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: 28th March 2021

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

TECHNISCHE
UNIVERSITÄT
DARMSTADT

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, 28th 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. Finally, 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 the 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 that try to identify data flows through a program. Khedker [10] 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 [10]. A special kind of data flow analysis is taint analysis, which concepts might be familiar from code reviews. In taint analysis, the goal is to determine whether a particular variables' information contents flow through the program to another variable. Variables

that contain such valuable information are *tainted*. 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]. For example, to detect apps tracking the user using taint analysis, sources could be methods returning a unique identifier and sinks could be methods that sent 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 [10, 2]:

- **Flow Sensitivity**: A flow-sensitive analysis can determine if a fact holds at a particular statement.

- **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 undecidable [17]. However, 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 [16].

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 data flow 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.

$$matched \;\to\; (_i \; matched \;)_i \; matched \; | \; \epsilon$$
$$valid \;\to\; valid \; (_i \; matched \; | \; matched$$

Figure 2.1.: Context-Free Grammar proposed by Reps et al.[16]

To achieve context-sensitivity, Reps et al. proposed a context-free grammar (c.f. Figure 2.1). Each call site gets a unique index, outgoing call edges are labeled with $(_i$ and incoming return edges are labeled with $)_i$. *matched* ensures the correct return edge is taken and *valid* models a partially balanced path, e.g., after a call for which the return is still pending. A path between two nodes is called valid iff the sequence of labeled edges is a string in *valid* [16]. Consider the example in Figure 2.2. After the first call to bar, the sequence is $(_1$, which is in *valid*. Now, there are multiple return edges from bar but only $)_1$ is in *matched*. Thus, the $)_1$ edge is taken. With the context-free language, this is also called meet-over-all-valid-paths (MOVP). MOVP is more precise but as sound as before ($MOVP \sqsubseteq MOP$). Still, the valid paths are an overapproximation because they include paths never taken at runtime, called infeasible paths.

```
void foo(){
    int i = 0;
    int j = bar(i);          int bar(int p){
                                 return p + 42;
    int k = bar(j);          }

    return k;
}
```
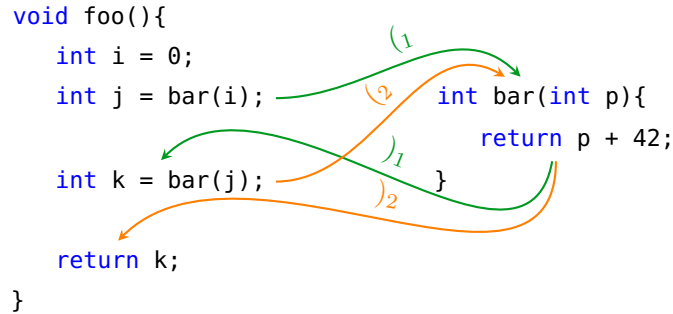
Figure 2.2.: Context-Sensitivity using a Context-Free Language

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: [16]

- **Call Flow**: Edges from call statement into a method. The flow function maps the facts visible in the callee into it.

- **Return Flow**: Edges returning from a method. The flow function maps the facts visible in the caller out of the callee.

- **Call To Return Flow**: Edges over a call statement. The flow function maps the facts not visible in the callee over the call statement.

- **Normal Flow**: The default case. Handles edges over every other statement, for example, assignments.

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 [16]. 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 [16].

Eventually, there are no facts to propagate anymore and the analysis will terminate. There are two ways for a fact to be not propagated further. Either a flow function killed the fact or the same fact was already observed at a statement, meaning the IFDS analysis reached a fixpoint [16].

However, we already started this section, hinting not all problems can be formulated in the IFDS framework. 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. Distributiveness is essential for the correctness of IFDS, because only if the flow functions are distributive the maximum fixed point (MFP) equals MOP and MFP is computable in polynomial time [10, 16].

Finite Another restriction is that the set of data flow 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 data flow fact is a tuple of the variable together with the stored value $\langle x, v \rangle$. Consider Figure 2.3. x is initialized to an empty string and in every loop iteration, "a" gets appended to the string. The value of x changes every time and never repeats itself. In theory, the algorithm will never observe a taint twice in line 4. Because the algorithm can not reach a fixpoint, it will not terminate. In practice, every data type is bounded either by the heap or stack size, but the domain is cubic in the time-complexity $O(|E| \cdot |D|^3)$ making IFDS infeasible for large domains [16].

```
1  void main() {
2      String x = "";
3      while (condition)
4          x += "a";
5  }
```

$$\langle x, \epsilon \rangle \to \langle x, a \rangle$$
$$\langle x, a \rangle \to \langle x, aa \rangle$$
$$\langle x, aa \rangle \to \langle x, aaa \rangle$$
$$\dots$$

(a) Code   (b) Taint Derivations

Figure 2.3.: Finiteness example

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 [10]. 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.
The paper by Reps et al. later decides on set union due to the duality of must and may not [16]. 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 [14].

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 [14]. 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 [5].

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 [14].

Furthermore, the original definition starts the IFDS algorithm at the entry point of the interprocedural call-graph. As described in subsection 2.2.1, 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. Lerch et al. first described the extension [13] and it is also present in FLOWDROID [2].

Another optimization is possible if the merge operator is set union thanks to the $A \subseteq A \cup B$ property of 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. Hence, the IFDS solver can skip the in-set construction and immediately propagate the outcoming facts as singleton sets, sometimes referred to as point-wise propagation. Especially parallelized IFDS implementations benefit from point-wise propagation [18].

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 [10]. 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 with loops such as doubly linked lists, this gets problematical. Consider Figure 2.4, 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. This cut-off comes with a loss of precision [9]. Consider again Figure 2.4. With $k = 2$ the analysis would reach the fixpoint $lst.next.prev.*$ after two iterations.

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

$$lst \rightarrow lst.next.prev$$
$$lst.next.prev \rightarrow lst.next.prev.next.prev$$
$$...$$

(a) Code
(b) Access Path Iterations
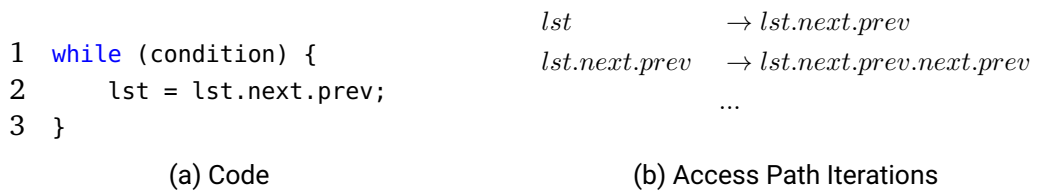
Figure 2.4.: Infinite Access Path

Although with k-limiting, the algorithm terminates, it does have another problem. After a loop like in Figure 2.4, 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. As a solution, Deutsch proposed symbolic access paths, which try to eliminate loops in access paths [7]. 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]. With symbolic access paths k-limiting is theoretically obsolete but still often applied in practice to speed-up the analysis.

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 [21]. Such an IR allows compilers to apply machine-independent optimizations to the code without worrying about complex expressions in the source code or 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 2.5 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 [22].

```
1  int a = 21;
2  int b = 21;
3  int c = a + b;
```

(a) Java code

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

(b) Java bytecode

Figure 2.5.: Java bytecode example

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. Operands are represented by variables instead of registers or stacks. Such a representation fixes the expression length to be better suited for static analysis than assembly. It also

---

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

reduces the possible combinations to a manageable amount compared to the source code written by a human [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 statements are split up into multiple statements. For example, there can be only one field reference per statement and arguments are always local variables or constants [22]. This groundwork dramatically reduces the possible cases the data flow analysis needs to consider and eases the analysis.

2.5. FlowDroid

FLOWDROID is a precise context-, flow-, object- and field-sensitive static taint analysis tool for Android apps[4]. 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 [11]. 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, multiple callbacks are 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 provide a single entry point for the call graph generation. [4].

