南京邮电大学通达学院

毕业设计(论文)外文资料翻译

学　　院	计算机工程学院
专　　业	软件工程（嵌入式培养）
学生姓名	项伟伟
班级学号	18240125
外文出处	http://dx.doi.org/10.1145/2594291.2594299

指导教师评价：

1．翻译内容与课题的结合度：☐ 优　☐良　☐ 中　☐ 差

2．翻译内容的准确、流畅：☐ 优　☐ 良　☐ 中　☐ 差

3．专业词汇翻译的准确性：☐ 优　☐ 良　☐ 中　☐ 差

4．翻译字符数是否符合规定要求：☐ 符合　☐ 不符合

指导教师签名:

2022 年 月 日

FlowDroid:Android应用程序的精确上下文、流、场、对象敏感和生命周期感知污染分析

作者:

StevenArzt,SiegfriedRasthofer,ChristianFritz,EricBodden

ECSPRIDE 安全软件工程组

达姆施塔特工业大学

firstName.lastName@ec-spride.de

AlexandreBartel,JacquesKlein,andYvesLeTraon

安全、可靠性和信任跨学科中心

卢森堡大学

firstName.lastName@uni.lu

DamienOcteau,PatrickMcDaniel

计算机科学与工程系

宾夕法尼亚州立大学

{octeau,mcdaniel}@cse.psu.edu

摘要

今天的智能手机是私人和机密数据的普遍来源。与此同时，智能手机用户受到程序粗心的应用程序的困扰，这些应用程序会意外泄露重要数据，以及恶意应用程序利用他们的给定权限故意复制此类数据。虽然现有的静态污点分析方法有可能提前检测到此类数据泄漏，但所有适用于 Android 的方法都使用许多粗粒度近似，可能会产生大量漏失和误报。

因此，在这项工作中，我们提出了 FLOWDROID，这是一种针对 Android 应用程序的新颖且高度精确的静态污点分析。Android 生命周期的精确模型允许分析正确处理 Android 框架调用的回调，而上下文、流、字段和对象敏感性允许分析减少误报的数量。新颖的按需算法帮助 FLOWDROID 同时保持高效率和精度。

我们还提出了 DROIDBENCH，这是一个开放式测试组件，用于评估专门针对 Android 应用程序有效性和准确性的污点分析工具。正如我们通过使用 SecuriBenchMicro、DROIDBENCH 和一组著名的 Android 测试应用程序进行的一组实验所表明的那样：FLOWDROID 发现了非常高比例的数据泄漏，同时保持低误报率。在 DROIDBENCH 上，FLOWDROID 实现了 93%的召回率和 86%的准确率，大大优于商业工具 IBMAppScanSource 和 FortifySCA。FLOWDROID 成功地在来自 GooglePlay 的 500 个应用程序和来自 VirusShare 项目的大约 1,000 个恶意软件应用程序的子集中发现了漏洞。

类别和主题描述符：F.3.2[编程语言的语义]：程序分析;D.4.6[安全和保护]：信息流控制

PLDI’14,June9-112014,Edinburgh,UnitedKingdom.Copyrightc2014ACM978-1-4503-2784-8/14/06...$15.00.

http://dx.doi.org/10.1145/2594291.2594299

1. 简介

根据最近的一项研究[9]，Android 在手机市场的市场份额不断增长，目前已达到 81%。随着 Android 手机无处不在，它们成为攻击用户隐私敏感数据的有价值的目标。毛毡等人。对不同类型的 Android 恶意软件[12]进行了分类，发现恶意 Android 应用程序构成的主要威胁之一是隐私侵犯，它将位置信息、联系人数据、图片、SMS 消息等敏感信息泄露给攻击者。但即使是非恶意且经过精心编程的应用程序也可能遭受此类泄漏，例如当它们包含广告库时[16]。许多应用程序开发人员包含此类库是为了获得一些报酬，但很少有人完全了解其隐私含义，也无法完全控制这些库处理哪些数据。公共图书馆提取用于识别目标广告的个人的私人信息，例如唯一标识符（例如，IMEI、MAC 地址等）、国家或位置信息。

污点分析通过分析应用程序并将潜在的恶意数据流呈现给人类分析师或自动恶意软件检测工具来解决这个问题，然后这些工具可以确定泄漏是否真的构成违反策略的行为。这些方法通过应用程序跟踪敏感的“污染”信息，方法是从预定义的源（例如，返回位置信息的 API 方法）开始，然后跟踪数据流，直到它到达给定的接收器（例如，方法将信息写入套接字），提供有关哪些数据可能在何处泄漏的精确信息。分析可以动态和静态地检查应用程序。但是，动态程序分析需要多次测试运行才能达到适当的代码覆盖率。此外，当前的恶意软件可以在分析的应用程序执行时识别动态监视器，从而导致应用程序在这些情况下伪装成良性程序。

虽然静态代码分析不存在这些问题，但它们存在不精确的风险，因为它们需要从程序输入中抽象出来并近似运行时对象。运行时执行的精确建模对于 Android 应用程序来说尤其具有挑战性，因为这些应用程序不是独立的应用程序，而是实际上是 Android 框架的插件。应用程序由具有不同生命周期的不同组件组成。在应用程序执行期间，框架调用应用程序内的不同回调，通知它系统事件，可以启动/暂停/恢复/关闭应用程序等[17]。为了能够有效地预测应用程序的控制流，静态分析不仅必须对该生命周期进行建模，还必须为系统事件处理（例如，对于 GPS 等手机传感器）、UI 交互等集成进一步的回调。正如我们在这项工作中所展示的，识别回调绝非易事，需要专门的算法。另一个挑战来自敏感信息的来源，例如用户界面中的密码字段。仅基于程序代码无法检测到返回其内容的各个 API 调用。相反，它们的检测需要存储在清单和布局 XML 文件中的辅助信息模型。最后但同样重要的是，与任何用 Java 编写的应用程序一样，Android 应用程序也包含走样和虚拟调度结构。Java 的典型静态分析通过某种程度的上下文和对象敏感性来处理这些问题。Android 的框架特性使得这个问

题比平常更难，因为我们发现它暴露了非常深的重叠关系。

Android[14,15,24,40]过去的数据流分析方法使用粗粒度的过近似和欠近似以不令人满意的方式处理上述挑战。通常由于缺乏可靠的生命周期模型而导致的近似不足，可能会导致这些分析错过重要的数据流。然而，在实践中更糟糕的是，工具的过度近似，会导致许多报错，很容易让安全分析师不堪重负，以至于他们完全停止使用分析工具。

因此，在这项工作中，我们提出了 FLOWDROID，这是一种专门为 Android 平台量身定制的新型静态污点分析系统，它基于新颖的按需算法，可在保持可接受性能的同时产生高精度。FLOWDROID 分析应用程序的字节码和配置文件，以发现潜在的隐私泄漏，无论是由于粗心还是出于恶意。与早期的分析相反，FLOWDROID 是第一个完全上下文、流、字段和对象敏感的静态污点分析系统，同时精确地建模完整的 Android 生命周期，包括正确处理回调和应用程序中用户定义的 UI 小部件。这种设计最大限度地提高了精度和召回率，即旨在最大限度地减少遗漏泄漏和错误警告的数量。为了在保持可接受的性能的同时获得深度上下文和对象敏感性，FLOWDROID 使用了一种新颖的按需走样分析。分析算法受到 Andromeda[37]的启发，但在精度方面比 Andromeda 有所改进。我们在 2013 年夏天开源了 FLOWDROID。该工具已经被多个研究小组采用，我们正在与一家领先的反病毒工具生产商联系，他们计划在分析后端高效地使用 FLOWDROID。

为了让我们和其他人能够衡量这一重要研究领域的科学进展，研究人员必须能够对 Android 污点分析工具进行比较研究。不幸的是，到目前为止，还没有可以进行系统研究的基准。作为这项工作的另一个贡献，我们因此提供了 DROIDBENCH，这是一个新颖的开源微基准套件，用于比较 Android 污点分析的有效性。我们已经在 2013 年春季在线提供了 DROIDBENCH，并且知道有几个研究小组已经使用它来衡量和提高他们的 Android 分析工具的有效性[19]。第一组外部研究人员已经同意为套件贡献更多的微基准[35]。

