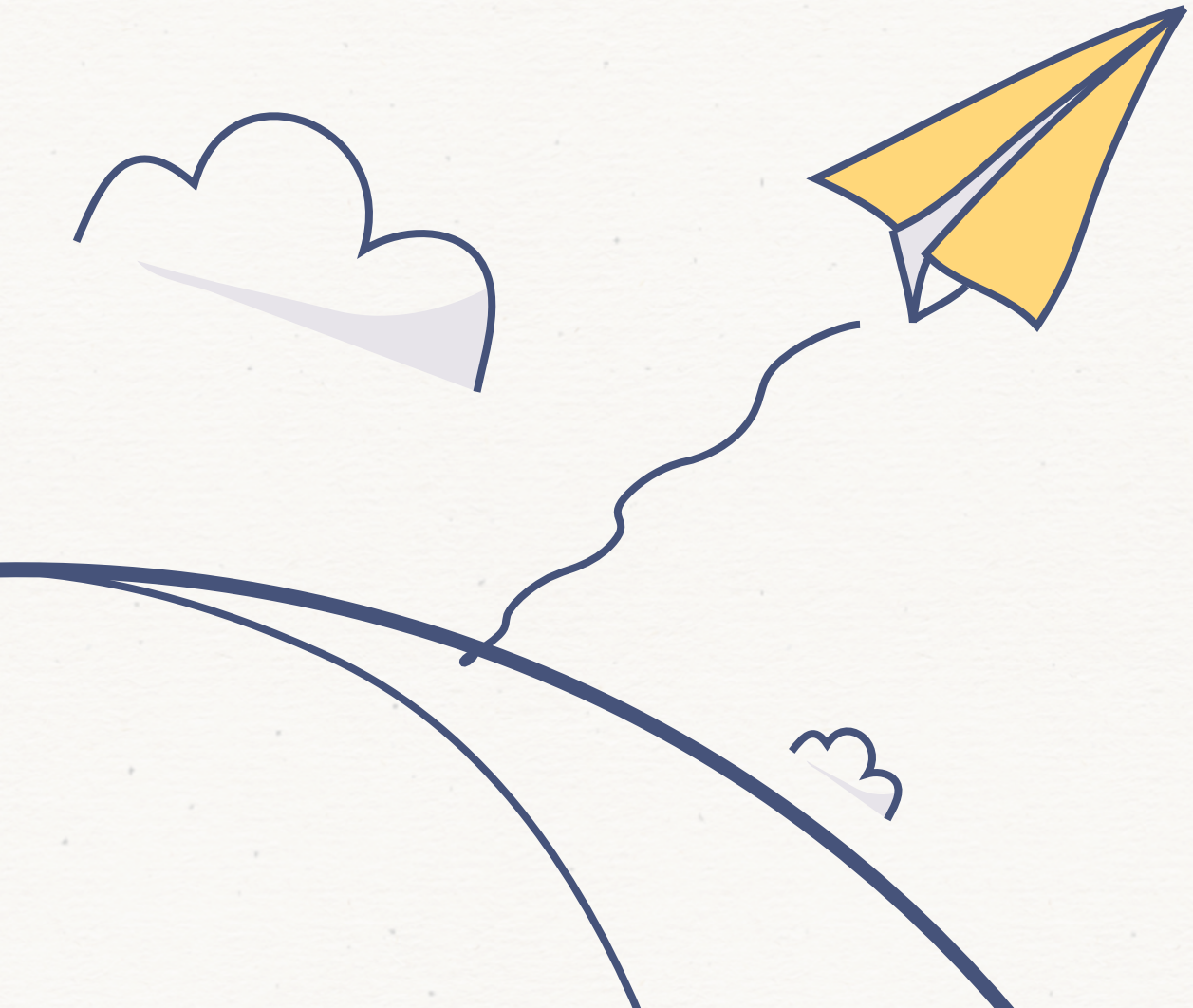


Soot & Flowdroid

Java 代码分析

Contents
目录

Soot

简介

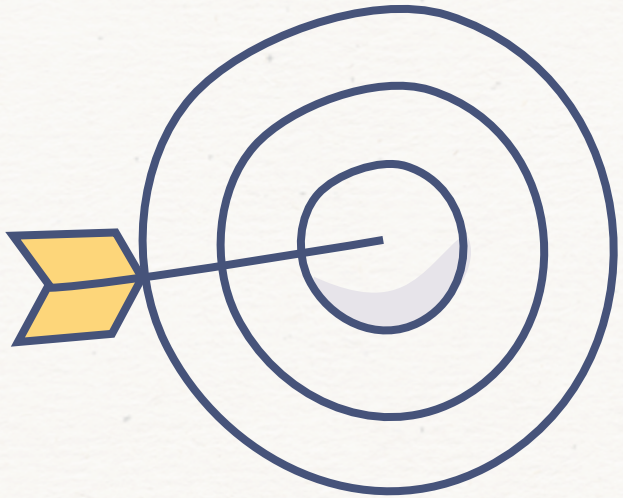
"Soot is a Java optimization framework. It provides four intermediate representations for analyzing and transforming Java bytecode"

Soot作为一个静态java代码优化框架提供了一些API,
帮助我们进行代码分析（比如生成函数调用图、进行数据流分析等等）

Soot能做到的

- Call-graph construction
- Point-to analysis
- Def/use chains
- Template-driven Intra-procedural data-flow analysis
- Template-driven Inter-procedural data-flow analysis

.....