FLOWDROID inherits flow- and context-sensitivity from IFDS and the object- and field-sensitivity from symbolic access paths. To provide precise results even with aliases in use, FLOWDROID contains, besides the taint analysis, another IFDS problem to resolve all encountered aliases on-demand. The two IFDS analyses are intertwined to maintain flow- and context-sensitivity between both analyses.[4].

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

---

[2]https://github.com/secure-software-engineering/FlowDroid
[3]Taken from https://developer.android.com/guide/components/activities/activity-lifecycle.
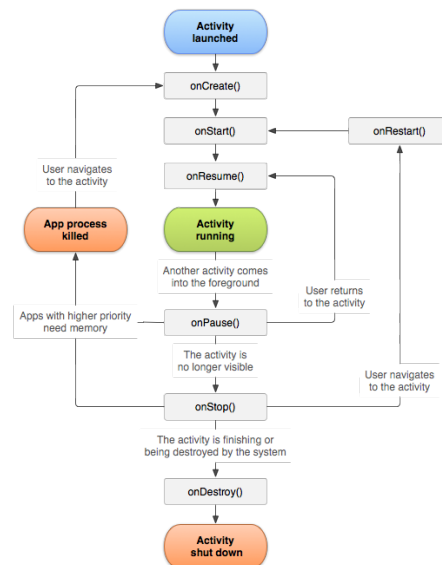
Figure 2.6.: Activity Lifecycle[3]

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 [4]. 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 for the IFDS analysis. In the second part, we discuss taint analysis's runtime, highlighting possible differences between forward and backward analysis.

3.1. Flow Functions

We describe the flow functions' behavior based on the Jimple language and define semi-formal rules analogous to the publication[2] on FLOWDROID. These rules only focus on the basic language constructs. We describe flows for additional language features such as arrays and exceptions later and more informal in section 4.3.

3.1.1. Normal Flow

Normal flow functions handle every statement that does not contain an `InvokeExpr`. For the base cases in normal flow, new taints are only produced at assignments. Assignments are always explicit in Jimple and are either `AssignStmt`'s or `IdentityStmt`'s. The `IdentityStmt`'s are at the top of a method[1] and assign special values to locals, e.g., parameters and the `this` reference. We perform the identity function over those because we want to keep those taints alive to reach the return edge. Then the Return Flow function takes care of mapping all parameters back into the caller[2]. So in the following, we only consider `AssignStmt`'s.

---

[1]With the exception of `local_name := @caughtexception`, which is outside of the base cases.
[2]Note that traversing the interprocedural control-flow graph backward means call edges are now return edges and vice versa.

Now, let us consider an `AssignStmt`. The left side is either a field reference or a local and the right side is an expression. In the following, we assume the right side is always a variable. The assignment has the structure $x.f^n \leftarrow y.g^m$ with $n, m \in \{0, 1\}$ modeling a possible field reference. As taints may have an access path of an arbitrary length, we denote this as h^k.[3] Jimple also ensures only one field dereference per statement, which Arzt chose not to represent in the semi-formal definitions and neither did we.

In the first case, we look at exact matches. Either we have an assignment with a local ($n = 0$) or a field dereference ($n = 1$). For both, the base variable needs to match. For the latter, also the first field of the access path has to match the field dereference. The first field dereference is removed from the taint and the remaining access path is copied to the newly created taint. The incoming taint is killed because, thinking forward, it received the taint at this statement.

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

Next, we need a rule for the case the field dereference f is included in a cut-off approximation. Recall section 2.3, symbolic access paths can also be k-limited to speedup the analysis and are $k = 5$-limited in FLOWDROID by default. Thus, we might encounter a taint with no field dereferences and a wildcard $*$ appended. In this case, just the base needs to match. However, this time, the left side is kept alive because we can not reason which field is tainted due to the cut-off approximation.

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

Lastly, we could also observe a taint on the right side. In this case, we apply the identity, so propagate the taint over the statement. This case is necessary later when we consider aliasing in section 4.2.

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

> **Rule 4:** An incoming taint $T = y.*$ produces the outflowing taint set $\{T\}$.

Whenever a taint neither matches on the left nor the right side, we also perform the identity as the statement does not touch the tainted variable's contents.

---

[3] h^k is a k-length chain of field dereferences, not k-times the same field dereference

3.1.2. Call Flow

The Call Flow function and subsequently Return and Call To Return Flow function apply whenever a statement contains an `InvokeExpr`. For call statements without an assignment, 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 instance of the callee's base object.

When we encounter a tainted argument in the caller, the taint needs to go through the callee. Java uses pass-by-value, so the arguments are copied into the callee. Thus, for primitives, the value is copied and pushed on the stack. For reference types, the pointer to the object is pushed on the stack. In the second case, if only the base reference is tainted but nothing more ($k = 0$ and no wildcard), the callee can only access and overwrite the reference saved in the parameter on the stack but not change the object's data on the heap. We know the update that tainted the primitive or reference without field dereferences can not be inside the callee due to the backward direction. This property becomes apparent when we get specific to Java's types. Primitives do not have fields and strings are immutable[4]. Consider the example in Figure 3.1. On the left, we use the built-in String type. In line 2, `str` is copied into `callee`. After this statement, both `str` hold the value 42 but point to another memory location[5]. Thus, `main` carries on with the original value of `str` no matter what `callee` writes to `str`. In contrast, on the right, the callee can update the field on the heap. Therefore, the taint needs to be propagated into the callee to find the leak. Conclusively, k needs to be greater than 0.

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

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

A non-static callee can also access instance fields of the base object. When we observe a tainted base object, the taint also needs to flow through the callee. The tainted object transforms into a *this* reference. In Java, `this` references the current instance the method operates on.

besser formulieren

---

[4]The special handling of strings results in transparent fields, e.g. we can treat strings as if they were primitives in this case.

[5]The JVM might only set a copy-on-write flag on `str` in `callee` and point it to the identical location as `str` in `main` to save memory. At least right before the update happens, it is guaranteed that the variable points to a different location.

```
                                            1  void main() {
1  void main() {                            2      SomeObject o = new
2      String str = "42";                            ↪ SomeObject();
3      callee(str);                         3      callee(o);
4      sink(str); // no leak                4      sink(o.str); // leak
5  }                                        5  }
6                                           6
7  void callee(String str) {               7  void callee(SomeObject o) {
8      str = source();                      8      o.str = source();
9  }                                        9  }
```

    (a) Taint Without Fields   (b) Taint With Fields

Figure 3.1.: Call Flow Example

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

Rule 4: An incoming taint $T = o.*$ produces the outflowing taint set $\{this_c.*\}$.

Static fields form a special case. Their scope extends over the whole program and thus, tainted static fields always have to go through the callee. The taint is untouched as the access path to those is the same everywhere.

Rule 5: An incoming taint $T = S.f.h^k$ with $k \geq 0$ produces the outflowing taint set $\{T\}$.

In Jimple, `AssignStmt`'s can also consist of an `InvokeExpr` on the right side. The structure of the statement is in this case $x \leftarrow o.m(a_0, ..., a_n)$. r_i denotes a return value. m is the number of return statements in the callee. If we observe such a statement and the left side is tainted, we need to map the returned value back into the callee. Now, methods can have multiple return statements and as we traverse the reversed interprocedural control-flow graph, there are multiple outgoing edges. We can not reason which return statement is the right one, so we need to taint every return statement's operand in the callee.

Rule 6: An incoming taint $T = x.h^k$ with $k \geq 0$ produces the outflowing taint set $\{r_i.h^k \mid 0 \leq i < m\}$.

Unlike at normal flows, we kill all taints not matching any of the rules. In the case of a taint being out of the callee's scope, the Call To Return flow function propagates the taint over the statement.

3.1.3. Return Flow

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

The first rule is the counterpart rule 1 and 2 of Call Flow[6] and map all parameters back into the caller. In contrast to the Call Flow, we also map primitives and strings back into the caller. Thus, the taints are also visible in the caller.

> **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 $\{a_i.h^k\}$.

The *this* reference also needs to be mapped back into the caller.

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

> **Rule 3:** An incoming taint $T = this_c.*$ with $k \geq 0$ produces the outflowing taint set $\{o.*\}$.

