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: 17th April 2021

1. Review: Prof. Dr. Michael Waidner
2. Review: Dr. Steven Arzt
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, 17th April 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 runtime behavior, while static analysis works on a representation of the code. Both have different tradeoffs. Achieving good code coverage is a challenge in dynamic analysis. In contrast, a static analysis often can not infer runtime properties, hence follows paths never taken at runtime, also called *infeasible paths* [3]. 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 [12] 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 [12]. 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 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. Values produced from sources are considered tainted. On the other end, *sinks* leak valuable information. A data flow between a source and a sink is called a *leak* [3]. 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 [12, 3]:

- **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 [3].

The analysis direction of a data flow analysis is also decided by the problem to be solved. A live variables analysis computes whether a variable is read before written in the future to pontentialy eliminate dead assignments. The problem is traditionally solved by a backward pass. On the other hand, a reaching definitions analysis finds out if a definition reaches a statement without a intermediate assignment which is certainly solved by a forward pass. Additionally, there are also data flow analyses for which the direction is a design-choice. For example, program slicing identifies a slice, a subset of the programs statements, which influence a statement (backward pass) or are influenced by a statement (forward pass).

Taint analysis on the other hand can be solved in both directions with the same results [12].

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 [21]. 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 [20].

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.

$$
\begin{array}{lcl}
\mathit{matched} & \rightarrow & (_i \ \mathit{matched} \ )_i \ \mathit{matched} \ | \ \epsilon \\
\mathit{valid} & \rightarrow & \mathit{valid} \ (_i \ \mathit{matched} \ | \ \mathit{matched}
\end{array}
$$

Figure 2.1.: Context-Free Grammar proposed by Reps et al.[20]

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* if the sequence of labeled edges is a string in *valid* [20]. 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, only the $)_1$ edge is taken. MOP with this context-free language is also called meet-over-all-valid-paths (MOVP). MOVP increases the precision but does not compromise soundness ($MOVP \sqsubseteq MOP$). Still, the valid paths are an overapproximation because they also contain infeasible paths.

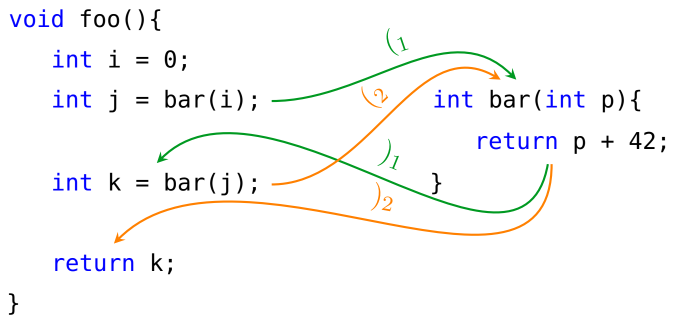


Figure 2.2.: Context-Sensitivity using a Context-Free Language

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: [20]

- **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 [20]. 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 [20].

Eventually, the worklist is empty as 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 [20].

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 [12, 20].

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 [20].

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 [12]. 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.

```java
void main() {
    String x = "";
    while (condition)
        x += "a";
}
```

(a) Code

$$\langle x, \epsilon \rangle \rightarrow \langle x, a \rangle$$
$$\langle x, a \rangle \rightarrow \langle x, aa \rangle$$
$$\langle x, aa \rangle \rightarrow \langle x, aaa \rangle$$
$$\ldots$$

(b) Taint Derivations

Figure 2.3.: Finiteness example

The paper by Reps et al. later decides on set union due to the duality of must and may not [20]. 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 [18].

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 [18]. 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 [7].

Also, the original IFDS definition ignores the type structure of the programming language. Type information 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 [18].

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 returns 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 [16] and it is also present in FlowDroid [3].

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 [22].

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 [12]. 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 [11]. Consider again Figure 2.4. With $k = 2$ the analysis would reach the fixpoint $lst.next.prev.*$ after two iterations.

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 [9]. In practice, Deutsch's approach needs some adaptions as he only considered fields but not base objects and he defines loops simply by type [3]. With symbolic access

```java
while (condition) {
    lst = lst.next.prev;
}
```

(a) Code

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

(b) Access Path Iterations

Figure 2.4.: Infinite Access Path

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 [25]. 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 [29].

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] Source: `https://docs.oracle.com/javase/specs/jvms/se8/html/` (visited on 17.04.2021)

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

(a) Java code

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

(b) Java bytecode

Figure 2.5.: Java bytecode example

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 [29]. 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[6]. 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 [14]. Soot provides the call graph and the conversion from Java and Dalvik bytecode to Jimple, the intermediate representation of choice for FlowDroid [3].

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

[2]Source: `https://github.com/secure-software-engineering/FlowDroid` (visited on 17.04.2021)

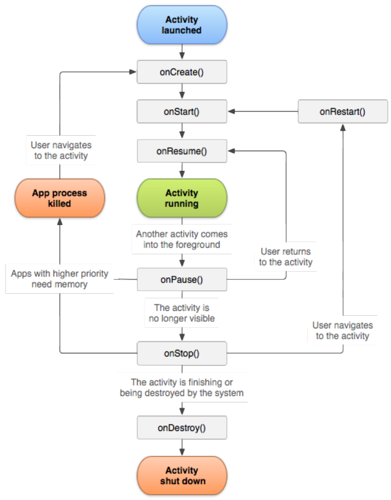


Figure 2.6.: Activity Lifecycle[3]

entire Android lifecycle to be precise and generates a dummy main method to provide a single entry point for the call graph generation. [6].

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 comprises an alais analysis. The alias analysis is encoded as another IFDS problem and resolves all encountered aliases on-demand. The two IFDS analyses are intertwined to maintain flow- and context-sensitivity between both analyses[6].

The implementation of FlowDroid is modular, easily extensible and offers many additional features. Two of them are noteworthy for this work: native call handler and taint wrappers. As both Java and Android allow calling native methods, FlowDroid also needs to model those cases. It currently does not support the analysis of those methods but contains rules for essential methods. The second feature is taint wrappers. They allow defining rules for methods, e.g., from a commonly used feature such as StringBuilder, which allows the taint analysis to skip the method and apply a summary [6]. 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 third-party libraries are often used [5].

[3]Source: `https://developer.android.com/guide/components/activities/activity-lifecycle` (visited on 17.04.2021)

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[3] 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 `AssignStmts` or `IdentityStmts`. 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 `AssignStmts`.

[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` where 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 local or a field dereference. If the expression is a `InvokeExpr`, we handle this in the call flow. For unary operators, the local can simply be extracted and for binary operators, we consider both locals separately. An 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 $\{T\}$.

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

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.

3.1.2. 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 is not able to change the reference in the callee. We know an 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]. [6] 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

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

[6]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.

```java
void main() {
    String str = "42";
    callee(str);
    sink(str); // no leak
}

void callee(String str) {
    str = source();
}
```

(a) Taint Without Fields

```java
void main() {
    SomeObject o = new
        ↪ SomeObject();
    callee(o);
    sink(o.str); // leak
}

void callee(SomeObject o) {
    o.str = source();
}
```

(b) Taint With Fields

Figure 3.1.: Call Flow Example

transforms into a $this$ reference. In Java, `this` references the current instance the method operates on.

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 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, `AssignStmts` 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 left-hand side of the `AssignStmt` 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[7] and map all parameters back into the caller. In contrast to the Call Flow, we also map primitives and strings back into the caller. This is because if a taint reaches the end of the method, as we traverse backward this is the beginning of a method body, the contents of the parameter were copied from the caller into the callee.

> **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. As already written in the corresponding rule 5 of Call flow, the taints is untouched.

> **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.

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

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 independent from a callee 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 $k \geq 0 \wedge (\forall i \in [0, n] \cap \mathbb{N} : a_i \neq x) \wedge x \neq o \wedge x \notin \mathit{StaticVariables}$ produces the outflowing taint set $\{T\}$.

Now, consider again the left side of Figure 3.1. In line 3, the `str` taint is in the kill set of Call Flow because the callee can not be responsible for the tainted variable in the caller. But the taint is still valid after the call. As we want to preserve the taint, 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, the complexity 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, i.e. the exploded supergraph's observed edges. They correlate to the runtime and depend on two factors: taints' lifetime and the number of taints. In the following paragraphs, we show examples where the analysis direction influences both factors. After that, we discuss clues suitable for an apriori decision on the direction.

Concluding

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 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

```java
String returnParam(int i, String s1, String s2, String s3) {
    if (i == 1)
        return s1;
    else if (i == 2)
        return s2;
    else if (i == 3)
        return s3;
    else
        return "default";
}
```

Figure 3.2.: Call Flow Rule 6 Example

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 $retVal \rightarrow \{s1, s2, s3\}$. Forward, 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 a case favors the forward analysis.

In contrast, a strict right-to-left flow favors backward analysis as taints are killed more often due to a stricter 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.
A prominent real-world example of an right-to-left order is Java's `StringBuilder`. The implementations of `append()` and `insert()` return the `this` instance to allow for easy chaining of multiple calls. Consider the corresponding Jimple code in Figure 3.4. Soot does not reuse the local variables when translating JVM bytecode to Jimple[8]. Although all locals from `$stack9` to `$stack15` refer to the same object, the forward analysis can not kill any of those. In a simple view our backward analysis has a huge advantage here. However, when we also consider aliases the advantage is weakened by the aliasing

[8]Locals are reused when converting Dalvik bytecode to Jimple.

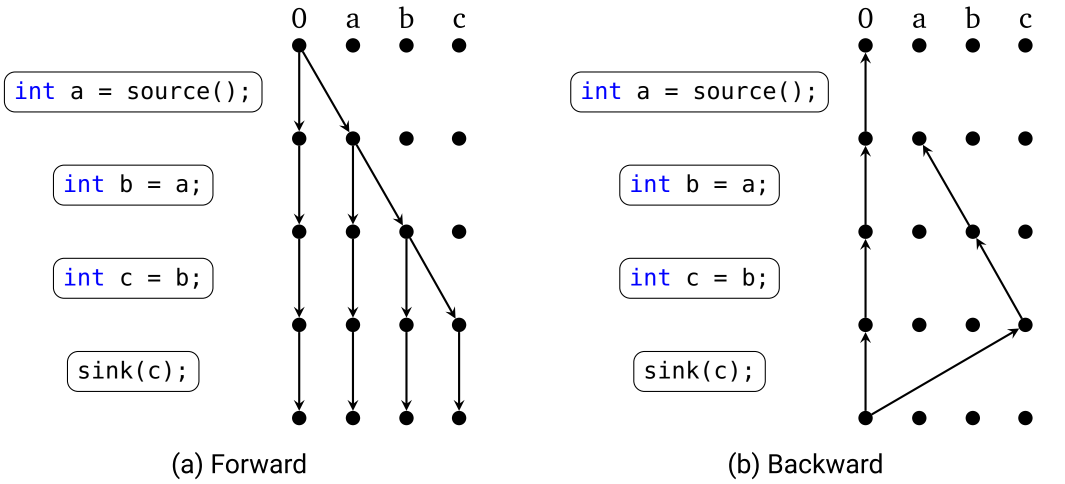


Figure 3.3.: Right-to-Left Order

queries. Nevertheless, because the Java compiler internally transforms non-constant string concatenation into a `StringBuilder`, `StringBuilder` are present in nearly every JVM-based program.