Soot的三种部署方式

通过命令行中使用

在github上直接下载对应的jar包，通过设置不同的options直接使用soot的一些功能：

- 不同形式代码的转换
- 构造call graph
- 代码优化
- ……

在IDE中导入soot包

安装maven -> 开一个maven工程 -> 在pom.xml中设置soot依赖

下载github源码 -> 直接在源码工程中开发

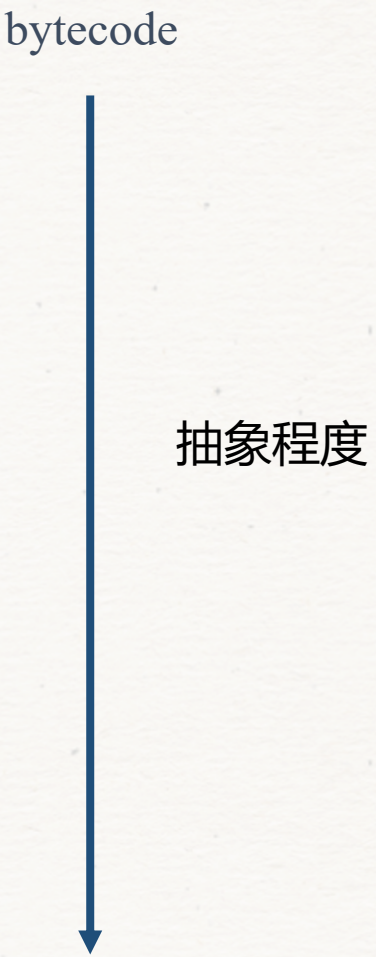
可以直接使用soot中的所有api，方便开发

Eclipse的soot插件

Soot的四种中间代码(IR)

不同形式的中间代码可以帮助我们从不同的角度去分析程序

- **Baf** 基于栈操作，忽略了bytecode中的type依赖
- **Jimple** 三地址编码，15条指令，最核心的IR
- **Shimple** 静态单赋值（SSA）版本的Jimple
- **Grimp** 更接近java代码，适合阅读



Jimple的15条指令

1. NopStmt
2. IdentityStmt
3. AssignStmt
4. IfStmt
5. GotoStmt
6. TableSwitchStmt
7. LookUpSwitchStmt
8. BreakpointStmt
9. InvokeStmt
10. ReturnStmt
11. ReturnVoidStmt
12. EnterMonitorStmt
13. ExitMonitorStmt
14. ThrowStmt
15. RetStmt

Soot中的几个重要概念(Jimple)

Scene

The Scene class represents the complete environment the analysis takes place in. Through it, you can set e.g., the application classes, the main class and access information regarding interprocedural analysis.

Body

Body represents a single method body, we can retrieve a Collection of the locals declared, the statements which constitute the body and all exceptions handled in the body.

Statement

Through a Statement we can retrieve values used and values defined. Additionally, we can get at the units jumping to this unit and units this unit is jumping to.

Value

A single datum is represented as a Value. Examples of values are: locals , constants, expressions, and many more.

Soot中的几个重要数据结构(Jimple)

