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: February 5, 2021

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Fraunhofer SIT

Computer Science Department
Fraunhofer SIT
Secure Software Engineering

Contents

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Tim Lange, die vorliegende Bachelorarbeit gemäß §22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, February 5, 2021

T. Lange

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 Flow Functions

In this section, we describe the behavior of the flow functions based on the Jimple language and define semi-formal rules.

3.1.1 Normal Flow

Normal flow functions handle every statement that does not contain an `InvokeExpr`. The only case where a new taint can be produced is at an `AssignStmt`. It is straight-forward that this is true for statements like `IfStmt` if we recall section 2.3. The conditition is either an `UnopExpr` or `BinopExpr` of which both have no effect on the taint set. But we also skip over `IdentityStmt` even though they define a value. This is because we wait for the return site to map all parameters back into the callee.

Now, lets consider the current statement is an `AssignStmt`. It consists of a variable, either a reference or a local, on the left side and an expression on the right side. Jimple ensures we just see one field reference at a time but to reduce the semi-formal rules, we take a shortcut here. So our assigment has the structure $x.f^n \leftarrow y.g^m$ with $n, m \in \{0, 1\}$ modelling a possible field reference. Note that the taints can have an access path of an arbitrary length k which is denoted as h^k.

First, we look at the case when the access path matches exactly. Either we have a local $(n = 0)$ or a field reference $(n = 1)$ on the left. In the first case, the base of our taint needs to match and in the latter, the first field must also match. If the field references another heap object, we might encounter a non-empty access path h^k. This access path needs to be added to the newly created taint. We conclude:

Rule 1: An incoming taint $t = x.f^n.h^k$ with $k \geq 0$ produces the outflowing taint set $T = \{y.g^m.h^k\}$.

Next, we might have a whole object tainted. In this case, just the base needs to match:

Rule 2: An incoming taint $t = x.*$ with $k \geq 0$ produces the outflowing taint set $T = \{y.g^m.*\}$.

Lastly, the right side could also be tainted:

Rule 3: An incoming taint $t = y.g^m.h^k$ with $k \geq 0$ produces the outflowing taint set $T = \{t\}$.

Whenever the taint neither matches on the left nor on the right side, we propagate it further untouched.

Rule 1 and Rule 3 also work with $*$ appended.

3.1.2 Call Flow

For call statements, we have statements of the structure $o.m(a_0, ..., a_n)$ with $n \in \mathbb{N}$. a_i denotes the i-th argument, p_i the i-th parameter and c the class the method is defined in.

If we encounter a tainted argument in the caller, the taint need to go through the callee. Due to the backwards direction this is only true for heap objects because only they have references. For primitives or strings we already know the tainted value is not visible in the callee.

Rule 1: An incoming taint $t = a_i.h^k$ with $k \geq 0 \wedge 0 \leq i \leq n \wedge \text{typeof}(a_i) \in \mathit{HeapTypes}$ produces the outflowing taint set $T = \{p_i.h^k\}$.

If the object the method is called on is tainted, the tainted path is visible inside the callee. The callee must be not static.

Rule 2: An incoming taint $t = o.h^k$ with $k \geq 0$ produces the outflowing taint set $T = \{\mathit{this}_c.h^k\}$.

Tainted static fields are propagated untouched and unconditionally in the callee as they are always visible.

Rule 3: An incoming taint $t = S.h^k$ with $k \geq 0$ produces the outflowing taint set $T = \{t\}$.

Next, if the call statement is also an assign statement and the left side is tainted we also need to taint the return value. Methods can have multiple return statements and as we traverse the reversed interprocedural control flow graph, we need to taint all possible return values. The structure of the statement is in this case $x \leftarrow o.m(a_0, ..., a_n)$. r_i denotes a return value. n is the number of return statements in the callee.

> **Rule 4:** An incoming taint $t = x.h^k$ with $k \geq 0$ produces the outflowing taint set $T = \{r_i.h^k \mid 0 \leq i < n\}$.

The taint is killed if it is not matched inside a rule. Instead, it is propagated over the call statement in the CallToReturn flow function.

3.1.3 Return Flow

All taints reaching the end of a callee need to be mapped back into the caller. The statement is of the structure $o.m(a_0, ..., a_n)$ with $n \in \mathbb{N}$. a_i denotes the i-th argument, p_i the i-th parameter and c the class the method is defined in.

