文献[25-27]中分别检测到了与勒索软件、更新攻击和重新包装相关的系列恶意软件，并在此基础上提出了一种通用的分类方法，命名为 LEILA^[28]。这一系列研究通过基于通信系统演算（CCS，calculus of communicating system）的 Java 字节码建模^[29]来解决相关问题，定义选择性演算公式^[30]，对恶意软件家族的时序逻辑属性进行编码，并通过模型检查获得家族分类。

形式化检测的关键是能够用形式化语言准确、全面地描述代码行为，并建立行为模型。模型检查过程通过数学推导由形式化工具实现自动化。然而，现有的形式化信息流分析方法存在不足。例如，文献[28]中的 LEILA 系统，无法处理虚拟调用和反射调用的混淆。此外，数据流没有

被考虑,因为它没有对赋值语句建模。

为了弥补传统信息流分析方法和现有形式化分析方法在复杂信息流分析方面存在的不足,本文提出了一种基于通信顺序进程的Android程序复杂信息流分析方法,其核心思想是利用CSP对信息流传播过程进行建模,利用形式化的方法,更加详细地刻画出复杂信息流的传播过程,全面捕捉信息流行为,完成复杂情况下的信息流分析。

本文的主要贡献如下。

(1) 提出了基于通信顺序进程的Android程序复杂信息流分析方法,解决了传统信息流分析方法难以对复杂信息流进行分析的问题,包括多信息流关联关系分析、控制行为触发的复杂信息流分析等,进而判断Android程序是否存在敏感信息泄露。

(2) 按照信息流性质进行分类,对不同种类信息流路径分别进行建模分析,基于Jimple中间语言建立了适合Android应用程序的CSP信息流分析模型。

(3) 对信息流的多种特征进行了建模,包括反射、过程间函数调用等,解决了传统静态信息流分析的短板问题。实验表明,所提方法能够正确执行信息流分析功能,并达到90.99%

的准确率,高于传统静态信息流分析方法FlowDroid。

2 系统设计

本文提出的基于通信顺序进程的Android程序复杂信息流分析方法,是基于将应用程序的信息流传播过程用CSP模型进行建模而完成的,在此基础上,再进行信息流行为提取,判断是否存在敏感信息泄露,最后报告敏感信息流路径。整体工作流程和系统框架如图1所示。

首先,对目标APK文件进行分析和代码转换,将APK文件转换为中间表示形式再建立CSP模型。然后,建立CSP模型,刻画典型信息流性质。最后,将目标应用程序的CSP模型和典型信息流CSP模型同时输入FDR(failures divergence refinement)检测器进行检测,获得信息流路径的进程报告。

由于Java源代码和Java字节码中的语句类型非常复杂,因此快速有效地提取代码中的信息非常重要。基于以上考虑,首先进行APK文件反编译,然后调用SOOT将Java字节码转换成Jimple中间语言^[31]。Jimple具有语句类型少、类型清晰、3种地址形式等优点,有利于后续的形式化建模。

将Jimple语句转化为对应的CSP进程,即可

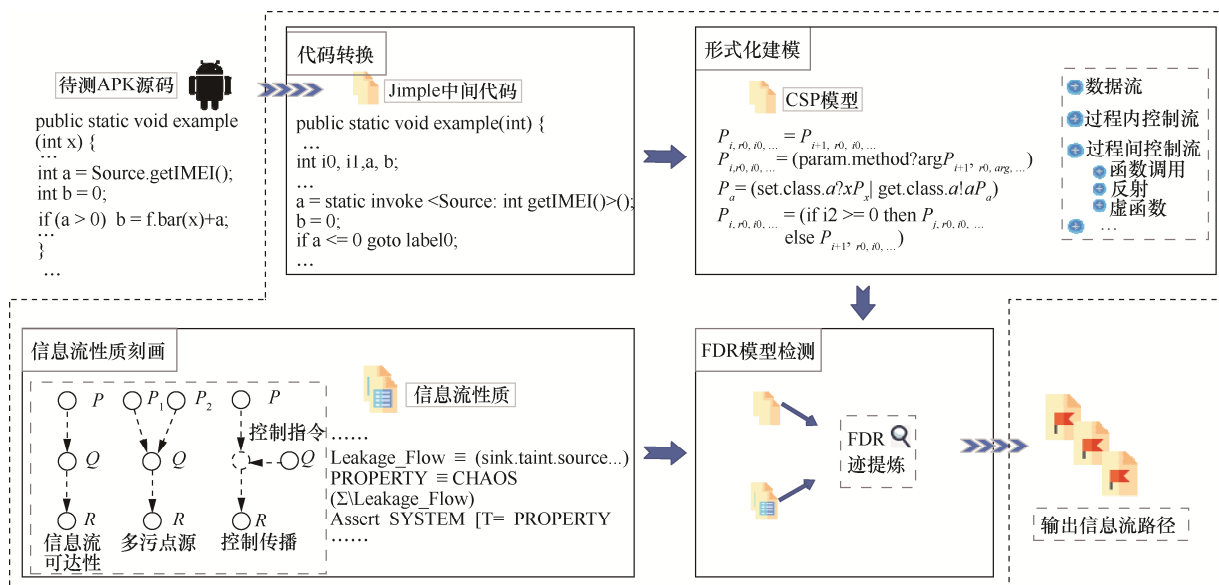


图1 整体工作流程系统框架

Figure 1 The framework of the work process system

完成对 Android 应用程序的正式建模。由于 Jimple 中只有 15 条语句, 而且 3 个地址形式使语义非常清晰, 因此将 Jimple 表示为 CSP 会更容易。每一个语句的过程范式都是建立在其特征的基础上的, 是对代码行为的准确抽象表达。本文在此基础上提出了一个完整的 Android 应用程序模型。

本文使用 FDR 模型检测器检测 Android 恶意软件, FDR 模型检测器是一个应用于 CSP 的正式工具。对样本行为建模的 CSP 过程被认为是需要验证的过程, 它被简化为只包含属性过程中出现的事件, 最终结果是断言测试 (如果存在) 的反例。

本文使用模型检测工具 FDR 对目标信息流进行形式化分析。该工具引入了迹的概念, 用可以执行的有限通信序列的集合表示进程。基于迹模型, FDR 用迹提炼来定义两个进程之间的特殊关系。

