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: 14th March 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, 14th March 2021

T. Lange

1. Introduction

2. Background

In this chapter, we introduce the necessary background. In section 2.1, we explain the term static data flow analysis. We introduce concepts used to solve data flow problems precisely in section 2.2 and section 2.3. We reason the need for a more manageable code representation in section 2.4 and last, we introduce FlowDroid, the tool our work is based on, in section 2.5.

2.1. Static Data Flow Analysis

In the field of compilers, there is a distinction between static and dynamic. Static generally refers to something that is decided at compile-time, while dynamic refers to runtime decisions [1]. The same distinction is also present for analyses. Dynamic analysis observes the program's run behavior, while static analysis works on a representation of the code. Both have different tradeoffs. Achieving good code coverage is hard in dynamic analysis and it is hard only to traverse the actually taken paths in static analysis [2]. Thus dynamic analysis is an underapproximation and static analysis is an overapproximation. In the following, we only consider static analysis.

Data flow analysis is a broad term for analyses which try to identify data flows through a program. Khedker [7] defines data flow analysis as follows:

> *Data flow analysis is a process of deriving information about the run time behavior of a program.*

Data flow analyses are used in many different ways. Compilers use it to apply optimizations, others use it for software verification and it is also used for reverse-engineering [7]. A special kind of data flow analysis is taint analysis which falls under the category of reverse-engineering and its concepts might be familiar from code reviews. In taint analysis, the goal is to determine whether a particular statement's information contents flow through

the program to another statement. Variables that contain such valuable information are tainted variables. This valuable information has to come from somewhere, the so-called sources. Sources can be any expression but are often methods and their returned values are considered tainted. On the other end, sinks leak valuable information. A data flow between a source and a sink is called a leak [2]. If taint analysis is applied to the use case of tracking detection, a source would be a method returning a unique identifier and a sink would be a method that sents out data to the internet. When finding a leak, we know the receiving server can identify the device.

There is also a categorization for data flow analyses. Sensitivities describe whether an analysis is capable of considering an aspect. There are five common sensitivities:

Is this common knowledge? Didn't find a good source for this

- **Flow Sensitivity**: A flow-sensitive analysis can determine if a fact holds at a particular statement. Data flow analyses are always flow-sensitive [7].
- **Context Sensitivity**: An interprocedural analysis can distinguish the context of a called method, e.g., knows the original call site at a return statement.
- **Object Sensitivity**: An analysis can distinguish field accesses on different objects.
- **Field Sensitivity**: An analysis can distinguish different field accesses on the same object.
- **Path Sensitivity**: An analysis takes conditional branches into account, e.g., the condition holds after the branch.

We also need a representation for the information the analysis gathered: the data flow fact. A data flow fact is a logical assertion that is either true or false at a statement. Now, there are two different kinds of facts: may and must. For a must analysis, the fact must hold on all paths to this statement, while a may analysis only guarantees the fact holds on one path. The decision of which kind fits depends on the type of data flow analysis. Taint analyses like FlowDroid are based on the may analysis [2].

2.2. IFDS

2.2.1. Original Definition

Interprocedural finite distributive subset (IFDS) problems are a special class of a data flow analysis problem. Generally, the solution to a data flow problem is the meet-over-

all-paths (MOP) solution, which is undecideable [13]. But all problems adhering to IFDS can be transformed into a graph-reachability problem and consequently, the solution is computable in polynomial time. It is context-sensitive and flow-sensitive by default [12].

IFDS operates on a so-called exploded supergraph. Every node in the exploded supergraph is a tuple $\langle s, d \rangle$ of a 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. Propagating facts along the control-flow graph already ensures flow-sensitivity. For context-sensitivity, IFDS only visits valid paths. Reps et al. proposed a context-sensitive grammar that acts as a call stack to ensure no mismatch and the path is a valid execution path [12].

To propagate facts over statements, we need to define rules on how the data flow changes when observing a statement. These rules are called flow functions. There are four types of flow functions:

- **Call Flow**: Edges from call statement into a method. The flow function maps the facts visible in the callee into it [12].
- **Return Flow**: Edges returning from a method. The flow function maps the facts visible in the caller out of the callee [12].
- **Call To Return Flow**: Edges over a call statement. The flow function maps the facts not visible in the callee over the call statement [12].
- **Normal Flow**: Edges over every other statement. Often, this flow function only handles assign statements [12].

The incoming set of facts is all predecessors' outgoing facts merged using a merge operator $\sqcap$:

$$in(s) := \bigsqcap_{p \in Preds(s)} out(p)$$

Now, we also want to introduce new facts. For that reason, the domain contains a zero fact and all nodes with $d = \mathbf{0}$ are always reachable; thus, the zero fact is a tautology. Whenever we want to introduce a fact, we can model this in the flow function by deriving such facts from the zero fact [12]. For example, in taint analysis, the flow functions map zero facts at sources to a tainted variable.

IFDS also utilizes summaries. After returning from a method, the algorithm solved a subproblem for which it remembers the results to be applied later. So, the proposed tabulation algorithm for solving the realizable path problem is a dynamic algorithm [12].

Eventually, there is no fact to propagate anymore and the analysis will terminate. Facts die because a flow function killed them or the algorithm already observed the same fact at a statement and reached a fixpoint [12].

We already started this section, hinting not all problems can be formulated in IFDS. The restrictions the problems have to abide by are eponymous in IFDS and explained in the following paragraphs.

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. By defining the flow function signature as $f : Fact \to Facts$ with a single fact as input but a set of facts as output, this property is trivially satisfied. This property is required for correctness because with distributiveness, the maximum fixed point (MFP) equals MOP [7, 12].

Finite Another restriction is that the set of dataflow facts has to be finite. Let us 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 $\langle x, v \rangle$. Consider Figure 2.1. 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. Now consider the summaries in Figure 2.1b. We never get to use one summary because the value always increases. Because of the same reason, we also never reach a fixpoint. Thus, the domain has to be finite and, in practice, also small as the domain is cubic in the time-complexity $O(|E| \cdot |D|^3)$ [12].

Subset Data flow frameworks need to deal with merging the outcoming sets to a single incoming set. Essentially, to formalize the approximation and satisfy ordering constraints, data flow frameworks rely on lattices [7]. IFDS also defines an underlying lattice on the powerset of the domain. The lattice ordering must be set inclusion. Therefore, the merge operator is set union or set intersect. Now recall may and must from the last subsection. Here we can see the connection between the merge operator and may or must.

```
1  void main() {
2      RealInteger x = 0;
3      while (condition)
4          x = foo(x);
5  }
6
7  RealInteger foo(RealInteger x) {
8      return x + 42;
9  }
```

(a) Code

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

(b) Summaries

Figure 2.1.: Finiteness example

The paper by Reps et al. later decides on set union due to the duality of must and may not [12]. This decision is also efficient in practice as discussed in the following subsection.

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 [10].

The original algorithm demands a fully built exploded supergraph. Even in moderate programs, the domain can get quite large. 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 the analysis demands, they propose to generate it ad-hoc. That also removes the restriction on a small domain. Now IFDS is also feasible if the domain's encountered subset is small enough [10]. The restrictions on the domain set can be loosened even more. Bodden suggests in-practice, the domain can be infinite. Only the observed facts must adhere to the ascending-chain condition over the flow functions when using the on-demand supergraph [4].

Also, the original IFDS definition 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 with the superclass as its new type [10].