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. This obviously creates many unneccessary edges. FlowDroid already applies an optimization and looks ahead to skip methods in which the static field is not used. Still, the long lifetime stays an issue for real-world application and the direction can not generally change this issue [3].

We discussed the complexity based on the taint propagations. However, they are only known after the analysis. Now, 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. 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[3]. This result indicates that the observed statements have way more influence on the number of taint propagations 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.

```java
StringBuilder sb = new StringBuilder();
sb = sb.append("DeviceID: ").append(id).append("\n IMEI: ")
    .append(imei).append("\n ISMI: ").append(imsi);
```

(a) Java

```
$stack9 = new java.lang.StringBuilder;
specialinvoke $stack9.<java.lang.StringBuilder: void <init>()>();
$stack10 = virtualinvoke $stack9.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>("My device id is ");
$stack11 = virtualinvoke $stack10.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>($stack6);
$stack12 = virtualinvoke $stack11.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>(",my IMEI is ");
$stack13 = virtualinvoke $stack12.<java.lang.StringBuilder: java.lang.StringBuilder append(int)>($stack7);
$stack14 = virtualinvoke $stack13.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>(" and my IMSI is ");
$stack15 = virtualinvoke $stack14.<java.lang.StringBuilder: java.lang.StringBuilder append(int)>($stack8);
```

(b) Jimple

Figure 3.4.: StringBuilder example

Even though the starting taints possibly do not correlate with the runtime, whenever there is a tiny number of sinks but hundreds of sources, the backward analysis should perform better. Likewise, Lerch et al. claim in their work that a magnificent smaller amount of sinks than sources are advantageous for a backward-directed search [15].

The choice on the direction might also be useful for special analysis applications. Think of a case where sanitization methods[9] 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.

[9] 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.

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

(a) Example for alias analysis initiated by rule 1

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

(b) Example for alias analysis initiated by rule 3

Figure 4.1.: Normal flow Aliasing examples

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 [3, 23]. 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 [3].

```java
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

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.

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.

```
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

The turn unit is a new field in the `Abstraction` class, which is representing a taint. A 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 [16]. 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 [3]. 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 [3].

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 .

Additionally, we added an optimization to the taint wrapper rule. Recall section 3.2 where we explained our backward implementation benefits from a right-to-left order. However, the StringBuilder can alias and the alias search offsets the advantage. We use the observation that Jimple does not reuse the local variables when compiled from Java bytecode and all locals pointing to the same StringBuilder except one are not reused either. Thus, we apply a preanalysis searching for base object uses below. Only if the preanalysis finds an use, the alias analysis is used. The preanalysis is cheaper than the alias analysis and based on our observations, it should rarely find an use.

4.3.5. Backwards Implicit Propagation Rule

Flows which are influenced by a condition are called implicit flows. A common example is a password check. Such a method could return a boolean signifying the password is correct or not. Without implicit flows, the taint analysis would be unable to find the path between the password input and the output action. The semantics of implicit flows in FlowDroid are that every update flowing into a sink or a sink call inside a conditional branch, even in transitive callees, are considered as a leak.

The existing forward-directed implementation derives a wildcard and propagates it until the conditional branch is left again at the postdominator. This behavior does not scale well because the semantics demand tainting every update and following every call in conditionals leading to many unneccessary taints never reaching a sink.

```java
1  void foo() {
2      int tainted = source();
3      if (tainted == 42) {
4          x = 0;
5          transitiveSink();
6      }
7  }
8
9  void transitiveSink() {
10     sink();
11 }
```

Figure 4.4.: Implicit Flow Challenges

Backward, the branching factor of a conditional should be lower. However, it is not that easy to reconstruct the conditional branches while traversing the exploded supergraph. Consider the code in Figure 4.4. We expect from the analysis to find the path from line 2 to line 10. First of all, starting at the sink it is unknown whether a conditional influenced the call to the `transitiveSink()` method. Also, assuming we created a taint representing the sink call, consider the return edge into `foo()`. The edge is already inside a conditional branch, thus the dominator is per-definition line 4. We are unable to use the dominator as a indication whether a taint enters a conditional in such cases. Both of those challenges are not a data-flow problem but rather a control-flow graph reachability problem. We extended the backward interprocedural control-flow graph with two methods:

- `List<Unit> getConditionalBranchesInterprocedural(Unit unit)`
- `Unit getConditionalBranchIntraprocedural(Unit unit)`

The first one is used for sink calls. It traverses the interprocedural graph using a worklist algorithm to find all possible reachable conditional statements. We then use those found conditionals to derive sink taints with the correct conditional dominator. The second one is used at return edges and returns the conditional statement if the call site is inside a conditional else `null`.

4.3.6. 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.

4.3.7. 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 4.5. 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].

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.

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

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

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

(a) static variable initialization

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

(b) static block

Figure 4.5.: Examples of statements being in `<clinit>`

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

- **Clinit Rule 1**: If the tainted static variable is a field of the methods class, do not inject because we will at least encounter a `NewExpr` of the same class further in the call graph.
- **Clinit Rule 2**: Else if the tainted static variable matches the `StaticFieldRef` on the right hand side: Inject the edge because we can not be sure whether we see another edge to `<clinit>`.
- **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 4.5a, 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 4.5b.

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

Figure 4.6.: Easy Taint Wrapper Example

4.3.8. 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[3]. 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.

Note that these simple rules are disadvantageous for the backward direction the more parameters a method has. Consider the code snippet in Figure 4.6, 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` and `reflect.Array#newArray`. The handling of `System#arraycopy` is direction-dependent. Thus, we adapted the existing implementation and 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 4.7. 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.

```java
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

```
public static void static2Test() {
    tainted = staticinvoke
        ↪ <soot.jimple.infoflow.test.android.TelephonyManager:
        ↪ java.lang.String getDeviceId()>(); // Line 2 in (a)

    // [...]

    virtualinvoke cm.<soot.jimple.infoflow.test.android.ConnectionManager:
        ↪ void publish(java.lang.String)>(alsoTainted); // Line 9 in (a)