FLOWDROID 可用于保护内部开发的 Android 应用程序以及协助对 Android 恶意软件进行分类。这两个用例都不完美但误报率和误报率相当低。一组使用 SecuriBenchMicro、DROIDBENCH 和一些包含数据泄漏的知名应用程序的实验表明，FLOWDROID 发现了非常高比例的数据泄漏，同时保持低误报率。在 DROIDBENCH1.0 上，FLOWDROID 实现了 93%的召回率和 86%的准确率，大大优于商业工具 AppScanSource[2]和 FortifySCA[3]。对真实应用程序的进一步实验证实了 FLOWDROID 在实践中的实用性。

总而言之，这项工作提出了以下原始贡献：

- FLOWDROID，第一个完全上下文、字段、对象和流敏感的污点分析，它考虑了

Android 应用程序生命周期和 UI 小部件，并且具有按需走样分析的新颖、特别精确的变体

- FLOWDROID 完全开源
- DROIDBENCH，一个新颖、开放和全面的 Android 流分析微型基准套件
- 一组实验证实了与商业工具 AppScanSource 和 FortifySCA 相比，FLOWDROID 具有更高的精度和召回率
- 一组实验将 FLOWDROID 应用于来自 GooglePlay 的 500 多个应用程序和来自 VirusShare 项目[1]的大约 1000 个恶意软件应用程序。

我们在线提供我们作为开源项目的完整实施，以及所有基准和脚本以重现我们的实验结果：

http://sseblog.ec-spride.de/tools/flowdroid/

因为空间限制使我们无法包含完全重现我们的方法所必需的一些细节。因此，我们发布了一份随附的技术报告[13]，其中正式确定了 FLOWDROID 的传递函数并提供了有关实施的更多细节。

接下来的论文中。第 2 节给出了一个启发性的例子，并解释了 Android 安全的必要背景。第 3 节解释了 FLOWDROID 如何对 Android 生命周期进行建模，而第 4 节给出了有关实际污点分析的重要细节。在第 5 节中，本文讨论了实现细节和限制，而第 6 节评估了 FLOWDROID。第 7 节讨论相关工作，第 8 节总结。

2. 背景和示例

我们首先给出一个启发性的例子，然后解释这项工作假设的攻击者模型。清单 1 中的示例（摘自一个真实的恶意软件应用程序[42]）实现了一个 Activity，它在 Android 中表示用户界面中的一个屏幕。每当框架重新启动应用程序时，应用程序都会从文本字段（第 5 行）读取密码。当用户单击活动的按钮时，密码将通过 SMS 发送（第 24 行）。这构成了从密码字段（源）到 SMSAPI（接收器）的受污染数据流。在此示例中，sendMessage()与应用程序 UI 中的按钮相关联，当用户单击该按钮时会触发该按钮。在 Android 中，侦听器要么直接在代码中定义，要么在布局 XML 文件中定义，如此处所假设的。因此，仅分析源代码是不够的——分析还必须处理元数据文件以正确关联所有回调方法。在此代码中，仅当在 sendMessage()执行之前调用 onRestart()（初始化用户变量）时才会发生泄漏。为避免误报，

污点分析必须正确建模应用程序生命周期，识别用户确实可能在应用程序重新启动后点击按钮。

为了避免误报，对这个例子的分析必须是字段敏感的：用户对象包含用户名和密码两个字段，但只有后者应该被认为是私有值。虽然此示例不需要对象敏感性，但它对于区分源自不同分配站点但到达相同代码位置的对象至关重要。在我们的实验中，我们发现了一些需要深度对象敏感度才能自动消除误报的情况。这是由于 Android 框架的调用和分配链相对较深。

诸如字符串连接之类的操作需要一个模型来定义数据是否以及如何流经这些操作。将此类操作视为普通方法调用并分析应用程序代码之类的库方法可能是不精确的（因为它忽略了操作的语义），而且正如我们发现的那样，在实践中这通常代价高昂。

```
1  public class LeakageApp extends Activity{
2  private User user = null;
3  protected void onRestart(){
4    EditText usernameText =
       (EditText)findViewById(R.id.username);
5    EditText passwordText =
       (EditText)findViewById(R.id.pwdString);
6    String uname = usernameText.toString();
7    String pwd = passwordText.toString();
8    if(!uname.isEmpty() && !pwd.isEmpty())
9      this.user = new User(uname, pwd);
10 }
11 //Callback method in xml file
12 public void sendMessage(View view){
13   if(user == null) return;
14   Password pwd = user.getpwd();
15   String pwdString = pwd.getPassword();
16   String obfPwd = "";
17   //must track primitives:
18   for(char c : pwdString.toCharArray())
19     obfPwd += c + "_"; //String concat.
20
21   String message = "User: " +
22      user.getName() + " | Pwd: " + obfPwd;
23   SmsManager sms = SmsManager.getDefault();
24   sms.sendTextMessage("+44 020 7321 0905",
25     null, message, null, null);
26 }
```

代码 1 示例 Android 应用程序

攻击者模型 FLOWDROID 通常可用于检测数据流，无论它们是由粗心还是恶意意图引起的。对于恶意案例，我们假设以下攻击者模型。攻击者可能会向应用程序提供任意恶意 Dalvik 字节码。通常，攻击者的目标是通过用户授予的一组危险的广泛权限来泄露私人数据[4]。FLOWDROID 对安装环境和应用程序输入做出了合理的假设，这意味着攻击者也可以随意篡改这些内容。然而，FLOWDROID 确实假设攻击者无法绕过 Android 平台的安全措施或利用侧通道。此外，我们假设攻击者不使用隐式流[20]来掩饰数据泄漏。鉴于当前可用的恶意软件类型，这是一个非常合理的假设。

3. 生命周期的精确建模

下面我们解释 FLOWDROID 对生命周期的精确建模，包括入口、异步执行的组件和回调。

多入口 与 Java 程序不同，Android 应用程序没有 main 方法。相反，应用程序包含许多入口点，即由 Android 框架隐式调用的方法。Android 操作系统为应用程序中的所有组件定义了完整的生命周期。应用程序开发人员可以定义四种不同类型的组件：活动是单一关注的用户操作，服务执行后台任务，内容提供者定义类似数据库的存储，广播接收器侦听全局事件。所有这些组件都是通过从预定义的操作系统类派生一个自定义类，在 AndroidManifest.xml 文件中注册并覆盖来实现的。

有生命周期的方法 Android 框架调用这些方法来启动或停止组件，或者暂停或恢复它，具体取决于环境需要。例如，它可以因为内存耗尽而停止应用程序，然后在用户返回时重新启动它[17]。因此，在构建调用图时，Android 分析不能简单地从检查预定义的“main”方法开始。相反，必须对 Android 生命周期中所有可能的转换进行精确建模。为了解决这个问题，FLOWDROID 构建了一个自定义的虚拟主方法来模拟生命周期。下面我们解释这个方法是如何构造的。

异步执行组件 一个应用程序可以包含多个组件，例如，三个活动和一个服务。尽管活动按顺序进行，但无法预先确定它们的顺序。例如，一项活动可能是最初对用户可见的主要活动，然后根据用户输入启动其他活动之一。服务作为并行后台任务运行。FLOWDROID 通过假设应用程序内的所有组件（活动、服务等）可以以任意顺序（包括重复）运行来模拟此执行。一些静态分析是路径敏感的，即分别考虑每个可能的程序路径。在这种情况下，考虑所有可能的排序将非常昂贵。FLOWDROID 的分析基于 IFDS[32]，这是一个对路径不敏感的分析框架，而是在任何控制流合并点立即加入分析结果。因此，FLOWDROID 可以生成并有效地分析一个虚拟主方法，其中各个组件生命周期和回调的每个顺序都是可能的；它不需要遍历所有可能的路径。

回调 Android 操作系统允许应用程序为各种类型的信息注册回调，例如位置更新或 UI 交互。FLOWDROID 在其虚拟 main 方法中对这些回调进行建模，例如，识别应用程序存储框架作为参数传递给回调的位置数据，然后在活动停止时将此数据发送到 Internet 的情况。通常无法预测调用回调的顺序，这就是为什么 FLOWDROID 假设所有回调都可以以任何可能的顺序调用的原因。但是，回调只能在父组件（例如活动）运行时发生。为了精确起见，

FLOWDROID 将组件（活动、服务等）与它们注册的回调相关联。例如，一个活动可以注册在按下按钮时调用的回调。然后，必须仅在此活动的 onResume()和 onPause()事件之间分析相应的回调处理程序。