Tainted static fields are also mapped back untouched. This rule is the same as rule 5 of the Call Flow.

> **Rule 4:** An incoming taint $T = S.h^k$ with $k \geq 0$ produces the outflowing taint set $\{T\}$.

Again, taints not matching any rule are killed. For example, this kills taints, which are not in the caller's scope, when returning from a method.

3.1.4. CallToReturn Flow

The statement structure is $o.m(a_0, ..., a_n)$ with $n \in \mathbb{N}$. a_i denotes the i-th argument.

A taint is not in the callee's scope if it is not static and neither matches an argument nor the base object the method is called on. Such a taint is not matched inside Call Flow and needs to be propagated over the call statement.

> **Rule 1:** An incoming taint $x.h^k$ with $k \geq 0 \wedge (\forall i \in [0, n] \cap \mathbb{N} : a_i \neq x) \wedge x \neq o \wedge x \notin StaticVariables$ produces the outflowing taint set $\{T\}$.

---

[6]Note that if k can be 0, the wildcard also works.

Consider again the left side of Figure 3.1. In line 3, the taint is in the kill set of Call Flow. As we want to preserve the taint after the call, we need to propagate the taint over the call statement in such cases.

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

Like in Call and Return Flow, we also kill taints that do not match any of these rules.

3.2. Runtime of the Data Flow Analysis

IFDS has a worst-case time complexity of $O(|E| \cdot |D|^3)$. $|E|$ is the number of observed edges in the control-flow graph and $|D|$ is the number of tainted variables observed by the IFDS analysis. Concluding, it highly depends on the choice of sources and sinks and the analyzed app. Therefore it is not possible to make a general statement about a better analysis direction. Nevertheless, what we can do is discuss certain cases where one direction is favorable. We decide favorably based on the number of taint propagations, the observed edges in the exploded supergraph. They correlate to the runtime and depend on two factors: taints' lifetime and the number of taints. We explain how the analysis direction influences both factors 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 3.2.: Branching Factor Example

First, we take a look at the is the branching factor. The branching factor describes the number of outgoing edges from a node. A smaller branching factor is favorable. Consider a binary operator expression such as int c = a + b;, backward we can not argue which

(a) Forwards   (b) Backwards
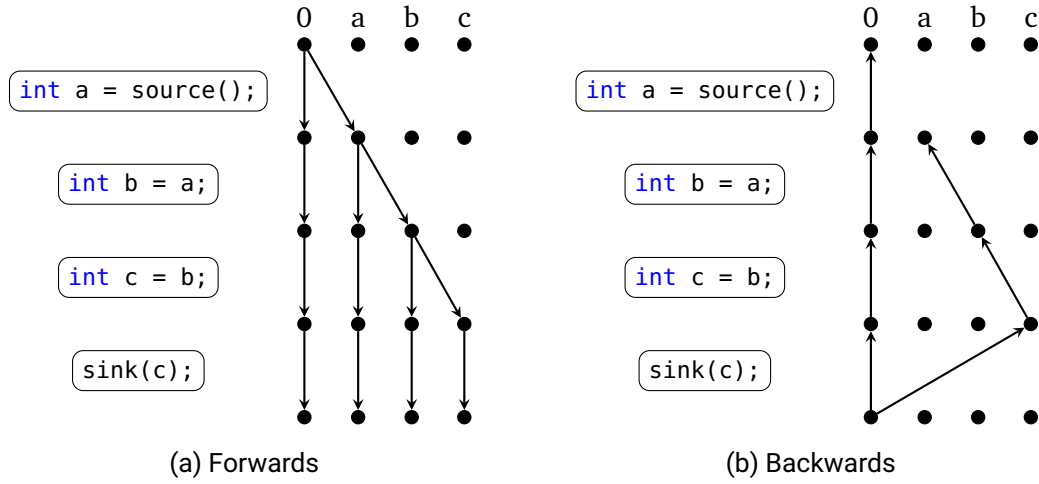
Figure 3.3.: Right-to-Left Order

operand is responsible for the tainted output and thus proceed with both operands tainted. The same restriction is present in rule 6 of Call Flow which describes how the returned value is mapped back into the callee. This time the branching factor can be even larger. For example, in Figure 3.2 is a method which conditionally returns one of its parameters and is part of the leak path. Let us assume the returned value of a call to `returnParam()` is tainted. Backward, every returned operand is tainted and later on mapped according to Return Flow rule 2 back into the caller. The effect gets apparent when looking at the method summary. 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 three possible summaries $s1 \rightarrow \{retVal, s1\}$, $s2 \rightarrow \{retVal, s2\}$ or $s3 \rightarrow \{retVal, s3\}$. Such cases favor forward analysis.

In contrast, a strict right-to-left flow favors backward analysis as taints are killed more often due to a stronger overwrite rule. In Figure 3.3 is such a right-to-left flow displayed. Forward, the right-hand side is always kept alive because it still holds the tainted value below the statement and could be leaked. When traversing a right-to-left flow backward, the left side is always killed. A visited statement is the update responsible for tainting, leading to a branching factor of 1 and a shorter lifetime per taint.

We already briefly mentioned the global scope of static field taints in the last section. Hence, unless the static taint is overwritten, it traverses the whole interprocedural control-flow graph. FLOWDROID already applies an optimization and looks ahead to skip methods

in which the static field is not used. Still, the longer lifetime stays an issue [2].

Now, we discussed the complexity based on the taint propagations. Though the taint propagations are only known after the analysis, it would be beneficial to decide which direction is the best before analyzing an app. An obvious choice for a clue would be the ratio of sources and sinks. If one is much less than the other, we could argue less taints to start with should also lower the taint propagations. Likewise, Lerch et al. claim in their work that a magnificent smaller amount of sinks than sources are advantageous for a backward-directed search [12]. Sadly, it is not as easy to generalize the statement to less starting taints means less runtime. Arzt's evaluation of FLOWDROID has shown no correlation between the number of sources and the runtime in FLOWDROID[2]. This result is probably owed to the unpredictable taint's lifetime balancing out the initial advantage. As a part of this work, we evaluate whether it is possible to decide the favorable analysis direction in `FlowDroid` on the same app in section 6.2.

In general, we can not predict the lifetime of taints before actually analyzing the app. However, in special cases of application, this might be naturally given. Think of a case where sanitization methods[7] are in use. Depending on the use case, it might be possible to deduce the proximity between sanitization methods and sources or sinks. Whenever this is possible, there should be a clear favorite for one direction due to the taints' lower lifetime.

Summarized, we discussed cases where one direction seems favorable, but no generalized statement can be made on which analysis direction is better. Further, most of the influencing factors depend on the app and thus, at most, a favorable direction can be determined for a single app.

---

[7]Sanitization methods are run against user input to ensure the input is safe to be processed.

4. Implementation

This chapter describes the details of our backward-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 many components of FlowDroid as possible.

First, we needed a backward interprocedural control-flow graph. FlowDroid already contained one for the on-demand aliasing, which only missed the `notifyMethodChanged()` method. Next, we need to introduce unconditional taints at sinks and check for the matching access paths at sources. The methods for retrieving sources and sinks from a Source Sink Manager have different signatures because, in the forward analysis, access paths only have to match at sinks. 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 source sink managers needed for the data flow analysis now implement the corresponding interface. Note that reversible source sink managers currently do not support the one-source-at-a-time mode.

For the core flow functions, we created two new classes implementing `IInfoflowProblem`. `BackwardsInfoflowProblem` implements the flow functions described in section 3.1. We also refer to this as the main analysis. Additional language features are sourced out into rules which are informally described in section 4.3. The second class is `BackwardsAliasProblem` which is responsible for the on-demand forward alias analysis. We describe the on-demand aliasing in greater detail in section 4.2.

After the analysis, the path builder constructs a path out of the leaked taint and its predecessors. Because the path builder expects a forward-built taint, the path ends up being the wrong way round. To hide the fact that we internally searched backward, we also created a `BackwardsInfoflowResults` extending `InfoflowResults`. The implementation is quite simple. It overwrites only the `addResult` implementations to swap the start and end. If full path reconstruction is enabled, it also reverses the path in between.