    return;
}
```

(b) Jimple

Figure 4.7.: 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 4.3.7 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.

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 [6]. 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.

Test Case	**Forwards**	**Backwards**
Aliasing		
FlowSensitivity1		
Merge1	$\star$	$\star$
SimpleAliasing1	$\circledast$	$\circledast$
StrongUpdate1		
Arrays and Lists		
ArrayAccess1	$\star$	$\star$
ArrayAccess2	$\star$	$\star$
ArrayAccess3	$\circledast$	$\circledast$
ArrayAccess4		
ArrayAccess5		
ArrayCopy1	$\circledast$	$\circledast$
ArrayToString1	$\circledast$	$\circledast$
HashMapAccess1	$\star$	$\star$
ListAccess1	$\star$	$\star$
MultidimensionalArray1	$\circledast$	$\circledast$
Callbacks		
AnonymousClass1	$\circledast$	$\circledast$
Button1	$\circledast$	$\circledast$
Button2	$\circledast\circledast\circledast$ $\star$	$\circledast\circledast\circledast$ $\star$
Button3	$\circledast\circledast$	$\circledast\circledast$
Button4	$\circledast$	$\circledast$
Button5	$\circledast$	$\circledast$
LocationLeak1	$\circledast\circledast$	$\circledast\circledast$
LocationLeak2	$\circledast\circledast$	$\circledast\circledast$
LocationLeak3	$\circledast$	$\circledast$
MethodOverride1	$\circledast$	$\circledast$
MultiHandlers1		
Ordering1		
RegisterGlobal1	$\circledast$	$\circledast$
RegisterGlobal2	$\circledast$	$\circledast$

Test Case	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	⊛	⊛

Test Case	**Forwards**	**Backwards**
ApplicationLifecycle3	$\circledast$	$\circledast$
AsynchronousEventOrdering1	$\circledast$	$\circledast$
BroadcastReceiverLifecycle1	$\circledast$	$\circledast$
BroadcastReceiverLifecycle2	$\circledast$ $\star$	$\circledast$ $\star$
BroadcastReceiverLifecycle3	$\circledast$	$\circledast$
EventOrdering1	$\circledast$	$\circledast$
FragmentLifecycle1	$\circledast$	$\circledast$
FragmentLifecycle2	$\bigcirc$	$\bigcirc$
ServiceEventSequence1	$\bigcirc$	$\bigcirc$
ServiceEventSequence2	$\bigcirc$	$\bigcirc$
ServiceEventSequence3	$\bigcirc$	$\bigcirc$
ServiceLifecycle1	$\circledast$	$\circledast$
ServiceLifecycle2	$\circledast$	$\circledast$
SharedPreferenceChanged1	$\circledast$ $\star$	$\circledast$ $\star$
General Java		
Clone1	$\circledast$	$\circledast$
Exceptions1	$\circledast$	$\circledast$
Exceptions2	$\circledast$	$\circledast$
Exceptions3	$\star$	$\star$
Exceptions4	$\circledast$	$\circledast$
Exceptions5	$\circledast$	$\circledast$
Exceptions6	$\circledast$	$\circledast$
Exceptions7		
FactoryMethods1	$\circledast\circledast$	$\circledast\circledast$
Loop1	$\circledast$	$\circledast$
Loop2	$\circledast$	$\circledast$
Serialization1	$\bigcirc$	$\bigcirc$
SourceCodeSpecific1	$\circledast$	$\circledast$
StartProcessWithSecret1	$\circledast$	$\circledast$
StaticInitialization1	$\bigcirc$	$\circledast$
StaticInitialization2	$\circledast$	$\circledast$
StaticInitialization3	$\bigcirc$	$\bigcirc$
StringFormatter1	$\bigcirc$	$\bigcirc$
StringPatternMatching1	$\circledast$	$\circledast$
StringToCharArray1	$\circledast$	$\circledast$
StringToOutputStream1	$\circledast$ $\star$	$\circledast$ $\star$

Test Case	Forwards	Backwards
UnreachableCode		
VirtualDispatch1	⊛ ⋆	⊛ ⋆
VirtualDispatch2	⊛ ⋆	⊛ ⋆
VirtualDispatch3	⋆	⋆
VirtualDispatch4		
Implicit Flows		
ImplicitFlow1	⊛	⊛
ImplicitFlow2	⊛⊛	⊛⊛
ImplicitFlow3	⊛⊛	⊛⊛
ImplicitFlow4	○	○
ImplicitFlow5		
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		

Test Case	Forwards	Backwards
AsyncTask1	⊛	⊛
Executor1	⊛	⊛
JavaThread1	⊛	⊛
JavaThread2	⊛	⊛
Looper1	⊛	⊛
TimerTask1	⊛	⊛
⊛	108	109
⋆	14	14
◯	13	12
Precision	88.52%	88.62%
Recall	89.26%	90.08%
F1 measure	0.89	0.89

Table 5.2.: DroidBench Validation Results

5.2.3. Results Explanation

The analyses only differ in `StaticTests#StaticInitialization1` where we do not miss the leak. 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 4.3.7, leaks inside static blocks are missed in the forward analysis. This test case is quite similar to Figure 4.5b, 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 4.3.7. 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.

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.

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

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

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

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

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

Figure 5.1.: StaticInitialization3 code

6. Performance Evaluation

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

6.1. DroidBench

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

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

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

The configuration is the same as described in subsection 5.2.1.

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

6.1.1. Results

The full results are listed in Table 6.1. When rows only contain hyphens, 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 with the existing implementation as the reference: $Result_B - Result_F$. The relative difference is calculated similar: $\frac{TotalDifference}{|\#I_F + \#A_F|}$. Hence, negative values signify the backward analysis performed better.

On average, our implementation needs more edge propagations to finish the analysis. Even though for explicit flows the backward analysis needs less propagations in the infoflow analysis, it then suffers from more encountered aliases. If we look at it on a per test basis, 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. However, we did not expect cases that let the edge propagations of our implementation explode up to a factor of 100, as seen in `LifecycleTest#BroadcastReceiverLifecycle3`. In contrast, the existing forward implementation only at most a relative difference of -0.95.

Test Case	Forwards		Backwards		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
AliasingTest								
FlowSensitivity1	175	72	39	4	-136	-68	-204	-0.83
Merge1	137	65	89	47	-48	-18	-66	-0.33
SimpleAliasing1	35	13	20	3	-15	-10	-25	-0.52
StrongUpdate1	30	13	11	3	-19	-10	-29	-0.67
AndroidSpecificTest								
ApplicationModeling1	212	96	851	1208	639	1112	1751	5.69
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	86	93	-58	78	20	0.13
PrivateDataLeak1	410	110	608	766	198	656	854	1.64
PrivateDataLeak2	15	0	5	3	-10	3	-7	-0.47
PrivateDataLeak3	17	2	210	140	193	138	331	17.42
runPublicAPIField1	89	1	43	0	-46	-1	-47	-0.52
runPublicAPIField2	5	0	11	0	6	0	6	1.2
runView1	71	50	69	0	-2	-50	-52	-0.43
ArrayAndListTest								
ArrayAccess1	77	34	51	100	-26	66	40	0.36
ArrayAccess2	16	4	12	0	-4	-4	-8	-0.4

Test Case	Forwards		Backwards		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
ArrayAccess3	77	34	51	100	−26	66	40	0.36
ArrayAccess4	164	84	42	21	−122	−63	−185	−0.75
ArrayAccess5	75	5	34	23	−41	18	−23	−0.29
ArrayCopy1	18	2	9	2	−9	0	−9	−0.45
ArrayToString1	10	1	6	0	−4	−1	−5	−0.45
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	0	56	0	56	0.37
Button1	58	39	43	0	−15	−39	−54	−0.56
Button2	444	66	155	254	−289	188	−101	−0.2
Button3	360	89	109	408	−251	319	68	0.15
Button4	58	39	43	0	−15	−39	−54	−0.56
Button5	80	40	6	3	−74	−37	−111	−0.93
LocationLeak1	617	222	260	298	−357	76	−281	−0.33
LocationLeak2	212	121	152	0	−60	−121	−181	−0.54
LocationLeak3	220	73	104	115	−116	42	−74	−0.25
MethodOverride1	3	0	2	0	−1	0	−1	−0.33
MultiHandlers1	17	0	145	149	128	149	277	16.29
Ordering1	456	151	44	0	−412	−151	−563	−0.93
RegisterGlobal1	291	162	49	0	−242	−162	−404	−0.89
RegisterGlobal2	52	37	43	0	−9	−37	−46	−0.52
Unregister1	11	0	9	0	−2	0	−2	−0.18
EmulatorDetectionTest								
Battery1	7	0	39	15	32	15	47	6.71
Bluetooth1	4	0	4	0	0	0	0	0.0
Build1	4	0	4	0	0	0	0	0.0
Contacts1	53	0	200	19	147	19	166	3.13
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	137	0	422	1	285	1	286	2.09
IP1	4	0	29	0	25	0	25	6.25
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
FieldSensitivity2	35	15	19	0	−16	−15	−31	−0.62
FieldSensitivity3	38	15	16	0	−22	−15	−37	−0.7

Test Case	Forwards		Backwards		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
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	21	1	22	0	1	−1	0	0.0
Exceptions5	13	1	16	0	3	−1	2	0.14
Exceptions6	78	12	23	0	−55	−12	−67	−0.74
Exceptions7	71	12	6	0	−65	−12	−77	−0.93
FactoryMethods1	40	0	14	0	−26	0	−26	−0.65
Loop1	93	2	51	0	−42	−2	−44	−0.46
Loop2	123	2	79	0	−44	−2	−46	−0.37
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	26	27	9	0	−17	−27	−44	−0.83
StaticInitialization2	57	29	86	0	29	−29	0	0.0
StaticInitialization3	35	9	5	0	−30	−9	−39	−0.89
StringFormatter1	16	1	10	0	−6	−1	−7	−0.41
