**4.4 基于数据流分析技术的应用漏洞检测模块**

在4.3所述的恶意应用鉴别模块判别出是否为恶意应用后，紧接着运行的是静态检测中的第二个模块——漏洞检测模块。Android应用中或多或少都存在有编写漏洞的代码，这些Android应用漏洞虽然可能并不含有恶意代码，无法直接被4.3的恶意应用鉴别模块检测出来，但在被攻击者恶意利用的情况下，同样可以对用户产生相当严重的威胁，降低应用的安全性。比如在Android应用开发中对证书校验部分的代码编写存在问题，而没有实现证书的有效校验，则可能产生中间人劫持攻击，攻击者通过拦截正常的网络通信数据，可以进行数据篡改和嗅探，而通信的双方却毫不知情。基于上述情况，本作品从静态分析角度入手，设计了Android应用漏洞检测模块。另一方面，现有的基于Java源码的静态检测技术在检测精度和运行效率上都仍有很大的提高空间，在漏洞检测模块中，我们利用Soot提供的流分析技术来进行漏洞检测，自行设计并实现了基于过程内数据流分析技术的对象行为记录算法，以实现函数内的变量参数到相关常量的追溯等功能，提高漏洞检测精度。

**4.4.1 Soot的基本介绍**

Soot是一款在Github上开源的Java代码分析框架，最初的功能只是为Java程序的分析优化提供一款方便实用的工具，之后Soot工作组在此基础上进行了延伸，增加了诸如程序间控制流图构建，过程内数据流分析框架，基于IFDS问题的数据流分析框架等功能，支持使用者在静态代码分析过程中结合流分析技术。

Soot提供的最基础的功能是将程序转化为一种易于分析的中间表示形式，并将程序语句和逻辑关系转化成一系列Soot定义的对象，使用者在分析中可以基于这些Soot定义的对象来结构化的对代码进行优化或分析。Soot提供了四种Java字节码的中间表示形式：

* Baf 一种易于操作的，精简的字节码表示形式
* Jimple 提供一种适合分析优化的三地址码表示形式
* Shimple Jimple中间代码的SSA变体
* Grimp 适用于反编译和代码分析的Jimple汇总版本

在作品中，我们使用Jimple类型用于漏洞分析，Jimple中间代码将复杂的Java字节码都转化为三地址指令，并且只有15种基本语句类型，分类如下所示：

|  |  |
| --- | --- |
| 类型 | 语句类 |
| 核心语句 | NopStmt, IdentityStmt, AssignStmt |
| 有关过程内控制流的语句 | IfStmt Goto-Stmt,  TableSwitchStmt LookupSwitchStmt |
| 有关过程间控制流的语句 | InvokeStmt, ReturnStmt, Return-VoidStmt |
| 监控语句 | EnterMonitorStmt, ExitMonitorStmt |
| 其他 | ThrowStmt, RetStmt |

Jimple中间代码除了对不同语句类型进行了分类，还对语句内部进行了基本数据结构的填充，如类，方法，函数体，语句，表达式，变量等都进行了解析，按照常见的数据结构分类方式进行整理后如下图所示：

|  |  |
| --- | --- |
| 数据类型 | 描述 |
| Scene | 用于提供反编译后完整的程序环境 |
| SootClass | Soot对反编译后Java类的表示 |
| SootField | Soot对Java类中成员变量的表示 |
| SootMethod | Soot对反编译后函数的表示 |
| Body | Soot对函数方法体的表示 |
| Local | Soot对函数内定义的局部变量的表示 |
| Unit / Stmt | 两种都可以用来表示一条Jimple语句 |
| Expr | Jimple语句中的表达式，如AddExpr |
| ValueBox | Jimple语言对语句中元素块的封装 |