如果存在两个进程 P 和 Q , Q 的所有特定行为对于 P 来说都具有, 那么称 Q 提炼 P , 或 Q 是 P 的一个提炼, 记为 $P \subseteq Q$ 。进一步, 将迹模型应用到提炼的概念中: 如果 Q 的所有通信事件序列对于 P 来说也是可能的, 即 $\text{traces}(Q) \subseteq \text{traces}(P)$, 那么称 Q 迹提炼 P , 或 Q 是 P 的一个迹提炼, 记为 $P \subseteq_T Q$ 。

在对信息流进行 FDR 分析检测时, 将 P 作为抽象每条目标信息流特有性质的进程, 将 Q 作为建模目标 APK 文件样本中信息流行为的进程。在一次测试中, 只将一个样本的待测进程与一个信息流进程同时输入 FDR 检测器, 在样本进程只保留该性质中事件的基础上, 判断其是否是信息流进程的迹提炼。如果模型检测通过, 则说明该样本具有该信息流类型所具有的性质, 因此报告该信息流路径; 否则, 认为样本不存在该进程类型的信息流路径, 可以继续将该样本进程与其他信息流进程进行测试。

3 CSP 形式化建模

为了更加便捷高效地对信息流传播过程进行建模, 本文引入了 Jimple 中间语言^[31]。这种基于 SOOT 框架的三地址中间表示形式, 语句种类较少且每条语句最多只会出现 3 个量值, 非常适合

对 Java 语言的分析及优化。信息流传播中涉及的 Jimple 语句类型如图 2 所示。

NopStmt	IdentityStmt	AssignStmt
If/GotoStmt	TableSwitchStmt	LookupSwitchStmt
InvokeStmt	ReturnStmt	ReturnVoidStmt
EnterMonitorStmt	ExitMonitorStmt	ThrowStmt
BreakpointStmt	RetStmt	

图 2 15 种 Jimple 语句类型 (If/Goto 语句为一组)
Figure 2 15 Jimple statements (If/Goto statements are a group)

如果将一段 Jimple 代码视为一个整体系统, 那么其中包含的每一条语句都是它的组成部分, 都可以建模成一个 CSP。语句的进程名称由带参数类型的完整方法名和下标组成, 第一个下标是当前语句在方法中的编号, 然后是当前方法中的所有非静态局部变量、方法所在类的所有非静态成员变量以及方法中创建的所有类对象的非静态成员变量, 以便更好地表明变量值的变化。例如, 某方法的第 i 条语句对应的进程名可以表示为 $\text{Test_example}(\text{int})_{i,r0,i0,\dots}$, 其中 i 的值不大于该方法的 Jimple 语句总数 (从定义语句后开始计数)。将不同种类的 Jimple 语句转换成对应的 CSP 时, 所有的进程均可以被命名为 $P_{i,r0,i0,\dots}$ 的形式。

当涉及类的成员变量或成员方法的变量时, 需要在两个进程间传递数据。若对其进行赋值, 如 $c.<\text{Class: int } a> = \text{temp}\$1$ 对应的 CSP 进程为 $P_{i,r0,i0,\dots} = (\text{set.class.a!temp}\$1 \rightarrow P_{i+1,r0,i0,\dots})$; 否则, 如 $\text{temp}\$1 = c.<\text{Class: int } a>$ 对应的 CSP 进程为 $P_{i,r0,i0,\dots,\text{temp}\$1,\dots} = (\text{get.class.a?a} \rightarrow P_{i+1,r0,i0,\dots,a,\dots})$ 。其中, $\text{set.class.a!temp}\$1$ 是输出 $\text{temp}\$1$ 作为 class.a 赋值的事件, get.class.a?a 是接收 class.a 的值的值的事件。

本文为每个类建模一个进程来管理类中所有成员变量和成员方法中变量的值, 同时, 为每个实例化的类建立编号 ID。若类 class 中有一个上述类型的变量 a , 则它对应的 CSP 进程为 $P_{(\text{ID},a)} = (\text{ID.set.class.a?x} \rightarrow P_x \mid \text{ID.get.class.a!a} \rightarrow P_a)$ 。其中, 编号 ID 能够实现对类的记录, 通过编号区分不同类的成员变量和成员方法中变量的值。 set.class.a?x 是 class 接收 x 作为 a 的赋值的事件, get.class.a!a 是 class 输出 a 的值的值的事件。

典型的信息流行为模式包括数据流行为、控

制流行为、反射行为、虚函数等。基于对不同种类的 Jimple 语句进行 CSP 建模分析, 本文分别建立了 5 种信息流行为的 CSP 模型。

3.1 数据流行为建模

数据流行为可以认为是由 NopStmt、IdentityStmt 和 AssignStmt 这 3 种类型的语句进行组合的行为。NopStmt 表示变量值未发生变化的语句, 这条语句没有经过任何能引起变量值变化的事件, 此时直接进入下一条语句的进程; IdentityStmt 表示将参数值赋值给变量的语句, 将调用语句中的实参传入被调用方法; AssignStmt 表示变量的一般赋值语句, 体现变量被感染的过程。数据流行为建模如图 3 所示。

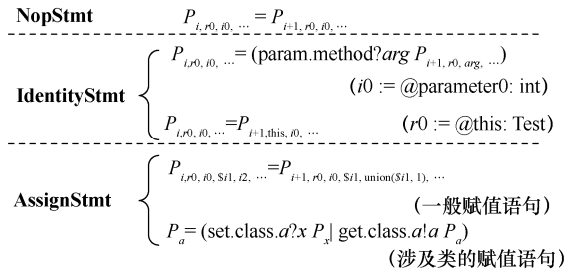


图 3 数据流行为建模
Figure 3 Modeling for data flow

IdentityStmt 语句的两种模型形式分别对应与参数有关的变量赋值语句和与 this 引用有关的变量赋值语句, param.method?arg 表示当前方法 method 接收实参 arg 的事件。对涉及参数的赋值语句, 需要在调用语句和被调用语句之间传递实参。在多参数情况下, 按序进行的参数赋值更便于查找输入值。