在 Android 平台上注册回调处理程序有两种不同的方法。首先，可以在 activity 的 XML 文件中以声明方式定义回调。或者，它们也可以使用对特定系统方法的众所周知的调用来强制注册。FLOWDROID 支持这两种方式。此外，对于恶意软件，攻击者可能会通过覆盖 Android 基础设施的方法来注册未记录的回调，其中一些甚至可以被本机代码调用。FLOWDROID 识别这些被覆盖的方法，处理它们类似于正常的回调处理程序，例如按钮单击。

为了查找在应用程序代码中注册的回调，FLOWDROID 首先计算每个组件的一个调用图，从各个组件类中实现的生命周期方法（onCreate()、onStop()等）开始。然后，此调用图被用于扫描使常用的回调接口之一作为形式参数类型的 Android 系统方法的调用。之后，调用图被增量扩展以包含这些新发现的回调，并且再次运行扫描，因为回调处理程序可以自由地自行注册新的回调，可能需要 FLOW-DROID 重新扩展调用图并重新分析直到它到达一个固定点。虽然这种方法比仅仅扫描实现回调接口的类更昂贵，但它在组件和回调之间提供了更精确的映射。这不仅减少了误报，而且我们还发现它大大减少了以下污点分析的运行时间。一旦构建了虚拟 main 方法，FLOWDROID 就会使用此方法作为应用程序的入口点来计算最终调用图。

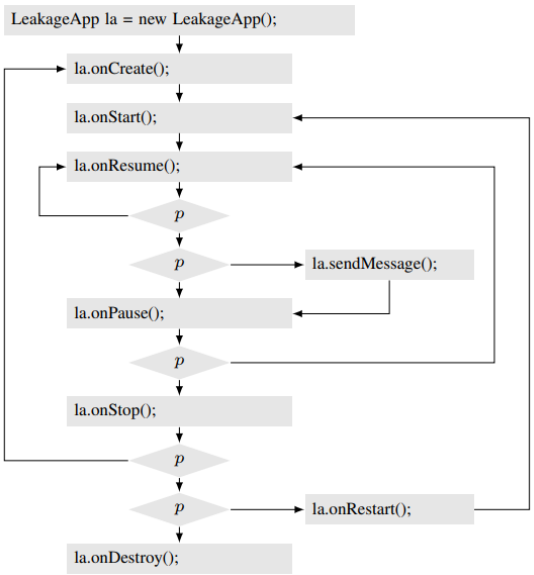

图 1 虚拟 Main 方法的 CFG

对于布局 XML 文件中定义的回调，相应的 XML 文件使用相应的布局控件映射到一个

或多个应用程序组件。例如，按钮单击处理程序仅对承载相应按钮的活动有效。FLOWDROID 分析每个活动以查看它注册的 XML 文件中的哪些标识符。然后使用此信息来创建映射。

举例 请注意，为了获得最大精度，FLOWDROID 为每个分析的应用程序生成一个新的虚拟主方法。根据应用程序的 XML 配置文件，每个主要方法将只涉及生命周期中实际可以在运行时发生的部分。禁用的活动会被自动过滤，回调方法只会在它们实际所属的组件的上下文中被调用。例如，按钮单击处理程序仅在其各自活动的上下文中进行分析。在图 1 中，我们显示了前面示例的虚拟 main 方法的控制流程图。该图模拟了一个通过 sendMessage 回调增强的通用活动生命周期。在这个图中，p 表示一个不透明的谓词，我们知道 FLOWDROID 将无法对其进行静态评估。结果，分析将自动以相等的条件考虑涉及 p 的条件的两个分支。

4. 精确的流量敏感度分析

分析中的一个主要困难是如何实现高对象敏感度以有效解决走样。图 2（从真实案例中提取）显示了 FLOWDROID 如何结合前向污染分析和按需后向走样分析来推断 b.f 在接收器处受到污染。在步骤 1 中，被污染的变量 w 向前传播，污染了堆对象 x.f。步骤 2 继续对 w 和 x.f 进行污点跟踪。重要的步骤是 3：每当堆对象被污染时，向后分析向上搜索相应对象的走样（在本例中为 x.f）。在 7 处，走样 b.f 被发现，然后作为正常污点向前传播。

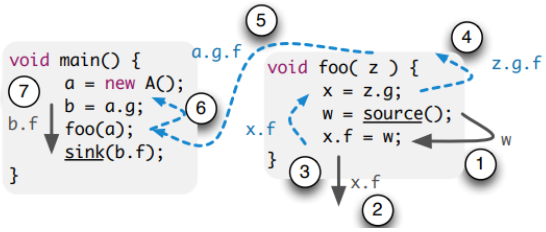

图 2 真实走样下的污点分析

FLOWDROID 对 IFDS[32]框架内的污点分析问题进行建模，以解决程序间子问题的分配。4.1 节解释了分析使用的传递函数。大多数功能都是相对标准的。然而，在一种重要情况下，FLOWDROID 的分析不同于标准污染分析算法，即在将污染值分配给堆（即字段或数组）的语句中。这种情况会导致反向走样分析被调用，细节我们将在 4.2 节解释。由于篇幅限制，我们将流函数的描述保留在非正式的层面。为了让其他人重现我们的方法，随附的技术报告包含完整的形式化[13]

4.1 污点分析

前向和后向分析都传播访问路径。访问路径的格式为 x.f.g，其中 x 是局部变量或参数，f 和 g 是字段。访问路径可以有不同的长度，最多可达用户可自定义的最大长度（默认为 5）。长度为 0 的访问路径是一个简单的局部变量或参数，例如 x。在 FLOWDROID 中，访问路径隐含地描述了通过该路径可到达的所有对象的集合，例如，x.f 包括污点 x.f.g、x.f.h、x.f.g.h 等。

如果右侧的任何操作数被污染，则分配的传递函数会污染左侧。通过污染整个数组来保守地处理对数组元素的赋值。将“新”表达式分配给变量 x 会清除所有由以 x 为根的访问路径建模的污点。方法调用通过将实际参数替换为形式参数来将访问路径转换为被调用者的上下文；反向转换发生在方法返回时，包括返回值（如果存在）。与基于 IFDS 的分析一样，FLOWDROID 还包括一个调用返回流函数（绕过调用方的每个方法调用）。此函数传播与调用无关的污点，在源处生成新的污点，在接收器处报告污点并传播本地调用的污点。第 5 节提供了关于后者的更多信息。

4.2 按需走样分析

每当将污染值分配给堆位置（例如字段或数组）时，FLOWDROID 都会向后搜索目标变量的走样，然后也污染它们。在代码 2 中，现考虑第一次调用 taintIt（第 3 行），这会污染形式参数 in。在第 10 行，这将导致访问路径 xf 由于赋值 xf=in 而被污染。在这种情况下（通常在所有分配给堆的情况下），FLOWDROID 将开始向后搜索 xf 的走样，在第 9 行找到 out.f。此时，从该语句开始为 out.f 开始新的正向污染传播，最终将在第 11 行发现泄漏。不过，后向分析还将继续向后搜索，在 main 中发现走样 pf，然后它会产生前向分析，从而在第 4 行产生第二个污染流报告。

保持上下文敏感 算法 1 和 2 以伪代码显示了前向和后向分析求解器的主循环。

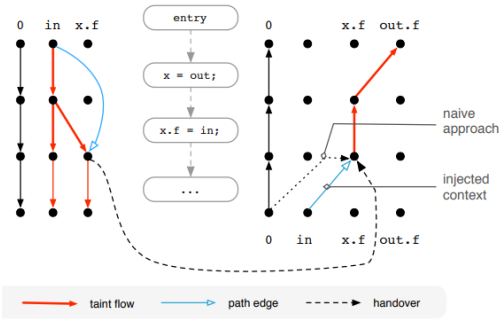

图 3 在 **taintIt** 中

使用上下文注入进行分析切换

```
1  void main() {
2    Data p = new ..., p2 = new ...
3    taintIt(source(), p);
4    sink(p.f);
5    taintIt("public", p2);
6    sink(p2.f);
7  }
8  void taintIt(String in, Data out) {
9    x = out;
10   x.f = in;
11   sink(out.f);
12 }
```

代码 3 上下文注入的示例

算法 1 前向求解器的主循环

while $WorkList_{FW} \neq \emptyset$ **do**
  pop $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$ off $WorkList_{FW}$
  switch (n)
  case n is call statement:
    if summary exists for call **then**
      apply summary
    else
      map actual parameters to formal parameters
    end if
  case n is exit statement:
    install summary $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$
    map formal parameters to actual parameters
    map return value back to caller's context
  case n is assignment $lhs = rhs$:
    $d_3 :=$ replace rhs by lhs in d_2
    insert $\langle s_p, d_1\rangle \rightarrow \langle n, d_3\rangle$ into $WorkList_{BW}$
  extend path-edges via the *propagate*-method of the classical IFDS algorithm
end while