一些常见类型之间的关系如下图所示：



通过上述定义的基本数据类型，使用Soot可以结构化的对反编译后的Java代码进行分析。

**4.4.2 数据流分析技术及Soot提供的过程内数据流分析框架**

数据流分析技术是一种用于收集程序在不同点计算的值得技术，用来确定对变量的赋值可能传播到程序中的那些部分，通过Soot提供的数据流分析框架，我们可以很方便的模拟程序运行中的走向，进而在模拟的过程中求解问题。流分析技术从大体上可以分为两类——控制流分析与数据流分析，在应用中，控制流分析和数据流分析需要结合使用，对程序的控制流分析是对源程序或者源程序的中间表示形式的直接操作，形成控制流图；数据流分析是在控制流分析后得出的控制流图的基础上，将程序中的包含数据的变量沿着控制流图的路径，进行赋值和传递，直至程序完成，变量回收或者未被回收。

数据流分析的基础是控制流图，在构造完成控制流图之后，即可沿着控制流图进行传递并求解答案。在控制流图可以分为两个大类，过程内的控制流图（CFG）和过程间的控制流图（ICFG），对它们的解释如下：

* 过程间控制流图——ICFG(interprocedural control flow graph)

基于函数间的调用关系以及函数内的控制流所得出的，可以用于描述主程序的执行逻辑

* 过程内控制流图——CFG(control flow graph)

主要表示为程序中的某个程序块内部的顺序执行逻辑图的表示，例如主程序中的某个函数是一个具有单独逻辑功能的程序块，它内部的控制流图就是过程内控制流图

* 调用图——CG(call graph)

函数的调用关系图，主要用于表示程序执行逻辑块之间的相互调用关系，例如主程序在执行到某条语句时，调用了具有某个功能的函数，之后程序的执行顺序就会从主程序跳转至该函数块中，当函数执行完成后，通过一定的执行动作或返回值跳转回主函数，继续执行主函数的顺序逻辑

由此可见，控制流图间的基本关系为ICFG = CFG + CG，如果想要进行过程间控制流图分析的话，需要首先求出CFG与CG，然后分步进行求解

Soot提供了三种功能不同的过程内数据流分析框架，分别有对流向敏感的，对分支敏感的，前向分析，后向分析等不同类别，它们之间的类关系如下入所示：



在作品中，我们选则继承前向数据流分析框架——ForwardFlowAnalysis类来实现数据流分析功能，在框架中，需要实现的方法有：

* flowThrough()

在数据流的每个节点，都会调用一次flowThrough函数，传过来上一个节点的记录值，并返回下一节点的记录值，而flowThrough函数要做的，就是对节点中的语句进行分析，适当修改记录值并返回

* merge()

Soot提供的数据流分析框架会遍历代码中的每个分支，因此在出现不同分支的时候，节点的记录值也相应会出现分支，而merge函数需要做的，就是在出现两个不同记录值的分支下，进行适当操作，例如取并集，或是或集

* copy()

如merge函数中所述，在数据流进入分支时，记录值将复制成两份进行记录，但记录值是可以自定义类型的，因此需要使用者提供记录值类型的copy方法

在实现上述方法后，调用doAnalysis()即可开始执行函数内的数据流分析，在数据流分析结构中，通过getFlowAfter(Stmt)或者getFlowBefore(Stmt)即可获得运行后的记录值情况。

**4.4.3 基于过程内数据流分析技术的对象行为记录算法**

利用上面介绍的数据流分析技术，我们设计并实现了一种函数内的对象行为记录算法，通过Soot提供的过程内数据流分析框架，可以方便的实现函数内基于控制结构的完整遍历，由此可以完整的为函数内的所有变量对象的行为进行建模，数据流分析框架执行所产生的记录值中将包含每个变量的初始化、变量的方法和成员变量调用、变量赋值、变量值传递等所有变量相关的行为，对这些行为的建模往往是很有用的。例如可以用于变量到常量的追溯，一个最常见的例子就是使用StringBuilder连接字符串，最后调用toString()方法将生成的字符串作为参数传给函数，这样就将参数的常量特征隐蔽掉了，对于这种情况，不使用数据流技术的静态检测方案是很难匹配到特征的，而使用本小节设计的算法则可以有效对抗这些混淆手段。对象行为记录算法另一方面的应用是用于梳理漏洞利用链中的关系，在一些较为复杂的漏洞利用中，漏洞识别并不能只靠几个单一的特征函数就确定，依靠对变量行为的建模，可以将不同漏洞特征函数对应到每一个变量对象的函数调用下面，依靠变量间的相互作用关系达到更精准的识别。