SootClass

一个SootClass对应原代码中的一个java对象，可以通过Scene.v().getClasses()来获取程序中所有对象。

SootMethod

一个SootMethod对应原代码中的一个java方法，可以通过SootClass.getMethods()来获取对象中所有类。

Unit

一个Unit对应原代码中的一条java语句，可以通过SootMethod.retrieveActiveBody().getUnits()获取方法中所有语句。

Box

一个Box对应原代码中的一个“变量”或者说“指针”，可以通过Unit.getDefBoxes()/Unit.getUseBoxes()获取语句中的“指针”。

Soot使用——框架简述

Soot中的每一个分析模块称之为pack

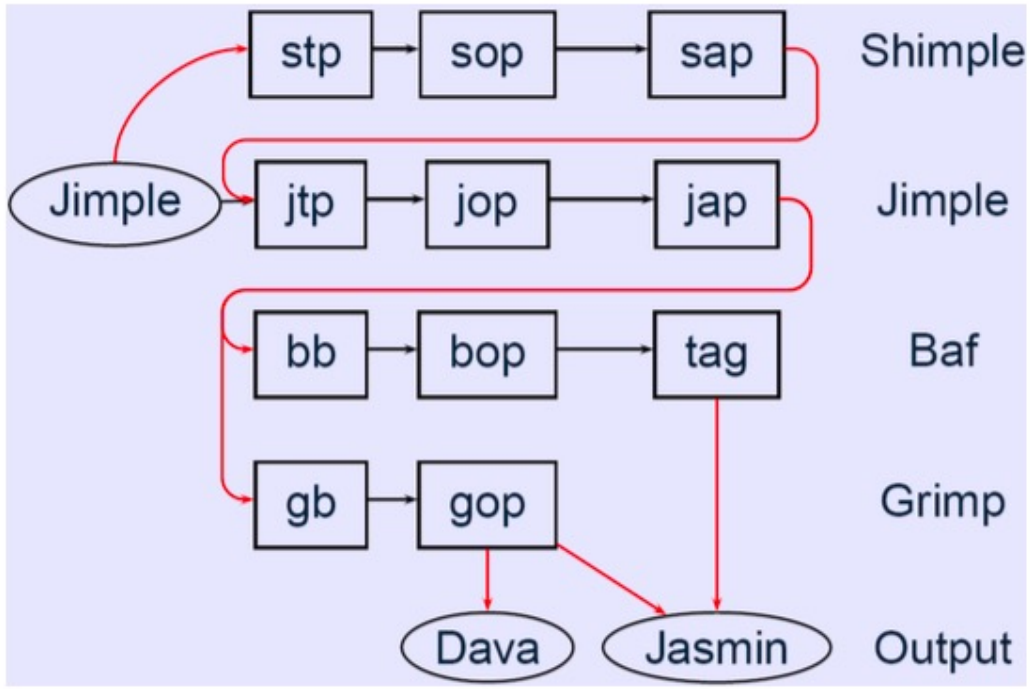


Figure 1: Intra-procedural execution flow. Image taken from [5].

第一个字母表示IR: s, j, b, g。

第二个字母表示执行的功能:
b (body creation)
o (optimization)
t (user-defined transformation)
a (annotation)

第三个字母为模块p(pack)

在函数间分析时还有几个额外的packs：

cg(call graph), wjtp(whole jtp), wjop(whole jop), wjap(whole jap)

Soot使用——api使用示例

```
public static void main(String[] args) {
    //spotbugs -- testing
    String classesDir = "D:\\wkspace\\seed8\\dir\\spotbugs";
    String mainClass = "edu.umd.cs.findbugs.LaunchAppropriateUI";

    //set classpath
    String jreDir = System.getProperty("java.home") + "\\lib\\jce.jar";
    String jceDir = System.getProperty("java.home") + "\\lib\\rt.jar";
    String path = jreDir + File.pathSeparator + jceDir + File.pathSeparator + classesDir;
    Scene.v().setSootClassPath(path);

    //add an intra-procedural analysis phase to Soot
    TestCallGraphSootJar_3 analysis = new TestCallGraphSootJar_3();
    PackManager.v().getPack("wjtp").add(new Transform("wjtp.TestSootCallGraph", analysis));

    excludeJDKLibrary();

    Options.v().set_process_dir(Arrays.asList(classesDir));
    Options.v().set_whole_program(true);

    SootClass appClass = Scene.v().loadClassAndSupport(mainClass);
    Scene.v().setMainClass(appClass);
    Scene.v().loadNecessaryClasses();

    // 或
    /* Scene.v().loadNecessaryClasses();
    SootClass sc = Scene.v().getSootClass(mainClass);
    Scene.v().setMainClass(sc);*/
    //enableCHACallGraph();
    enableSparkCallGraph();
    PackManager.v().runPacks();
}
```

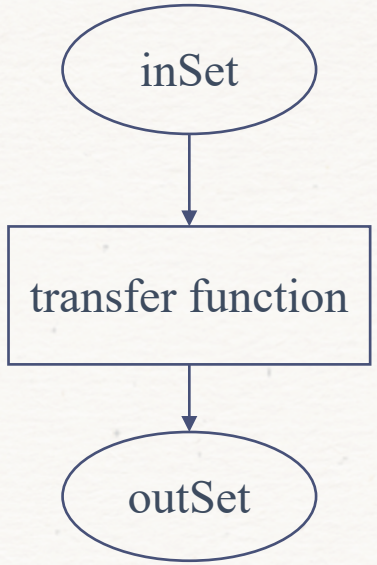
设置想要执行的pack

加载需要分析的类

执行之前设置的pack

Soot使用——data-flow analysis

数据流问题研究的是程序的某个点处的数据流值，数据流分析的通用方法是在控制流图上定义一组方程并迭代求解。



- **正向传播(forward)**，就是沿着控制流路径，状态向前传递，前驱块的值传到后继块
- **逆向传播(backward)**，就是逆着控制流路径，后继块的值反向传给前驱块。

- **到达定值(forward)**，分析语句d中赋值的变量可能能够到达哪条语句。
- **活跃变量(backward)**，分析变量v可能在之后的程序路径中被使用且未被重新赋值。

$$in[B] = \cup\, out[B_{\text{last}}]$$
$$out[B] = gen[B] \cup (in[B] - kill[B])$$

$$out[B] = \cup\, in[B_{next}]$$
$$in[B] = use[B] \cup (out[B] - def[B])$$

Soot使用——data-flow analysis framework

Soot提供了两个基础类供我们自定义数据流分析过程：BackwardFlowAnalysis, ForwardFlowAnalysis

重写类的构造方法与flowThrouph方法，形式如下：

```
1  public class MyForwardAnalysis extends ForwardFlowAnalysis {
2      public MyForwardAnalysis(DirectedGraph graph) {
3          super(graph);
4          doAnalysis();
5      }
6      @Override
7      protected void flowThrough(Object in, Object node, Object out) {
8          FlowSet inSet = (FlowSet) in, outSet = (FlowSet) out;
9          Unit u = (Unit) node;
10         gen_filter(inSet, outSet, u);
11         kill_filter(inSet, outSet, u);
12     }
13 }
14
```