StringPatternMatching1	23	1	8	4	−15	3	−12	−0.5
StringToCharArray1	91	4	47	0	−44	−4	−48	−0.51
StringToOutputStream1	26	3	25	1	−1	−2	−3	−0.1
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	–	–	–	–	–	–	–	–
ImplicitFlowTest								
ImplicitFlow1	1823	144	3315	11	1492	−133	1359	0.69
ImplicitFlow2	146	63	991	3	845	−60	785	3.76
ImplicitFlow3	148	50	1023	20	875	−30	845	4.27
ImplicitFlow4	67	0	1864	12	1797	12	1809	27.0
ImplicitFlow6	18	0	112	0	94	0	94	5.22
LifecycleTest								
ActivityEventSequence1	58	35	72	0	14	−35	−21	−0.23
ActivityEventSequence2	32	24	77	0	45	−24	21	0.38
ActivityEventSequence3	209	116	156	0	−53	−116	−169	−0.52
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

Test Case	Forwards		Backwards		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
ActivityLifecycle4	49	33	14	0	−35	−33	−68	−0.83
ActivitySavedState1	20	0	7	0	−13	0	−13	−0.65
ApplicationLifecycle1	37	10	82	0	45	−10	35	0.74
ApplicationLifecycle2	86	17	94	155	8	138	146	1.42
ApplicationLifecycle3	32	12	21	0	−11	−12	−23	−0.52
AsynchronousEventOrdering1	58	31	16	0	−42	−31	−73	−0.82
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	124	34	71	14	85	1.16
ServiceEventSequence2	105	49	389	220	284	171	455	2.95
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	11	0	−2	0	−2	−0.15
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	85.46	23.41	117.64	38.6	32.19	15.19	47.37	1.61
∅ without Implicit Flow	70.61	22.46	60.56	40.1	−10.05	17.63	7.59	1.34

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 an 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 possible callees in the interprocedural control-flow graph. Backward all of these callees are visited, of which only one contains a source statement. Forward, the taint is introduced in the callee at the source and just one return site needs to be processed.

A Note On Implicit Flows All implicit flow tests and the IMEI1 test need the implicit flow rule to find the leaks. In those test cases our implementation does not stand a chance. We especially want to highlight the "every sink call influenced by conditional" semantics here. This semantic forces us to derive an empty taint for every conditional that is theoretically reachable from a sink. Beyond, we also taint the base object without any fields at every sink to detect a possible conditional object instantiation. Even in simple test cases such as ImplicitFlow4 this results in 10 additional taints per sink. Important to note is also that the prior computation of reachable conditionals is not represented in the edge propagations. We thus conclude that it is probably better to live without a backward-directed implicit data flow analysis.

6.1.3. Using A More Precise Taint Wrapper

We noticed the overtainting in `PrivateDataLeak3` is caused by the `EasyTaintWrapper`. Thus, we now look how using the `SummaryTaintWrapper` influences the edge propagations. The full results are in Table 6.2. In the table, we take the `EasyTaintWrapper` as the reference and compare it against the `SummaryTaintWrapper` on our implementation. The structure of the table is as in the last subsection.

As we already described, `PrivateDataLeak3` benefits from the more precise taint wrapper. Similarily, many other test cases also benefit. Others, especially Serialization1 have more edge propagations because the `SummaryTaintWrapper` has a summary for a method which the `EasyTaintWrapper` does not handle[2] resulting in a premature kill of a taint. Even with Serialization1 included in the average, the `SummaryTaintWrapper` needs less total edge propagations. Excluding it also equals out the relative difference. Altogether, the `SummaryTaintWrapper` should be the default choice for real-world applications because it is more precise without compromising on the edge propagations.

Test Case	EasyTW		SummaryTW		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
AliasingTest								
FlowSensitivity1	39	4	71	13	32	9	41	0.95
Merge1	89	47	109	91	20	44	64	0.47
SimpleAliasing1	20	3	20	3	0	0	0	0.0
StrongUpdate1	11	3	11	3	0	0	0	0.0
AndroidSpecificTest								

[2]The `EasyTaintWrapper` contains a list of supported classes. Every method from those classes is excluded from the analysis, regardless of the method being in the list of the handled methods.

Test Case	EasyTW		SummaryTW		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
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	86	93	87	76	1	−17	−16	−0.09
PrivateDataLeak1	608	766	585	766	−23	0	−23	−0.02
PrivateDataLeak2	5	3	5	3	0	0	0	0.0
PrivateDataLeak3	210	140	38	12	−172	−128	−300	−0.86
runPublicAPIField1	43	0	36	0	−7	0	−7	−0.16
runPublicAPIField2	11	0	14	0	3	0	3	0.27
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	34	23	34	23	0	0	0	0.0
ArrayCopy1	9	2	9	2	0	0	0	0.0
ArrayToString1	6	0	6	0	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	0	208	0	0	0	0	0.0
Button1	43	0	43	0	0	0	0	0.0
Button2	155	254	184	272	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	3	7	3	1	0	1	0.11
LocationLeak1	260	298	286	314	26	16	42	0.08
LocationLeak2	152	0	152	0	0	0	0	0.0
LocationLeak3	104	115	107	115	3	0	3	0.01
MethodOverride1	2	0	2	0	0	0	0	0.0
MultiHandlers1	145	149	148	149	3	0	3	0.01
Ordering1	44	0	44	0	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	0	9	0	0	0	0	0.0
EmulatorDetectionTest								
Battery1	39	15	39	15	0	0	0	0.0
Bluetooth1	4	0	4	0	0	0	0	0.0
Build1	4	0	4	0	0	0	0	0.0

Test Case	EasyTW		SummaryTW		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
Contacts1	200	19	167	4	−33	−15	−48	−0.22
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
IP1	29	0	52	0	23	0	23	0.79
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	0	22	0	0	0	0	0.0
Exceptions5	16	0	16	0	0	0	0	0.0
Exceptions6	23	0	23	0	0	0	0	0.0
Exceptions7	6	0	6	0	0	0	0	0.0
FactoryMethods1	14	0	14	0	0	0	0	0.0
Loop1	51	0	47	0	−4	0	−4	−0.08
Loop2	79	0	75	0	−4	0	−4	−0.05
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
StaticInitialization2	86	0	86	0	0	0	0	0.0
StaticInitialization3	5	0	5	0	0	0	0	0.0
StringFormatter1	10	0	10	0	0	0	0	0.0
StringPatternMatching1	8	4	7	0	−1	−4	−5	−0.42
StringToCharArray1	47	0	43	0	−4	0	−4	−0.09
StringToOutputStream1	25	1	24	1	−1	0	−1	−0.04
UnreachableCode	–	–	–	–	–	–	–	–
VirtualDispatch1	88	28	110	88	22	60	82	0.71
VirtualDispatch2	12	0	12	0	0	0	0	0.0

Test Case	EasyTW		SummaryTW		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
VirtualDispatch3	6	0	6	0	0	0	0	0.0
VirtualDispatch4	–	–	–	–	–	–	–	–
LifecycleTest								
ActivityEventSequence1	72	0	73	0	1	0	1	0.01
ActivityEventSequence2	77	0	77	0	0	0	0	0.0
ActivityEventSequence3	156	0	156	0	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	0	7	0	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	124	34	122	38	−2	4	2	0.01
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	11	0	8	0	−3	0	−3	−0.27
ReflectionTest								
Reflection1	8	0	8	0	0	0	0	0.0
Reflection2	11	0	11	0	0	0	0	0.0
Reflection3	62	25	50	0	−12	−25	−37	−0.43
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
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

Test Case	EasyTW		SummaryTW		Difference			
	#I	#A	#I	#A	#I	#A	Total	Relative
∅ Propagations	60.56	40.1	57.29	39.16	−3.27	−0.93	−4.2	0.13
∅ without Serialization1	60.88	40.19	55.04	35.0	−5.84	−5.19	−11.02	0.0

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[3]. If there were outliers[4], we repeated the run. [5] Some runs did not comply to our outlier norm even after we ran them multiple times, but this only concers 7 runs being well below the 2% mark.

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 [3]. We borrowed the memory evaluation tool from CleanDroid, which internally depends on a memory calculation tool from Twitter[6]. The memory evaluation tool measures the size of the exploded supergraph in 15 seconds intervals [4]. Because we do not want to pollute the measured data flow time with the memory evaluation tool's latency, the memory measuring runs were run 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.

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

[5]Outliers are runs with at least 25% difference to the median run and a minimum of 5 seconds absolute difference.

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

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

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[7]. 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.

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, to find sources and sinks in the Android framework automatically [19]. 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 6.4 and Table 6.5.

