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 16, 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

1 Introduction **5**

2 Background **6**
2.1 Static Data Flow Analysis 6
2.2 IFDS 6
2.2.1 Original Definition 6
2.2.2 Practical Extensions 8
2.3 Intermediate Representations 8
2.4 Soot 10
2.5 FlowDroid 10

3 Theory **11**
3.1 Flow Functions 11
3.1.1 Normal Flow 11
3.1.2 Call Flow 12
3.1.3 Return Flow 13
3.1.4 CallToReturn Flow 14
3.2 Complexity of Data Flow Analysis 14

4 Implementation **16**
4.1 Integration 16
4.2 Problems 17
4.2.1 Flow-Sensitive Alias Analysis 17
4.3 Rules 18
4.3.1 Backwards Sink Propagation Rule 19
4.3.2 Backwards Source Propagation Rule 19
4.3.3 Backwards Array Propagation Rule 19
4.3.4 Backwards Exception Propagation Rule 19
4.3.5 Backwards Wrapper Propagation Rule 19

4.3.6 Backwards Implicit Propagation Rule . . . 19
4.3.7 Backwards Strong Update Rule . . . 19
4.3.8 Backwards Clinit Rule . . . 19
4.3.9 Other Rules . . . 21
4.4 Code Optimizer . . . 21
4.4.1 AddNOPStmts . . . 21

5 Validation **23**
5.1 Unit Tests . . . 23
5.2 DroidBench . . . 24
5.2.1 Configuration . . . 24
5.2.2 Results . . . 24
5.2.3 Discussion . . . 28

6 Evaluation **32**
6.1 Configuration . . . 32
6.2 Performance . . . 32
6.3 Comparison to forwards analysis . . . 32

7 Related Work **33**

8 Conclusion **34**

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 16, 2021

T. Lange

1 Introduction

2 Background

2.1 Static Data Flow Analysis

Explain key terms such as static, fact, taint, source, sink, leak, sensitivity.

2.2 IFDS

2.2.1 Original Definition

Interprocedural finite distributive subset (IFDS) problems are a special class of a dataflow analysis problem. All problems adhering to IFDS can be transformed into a graph-reachability problem and thus the solution is computable in polynomial time. It is context-sensitive and flow-sensitive by default.

IFDS operates on a so-called exploded supergraph. Every node in the exploded supergraph is a tuple $\langle s, d \rangle$ of the statement s in the interprocedural control-flow graph and a dataflow fact d. The domain is typically the set of variables in the program. Edges between two nodes $\langle s, d \rangle$ and $\langle s', d' \rangle$ exist if d propagated over s yields d' and s' is a successor of s. This already ensures flow-sensitivity.

To propagate facts over statements, flow functions need to be defined. There are four types of flows:

- Call Flow: Edges from call statement into a method. Flow function maps the facts visible in the callee into it.
- Return Flow: Edges returning from a method. Flow function maps the facts visible in the caller out of the method.

- Call To Return: Edges over a call statement. Flow function maps the facts not visible in the callee over the call statement.
- Normal Flow: Edges over every other statement. Often, this flow functions only handles assign statements.

All outgoing facts are merged using the merge operator $\sqcap$. The incoming set of facts are all predecessors outgoing facts merged together: $in(s) := \prod_{p \in Preds(s)} out(p)$. The domain also contains a zero fact and all nodes with $d = \mathbf{0}$ are always reachable, thus the zero fact holds at every statement. As an example, in taint analysis the flow functions map zero facts at sources to a tainted variable.

To ensure context-sensitivity, IFDS only visits valid paths. For this, a context-sensitive grammar is constructed which acts like a call stack to make sure there is no mismatch and the path is a valid execution path. The proposed tabulation algorithm to solve the reachable realizable path problem is a dynamic programming algorithm. Whenever a method was fully visited, a summary is saved and later on applied if the same input fact is observed.

For all this to work, the problems which can be formulated in IFDS have to abide to restrictions which are also eponymous:

Distributive: The flow function must be distributive over the merge operator. Formally, $f(x \sqcap y) = f(x) \sqcap f(y)$ must hold at any time. Informally speaking, it does not matter whether facts get merged before or after applying the flow functions.

Finite: Another restriction is that the set of dataflow facts has to be finite. Let's go by a counterexample of what IFDS is not capable of: Answering "Which value is stored in variable x at statement s?". Now the dataflow fact is a tuple of the variable together with the stored value $\langle x, v \rangle$. Assume x is an integer of infinite precision for the domain to be infinite. x is initialized to zero and passed into the method `foo()` multiple times. Recall the subset problem, then look at the summaries in Figure 2.1. Clearly the purpose of creating summaries is lost because we never get to use the summary and also, there is no fixpoint for the ever growing subset to stop. Thus, the domain has to be finite and in practice, also small as the domain is cubic in the time-complexity $O(|E| \cdot |D|^3)$.

