Implementation and Evaluation of a Static Backwards Data Flow Analysis in FlowDroid

Implementierung und Evaluation einer statischen rückwärtsgerichteten Datenflussanalyse in FlowDroid
Bachelor thesis by Tim Lange
Date of submission: January 24, 2021

1. Review: Dr. Steven Arzt
2. Review: Prof. Dr. Michael Waidner
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Fraunhofer
SIT

Computer Science
Department
Institut
Arbeitsgruppe

Contents

1 Introduction

2 Background

2.1 Data Flow Analysis

Explain key terms such as taint, source, sink, leak

2.2 IFDS & Practical Extensions

2.3 Intermediate Representations

Explain what jimple and why it is useful to operate on an IR

- Like 25 possible statements instead of way too many instructions
- Everything is explicit. No implicit writes whatsoever

2.4 Soot

just short, but probably needs to be introduced before FlowDroid and especially before clinit rule

2.5 FlowDroid

3 Theory

3.1 Complexity of Data Flow Analysis

Explain where the run-time comes from. Depends the number of edge propagations

- "Branching factor" might be different for forwards/backwards, with some simple examples?
  - tainted = a + b. BW we don’t know which was responsible for the tainted c → 2 new taints
  - Simple assigments in a strict r-to-l order: a = b. FW a, b while BW we can kill a and just go with b
- Lifetime of taints
  - Static taints are valid everywhere
  - Best practise "sanitize just before displaying" might favor backwards
- Number of taints
  - There seems to be no correlation between source count and analysis time
  - Probably also holds for sinks?
  - There might be indicator for a single app whether it is better to start at sources or sinks

3.2 Flow Functions

3.2.1 Normal Flow

In the following, we consider an assignment of the structure $x.f^n = y.g^m$ with $n, m \in \{0, 1\}$.

First, we take a look at the left hand side. If the incoming taint $t = x.f^n$

- If T={a} and a = b, T'={b}
- If T={b} and a = b, T={b} and alias triggered
- ...

3.2.2 Call Flow

3.2.3 Return Flow

3.2.4 CallToReturn Flow

4 Implementation

4.1 Integration

Document changes needed in non problems/ to fit backwards.

- BackwardsSourceSinkManager
- BackwardsInfoflowResults

4.2 InfoflowProblem

4.3 Rules

4.3.1 Backwards Source Propagation Rule

4.3.2 Backwards Clinit Rule

Soot creates CG edges also at AssignStmts given there is a static variable involved. These AssignStmts are not considered as CallStmts by FlowDroid. Manual injection...

```
public class ConstantTestCode {
    static final String tainted =
       TelephonyManager.getDeviceId();

    public void easyConstantFieldTest(){
        ConnectionManager cm = new ConnectionManager();
        cm.publish(tainted);
    }
}
```

Figure 4.1: ConstantTestCode excerpt

4.4 Code Optimizer

Before starting the analysis, FlowDroid applies code optimization to the interprocedural call graph. By default, dead code elimination and within constant value propagation is performed. Those are also applied before backwards analysis but we needed another code optimizer to handle an edge case in backwards analysis.

4.4.1 AddNOPStmts

First, take a look at StaticTestCode#static2Test in Figure 4.2. The method and entry point static2Test is static and does not has any parameters. Same is true for the source method TelephonyManager#getDeviceId. Due to these conditions, static2Test in Jimple has neither identity statements nor assign statements before the source statement and therefore the source statement is the first statement in the graph. Next, a detail of FlowDroid's IFDS solver is important. The Return and CallToReturn flow function is only applied if a return site is available. When searching backwards, the source statement is the last statement and thus has no return sites. Now recall subsection 4.3.1, taints flowing into sources are registered in the CallToReturn flow function. Altogether, leaks can not be found if the source statement is the first statement.

Moving the detection of incoming taints flows into sources from the CallToReturn to the Call flow function was not an option because by default source methods are not visited. Our solution is to just add a NOP statement in such cases. This saves us from introducing

```java
ic static void static2Test() {
String tainted = TelephonyManager.getDeviceId();
ClassWithStatic static1 = new ClassWithStatic();
static1.setTitle(tainted);
ClassWithStatic static2 = new ClassWithStatic();
String alsoTainted = static2.getTitle();

ConnectionManager cm = new ConnectionManager();
cm.publish(alsoTainted);
```

Figure 4.2: static2Test Java Code

new edge cases inside the flow functions which are already complex enough. Due to the entry points being known beforehand, the overhead is negligible.

5 Validation

5.1 DroidBench

5.1.1 Results

App Name	Forwards	Backwards
Aliasing		
FlowSensitivity1		⋆
Merge1	⋆	⋆
SimpleAliasing1	⊛	⊛
StrongUpdate1		
Arrays and Lists		
ArrayAccess1	⋆	⋆
ArrayAccess2	⋆	⋆
ArrayAccess3	⊛	⊛
ArrayAccess4		
ArrayAccess5		⋆
ArrayCopy1	⊛	○
ArrayToString1	⊛	⊛
HashMapAccess1	⋆	⋆
ListAccess1	⋆	⋆
MultidimensionalArray1	⊛	⊛
Callbacks		
AnonymousClass1	⊛	⊛ ⋆
Button1	⊛	⊛
Button2	⊛⊛⊛ ⋆	⊛○○
Button3	⊛⊛	⊛⊛
Button4	⊛	⊛