3.2 过程内控制流行为建模

本文将过程内控制流行为认为是由 IfStmt/GotoStmt、TableSwitchStmt 和 LookupSwitchStmt 这 3 种类型的语句组合而成的行为。IfStmt/GotoStmt 表示条件判断和跳转语句, 能够指示满足判断条件时要进行处理的语句位置; TableSwitchStmt 是表转换语句, 用于 switch 分支的条件值分布比较集中的情况; LookupSwitchStmt 是查找转换语句, 用于 switch 分支的条件值分布比较稀疏的情况。过程内控制流行为建模如图 4 所示。

IfStmt/GotoStmt 语句的第一种模型形式中的

j 是跳转目标的语句序号。为了得到 j 的值, 需要在当前方法的后续语句范围内搜索该 label 标记。该进程不执行任何实际的事件, 只是跳转到标号所在的语句进程继续执行。第二种模型形式的进程提供两个选择, 如果 IfStmt 给出的判断条件成立, 则按照相应 label 标记处的语句进程继续执行; 否则按照下一条语句的进程顺序执行。

$$\text{IfStmt/GotoStmt} \begin{cases} P_{i,r0,r0,...} = P_{j,r0,r0,...} & (\text{直接Goto跳转}) \\ P_{i,r0,r0,...} = (\text{if } i2 \geq 0 \text{ then } P_{j,r0,r0,...} \text{ else } P_{i+1,r0,r0,...}) & (\text{先判断后跳转}) \end{cases}$$

$$\text{TableSwitchStmt / LookupSwitchStmt} \quad P_{i,r0,r0,...} = |k: \text{switch } @ P_{\text{goto}(k), r0, r0, ...}$$

图 4 过程内控制流行为建模
Figure 4 Modeling for intraprocedural control flow

TableSwitchStmt 和 LookupSwitchStmt 语句分别通过键值匹配和索引访问跳转表, 模型中 k 是 case 分支的条件值, goto(k) 是处理该情况的语句的起始编号。该进程表示在选择合适的 case 分支后, 跳转到相应标号位置处继续执行。两种代码的形式完全相同, 因此共享相同的 CSP 进程。

3.3 过程间控制流行为建模

过程间控制流行为包括函数调用行为、反射行为以及虚函数, 本节将依次对这 3 种情况进行建模分析。

3.3.1 函数调用建模

本文将函数调用行为认为是由 InvokeStmt、ReturnStmt 和 ReturnVoidStmt 这 3 种类型的语句进行组合形成的。InvokeStmt 表示调用语句; ReturnStmt 指有返回值的返回语句, ReturnVoidStmt 指没有返回值的返回语句“return”。函数调用行为建模如图 5 所示。

$$\text{InvokeStmt} \begin{cases} P_{i,r0,r0,...} = (\text{call.method param.method!arg0 param.method!arg1 ... retval.method ?valreturn.method } P_{i+1,r0,r0,...}) & (\text{被调用的method有参数和返回值}) \\ P_{i,r0,r0,...} = \text{thread.run} \parallel P_{i+1,r0,r0,...} & (\text{创建新线程}) \end{cases}$$

$$\text{ReturnStmt} \quad P_{i,r0,r0,...} = (\text{retval.method!val end.method } P)$$

$$\text{ReturnVoidStmt} \quad P_{i,r0,r0,...} = (\text{end.method } P)$$

图 5 函数调用行为建模
Figure 5 Modeling for function call

对于 InvokeStmt 调用语句, 当被调用的 method 有参数和返回值时, CSP 模型中的 param.method!arg0、param.method!arg1 等表示向 method 依次传递实参的事件, retval.method?val 表示从 method 接收 val 的事件, call.method 和 return.method 分别表示 method 调用开始和结束的事件。当被调用的方法没有参数或返回值时, 在进程表达式中去掉相应的事件, 通过 IdentityStmt 语句进行实参接收。

新线程的创建过程也遵循 InvokeStmt 方法调用的形式, 但有所不同的是, 它不需要像一般调用中那样等待被调用方完成后才能继续执行, 因此具有如图 5 所示的新的 CSP 模型形式, 其中 thread.run 指要创建的线程类 thread 的 run 方法对应的进程。

对于 ReturnStmt 语句的 CSP 模型进程, retval.method!val 是当前方法 method 返回变量 val 的事件。该进程表示返回需要的结果后结束所在方法的此次执行, 并回到方法开始处等待下次调用。

对于 ReturnVoidStmt 语句的 CSP 模型进程, 用 method 表示当前方法, 该进程不返回任何值, 只用于结束所在方法的此次执行。

在对过程间通信行为进行建模时, 需要考虑系统调用的同步机制。为了确保任意方法在同一时刻最多只能被另一个方法调用, 用 SYNM 进程来描述系统的调用同步机制, 如图 6 所示。系统调用同步机制模型由各个调用同步事件的控制进程 SYNM_method 组成, 其中 method 指代被调用的方法, method1、method2 等则是系统中不同的被调用方法。

```
SYNM_method = (call.method begin.method end.method
               return.method SYNM_method)
SYNM = (SYNM_method1 ||| SYN_method2 ||| ...)
```

图6 系统的调用同步机制

Figure 6 System call synchronization mechanism

被调用方法有两个参数和一个返回值, 每个方法的开始和结尾处分别设置了提示事件 begin 和 end。由于方法的起始语句不确定, 因此将 begin 事件额外作为第一个进程的前缀; 而返回语句通常为方法的最后一条语句, 因此将 end 事件放入返回语句的进程表达式中。由此得出

method 对应的 CSP 进程为 method = (begin.method → method_{1, r0, r1, ...})

调用语句与被调用方法之间的同步关系如图 7 所示。在调用语句中, call 和 return 事件将分别作为本次调用的开始和结束事件。SYNM 进程的同步集是 call、begin、end、return 事件, 被调用方法执行完毕后将控制返回到 InvokeStmt 的下一条语句。