算法 2 后向求解器的主循环

while $WorkList_{BW} \neq \emptyset$ **do**
  pop $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$ off $WorkList_{BW}$
  switch (n)
  case n is call statement:
    if summary exists for call **then**
      apply summary
    else
      map actual parameters to formal parameters
    end if
    extend path-edges via the *propagate*-method of the classical IFDS algorithm
  case n is method's first statement:
    install summary $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$
    insert $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$ into $WorkList_{FW}$
    do *not* extend path-edges via the *propagate*-method of the classical IFDS algorithm, killing current taint d_2
  case n is assignment $lhs = rhs$:
    $d_3 :=$ replace lhs by rhs in d_2
    insert $\langle s_p, d_1\rangle \rightarrow \langle n, d_3\rangle$ into $WorkList_{FW}$
    extend path-edges via the *propagate*-method of the classical IFDS algorithm
end while

```
1 Data p = new ..., p2 = p;
2 sink(p2.f);
3 p.f = source();
4 sink(p2.f);
```

代码 2 激励语句示例

算法表示假设读者熟悉原始 IFDS 算法的算法描述[32]。两个求解器都在自己的工作列表上运行，包含所谓的路径边，这些路径边汇总了到目前为止计算的数据流，直到当前语句/节点 n。一条边(sp,d1)→(n,d2)有效地表明分析得出结论:如果 d1 在 n 的过程 p 的起点 sp 成立，则 d2 在 n 成立。在我们的特定实现中，抽象域值 di 是描述对污染值的引用的有效访问路径。两种分析之间的切换非常重要。如果以一种简单的方式进行协调，人们将很容易获得两个独立的分析，每个分析本身可能是上下文敏感的，但会结合起来产生沿着冲突上下文的不可实现路径的分析信息。例如，请注意，在清单 2 中的示例中，只有当 in 之前被污染时，x.f 的联系才会被污染。特别是，分析不应在第 6 行报告泄漏，其对应的 taintIt-call 仅传播字符串“public”。

在我们的特定实现中，抽象域值 di 是有效的访问路径，描述了对受污染值的引用。两种分析之间的切换非常重要。如果以一种简单的方式进行协调，人们将很容易获得两个独立的分析，每个分析本身可能是上下文敏感的，但会结合起来产生沿着冲突上下文的不可实现路径的分析信息。例如，请注意，在清单 2 的示例中，x.f 的走样只有在 in 之前被污染时才会被污染。特别是，分析不应在第 6 行报告泄漏，其对应的 taintIt-call 仅传播字符串“public”。

图 3 显示了一个简单的实现如何导致这种误报，以及 FLOWDROID 如何通过将上下文从一个分析注入到另一个分析来处理问题。该图假设您熟悉 IFDS 框架内流函数的典型符号[32]。此处黑色节点表示相应语句之前/之后的数据流事实，黑色和红色边缘表示数据流。事实 0 是始终正确的重言式事实，这就是为什么一个 0 节点总是连接到下一个节点的原因。图的左侧显示了正向污染分析如何确定 x.f 被污染。在处理对 x.f 的分配时，前向分析会产生一个后向走样分析的实例，如右侧所示。产生这种分析的简单方法是用从 0 到 x.fx.f（虚线）的边对其进行初始化。这种实现虽然简单，但会导致不精确，因为它的语义表明 x.f 的走样无论如何都会被污染。在代码 2 中，这可能导致分析错误地报告污染违规，即使对于 p2.f 也是如此。因此，正确的方法是将前向分析的上下文注入到后向分析中：FLOWDROID 参考 x.f 的“路径边缘”，IFDS 算法将其存储为汇总计算的副作用。然后它将整个边缘注入到反向求解器中。（参见算法 1，第 16 行）上下文注入是双向发生的。在示例中的第 9 行，当反向分析生成 out.f 的正向分析时，它会将原始上下文注入到正向分析中。（参见算法 2，第 17 行）从语义上讲，对于示例，这意味着所有污染两种分析都发现 taintIt 是有条件的在最初被污染。

第二个问题是避免由于无法实现的路径导致的误报：FLOWDROID 需要防止向后分析返回到前向分析未分析的上下文中（反之亦然）。为了实现这个约束，FLOWDROID 中的后

向分析实际上根本不会返回到调用者。相反，每当找到走样时，它都会触发对该走样的前向分析，例如第 9 行中的 forout.f。然后，前向分析的任务是将任何相关的污点映射回调用者的上下文。在示例中，前向分析知道它源自的调用上下文，这就是为什么它可以轻松确保仅将污点映射回正确的上下文。在该示例中，前向传递将仅在第 3 行中将 out.f 映射到 p.f，而不是在第 5 行中映射到 p2.f。本质上，后向分析可以下降到被调用者，但永远不会返回到调用者；所有退货均由前向分析处理。当向后分析下降到调用时，它最终会在到达方法头时产生前向分析。（参见算法 2，第 13 行）然后，前向分析可以确保只返回到正确的调用者，因为它的上下文是由后向分析注入的。从技术上讲，它的传入集[26]被注入）每当前向分析将与堆对象关联的污点映射回调用者时，它都会在调用者内部产生一个新的走样搜索。

保持流敏感 Andromeda[37]是另一个启发 FLOWDROID 按需走样分析的污点分析工具。然而，Andromeda 的分析可能会导致对流量不敏感的结果。在代码 3 的示例中，分析将在第 2 行和第 4 行报告两个泄漏，即使对 sink 的第一次调用肯定发生在 p2.f 被污染之前。事实上，同样的问题也适用于我们上面描述的 FLOWDROID 的分析：反向分析会在第 1 行发现受污染的走样 p2.f 并触发具有该值的前向传递，从而导致此后在任何地方报告污染 p2.f 泄露的地方。

FLOWDROID 通过跟踪我们称之为激活语句的内容来解决这个问题。每当产生一个反向走样分析的实例时，相应的访问路径都会增加当前语句，即走样的激活语句。此外，受污染的走样被标记为非活动状态。从语义上讲，只有活跃的污点在到达水槽时才会导致泄漏。非活动污点是尚未被污染的内存位置的走样。每当反向分析再次产生正向分析时，当正向分析在其激活语句上传播走样污点时，污点就会被激活，从而获得实际导致报告泄漏的能力。在该示例中，激活语句位于第 3 行，因此导致分析仅在随后的第 4 行报告泄漏，从而避免了第 2 行的错误警报。

一般来说，激活语句是调用树的代表。暂时假设代码 3 中的堆分配包含在方法调用中，例如代码 2 第 10 行的分配就是这种情况，它发生在对 taintIt 的方法调用中。在该示例中，当前向分析将返回边处理回 main 的第 3 行时，分析会将对 taintIt（第 3 行）的调用与激活语句全局关联，因为每当此调用完成时，激活语句也已被处理并且因此污点将被激活。换句话说，激活语句用于查找它们发生的调用树，以将它们转换回（传递）调用者。

据我们所知，FLOWDROID 是第一种实现按需分析的方法，该分析完全保持上下文和流量敏感性。未来我们计划研究这里解释的原则在多大程度上可以在污点分析范围之外重用，理想情况下产生一个相当通用的 IFDS 扩展。

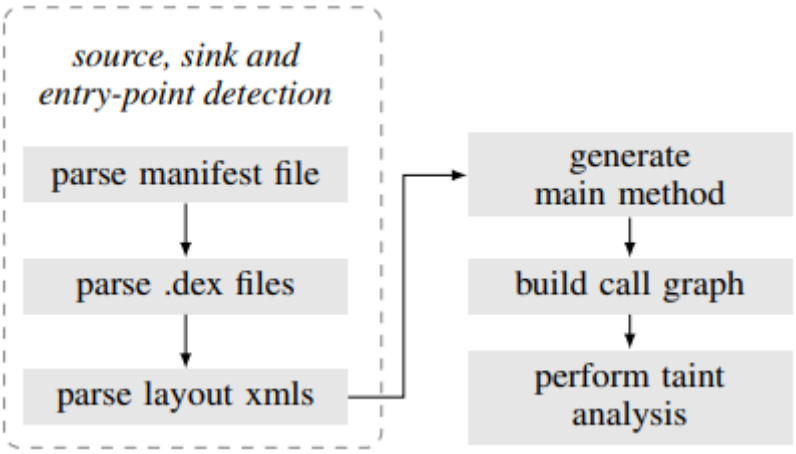

图 4FLOWDROID 的概述

更多算法细节 我们的 IFDS 实现使用了 Naeem 和 Lhotak[26]解释的扩展。有了这个扩展，IFDS 实现可以即时计算程序的超级图，这意味着在我们的例子中，我们只计算那些确实被污染的变量/访问路径的污染信息。我们的实现使用了 IFDS 求解器的两个实例，每个实例都进行了轻微的调整，如算法 1 和 2 中所述。每个实例都包含一个单独的汇总函数表，与原始 IFDS 算法一样，用于避免重新计算相同上下文下的相同被调用者 di。