App Name	Forwards	Backwards
Button5	⊛	⊛
LocationLeak1	⊛⊛	⊛⊛
LocationLeak2	⊛⊛	⊛⊛
LocationLeak3	⊛	⊛ ★
MethodOverride1	⊛	⊛
MultiHandlers1		
Ordering1		
RegisterGlobal1	⊛	⊛
RegisterGlobal2	⊛	⊛
Unregister1	★	★
Emulator Detection		
Battery1	⊛	⊛
Bluetooth1	⊛	⊛
Build1	⊛	⊛
Contacts1	⊛	⊛ ★
ContentProvider1	⊛⊛	⊛○
DeviceId1	⊛	⊛
File1	⊛	⊛
IMEI1	⊛⊛	○○
IP1	⊛	⊛
PI1	⊛	⊛
PlayStore1	⊛⊛	⊛
PlayStore2	⊛	⊛
Sensors1	⊛	⊛
SubscriberId1	⊛	⊛ ★
VoiceMail1	⊛	⊛
Field and Object Sensitivity		
FieldSensitivity1		
FieldSensitivity2		
FieldSensitivity3	⊛	⊛
FieldSensitivity4		
InheritedObjects1	⊛	⊛
ObjectSensitivity1		★
ObjectSensitivity2		
Inter-Component Communication		
ActivityCommunication1	⊛	⊛

App Name	Forwards	Backwards
ActivityCommunication2	⍟ ⋆	○
ActivityCommunication3	⍟ ⋆	○
ActivityCommunication4	⍟ ⋆	○
ActivityCommunication5	⍟ ⋆	○
ActivityCommunication6	⍟ ⋆	○
ActivityCommunication7	⍟ ⋆	○
ActivityCommunication8	⍟	
BroadcastTaintAndLeak1	⍟	⍟
ComponentNotInManifest1	⋆	
EventOrdering1	○ ⋆	○ ⋆
IntentSink1	⍟	○
IntentSink2	⍟	○
IntentSource1	⍟⍟	○○
ServiceCommunication1	⍟	○
SharedPreferences1	○	⍟
Singletons1	○	⍟
UnresolvableIntent1	⍟⍟	○○
Lifecycle		
ActivityEventSequence1	⍟	⍟
ActivityEventSequence2	⍟	○
ActivityEventSequence3	⍟	○
ActivityLifecycle1	⍟	⍟
ActivityLifecycle2	⍟	⍟
ActivityLifecycle3	⍟	⍟
ActivityLifecycle4	⍟	⍟
ActivitySavedState1	⍟	⍟
ApplicationLifecycle1	⍟	⍟
ApplicationLifecycle2	⍟	⍟
ApplicationLifecycle3	⍟	⍟
AsynchronousEventOrdering1	⍟	⍟
BroadcastReceiverLifecycle1	⍟	⍟
BroadcastReceiverLifecycle2	○	⍟
BroadcastReceiverLifecycle3	⍟	⍟
EventOrdering1	⍟	⍟
FragmentLifecycle1	○	○
FragmentLifecycle2	○	○

App Name	Forwards	Backwards
ServiceEventSequence1	○	○
ServiceEventSequence2	○	○
ServiceEventSequence3	⊛	⊛
ServiceLifecycle1	⊛	⊛
ServiceLifecycle2	⊛	⊛
SharedPreferenceChanged1	⊛	⊛
General Java		
Clone1	⊛	⊛
Exceptions1	⊛	⊛
Exceptions2	⊛	⊛
Exceptions3	⋆	⋆
Exceptions4	⊛	⊛
Exceptions5	⊛	⊛
Exceptions6	⊛	⊛
Exceptions7		
FactoryMethods1	⊛⊛	⊛⊛ ⋆
Loop1	⊛	⊛
Loop2	⊛	⊛
Serialization1	○	○
SourceCodeSpecific1	⊛	⊛
StartProcessWithSecret1	⊛	⊛
StaticInitialization1	○	⊛
StaticInitialization2	⊛	○
StaticInitialization3	○	○
StringFormatter1	○	⊛
StringPatternMatching1	⊛	⊛
StringToCharArray1	⊛	○
StringToOutputStream1	⊛	⊛
UnreachableCode		
VirtualDispatch1	⊛ ⋆	⊛
VirtualDispatch2	⊛ ⋆	⊛
VirtualDispatch3	⋆	⋆
VirtualDispatch4		
Miscellaneous Android-Specific		
ApplicationModeling1	⊛	⊛
DirectLeak1	⊛	⊛

App Name	Forwards	Backwards
InactiveActivity		
Library2	⊛	⊛
LogNoLeak		
Obfuscation1	⊛	⊛
Parcel1	⊛	○
PrivateDataLeak1	⊛	○
PrivateDataLeak2	⊛	⊛
PrivateDataLeak3	○	○
PublicAPIField1	⊛	⊛
PublicAPIField2	⊛	⊛
View1	⊛	⊛
Reflection		
Reflection1	⊛	⊛
Reflection2	⊛	⊛
Reflection3	⊛	⊛
Reflection4	⊛	⊛
Reflection5	⊛	⊛
Reflection6	⊛	⊛
Reflection7	○	⊛
Reflection8	⊛	⊛
Reflection9	⊛	⊛
Threading		
AsyncTask1	⊛	⊛
Executor1	⊛	⊛
JavaThread1	⊛	⊛
JavaThread2	⊛	⊛
Looper1	⊛	⊛
TimerTask1	⊛	⊛

5.1.2 Discussion

Button2

Found 4 paths like in forwards but built into one.

6 Evaluation

6.1 Configuration

Test setup... Test server is shared, so use less cores than available to minimize variation due to background tasks?

6.2 Performance

Basically the answer to RQ1: Is the backwards search efficient enough to perform analysis on real world apps?

6.3 Comparison to forwards analysis

Basically the answer to RQ2: Can we find a pre-analysis known parameter to decide which analysis is more efficient?

7 Conclusion