We used FlowDroid's forward implementation on the to that date latest upstream commit[8] from the develop branch for the point of comparison. The backward implementation ran on our latest commit[9] at that time with all changes from the upstream merged into.

We chose 200 apps randomly out of a Google Playstore dump from 2021 containing over

[7]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.

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

[9]Our latest commit was `87bf33ba40ef8b4fb25f33439d887ebc98c2c184`. Note that during the real-world evaluation we found some bugs and also later on fixed some edge cases in the analysis. All fixes should not influence the runtime in a bad way.

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 6.4.: Sources for Real World Apps Evaluation

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 6.5.: Sinks for Real World Apps Evaluation

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 six apps, the analysis aborted with errors on at least one run. All thrown exceptions happened outside of FlowDroid. We are left with 131 apps for which both implementations completed all runs without errors. The full list is appended to this work in Table A.1.

6.2.2. Time Evaluation

In general, the individual apps' runtimes were far apart from each other. We had many apps with a single-digit analysis time and on the other side, we also found many apps that triggered a timeout or were close to triggering one. In between those extrema are only a few apps. Recall that we set a soft limit on the runtime at 600 seconds. The reference forward runs have a standard deviation of $209s$ and the runs of our implementation has $277s$ standard deviation. It is important to keep this in mind when interpreting the results.

We first begin with an overview of the results. Table 6.6 shows the results, including timeouts. Notably, the backward analysis had 8% less time timeouts than the forward analysis. In return, it seems a bit more memory-hungry with 3.63% more memory timeouts. We conducted a t-test to check the significance of those differences with the null hypothesis of equal average expected values. The p-value for the memory timeouts is 0.156, thus being insignificant. A t-test over the runtime yielded a p-value of 0.00036, meaning the advantage for our implementation is significant. We cover the memory consumption extensively in

Metric	**Forward** Avg	Median	$\mathbf{P_{85}}$
Data Flow Time	$518.93s$	$600.00s$	$605.10s$
Edge Propagations Infoflow	34555326.97	41743088.00	52163969.60
Edge Propagations Alias	12562479.07	14598571.50	18638900.10
Total Edge Propagations	47117806.04	57697027.00	70602469.30
Memory Timeouts			2.99%
Time Timeouts			60.45%

Metric	**Backward** Avg	Median	$\mathbf{P_{85}}$
Data Flow Time	$413.19s$	$600.00s$	$606.00s$
Edge Propagations Infoflow	13826566.90	14981108.50	23712802.00
Edge Propagations Alias	33567561.46	43444060.00	56773141.00
Total Edge Propagations	47394128.36	60855935.50	79405729.00
Memory Timeouts			6.62%
Time Timeouts			52.21%

Table 6.6.: Results With Timeouts

the next subsection and focus on the time for now. Interestingly, the propagated edges along the same interprocedural call-graph are of the same order of magnitude. Also, the 85^{th} percentile runtime is nearly equal and the median is equal. However, claims based on the runtime and edges with timeouts are only possible to a limited extent because the timeout highly influences both values.

Next, we only consider the runs without any timeouts in Table 6.7. This time we can still observe a relation between backward infoflow edges and forward alias edges even though to a lesser extent. More significant, backward needed way less forward propagations either because fewer aliases were on the path or the alias analysis could be stopped earlier due to a near turn unit. The runtimes also represent this fact. In the 85^{th} percentile, both analyses are more close than the average suggest, with the backward analysis needing 2.25 seconds less. The median here renders useless as a comparison point because of the huge variance in the data set.

A knowledgeable reader might have noticed the results in Table 6.6 are worse than in previous publications where FlowDroid was evaluated [3, 4]. We want to emphasize that none of our changes did influence the reference runs in a bad way as we used the

Metric	Forward Avg	Forward Median	Forward P_{85}
Data Flow Time	$364.00s$	$596.00s$	$599.00s$
Edge Propagations Infoflow	21179450.04	17131840.00	47411443.20
Edge Propagations Alias	7613696.10	6557951.00	16530488.80
Total Edge Propagations	28793146.14	22416842.00	63123292.80

Metric	Backward Avg	Backward Median	Backward P_{85}
Data Flow Time	$135.75s$	$1.50s$	$596.75s$
Edge Propagations Infoflow	5186970.23	192787.00	14438463.25
Edge Propagations Alias	11459343.68	258834.50	33860441.50
Total Edge Propagations	16646313.91	451621.50	62000571.75

Table 6.7.: Results without Timeouts

upstream version without a single line changed to conduct the first run[10]. The existing implementation suffers as ours, so we suspect it partly depends on an unfortunatly drawn app set and further development in the call-graph generation leading to more possible edges.

With that out of the way, let us look at the results in greater detail. We now compare the analysis on a per-app basis. The histogram is in Figure 6.1. We compiled the delta data flow time of the analyses per app, calculated as in the last section with the forward implementation being the reference: $t_{Backward} - t_{Forward}$. Hence, negative values represent that our implementation performed better. The delta on the x-axis is given in seconds and the frequency on the y-axis in number of apps. The bins always span over 50 seconds. The graph shows a large number of apps around 0 with a slight bias towards the forward implementation. Equivalent to the distribution of the data flow times, there are only few deltas in the range from ± 100 to ± 500. More interestingly, there are significantly more apps around -600 than around 600. Recall, the timeout is set to $600s$. So, our implementation terminates nearly instantaneous in some cases on which the forward analysis times out. As expected, there is no general advantage for a direction. Instead, we observe a per-app advantage in around 60% of the test set, while for the rest, the performance is similar.

[10] We found some exceptions in the upstream project while evaluating, so after the first run we switched to our version with the fixes included. This did not change the results we got.

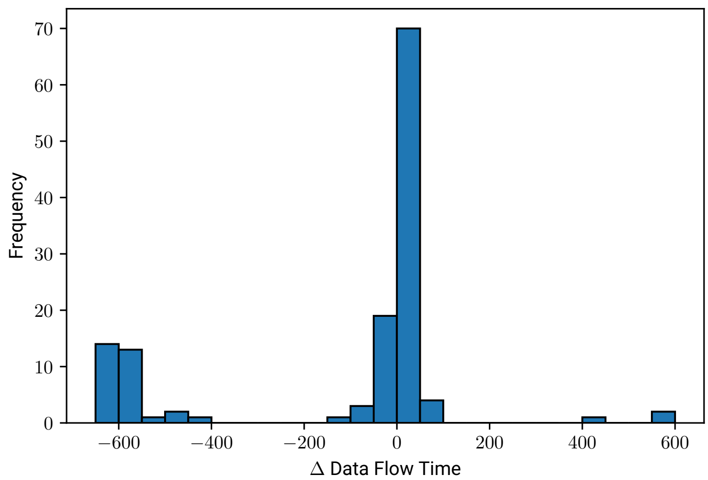
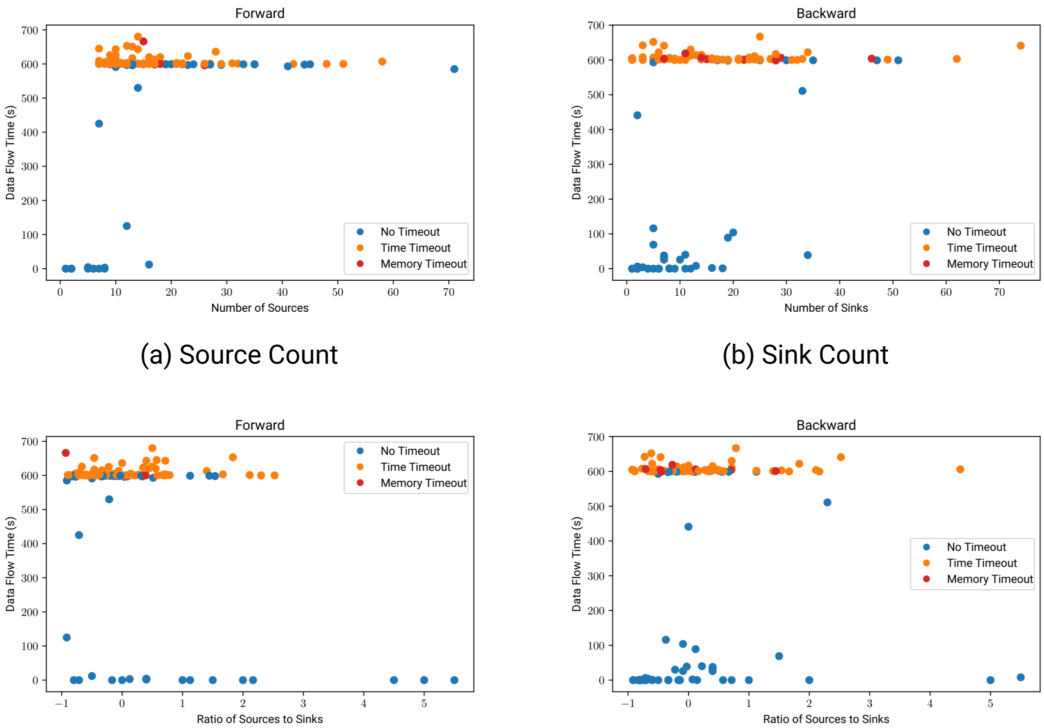


Figure 6.1.: Histogram of the Delta Data Flow Time

We confirmed that the right direction choice can speed up the analysis by a magnificent amount. To take advantage of the favorable direction, we now investigate the correlating conditions for the advantageous direction. Most straightforward would be a correlation between the difference of source and sink count and the data flow time. In Figure 6.2c are two graphs with the ratio of sources and sinks ($\frac{Sinks-Sources}{Sources}$) on the x-axis and the data flow time in seconds on the y-axis. The left graph is always the forward implementation and the right graph is our implementation. Blue dots represent apps without a timeout, orange a time timeout and red a memory timeout. Intuitively, a negative ratio should put our implementation at an advantage. The graphs show no correlation between the ratio and the runtime, neither forward nor backward. We also included the forward data flow time by sources and the backward data flow time by sinks in Figure 6.2a and Figure 6.2b. The number of sinks backward and the number of sources forward do not influence the runtime. So we can confirm Arzt's evaluation[3] as there is no correlation between sources and the forward runtime in our app set. Parallel to this observation, the sink count does not influence the backward runtime. The sink count for forward and the source count for backward can not influence the runtime they have no influence on the edge propagations.

Even though Arzt's evaluation also showed no correlation between the code size [3], we do for completeness also compare the runtime to the number of statements, methods and classes. Note that these refer to the Jimple intermediate representation and not Java.



(a) Source Count

(b) Sink Count

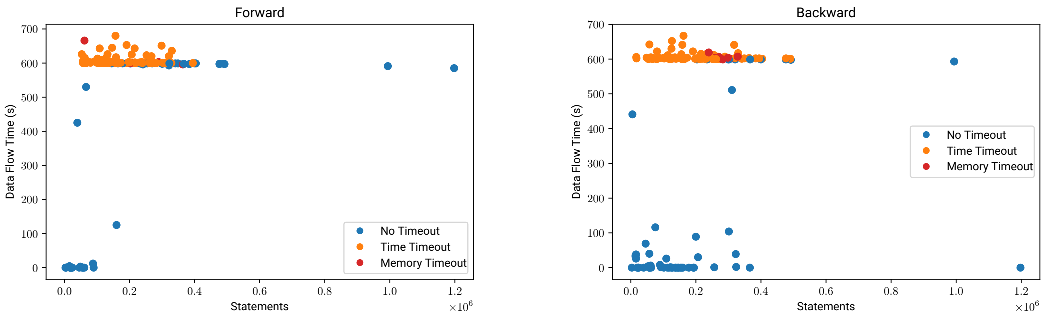
(c) Ratio

Figure 6.2.: Data Flow Time in Comparison to Sources, Sinks and the Ratio of Those

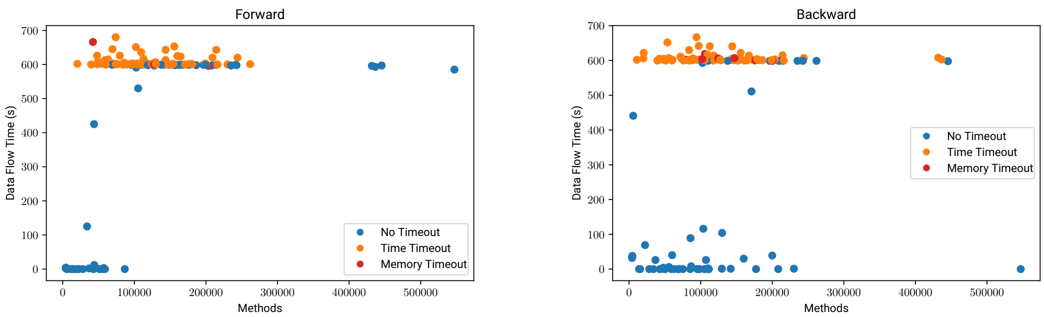