5. 实现

FLOWDROID 扩展了 Soot 框架[21]，它为精确分析提供了重要的先决条件，特别是三地址代码中间表示 Jimple 和精确的调用图分析框架 Spark[22]。一个名为 Dexpler[5]的插件允许 FLOWDROID 将 Android 的 Dalvik 字节码转换为 Jimple。在 Soot 和 Dexpler 之上，FLOWDROID 进一步使用了 Heros[7]，这是 IFDS 框架[32]的可扩展、高度多线程的实现。我们接下来解释 FLOWDROID 的架构，而后续部分解释重要的实现细节和 FLOWDROID 当前的限制。

架构 第四节展示了 FLOWDROID 的架构。Android 应用程序打包在 apk 文件（Android 包）中，这些文件本质上是 zip 压缩文件。解压缩文件后，FLOWDROID 会在应用程序中搜索生命周期和回调方法以及对源和接收器的调用。这是通过解析各种 Android 特定文件来完成的，包括布局 XML 文件、包含可执行代码的 dex 文件以及定义应用程序中的 Activity,Service、BroadcastReceiver、ContentProvider 的代码文件。接下来，FLOWDROID 从生命周期和回调方法列表中生成虚拟 main 方法。然后使用此主要方法生成调用图和过程间控制流图(ICFG)。从检测到的源开始，污点分析然后通过遍历 ICFG 来跟踪污点，如第 4 节

所述。FLOWDROID 配置有我们的 SuSi 项目[30]推断的源和终，这是迄今为止最全面的可用项目。源和汇的具体列表可从 FLOWDROID 网站获得。最后，FLOWDROID 报告所有发现的从源到汇的流。报告包括完整路径信息。为了获得这些信息，实现将数据流抽象对象链接到它们的前辈和它们的生成语句。这允许 FLOWDROID 的报告组件完全重建可能在给定接收器处导致污染从而违背的所有相关分配语句的图表。

定义快捷方式 在分析中包含完整的 JRE 或 Android 平台运行时不仅需要大量的分析时间和内存，而且由于在库分析期间执行的近似，还会导致不希望的不精确性。因此，FLOWDROID 包含一个用于外部库模型的接口。该工具支持用于定义某些“shortcurules”的简单文本文件格式。预定义规则处理集合类、字符串缓冲区和类似的常用数据结构，例如，指定将污染元素添加到集合会污染整个集合。从技术上讲，快捷方式是使用 call-to-return 边缘实现的。如果库调用没有关联规则，则对其进行全面分析

本地调用 Java 和 Android 平台都支持调用 C 或其他非托管语言编写的本机方法。对于基于 Java 的分析，这些方法是无法分析的黑盒。FLOWDROID 为最常见的本机方法（例如“System.arraycopy”）提供了明确的污点传播规则。在此示例中，如果第一个参数（输入数组）在调用之前被污染，则规则定义第三个参数（输出数组）被污染。对于没有明确规则的本地方法，FLOWDROID 假定一个合理的默认值：如果之前至少有一个参数被污染，则调用参数和返回值被污染。这既不完全合理，也不最精确，但可能是黑盒设置中最好的实用近似值。

组件间通信 FLOWDROID 通过将发送意图的方法视为接收器和将接收意图的回调视为源来过度近似显式的组件间通信。Android 还支持基于意图的隐式通信，例如，通过设置被调用活动的结果值，然后由操作系统自动将其传递回调用活动。支持这种行为以及更准确的组件间连接模型留待未来工作。特别是，我们正在努力将 FLOWDROID 与 EPICC[27]集成，这是一种新颖的静态分析，它使用 Soot 和 Heros 执行字符串分析，以更精确地解决应用间通信。

限制 尽管 FLOWDROID 通常旨在进行可靠的分析，但它确实与大多数其他静态分析工具有一些固有的局限性。例如，只有当参数是字符串常量时，FLOWDROID 才会解析反射调用，但情况并非总是如此。在 Java 平台上，可以使用 TamiFlex[8]等反射分析工具使静态分析工具了解在运行时发出的反射调用。不过，此类工具需要通过 java.lang.instrument 进行加载时检测，Android 平台目前不支持。如果 Android 生命周期包含我们不知道的回调，或者通过我们的规则模型不正确的本机方法，也会出现不健全的情况。目前，FLOWDROID 也忽略了多线程：它假定线程以任意但顺序的顺序执行，这通常也是不合理的。完全整合对

多线程的良好支持本身就是一个巨大的挑战，因此我们将其留给未来的工作

6. 实验评估

我们的评估涉及以下研究问题：

RQ1 FLOWDROID 在精度和召回率方面与 Android 的商业污点分析工具相比如何？

RQ2 FLOWDROID 能否找到 InsecureBank 中的所有隐私泄露，InsecureBank 是其他人专门设计用于挑战 Android[28]漏洞检测工具的应用程序，它的性能如何？

RQ3 FLOWDROID 能否在实际应用中发现泄漏？速度有多快？

RQ4 FLOWDROID 在应用于 Java 而非 Android 相关的污点分析问题时，在精度和召回率方面的表现如何？

接下来的部分将详细讨论每个研究问题。6.5 节解释了为什么我们无法直接将 FLOWDROID 与其他学术 Android 分析工具进行比较。

6.1 RQ1：商业污点分析工具

虽然有用于分析 Web 应用程序或专门用于检测不同类型 Java 漏洞的基准套件[23]，但目前还没有针对 Android 的分析基准套件。这是有问题的，因为通用 Java 测试套件不涵盖 Android 生命周期、回调或与密码字段等 UI 元素的交互等方面。因此，它们不能用于评估 Android 分析工具的实际有效性。

DroidBench 为此，我们专门针对这项工作开发了一个特定于 Android 的测试套件，称为 DROIDBENCH。在本次评估中，我们参考了 1.0 版，其中包含 39 个手工制作的 Android 应用程序。该套件可用于评估静态和动态污点分析，但特别是它包含有趣的静态分析问题（字段敏感性、对象敏感性、访问路径长度的权衡等）以及特定于 Android 的测试用例诸如正确建模应用程序的生命周期、充分处理异步回调以及与 UI 交互等挑战。我们的技术报告[13]提供了有关各个应用程序的更多信息。我们在 2013 年春季推出了在线 DROIDBENCH，并且知道有几个研究小组[19]已经使用它来衡量和提高他们的 Android 分析工具的有效性。第一组外部研究人员已经同意为套件贡献更多的微基准[35]

表 1 展示了 FLOWDROID 和两个商业分析工具（如下所述）应用于 DROIDBENCH 时的分析结果。如结果所示，FLOWDROID 实现了 93%的召回率和 86%的准确率。[1] 如前所述，出于性能原因，FLOWDROID 处理数组索引并不精确。同样的限制也适用于 ListAccess1，

导致第一类出现误报。准确有效地处理索引本身就是一个有趣的研究问题[10]。Button2 会导致误报，因为 FLOWDROID 目前不支持强更新。结果，它不能杀死某些按钮组合的污点。允许强更新需要（可能相当昂贵的）必须走样分析。将这样的分析纳入 FLOWDROID 超出了这项工作的范围。未检测到 IntentSink1，因为测试用例不包含实际接收器。相反，受污染的值存储在意图中，然后由框架返回给活动。这种情况如果不进行特殊处理很难处理。StaticInitialization1 失败是因为 Soot 当前假定所有静态初始化程序都在程序开始时执行，在这种情况下这是不正确的。确定此类初始化程序在运行时可以执行的确切位置是一个有趣的研究问题。我们计划在未来提供更好的支持。