The original definition starts the IFDS algorithm at the entry point of the interprocedural call-graph. As described in section 2.2, a flow function can derive an initial fact from the zero fact whenever needed. If the methods where initial facts will be introduced are known a priori, the supergraph can be traversed without applying flow functions until such a method is found on the path. This optimization introduces unbalanced problems where a method return but no corresponding call site is found, which can be solved by a small extension to the tabulation algorithm. The extension was first described by Lerch [9] and is also present in FlowDroid [2].

If the merge operator is set union, there is no need to wait for other predecessors to finish as a set is always a subset of a union with itself and another set ($A \subseteq A \cup B$ holds) . Hence, the IFDS solver can skip the in-set construction and immediately propagate the outcoming facts as singleton sets, which is beneficial in a parallelized solver [14].

Is there a name for this property?

2.3. Access Paths

We have already seen IFDS fulfills context- and flow-sensitivity by default. Now, a precise analysis for Java also needs object- and field-sensitivity. Thus, we also need to model the heap.

Access paths are one possible heap model. They consist of a list of field dereferences linked to a tainted variable of a reference type [7]. Note, this increases the domain size because now not only "object o is tainted" is a data flow fact, but also all of its fields can be tainted. Especially when encountering recursive data structures such as linked lists, this gets problematical. Consider Figure 2.2, the loop would let the observed domain grow indefinitely and never reach a fixpoint. As a solution, access paths are limited in length which is also called k-limiting, whereas the constant k is the maximum access path length. If an access path passes this length, it is cut off and the entire last reference is considered tainted [6]. Consider again Figure 2.2 with $k = 2$. This time the analysis would reach the fixpoint $lst.next.prev.*$ after two iterations. The cut-off comes with a loss of precision but is inevitable.

Although with k-limiting, the algorithm terminates again, it introduces a new problem. After a loop like in Figure 2.2, the access path is polluted with a dereference chain to its base object even though the $next.prev$ dereference could be omitted without precision loss. Thus, Deutsch proposed symbolic access paths, which try to eliminate loops in access

```
1 while (condition) {
2     lst = lst.next.prev;
3 }
```

(a) Code

$$
\begin{aligned}
lst \quad & \rightarrow lst.next.prev \\
lst.next.prev \quad & \rightarrow lst.next.prev.next.prev \\
& \ldots
\end{aligned}
$$

(b) Access Path Iterations

Figure 2.2.: Infinite Access Path

paths [5]. In practice, Deutsch's approach needs some adaptions as he only considered fields but not base objects and he defines loops simply by type [2].

2.4. Intermediate Representations

Most compilers 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 used by various tools [16]. Such an IR 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 2.3 is an example of a simple code snippet translated to Java bytecode. Simple expressions such as `c = a + b` translate 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. Furthermore, Java bytecode has over 200 possible instructions[1], which need to be considered and only knows primitive types and references. Concluding, stack-based IRs are suitable for just-in-time interpretation but inconvenient for data flow analysis [17].

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

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

```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 2.3.: Java bytecode example

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 statements are split up into multiple statements. For example, there can be only one field reference per statement and arguments are always local variables. Jimple also reconstructs reference types [17]. This groundwork greatly reduces the possible cases the data flow analysis needs to consider and eases the analysis.

2.5. FlowDroid

FlowDroid FlowDroid is a precise context-, flow-, object- and field-sensitive static taint analysis tool [3].Since its initial release in 2014, it is actively maintained and gained traction in research and academia[2]. It is based on Soot, a Java optimization framework, which later has been extended for static analysis [8]. Soot provides the call graph and the conversion from Java and Dalvik bytecode to Jimple, the intermediate representation of choice for FlowDroid [2].

Androids activity-lifecycle concept does not have a single entry point; instead, there are multiple callbacks as a possible entry point. Also, an Android app can contain multiple components and register callbacks in various of Android's standard libraries. FlowDroid models the entire Android lifecycle to be precise and generates a dummy main method to

[2] https://github.com/secure-software-engineering/FlowDroid

[3] Taken from https://developer.android.com/guide/components/activities/activity-lifecycle.

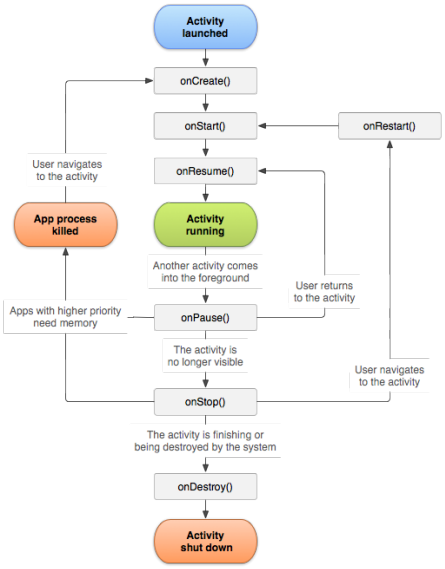


Figure 2.4.: Activity Lifecycle[3]

provide a single entry point for the call graph generation. It assumes multiple components can run in an arbitrary sequential order with repetitions [3].

To provide precise results even with aliases, FlowDroid contains besides the taint analysis another IFDS problem to resolve all encountered aliases on-demand. For this, FlowDroid needs a high object-sensitivity. Hence it makes use of symbolic access paths [3] .

is symbolic access paths right?