为利用Soot的数据流分析框架实现本算法，首先需要确定数据流分析中所使用的记录值。因为目的是为函数内的变量行为建模，所以使用HashMap<Local, LocalInfo>来进行记录，其中Local为Soot中定义的局部变量对象，LocalInfo则设计用于存储此Local变量在函数中的行为，其数据结构如下：

**public boolean isDeriveFromOrtherObject** = **false**;  
**public** Value **rootObject** = **null**;  
**public** Map.Entry<Local, InvokeExpr> **rootLink** = **null**;  
**public** List<Value> **invokeInfo** = **new** LinkedList<Value>();

rootObject记录此变量对象的初始化类型，isDeriveFromOrtherObject，rootLink两个变量则用于刻画不同变量对象之间的关系，其中isDeriveFromOrtherObject记录此变量初始化时是否来自其他变量，rootLink则在isDeriveFromOrtherObject值为true时记录此变量初始化时所用到的其他变量对象信息，最后invokeInfo是一个Value对象的列表，用于记录此变量对象在函数内所有的函数调用情况。

在记录值的数据结构建立完成后，还需要实现数据流分析框架中的几个方法，其中最关键的是flowThrough()以及merge()方法的实现，其中为了保证对象行为建模的准确性，在分支情况下，对不同的记录值需要做并集处理，因此merge()函数中对记录值做并集处理即可，下面将详细介绍flowThrough()函数的实现。

如4.4.2小节所述，flowThrough()函数用于对数据流分析中每个节点的语句进行分析处理来增减记录值，为了实现对象行为的建模，需要基于Jimple语言的15种基本语句来对语句进行解析，从中读取出有效信息并添加到记录值中，在作品中我们初步对变量相关语句的语义进行如下分类：

1. IdentityStmt

变量初始化操作

1. InvokeStmt

如果可以从中提取出Local的话，即为变量对象的方法调用操作

3） DefinitionStmt

|  |  |  |
| --- | --- | --- |
| 左操作数 | 右操作数 | 含义 |
| Local | Local | 对象间的赋值操作 |
| Local | Constant | 使用常量对变量对象的赋值操作 |
| Local | ArrayRef | 使用数组对变量对象的赋值操作 |
| Local | AnyNewExpr | 使用new语句进行变量初始化 |
| Local | InvokeExpr(含Local) | 使用函数调用进行的赋值操作 |
| Local | InvokeExpr(无Local) | 使用工厂函数进行变量初始化 |
| Local | AddExpr | 使用运算表达式的赋值操作 |
| Local | FieldRef | 使用对象成员对象进行的赋值操作 |

基于对上述规则，我们将语句按照不同的语义划分，并分别从中提取有效信息录入到记录值中，在借助Soot的数据流分析框架实现函数内所有语句的结构化完整遍历后，即可生成函数内所有局部变量的行为建模，供之后的漏洞分析使用。

**4.4.4 漏洞检测模块的工作流程**

漏洞检测模块的完整工作流程如下图所示。首先使用Soot对APK进行初始化工作，输出的两个结果一个是反编译之后的Jimple中间代码，另一个则是控制流图，其中控制流图使用上一小节中介绍的算法为函数内的变量生成行为记录。初始化工作结束后，则开始进行漏洞特征匹配，最后输出检测结果