与 IBMAppScanSource 的比较 在 DROIDBENCH 的所有测试中，我们将 FLOW-DROID 与 IBMAppScanSource[2]版本 8.7 进行了比较。AppScanSource 区分了三种不同的发现类别：漏洞、类型 1 的异常和类型 2 的异常。与 FLOWDROID 的报告一样，漏洞包括从源到接收器的完整路径。对于类型 1 异常，也存在从源到接收器的流，但沿传播路径的某些方法的语义是未知的（例如，可能的净化）。由于 FLOWDROID 目前不支持清理，我们将漏洞和 1 类异常都视为发现。另一方面，对于类型 2 异常，没有跟踪。当检测到某些代码结构（例如，将变量值写入日志文件）时，会生成这些报告。由于这些发现非常不精确并且完全无视数据流，因此我们不将它们视为发现。如表 1 所示，AppScanSource 仅发现大约 50%的泄漏。处理回调和 Android 组件时会出现主要问题。看起来对 Android 的广告支持主要限于配置了一些适当的源和接收器的 AppScan。AppScan 显示 74%的相对不错的精度。

[1] 我们排除了对由控制流依赖引起的隐式流[20]的分析，因为包括 FLOWDROID 在内的任何工具都没有被设计用于分析此类流

⊙ = correct warning, ⋆ = false warning, ○ = missed leak
multiple circles in one row: multiple leaks expected
all-empty row: no leaks expected, none reported

App Name	AppScan	Fortify	FlowDroid
Arrays and Lists			
ArrayAccess1			⋆
ArrayAccess2	⋆	⋆	⋆
ListAccess1	⋆	⋆	⋆
Callbacks			
AnonymousClass1	○	⊙	⊙
Button1	○	⊙	⊙
Button2	⊙ ○ ○	⊙ ○ ○	⊙ ⊙ ⊙ ⋆
LocationLeak1	○ ○	○ ○	⊙ ⊙
LocationLeak2	○ ○	○ ○	⊙ ⊙
MethodOverride1	⊙	⊙	⊙
Field and Object Sensitivity			
FieldSensitivity1			
FieldSensitivity2			
FieldSensitivity3	⊙	⊙	⊙
FieldSensitivity4	⋆		
InheritedObjects1	⊙	⊙	⊙
ObjectSensitivity1			
ObjectSensitivity2	⋆		
Inter-App Communication			
IntentSink1	⊙	⊙	○
IntentSink2	⊙	⊙	⊙
ActivityCommunication1	⊙	⊙	⊙
Lifecycle			
BroadcastReceiverLifecycle1	⊙	⊙	⊙
ActivityLifecycle1	⊙	⊙	⊙
ActivityLifecycle2	○	⊙	⊙
ActivityLifecycle3	○	○	⊙
ActivityLifecycle4	○	⊙	⊙
ServiceLifecycle1	○	○	⊙
General Java			
Loop1	⊙	○	⊙
Loop2	⊙	○	⊙
SourceCodeSpecific1	⊙	⊙	⊙
StaticInitialization1	○	⊙	○
UnreachableCode		⋆	
Miscellaneous Android-Specific			
PrivateDataLeak1	○	○	⊙
PrivateDataLeak2	⊙	⊙	⊙
DirectLeak1	⊙	⊙	⊙
InactiveActivity	⋆	⋆	
LogNoLeak			
Sum, Precision and Recall			
⊙, higher is better	14	17	26
⋆, lower is better	5	4	4
○, lower is better	14	11	2
Precision $p = ⊙/(⊙ + ⋆)$	74%	81%	86%
Recall $r = ⊙/(⊙ + ○)$	50%	61%	93%
F-measure $2pr/(p + r)$	0.60	0.70	0.89

表格 1 DROIDBENCH 测试结果

与 FortifySCA 的比较 HP 的 FortifySCA[3]是安全分析师广泛使用的另一种商业工具。与 IBMAppScanSource 类似，Fortify 也提供了不同类型的发现，例如从敏感源到公共接收器的数据流、对安全敏感权限的请求、对安全敏感方法的调用等。在我们的评估中，我们只考

虑了关于数据流。所有测试均使用 5.14 版本进行。如表 1 所示，FortifySCA 显示了与 IBMAppScan 类似的问题，例如 Android 组件生命周期和回调的处理。图 1 显示 Fortify 在生命周期测试中检测到 6 个数据泄漏中的 4 个，但仔细检查表明这只是偶然发生的。在这些测试中，数据源涉及一个静态字段，Fortify 显然以一种特殊的方式处理该字段，巧合地导致报告泄漏。当移除不会改变测试用例语义的静态修饰符时，Fortify 不再检测泄漏。Fortify 的精确度为 81%。

结论 从实验中，我们得出结论，为了不给用户带来过多的误报负担，AppScanSource 和 FortifySCA 的目标是在牺牲召回率的同时获得相对较高的精度，从而冒着错过实际隐私泄露的风险。相比之下，FLOWDROID 显示出显着更高的召回率，甚至略微提高了精度。

6.2 RQ2：InsecureBank 的性能

InsecureBank [28] 是由 Paladion Inc. 创建的一个易受攻击的 Android 应用程序，专门用于评估 FLOWDROID 等分析工具。它包含与现实世界应用程序中发现的漏洞和数据泄漏类似的各种漏洞。在默认设置为 Oracle 的 Java Runtime 版本 1.7（64 位）的 Windows 7 上运行 Intel Core 2 Centrino CPU 和 4 GB 物理内存的笔记本电脑上分析应用程序大约需要 31 秒。FLOWDROID 发现了我们都手动验证的所有七个数据泄漏。没有假阳性或假阴性。

6.3 RQ3：实际应用程序的性能

为了在真实应用程序上评估 FLOWDROID，我们将其应用于来自 Google Play 的 500 个最受欢迎的 Android 应用程序。[2] 幸运的是，尽管 FLOWDROID 的召回率很高，但分析并未发现任何暗示真正恶意行为的泄漏。尽管如此，据报道，大多数应用程序（可能是意外）将 IMEI（唯一 ID）或位置数据等敏感信息泄露到日志和偏好文件中。

例如，三星的推送服务会记录手机的 IMEI。日志是有问题的，因为操作系统对日志的访问限制与对文件的访问限制不同：对于运行 Android 4.0 或更低版本的设备，任何具有 READ LOGS 权限的应用程序都可以读取所有日志。这个问题被认为非常重要，以至于在 Android 4.1 中改变了机制。从这个版本开始，日志只对私人可见，除非应用程序在启用调试的情况下运行。此外，三星的推送服务还广播包含 IMEI 的 Android 意图。所有其他应用程序都可以简单地订阅这个意图并获得广播 IMEI ，从而绕过 Android 对该数据项的权

限系统。

游戏 Hugo Runner 将经度和纬度存储到首选项文件中。但是，正如我们手动验证的那样，这些偏好设置是在私有模式下正确写入的，从而排除了其他应用程序的任何访问。这再次表明精确的环境模型对于减少误报的数量是多么重要。因此，未来的工具应该更精确地模拟相应的 API。

对于大多数被检查的应用程序，FLOWDROID 在一分钟内终止。完成时间最长的实例是三星的 Push Service，分析时间约为 4.5 分钟。我们还对来自 VirusShare 项目 [1] 的大约 1000 个已知恶意软件样本运行了 FLOWDROID。平均运行时间为 16 秒，因为恶意软件样本似乎相对较小。最小运行时间为 5 秒，最大运行时间为 71 秒，这仅发生在单个相对较大的应用程序中。大多数应用程序包含两次数据泄漏（平均每个应用程序泄漏 1.85 次），通常将识别信息（如 IMEI）发送到远程服务器以注册手机，或作为 SMS 消息的一部分发送，有时会收取额外费用-率数。甚至发现一些恶意软件应用程序通过广播接收器接收数据，然后在 SMS 消息中发送这些数据。这可以允许其他应用程序间接发送 SMS 消息，而不需要它们自己的相应权限。

[2] 出于法律原因，我们无法在线提供这些申请。不过，为了能够重现我们的结果，研究人员可能会通过电子邮件向第一作者发送电子邮件以获取这些应用程序的副本。

Test-case group	TP	FP
Aliasing	11/11	0
Arrays	9/9	6
Basic	58/60	0
Collections	14/14	3
Datastructure	5/5	0
Factory	3/3	0
Inter	14/16	0
Pred	n/a	n/a
Reflection	n/a	n/a
Sanitizer	n/a	n/a
Session	3/3	0
StrongUpdates	0/0	0
Sum	117/121	9