The implementation of FlowDroid is modular, easily extensible and offers many additional features. Two of them are noteworthy for this work: native call handler and taint wrappers. As both Java and Android allow calling native methods, FlowDroid also needs to model those cases. It currently does not support the analysis of those methods but contains rules for essential methods. The second feature is taint wrappers. They allow defining rules for methods, e.g., from a commonly used feature such as StringBuilder, which allows the taint analysis to skip the method and apply a summary [3]. StubDroid, an extension to FlowDroid by Arzt et al., allows precomputing summaries using FlowDroid and serializes them in an XML format for tool-independent use. These summaries are handy for real-world applications where often third-party libraries are used [2].

3. Theory

In the first part of this chapter, we define the flow functions and in the second part we discuss the time complexity of taint analysis specificly hightlighting differences between forward and backward analysis.

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 2.4. 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 encounter a whole object tainted. In this case, just the base needs to match but the left side is also kept alive because other fields also might be tainted if the object has more than one field.

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

Lastly, the right side could also be tainted. This rule is equivalent to the default behaivor but is important later when we consider aliasing in subsection 4.2.2.

> **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 $T = \{this_c.h^k\}$.

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 $k \geq 0 \wedge (\forall a \in \mathit{Arguments} : a \neq x) \wedge x \neq o \wedge x \notin \mathit{Static}$ 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

IFDS has a time-complexity of $O(E \cdot D^3)$. The edges in the control-flow graph are set by the to-be-analyzed app. The domain depends on the tainted variables observed by the IFDS analysis. Arzt et al evaluation of FlowDroid shows no correlation between a-priori known parameters and the runtime of the analysis [2]. So, we are left with the number of taint propagations as the only parameter correlated to the runtime but they are only known afterwards.

The number of taint propagations depends on two factors: the lifetime of taints and the number of taints. Both factors highly depend on the search direction which we will explain in the following paragraphs.

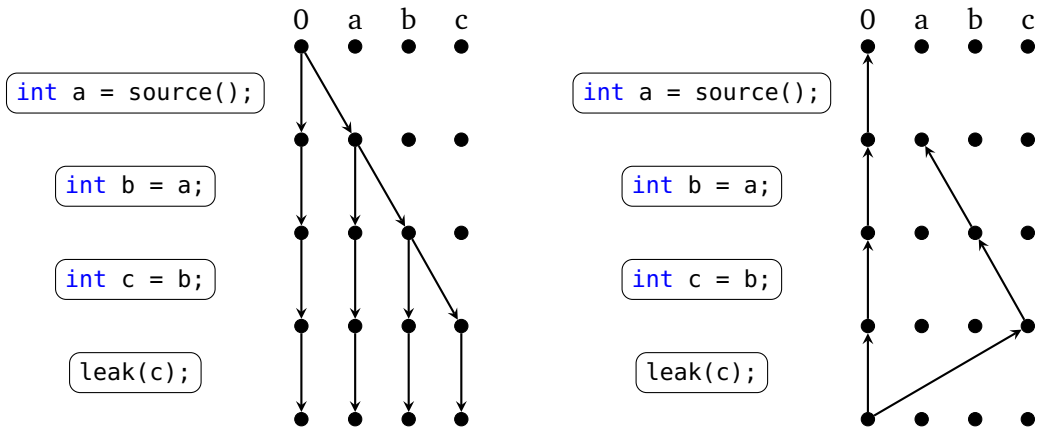
```
1  String returnParam(int i, String s1, String s2, String s3) {
2      if (i == 1)
3          return s1;
4      else if (i == 2)
5          return s2;
6      else if (i == 3)
7          return s3;
8      else
9          return "default";
10 }
```

Figure 3.1.: Branching Factor Example

First, we take a look at the is the branching factor. The branching factor describes the number outgoing edges from a node. A smaller branching factor is favorable. Think of binary operator expression such as `int c = a + b;`, backwards we can not argue which operand is responsible for the tainted output and thus proceed with both operands tainted. The same restriction is present in rule 4 of Call Flow which describes how the returned value is mapped back into the callee. This time the branching factor can be even larger. As an example, in Figure 3.1 is a method which conditionally returns one of its parameters and is part of the leak path. Lets assume the returned value of a call to `returnParam()` is tainted. Backwards, every returned operand is tainted and later on mapped according to Return Flow rule 2 back into the caller. Thus, the IFDS algorithm ends up with a summary $retVal \rightarrow \{s1, s2, s3\}$. Forwards, a tainted parameter is mapped into the callee and later on returned to the caller resulting in a summary $sX \rightarrow \{retVal, sX\}$ with $X \in \{1, 2, 3\}$. Such a case favors forwards analysis.

In contrast, a strict right-to-left flow favors backwards analysis as backwards taints are killed more often due to a stronger overwrite rule meaning a shorter lifetime per taint. In Figure 3.2 is such a right-to-left flow displayed. Forwards, the right hand side is always kept alive because it still holds the tainted value below the statement and could be leaked. Starting at the sinks this fact can be confidently ignored because the fact is known to be not on a leak path.

Static field taints are a special case and an issue in both directions. Their scope is global, so a static field can be accessed anywhere in the code. Thus such a taint needs to be propagated into every method and can only be killed on overwrite.



(a) Forwards (b) Backwards

Figure 3.2.: Right-to-Left Order

In special cases of application, the direction of the dtaflow analysis might be more important. We bring in classical web-oriented applications as an example. Typical vulnerabilities such as cross-site-scripting and SQL injection occur because untrusted user input was not validated. Normally the content is escaped right before output or sanitized before execution of the command, respectively. So the sanitization method is expected to be close to the sink. Being able to kill taints ealier is advantageous and therefore backwards analysis should be the better choice.

We discussed cases where one direction is better than the other but it still highly depends on the analyzed app.

- 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

In this chapter, we describe the details of our backwards-directed implementation and how we integrated it into FlowDroid.

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. Flow Function Implementation

4.2.1. Infoflow Analysis

4.2.2. 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.