**4.4.5 举例：JBOH漏洞的检测**

WebView远程代码执行漏洞是Android应用中相当常见的一个漏洞，因其任意代码执行的功能，所造成的危害也相当广泛。本文将以此为例介绍基于Jimple中间代码，以及结合数据流分析技术下的WebView远程代码执行漏洞的检测方案。

Android系统中提供JavaScript调用Java代码的方法，用户只需要在WebView中调用setJavaScriptEnabled(true)函数将JavaScript代码的运行开关打开，甚至都不需要向其中提供Java实例对象，系统就默认会在WebView中自动添加searchBoxJavaBridge\_的Java实例对象，对象中的所有方法在JavaScript中均可调用，因此攻击者可以通过反射调用攻击实现任意代码执行。在API>17之后，Android系统通过添加JavaScript注解机制来主动声明Java实例中允许被调用的方法，但实际上如果提供调用的方法中含有敏感API的话，同样可能对用户造成威胁。

在检测前首先提取出应用所使用的API版本号，对于API<17的应用，首先遍历出函数内所有WebView类型的局部变量，然后使用前面介绍的变量行为记录算法，生成函数内所有变量的行为集合，在集合中分别寻找setJavaScriptEnabled(boolean)

removeJavascriptInterface(java.lang.String)

addJavascriptInterface(java.lang.Object,java.lang.String)

三个方法的调用情况，对于将JavaScript调用设置为允许的WebView，则进行进一步的检查，对于没有移除默认接口”searchBoxJavaBridge\_”，以及又添加了新Java实例的WebView，均可判定为存在WebView远程代码执行漏洞。对于API>17的应用，因为提供了注解机制，所以首先寻找addJavascriptInterface()函数的调用，从其中提取出传入的Java实例对象，之后对此Java对象中存在JavaScript注解的方法进行分析，如果方法内含有危险API的话，同样判定此WebView存在漏洞，流程图如下所示：



**4.4.5 漏洞检测模块在批量测试中的效果**

我们针对11类，共计16项常见且危害性较大的漏洞编写了检测方案，并从公开的恶意软件库VirusShare中下载了6801款恶意Android应用程序进行批量测试，检测结果如下：

|  |  |
| --- | --- |
| 漏洞描述 | 总漏洞数 |
| WebView开启了Javascript，并没有移除默认接口，在API<17时可能导致Web组件远程代码执行漏洞 | 2524 |
| API>17时，javascript注解机制所提供的接口中检测到含有危险API | 55 |
| 自实现的HostnameVerifier返回值恒为true，非有效校验 | 704 |
| 自实现的校验证书的X509TrustManager接口的checkServerTrusted()方法实现为空， | 1706 |
| 证书校验中接受任意域名，可能受到中间人劫持攻击 | 39 |
| WebView忽略SSL证书错误检测，可能导致中间人攻击的威胁，可能导致隐私泄露 | 472 |
| 应用程序加解密时密钥使用硬编码，攻击者通过反编译拿到密钥即可解密APP通信数据 | 1595 |
| 检测到应用存在代码动态加载的行为，应用可能隐藏有未检查到的恶意功能 | 3366 |
| AndroidManifest.xml文件中allowBackup属性值被设置为true | 6686 |
| AndroidManifest.xml文件中debuggable属性值被设置为true | 3643 |
| 检测到存在SYSTEM\_ALERT\_WINDOW(系统弹窗)权限，该应用可能在应用外弹窗 | 839 |
| 应用可能尝试使用toast实现全局弹窗，此项全局弹窗并不需要权限声明 | 127 |
| WebView并没有调用setSavePassword(false)，存在WebView明文存储密码漏洞 | 3372 |
| 应用使用WebView，同时支持File协议，在特定情况下可能利用File协议获得应用的敏感数据 | 1515 |
| 文件读写使用全局模式，可能造成隐私数据泄露 | 542 |