Figure 6.3 includes all graphs with the existing implementation being on the left side and our implementation on the right side. The notation are the same as before, with the x-axis swapped out. The number of statements is uniformly distributed with some outliers. If we consider our data as two linear data sets with a structural break between the two groups, the linear regressions have a slope of close to 0. Resulting, the number of statements, methods and classes do not have an impact on the runtime.

Because the above mentioned parameters do not influence the runtime, we did further investigate to find a parameter to decide the favorable direction. First, we looked the methods containing sources and sinks. We counted the number of statements of the method, call statements and callers of the method and compared these numbers between sources and sinks. A advantage in those did not result in a faster analysis. Next, we implemented a fast intraprocedural taint analysis. It omits access paths and aliasing. Method calls are overapproximated in a similar fashion to the `EasyTaintWrapper`. We then counted the taints flowing into the callees and callers. Also, we did count the number of taints in the method. The drawback is that this only works when the state explosion happens inside the first method and this is not the case in the app set. Again, we could not find any resilient correlation. At this point, we run out of easily computable facts about an app that could correlate with the runtime and decided to leave the question up for future work.

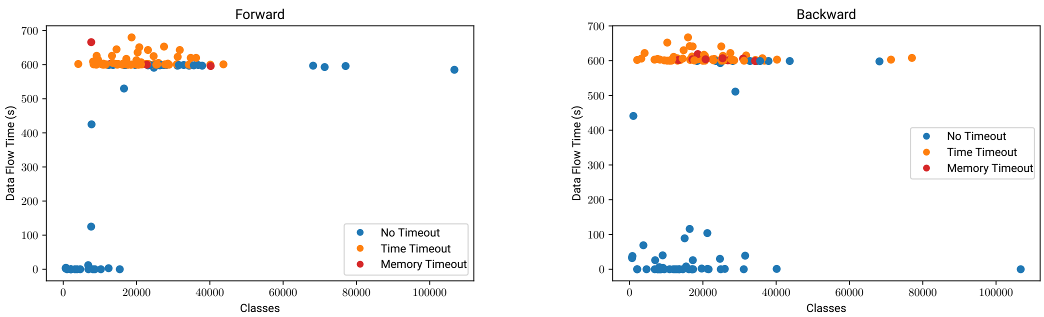
Finally, we compare the number of edges in the exploded supergraph, referred to as taint propagations in section 3.2. Note that the edges in the exploded supergraph are only known after the analysis, making them useless for predictions. In Figure 6.4a we plot the edge count on the x-axis to the data flow time on the y-axis. In both graphs, we observe a linear correlation for the apps with a runtime below 500 seconds. Then there is a structural break and after that the apps time out. Because a linear regression does not really fit well for our diverse data set, we decided to fit a function using the least squares method. We achieved a r^2 measure of greater than 0.9 for four degree polynomials and above. However, the good fitting curve seems overfitted to us because the apps not being close to timeouts have a good fitting linear correlation. When we look at the point where the structural break happens, we notice that backward the timeouts start after roughly $3 \cdot 10^7$ propagations. Forward on the other hand only gets to around $2 \cdot 10^7$ edge propagations before reaching a timeout. Such a large difference is unintuitive because the computation cost should not be much different. We split the edges up by IFDS problems in Figure 6.4b and Figure 6.4c. Interestingly, all curves have a similar steep curve with the exception of the backward alias analysis being more shallow. This gives a possible explanation linked to the ratio of infoflow and alias edges. The alias flow functions are way simpler and thus, should also cost less to compute. The backward analysis has a ratio



(a) Statements



(b) Methods

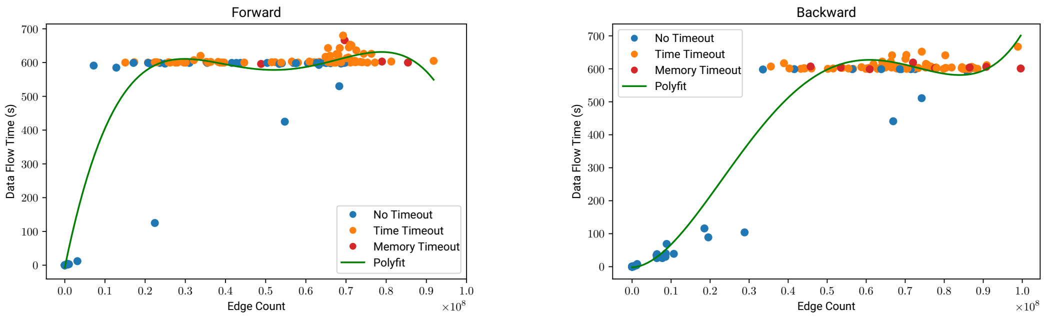


(c) Classes

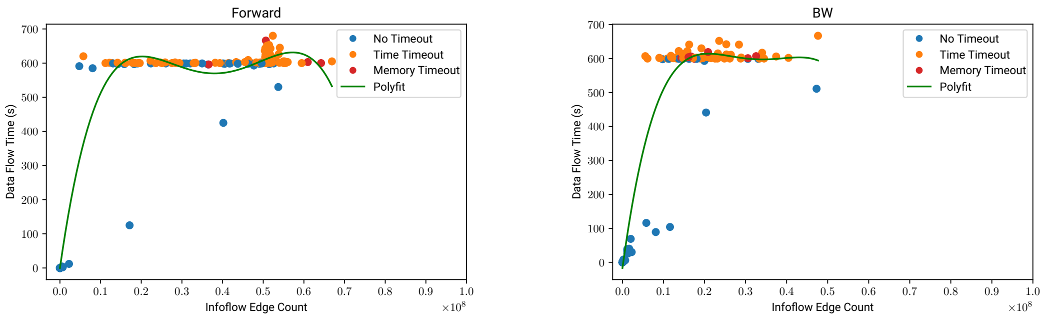
Figure 6.3.: Data Flow Time in Comparison to Code Size

biased toward the alias edges which could explain the higher edge count possible in ten minutes. Why the structural break happens could not be conclusively clarified in this work, so it is also hard to finally reason whether this also holds without such a structural break.

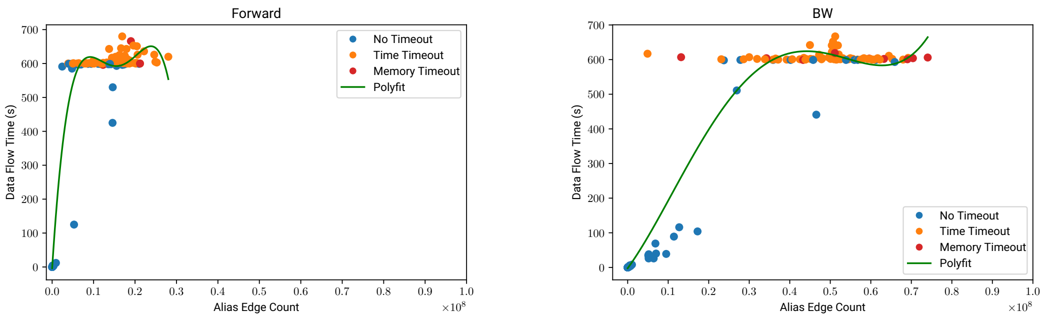
To conclude, our backward analysis is efficient enough to be an alternative to the existing implementation. We even found that it performed slightly better on our app set. Our evaluation shows that there is no correlation between an apriori known parameter and the runtime of FlowDroid - in both directions. Furthermore, we did not find any apriori known parameter to decide the favorable direction either. The edge propagations have shown that our implementation can analyze roughly 10^7 more edges than the existing implementation in ten minutes. Though, the sample size of 200 apps is too small to generalize statements and our data was rather challenging to interpret with a large standard deviation.



(a) Total Edges



(b) Infoflow Edges



(c) Alias Edges

Figure 6.4.: Data Flow Time in Comparison to Edge Count

6.2.3. Memory Evaluation

Table 6.8 shows an overview of the results from the memory evaluation. Note that we only measured the memory usage of the edges in the exploded supergraph and not of the full program. And unlike the time measurements, the memory consumption is much more distributed across The measurements with timeouts show similar values for both directions. Though, maximum measurements with timeouts are not really meaningful. Without timeouts it is a different story. Our implementation has the advantage in all values. The average maximum memory consumption of our implementation is $4.6GB$ lower than the existing one. The backward analysis also needs about $5.7GB$ less memory in the 85$^{\text{th}}$ percentile.

Next, we look at the memory consumption difference per app in Figure 6.5. The x-axis shows the delta maximum memory consumption in MB and the y-axis the frequency. Each bin is $1GB$ wide. The delta is calculated with forward as the refrence: $m_{Backward} - m_{Forward}$. Again, we see a gathering around 0. Otherwise, the histogram has a more uniform distribution than its time counterpart. Still, there is a slight bias towards the backward analysis. Because we only measured the exploded supergraph, there is a linear correlation between edges and memory usage (c.f. Figure 6.6). Likewise, we observed a correlation between time and edges. Thus, this bias could be related to the faster backward analysis on the app set. We looked at this by comparing the sign of the delta data flow time with the sign of the delta memory consumption. 48 apps had different signs, with 23 being negligibly close to 0. Hence, the claim is true for 109 of 134 apps.

Also beneficial for the real-world usage of FlowDroid would be to estimate the memory consumption to utilize the available resources efficiently. In Figure 6.7, we contrast the memory consumption with the number of sources, sinks and the ratio of both. Figure 6.8 shows the memory consumption in contrast to the statement, method and class count. The arrangement and legend are the same as in the time evaluation. Unlike in the time evaluation, there is only one cluster of dots: those terminating nearly instantaneous. Otherwise, the dots seem to be randomly distributed. All graphs indicate no correlation.

To conclude, our backward analysis performed a bit better in the time evaluation, which is also reflected in the memory consumption. Again, the results show that the observed edges are way more important for memory consumption than the code size or the sources and sinks. It is not possible to estimate the memory consumption prior nor which analysis direction will use less memory.

Metric	**Forward** Avg	Median	P$_{85}$
Maximum Memory Consumption	$10005.68MB$	$10459.48MB$	$15482.98MB$
Maximum Memory Consumption Without Timeouts	$7168.50MB$	$8090.91MB$	$13544.93MB$

Metric	**Backward** Avg	Median	P$_{85}$
Maximum Memory Consumption	$8326.27MB$	$10008.52MB$	$14539.64MB$
Maximum Memory Consumption Without Timeouts	$2594.34MB$	$27.30MB$	$8786.48MB$

Table 6.8.: Memory Results

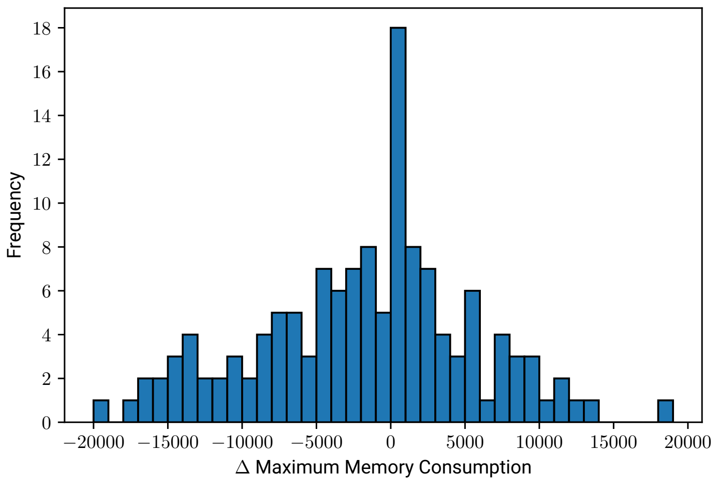


Figure 6.5.: Histogram of the Delta Maximum Memory Consumption

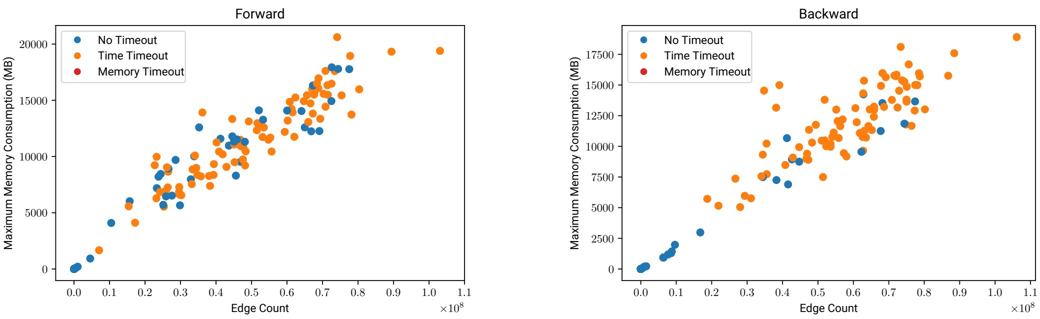
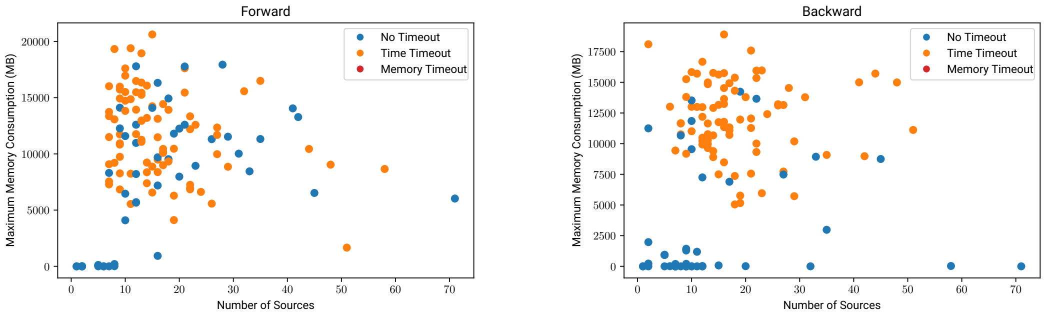
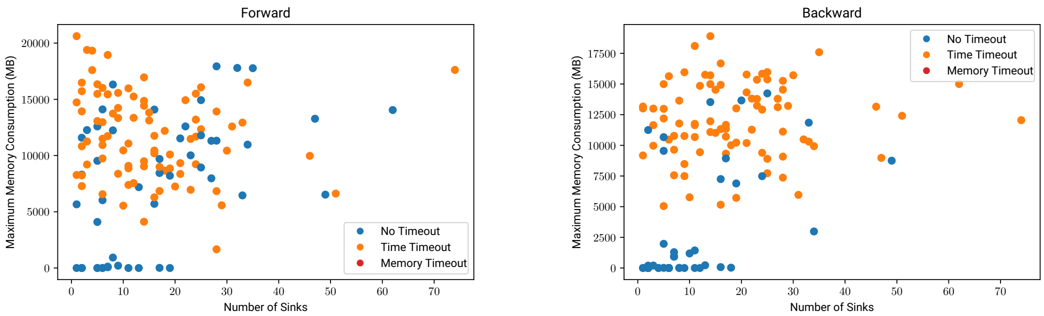


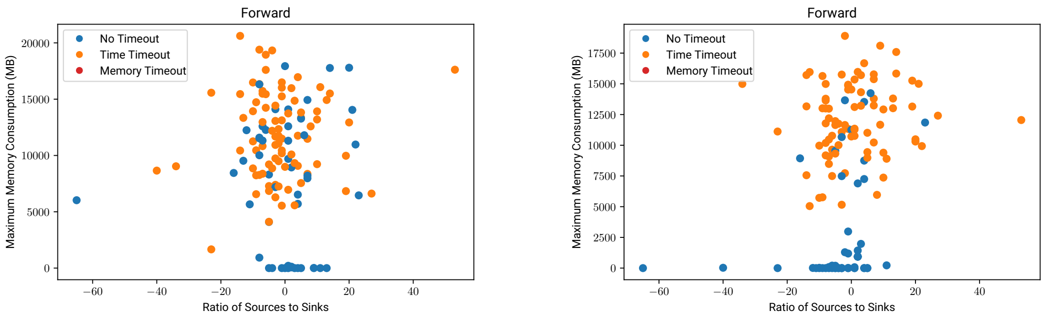
Figure 6.6.: Maximum Memory Consumption in comparison to the Edge Count



(a) Sources

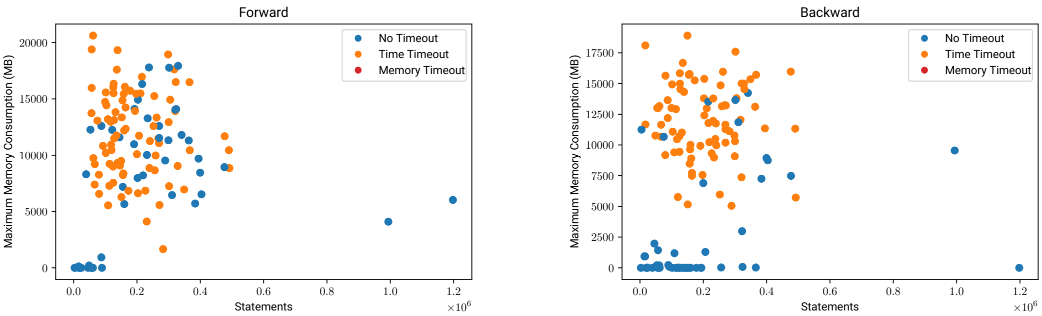


(b) Sinks

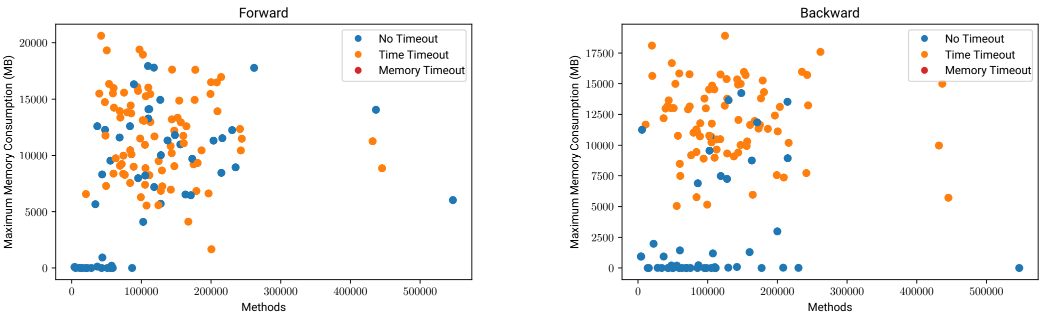


(c) Ratio

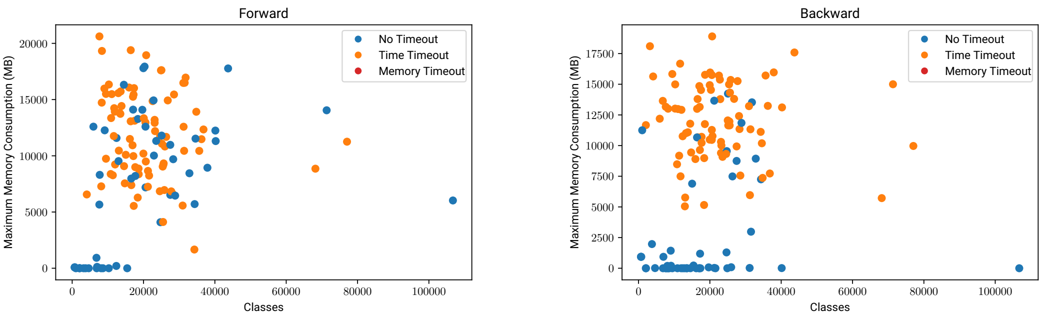
Figure 6.7.: Maximum Memory Consumption in Comparison to Source, Sink and Edge Count



(a) Statements



(b) Methods



(c) Classes

Figure 6.8.: Maximum Memory Consumption in Comparison to Code Size

7. Related Work