First, we match rule 1 of call flow and map all parameters back into the caller. This time even primitives are mapped back because if we find a tainted value at the start of the method it had to be passed as an argument into the method.

> **Rule 1:** An incoming taint $t = p_i.h^k$ with $k \geq 0 \wedge 0 \leq i \leq n$ produces the outflowing taint set $T = \{a_i.h^k\}$.

The *this* reference is visible in the caller. This is the reverse of rule 2 in call flow.

> **Rule 2:** An incoming taint $t = this_c.h^k$ with $k \geq 0$ produces the outflowing taint set $T = \{o.h^k\}$.

Tainted static fields are also mapped back untouched and unconditionally equivalent to rule 3 in call flow.

> **Rule 3:** An incoming taint $t = S.h^k$ with $k \geq 0$ produces the outflowing taint set $T = \{t\}$.

The taint is killed if it is not matched in a rule.

3.1.4 CallToReturn Flow

As already seen in call flow, not every taint is visible inside a callee. Again, the statement structure is $o.m(a_0, ..., a_n)$ with $n \in \mathbb{N}$. a_i denotes the i-th argument.

If the taint neither matches an argument nor the object the method is called on, it is not visible in the callee. Static fields are always visible and thus can not propagated over a statement.

> **Rule 1:** An incoming taint $t = x.h^k$ with $k \geq 0 \wedge (\forall a \in Arguments : a \neq x) \wedge x \neq o \wedge x \notin Static$ produces the outflowing taint set $T = \{t\}$.

If a taint is limited to its base, so no fields are tainted, the taint is also propagated over the statement as the reference is passed by copy-by-value and assigments to the parameter overwrites the reference in the callee but has no effect on the reference in the caller.

> **Rule 2:** An incoming taint $t = a_i$ with $0 \leq i \leq n$ produces the outflowing taint set $T = \{t\}$.

3.2 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 $\rightarrow$ 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

4 Implementation

4.1 Integration

FlowDroid is built to be extensible from to ground up. We wanted to reuse as much components of FlowDroid as possible. For the backwards analysis, we introduce unconditional taints at sinks and check for the matching access paths at sources. Facts are propagated through a reversed interprocedural control flow graph.

The methods for retrieving sources and sinks from a SourceSinkManager have different signatures because only at one end the access paths must match and at the other the taints are unconditional. We added the interface `IReversibleSourceSinkManager` extending the `ISourceSinkManager`. It enforces two additional methods:

- `SourceInfo getInverseSinkInfo(Stmt sCallSite, InfoflowManager manager)`
- `SinkInfo getInverseSourceInfo(Stmt sCallSite, InfoflowManager manager, AccessPath ap)`

`getInverseSinkInfo` returns the necessary information for introducing unconditional taints at sinks while `getInverseSourceInfo` also matches the access paths at sources. All three source sink managers `DefaultSourceSinkManager` for modelling Java, `AccessPathBasedSourceSinkM` for modelling Android and `SummarySourceSinkManager` for summaries now implement the `IReversibleSourceSinkManager` interface. Reversible source sink manager currently do not support the one-source-at-a-time mode.

Due to the flow-sensitive aliasing of FlowDroid using IFDS, FlowDroid already provides an implementation of a reversed interprocedural control flow graph called `BackwardsInfoflowCFG`. For the core - the flow functions - we created two new components implementing `IInfoflowProblem`: the backwards infoflow problem and an alias problem. More on that in section 4.2.

To hide the fact that we internally swapped the sources and sinks, we also created a `BackwardsInfoflowResults` extending `InfoflowResults`. The implementation is quite simple. It overwrites the `addResult` implementations and reverses the constructed paths.

The modularity of FlowDroid allowed us to easily use the newly created components. We created another implementation of `IInfoflow` responsible for initialization of those closely to the already existing default implementation `Infoflow`.

4.2 Problems

4.2.1 Flow-Sensitive Alias Analysis

FlowDroid offers multiple aliasing strategies. In this work, we focus on the flow-sensitive alias analysis which is implemented as another IFDS problem called `BackwardsAliasProblem`. Basically, this is a forwards IFDS search with flow functions using aliasing rules.