从图中可以看到，在应用中存在大量的代码漏洞，其中最多的当为Manifest文件中的allowBackup属性置为true，以及debuggable属性置为true这两个常见的配置问题，另外WebView相关的安全漏洞也不在少数，在应用开发中，这些比较细节的问题很容易忽略，一旦被利用则后果不堪设想。另一方面，漏洞检测模块在批量测试中稳定性较高，没有出现程序崩溃，死循环等问题。

**4.5 隐私泄露检测模块**

在智能设备大量普及的当下，手机支付、手机社交都与移动设备紧密相关，用户的手机中往往比电脑存放着更多用户的隐私信息，而另一方面，应用不论是为了改善用户体验，还是为了实现更精准的投放广告，都需要尽可能获得更详细的用户个人信息，一旦这些应用过量收集用户隐私，甚至是恶意加以利用，就可能给用户带来无法挽回的损失。因此用户隐私安全性与Android应用的安全性是紧密相关的，隐私泄露检测模块便从应用对用户隐私的收集行为入手，进行Android应用安全性的评估。

**4.5.1 静态污点分析算法与Sources点、Sinks点的分类**

应用的隐私收集行为可以大致分为以下步骤：

1. 调用获取用户隐私的函数，这些函数的返回值均包含用户的隐私信息，我们将这些称为Source点，即意为污点数据的源点
2. 携带用户隐私数据的变量在程序运行中传播，其中可能包括的行为有赋值给其他变量，销毁等，这些污点的传播过程是需要进行跟踪分析的
3. 调用发送隐私数据的函数，这些函数一般将导致用户隐私数据泄露，如发送短信的函数sendTextMessage，我们将这些成为Sink点，即意为污点数据发送出去的点

静态污点分析算法是一种基于数据流分析的软件分析技术，主要用于分析程序运行过程中敏感数据的传播路径，以及判断程序的运行过程中是否导致了敏感数据的泄露，通过追踪程序中与敏感信息相关的数据流，分析关键函数之间的数据可达路径。在隐私泄露检测中，静态污点分析问题的理论基础即为IFDS问题：假设数据流集合是一个确定的有限的集合，在这个数据流集合上的数据流函数对于集合操作符是可分配的，并且数据流的追踪是可跨函数的，那么这一类问题称之过程间(Interprocedure)，有限的(Finite)，可分配的(Distributive)子集(Subset)问题。

在静态污点分析过程中，首先要做的就是从代码中收集所有的Sources点与Sinks点，然后才能接着分析任意两点间的传播路径。在Android系统中包含有很多涉及到用户隐私的API，例如能返回用户设别MEID的getDeviceId()、获取用户精确位置的getLastKnownLocation()、获取浏览器书签信息的的getAllBookmarks()、获取浏览器历史记录的信息getAllVisitedUrls()等，通过归类整理本文将其大致分为以下6类，如下表所示

|  |  |
| --- | --- |
| **隐私类别** | **在测试中的发现数量** |
| 用户位置信息 | 7 |
| 用户唯一标志信息 | 6 |
| 用户私人数据 | 9 |
| 用户网络访问相关信息 | 10 |
| 应用组件信息 | 9 |
| 程序运行间数据信息 | 98 |

在Android系统中，同样可以使用多种手段将数据发送到攻击者手中，例如通过sendTextMessage()发送短信、通过openConnection()的HTTP请求发送数据、通过Log.d()以日志形式记录数据、通过FileOutputStream.write()将数据写入到本地文件等，在进行归类后我们将其大致分为以下5类

|  |  |
| --- | --- |
| **数据发送方式** | **Android系统中相关API条目数** |
| 通过短信发送数据 | 3 |
| 通过网络请求发送数据 | 13 |
| 通过日志记录数据 | 14 |
| 通过本地文件记录数据 | 21 |
| 通过应用组件传递数据 | 108 |