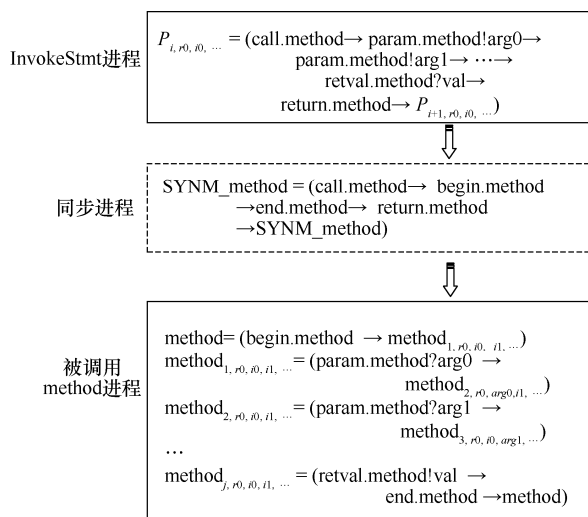


图7 InvokeStmt 及被调用方法之间的同步

Figure 7 Synchronization between invokeStmt and the called method

3.3.2 反射行为建模

反射行为作为 Java 语言独特的功能, 能够在过程间信息流中动态获取信息并调用对象方法, 需要单独进行建模分析。为了静态地解析反射, 本文定义 allclasses 数组来记录程序中的所有类。反射行为中包含 forName、getMethod 和 invoke 这 3 种比较重要的方法调用, 方法调用的示例如图 8 所示。

```
c = static invoke <java.lang.Class: java.lang.class
  forName(java.lang.string)>(cla) (a)
m = virtual invoke c.<java.lang.Class: java.lang.reflect.Method
  getMethod(java.lang.string, java.lang.class[])>
  ("send", ARG_TYPE) (b)
a = virtual invoke m.<java.lang.reflect.Method: java.lang.object
  invoke(java.lang.object, java.lang.object[])>
  (null, b) (c)
```

图8 反射行为中的方法调用示例

Figure 8 Example of method calls in reflection

forName 方法的调用, 在 allclasses 数组中查找名称为 cla 的值的类, 并将其索引 allclasses_index(cla)赋值给 c, 对应的 CSP 进程如图 9(a)所示。

getMethod 方法的调用, 在 cla 类的方法数组 allmethodc 中查找名称为 send, 参数类型为 ARG_type 的值的方法, 并将其索引 m_index (“send”, ARG_type)&c 赋给 m, 相应的 CSP 进程如图 9(b) 所示。

invoke 方法的调用, 主要负责反射调用的具体执行, CSP 进程如图 9(c) 所示。

$$\begin{aligned} P_{i, r0, r0, \dots, c, \dots} &= P_{i, r0, r0, \dots, \text{allclasses_index}(cla), \dots} & (a) \\ P_{i, r0, r0, \dots, m, \dots} &= P_{i, r0, r0, \dots, (c, m_index(\text{"send"}, ARG_TYPE)), \dots} & (b) \\ P_{i, r0, r0, \dots, \dots} &= (\text{invoke}.c.m \text{ param}.c.m!b \text{ retval}.c.m?x \\ &\quad \text{invokereturn}.c.m P_{i+1, r0, r0, \dots, x, \dots}) & (c) \end{aligned}$$

图 9 反射行为建模

Figure 9 Modeling for reflection

程序中每个动态加载的方法都被记录下来后, 构造相应的反射过程来控制方法的执行, 如图 10 所示。

```
REFLECTION_method(c,m)=
  invoke.c.m begin.method param.c.m!temp $4
  param.method?temp $4 retval.method!temp $5
  retval.c.m?temp $5 end.method invoke return.c.m
  REFLECTION_method(c, m)
REFLECTION =
  (REFLECTION_method1(c1, m1) |||
  REFLECTION_method2(c2, m2) ||| ...)
```

图 10 构造反射过程

Figure 10 Constructing for reflection

3.3.3 虚函数建模

以 Java 的继承机制为基础虚函数调用, 也是 Java 语言一种独特的函数调用方式, 因此单独进行建模分析。Java 支持继承, 允许创建层次类。子类不仅可以继承父类的特性和行为, 还可以通过虚调用使用父类的方法。

与反射行为建模过程类似, 同样要为程序中所有类依照从属关系建立一个数组 allclasses 来存储它们的父类。此外, 还要为类 c 建立一个数组 allmethodc 来记录类中的所有方法名。

在处理调用语句时, 首先检查被调用方法的类层次结构是否正确, 如果正确, 则根据上述规范对流程进行建模; 否则, 搜索其父类并依次执行, 直到找到方法为止, 并用模型中实际调用的方法名替换该方法。图 11(a) 所示的语句为一个虚函数调用, getIMEI() 方法在源类中不存在, 但可以在其父类 SourceFather 中找到该方法。因此, 相应的 CSP 过程模型如图 11(b) 所示。

$$a = \text{static invoke} \langle \text{Source: int getIMEI}() \rangle (a) \quad (a)$$

$$\begin{aligned} P_{i, r0, r0, a, \dots} &= (\text{call.SourceFather_getIMEI}() \\ &\quad \text{retval.SourceFather_getIMEI}() ? x \\ &\quad \text{return.SourceFather_getIMEI}() \\ &\quad P_{i+1, r0, r0, x, \dots}) \end{aligned} \quad (b)$$

图 11 虚函数建模

Figure 11 Modeling for virtual function

4 信息流性质刻画与验证

4.1 信息流性质刻画

在信息流泄露事件中, 部分信息流行为模式普遍存在, 是构成泄露事件的重要步骤。即使这些行为模式存在多种不同的具体实现方式, 然而信息流传播逻辑相同, 形式化表达方式也相同。因此本文将不同种类的信息流行为模式认定为信息流的性质, 结合对典型信息流行为模式的建模, 刻画出泄露事件的信息流性质。

为了简化 FDR 模型检测的复杂程度, 对 CSP 模型中与信息流传播不相关的代码行为进行隐藏, 流程为: uselessevents = { |transfer, calldeal, api.normalx | }, SAFE_SAMPLE = SAMPLE \ uselessevents。接下来定义信息流发生泄露的性质为 PROPERTY = source.taint->sink.taint->STOP。基于对简化信息流 CSP 模型和信息流泄露性质的定义, 得出在 CSP 模型中发现信息流泄露路径的方式为 assert SAFE_SAMPLE [T= PROPERTY]。

最终结果作为断言测试的反例生成, 当判定结果为真时, 则认为存在信息泄露行为, 并且 FDR 模型检测工具能够报告出满足判定结果为真的信息流路径, 该路径即为本方法所求的目标信息流泄露路径。当此处判定结果为假时, 则说明不存在信息泄露行为。

基于以上信息流性质和泄露路径提取方式, 本文针对典型信息流分析过程, 建立了细化的信息流检测 CSP 模型与 FDR 验证方法。

(1) 待分析程序中含有多个敏感信息源 (source) 和敏感信息泄露点 (sink) 时 (如将地理位置、麦克风、通话记录信息等认定为敏感源, 将网络发送、短信发送、写入本地文件等认定为敏感信息泄露点), 为了找出从所有 source 点到所有 sink 点的信息流, 即找出所有可能存在的隐私敏感信息泄露路径, 本文提出如下 CSP 模型检测方法: Leakage_Flow $\equiv$ (sink. {sinklist}. taint.