Handover to Alias Analysis Whenever we visit a statement and notice a taint could have an alias, the taint is handed over to the alias analysis. Normal flow rule 3 is such a case. The taint is on the right side and we notice that the left side also refers to the same value in memory due to being stored in the heap. The left side gets tainted and propagated forwards to find out if we missed a write to the alias. In normal flow rule 1 and 2, we also turn around. Figure 4.1 shows two cases where the turnaround is necessary.

```
void aliasRule1() {
    A a = b;
    b.str = source();
    sink(a.str);
}
```

(a) Example for alias analysis initiated by rule 1

```
void aliasRule3() {
    A a = b;
    a.str = source();
    sink(b.str);
}
```

(b) Example for alias analysis initiated by rule 3

Figure 4.1: Aliasing examples

Handing back to Infoflow

TurnUnit We added another field to the `Abstraction` class called `turnUnit`. This is the equivalent to the `activationUnit` in forwards analysis. The `turnUnit` references the last statement for which the taint is relevant for the infoflow search. At start, it is the sink it originated from. Later on, it is set whenever we visit an assigment with a primitive or string on the left side. An example can be found in Figure 4.2. Line 5 introduces the taint, line 3 taints $b.str$ and sets the turnUnit to this statement. In line 2, `a` is found to be an alias of `b` and causes a handover to the alias problem. The `turnUnit` now stops the alias search at line 3 and prevents a false positive.

```
1 void turnStmtNeeded() {
2     A a = b;
3     String str = b.str;
4     a.str = source();
5     sink(str);
6 }
```

Figure 4.2: Aliasing example with turn unit

Explain TurnUnit, SkipUnit What the core problem tackles

4.3 Rules

Flow functions can get quite large, complicated to understand and hard to maintain [3]. To counteract this, FlowDroid outsources certain features into rules. These rules also provide the four flow functions and are applied in the corresponding flow function.

4.3.1 Backwards Sink Propagation Rule

4.3.2 Backwards Source Propagation Rule

4.3.3 Backwards Array Propagation Rule

4.3.4 Backwards Exception Propagation Rule

4.3.5 Backwards Wrapper Propagation Rule

4.3.6 Backwards Implicit Propagation Rule

Not implemented.

4.3.7 Backwards Strong Update Rule

4.3.8 Backwards Clinit Rule

`<clinit>` is a special method in the JVM and stands for class loader init. The function is generated by the compiler and can not be called explicitly. Examples of statements which get compiled into clinit can be seen in Figure 4.3. The invokation is implicit at the initialization phase of the class and is executed at most once for each class [1]. This behavior is modelled as an overapproximation in FlowDroid's default call graph algorithm SPARK. SPARK adds an edge to `<clinit>` at each statement containing a `StaticFieldRef`, `StaticInvokeExpr` or `NewExpr` [2].

The need for this rule is rooted in the IFDS solver of FlowDroid. The solver decides whether to use normal flow or call flow by calling `isCallStmt(Unit u)` on the interprocedural control flow graph generated by Soot. Internally, this method calls `containsInvokeExpr()` on the `Unit` object. `containsInvokeExpr()` for `AssignStmt` only returns true if the right hand side is an instance of `InvokeExpr`. Resulting, we miss the call to `<clinit>` for AssignStmts with NewExpr or StaticFieldRef on the right side.

[1] https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.9

[2] https://github.com/soot-oss/soot/blob/59931576784b910a7d38f81910b7313aa2feafea/src/main/java/soot/jimple/toolkits/callgraph/OnFlyCallGraphBuilder.java#L969

```java
class ClinitClass1 {
    public static string str =
        source();
}
```

(a) static variable initialization

```java
class ClinitClass2 {
    static {
        ClinitClass2.sink();
    }
}
```

(b) static block

Figure 4.3: Examples of statements being in <clinit>

The Backwards Clinit Rule manually injects an edge to the <clinit> method in the infoflow solver when appropriate during the analysis. Also, it lessens the overapproximation of SPARK by carefully choosing whether to inject the edge. The rule works as follows:

- If the tainted static variable is a field of the methods class: Do not inject because we will at least encounter a `NewExpr` of the same class further in the call graph.
- Else if the tainted static variable matches the `StaticFieldRef` on the right hand side: Inject the edge because we can not be sure whether we see another edge to <clinit>.
- Else if the class of the tainted static variable matches the class of the `NewExpr`: Inject the edge because we can not be sure whether we see another edge to <clinit>.