```
1 void aliasRule1() {
2     A a = b;
3     b.str = source();
4     sink(a.str);
5 }
```

(a) Example for alias analysis initiated by rule 1

```
1 void aliasRule3() {
2     A a = b;
3     a.str = source();
4     sink(b.str);
5 }
```

(b) Example for alias analysis initiated by rule 3

Figure 4.1.: Normal flow Aliasing examples

Handover 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. In 4.1a, at line 2 `b` is assigned to `a`. Above this statement, the contents of `a` do not matter anymore, thus `a` is killed. On the other side, `b` is now tainted but as `a` and `b` aliases, we missed all updates to `b` below the statement. Concluding, at line 2 the `a.str` taint is killed, a new taint `b.str` is created and alias analysis for `b.str` is triggered. In 4.1b, we also observe that all writes to the alias were neglected but this time the incoming taint is propagated over the statement and for `a.str` the alias analysis is triggered.

A taint is handed back to the infoflow search if the alias analysis encounters an update to the taint, e.g. the taint is on the left side of the assignment. Again consider Figure 4.1a. At line 3, `b.str` is tainted and on the left side of the assignment. At this point, `b.str` is handed over to the infoflow analysis with the statement on line 3. The backwards infoflow search then follows the missed path to detect possible leaks, in this case it right away reports a leak.

Turn Unit 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 in the infoflow search. At start, it is the sink where the taint was introduced. Later on, it is set whenever we visit an assigment with a primitive or string on the left side[1]. Consider the code in Figure 4.2. At line 5, the taint is introduced, 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

[1]The String object is a special case in Java. It is saved on the heap but immutable and therefore can not alias.

4.3. Rules

Flow functions can get quite large, complicated to understand and hard to maintain [9]. To counteract this, FlowDroid outsources certain features into rules. These rules also implement the four flow functions and are applied in the corresponding flow function.

4.3.1. Source & Sink Propagation Rule

In backwards analysis, sources act like sinks and vice versa. Thus, the Source Propagation Rule records taints flowing into sources and the Sink Propagation Rule unconditionally introduces taints at sinks.

Notably, the `DefaultSourceSinkManager` assumes the return value to be tainted and only if the return value is ignored or the method has no return value the base object is assumed to be tainted unless specified otherwise while at sinks base object and parameters are leaked [2]. Thus, starting at sinks results in more taints per start statement than in forwards analysis. As written in section 3.2, Arzt's evaluation has shown that the initial source count does not correlate with the runtime which implies that this should be is insignificant.

4.3.2. Backwards Array Propagation Rule

The Array Propagation Rule handles `ArrayNewExpr`, `LengthExpr` and `ArrayRef` on the right hand side. Further, we describe the three cases of this rule.

- **Array Rule 1**: If the length of the left side is tainted and the right side is an `ArrayNewExpr`, the outcoming taint is the size local of the `ArrayNewExpr`.
- **Array Rule 2**: If the left side is tainted and the right side is a `LengthExpr`, the outcoming taint is the operand of the `LengthExpr` with only its length tainted.
- **Array Rule 3:** If the left side is tainted and the right side is an `ArrayRef`, the outcoming taint is the array base with only its content tainted.

By default, the whole array is tainted and indices are not tracked. Following, all three rules kill the incoming taint unless the left side is an `ArrayRef`.

4.3.3. Backwards Exception Propagation Rule

The Backwards Exception Propagation Rule handling is

4.3.4. Backwards Wrapper Propagation Rule

FlowDroid already provided a `IReversibleTaintWrapper` interface. Implementing taint wrappers support `getInverseTaints()` which takes the outcoming taint as an input and computes the incoming taints.

This rule is similiar to its forward equivalent but enforces a reversible taint wrapper. Consequently, tainted return values are also passed into the taint wrapper.

Inverse taint wrappers do have one limitation. Often tainted parameters result in a tainted base object. `EasyTaintWrapper` uses this pattern to provide a fast and simple taint wrapper [2]. But backwards, only a tainted base object is observed. Similiar to binary operators in assignments, we can not do anything but to taint every parameter.

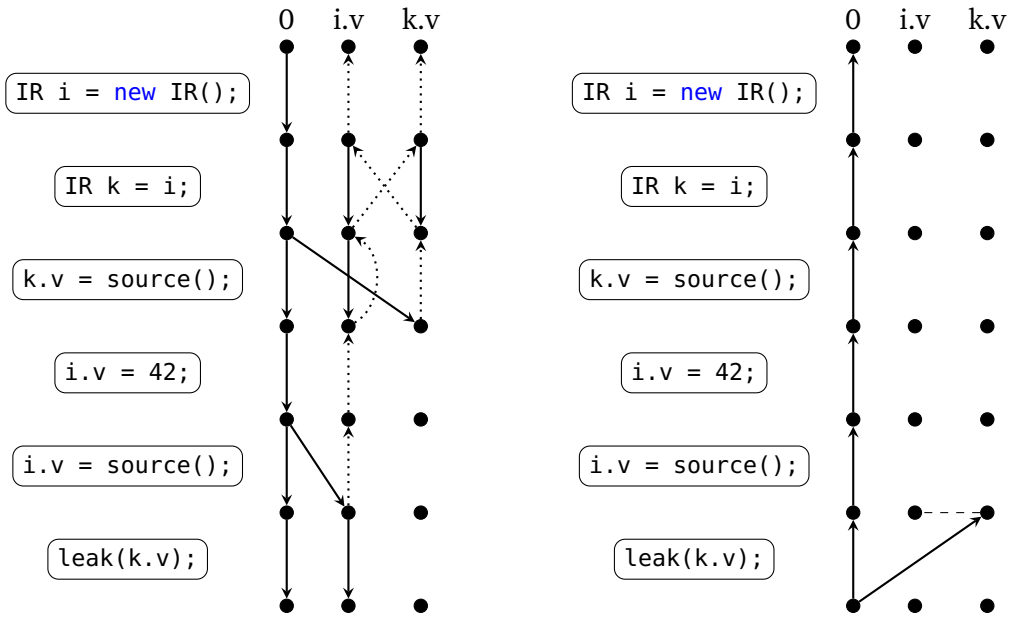
4.3.5. Backwards Strong Update Rule

Strong updates are assignments where the content of a variable gets overwritten. In our normal flow rules, this is modelled in rule 1. When a statement is observed with its left side tainted, we know it got its tainted content at this statement. Thus we kill the taint because the content above this statement is of no interest and taint the right hand side. So we performed a strong update on the left side.

But with aliasing this gets quite more complicated. Now, we can observe a taint not matching the left side and propagate it over the statement according to the default rule of normal flow but the taint is an alias of the left side and should have been killed. Linking aliasing taints to support such strong updates would lose the distributiveness property of the flow functions.

In this case, FlowDroid falls back to Soot's must-aliasing analysis. However, the built-in must-aliasing is only intraprocedural. Thus, the strong update rule can not detect strong updates split over methods.

- **Strong Update Rule**: If the incoming taint must-aliases the left side then apply the normal flow rules just as if the left side was tainted.



(a) Forwards (b) Backwards

Figure 4.3.: Strong Update Example

Searching backwards also removes one limitation of strong updates. Consider the example in Figure 4.3. `IR` is a wrapper around an integer to provide integers by reference. The main idea behind this example is a strong update on an alias of a tainted variable and later retaint the alias. In 4.3a, taints are created at both source statements. The critical point is `i.v = 42;` where the forwards strong update rule kills `k.v` because it must aliases `i.v` and the taint `i.v`, later found through aliasing, is killed according to normal flow. Both kills are appropriate because it is unknown whether there will be another strong update. This is not the case in backwards analysis. Here the last write before the sink to the leaked variable or one of its aliases is found first. So after one propagation the taint already reaches a source statement.

4.3.6. 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 4.4. The invokation is implicit at the initialization phase of the class and is executed at most once for each class [2]. 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` [3].

```
1 class ClinitClass1 {
2     public static string str =
          source();
3 }
```

(a) static variable initialization

```
1 class ClinitClass2 {
2     static {
3         ClinitClass2.sink();
4     }
5 }
```

(b) static block

Figure 4.4.: 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`. Consequently, the calls to `<clinit>` from `AssignStmts` with `NewExpr` or `StaticFieldRef` on the right side are missed.

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:

- **Clinit Rule 1**: 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.
- **Clinit Rule 2**: 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>`.

[2] https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.9
[3] https://github.com/soot-oss/soot/blob/59931576784b910a7d38f81910b7313aa2feafea/src/main/java/soot/jimple/toolkits/callgraph/OnFlyCallGraphBuilder.java#L969

- **Clinit Rule 3**: 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.

In fowards analysis, the issue is not as severe. As taints are introducted at sources, if the source statement is a static initialization as shown in Figure 4.4a, 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 4.4b.

4.3.7. 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.

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 4.5. 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 4.3.1, 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 new edge cases inside the flow functions which are already complex enough. Due to the entry points being known beforehand, the overhead is negligible.