flowThrouph实际上就是transfer function

Soot使用——data-flow analysis framework

数据流分析的输入是一个由body构建的CFG (control-flow graph)

```
// 得到想要分析的类
SootClass c = Scene.v().loadClassAndSupport("MyClass");
c.setApplicationClass();
// 得到想要分析的方法的body
SootMethod m = c.getMethodByName("myMethod");
Body b = m.retrieveActiveBody();
// 构造cfg
UnitGraph g = new ExceptionalUnitGraph(b);
// 执行数据流分析
MyForwardAnalysis an = new MyForwardAnalysis(g);
// 迭代查看结果
Iterator i = g.iterator(); while (i.hasNext()) {
    Unit u = (Unit)i.next();
    List IN = an.getFlowBefore(u);
    List OUT = an.getFlowAfter(u);
    // Do something clever with the results
}

```

此框架只能进行过程内的数据流分析，但可以基于分析结果再自定义规则进行过程间数据流分析。

一些有用的链接

Soot的github地址： https://github.com/soot-oss/soot

Soot的一个较为全面的教程： https://github.com/soot-oss/soot/wiki

Soot的一个入门级教程：https://www.brics.dk/SootGuide/sootsurvivorsguide.pdf

Soot命令行选项列表：https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/options/soot_options.htm#phase_2

Soot的api列表：https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/jdoc/

Soot的一个工程示例：https://blog.csdn.net/liu3237/article/details/48827523

Soot的一个比较详细的介绍：https://zhuanlan.zhihu.com/p/79801764

FlowDroid

简介

"FlowDroid is a context-, flow-, field-, object-sensitive and lifecycle-aware static taint analysis tool for Android applications."

Context-sensitive：过程间分析时考虑函数调用的上下文

Flow-sensitive：考虑程序语句的执行顺序

field-sensitive：能够区分类内的不同成员变量

Object-sensitive：能够区分到达同一个位置的不同对象

Taint analysis

污点分析（taint analysis）是一种跟踪并分析污点信息在程序中流动的技术。将感兴趣的数据标记为污点，跟踪其流向可以知道其是否会影响某些关键程序操作，进而发现程序漏洞。

利用特定的规则跟踪分析污点信息在程序中的传播过程

识别污点信息在程序中的产生点（Source点）并对污点信息进行标记

在一些关键的程序点（Sink点）检测关键的操作是否会受到污点信息的影响

Flowdroid介绍

Precise Modelling of Lifcycle
- Callbacks
- Dummy main

Precise Flow-Sensitive Analysis
- IFDS
- Alias analysis

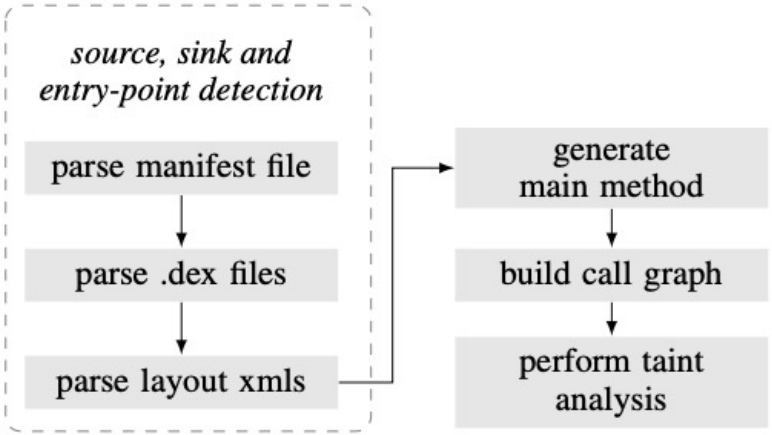
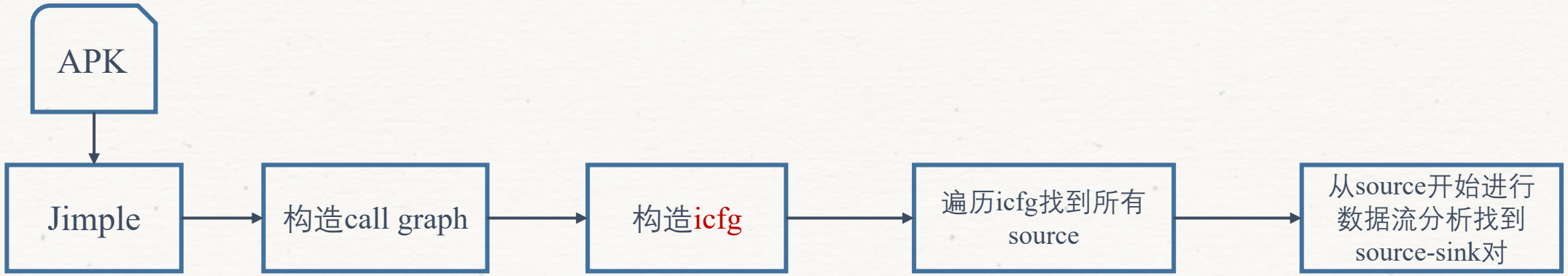