This is still an overapproximation of course. A precise solution would require bookkeeping of the first occurence in the code of every class.

This rule has no equivalent in forwards analysis because in fowards analysis the problem is not as severe. As taints are introducted at sources, if the source statement is a static initialization as shown in Figure 4.3a, the propagation starts inside the <clinit> method. The solver has a `followReturnsPastSeeds` option which propagates return flows for unbalanced problems, for example when the taint was introducted inside a method and therefore there was no incoming flow. This allows the forwards analysis to detect leaks originated from static variable initializations but misses leaks inside static blocks as shown in Figure 4.3b.

4.3.9 Other Rules

Skip System Class Rule and Stop After First K Flows Rule are not direction-dependent. Both are shared with the forwards search and therefore use the existing implementation in FlowDroid.

Typing Propagation Rule has no backwards equivalent. We decided to implement type checking in the infoflow problem instead.

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 `StatictTestCode#static2Test` in Figure 4.4. The method and entry point `static2Test` is static and does not have any parameters. Same is true for the source method `TelephonyManager#getDeviceId`. Due to the first condition, `static2Test` has no identity statements and because of the second condition there are also no assign statements before the source statement in Jimple. 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 [1]. When searching backwards, the source statement is the last statement and thus has no return sites. Now recall subsection 4.3.2, 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 new edge cases inside the flow functions which are already complex enough. Due to the entry points being known beforehand, the overhead is negligible.

```java
public 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 4.4: static2Test Java Code

5 Validation

5.1 Unit Tests

FlowDroid already contains 519 unit tests for the core infoflow component. We also validate the backwards analysis with these tests.

Forwards and backwards analysis are not exactly the same. In some cases the results might differ because of limitations or differences in the implementation. In the following textbfs, we provide rationale for these differences.

EasyTaintWrapperTests `equalsTest` and `hashCodeTest` are expected to return one leak but the backwards analysis does report no leaks. This difference is related to the `EasyTaintWrapper` implementation. The implementation marks `equals()` and `hashCode()` as exclusive. This means we can skip this method because we already have a rule for it. The check for exclusiveness is part of the Call and CallToReturn flow function. In both tests, the source is inside the `equals()` or `hashCode()` method. The IFDS solver behaves as already observed in subsection 4.3.8 and when searching forwards it creates a return edge returning from the method while going backwards we do not propagate into the method because it is exclusive. We marked those two tests forwards-specific and created two equivalent backwards-specific tests with sinks inside the `equals()` or `hashCode()` method with one expected leak.

SourceSinkTests These tests ensure the source sink manager can be swapped out. This is not relevant for the correctness of the backwards analysis and therefore are ignored.

5.2 DroidBench