4.2. Flow-Sensitive Alias Analysis

FLOWDROID offers multiple aliasing strategies. In this work, we focus on flow-sensitive alias analysis. The analysis is another IFDS problem. However, this time, it is a forward-directed IFDS analysis using flow function with aliasing rules. The main analysis invokes the alias analysis on-demand when it discovers an alias. The alias analysis runs independent from the main analysis and later injects found aliases back into the main analysis.

Note that pointer analysis itself is a non-distributive problem [2, 19]. Nonetheless, the alias analysis is encoded in IFDS and we just accept the possibly imprecise results due to the overapproximation. Not using the intertwined alias analysis is too imprecise and the cases of overapproximation are rare in practice [2].

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

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

(a) Example for alias analysis initiated by rule 1

(b) Example for alias analysis initiated by rule 3

Figure 4.1.: Normal flow Aliasing examples

Handover between the analyses   The main analysis discovers aliases at assignments. Consider Figure 4.1 where two different cases are displayed. On the left is a normal flow according to rule 1. In line 2, the in taint `a.str` produces the outcoming taint `b.str`. Because the assignment type is a heap type, the backward analysis now recognizes that it possibly missed updates to `b.str` below of line 2. It invokes the alias analysis with `b.str`. On the right is a normal flow according to rule 3. This time the assignment in line 2 is swapped. The main analysis leaves the incoming taint `b.str` untouched but notices `a` aliases `b` below line 2, hence invoking the alias analysis with `a.str`.

The alias analysis searches for missed updates. If the analysis found an update, e.g., the taint is on the left side of the assignment, the analysis injects an edge to the statement with the taint into the main analysis' worklist. Consider again Figure 4.1a. In line 3, the alias analysis encounters the tainted `b.str` on the left side. At this point, `b.str` gets handed back to the main analysis, following the missed update to find a possible leak. In this case, the leak happens right away.

```
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

Maintaining Flow Sensitivity   Arzt solved this in the existing forward implementation using an activation unit. This statement marks the update at which the alias gets tainted and can leak at sinks. This concept does not work for our backward implementation as the alias analysis traverses forward where a write to a variable means a leak. Thus we introduce the turn unit. The turn unit holds the last non-aliasing assignment. When a taint reaches its turn unit in the aliasing analysis, the analysis kills the taint. Consider Figure 4.2. The introduced taint `str` in line 6 also has line 6 as a turn unit. In line 4, a non-aliasing assignment happens. In line 2, the alias analysis starts for `a.str`. Without the turn unit, the taint would pass line 3. Further in line 5, the taint is handed to the main analysis. The main analysis then reports a leak. With the turn unit, the alias analysis kills the taint in line 4, preventing the false positive.

The turn unit is a new field in the `Abstraction` class, which is representing a taint. A

```
1  void foo() {
2      // [...]
3      bar(someObject1);
4      sink(someObject1);
5
6      bar(someObject2);
7      sink(someObject2);
8  }
```

Figure 4.3.: Summaries with Turn Units

possible drawback of the backward analysis could be the reusability of the IFDS summaries. Because the turn unit is part of the taint, IFDS treats equal taints with different turn units as if they have a different context. Consider Figure 4.3. str1 and str2 are equal taints, but one has the turn unit at line 7 while the other one has the turn unit set to line 4. Let us assume IFDS already traversed the call to bar(someObject2); in line 6 and created a summary from it. Later, it observes the same callee but with someObject1 as an argument. Though, because the turn units differ, IFDS can not apply the already existing summary. In this case, applying the summary would not be harmful. However, if the turn unit is inside the callee or the transitive callees, we would effectively lose the flow sensitivity as the turn unit is ignored.

4.3. Rules

Flow functions can get quite large, complicated to understand and hard to maintain [13]. To counteract this, FLOWDROID outsources certain features into rules. These rules also implement the four flow functions and are applied in the main analysis's corresponding flow function. In this section, we describe our implementation and informally state the rule behavior.

4.3.1. Source & Sink Propagation Rule

In backward 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 requiring an IReversibleSourceSinkManager.

Notably, the `DefaultSourceSinkManager` assumes the return value to be tainted. Only if the return value is ignored or the method has no return value, the base object is assumed to be tainted while at sinks base object and parameters are leaked [2]. Thus, starting at sinks results in more taints per statement than in forwards analysis. Recall 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 on real-world apps.

4.3.2. Backwards Array Propagation Rule

In FLOWDROID, array taints are overapproximated by only distinguishing contents and length but not elements. Meaning if one element of an array is tainted, FLOWDROID considers all elements tainted. Indices are often computed at runtime and thus not available for a static analysis without applying another analysis beforehand. So, the approximation is not as severe because we could only track constant indices regardless. Furthermore, distinguishing elements would increase the domain even more, subsequently increasing the runtime [2].

The Array Propagation Rule handles `ArrayNewExpr`, `LengthExpr` and `ArrayRef` on the right-hand side.

- **Array Rule 1**: If the left side's length 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.

The overapproximation of arrays also implies that array taints can not be killed if the left side is an `ArrayRef`.

4.3.3. Backwards Exception Propagation Rule

The backwards analysi first finds a catched exception in the form of `$someVar := @caughtexception`. Then it sets an exception flag at the taint and propagates the taint onwards. The subsequent propagation then finds the corresponding throw statement.

- **Exception Rule 1**: On a caught exception expression, derive a new taint with an exception flag set.

- **Exception Rule 2**: If a taint with the exception flag set occurs at a `ThrowStmt`, derive taint the operand of the `ThrowStmt`.

The second rule is present in Call and Normal Flow because the throw statement can be inside the same method or in a callee.

4.3.4. Backwards Wrapper Propagation Rule

The implementation of this rule is similar to the existing implementation. A tainted returned value also needs to be passed into the taint wrapper because of the backward direction. The rule calls `getInverseTaints()` and thus requires the taint wrapper to implement the `IReversibleTaintWrapper` interface .

4.3.5. Backwards Strong Update Rule

Until now, we always assumed that a taint is only affected if the variable occurs in a statement. However, with aliasing, this gets quite more complicated. A taint could not match the left side and, thus, is propagated 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. Also, we can not just link aliases to taints for such strong updates because that would violate the flow functions' distributiveness property.

In this case, FLOWDROID falls back to Soot's must-aliasing analysis. However, the must-aliasing analysis is only intraprocedural. Thus, strong updates split over methods are not detected and produce a false positive.

Backward, the first observed update is the correct one. We treat a must-alias like a regular match:

- **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.

4.3.6. Backwards Clinit Rule

`<clinit>` is a special method in the JVM and stands for class loader init. The compiler generates the method and calls it implicitly. Examples of statements that get compiled into clinit are in Figure 4.4. The invokation is implicit at the class's initialization phase and is executed at most once for each class[1]. SPARK, which default call graph algorithm in `FlowDroid`, overapproximates the `<clinit>` behavior. It adds an edge to `<clinit>` at each statement containing a `StaticFieldRef`, `StaticInvokeExpr` or `NewExpr` [2].

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

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

(a) static variable initialization   (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.

---

[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

- **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>`.

- **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>`.

The behavior is still an overapproximation, of course. A more precise solution would require bookkeeping of every class's last observation equal to the first occurrence in the code.

In the existing implementation, there is no such explicit. As taints are introduced 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` feature which propagates return flows for unbalanced problems, for example when the taint was introduced inside a method and therefore there was no incoming flow. This allows the forward 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. Other Components

4.4.1. Taint Wrappers

FLOWDROID already has an interface `IReversibleTaintWrapper` for taint wrappers providing inversed summaries. The `SummaryTaintWrapper` using StubDroid's summaries already implemented this interface. For the `EasyTaintWrapper`, we contributed the inverse implementation. Its implementation follows simple rules which cover most cases[2]. The rules are inverted to:

- If the return value is tainted, taint the object and the parameters.

- If the base object is tainted, taint all parameters.