表格 2 SecuriBench Micro 的测试结果

6.4 RQ4: SecuriBench Micro

FLOWDROID 是专为 Android 设计的，通过对 Android 生命周期的完整和精确处理，在这个领域获得了很高的精度。尽管如此，也没有什么可以阻止软件开发人员将 FLOWDROID 应用于 Java 应用程序。为了评估 FLOWDROID 为这个用例设置的效果如何，我们针对斯坦福 SecuriBench Micro [23] 版本 1.08 评估了 FLOWDROID，这是一组最初用于基于 Web 的应用程序的 96 个 J2EE 微基准的通用集合。对于套件中的每个基准，我们手动定义了必要的源、汇和入口点列表。由于 FLOWDROID 支持用于定义这些参数的简单文本文件格式，并且由于所有基准测试案例具有相同的结构，因此这并不费力。我们在实验中省略了涉及清理、反射、谓词和多线程的测试用例。正如我们之前解释的那样，这些功能超出了我们的分析工具的范围，就像所有其他现有的 Android 分析工具一样。

表 2 显示了按测试类别分组的测试结果。TP 列显示真阳性，即 FLOWDROID 发现的实际泄漏数量。例如，以 Basic 为例，FLOWDROID 在 60 个中找到 58 个。FP 列显示误报的数量，即 FLOWDROID 报告的发现与实际泄漏不对应，而是过度近似分析的伪影。在大多数情况下，这个数字相当低甚至为零，除了 FLOWDROID 报告 6 个误报的 Arrays 类别。同样，这些是由于未能精确地建模数组索引。

6.5 与其他工具相比

我们还尝试将 FLOWDROID 与科学文献中的许多其他工具进行比较，即 TrustDroid

[41]、LeakMiner [40] 和 Batyuk 等人的工具。[6]。不幸的是，这些工具都不能在线使用，相应的作者也没有回复我们的询问我们还尝试将 FLOWDROID 与科学文献中的许多其他工具进行比较，即 TrustDroid [41]、LeakMiner [40] 和 Batyuk 等人的工具。[6]。不幸的是，这些工具都不能在线使用，相应的作者也没有回复我们的询问。

我们尝试在 SCanDroid [14] 上运行 DROIDBENCH，但遇到了技术难题。该工具在我们的设置中根本没有报告任何发现。尽管与作者保持联系，但我们无法解决这些问题，最终双方都放弃了。AndroidLeaks [15] 的作者承诺在 DROIDBENCH 上运行他们的工具，但从未交付。我们还联系了 CHEX [24] 的作者，但由于 NEC 声称拥有知识产权，他们无法提供该工具或任何基准测试结果。Starostin [25] 拒绝参与实验，因为他的工具忽略了走样，使得任何比较都毫无意义。由于三星声称拥有知识产权，ScanDal [19] 的作者无法提供他们的工具，但使用我们的基准套件和反馈来改进他们的分析。据作者称，在 DROIDBENCH 上，分析现在报告的结果类似于 FLOWDROID。

结果，我们自己甚至无法成功评估一个针对 Android 的科学污点分析工具。这是非常不幸的，因为它限制了我们仅根据可用的出版物与那些工具进行比较。我们希望 FLOWDROID 和 DROIDBENCH 的可用性在未来将大大改善这种情况。毕竟，发表不可复制的结果会阻碍该领域的进步，并且在大多数其他科学领域被认为是不可接受的。

7. 其他工作

有几种方法可以对 Android 应用程序进行静态分析，这些方法在精度、运行时间、范围和关注点上有所不同。

最复杂的一个是 CHEX [24]，这是一种通过跟踪外部可访问接口和敏感源或接收器之间的污点来检测 Android 应用程序中的组件劫持漏洞的工具。虽然不是为该任务而构建的，但 CHEX 原则上可以用于污点分析。CHEX 不分析对 Android 框架本身的调用，而是需要一个（希望是完整的）框架模型。在 FLOWDROID 中，这样的模型是可选的，除了本地调用外，仅用于提高精度和性能。因此，用户可以完全省略该模型，但仍确保不会丢失污点。CHEX 的入口点模型需要枚举所有可能的“拆分排序”，这在 FLOWDROID 中是不必要的。此外，CHEX 最多只能对 1 个对象敏感，而 FLOWDROID 的需求驱动别名分析允许任意长度的上下文（使用默认值 5）。我们发现“单对象敏感”在实践中过于不精确

LeakMiner [40] 从技术角度来看与我们的方法相似：像 FLOWDROID，它基于 Soot，

使用 Spark 生成调用图，它实现了 Android 生命周期，并且论文指出应用程序可以在 平均 2.5 分钟。但是，该分析不是上下文相关的，因此无法对 DROIDBENCH 中的大多数测试用例进行精确分析。

AndroidLeaks [15] 还说明了处理 Android 生命周期的能力，包括回调方法。它基于 WALA 的上下文敏感系统依赖图，具有用于堆跟踪的上下文不敏感覆盖，但不如 FLOWDROID 精确，因为如果将受污染的数据存储在其字段之一中，它会污染整个对象，即两者都不是 领域也不对象敏感。这排除了对许多实际情况的精确分析。

SCanDroid [14] 是另一个用于推理 Android 应用程序中数据流的工具。它的主要关注点是组件间（例如，同一应用程序中的两个活动之间）和应用程序间数据流。这带来了将意图发送者连接到其他应用程序中各自的接收者的挑战。SCanDroid 修剪所有对 Android OS 方法的调用边缘，并保守地假设基对象、参数和返回值从参数继承污染。这远不如 FLOWDROID 的处理精确； FLOWDROID 仅对未明确建模的本机调用应用此默认规则。FLOWDROID 目前将意图发送建模为接收器，将意图接收建模为源，从而对应用间通信进行了良好的处理。未来，我们计划将 FLOWDROID 与 EPICC [27] 集成，这是一种新颖的静态分析，使用字符串分析来精确解决应用间通信。其他方法如 CopperDroid [31] 动态观察 Android 组件和底层 Linux 系统之间的交互以重建更高级别的行为。特殊的刺激技术用于锻炼应用程序以发现恶意活动。然而，攻击者可以轻松地修改应用程序以检测它是否在虚拟机内运行，然后在此期间不泄露任何数据 [29]。或者，数据泄漏可能仅在某个运行时阈值之后发生。Aurasium [38] 和 DroidScope [39] 在静态泄漏检测方面存在相同的缺点。

TaintDroid [11] 是迄今为止最复杂的 Android 污点跟踪系统之一。然而，作为一种动态方法，与 FLOWDROID 相比，它产生了一些完全不同的权衡。例如，TaintDroid 通过反射方法调用跟踪污点没有问题，因为 TaintDroid 是作为执行环境的扩展实现的，无论是否通过反射调用方法都无关紧要。另一方面，如果用于在安装之前对恶意软件进行分类，那么 TaintDroid 只有与能够产生良好代码覆盖率的动态测试方法配合使用才能成功检测恶意软件。像 FLOWDROID 这样的静态提前分析没有这个缺点，因为它们涵盖了所有执行路径。其次，诸如 TaintDroid 之类的动态方法可能会被恶意应用程序愚弄，该恶意应用程序识别出正在对其进行分析，在这种情况下，应用程序可以简单地避免执行任何恶意活动 [29]。如果动态分析安装在最终用户的手机上，这不是问题（在这种情况下，恶意软件将被有效驯服），但如果动态分析仅用于提前分类恶意软件随后可能安装在不受动态分析保护的系统上（在这种情况下，应用程序可以恢复其恶意活动）。FLOWDROID 等静态方法没有这个特殊的缺点，

因为它们从不实际执行应用程序。

F4F [36] 是一个使用称为 WAFL 的规范语言对基于框架的应用程序执行污点分析的框架，用于描述各个框架的功能行为。虽然最初是为 Web 应用程序创建的，但它也可以通过为 Android 添加 WAFL 生成器来扩展为对 Android 框架进行建模。FLOWDROID 的 dummy-main 生成具有很大的优势，即仅包含应用程序确实可以访问的组件和回调。然而，这需要应用程序清单、布局 XML 文件、编译资源文件和应用程序源代码的语义模型，这些都是交错的。F4F 充其量只能用于对所有可能应用程序的公分母进行粗略近似建模。

FLOWDROID 目前通过粗略的过度近似处理异常流。Kastrinis 和 Smaragdakis 最近提出了一种新颖且特别有效的方法，用于分析异常和点对点分析的组合 [18]。看看 FLOWDROID 是否可以轻松地从其中一些概念的集成中受益将会很有趣。