Starting with a taint analysis based on point-to analysis [17] in 2005, taint analyses made huge progress and gained traction in the last decade. TAJ[28] uses a context-sensitive forward thin-slice[1] for local variables and a context-insensitive points-to analysis for the heap. Other taint analyses utilize slicing, either as a preprocessing step [] or standalone using a chop, a combination of a forward and backward slice [26]. Andromeda[27] was the first data flow analysis to incorporate a on-demand intertwined alias analysis. FlowDroid[6] then ported this concept to IFDS and also introduced a novel approach for modelling the Android lifecycle.

All of the taint analyses we mentioned use forward-directed analyses. There are also tools which have a main backward pass similar to our approach. We take a more detailed look at them in the following paragraphs.

Lerch et al.[15] contributed FlowTwist, a static taint analysis tool based on IFDS to detect confused deputy problems[2] 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.

Allen et al.[2] present another taint analysis based on IFDS for Java used internally at Oracle. They also rely on access paths as a heap model and chose a backward-directed analysis. They reason their direction choice with the use case of detecting web vulnerabilities where sinks are less frequent based on their intuition. Also, they have made good

[1] Thin-slices only include statements responsible for the explicit flow from or to a seed.

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

experiences with a backward analysis in the Parfait[30] project. Orthogonal to our work, they intentionally go without alias analysis and cut-off the access paths at $k = 5$ without appending a wildcard. Both are trade-offs to precision in favor of scalability. The taint analysis is compared to another non-public tool at Oracle on three benchmarks[3] and on a not further specified Oracle product. Also, the choice of sources and sinks remain unclear and only a short summary of the results is provided. Because both tools in the comparison are not public and the results are not detailed, we neither can comprehend the weak points of their analysis besides the missing alias resolving nor score the given runtimes. We are skeptical that their analysis is capable of finding non-trivial data flows because aliasing is ubiquitously in Java and access paths of $k = 13$ are observed in real-world applications [24].

Yan et al.[31] 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 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, FlowTwist and also Allen et al's tool are based on IFDS. Even though IFDS seems to be the most prominent choice for a taint analysis, there are also other frameworks capable to formulate a taint analysis.

Synchronized pushdown systems (SPDS) by Späth et al.[24] 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

[3] Securibench, WebGoat and OWASP.

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.
CogniCrypt [13] finds misuses of cryptographic APIs based on rules written in a domain-specific language. Internally, it also consists of a taint analysis and is based on SPDS. CodeShield[4] is a propietary taint analysis for cloud applications to detect vulnerabilities and also based on SPDS. The only open-source general purpose taint analysis based on SPDS we found is SWAN[5]. It targets the Swift programming language but is still in heavy development and not ready-to-use.

Doop[8] is a framework initially for pointer analysis. In contrast to all 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[10] extends Doop with a taint analysis. Doop is flow-insensitive and, thus, P/Taint as well.

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

8. Conclusion

In this thesis, we extended FlowDroid to feature a backward-directed static data flow analysis as an alternative to the existing forward implementation. The alternative analysis is equally precise and sound. Just like FlowDroid, our extensions are open-source and possibly will be integrated into FlowDroid in the future. To our knowledge, it is novel for a taint analysis to offer two distinct general purpose analysis directions. Moreover, our work broadens the applicability of FlowDroid for real-world applications with a amount of sources much greater than the amount of sinks.

Furthermore, we evaluated our implementation against the existing one in FlowDroid. We confirmed the assumption that the runtime and also the favorable direction highly depends on the analyzed app. Both analyses put up similar numbers. In the app set we used for evaluation we even had a statistically significant smaller runtime on our implementation. To fully utilize the benefits from a favorable direction, we investigated whether apriori known parameters can be used to predict which direction performes better. Our experiments included naturally known parameters such as code size, source and sink count, but also a fast preanalysis. None of which showed a correlation toward the runtime.

As the prediction of runtime remains unsolved, further research should continue on clues to choose the favorable direction beforehand. For example, it is still an open question whether there are certain taint analysis applications (e.g. to find SQL injections) that favor one direction. Also, further work could evaluate the impact of commonly used third-party libraries on the analysis time. Additionally, our work focused on the most-common context-sensitive alias analysis. Other aliasing strategies were not implemented for the backward analysis.

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] Nicholas Allen, François Gauthier, and Alexander Jordan. "IFDS Taint Analysis with Access Paths". In: *arXiv:2103.16240 [cs]* (Mar. 2021). arXiv: 2103.16240. URL: http://arxiv.org/abs/2103.16240 (visited on 04/14/2021).

[3] 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).

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

[5] Steven Arzt and Eric Bodden. "StubDroid: automatic inference of precise data-flow summaries for the android framework". In: May 2016, pp. 725–735. DOI: 10.1145/2884781.2884816.

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

[7] 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.

[8] 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.

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

[10] 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.

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

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

[13] S. Krüger et al. “CogniCrypt: Supporting developers in using cryptography”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Oct. 2017, pp. 931–936. DOI: 10.1109/ASE.2017.8115707.

[14] 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).

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

[16] 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.

[17] B. Livshits and M. Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis”. In: *USENIX Security Symposium*. 2005.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

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

[26] D. Titze and J. Schütte. "Apparecium: Revealing Data Flows in Android Applications". In: *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*. ISSN: 2332-5658. Mar. 2015, pp. 579–586. DOI: 10.1109/AINA.2015.239.

[27] Omer Tripp et al. "Andromeda: Accurate and Scalable Security Analysis of Web Applications". en. In: *Fundamental Approaches to Software Engineering*. Ed. by Vittorio Cortellessa and Dániel Varró. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 210–225. ISBN: 9783642370571. DOI: 10.1007/978-3-642-37057-1_15.

[28] Omer Tripp et al. "TAJ". In: *ACM SIGPLAN Notices* 44.6 (May 2009), pp. 87–97. DOI: 10.1145/1543135.1542486.

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

[30] Kirsten Winter et al. "Path-Sensitive Data Flow Analysis Simplified". en. In: *Formal Methods and Software Engineering*. Ed. by Lindsay Groves and Jing Sun. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 415–430. ISBN: 9783642412028. DOI: 10.1007/978-3-642-41202-8_27.

[31] 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

App Name	SHA256 Checksum
kr.co.smartstudy.cartown_android_googlemarket.apk	8418072e714bcc1aeb0fcc680e28597593cb58cff2620c491f5553f79d88ecea
org.mopria.scan.application.apk	4b8a16d52786fb93fa73dc369f33ae162dbd2511d53b6c7e8fe23c9e3e8c18e3
com.groplay.tutalfonsaberg.apk	8dadce517f3eb70a0290a46c9e8e03b904b11c7ee081e7152d8307e0978b543d
jp.co.canon.oip.android.opal.apk	68ad1ea695b166bce8ceedcdb868a72331ca84464bd09c1a2ef7f1240c2975d2
com.binibambini.minidancing.apk	37c8b39f21c13d93501b13fbae9333276bddc8cc91dd1d33adfb8c2199fe9978
com.amazon.dee.app.apk	cd7cb55c7ac0d07f7cc77c0dfec8eb5e997f24e646c41cd9d79c98d872342581