```
1  char[] tainted = source();
2  StringBuilder sb = new StringBuilder();
3  sb.append(tainted, offset, len);
4  sb.append("untainted");
5  sink(sb.toString());
```

Figure 4.5.: Easy Taint Wrapper Example

Note that these simple rules are disadvantageous for the backward direction the more parameters a method has. Consider the code snippet in Figure 4.5, especially line 3. Forwards, `tainted` is the incoming taint and the `EasyTaintWrapper` produces the taint set `{tainted, sb}`. Backward, the incoming taint is `sb` and the taint wrapper produces four taints `{sb, tainted, offset, len}`. Luckily, most methods supported by the `EasyTaintWrapper` have less than three arguments.

4.4.2. Native Call Handler

The native call handler of FlowDroid handles two methods:

- `System#arraycopy`: If the first parameter is tainted, taint the second parameter.
- `reflect.Array#newArray`: If the length is tainted, propagate it over the statement.

We adapted the existing implementation and only reversed the logic of `System#arraycopy` to reflect the analysis direction.

4.4.3. Code Optimizer: AddNOPStmts

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 backward analysis,but we needed another code optimizer to handle an edge case in backward analysis.

First, consider the `static2Test` test case in the `StatictTestCode` class of FlowDroid in Figure 4.6. The method is also the entry point for the analysis, is static and does not have any parameters. The same is true for the source `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. When traversing backward, the source statement is the last and thus has no return sites. Now, the taints flowing into source methods are registered in the Call To Return flow function. Altogether, leaks are missed 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 and changing this would require multiple changes in the existing implementation and also ours. Our solution is to add a NOP statement in such cases before the analysis. Due to the entry points being known beforehand, the overhead is nearly zero.

```
 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 4.6.: static2Test Code

5. Validation

5.1. Unit Tests

FLOWDROID already contains 519 unit tests for the core component. We also validated the backward analysis with these tests. 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 backward analysis does report no leaks. This difference is related to the `EasyTaintWrapper` implementation. `equals()` and `hashCode()` are exclusive in the `EasyTaintWrapper`, which means the analysis can skip these methods because the taint wrapper provides a summary for them. The exclusive check happens in the Call Flow function. In both tests, the source is in an exclusive method. The IFDS solver behaves as already observed with `<clinit>` in subsection 4.3.6 and creates a return flow for an unbalanced problem; while going backward, the exclusive check kills the taint in the Call Flow function and applies the summary unaware of the override. We marked those two tests forwards-specific and created two equivalent backward-specific tests with sinks inside the `equals()` or `hashCode()` method with one expected leak.

HeapTestPtsAliasing We focused in this work on flow-sensitive aliasing, which is the default aliasing strategy of FLOWDROID. Other aliasing strategies are left for future work.

ImplicitFlowTests + Set contains. Not implemented. ⸻ implement later

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 [4]. The latest development version 3 includes 190 test cases[1]. We used the newest commit on develop at the time of writing[2] to validate our implementation. We aim to achieve similar results as FLOWDROID's existing forward implementation.

5.2.1. Configuration

For the validation, we ran FLOWDROID with the Android module's default configuration using the `EasyTaintWrapper` as the taint wrapper. The configuration summary is 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's 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 call-graph related and the latter is not supported because FLOWDROID has no Android native call handler for now. Also, Inter Component Communication (ICC), Reflection Inter Component Communication and Inter App Communication were left out because the ICC module is - at the time of this work - not maintained anymore. All of the tests stated above are flow function independent. If FLOWDROID gets support for those features in the future, they should also work in backward analysis.

5.2.2. Results

The complete overview of the results is in Table 5.2. ⊛ denotes true positive, ⋆ false positive and ◯ false negative. If a row is empty, the test expects no leaks and also none

---

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

were found.

Our backward-directed implementation yields nearly the same result as the existing forward implementation, with one missed leak more than the baseline. We achieve a F1 measure of 0.89 equally to 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	⊛	⊛
MultiHandlers1		
Ordering1		
RegisterGlobal1	⊛	⊛
RegisterGlobal2	⊛	⊛

App Name	Forwards	Backwards
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	✪	✪
ActivityLifecycle4	✪	✪
ActivitySavedState1	✪	✪
ApplicationLifecycle1	✪	✪
ApplicationLifecycle2	✪	✪

App Name	Forwards	Backwards
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	○	○
StringFormatter1	○	○
StringPatternMatching1	⊛	⊛
StringToCharArray1	⊛	⊛
StringToOutputStream1	⊛ ★	⊛ ★

App Name	Forwards	Backwards
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	⊛	⊛
JavaThread1	⊛	⊛
JavaThread2	⊛	⊛
Looper1	⊛	⊛
TimerTask1	⊛	⊛

App Name	Forwards	Backwards
⍟	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 explain why below.

IMEI1 Implicit flows not implement atm

TODO: implement

General Java

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

StaticInitialization1 differs between forward and backward analysis. Backward reports one leak due to the explicit modeling of `<clinit>` edges instead of relying on SPARK. Recall subsection 4.3.6, leaks inside static blocks are missed in the forward analysis. This test case is quite similar to Figure 4.4b, and therefore, only the backward analysis reports the leak. The Clinit Rule could also be ported to the forward analysis but a larger overapproximation because, unlike backward, there is no guarantee that there will be another edge to `<clinit>` if the statement is in the same class as the `<clinit>` method.

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. Backward, 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 modeling of clinit. The code is provided in Figure 5.1. The `MainActivity` is using the singleton pattern and thus has a static field `v` referring to its instance. The source statement is inside the `Test` class's