{sourcelist}|sinklist<-{1...MAX},sourcelist<-{1...MAX}); PROPERTY $\equiv$ CHAOS(Σ \Leakage_Flow); PROPERTY [T= Assert SYSTEM。此方法将从所有 source 点出发到达所有 sink 点的信息流路径作为系统 CSP 模型的迹提炼输出,能够得到覆盖所有 source 点和 sink 点的信息流路径。

(2) 针对某应用程序进行分析时,有需要挖掘从指定 source 点到指定 sink 点的信息流路径的需求,即信息流定向搜索需求。例如,针对地理位置信息,搜索是否存在以地理位置信息为敏感源,以写入本地文件为泄露点的信息流路径,就属于信息流定向搜索。本文针对此类情况提出如下 CSP 模型检测方法: Leakage_Flow $\equiv$ (sink.sinkx.taint.sourcecx); PROPERTY $\equiv$ CHAOS(Σ \Leakage_Flow); PROPERTY [T= Assert SYSTEM。此方法通过将确定的 source 点和 sink 点引入模型检测过程,完成定向搜索功能。

(3) 在恶意应用程序中,某条信息流同时包含多个敏感信息的情况也比较常见,即一条信息流路径同时被多个敏感源感染,有多个 source 点。例如,恶意软件可能将通话记录和通信录以及其他非敏感信息打包,经过一条信息流路径,向网络发送。这种多污点聚合类的信息流分析对于传统的静态信息流分析工具也是不小的挑战。本文针对此类多污点聚合发送的信息流分析问题提出如下 CSP 模型检测方法: Leakage_Flow $\equiv$ (sink.taint.{sourcelist}|sourcelist<-{sourcea,source b,sourcec...sourcen}); PROPERTY $\equiv$ CHAOS(Σ \Leakage_Flow); PROPERTY [T= Assert SYSTEM。此方法针对指定的 sink 点,对 n 个 source 点进行迹提炼,从而挖掘出多污点聚合发送复杂信息流路径。

(4) 由控制行为触发的复杂信息流路径是传统静态信息流分析的难点,控制行为对信息流分析产生的影响在进行静态分析时很难识别。当某条信息流需要控制行为才能触发时(如当用户点击发送某张图片时,触发泄露地理位置信息),将会产生传统静态分析方法无法识别的信息流。本文针对这种情况提出如下 CSP 模型检测方法: Leakage_Flow $\equiv$ (sink.event.taint.source|event<-{button.click.OK}); PROPERTY $\equiv$ CHAOS (Σ \Leakage_Flow) ;

PROPERTY [T=Assert SYSTEM。

形如上述类型的复杂信息流分析模型,都能够转化为以 CSP 模型描述的信息流性质,进而应用到模型检测中实现信息流分析挖掘功能。

在实际应用中,往往不是针对应用程序的源码进行分析,而是只有 APK 文件资源。此时需要首先调用 SOOT 框架,直接将 APK 文件转化为 Jimple 中间代码形式,再进行后续 CSP 建模和 FDR 模型检测分析。

4.2 信息流性质验证

本文使用精化检测工具 FDR 对建立的形式化模型进行分析。FDR 已广泛应用于学术界和工业界,在协议漏洞挖掘、软件安全性验证、数学逻辑运算方面获得了成功应用。FDR 实现了 CSP 的 3 个验证模型,分别为迹模型、稳定失败模型和失败/发散模型,3 个模型验证进程间等价性的程度和需要验证的状态空间个数都是递增的。稳定失败模型可以验证非确定性行为,如验证活性。失败/发散模型避免因进程加入屏蔽算子导致的发散对分析带来的影响,实际上如果不是实际存在发散迹,而是用于验证分析时出现发散迹,则发散攻击是不存在的,不一定要要求发散迹也等价。而且迹模型可以对“不存在违背某个安全性质的反例”之类的断言进行验证。所以本文选取迹模型作为验证模型,这样可以降低复杂度且能满足本文的安全性要求。

FDR 内部通过有限状态系统等价性分析验证断言的正确性,主要分析复杂信息流的可达性和关联关系。对于单个污点源,假设程序的变量个数为 m ,语句个数为 n ,则程序的 CSP 对应的有限状态机的状态个数最大为 2^m ,每个状态的出度最大为 n 。FDR 的精化算法采用互模拟等价原则分析程序的有限状态机 M1 和安全属性的有限状态机 M2 之间的等价包含关系。首先选择程序运行时的初始状态然后递归分析,对于 M1 中的每个状态 q_{10} ,查找 M2 中是否存在等价状态 q_{20} ,则查找复杂度为 2^m ,对于每一个备选的 q_{20} ,验证所有从该状态的出边是否等价,则验证复杂度为 n 。所以最坏情况下总的计算开销为 $n \times 2^{2m}$,假设做 k 个不同污点源复杂信息流分析,则最坏情况的计算开销为 $n \times 2^{2mk}$ 。

FDR 采用了很多高效的方法进行了优化。状态可达性分析, 只分析从初始状态可达的状态; 状态压缩技术, 包括 `explicate` (枚举), `sbsim` (强节点标记互模拟), `tau_loop_factor` (循环消除), `diamond` (菱形消除), `normalise` (标准化) 和 `model_compress` (语义等价分解)。在本文研究的信息流复杂性分析中, 一般程序全局/静态变量相对较少, 大多数变量是局部变量, 离开了所在函数, 这些局部变量没有意义, 意味着绝大部分的状态变迁根本不会发生。每条语句通常只会影响少量变量的状态变化, 意味着状态机中总的边数通常不会超过 $10n$ 。所以这些优化技术可以极大地降低信息流分析的复杂度。另外, 大部分的信息流性质验证公式验证的是作用了隐藏算子后的进程等价性分析, 这其实是弱互模拟等价分析。验证所有从该状态的出边是否等价时, 复杂度进一步大幅降低。本文的实验表明了 FDR 在验证具体的信息流分析实例时具有良好的性能。