Rountev 等人。已经提出了一种为大型库预先计算摘要的方法，旨在加快对客户端代码的重复分析 [33]。对于带有庞大 Android 框架的 Android 应用来说，这样的做法很有意义。Roundev 的工作基于 IDE 框架 [34]，FLOWDROID 也在内部使用该框架进行基于 IFDS 的分析。因此，应该完全有可能将作者的想法融入 FLOWDROID。

迪利格等人。开发了一种更精确地分析集合和数组内容的方法 [10]。所需的分析工作并非微不足道，但鉴于我们的结果，很明显 FLOWDROID 可以通过沿着这些思路实施分析支持来进一步提高其精度。

8. 总结

我们展示了 FLOWDROID，一种用于 Android 应用程序的新颖且高精度的静态污点分析工具。与以前的方法不同，FLOWDROID 充分模拟了 Android 特定的挑战，如应用程序生命周期或回调方法，这有助于减少漏失或误报。新颖的按需算法允许 FLOWDROID 保持效率，尽管它具有强大的上下文和对象敏感性。为了评估分析工具的有效性，我们提出了特定于 Android 的基准测试套件 DROIDBENCH，并将其用于将 FLOWDROID 与商业工具 AppScan Source 和 Fortify SCA 进行比较，结果表明除了发现更多真正的泄漏（总共 93% 的泄漏），FLOW-DROID 还具有更高的精度 (86%)，从而减少误报。我们希望未来 DROIDBENCH 将作为 Android 污点分析的标准测试集。

为了分析来自 Google Play 商店的前 500 个真实世界应用程序，FLOWDROID 每个应用程序只用了不到一分钟的时间就发现了几个漏洞。在大约每分钟 16 秒的时间内分析了大

约 1000 个恶意软件样本，平均每个样本发现 2 个泄漏。在 SecuriBench Micro 上对 FLOWDROID 的评估显示 96% 的召回率只有 9 个误报

在未来的工作中，我们计划改进对处理反射的支持。我们也对自动预计算库抽象感兴趣。

致谢

我们要感谢 Fraunhofer SIT 的 Stephan Huber 支持我们使用 Google Play 市场的实际应用程序，以及 TZI Bremen 的 Karsten Sohr 博士支持我们进行 Fortify SCA 评估。感谢 Marc-Andre´ Laverdie`re 和其他人为我们实现 FLOWDROID、Soot 和 Heros 所做的贡献。这项工作得到了谷歌教师研究奖、EC SPRIDE 和 ZertApps 内的 BMBF、CASED 内的 Hessian LOEWE 卓越倡议、DFG 项目 RUNSECURE 中的支持，该项目与 DFG 优先计划 1496 “Re -liably Secure Software Systems – RS3”，由卢森堡国家科学基金会 (FNR) 在 AndroMap 项目下，并由美国国家科学基金会资助编号 CNS-1228700、CNS-0905447、CNS-1064944 和 CNS- 0643907。本材料中表达的任何意见、发现和结论或建议均为作者的观点，不一定反映国家科学基金会或任何其他资助合作伙伴的观点。

引用

[1] Virus share, aug 2013. http://virusshare.com/.

[2] IBM Rational AppScan, Apr. 2013. http://www-01.ibm.com/ software/de/rational/appscan/.

[3] Fortify 360 Source Code Analyzer (SCA), Apr. 2013. http://www8.hp.com/us/en/software-solutions/software. html?compURI=1214365#.UW6CVKuAtfQ.

[4] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically securing permission-based software by reducing the attack surface: an application to android. In ASE 2012, pages 274–277, 2012.

[5] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP ’12, pages 27–38, 2012.

[6] L. Batyuk, M. Herpich, S. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In Malicious and Unwanted Software (MALWARE), 2011 6th

International Conference on, pages 66–72, 2011.

[7] E. Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP ’12, pages 3–8, 2012.

[8] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In ICSE ’11: International Conference on Software Engineering, pages 241–250. ACM, May 2011.

[9] I. D. Corporation. Worldwide quarterly mobile phone tracker 3q12, Nov. 2012. http://www.idc.com/tracker/showproductinfo. jsp?prod_id=37.

[10] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL ’11, pages 187–200, 2011.

[11] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and

A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In R. H. Arpaci-Dusseau and

B. Chen, editors, OSDI, pages 393–407. USENIX Association, 2010.

[12] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, SPSM ’11, pages 3–14, New York, NY, USA, 2011. ACM. . URL http://doi.acm.org/10.1145/2046614.2046618.

[13] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein,

Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analy- sis for android applications. Technical Report TUD-CS-2013-0113, EC SPRIDE, May 2013. URL http://www.bodden.de/pubs/ TUD-CS-2013-0113.pdf.

[14] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications.

[15] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In Proceedings of the 5th international conference on Trust and Trustworthy Computing, TRUST’12, pages 291–307, 2012.

[16] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-

app advertisements. In Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks, WISEC ’12, pages 101–112, New York, NY, USA, 2012. ACM. URL http://doi.acm.org/10.1145/2185448.2185464.

[17] G. Inc. Application fundamentals. 2013. URL http://developer. android.com/guide/components/fundamentals.html.

[18] G. Kastrinis and Y. Smaragdakis. Efficient and effective handling of exceptions in java points-to analysis. In R. Jhala and K. D. Bosschere, editors, CC, volume 7791 of Lecture Notes in Computer Science, pages 41–60. Springer, 2013.

[19] J. Kim, Y. Yoon, K. Yi, and J. Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In H. Chen, L. Koved, and D. S. Wallach, editors, MoST 2012: Mobile Security Technologies 2012, Los Alamitos, CA, USA, May 2012. IEEE.

[20] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can’t live with ‘em, can’t live without ‘em. In Proceedings of the 4th International Conference on Information Systems Security, ICISS ’08, pages 56–70, Berlin, Heidelberg, 2008. Springer-Verlag. .

[21] P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The soot framework for java program analysis: a retrospective. In Cetus Users and Compiler Infastructure Workshop (CETUS 2011), Oktober 2011.

[22] O. Lhota´k and L. Hendren. Scaling java points-to analysis using spark. In G. Hedin, editor, Compiler Construction, volume 2622 of LNCS, pages 153–169. Springer Berlin Heidelberg, 2003.

[23] B. Livshits. Securibench micro, Mar. 2013. http://suif.stanford. edu/~livshits/work/securibench-micro/.

[24] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In CCS 2012, pages 229–240, 2012.

[25] C. Mann and A. Starostin. A framework for static detection of privacy leaks in android applications. In Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC ’12, pages 1457–1462, 2012.

[26] N. A. Naeem, O. Lhota´k, and J. Rodriguez. Practical extensions to the ifds algorithm. In Compiler Construction 2010, pages 124–144, 2010.

[27] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In USENIX Security Symposium 2013, Aug. 2013.

[28] Paladion. Insecurebank test app. http://www.paladion.net/ downloadapp.html.

[29] N. J. Percoco and S. Schulte. Adventures in bouncerland. Blackhat USA, 2012.

[30] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In 2014 Network and Distributed System Security Symposium (NDSS), Feb. 2014. URL http://www.bodden.de/pubs/rab14classifying. pdf. To appear.

[31] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In EUROSEC, Prague, Czech Republic, April 2013.

[32] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In POPL '95, pages 49–61, 1995.

[33] A. Rountev, M. Sharp, and G. Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In Compiler Construction, volume 4959 of LNCS, pages 53–68. Springer, 2008.

[34] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In TAPSOFT '95, pages 131–170, 1996.

[35] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices, 2013.

[36] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: taint analysis of framework-based web applications. In OOPSLA 2011, pages 1053–1068, 2011.

[37] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. An- dromeda: Accurate and scalable security analysis of web applications. In FASE 2013, pages 210–225, 2013.

[38] R. Xu, H. Sa¨ıdi, and R. Anderson. Aurasium: practical policy enforce- ment for android applications. In USENIX Security 2012, Security'12, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association.

[39] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In USENIX Security 2012, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[40] Z. Yang and M. Yang. Leakminer: Detect information leakage on android with static taint analysis. In Software Engineering (WCSE), 2012 Third World Congress on, pages 101–104, 2012.

[41] Z. Zhao and F. Osono. Trustdroid: Preventing the use of smartphones for information leaking

in corporate networks through the used of static analysis taint tracking. In Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on, pages 135–143, 2012.

[42] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP ’12, pages 95–109, 2012.