在对Sources点与Sinks点归类整理后，便可以使用污点分析算法开始寻找从Sources点到Sinks点的传播路径，从而发现这些隐私泄露问题。

**4.4.2 对Android应用的生命周期建模与IFDS问题求解**

污点分析算法在实际情况中的解决方案是基于过程间数据流分析技术的，首先需要为待分析程序建立过程间控制流图，然后在此图上，使用过程间数据流分析技术进行IFDS问题的求解，以寻找污点传播的路径。Android应用程序的运行方式不像其他程序一样有个固定的main函数入口点，然后顺序执行下去。Android应用中组件大都有自己完整的生命周期，组件在运行过程中以触发调用的方式运行，有区别于一般应用程序的顺序执行，下图为Activity组件的完整生命周期。



因此，想要对Android应用程序进行过程间数据流分析，就需要先为Android应用程序的生命周期进行完整建模，通过构造一个虚拟的main方法，在方法内模拟触发所有组件，然后再对这个虚拟的main方法进行数据流分析，寻找Sources到Sinks间的路径。

在控制流图制作结束后，接着就需要在其上使用过程间数据流分析技术进行IFDS问题的求解。得益于FlowDroid项目为Android生命周期做的完整建模，以及Soot项目提供的通用的IFDS问题求解框架，我们可以很方便的在Android应用程序中使用静态污点分析算法来发现潜在的隐私泄露行为，具体操作步骤在下一小节中描述 。

**4.4.3 使用FlowDroid进行隐私泄露行为检测的流程**

FlowDroid的运行需要提供一系列的配置，具体如下：

1. 所有回调函数的列表：FlowDroid会在应用程序中提取出所有列表中记录的回调函数，并将其添加到虚拟的dummyMain方法中去。
2. Sources点与Sinks点的列表：FlowDroid会在应用程序中提取列表中记录的Sources点与Sinks，并调用污点分析算法分析传播路径。
3. Android各版本的SDK：可以在Android官网中下载，为了保证分析结果的准确性，至少需要提供1-26版本的API。
4. Soot运行结果：FlowDroid的运行需要基于Soot的运行结果，为了提高运行效率，我们将上一部分漏洞分析中Soot的初始化结果保存下来，在隐私泄露检测中再次利用
5. Config信息：在测试中发现在组件的回调分析中使用递归的方式，在单个组件的的回调函数过多时会因为递归深度上限而产生异常，因此将配置中的MaxCallbacksPerComponent设置为5防止深度递归带来的异常。

在所有配置完毕且均正确后，调用runInfoflow()函数即可开始进行隐私泄露的分析，分析结果存储在自定义的results中，从中取出分析结果并加以解析即可，完整的工作流程如下：



**4.4.4 隐私泄露模块在批量测试中的效果**

基于4.4.1小节中整理的Sources点与Sinks点，使用上一小节描述的FlowDroid检测过程，对3053款VirusShare中下载的恶意应用做了隐私泄露检测，并对检测结果基于3.1中的分类作了整理，分类情况如下：

|  |  |
| --- | --- |
| **隐私类别** | **在测试中的发现数量** |
| 用户位置信息 | 79 |
| 用户唯一标志信息 | 145 |
| 用户私人数据 | 176 |
| 用户网络访问相关信息 | 270 |
| 应用组件信息 | 409 |
| 程序运行间数据信息 | 139 |

|  |  |
| --- | --- |
| **隐私发送方式** | **在测试中的发现数量** |
| 通过短信发送数据 | 388 |
| 通过网络请求发送数据 | 223 |
| 通过日志记录数据 | 267 |
| 通过本地文件记录数据 | 157 |
| 通过应用组件传递数据 | 183 |

从统计中可以看到，在3053款应用中，共发现了1218条对用户隐私的窃取行为，平均每不到3个软件就会有一条隐私泄露行为产生，其中不乏getLastKnownLocation(java.lang.String) ，getAccounts()等敏感的用户数据，这些隐私泄露行为均可通过本模块有效的检测出来。