```
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  }
```

(a) Java

```
1   public static void static2Test() {
2       tainted = staticinvoke
            <soot.jimple.infoflow.test.android.TelephonyManager:
            java.lang.String getDeviceId()>(); // Line 2 in (a)
3
4       // [...]
5
6       virtualinvoke cm.<soot.jimple.infoflow.test.android.ConnectionManager:
            void publish(java.lang.String)>(alsoTainted); // Line 9 in (a)
7
8       return;
9   }
```

(b) Jimple

Figure 4.5.: static2Test Code

5. Validation

5.1. Unit Tests

FlowDroid already contains 519 unit tests for the core infoflow component. We also validated the backwards analysis with these tests with positive results. Because we focused on the context-sensitive analysis, not all analysis-related tests were applicable. In the following, we briefly explain why tests were left out or did not return the same results.

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 4.3.6 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 Not implemented.

implement later if enough time

SetTests `containsTest` needs implicit flows to find the leak.

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]. 190 test cases are included in the latest development version 3[1]. We use the newest commit at the time of writing[2]. We aim to achieve similiar results as FlowDroid's existing forwards implementation.

5.2.1. Configuration

The validation was run with the default configuration of the Android module of FlowDroid using EasyTaintWrapper as the taint wrapper. The configuration is summarized in Table 5.1.

Option	Value
Array Size Tainting	disabled
Inspect Sources & Sinks	disabled
Static Field Tracking	enabled
Ignore Flows in System Packages	enabled
Exclude Soot Library Classes	enabled
Timeout	-
Taint Wrapper	`EasyTaintWrapper`

Table 5.1.: Real World Apps Configuration

We only used a subset of DroidBench tests to validate our results. Dynamic Code Loading, Self Modification, Unreachable Code and Native Code are all not supported by FlowDroid. The first three are all callgraph related and the latter is not supported because FlowDroid has no Android native call handler for now. Also Inter Component Communication, Reflection Inter Component Communication and Inter App Communication were left out. The Inter Component Communication module was - at the time of this work - not maintained anymore. As all left out tests do not depend on the flow functions, if FlowDroid gets support for those in the future they should also work in backwards analysis.

[1] `https://github.com/secure-software-engineering/DroidBench/`
[2] 6. March 2021, Commit `ddbd50c`

5.2.2. Results

The full overview of the results are in Table 5.2. ⊛ denotes true positive, ⋆ false positive and ◯ false negative. If a row is empty, there are no expected leaks and also none were found.

Our backwards-directed implementation yields nearly the same result as the existing forwards implementation with just a one missed leak more than the baseline. We achieve a F1 measure of 0.89 which is the same as the baseline.

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	⊛	⊛
Button5	⊛	⊛
LocationLeak1	⊛⊛	⊛⊛
LocationLeak2	⊛⊛	⊛⊛
LocationLeak3	⊛	⊛
MethodOverride1	⊛	⊛

App Name	Forwards	Backwards
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		
Lifecycle		
ActivityEventSequence1	⊛	⊛
ActivityEventSequence2	◯	◯
ActivityEventSequence3	◯	◯
ActivityLifecycle1	⊛	⊛
ActivityLifecycle2	⊛	⊛
ActivityLifecycle3	⊛	⊛

App Name	Forwards	Backwards
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	⊛	⊛
Exceptions7		
FactoryMethods1	⊛⊛	⊛⊛
Loop1	⊛	⊛
Loop2	⊛	⊛
Serialization1	◯	◯
SourceCodeSpecific1	⊛	⊛
StartProcessWithSecret1	⊛	⊛
StaticInitialization1	◯	⊛
StaticInitialization2	⊛	⊛
StaticInitialization3	◯	◯

App Name	Forwards	Backwards
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	⊛	⊛
Reflection3	⊛	⊛
Reflection4	⊛	⊛
Reflection5	⊛	⊛
Reflection6	⊛	⊛
Reflection7	◯	◯
Reflection8	⊛	⊛
Reflection9	⊛	⊛
Threading		
AsyncTask1	⊛	⊛
Executor1	⊛	⊛

App Name	Forwards	Backwards
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

Table 5.2.: DroidBench Validation Results

5.2.3. Results Explanation

In greater detail, we miss both leaks in `EmulatorDetectionTests#IMEI1` whereas in `StaticTests#StaticInitialization1` we do not miss the leak. We provide explanation on why below.

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 the variable `imei` without an implicit taint created in line 8.

could be fixed in the future

General Java

As all StaticInitialization tests depends on the modelling of the `<clinit>` behavior, we decided to explain all three even though only StaticInitialization1 is different.

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 4.3.6, leaks inside static blocks are missed in forward analysis. This test

```
1  String imei = telephonyManager.getDeviceId(); // source
2  String suffix = "000000000000000"; // T={}
3  String prefix = "secret"; // T={}
4  String msg = prefix + suffix; // T={prefix, suffix}
5
6  int zeroPos = 0; // zeroPos dies here
7  while (zeroPos < imei.length()) {
8      if (imei.charAt(zeroPos) == '0') // implicit flow needed
9          zeroPos++;
10     else {
11         zeroPos = 0;
12         break;
13     }
14 }
15
16 String newImei = msg.substring(zeroPos, zeroPos + Math.min(prefix.length(),
       msg.length() - 1)); // T={msg, zeroPos}
17 Log.d("DROIDBENCH", newImei); // T={newImei}
18
19 SmsManager sm = SmsManager.getDefault();
20 sm.sendTextMessage("+49 123", null, newImei, null, null); // T={newImei}
```

Figure 5.1.: IMEI1 excerpt

case is quite similiar to Figure 4.4b 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 4.3.6. 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. This is one limitiation of the alias analysis where only encountered aliases are tracked. Tracking all alias is too inefficient for the analysis to be applicable [4].