static block 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 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`. This is a limitiation of the alias handling.

```java
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 5.1.: StaticInitialization3 code

5.2.4. Improvements From The Summary Taint Wrapper

We briefly explained the simple but not always precise rules of the `EasyTaintWrapper` in

subsection 4.4.1. Using STUBDROID's more precise summaries yields even better results for both directions. The false positives in the test cases `BroadcastReceiverLifecycle2` and `SharedPreferenceChanged1` are gone and the leak in `Serialization1` is found.

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%.

App Name	Forwards		Backwards		Difference			
	#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 enormous 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.

6.1.3. Using A More Precise Taint Wrapper

We noticed the overtainting in `PrivateDataLeak3` is caused by the `EasyTaintWrapper`, thus we now compare how using the `SummaryTaintWrapper` lowers the edge propagations.

As we already described, `PrivateDataLeak3` benefits from the more precise taint wrapper.

App Name	EasyTW #I	EasyTW #A	SummaryTW #I	SummaryTW #A	Difference #I	Difference #A	Difference Total	Difference Relative
AliasingTest								
FlowSensitivity1	39	4	71	13	32	9	41	0.95
Merge1	61	9	109	91	48	82	130	1.86
SimpleAliasing1	20	3	20	3	0	0	0	0.0
StrongUpdate1	11	3	11	3	0	0	0	0.0
AndroidSpecificTest								
ApplicationModeling1	851	1208	427	792	−424	−416	−840	−0.41
DirectLeak1	4	0	4	0	0	0	0	0.0
InactiveActivity	−	−	−	−	−	−	−	−
Library2	6	0	6	0	0	0	0	0.0
LogNoLeak	−	−	−	−	−	−	−	−
Obfuscation1	4	0	4	0	0	0	0	0.0
Parcel1	66	68	87	76	21	8	29	0.22
PrivateDataLeak1	599	835	576	835	−23	0	−23	−0.02
PrivateDataLeak2	5	6	5	6	0	0	0	0.0
PrivateDataLeak3	212	143	41	16	−171	−127	−298	−0.84
runPublicAPIField1	62	31	55	16	−7	−15	−22	−0.24
runPublicAPIField2	11	1	14	1	3	0	3	0.25
runView1	69	0	69	0	0	0	0	0.0
ArrayAndListTest								
ArrayAccess1	51	100	51	100	0	0	0	0.0
ArrayAccess2	12	0	12	0	0	0	0	0.0
ArrayAccess3	51	100	51	100	0	0	0	0.0
ArrayAccess4	42	21	42	21	0	0	0	0.0
ArrayAccess5	67	63	67	63	0	0	0	0.0
ArrayCopy1	9	2	9	2	0	0	0	0.0
ArrayToString1	6	1	6	1	0	0	0	0.0
HashMapAccess1	15	1	15	1	0	0	0	0.0
ListAccess1	60	97	77	118	17	21	38	0.24
MultidimensionalArray1	16	23	16	23	0	0	0	0.0
CallbackTest								
AnonymousClass1	208	1	208	1	0	0	0	0.0
Button1	43	0	43	0	0	0	0	0.0
Button2	155	257	184	275	29	18	47	0.11
Button3	109	408	120	357	11	−51	−40	−0.08
Button4	43	0	43	0	0	0	0	0.0
Button5	6	6	7	7	1	1	2	0.17
LocationLeak1	260	300	286	316	26	16	42	0.07
LocationLeak2	152	2	152	2	0	0	0	0.0
LocationLeak3	104	117	107	117	3	0	3	0.01
MethodOverride1	2	0	2	0	0	0	0	0.0
MultiHandlers1	145	151	148	151	3	0	3	0.01

App Name	EasyTW #I	EasyTW #A	SummaryTW #I	SummaryTW #A	Difference #I	Difference #A	Difference Total	Difference Relative
Ordering1	44	2	44	2	0	0	0	0.0
RegisterGlobal1	49	0	49	0	0	0	0	0.0
RegisterGlobal2	43	0	43	0	0	0	0	0.0
Unregister1	9	1	9	1	0	0	0	0.0
EmulatorDetectionTest								
Battery1	43	15	39	15	−4	0	−4	−0.07
Bluetooth1	4	0	4	0	0	0	0	0.0
Build1	4	0	4	0	0	0	0	0.0
Contacts1	210	19	167	4	−43	−15	−58	−0.25
ContentProvider1	8	0	8	0	0	0	0	0.0
DeviceId1	6	0	6	0	0	0	0	0.0
File1	4	0	4	0	0	0	0	0.0
IMEI1	140	34	111	74	−29	40	11	0.06
IP1	29	1	54	6	25	5	30	1.0
PI1	4	0	4	0	0	0	0	0.0
PlayStore1	8	0	8	0	0	0	0	0.0
PlayStore2	4	0	4	0	0	0	0	0.0
Sensors1	4	0	4	0	0	0	0	0.0
SubscriberId1	4	0	4	0	0	0	0	0.0
VoiceMail1	4	0	4	0	0	0	0	0.0
FieldAndObjectSensitivityTest								
FieldSensitivity1	25	3	25	3	0	0	0	0.0
FieldSensitivity2	19	0	19	0	0	0	0	0.0
FieldSensitivity3	16	0	16	0	0	0	0	0.0
FieldSensitivity4	8	0	8	0	0	0	0	0.0
InheritedObjects1	6	0	6	0	0	0	0	0.0
ObjectSensitivity1	14	1	14	1	0	0	0	0.0
ObjectSensitivity2	10	0	10	0	0	0	0	0.0
GeneralJavaTest								
Clone1	12	4	19	10	7	6	13	0.81
Exceptions1	13	0	13	0	0	0	0	0.0
Exceptions2	13	0	13	0	0	0	0	0.0
Exceptions3	11	0	11	0	0	0	0	0.0
Exceptions4	22	1	53	26	31	25	56	2.43
Exceptions5	16	1	29	6	13	5	18	1.06
Exceptions6	23	0	23	0	0	0	0	0.0
Exceptions7	6	0	6	0	0	0	0	0.0
FactoryMethods1	14	2	14	2	0	0	0	0.0
Loop1	46	7	46	7	0	0	0	0.0
Loop2	74	14	74	14	0	0	0	0.0
Serialization1	22	29	332	547	310	518	828	16.24
SourceCodeSpecific1	45	7	45	7	0	0	0	0.0
StartProcessWithSecret1	17	3	18	4	1	1	2	0.1
StaticInitialization1	9	0	9	0	0	0	0	0.0

App Name	EasyTW		SummaryTW		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
StaticInitialization2	86	0	85	0	−1	0	−1	−0.01
StaticInitialization3	5	0	5	0	0	0	0	0.0
StringFormatter1	10	1	10	1	0	0	0	0.0
StringPatternMatching1	8	6	7	1	−1	−5	−6	−0.43
StringToCharArray1	42	6	42	6	0	0	0	0.0
StringToOutputStream1	30	3	26	3	−4	0	−4	−0.12
UnreachableCode	−	−	−	−	−	−	−	−
VirtualDispatch1	88	28	110	88	22	60	82	0.71
VirtualDispatch2	12	0	12	0	0	0	0	0.0
VirtualDispatch3	6	0	6	0	0	0	0	0.0
VirtualDispatch4	−	−	−	−	−	−	−	−
LifecycleTest								
ActivityEventSequence1	73	0	73	0	0	0	0	0.0
ActivityEventSequence2	77	0	77	0	0	0	0	0.0
ActivityEventSequence3	156	1	156	1	0	0	0	0.0
ActivityLifecycle1	156	7	156	7	0	0	0	0.0
ActivityLifecycle2	33	0	33	0	0	0	0	0.0
ActivityLifecycle3	28	0	28	0	0	0	0	0.0
ActivityLifecycle4	14	0	14	0	0	0	0	0.0
ActivitySavedState1	7	1	7	1	0	0	0	0.0
ApplicationLifecycle1	82	0	82	0	0	0	0	0.0
ApplicationLifecycle2	94	155	94	155	0	0	0	0.0
ApplicationLifecycle3	21	0	21	0	0	0	0	0.0
AsynchronousEventOrdering1	16	0	16	0	0	0	0	0.0
BroadcastReceiverLifecycle1	4	0	4	0	0	0	0	0.0
BroadcastReceiverLifecycle2	248	114	208	98	−40	−16	−56	−0.15
BroadcastReceiverLifecycle3	195	110	144	82	−51	−28	−79	−0.26
