

# 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 29, 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

---

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	Data Flow Analysis . . . . .	6
2.2	IFDS & Practical Extensions . . . . .	6
2.3	Intermediate Representations . . . . .	6
2.4	Soot . . . . .	6
2.5	FlowDroid . . . . .	6
<b>3</b>	<b>Theory</b>	<b>7</b>
3.1	Complexity of Data Flow Analysis . . . . .	7
3.2	Flow Functions . . . . .	7
3.2.1	Normal Flow . . . . .	8
3.2.2	Call Flow . . . . .	9
3.2.3	Return Flow . . . . .	9
3.2.4	CallToReturn Flow . . . . .	9
<b>4</b>	<b>Implementation</b>	<b>10</b>
4.1	Integration . . . . .	10
4.2	Problems . . . . .	11
4.3	Rules . . . . .	11
4.3.1	Backwards Sink Propagation Rule . . . . .	11
4.3.2	Backwards Source Propagation Rule . . . . .	11
4.3.3	Backwards Array Propagation Rule . . . . .	11
4.3.4	Backwards Exception Propagation Rule . . . . .	11
4.3.5	Backwards Wrapper Propagation Rule . . . . .	11
4.3.6	Backwards Implicit Propagation Rule . . . . .	11
4.3.7	Backwards Strong Update Rule . . . . .	12
4.3.8	Backwards Clinit Rule . . . . .	12

---

4.3.9	Other Rules . . . . .	13
4.4	Code Optimizer . . . . .	13
4.4.1	AddNOPStmts . . . . .	14
<b>5</b>	<b>Validation</b>	<b>15</b>
5.1	Unit Tests . . . . .	15
5.2	DroidBench . . . . .	16
5.2.1	Configuration . . . . .	16
5.2.2	Results . . . . .	16
5.2.3	Discussion . . . . .	21
<b>6</b>	<b>Evaluation</b>	<b>22</b>
6.1	Configuration . . . . .	22
6.2	Performance . . . . .	22
6.3	Comparison to forwards analysis . . . . .	22
<b>7</b>	<b>Related Work</b>	<b>23</b>
<b>8</b>	<b>Conclusion</b>	<b>24</b>

---

## **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, January 29, 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 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 assignments 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

---

In this section, we describe the behavior of the flow functions based on the Jimple language.

---

---

### 3.2.1 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 condition 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, let's 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 assignment 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.



---

### 3.2.2 Call Flow

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

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  $k \geq 0 \wedge 0 \leq i \leq n \wedge \text{typeof}(a_i) \in \text{HeapTypes}$  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  $T = \{this.h^k\}$ .

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  $x = o.m(a_0, \dots, a_n)$ .  $r_i$  denotes a return value.  $n$  is the count of return statements in the callee.

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

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.

### 3.2.3 Return Flow

### 3.2.4 CallToReturn Flow

---

## 4 Implementation

---

---

### 4.1 Integration

---

FLOWDROID is built to be extensible from the 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, `AccessPathBasedSourceSinkManager` for modelling Android and `SummarySourceSinkManager` for summaries now implement the `IReversibleSourceSinkManager` interface.

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

---

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 4.1. The invocation is implicit at the initialization phase of the class and is executed at most once for each class <sup>1</sup>. 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 <sup>2</sup>.

<pre>1 class ClinitClass1 { 2     public static string str = source(); 3 }</pre>	<pre>1 class ClinitClass2 { 2     static { 3         ClinitClass2.sink(); 4     } 5 }</pre>
(a) static variable initialization	(b) static block

Figure 4.1: Examples of statements being in <clinit>

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.

The Backwards Clinit Rule manually injects an edge to the <clinit> method in the infow flow 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.

---

<sup>1</sup><https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.9>

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

- 
- 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 occurrence in the code of every class.

This rule has no equivalent in forwards analysis because in forwards analysis the problem is not as severe. As taints are introduced at sources, if the source statement is a static initialization as shown in Figure 4.1a, 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 introduced 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 4.1b.

#### 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 infowflow 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.

---

```
1 public static void static2Test() {
2     String tainted = TelephonyManager.getDeviceId();
3     ClassWithStatic static1 = new ClassWithStatic();
4     static1.setTitle(tainted);
5     ClassWithStatic static2 = new ClassWithStatic();
6     String alsoTainted = static2.getTitle();
7
8     ConnectionManager cm = new ConnectionManager();
9     cm.publish(alsoTainted);
10 }
```

Figure 4.2: static2Test Java Code

#### 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 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.

---

## 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 paragraphs, 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 created two equivalent backwards-specific tests with sinks inside the `equals()` or `hashCode()` method which expect 1 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<sup>1</sup>. 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 similar results but they may have subtle differences.

### 5.2.1 Configuration

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

### 5.2.2 Results

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

---

<sup>1</sup><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		
Inter-Component Communication		
ActivityCommunication1	⊛	⊛
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	⊛	⊛

App Name	Forwards	Backwards
BroadcastReceiverLifecycle2	○	⊛
BroadcastReceiverLifecycle3	⊛	⊛
EventOrdering1	⊛	⊛
FragmentLifecycle1	○	○
FragmentLifecycle2	○	○
ServiceEventSequence1	○	○
ServiceEventSequence2	○	○
ServiceEventSequence3	⊛	⊛
ServiceLifecycle1	⊛	⊛
ServiceLifecycle2	⊛	⊛
SharedPreferencesChanged1	⊛	⊛
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	⊛★	⊛

App Name	Forwards	Backwards
VirtualDispatch3	★	★
VirtualDispatch4		
Miscellaneous Android-Specific		
ApplicationModeling1	★	★
DirectLeak1	★	★
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	★	★



---

### 5.2.3 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 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.