DroidBench is a test suite to evaluate data flow analysis tools targeting the Android ecosystem. It originated from the initial work on FlowDroid to assess it in comparison to other tools [2]. 120 test cases are included in version 2[1]. We do not use it to evaluate our tool against others but to compare it against the forwards analysis of FlowDroid. We aim to achieve similiar results but they may subtle differences.

5.2.1 Configuration

Only using the soot-android-infoflow component, everything else default.

5.2.2 Results

ICC is buggy atm

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

[1] `https://github.com/secure-software-engineering/DroidBench`

App Name	Forwards	Backwards
AnonymousClass1	⊛	⊛
Button1	⊛	⊛
Button2	⊛⊛⊛ ⋆	⊛⊛⊛ ⋆
Button3	⊛⊛	⊛⊛
Button4	⊛	⊛
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		

App Name	Forwards	Backwards
InheritedObjects1 ObjectSensitivity1 ObjectSensitivity2	⊛	⊛
Lifecycle		
ActivityEventSequence1	⊛	⊛
ActivityEventSequence2	⊛	○
ActivityEventSequence3	⊛	○
ActivityLifecycle1	⊛	⊛
ActivityLifecycle2	⊛	⊛
ActivityLifecycle3	⊛	⊛
ActivityLifecycle4	⊛	⊛
ActivitySavedState1	⊛	⊛
ApplicationLifecycle1	⊛	⊛
ApplicationLifecycle2	⊛	⊛
ApplicationLifecycle3	⊛	⊛
AsynchronousEventOrdering1	⊛	⊛
BroadcastReceiverLifecycle1	⊛	⊛
BroadcastReceiverLifecycle2	⊛ ⋆	⊛ ⋆
BroadcastReceiverLifecycle3	⊛ ⋆	⊛
EventOrdering1	⊛	⊛
FragmentLifecycle1	⊛	⊛
FragmentLifecycle2	○	○
ServiceEventSequence1	○	○
ServiceEventSequence2	○	○
ServiceEventSequence3	⊛	○
ServiceLifecycle1	⊛	⊛
ServiceLifecycle2	⊛	⊛
SharedPreferenceChanged1	⊛	⊛
General Java		
Clone1	⊛	⊛
Exceptions1	⊛	⊛
Exceptions2	⊛	⊛
Exceptions3	⋆	⋆
Exceptions4	⊛	⊛
Exceptions5	⊛	⊛
Exceptions6	⊛	⊛

App Name	Forwards	Backwards
Exceptions7		
FactoryMethods1	⊛⊛	⊛⊛
Loop1	⊛	⊛
Loop2	⊛	⊛
Serialization1	○	○
SourceCodeSpecific1	⊛	⊛
StartProcessWithSecret1	⊛	⊛
StaticInitialization1	○	⊛
StaticInitialization2	⊛	○
StaticInitialization3	○	○
StringFormatter1	○	⊛
StringPatternMatching1	⊛	⊛
StringToCharArray1	⊛	○
StringToOutputStream1	⊛ ⋆	⊛ ⋆
UnreachableCode		
VirtualDispatch1	⊛ ⋆	⊛ ⋆
VirtualDispatch2	⊛ ⋆	⊛ ⋆
VirtualDispatch3	⋆	⋆
VirtualDispatch4		
Miscellaneous Android-Specific		
ApplicationModeling1	⊛	⊛
DirectLeak1	⊛	⊛
InactiveActivity		
Library2	⊛	⊛
LogNoLeak		
Obfuscation1	⊛	⊛
Parcel1	⊛	⊛
PrivateDataLeak1	⊛	⊛
PrivateDataLeak2	⊛	⊛
PrivateDataLeak3	⊛○	⊛○
PublicAPIField1	⊛	⊛
PublicAPIField2	⊛	⊛ ⋆
View1	⊛	⊛
Reflection		
Reflection1	⊛	⊛
Reflection2	⊛	⊛

App Name	Forwards	Backwards
Reflection3	⊛	⊛
Reflection4	⊛	⊛
Reflection5	⊛	⊛
Reflection6	⊛	⊛
Reflection7	○	⊛
Reflection8	⊛	⊛
Reflection9	⊛	⊛
Threading		
AsyncTask1	⊛	⊛
Executor1	⊛	⊛
JavaThread1	⊛	⊛
JavaThread2	⊛	⊛
Looper1	⊛	⊛
TimerTask1	⊛	⊛
⊛	106	102
⋆	14	16
○	9	12
Precision	88.33%	86.44%
Recall	92.17%	89.47%
F1 measure	0.9	0.88

5.2.3 Discussion

In this part, we discuss the differences between the forwards and backwards analysis.

Arrays and Lists

ArrayAccess5

ArrayCopy1

ListAccess1

Callbacks

Button2

Emulator Detection

Contacts1

Lifecycle

ActivityEventSequence2

ActivityEventSequence3

BroadcastReceiverLifecycle3

General Java

StaticInitialization1

StaticInitialization2

StaticInitialization3

StringFormatter1

StringToCharArray1

Miscellaneous Android-Specific

Parcel1 PrivateDataLeak1 PublicAPIField2

Reflection

Reflection7

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 Related Work

8 Conclusion

Bibliography

[1] Steven Arzt. “Static Data Flow Analysis for Android Applications”. en. PhD thesis. Darmstadt: Technische Universität, 2017. URL: `https://tuprints.ulb.tu-darmstadt.de/5937/` (visited on 01/28/2021).

[2] Steven Arzt et al. “FlowDroid”. In: *ACM SIGPLAN Notices* 49.6 (June 2014), pp. 259–269. DOI: `10.1145/2666356.2594299`.

[3] Johannes Lerch and Ben Hermann. “Design your analysis: a case study on implementation reusability of data-flow functions”. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM, June 2015. DOI: `10.1145/2771284.2771289`.