EventOrdering1	30	0	30	0	0	0	0	0.0
FragmentLifecycle1	90	0	90	0	0	0	0	0.0
FragmentLifecycle2	−	−	−	−	−	−	−	−
ServiceEventSequence1	152	34	122	38	−30	4	−26	−0.14
ServiceEventSequence2	389	220	315	176	−74	−44	−118	−0.19
ServiceEventSequence3	275	151	232	110	−43	−41	−84	−0.2
ServiceLifecycle1	42	0	42	0	0	0	0	0.0
ServiceLifecycle2	89	21	89	21	0	0	0	0.0
SharedPreferenceChanged1	20	1	8	0	−12	−1	−13	−0.62
ReflectionTest								
Reflection1	8	0	8	0	0	0	0	0.0
Reflection2	11	0	11	0	0	0	0	0.0
Reflection3	62	25	49	0	−13	−25	−38	−0.44
Reflection4	8	0	8	0	0	0	0	0.0
Reflection5	11	0	11	0	0	0	0	0.0
Reflection6	134	51	122	31	−12	−20	−32	−0.17
Reflection7	15	11	3	0	−12	−11	−23	−0.88

App Name	EasyTW		SummaryTW		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
Reflection8	14	0	14	0	0	0	0	0.0
Reflection9	21	0	21	0	0	0	0	0.0
ThreadingTest								
AsyncTask1	11	1	11	1	0	0	0	0.0
Executor1	17	0	17	0	0	0	0	0.0
JavaThread1	17	0	17	0	0	0	0	0.0
JavaThread2	31	8	28	8	−3	0	−3	−0.08
Looper1	20	16	20	16	0	0	0	0.0
TimerTask1	32	33	45	37	13	4	17	0.26
∅ Propagations	61.52	41.2	58.44	41.27	−3.07	0.06	−3.01	0.17
∅ without Serialization1	61.84	41.3	56.22	37.15	−5.62	−4.15	−9.76	0.04

Table 6.2.: DROIDBENCH Evaluation with Summary Taint Wrapper

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 with a distance of time[2]. If there were outliers[3], we repeated the runs.

We also measured the memory usage of both implementations. Using the memory amount reported by the JVM is not precise because the JVM prefers to take up free memory before running the garbage collector [2]. We borrowed the memory evaluation tool from CleanDroid which internally depends on a memory calculation tool from Twitter[4]. The memory evaluation tool measures the size of the exploded supergraph in 15 seconds intervals [3]. Because we do not want to pollute the measured data flow time with the

---

[2]The time distance between each run is at least the elapsed time from the analysis of the remaining 199 apps.

[3]Outliers are runs with at least 15% difference to the median run and a minimum of 5 seconds absolute difference.

[4]https://mvnrepository.com/artifact/com.twitter.common/objectsize

latency of the memory evaluation tool, the memory measuring runs were ran independently of the time measuring runs. The memory sampling also takes up memory and because our test system has enough memory available, we bumped the maximum heap size up to 100GB, effectively eliminating memory timeouts.

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. We set the timeout for the data flow analysis to 10 minutes[5]. The call graph generation was limited to 180 seconds and the call-graphs were serialized before, so every run was on the same call-graph. The configuration summary is in Table 6.3.

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 6.3.: 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 [15]. 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 6.5 and Table 6.4.

We used FlowDroid's forward implementation on the to that date latest upstream commit[6] from the develop branch for the point of comparison. The backward implementation ran on our latest commit with all changes merged from the upstream.

---

[5]A timeout in FlowDroid prevents processing new edges but lets the solver finish the current edge propagation. Thus, some apps may have a data flow time of above 600 seconds.

[6]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

We chose 200 apps randomly out of a Google Playstore dump from 2021 containing over 6000 apps for our evaluation set. Out of 200 apps, 60 apps do not have any sources or sinks and thus, the analysis did not start. For 9 apps, the analysis aborted with errors on at least one run. We are left with 131 apps for which both implementations completed all runs without errors. The full list is appended to this work in **??**.

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 6.5.: Sources for Real World Apps Evaluation

6.2.2. Time Experiment

Metric	Forwards	Backwards
Average Runtime	$518.93s$	$408.97s$
Median Runtime	$600.0s$	$600.0s$
85th Percentile	$605.1s$	$606.2s$
Average Abstractions Infoflow	34555327	13702123
Average Abstractions Alias	12562479	33447121
Average Total Abstractions	47117806	47149244
Memory Timeouts	2.99%	6.77%
Time Timeouts	60.45%	51.88%

Table 6.6.: Results

Metric	Forwards	Backwards
Average Runtime	$364.0s$	$127.33s$
Median Runtime	$596.0s$	$1.0s$
85th Percentile	$599.0s$	$584.8s$
Average Abstractions Infoflow	7744724	2048023
Average Abstractions Alias	2784113	4480848
Average Total Abstractions	10528837	6528871

Table 6.7.: Results without Timeouts

6.2.3. Memory Experiment

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

Basically the answer to RQ2: Can we find a pre-analysis known parameter to decide which analysis is more efficient?
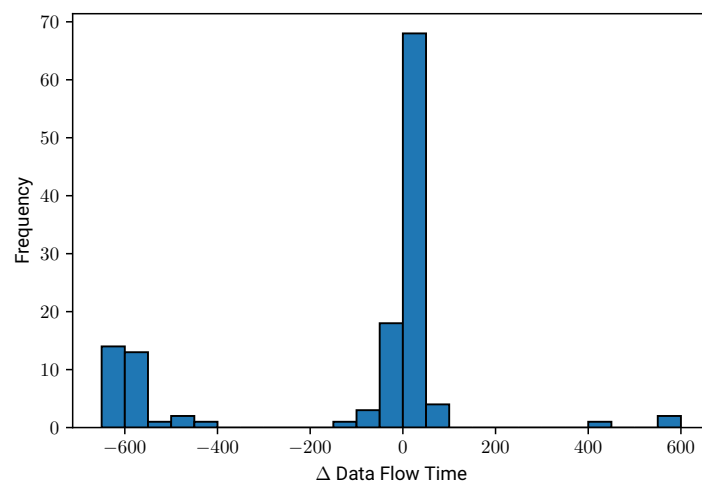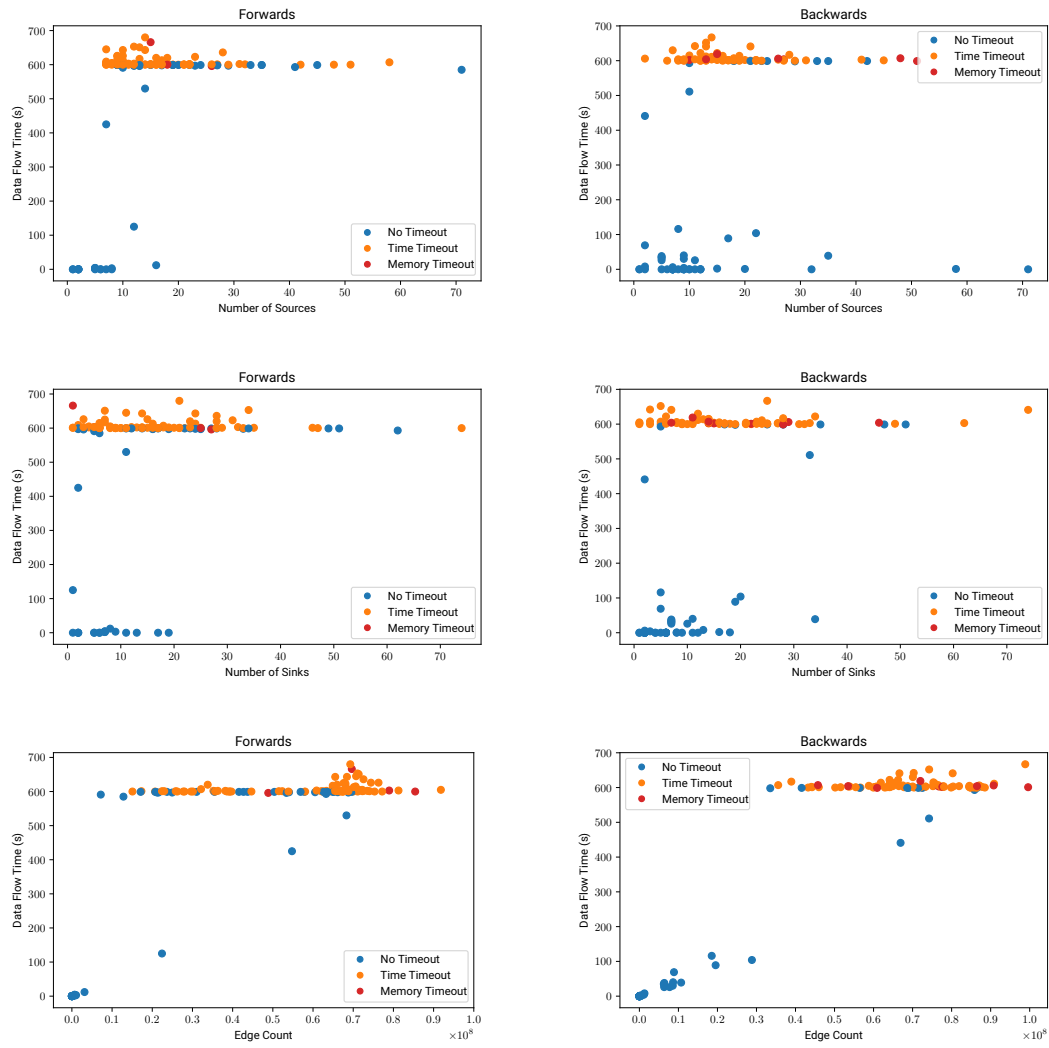
Figure 6.1.: Histogram of the Delta Data Flow Time

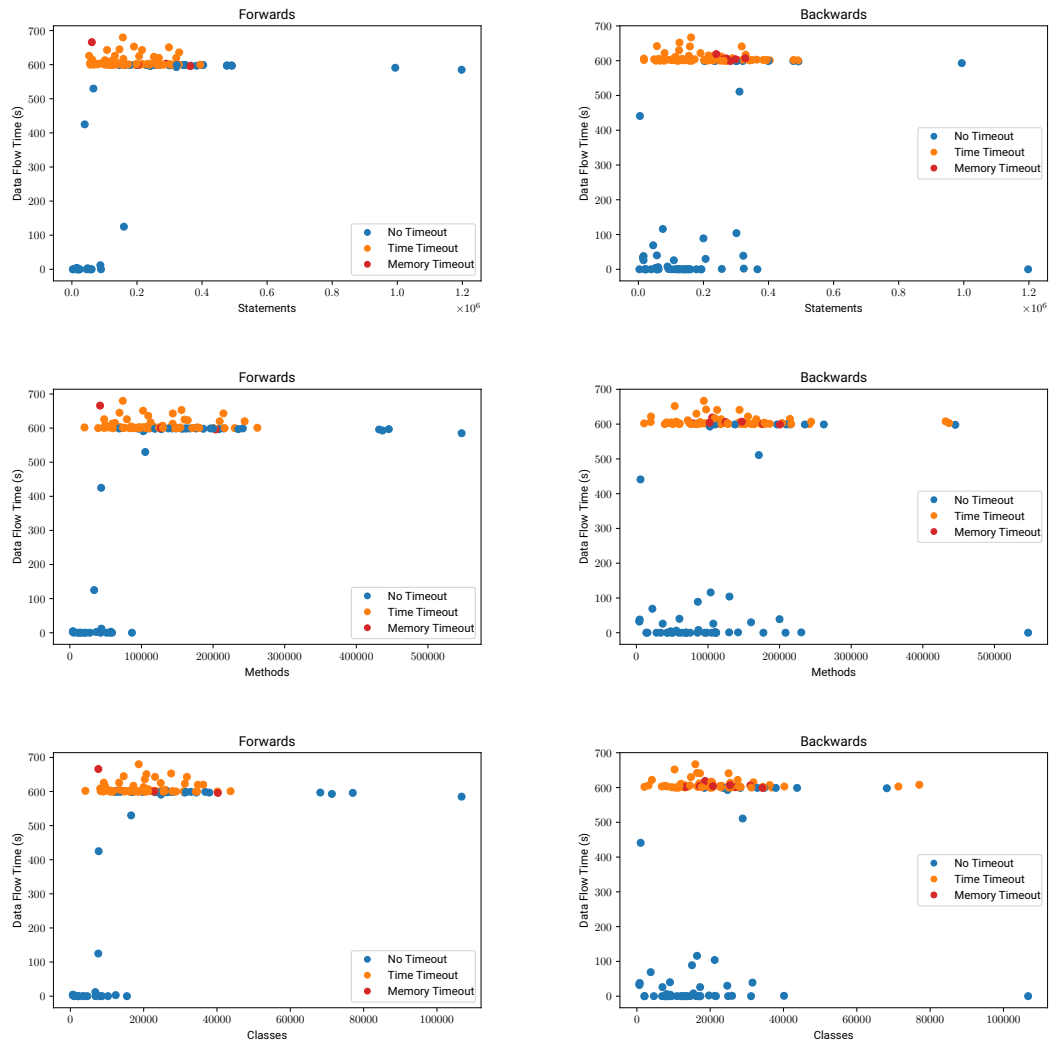Figure 6.2.: Data Flow Time in Comparison to Source, Sink and Edge Count

Figure 6.3.: Data Flow Time in Comparison to Code Size

7. Related Work

While many data flow analyses run backward, we found very little about backward-directed taint analysis in our literature research. Most so-called backward taint analyses are targeted at reconstructing the full path either from a trace log[1][2][3] which is closer to the path builder of FLOWDROID than our taint analysis. Besides those, we also found two taint analysis tools with a similar approach to ours.