5 实验评估

本节将会对本文提出的信息流分析方法进行实验评估, 主要分为功能测试和性能测试两部分。为了保证实验条件的一致性, 本文的实验部分采用了与 FlowDroid 相同的 source 列表和 sink 列表。实验中 FlowDroid 涉及的部分参数配置如表 1 所示。

5.1 功能测试

功能测试的目的是验证本文提出的信息流分析方法的基本功能是否正确, 即本文方法是否能够正确执行信息流分析功能, 正确报告信息流泄露路径。

文献[32]提出了一种恶意应用程序中包含的复杂信息流情况, 文献举例的恶意应用程序伪装成视频播放器, 但却开启后台服务窃取包括 IMEI、IMSI 在内的用户敏感信息; 另一个良性应用程序同样使用了这两项敏感信息并进行发送。两种应用程序中都含有通过 Internet 发送国际移动设备识别码 (IMEI, international mobile equipment identity)、国际移动用户识别码 (IMSI, international mobile subscriber identity) 的信息流路径, 但两种信息流结构却大不相同, 如图 12 所示。

表 1 FlowDroid 参数配置
Table 1 Parameter configuration of FlowDroid

参数名称	参数状态
<code>staticTracking</code>	true
<code>implicitFlows</code>	true
<code>enableCallbacks</code>	true
<code>enableExceptions</code>	true
<code>accessPathLength</code>	5
<code>flowSensitiveAliasing</code>	true
<code>computeResultPaths</code>	true
<code>aggressiveTaintWrapper</code>	false
<code>librarySummaryTaintWrapper</code>	false
<code>enableImplicitFlows</code>	true
<code>enableStaticFields</code>	true
<code>enableTypeChecking</code>	true
<code>ignoreFlowsInSystemPackages</code>	true
<code>inspectSources</code>	false
<code>inspectSinks</code>	false
<code>stopAfterFirstFlow</code>	false

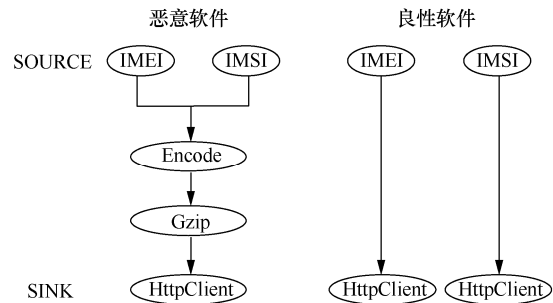


图 12 良性和恶意软件应用程序的行为比较
Figure 12 Behavior comparison between benign and malicious applications

实验表明, 恶意 App 中的信息流泄露行为大多数采用这种捆绑式的泄露方式。使用文献[32]提供的良性和恶意应用程序进行测试, 结果表明本文提出的基于通信顺序进程的复杂信息流分析方法, 能够正确执行多敏感源捆绑式的恶意信息流分析, 并能够得出与良性应用程序的区别。本文还对传统数据流分析工具 FlowDroid 进行了测试, 由于忽视了不同信息流的关系以及这两个应用程序的不同行为, FlowDroid 无法区分恶意应用程序行为。

本文提出的方法属于静态信息流分析方法, 因此选取静态信息流分析方法 FlowDroid 作为实

验对比对象,对比两种分析方法检测到的信息流路径是否一致。

FlowDroid^[18]是 Android 应用程序的高精度静态信息流分析方法,文献[18]使用了 InsecureBank APP 和 DroidBench 测试套件,测试方法的有效性和准确性。本文方法也将分别在这两个平台上与 FlowDroid 进行对比分析。

(1) 首先选取 InsecureBankv2 作为测试用例进行实验分析,测试本文方法挖掘复杂 App 内多条信息流路径的能力。对 InsecureBankv2 中存在的 9 个典型的信息流泄露漏洞进行测试的结果如表 1 所示。表中的★表示找到了信息流泄露路径。实验结果表明,两种信息流分析方法都发现了所有的信息流泄露路径。证明本文方法能够正确执行信息流分析功能。

表 2 InsecureBankv2 信息流泄露路径检测结果
Table 2 Results of information flow leak path detection of InsecureBankv2

信息流泄露漏洞名称	FlowDroid	本文方法
Insecure Content Provider access	★	★
Insecure Webview implementation	★	★
Sensitive Information in Memory	★	★
Insecure Logging mechanism	★	★
Android Pasteboard vulnerability	★	★
Android keyboard cache issues	★	★
Android Backup vulnerability	★	★
Insecure SD Card storage	★	★
Insecure HTTP connections	★	★

(2) 接下来利用 DroidBench2.0^[18]验证本文方法挖掘不同 App 内多条信息流路径的能力。DroidBench2.0 是一个开源的基准测试套件,常用于测试针对 Android 系统的信息流分析工具在数组、方法、指令、应用内外通信等方面的跟踪分析性能及有效性。DroidBench2.0 中 13 个类别的 120 个应用程序都被用于本实验,实验结果如表 3 所示。其中,TP 表示正确报告 App 存在的信息泄露路径数,FP 表示误报数,FN 表示漏报数。

所有测试用例中共有 115 条泄露路径。FlowDroid 和本文的分析方法都发现了 101 条路径,但 FlowDroid 误报了 13 条路径,本文方法误

报了 10 条路径。FlowDroid 误报但本文方法未误报的 3 条路径实际上从未出现在测试用例 Unregister1/Callbacks、Exceptions3/General Java 和 VirtualDispatch3/General Java 中。FlowDroid 没有为从未注册的回调、从未发生的异常和从未调用的方法中的代码提供更精确的分析。本文分析方法误报的 10 条路径涉及粗粒度数组或列表跟踪、未定义的源、不准确的反向别名分析等。为了避免巨大的存储开销,数组或列表中的所有数据项共享相同的污点标记,这会导致粗粒度跟踪和 4 次误报。FlowDroid 和本文提出的信息流分析方法定义的源比 DroidBench 定义的源更多,如 getLastKnownLocation、findViewById、getIntent 等,因此这两种方法都跟踪了过多的 source,所以这些误报事实上是合理的。

表 3 DroidBench2.0 信息流泄露检测结果
Table 3 Results of information flow leak path detection of DroidBench2.0