```
1  public class MainActivity extends Activity {
2      public static MainActivity v;
3      public String s;
4
5      @Override
6      protected void onCreate(Bundle savedInstanceState) {
7          v = this;
8
9          super.onCreate(savedInstanceState);
10         setContentView(R.layout.activity_main);
11
12         s = ""; // T={}
13         Test t = new Test(); // T={this.s}
14         Log.i("DroidBench", s); // T={this.s}
15     }
16 }
17
18 class Test {
19     static {
20         TelephonyManager mgr = (TelephonyManager)
               MainActivity.v.getSystemService(Activity.TELEPHONY_SERVICE);
21         MainActivity.v.s = mgr.getDeviceId(); // source
22     }
23 }
```

Figure 5.2.: StaticInitialization3 code

6. Performance Evaluation

In the last chapter, we have shown that our implementation has the necessary soundness to be viable and yields the expected results. We now evaluate our implementation against the existing implementation in FlowDroid.

6.1. DroidBench

We already introduced DroidBench in section 5.2 to validate the soundness of our backward-directed implementation. In this section, we focus on the performance in comparison to the existing forward-directed implementation in FlowDroid.

DroidBench has the advantage that all apps are crafted explicitly for benchmarking taint analysis. So, most tests only contain a single-figure number of sources and sinks. Also, the number of sources and sinks are often equal or differ by one to test whether the tool can differentiate something. These simplify the comparison between both analysis directions as neither one has an initial disadvantage.

Most test cases are small enough to be analyzed in sub-two seconds on an average four-core desktop CPU from 2012. Our test environment is not isolated, so background tasks and the process scheduler can affect the runtime. The short runtime, together with the variance of the unisolated testing environment, render the runtime unusable as a comparison point. In contrast, edge propagations are deterministic[1] and correlate with the runtime. Thus, we only use the number of propagations to compare both implementations.

The configuration is the same as described in subsection 5.2.1.

[1] This is only true if there are enough resources. FlowDroid tries to gracefully terminate when running low on memory. Also, timeouts result in a non-reproducible number of edge propagations.

6.1.1. Results

We compare all test cases where both implementations yield the same result. When rows only contain hyphens, either the result of the test case differed between the two analyses or the IFDS analysis did not start, e.g., because no sink is in the reachable code. #I denotes the number of edge propagations inside the infoflow analysis and #A the number of edge propagations inside the alias analysis. We calculated the absolute difference as $Result_B - Result_F$. The relative difference calculates as follows: $\frac{TotalDifference}{|\#I_F + \#A_F|}$. Hence, negative values signify the backward analysis performed better. The full results are in Table 6.1.

In general, both implementations have similar average edge propagation counts. There are not many test cases where both perform identically; instead, dependent on the specific test case, the relative difference is between -1 and 1. So, the expected behavior from section 3.2 occurred: it highly depends on the analyzed app. However, we did not expect cases that let the backward edge propagations explode up to a factor of 10000%, as seen in `LifecycleTest#BroadcastReceiverLifecycle3` and others. In contrast, the existing forward implementation only at most a relative difference of 100%.

	Forwards		**Backwards**		**Difference**			
App Name	**#I**	**#A**	**#I**	**#A**	**#I**	**#A**	**Total**	**Relative**
AliasingTest								
FlowSensitivity1	175	72	39	4	−136	−68	−204	−0.83
Merge1	94	44	61	9	−33	−35	−68	−0.49
SimpleAliasing1	35	13	20	3	−15	−10	−25	−0.52
StrongUpdate1	30	13	11	3	−19	−10	−29	−0.67
AndroidSpecificTest								
ApplicationModeling1	235	103	851	1208	616	1105	1721	5.09
DirectLeak1	3	0	4	0	1	0	1	0.33
InactiveActivity	–	–	–	–	–	–	–	–
Library2	5	0	6	0	1	0	1	0.2
LogNoLeak	–	–	–	–	–	–	–	–
Obfuscation1	4	0	4	0	0	0	0	0.0
Parcel1	144	15	66	68	−78	53	−25	−0.16
PrivateDataLeak1	410	110	599	835	189	725	914	1.76
PrivateDataLeak2	15	0	5	6	−10	6	−4	−0.27
PrivateDataLeak3	17	2	212	143	195	141	336	17.68
runPublicAPIField1	89	1	62	31	−27	30	3	0.03
runPublicAPIField2	5	0	11	1	6	1	7	1.4
runView1	71	50	69	0	−2	−50	−52	−0.43
ArrayAndListTest								

App Name	Forwards		Backwards		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
ArrayAccess1	77	34	51	100	−26	66	40	0.36
ArrayAccess2	16	4	12	0	−4	−4	−8	−0.4
ArrayAccess3	77	34	51	100	−26	66	40	0.36
ArrayAccess4	164	84	42	21	−122	−63	−185	−0.75
ArrayAccess5	75	5	67	63	−8	58	50	0.62
ArrayCopy1	18	2	9	2	−9	0	−9	−0.45
ArrayToString1	10	1	6	1	−4	0	−4	−0.36
HashMapAccess1	22	5	15	1	−7	−4	−11	−0.41
ListAccess1	85	9	60	97	−25	88	63	0.67
MultidimensionalArray1	29	3	16	23	−13	20	7	0.22
CallbackTest								
AnonymousClass1	152	0	208	1	56	1	57	0.38
Button1	58	39	43	0	−15	−39	−54	−0.56
Button2	454	66	155	257	−299	191	−108	−0.21
Button3	355	89	109	408	−246	319	73	0.16
Button4	58	39	43	0	−15	−39	−54	−0.56
Button5	80	40	6	6	−74	−34	−108	−0.9
LocationLeak1	617	222	260	300	−357	78	−279	−0.33
LocationLeak2	212	121	152	2	−60	−119	−179	−0.54
LocationLeak3	259	73	104	117	−155	44	−111	−0.33
MethodOverride1	3	0	2	0	−1	0	−1	−0.33
MultiHandlers1	17	0	145	151	128	151	279	16.41
Ordering1	456	151	44	2	−412	−149	−561	−0.92
RegisterGlobal1	207	103	49	0	−158	−103	−261	−0.84
RegisterGlobal2	52	37	43	0	−9	−37	−46	−0.52
Unregister1	11	0	9	1	−2	1	−1	−0.09
EmulatorDetectionTest								
Battery1	7	0	43	15	36	15	51	7.29
Bluetooth1	4	0	4	0	0	0	0	0.0
Build1	4	0	4	0	0	0	0	0.0
Contacts1	52	0	210	19	158	19	177	3.4
ContentProvider1	13	0	8	0	−5	0	−5	−0.38
DeviceId1	15	0	6	0	−9	0	−9	−0.6
File1	4	0	4	0	0	0	0	0.0
IMEI1	129	0	140	34	11	34	45	0.35
IP1	4	0	29	1	25	1	26	6.5
PI1	6	0	4	0	−2	0	−2	−0.33
PlayStore1	158	0	8	0	−150	0	−150	−0.95
PlayStore2	4	0	4	0	0	0	0	0.0
Sensors1	5	0	4	0	−1	0	−1	−0.2
SubscriberId1	29	0	4	0	−25	0	−25	−0.86
VoiceMail1	4	0	4	0	0	0	0	0.0
FieldAndObjectSensitivityTest								
FieldSensitivity1	98	50	25	3	−73	−47	−120	−0.81

App Name	Forwards		Backwards		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
FieldSensitivity2	35	15	19	0	−16	−15	−31	−0.62
FieldSensitivity3	38	15	16	0	−22	−15	−37	−0.7
FieldSensitivity4	14	6	8	0	−6	−6	−12	−0.6
InheritedObjects1	4	0	6	0	2	0	2	0.5
ObjectSensitivity1	19	7	14	1	−5	−6	−11	−0.42
ObjectSensitivity2	15	8	10	0	−5	−8	−13	−0.57
GeneralJavaTest								
Clone1	23	2	12	4	−11	2	−9	−0.36
Exceptions1	16	0	13	0	−3	0	−3	−0.19
Exceptions2	22	0	13	0	−9	0	−9	−0.41
Exceptions3	18	0	11	0	−7	0	−7	−0.39
Exceptions4	20	1	22	1	2	0	2	0.1
Exceptions5	13	1	16	1	3	0	3	0.21
Exceptions6	77	12	23	0	−54	−12	−66	−0.74
Exceptions7	71	12	6	0	−65	−12	−77	−0.93
FactoryMethods1	40	0	14	2	−26	2	−24	−0.6
Loop1	93	2	46	7	−47	5	−42	−0.44
Loop2	123	2	74	14	−49	12	−37	−0.3