Lerch et al.[12] contributed FlowTwist, a static taint analysis tool based on IFDS to detect confused deputy problems[4] in libraries. They identify the cause of such as a combination of an integrity and confidentiality problem. For the integrity part, the sinks perform sensitive operations and the sources are attacker-controlled. In the confidentiality part, an attacker can read the sinks and sources provide sensitive data. A combination of both naturally gives a centered statement. Now, the integrity sources and confidentiality sinks are way more frequent. Thus they propose to solve the integrity part backward and the confidentiality part forward. In contrast to FLOWDROID, FlowTwist focuses on a specific taint analysis case and the applicability is relatively narrow.

Yan et al.[23] 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 backward taint analysis. Instead of relying on nesting the problem in proven data flow frameworks, they seemingly define their own data flow algorithm. The proposed algorithm traverses the basic blocks backward and copies the taints left after traversing a basic block to its predecessors. They do not try to reach a fixpoint; instead, they do not follow circular paths in the control-flow graph. They also

---

[1] `https://recon.cx/2013/slides/Recon2013-Richard%20Johnson-Taint%20Nobody%20Got%20Time%20for%20Crash%20Analysis%20-%20slides.pdf` (visited on 03/26/2021)

[2] `https://blog.trailofbits.com/2019/08/29/reverse-taint-analysis-using-binary-ninja/` (visited on 03/26/2021)

[3] `https://github.com/scotty-kdw/ARM-Analyzer/` (visited on 03/26/2021)

[4] A confused deputy is a legitimate program with more privileges tricked into misusing its authority by a malicious program.

emphasize their concept of "cleans": a predefined list of sanitization methods that 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 backward, which is why we included it as related work, is not provided. Generally speaking, we doubt their tool is precise enough to be useful in practice.

FLOWDROID and also FlowTwist are based on IFDS. Now, IFDS is not the only way to realize taint analysis.

Synchronized pushdown systems (SPDS) by Späth et al.[20] are an alternative to IFDS with access paths for modeling a precise context-, flow- and field-sensitive data flow analysis. Similar to IFDS, a context-free grammar ensures the context-sensitivity. In addition, another context-free grammar model the field-sensitivity. Contrary to access paths, this does not increase the domain and needs no k-limiting to be fast enough in practice. Then it computes the acceptance state of both pushdown automata to combine context- and field-sensitivity. Now, in general, an automaton with two stacks is undecidable. The separation of the problems into two reachability problems and later combining the results is decidable. However, if both automata are in an acceptance state via different paths, the algorithm overapproximates the solution. Their results look promising with a performance close to access paths with $k = 1$. Also, they could not observe the overapproximation in practice when performing typestate analysis.

We did not find any ready-to-use taint analysis utilizing SPDS. Currently in development is SWAN, a taint analysis for the Swift programming language based on SPDS[5].

Doop [6] is a framework initially for pointer analysis. In contrast to others, it uses a declarative approach. Doop's frontend depends on Soot to create facts and encodes them in tables. The analyses are a declarative rule set written in Datalog. These rule sets are then fed into the datalog solver Soufflé[6]. P/Taint [8] extends Doop with a taint analysis. Doop is flow-insensitive and, thus, P/Taint as well.

---

[5] https://github.com/themaplelab/swan
[6] https://souffle-lang.github.io/

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. "Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection". In: *ICSE 2021 Technical Track*. 2021.

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

[5] 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.

[6] M. Bravenboer and Yannis Smaragdakis. "Strictly Declarative Specification of Sophisticated Points-to Analyses". In: vol. 44. Oct. 2009, pp. 243–262. DOI: 10.1145/1640089.1640108.

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

[8] Neville Grech and Yannis Smaragdakis. "P/Taint: unified points-to and taint analysis". In: *Proceedings of the ACM on Programming Languages* 1 (Oct. 2017), pp. 1–28. DOI: 10.1145/3133926.

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

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

[11]   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).

[12]   J. Lerch et al. "FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases". In: *SIGSOFT FSE* (2014). DOI: 10.1145/2635868.2635878.

[13]   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.

[14]   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.

[15]   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.

[16]   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.

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

[18]   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.

[19]   J. Spaeth et al. "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java". In: *ECOOP*. 2016. DOI: 10.4230/LIPIcs.ECOOP.2016.22.

[20]   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.

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

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

[23]   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