DroidBench2.0 类别	泄露路径数	FlowDroid			本文分析方法		
		TP	FP	FN	TP	FP	FN
Aliasing (1)	0	0	1	0	0	1	0
Arrays and Lists (7)	3	3	4	0	3	4	0
Callbacks (15)	17	17	1	0	17	0	0
Field and Object Sensitivity (7)	2	2	0	0	2	0	0
Inter-App Communication (3)	3	3	0	0	3	0	0
ICC (18)	19	17	0	2	17	0	2
Lifecycle (17)	17	17	2	0	17	2	0
General Java (23)	20	18	4	2	18	2	2
Android-Specific (13)	11	7	1	4	7	1	4
Implicit Flows (4)	8	7	0	1	7	0	1
Reflection (4)	4	1	0	3	1	0	3
Threading (5)	5	3	0	2	3	0	2
Emulator Detection (3)	6	6	0	0	6	0	0
总数	115	101	13	14	101	10	14
准确率 Precision=TP/(TP+FP)		88.60%			90.99%		
召回率 Recall=TP/(TP+FN)		87.83%			87.83%		

FlowDroid 和本文方法都漏掉了 14 条路径, 主要是因为一些特殊代码没有被跟踪, 包括在独立组件 (Singletons1/ICC)、ICC 处理程序 (ServiceCommunication1/ICC)、静态初始化方法 (StaticInitialization1/General java)、格式化程序 (StringFormatter1/General java)、文件系统访问 (PrivateDataLeak3/AndroidSpecific)、无目标类的类型信息的方法反射调用 (Reflection3/ Reflection) 中的污点传播等情况。

本文提出的信息流分析方法在此项实验中的召回率为 87.83%, 准确率为 90.99%。FlowDroid 的召回率同样为 87.83%, 但其准确率仅为 88.60%。

5.2 性能测试

本文的性能测试部分在具有 6 核 1.10 GHz, 英特尔 i7-10710U 处理器和 16 GB 内存的计算机上完成。

信息流静态分析工具分析应用程序的时间消耗可以作为性能指标, 反应分析方法的性能优劣。在进行性能测试时, 本文仍然以 FlowDroid 为实验对比方法, 以 DroidBench2.0 为实验测试集。性能测试结果如表 4 所示。

DroidBench2.0 共有 13 个类别, 如表 4 第一列所示, 本文对每个类中所有的应用程序分别进行测试并取平均值。表 4 中第二列表示 FlowDroid 的时间消耗, 记录的是将 APK 文件输入 FlowDroid 并最终获得信息流路径的时间; 第 3 列到第 5 列分别表示本方法中 3 个步骤分别消耗的时间, 即代码转换、形式化建模和模型检测分别消耗的时间; 最后一列表示本文提出的分析方法消耗的总时间。表中所有实验数据均为针对每个 APK 文件分别运行 3 次取平均值的結果。

代码转换和形式化建模消耗了部分时间来更全面地处理应用程序, 但占总体消耗时间的比例较小, 分别为 3.61%和 2.18%; 模型检测部分为本文方法输出信息流路径的关键部分, 需要更多的时间, 其时间消耗占总时间消耗的 94.22%。

实验结果表明, 本文提出的信息流分析方法虽然与其他形式化方法相同, 均具有分析过程耗时较长的问题, 但与传统信息流分析方法 FlowDroid 相比, 在时间消耗相差不大的情况下

表 4 性能测试结果
Table 4 Results of performance test

DroidBench2.0 类别	FlowDroid 平均耗时/s	代码转换平均耗时/s	形式化建模平均耗时/s	模型检测平均耗时/s	本文方法总平均耗时/s
Aliasing (1)	266.51	9.62	5.50	259.98	275.10
Arrays and Lists (7)	272.97	9.74	5.60	263.96	279.30
Callbacks (15)	253.05	10.24	6.06	256.41	272.71
Field and Object Sensitivity (7)	262.97	10.98	6.83	261.04	278.85
Inter-App Communication (3)	281.27	10.65	6.50	273.98	291.13
ICC (18)	302.77	10.10	5.90	294.47	310.47
Lifecycle (17)	296.91	10.44	6.21	292.82	309.47
General Java (23)	305.59	10.78	6.56	282.13	299.47
Android-Specific (13)	304.88	10.89	6.73	277.82	295.44
Implicit Flows (4)	306.32	10.31	6.14	290.75	307.20
Reflection (4)	300.34	10.52	6.34	291.95	308.81
Threading (5)	294.29	11.07	6.89	272.22	290.18
Emulator Detection (3)	269.85	11.42	7.24	254.32	272.98
平均值	285.98	10.52	6.35	274.76	291.62

(120 个测试用例的平均时间差值仅为 5.64 s), 获得了更高的准确率。

6 结束语

本文对 Android 敏感信息流进行分析, 以更加全面地挖掘隐私泄露信息流路径, 助力用户隐私信息防护为出发点, 提出了一种基于通信顺序进程的 Android 程序复杂信息流分析方法。该方法利用通信顺序进程模型对应用程序的信息流传播过程进行建模, 在此基础上, 刻画复杂信息流性质, 再进行信息流行为提取, 全面分析应用程序复杂信息流, 判断是否存在敏感信息泄露。该方法克服了传统信息流分析方法通过求解可达性问题获取路径, 难以处理复杂信息流的局限性, 实现了反射行为等信息流特征刻画, 解决了多种复杂信息流分析问题, 能够全面捕捉信息流行为, 完成复杂情况下的信息流泄露路径检测。通过与传统静态信息流分析方法对比分析, 表明本文方法能够正确执行信息流分析功能, 并能够达到 90.99% 的准确率, 高于传统静态信息流分析方法 FlowDroid。

参考文献:

- [1] HOARE C A R. Communicating sequential processes[M]. NJ, USA: Prentice Hall, 1985.
- [2] ENCK W, GILBERT P, CHUN B G, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones[J]. ACM Transactions on Computer Systems, 2010, 57(3): 393-407.
- [3] VACHHARAJANI N, BRIDGES M J, CHANG J, et al. RIFLE: an architectural framework for user-centric information-flow security[C]//Proceedings of 37th International Symposium on Microarchitecture (MICRO-37'04). 2004: 243-254.
- [4] BANERJEE S, DEVECSERY D, CHEN P M, et al. Iodine: fast dynamic taint tracking using rollback-free optimistic hybrid analysis[C]//Proceedings of 2019 IEEE Symposium on Security and Privacy (SP). 2019: 490-504.
- [5] ZHANG M, YIN H. Efficient, context-aware privacy leakage confinement for android applications without firmware modding[C]//Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security. 2014: 259-270.
- [6] JIA Y J, CHEN Q A, WANG S Q, et al. ContextIoT: towards providing contextual integrity to appified IoT platforms[C]//Proceedings of 2017 Network and Distributed System Security Symposium. 2017.
- [7] ZONG P, LV T, WANG D, et al. FuzzGuard: filtering out unreachable inputs in directed grey-box fuzzing through deep learning[C]//Proceedings of 29th USENIX Security Symposium (USENIX-SS'20). 2020.
- [8] SHE D D, CHEN Y Z, SHAH A, et al. Neutaint: efficient dynamic taint analysis with neural networks[C]//Proceedings of 2020 IEEE Symposium on Security and Privacy (SP). 2020: 1527-1543.
- [9] NGUYEN-TUONG A, GUARNIERI S, GREENE D, et al. Automatically hardening web applications using precise taint-ing[M]//Security and Privacy in the Age of Ubiquitous Computing. Boston, MA: Springer US, 2005: 295-307.
- [10] NEWSOME J, SONG D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software[C]//Proceedings of the Network and Distributed System Security Symposium (NDSS '05). 2005.
- [11] KONG J F, ZOU C C, ZHOU H Y. Improving software security via runtime instruction-level taint checking[C]//Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability. 2006: 18-24.
- [12] HALDAR V, CHANDRA D, FRANZ M. Dynamic taint propagation for Java[C]//Proceedings of 21st Annual Computer Security Applications Conference (ACSAC'05). 2005: 311.
- [13] VOGT P, NENTWICH F, JOVANOVIĆ N, et al. Cross site scripting prevention with dynamic data tainting and static analysis[C]//Proceedings of the Network and Distributed System Security Symposium (NDSS '07). 2007.
- [14] CABALLERO J, POOSANKAM P, MCCAMANT S, et al. Input generation via decomposition and restitching: finding bugs in malware[C]//Proceedings of the 17th ACM Conference on Computer and Communications Security. 2010: 413-425.
- [15] ENCK W, GILBERT P, CHUN B G, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones[C]//Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation. 2019: 393-407.
- [16] ZHU D Y, JUNG J, SONG D, et al. TaintEraser[J]. ACM SIGOPS Operating Systems Review, 2011, 45(1): 142-154.
- [17] KANG M G, MCCAMANT S, POOSANKAM P, et al. DTA++: dynamic taint analysis with targeted control-flow propagation[C]//Proceedings of the Network and Distributed System Security Symposium (NDSS '11). 2011.
- [18] ARZT S, RASTHOFER S, FRITZ C, et al. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android Apps[C]//Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2014: 259-269.
- [19] WEI F G, ROY S, OU X M, et al. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of Android Apps[C]//Proceedings of the ACM Conference on Computer and Communications Security. 2014: 1329-1341.
- [20] LI L, BARTEL A, BISSYANDÉ T F, et al. IccTA: detecting in-

ter-component privacy leaks in android Apps[C]//Proceedings of 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. 2015: 280-291.

- [21] BIANCHI A, CORBETTA J, INVERNIZZI L, et al. What the App is that? Deception and countermeasures in the android user interface[C]//Proceedings of 2015 IEEE Symposium on Security and Privacy. 2015: 931-948.
- [22] ZHAO Q C, ZUO C S, DOLAN-GAVITT B, et al. Automatic uncovering of hidden behaviors from input validation in mobile Apps[C]//Proceedings of 2020 IEEE Symposium on Security and Privacy (SP). 2020: 1106-1120.
- [23] SONG F, TOULILI T. Model-checking for android malware detection[M]//Programming Languages and Systems. Cham: Springer International Publishing, 2014: 216-235.
- [24] BAI G D, YE Q Q, WU Y Z, et al. Towards model checking Android applications[J]. IEEE Transactions on Software Engineering, 2018, 44(6): 595-612.
- [25] MERCALDO F, NARDONE V, SANTONE A, et al. Ransomware steals your phone. formal methods rescue it[M]//Formal Techniques for Distributed Objects, Components, and Systems. Cham: Springer International Publishing, 2016: 212-221.
- [26] MERCALDO F, NARDONE V, SANTONE A, et al. Download malware? No, thanks: how formal methods can block update attacks[C]//Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering. 2016: 22-28.
- [27] BATTISTA P, MERCALDO F, NARDONE V, et al. Identification of android malware families with model checking[C]//Proceedings of the 2nd International Conference on Information Systems Security and Privacy. 2016: 542-547.
- [28] CANFORA G, MARTINELLI F, MERCALDO F, et al. LEILA: formal tool for identifying mobile malicious behaviour[J]. IEEE Transactions on Software Engineering, 2019, 45(12): 1230-1252.
- [29] MILNER R. Communication and concurrency[M]. Prentice Hall, 1989.
- [30] BARBUTI R, DE FRANCESCO N, SANTONE A, et al. Selective mu-calculus and formula-based equivalence of transition systems[J]. Journal of Computer and System Sciences, 1999, 59(3): 537-556.
- [31] SURHONE L M, TENNOE M T, HENSSONOW S F, et al. Jimple[M]. Betascript Publishing, 2010.
- [32] SHEN F, VECCHIO J D, MOHAISEN A, et al. Android malware detection using complex-flows[C]//Proceedings of IEEE Transactions on Mobile Computing. 2017: 1231-1245.

[作者简介]



袁占慧（1997-），男，吉林辽源人，信息工程大学硕士生，主要研究方向为移动智能终端的信息流控制与隐私保护。



杨智（1975-），男，河南开封人，博士，信息工程大学副教授，主要研究方向为操作系统安全、云计算安全、隐私保护。



张红旗（1962-），男，河北遵化人，博士，信息工程大学教授、博士生导师，主要研究方向为网络安全、风险评估、等级保护和信息安全管理等。



金舒原（1974-），女，吉林白城人，博士，中山大学教授、博士生导师，主要研究方向为隐私保护、漏洞分析等。



杜学绘（1968-），女，河南新乡人，博士，信息工程大学教授、博士生导师，主要研究方向为空间信息网络、云计算安全。