Serialization1	50	4	22	29	−28	25	−3	−0.06
SourceCodeSpecific1	16	0	45	7	29	7	36	2.25
StartProcessWithSecret1	29	8	17	3	−12	−5	−17	−0.46
StaticInitialization1	–	–	9	0	–	–	–	–
StaticInitialization2	57	29	86	0	29	−29	0	0.0
StaticInitialization3	35	9	5	0	−30	−9	−39	−0.89
StringFormatter1	16	1	10	1	−6	0	−6	−0.35
StringPatternMatching1	23	1	8	6	−15	5	−10	−0.42
StringToCharArray1	91	4	42	6	−49	2	−47	−0.49
StringToOutputStream1	26	3	30	3	4	0	4	0.14
UnreachableCode	–	–	–	–	–	–	–	–
VirtualDispatch1	128	31	88	28	−40	−3	−43	−0.27
VirtualDispatch2	7	0	12	0	5	0	5	0.71
VirtualDispatch3	8	0	6	0	−2	0	−2	−0.25
VirtualDispatch4	–	–	–	–	–	–	–	–
LifecycleTest								
ActivityEventSequence1	58	35	73	0	15	−35	−20	−0.22
ActivityEventSequence2	32	24	77	0	45	−24	21	0.38
ActivityEventSequence3	233	116	156	1	−77	−115	−192	−0.55
ActivityLifecycle1	99	72	156	7	57	−65	−8	−0.05
ActivityLifecycle2	47	34	33	0	−14	−34	−48	−0.59
ActivityLifecycle3	65	31	28	0	−37	−31	−68	−0.71
ActivityLifecycle4	49	33	14	0	−35	−33	−68	−0.83
ActivitySavedState1	20	0	7	1	−13	1	−12	−0.6
ApplicationLifecycle1	37	10	82	0	45	−10	35	0.74
ApplicationLifecycle2	86	17	94	155	8	138	146	1.42

App Name	Forwards		Backwards		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
ApplicationLifecycle3	32	12	21	0	−11	−12	−23	−0.52
AsynchronousEventOrdering1	51	31	16	0	−35	−31	−66	−0.8
BroadcastReceiverLifecycle1	4	0	4	0	0	0	0	0.0
BroadcastReceiverLifecycle2	109	44	248	114	139	70	209	1.37
BroadcastReceiverLifecycle3	3	0	195	110	192	110	302	100.67
EventOrdering1	61	29	30	0	−31	−29	−60	−0.67
FragmentLifecycle1	187	127	90	0	−97	−127	−224	−0.71
FragmentLifecycle2	–	–	–	–	–	–	–	–
ServiceEventSequence1	53	20	152	34	99	14	113	1.55
ServiceEventSequence2	64	21	389	220	325	199	524	6.16
ServiceEventSequence3	46	12	275	151	229	139	368	6.34
ServiceLifecycle1	119	44	42	0	−77	−44	−121	−0.74
ServiceLifecycle2	68	20	89	21	21	1	22	0.25
SharedPreferenceChanged1	13	0	20	1	7	1	8	0.62
ReflectionTest								
Reflection1	15	5	8	0	−7	−5	−12	−0.6
Reflection2	21	5	11	0	−10	−5	−15	−0.58
Reflection3	42	9	62	25	20	16	36	0.71
Reflection4	9	0	8	0	−1	0	−1	−0.11
Reflection5	16	1	11	0	−5	−1	−6	−0.35
Reflection6	7	0	134	51	127	51	178	25.43
Reflection7	15	5	15	11	0	6	6	0.3
Reflection8	35	7	14	0	−21	−7	−28	−0.67
Reflection9	42	7	21	0	−21	−7	−28	−0.57
ThreadingTest								
AsyncTask1	22	2	11	1	−11	−1	−12	−0.5
Executor1	34	7	17	0	−17	−7	−24	−0.59
JavaThread1	34	7	17	0	−17	−7	−24	−0.59
JavaThread2	62	10	31	8	−31	−2	−33	−0.46
Looper1	49	3	20	16	−29	13	−16	−0.31
TimerTask1	203	28	32	33	−171	5	−166	−0.72
∅ Propagations	70.74	21.42	61.94	41.54	−8.8	20.11	11.32	1.41

Table 6.1.: DroidBench Performance Evaluation Results

6.1.2. Result Explanation

We define tests with a relative difference greater than 10 as worth investigating. In the following, we explain why our implementation performed worse than expected.

PrivateDataLeak3 This test contains two sinks and one source. The tainted data is written to a file, later read from the file and then leaked. FlowDroid does not support tracking taints over files, so it only finds a leak from source to file write but misses the leak from file read to send SMS. Due to EasyTaintWrapper's simplicity, overtainting happens in the backward direction. When `FileInputStream fis = openFileInput("out.txt");` is called with `fis` tainted, EasyTaintWrapper also taints the base object - the `MainActivity` in this case. As the `MainActivity` has a enourmous scope, the taint has a long lifetime and many other taints could derive from this taint. This taint explains the relative difference of 17.68. Using the more precise SummaryTaintWrapper, the edges reduce to $(51, 16)$ and a relative difference of 2.53, which is more reasonable. It is still higher because of the second sink.

MultiHandlers1 Two `LocationListeners` are registered in different activities. In both activities, an instance field is a parameter of a sink. So there are two possible paths where something could be leaked. The LocationListener does not call any source on the first path, while the second path has an empty setter method killing the taint. For the first path, the backward analysis has to propagate the taint into the `LocationListener` to notice that this is a dead-end while the forward's search does not even start there. For the second path, the backward analysis seems to suffer because it starts at an instance field taint with a larger scope than a local variable.

BroadcastReceiverLifecycle3 The test contains five sinks but only one source. If we only consider the leak path, both implementations perform equally. The four other sinks are responsible for the overhead on edge propagations.

Reflection6 The reflective call site has multiple callees in the interprocedural control-flow graph. Backward all of these callees are visited, of which only one contains a source statement. Forwards, the taint is introduced in the callee at the source and just one return site needs to be processed.

6.2. Real World Apps

6.2.1. Configuration

Our test machine is equipped with four Intel Xeon E5-4650 and 1 TB of RAM. We limited the JVM to 50 GB RAM and FlowDroid on 16 threads per instance. We ran at most four instances in parallel to ensure a one-to-one mapping between CPU threads and FlowDroid threads. Note that the test machine is a shared system, but we made sure there are always enough resources for our evaluation available. Still, background services might influence the performance of a single run. To stamp out this factor, we ran each app three times. If there were outliers, we repeated the runs.

For this evaluation, we chose to use a non-default configuration of FlowDroid. First, we disabled static field tracking due to the global scope as described in section 3.2. Next, instead of the `EasyTaintWrapper`, we use the `SummaryTaintWrapper`, which utilizes StubDroid. StubDroid's precomputed dataflow summaries are more precise than `EasyTaintWrapper`'s simple rules. We set the timeout for the dataflow analysis to 10 minutes. The configuration summary is in Table 6.2.

Option	Value
Array Size Tainting	disabled
Inspect Sources & Sinks	disabled
Static Field Tracking	disabled
Ignore Flows in System Packages	enabled
Exclude Soot Library Classes	enabled
Timeout	10 minutes
Taint Wrapper	`SummaryTaintWrapper`

Table 6.2.: Real World Apps Configuration

We did not use the full sources and sinks list included in FlowDroid because such would result in hundreds of sources and sinks per app and probably a long runtime. Instead, we chose to analyze which sensitive and possibly user-identifying data is sent out to the internet. As we want to compare the forwards and backward implementation, it is also essential to not put one at a disadvantage. We opted for a 2:1 ratio of sources to sinks. This decision is based on the results of SuSi, a tool to automatically find sources and sinks in the Android framework [11]. Their extracted list of sources and sinks contains roughly

2.17 times more sources than sinks. The list of sources and sinks used in this evaluation is in Table 6.3 and Table 6.4.