Subset: Last, the lattice ordering must be a subset. Following, the merge operator is either set union or set intersection.

```
RealInteger foo(RealInteger x) {
    return x + 42;
}
```

(a) Code

$$\langle x, 0\rangle \rightarrow \langle x, 42\rangle$$
$$\langle x, 42\rangle \rightarrow \langle x, 84\rangle$$
$$\langle x, 84\rangle \rightarrow \langle x, 126\rangle$$
$$\ldots$$

(b) Summaries

Figure 2.1: Finitness example

2.2.2 Practical Extensions

The original definition is inefficient in practice. Among others, Naeem et al proposed practical extensions to the IFDS framework to perform better in practice [6].

Starting at the exploded supergraph, the original algorithm demands a fully built graph. Even in moderate programs the domain can get quite large and as the nodes in the exploded supergraph are the cross product of the domain and interprocedural call-graph nodes, it is infeasible to generate the full graph beforehand. Because there is no way to know before which part of the supergraph is actually needed, it is generated ad-hoc. This also removes the restriction on a small domain, now IFDS is also feasible if the encoutered subset of the domain is small enough [6]. The restrictions on the domain set can be loosened even more. Bodden suggests in-practice the domain can be infinite and only the observed facts must adhere to the ascending-chain condition over the flow functions when using the on-demand supergraph [4].

Also, it also ignores the type structure of the programming language. It can be used to kill facts due to impossible casts. Also, facts with the same variable but different types can be merged to one fact with the superclass as a type [6].

2.3 Intermediate Representations

Most compiler these days use intermediate representations (IRs). IRs are an equivalent representation of the source code but are much simpler and more regular and are typically not architecture dependent. They are often in an interchangeable format and can be saved as text to be able to use them by a variety of tools [7]. This allows compilers to apply machine-independent optimizations to the code with neither worrying about complex expressions in the source code nor reimplementing the optimization for each architecture.

The Java Virtual Machine (JVM) also operates on an IR called Java bytecode. The JVM is mostly stack-based and so is the Java bytecode. In Figure 2.2 is an example of a simple code snippet translated to Java bytecode. Simple expressions such as `c = a + b` are translated into multiple statements and there is no fixed length of an expression in the bytecode. The analysis would also have to reconstruct the expressions ad-hoc. On top, the Java bytecode has over 200 possible instructions[1] which need to be taken into account. Concluding, stack-based IRs are suitable for just-in-time interpretation but inconvenient for data flow analysis [8].

```java
int a = 21;
int b = 21;
int c = a + b;
```

(a) Java code

```
bipush 21 // push 21
istore_1  // store in register 1
bipush 21 // push 21
istore_2 // store in register 2
iload_1 // push a
iload_2 // push b
iadd // pop a & b and push a + b
istore_3 // store in register 3
```

(b) Java bytecode

Figure 2.2: Java bytecode example

A more convenient representation for static analysis are three-address codes. Each statement consists of up to three operands and is either an assigment or a control-flow statement. Such a representation is closer to the original source code while reducing the number of the possible combinations to a managable amount [1].

Jimple is a three-address intermediate representation and can be constructed from the Java and Dalvik bytecode, the IR used for Android apps. It is a high-level representation and its syntax is close to Java. Complex expressions are split up into multiple statements, for example, there can be only one field reference per statement and arguments are always local variables [8]. This greatly reduces the possible cases the data flow analysis needs consider and therefore is the IR of choice for FlowDroid [2]. The conversion to Jimple is provided by the underlying framework Soot.

[1] https://docs.oracle.com/javase/specs/jvms/se8/html/

2.4 Soot

Soot is a 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 $k \geq 0 \land 0 \leq i \leq n$ 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 [5]. 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 [2]. 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, 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.

HeapTestPtsAliasing We focused in this work on flow-sensitive aliasing. Points-To-Aliasing is left for future work.

ImplicitFlowTests implement later if enough time

SetTests `containsTest` needs implicit flows to find the leak. It is supposed to fail.

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 [3]. 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.

Left out ICC, ReflectionICC, IAC (all 3 Buggy), Dynamic Code Loading (Not supported), native code (no android nc handler), Self-Modification (not supported), UnreachableCode (not supported)

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	⋆	⋆

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

App Name	Forwards	Backwards
MultidimensionalArray1	⊛	⊛
Callbacks		
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		

App Name	Forwards	Backwards
FieldSensitivity3	⊛	⊛
FieldSensitivity4		
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	⊛	⊛

App Name	**Forwards**	**Backwards**
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	⊛	⊛
InactiveActivity		
Library2	⊛	⊛
LogNoLeak		
Obfuscation1	⊛	⊛
Parcel1	⊛	⊛
PrivateDataLeak1	⊛	⊛
PrivateDataLeak2	⊛	⊛
PrivateDataLeak3	⊛○	⊛○
PublicAPIField1	⊛	⊛
PublicAPIField2	⊛	⊛
View1	⊛	⊛
Reflection		