Figure 4: Overview of FLOWDROID



Inter-procedure control-flow graph

Flowdroid——通过命令行使用

各个版本的AndroidSDK，即android platforms

设置SourceAndSink.txt、EasyTaintWrapperSource.txt、AndroidCallbacks.txt用于指定监控的数据源以及泄漏点等信息

下载release版本：soot-infoflow-cmd-jar-with-dependencies.jar

自行下载所有的依赖包以及flowdroid包：sootclasses-trunk-jar-with-dependencies.jar,soot-infoflow-android.jar,soot-infoflow.jar,slf4j-api-1.7.5.jar,axml-2.0.jar

运行命令得到污点分析结果

java –jar soot-infoflow-cmd-jar-with-dependencies.jar -a <apkfile> -p <android-sdk-folder> -s <SourcesSinks.txt>

java -Xmx4g -cp soot-trunk.jar:soot-infoflow.jar:soot-infoflow-android.jar:slf4j-api-1.7.5.jar:slf4j-simple-1.7.5.jar:axml-2.0.jar soot.jimple.infoflow.android.TestApps.Test <apkfile> <android-sdk-folder>

Flowdroid——通过eclipse工程构建

- 开一个maven工程
- 从github-wiki上下载所有项目：heros, jasmin, soot, soot-infoflow, soot-infoflow-android，依次导入工程
- Build工程，中间可能遇到各种各样的问题，比如：maven配置、需要额外导入依赖jar包、部分文件重复等等，，需要一一解决
- 运行soot.jimple.infoflow.android.TestApps中的test（通过设置运行参数来指定apk以及android platforms），启动污点分析

最好的办法是直接用已有的工程，在此基础上修改

Source and Sink

Flowdroid所能处理的source和sink都是函数形式

Source，为敏感数据的生成位置：

<android.database.Cursor: java.lang.String getString(int)> -> _SOURCE_

<android.content.pm.PackageManager: java.util.List getInstalledApplications(int)> -> _SOURCE_