Class	**Method**
android.location.Location	getLatitude() getLongitude()
android.location.LocationManager	getLastKnownLocation()
android.telephony.TelephonyManager	getDeviceId() getSubscriberId() getSimSerialNumber() getLine1Number() getImei() getMeid()
android.bluetooth.BluetoothAdapter android.net.wifi.WifiInfo	getAddress() getMacAddress() getSSID() getIpAddress()
java.net.InetAddress	getHostAddress()
android.telephony.gsm.GsmCellLocation	getCid() getLac()
android.content.pm.PackageManager	getInstalledApplications() getInstalledPackages() queryIntentActivities() queryIntentServices() queryBroadcastReceivers()
android.content.SharedPreferences android.provider.Browser	getDefaultSharedPreferences() getAllBookmarks() getAllVisitedUrls

Table 6.3.: Sources for Real World Apps Evaluation

We used FlowDroid's forwards implementation on the to that date latest upstream commit[2] from the develop branch for the point of comparison. The backward implementation ran on our latest commit with all changes merged from the upstream.

We chose 200 apps randomly out of a Google Playstore dump from 2021 containing over 6000 apps for our evaluation set. The full list is appended to this work in **??**.

[2]The latest upstream commit was at that time b436733fc4a5130dfe4ce8ddb3f76fd374e9a487.

Class	Method
java.net.URL	set() openConnection()
java.net.URLConnection	connect() setRequestProperty()
android.net.http.HttpsConnection	openConnection()
android.net.http.Headers	setEtag() setContentType() setLastModified() setLocation()
android.net.http.AndroidHttpClientConnection	sendRequestHeader()
android.net.http.RequestQueue	queueRequest()

Table 6.4.: Sinks for Real World Apps Evaluation

6.2.2. Results

Out of 200 apps, 60 apps do not have any sources or sinks and thus, the analysis did not start. For 15 apps, the analysis aborted with errors. We are left with 125 apps for which both implementations completed the analysis.

Preliminary! Only one run without memwatcher

Metric	Forwards	Backwards
Average Runtime	$498.192s$	$397.152s$
Median Runtime	$600.0s$	$600.0s$
Abstractions Infoflow	34223577	12318997
Abstractions Alias	12347994	30013930
Total Abstractions	46571571	42332927
Memory Timeouts	3.2%	6.4%
Time Timeouts	80.8%	57.6%

Table 6.5.: Results

Avg w/o timeouts is only per direction, so bw contains way more entries

6.2.3. Comparison to forwards analysis

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

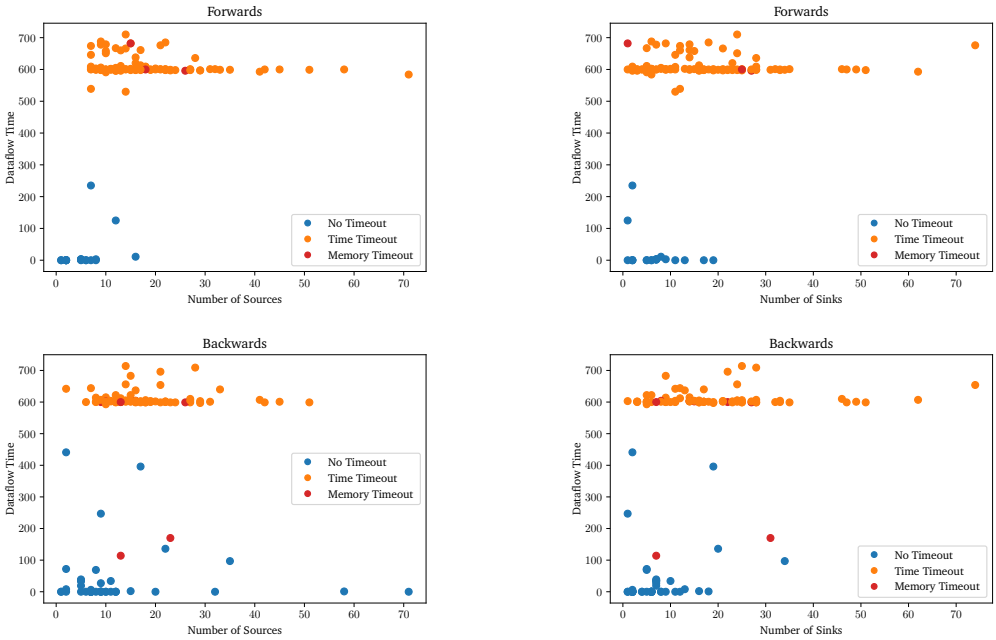


Figure 6.1.: Dataflow time in comparison to source and sink count

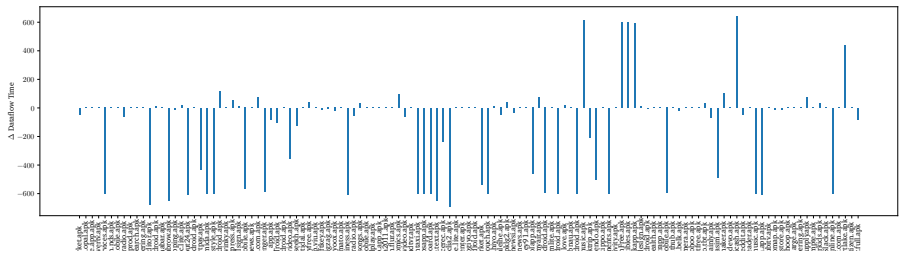


Figure 6.2.: Dataflow Delta

	Forwards	Backwards
Average Runtime	$19.15s$	$38.07s$
Median Runtime	$0.0s$	$0.0s$
Abstractions Infoflow	459868	629202
Abstractions Alias	155876	1403538
Total Abstractions	615744	2032740

Table 6.6.: Results without Timeouts

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

7. Related Work

Yan et al [18] proposed a vulnerability detection tool for PHP with a focus on web applications. They aim to detect typical web application vulnerabilities such as cross-site-scripting and SQL injections using backwards taint analysis. Instead of relying on reducing the problem to proven frameworks such as IFDS or IDE, they seemingly define their own data flow algorithm. The proposed algorithm traverses the basic blocks backwards and copies taints left after traversing the basic block to its predecessors. Unlike in our work and in general in data flow analysis, they do not try to reach a fixpoint, instead they just do not follow circular paths in the control-flow graph. They also emphasize their concept of "cleans": a predefined list of sanitization methods which kill the incoming taints. In FlowDroid the same is possible using taint wrappers and both shipped implementations support such a concept. A rationale for traversing backwards, which is why we included it as related work, is not provided.

Synchronized pushdown systems (SPDS) by Späth et al [15] are an alternative to IFDS with access paths for modelling a precise context-, flow- and field-sensitive data flow analysis. Similiar to IFDS it constructs a context-free grammar representing the call stack to ensure context-sensitivity. In addition, it also modells field-sensitivity using a context-free grammar. Then it computes the acceptance state of both pushdown automaton to combine context- and field-sensitivity. This allows to represent recursive access paths such as `lst.next.prev.next...` without loss of precision where with IFDS and access paths, k-limiting is needed to ensure termination. SPDS are still an overapproximation in the case where at a statement both automaton are in acceptance state but via a different paths.

Another way to realize data flow anlaysis

FlowTwist? He starts in the middle and searches fowards and backwards.

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] A. Deutsch. "Interprocedural may-alias analysis for pointers: beyond k-limiting". In: *PLDI '94*. 1994. DOI: 10.1145/178243.178263.

[6] Neil Jones and Steven Muchnick. "Flow Analysis and Optimization of Lisp-Like Structures." In: Jan. 1979, pp. 244–256. DOI: 10.1145/567752.567776.

[7] Uday Khedker, Amitabha Sanyal, and Bhaurao Sathe. *Data Flow Analysis: Theory and Practice*. Jan. 2009. ISBN: 9780849332517. DOI: 10.1201/9780849332517.

[8] Patrick Lam et al. "The Soot framework for Java program analysis: a retrospective". en. In: Oct. 2011. URL: http://www.bodden.de/pubs/lblh11soot.pdf (visited on 03/11/2021).

[9] 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.

[10] 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.

[11] Siegfried Rasthofer, Steven Arzt, and E. Bodden. “A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks”. In: *NDSS*. 2014. DOI: 10.14722/NDSS.2014.23039.

[12] Thomas Reps, Susan Horwitz, and Mooly Sagiv. “Precise interprocedural dataflow analysis via graph reachability”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’95*. ACM Press, 1995. DOI: 10.1145/199448.199462.

[13] H. Rice. “Classes of recursively enumerable sets and their decision problems”. In: (1953). DOI: 10.1090/S0002-9947-1953-0053041-6.

[14] Jonathan Rodriguez and Ondřej Lhoták. “Actor-Based Parallel Dataflow Analysis”. en. In: *Compiler Construction*. Ed. by Jens Knoop. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 179–197. ISBN: 9783642198618. DOI: 10.1007/978-3-642-19861-8_11.

[15] Johannes Späth, Karim Ali, and Eric Bodden. “Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems”. In: *Proceedings of the ACM on Programming Languages* 3 (Jan. 2019), pp. 1–29. DOI: 10.1145/3290361.

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

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

[18] X. Yan, H. Ma, and Q. Wang. “A static backward taint data analysis method for detecting web application vulnerabilities”. In: *2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN)*. ISSN: 2472-8489. May 2017, pp. 1138–1141. DOI: 10.1109/ICCSN.2017.8230288.

A. Appendix