App Name	Forwards	Backwards
Reflection1	⊛	⊛
Reflection2	⊛	⊛
Reflection3	⊛	⊛
Reflection4	⊛	⊛
Reflection5	⊛	⊛
Reflection6	⊛	⊛
Reflection7	◯	◯
Reflection8	⊛	⊛
Reflection9	⊛	⊛
Threading		
AsyncTask1	⊛	⊛
Executor1	⊛	⊛
JavaThread1	⊛	⊛
JavaThread2	⊛	⊛
Looper1	⊛	⊛
TimerTask1	⊛	⊛
⊛	103	102
⋆	13	13
◯	12	13
Precision	88.79%	88.7%
Recall	89.57%	88.7%
F1 measure	0.89	0.89

5.2.3 Discussion

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

Emulator Detection

IMEI1 needs implicit flows to find both leaks. The source is only used inside the condition of an if statement. See Figure 5.1, we are unable to taint imei without an implicit taint created in line 8.

could be fixed in the future

```java
String imei = telephonyManager.getDeviceId(); // source
String suffix = "000000000000000"; // T={}
String prefix = "secret"; // T={}
String msg = prefix + suffix; // T={prefix, suffix}

int zeroPos = 0; // zeroPos dies here
while (zeroPos < imei.length()) {
    if (imei.charAt(zeroPos) == '0') // implicit flow needed
        zeroPos++;
    else {
        zeroPos = 0;
        break;
    }
}

String newImei = msg.substring(zeroPos, zeroPos + Math.min(prefix.length(),
    msg.length() - 1)); // T={msg, zeroPos}
Log.d("DROIDBENCH", newImei); // T={newImei}

SmsManager sm = SmsManager.getDefault();
sm.sendTextMessage("+49 123", null, newImei, null, null); // T={newImei}
```

Figure 5.1: IMEI1 excerpt

General Java

StaticInitialization1 differs in forwards and backwards analysis. Backwards it reports one leak due to the explicit modelling of `<clinit>` edges instead of relying on SPARK. Recall subsection 4.3.8, leaks inside static blocks are missed in forward analysis. This test case is quite similiar to Figure 4.3b and therefore the leak is only reported in backwards analysis at the moment.

StaticInitialization2 yields the same result but because of different reasons. The test assigns a tainted value to a static field in the static initializer. Again, recall subsection 4.3.8. Backwards, the clinit rule takes care of visiting the `<clinit>` edge while forwards the followReturnsPastSeeds option of the IFDS solver is responsible.

StaticInitialization3's leak is missed despite the explicit modelling of clinit. The code is provided in Figure 5.2. `MainActivity` is using the singleton pattern and thus has a static field `v` refering to its instance. The source statement is inside the static block of the `Test` class using the singleton to access the instance field `s`. The taint is now introduced at the sink and refers to the field through the `this` instance. When we visit line 13, the `<clinit>` edge is not taken due to the taint being an instance field. Line 12 kills the only taint and stops the analysis as there is no taint to propagate anymore. We never get to see the statement where the static field `v` aliases `this`. We miss the leak because the test violates our assumption of the clinit rule that only static taints can leak inside static blocks. To catch this leak, an inactive taint could be propagated upwards after line 12 to later turn around and let the aliasing visit the clinit edge with the downside that we can not kill any overwrite of an instance field until the end of the callgraph. We decided to deliberately miss those kind of leaks in favor of much less edges to be propagated.

```java
public class MainActivity extends Activity {
    public static MainActivity v;
    public String s;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        v = this;

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        s = ""; // T={}
        Test t = new Test(); // T={this.s}
        Log.i("DroidBench", s); // T={this.s}
    }
}

class Test {
    static {
        TelephonyManager mgr = (TelephonyManager)
            MainActivity.v.getSystemService(Activity.TELEPHONY_SERVICE);
        MainActivity.v.s = mgr.getDeviceId(); // source
    }
}
```

Figure 5.2: StaticInitialization3 code

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] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. eng. Reading, Mass. : Addison-Wesley Pub. Co., 1986. ISBN: 9780201100884. URL: `http://archive.org/details/compilersprincip00ahoa` (visited on 02/15/2021).

[2] 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).

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

[4] Eric Bodden. "Inter-procedural data-flow analysis with IFDS/IDE and Soot". In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis - SOAP '12*. ACM Press, 2012. DOI: `10.1145/2259051.2259052`.

[5] 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`.

[6] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. "Practical Extensions to the IFDS Algorithm". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 124–144. DOI: `10.1007/978-3-642-11970-5_8`.

[7] Douglas Thain. *Introduction to Compilers and Language Design*. en. Google-Books-ID: 5mVyDwAAQBAJ. Lulu.com, July 2019. ISBN: 9780359138043.

[8] Raja Vallee-rai and Laurie Hendren. "Jimple: Simplifying Java Bytecode for Analyses and Transformations". In: (Jan. 2004).