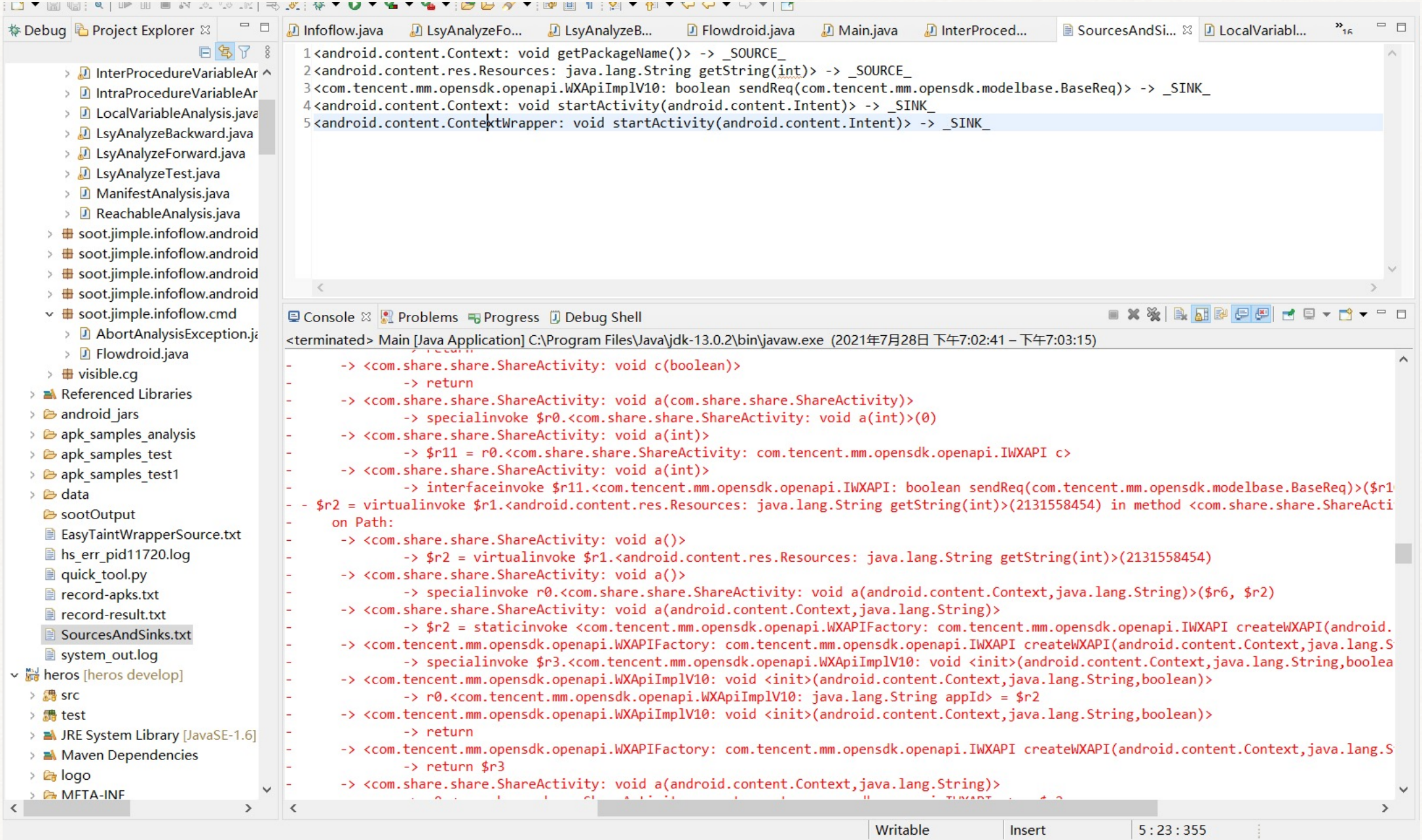
Sink，为敏感数据的泄漏位置：

<android.content.Intent: android.content.Intent putExtra(java.lang.String,int)> -> _SINK_

<android.content.Context: void sendBroadcast(android.content.Intent,java.lang.String)> -> _SINK_

最终flowdroid会针对给定的SourceAndSinks.txt返回一个分析结果，说明哪些source-sink存在可达路径，并将路径打印出来

实际效果展示


Debug
Project Explorer
InterProcedureVariableAr
IntraProcedureVariableAr
LocalVariableAnalysis.java
LsyAnalyzeBackward.java
LsyAnalyzeForward.java
LsyAnalyzeTest.java
ManifestAnalysis.java
ReachableAnalysis.java
soot.jimple.infoflow.android
soot.jimple.infoflow.android
soot.jimple.infoflow.android
soot.jimple.infoflow.android
soot.jimple.infoflow.cmd
AbortAnalysisException.ja
Flowdroid.java
visible.cg
Referenced Libraries
android_jars
apk_samples_analysis
apk_samples_test
apk_samples_test1
data
sootOutput
EasyTaintWrapperSource.txt
hs_err_pid11720.log
quick_tool.py
record-apks.txt
record-result.txt
SourcesAndSinks.txt
system_out.log
heros [heros develop]
src
test
JRE System Library [JavaSE-1.6]
Maven Dependencies
logo
META-INF
Infoflow.java
LsyAnalyzeFo...
LsyAnalyzeB...
Flowdroid.java
Main.java
InterProced...
SourcesAndSi...
LocalVariabl...
1 <android.content.Context: void getPackageName()> -> _SOURCE_
2 <android.content.res.Resources: java.lang.String getString(int)> -> _SOURCE_
3 <com.tencent.mm.opensdk.openapi.WXApiImplV10: boolean sendReq(com.tencent.mm.opensdk.modelbase.BaseReq)> -> _SINK_
4 <android.content.Context: void startActivity(android.content.Intent)> -> _SINK_
5 <android.content.ContextWrapper: void startActivity(android.content.Intent)> -> _SINK_
Console
Problems
Progress
Debug Shell
<terminated> Main [Java Application] C:\Program Files\Java\jdk-13.0.2\bin\javaw.exe (2021年7月28日 下午7:02:41 – 下午7:03:15)
- -> <com.share.share.ShareActivity: void c(boolean)>
- -> return
- -> <com.share.share.ShareActivity: void a(com.share.share.ShareActivity)>
- -> specialinvoke $r0.<com.share.share.ShareActivity: void a(int)>(0)
- -> <com.share.share.ShareActivity: void a(int)>
- -> $r11 = r0.<com.share.share.ShareActivity: com.tencent.mm.opensdk.openapi.IWXAPI c>
- -> <com.share.share.ShareActivity: void a(int)>
- -> interfaceinvoke $r11.<com.tencent.mm.opensdk.openapi.IWXAPI: boolean sendReq(com.tencent.mm.opensdk.modelbase.BaseReq)>($r1
- - $r2 = virtualinvoke $r1.<android.content.res.Resources: java.lang.String getString(int)>(2131558454) in method <com.share.share.ShareActi
- on Path:
- -> <com.share.share.ShareActivity: void a()>
- -> $r2 = virtualinvoke $r1.<android.content.res.Resources: java.lang.String getString(int)>(2131558454)
- -> <com.share.share.ShareActivity: void a()>
- -> specialinvoke r0.<com.share.share.ShareActivity: void a(android.content.Context,java.lang.String)>($r6, $r2)
- -> <com.share.share.ShareActivity: void a(android.content.Context,java.lang.String)>
- -> $r2 = staticinvoke <com.tencent.mm.opensdk.openapi.WXAPIFactory: com.tencent.mm.opensdk.openapi.IWXAPI createWXAPI(android.
- -> <com.tencent.mm.opensdk.openapi.WXAPIFactory: com.tencent.mm.opensdk.openapi.IWXAPI createWXAPI(android.content.Context,java.lang.S
- -> specialinvoke $r3.<com.tencent.mm.opensdk.openapi.WXApiImplV10: void <init>(android.content.Context,java.lang.String,boolea
- -> <com.tencent.mm.opensdk.openapi.WXApiImplV10: void <init>(android.content.Context,java.lang.String,boolean)>
- -> r0.<com.tencent.mm.opensdk.openapi.WXApiImplV10: java.lang.String appId> = $r2
- -> <com.tencent.mm.opensdk.openapi.WXApiImplV10: void <init>(android.content.Context,java.lang.String,boolean)>
- -> return
- -> <com.tencent.mm.opensdk.openapi.WXAPIFactory: com.tencent.mm.opensdk.openapi.IWXAPI createWXAPI(android.content.Context,java.lang.S
- -> return $r3
- -> <com.share.share.ShareActivity: void a(android.content.Context,java.lang.String)>
Writable
Insert
5 : 23 : 355

一些有用的链接

Flowdroid论文： https://orbilu.uni.lu/bitstream/10993/20223/1/far%2B14flowdroid.pdf

Flowdroid简介： https://blogs.uni-paderborn.de/sse/tools/flowdroid/

Flowdroid的github链接：https://github.com/secure-software-engineering/FlowDroid

Flowdroid的github-wiki构建官方教程：https://github.com/secure-software-engineering/soot-infoflow-android/wiki

Flowdroid 扩展

扩展思路

如果污点source是一个常量字符串，而Flowdroid只支持函数形式的source和sink，需要对其进行改进，改进思路有两条：

在寻找source的过程中，把常量形式的source也作为目标寻找，之后按照原本的数据流分析进行。

在flowdroid构建的精确icfg的基础上自己设定数据流分析规则进行分析。

举例

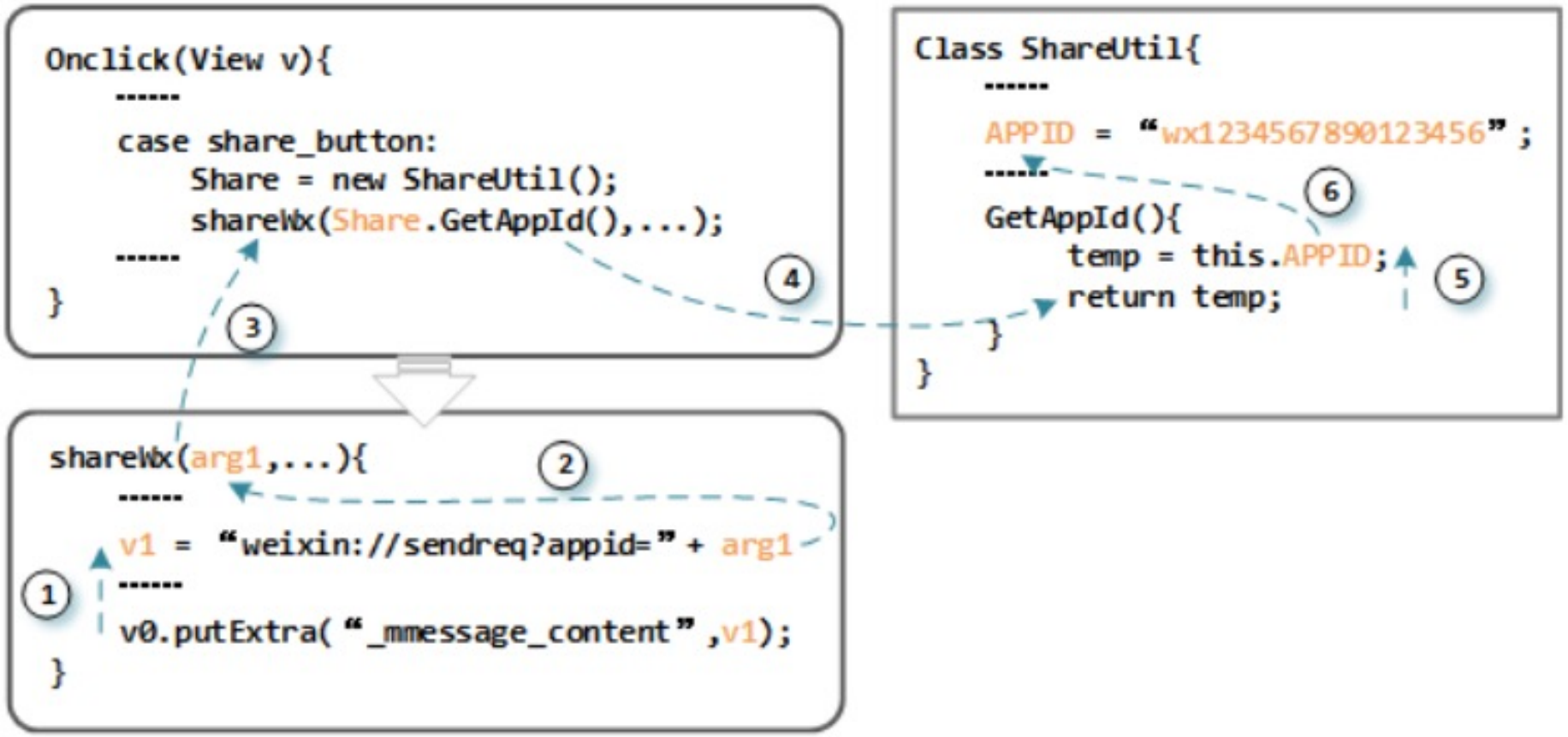


Figure 5: Backward data flow analysis.

构建过程内数据流分析

Algorithm 1 Intra-procedure Backward Data-flow Analysis

Input: CFG (def_S and use_S computed for each statement S)
Output: $IN[S]$ and $OUT[S]$ for each statement S
  for each statement S **do**
    $OUT[S] = \cup_{S' \ a\ successor\ of\ S} IN[S']$
    $IN[S] = OUT[S] - def_S$
    if S is a Invoke Expression **then**
      if S is equal to "append" **then**
        $IN[S] = IN[S] \cup params_S$
        $IN[S] = IN[S] - noparams_S$
      else if S is equal to "toString" or "split" **then**
        $IN[S] = IN[S] \cup use_S$
      end if
    else if S is not Field Reference or Native Invoke Expression **then**
      $IN[S] = IN[S] \cup use_S$
    end if
  end for

与活跃变量分析类似

Algorithm 2 Intra-procedure Forward Data-flow Analysis in De-Fash

Input: CFG (def_S and use_S computed for each statement S)
Output: $IN[S]$ and $OUT[S]$ for each statement S
  for each statement S **do**
    $IN[S] = \cup_{S' \ a\ predecessor\ of\ S} OUT[S']$
    $OUT[S] = IN[S] - def_S$
    if use_S in IN[S] **then**
      $OUT[S] \cup def_S$
    end if
    if S is Invoke Expression and no activeBody **then**
      $OUT[S] \cup uselocal_S$
    end if
  end for

与到达定值问题类似

构建一个过程间数据流分析（以backward举例）

- If it is a parameter assignment statement, we find all callers of the current method in ICFG, and set this parameter as a live variable continue to perform backward data-flow analysis in callers.
- If it is an invoke statement with no parameters, then we search for all return statements in the callee and set the variables in these statement as live variables for backward data-flow analysis in callee.
- If it is a member variable assignment statement, then we search for the initial value of this variable in the initialization methods <init> and <clinit> of the Class in which this field variable belongs.

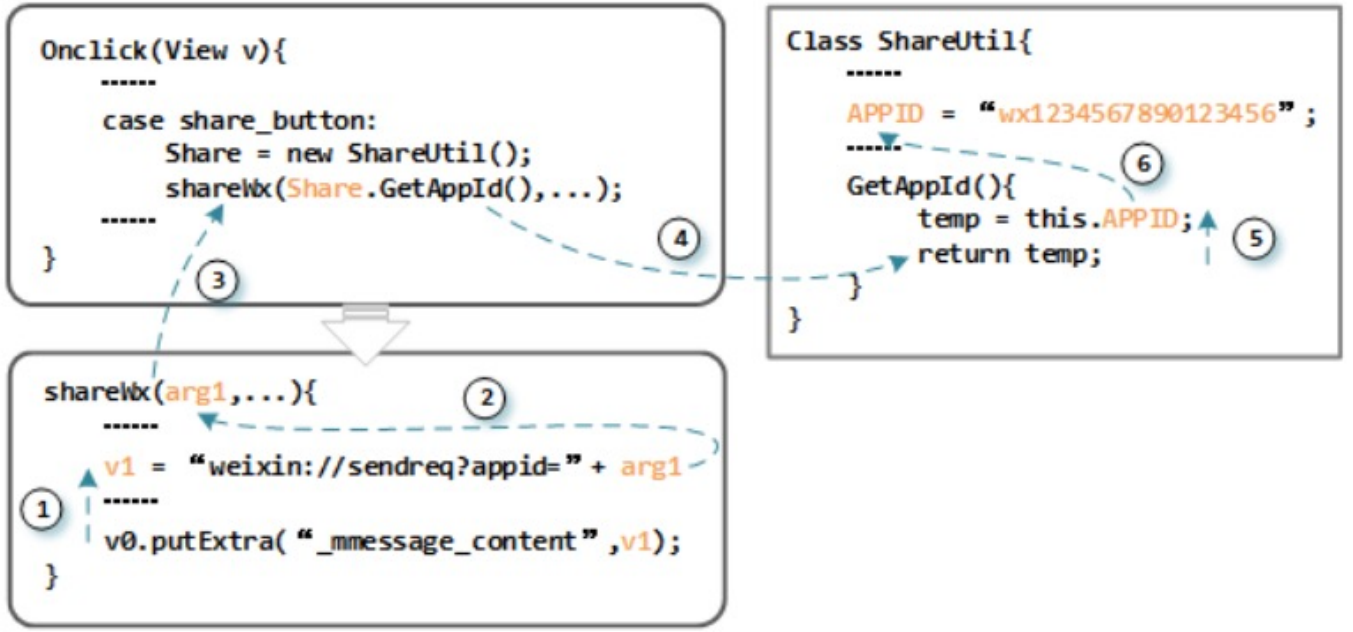


Figure 5: Backward data flow analysis.

Thank you

感谢聆听