com.musicplayer.player.mp3player.white.apk	862d93dbc58ffeee4e2c6aabae6cdb5d6cc13597c71d91065a577f2941664894
com.rvappstudios.sight.words.phonics.reading.kids.games.apk	8086ccf8d37dcab37aacffc8fc4f18e9d2cc95930f5d3455c61ed7d6924240c3
com.nhn.android.navertv.apk	df82e47d243e39c5d50f8d13275053c616c3c0e8b16a9de28ba6012f10bf7419
ru.beeline.services.apk	52341773824950e4837f568eb4dddac1b24287e8d783dd530b6bf44b4d8e71d7
com.fsn.nds.apk	ce9a6c7e7f4d19c8612be401df1b63e9ca47c0ca5e098534b91e8781c47940c8
org.familysearch.mobile.apk	78a71de5ea3201fe5ddbb78863cc660009f28bba99c2d02de59f70a001828026
com.radio.fmradio.apk	195cb39d61cec6352e9d9d81324ebbc00f5b5b1f3cb9f9885add78805d4957e8
com.edokicademy.montessoriacademy.apk	6a6ae06bd561f21bb3db23f5c8329f730bec7f91ca2c7b913bf09d7af0fa4e9b
com.fivory.prod.apk	5e71cd35a08542fc4a08ad9b4f2d11d31220f305935a8732c03c6f5a73c6d3f7
com.tocaboca.tocahairsalon4.apk	4741f734998795ef97a25d424b6e078b50c1657dc60f0537dbab4ddfa043dd3e
com.speeddating.ad.pro.apk	4c6bc4808a3d85e641c59349f1dc77e31680120a2e6a251e809cb0a2bb171875
com.mufumbo.android.recipe.search.apk	1ae688150b2ea8f61d2312c002c74e036beb8939818e76b675e4790a4fca1bba
fr.tisseo.android.apk	aad5d2f60bdec52256f98b75e031f8019e71d9ac8b3dc2062a747968b63cb758
com.kedronic.cbndinosaursfree.apk	e3981f77053713d3c8a548a33cb4ff6ed0addaad205ac0f9625551d040733398
com.labs.merlinbirdid.app.apk	b56750a84c97ec7a68ea9df287725aac9807fd6947821d05ed68e3a0ae9f6989
com.tacobell.ordering.apk	e71109bd49a559a07b53463baf06a73441046736858d548d86424d40130cba8e
kidzooly.fivelittle.apk	ae53861ad8ac4e0cc24cdec1cef8ca42a2dc1acc1cfe947e1478ecbeaaedb711
com.backgrounderaser.cutout.photoeditor.apk	3b2b960b015fe0856a428226666f9110c0bfe4f54805899363b0816ce0277a78
com.newspaperdirect.pressreader.android.apk	d8484f6c7223c21abc6089cde52becff8330206b2852fe5eb9cf6abc3829917c
com.khorasannews.akharinkhabar.apk	4afdd725f8fc67b4f3731513d54934ea129d23f92df14c5e72d1d989208284e8
com.microsoft.office.officehubrow.apk	9f843239b726f490a2507b7a0ec65b7016c2707cae89befb1e7f40bff7cbbcb3
com.cabs.apk	e7e50f08b7f84110ba4454d4d718bcbbaf91a0d59b20e52ae6aeb1acb80b60c1
com.hasbro.tf360appstore.apk	bcbf06c936a17d916d9bc919ece29799e04eea42eb06a4ea7b404206ee9c5f54
com.budgestudios.MissHollywood2.apk	4a353c4422edd629a5c7ec8f2edb7548be125944631d93010ef4265ccc79cfaf
com.amazon.mShop.android.shopping.apk	1088bd390ac9f563b2504bbb2198e716a19955649963fd22020a645d8bb73b1a
com.romwe.apk	3286701bd36d9aec5b27df8f5dc09c2b9bfb7052630810d2f0d8ec874ebbfff1
au.com.parrotfish.phonemic.lite.apk	78cb56109dcf59d25e9eb9198834ea438beb3f61e02eb6e653ac9cedd3dd63c1
com.budgestudios.googleplay.StrawberryShortcakeIceCreamIsland.apk	4e9449df1339e286d012bc7d7cabdd8d01d9bf9d511007685795e129a614c425
com.autoscout24.apk	e092be53ade9be07e4c240d7d03cb7556ef057c289987d273f21d7498a6e0eda
de.number26.android.apk	06e3936e3a86ef322f3f34ca161385c4ceedaf780441a50e37ed4eee9e8d532e
com.bplus.vtpay.apk	2cabed3808b291835995d4a76d8e54c82219b2dd836374bb77c8b432cf6617fe
hr.palamida.apk	27cf78d7ffacb9b9423002a2f28e3f27058df90c0e08217da5d4219e05fdcb41
com.kinky.fetlifestyle.apk	5b16b9e6fdf13f3d58a9580dd0607f58022e3882c37466f72d2932ee1a305065
com.indianexpress.android.apk	636e7e086f17641b97788bd7300a20e16aff7a00884e43a691ebdcac4d1d9ab9
com.fabernovel.ratp.apk	f3d3fa90757a28f0cb10a72b4b773d601f7b247b91d4afc699f415221104fb5f
cz.seznam.novinky.apk	97cd2918938ece6f424b48bd7e26331de1859010f92ead25e9ae20c405d7691b
com.dywx.larkplayer.apk	4f665ba83f38759ffcd543b45c5c70ec1378f44b446a4a849b2df624ef9d6a03
com.euronews.express.apk	b035eed311648ece2e14ec100e3364b2bdf519919363e2ce4d06529749d1833f
cn.wps.pdf.fillsign.apk	c9570f742fd38114c9bdcb259edb40d6448f8a8dcc6bfb33d0d84564362dfc34
com.citymobil.apk	b1193852f2af90f359f909fe9462e36346c2dc8fed4dbb22aacaaa6892ad3b3c
com.starfall.StarfallABCs.apk	1ee69079e23c3369b9a1ef8113e60cda6855a5427e4234c47da9f2f0df22b248
com.apartmentlist.mobile.apk	d6c97d944b806ac21b9f904ea194deacf593dce783c42f765132fc0088eaa5b3
tw.com.ctitv.ctitvnews.apk	74829666eeeb1994657bf8d91c6f4203a19c9a4c98258bd00491f1b62d454472
nineNewsAlerts.nine.com.apk	5cee9279a553e237e72bb722a55f8195a2bdf87940af6ce6f9106dc6abd6b4f7
com.grabtaxi.passenger.apk	e1bd94679df49deed01c62451fa4600a07db592d1d4d9994b7cf5e6d760846ff
com.hk01.news_app.apk	9588f493009410cf77d02743dd4c08f2532e6edbed8139869fe84d5c0b03ff1a
mnn.Android.apk	39faafcbf57d1d5304589c8821ec39fc2617b2c3516f93f9257fe5d06b80824d
com.ubieva.cura.userapp.droid.apk	78363ffe77b2990147eaab29f5b457c69edff3a47dad3b3345f12b3f6421a0fe

App Name	SHA256 Checksum
com.simplepractice.video.apk	b4e865f7a88ce5a93af23e0babe44a6297ff03e870780705c608bb3ccee2e23e
com.google.android.apps.seekh.apk	8e7d1497c1a47fa95696c9420108db519521f3b105ee6ed5fd1bbff93f328ec8
com.murphy.driverewards.apk	03ff479296df50738a22d3e53cbfa31c89587dabc7cc4e37e482b425fcedab26
com.pjmasks.heroacademy.apk	b334d463d46c32b4cea340eb4cf90a66a4f30c6f139f68ccfa290501eda3afac
com.aspiro.tidal.apk	76815743bcecfbc496dbea55548e6f8990a69f131877d7618e3371c7b9c829c2
com.skyhealth.glucosebuddyfree.apk	e44196157332eb962bc360a31fcc0cfc26940bfe609b6ca4ed58602edde38199
com.vuclip.viu.apk	a2656f844f94474a7127782bd15ce2cbc058c6e6119de7ee301873e0d7bcbd3b
air.com.pepiplay.pepigarage.apk	c7b91a6417bce6abbb5b2d4ce1323aff3f6734153f38ebe9a592e23272bf71c9
com.google.toontastic.apk	713bc62548c59f490f71befdc8b3fc3d87c8709c40d4fa6cade85188d851fb5e
com.touchtype.swiftkey.apk	7e468abed703af7460057a03894dcc0f662707273d74da37a69be39c4cb6661d
com.croquis.zigzag.apk	37e78c46bf7b3b83171f742a127845f32f959902aff348fe9faa147526505486
com.yoox.apk	229ce64b93b142ea9be0c466db8615b78dccabc5c82365cf27d84d7a8b32a5d1
com.application.zomato.apk	89c5d4ded29dbb7086817ca8b587de5bffff7940983097f99fd8b0ec16f4393c
com.rcmbusiness.apk	8e7fe9a43abe495ce33000cdfe5c82e61767f89ce1a21761b424148493598a58
com.zoho.show.app.apk	00ad2b874ddfb64a71a5acd009fa25e468879ed46dd95e03f9716155f9f2c1d0
com.streema.simpleradio.apk	ce25bf663eea858613fbdaad93f2a9b7686306d6d6519b1632ee12506d8d79b2
com.google.android.apps.docs.editors.sheets.apk	404839c316cf7da89aaadf76fc5d3ad5b83eeb388e66972c17c01cea61f5be69
com.orange.kidspiano.music.songs.apk	bec50a9d2fff2a10d7f00a07ee90342d0fd5020c53ccc1e82de2e7d151fa77fa
com.storytoys.sesame.elmolovesyou.free.android.googleplay.apk	abb874e0e937fa0e5f60a42aa88aebe7165f22aba51c1473fb5e0398dd3be4e0
com.creditkarma.mobile.apk	67e8ede728e14867360a225dd438a894e35167ed7e5ca02f185e2ea9e4fde0d5
com.originatorkids.EndlessWordplay.apk	3b3a99e848987b3a62b62d841a979094abb67e23fee6b6c22744bbd28552b0eb
vivino.web.app.apk	973fe4a43c2043e62b838e3b3f0c1c37ccf2f7df80ede965edd9c39694bf3bd9
com.playtoddlers.sweethomestories.free.apk	13733d16365110e5488447538c712948a759ca87af35dc3c07fed6c6588d1a4e
com.nbaimd.gametime.nba2011.apk	23bf97f3c22a77b9099e015f8c52564ba946b6b07b6b2e875f2cd28817935132
com.flur.apk	dcc34218c61cdeff2ee3c6585c2d35c317214fa2f46ccc20ee2307e1602fed61
com.budgestudios.googleplay.ThomasAndFriendsMinis.apk	fd33c8a08ccf34810efc72cc593fd92200782589fed11a2278126401a5ef57e7
com.lezhin.comics.apk	9e7d8ffb9058daf5c307a7d073edf9113155266dd94447a23d09155627092c60
HinKhoj.Dictionary.apk	0686392baac6af8bc914b0f24516b3ae157973511c75e0e85b22baaf2c5ef334
com.videocall.randomfriendvideo.apk	78f2192d37c09f462c7b5ae7ab5e6992c47a8f57683513704550121f1d1ba541
com.cupidmedia.wrapper.filipinocupid.apk	eed66095038d1ee8e64e90e683bd4fc2c3a8c84183c182e14e60580cae927739
com.xfinity.cloudtvr.apk	1f9dc3426f87910ec38c4edc4ffd1324e3203814bb28a962fa959153c203d162
dbx.taiwantaxi.apk	2095de12ff28e7608047c857d5ba94e95405e3753d412ea9c7447d1a69f150b7
com.socialnetwork.hookupsapp.apk	6f484feab595fcd69ef4780ec994ce2fe2fecb4c95afdb18e607c793ce8e4a5a
com.bilgapp.mydevs.mathapp.apk	c5ca6e542433a94351d200a8e022a4fb75e1d4756dc55f66d8234d2115d05381
com.clusterdev.hindikeyboard.apk	84ccada3b4ee531697823eab17082713aef2076d142f5213fe62cc11f031b7a9
document.scanner.scan.pdf.image.text.apk	a2756e08587e804b0f3fd694222a18c68368f1f988a45e42ccf2b229b2f72eb7
com.splendapps.voicerec.apk	c8f53b7f1918755c60ae16ace4e5ecefeabead04e3311314e3ff4d8b686d4fa2
air.com.turner.boomerangmakeandrace.apk	516ede7668b7f9d13ece8d9c8358f4a2c4044e84c8bf09050d4e7ac94681fc4c
app.habitaclia2.apk	a8e949ffe4e6b3a67b52a6a9d59c4880a5ec252148c2e7592c644f0b55b07e4e
com.FDGEntertainment.Oceanhorn.gp.apk	e53c6f684f4a01037238e9ebc9c79aa028291cd3d4007dfedf8d90f222be09a5
sweet.selfie.lite.apk	0afc498f4b19d2f33a9772ddcb0a0c5bce554a79931ffe1e676ad2d81c1a87bc
fr.anuman.HomeDesign3D.apk	9f7f2a4690c2c2de7ca24e457ec75d57c9e97b7f86726a12ae9e80166fe17b0e
com.espn.score_center.apk	643813688fee805d2f8ef33b2afc7fb2591d6e4d15a3179a79846332558bfd25
mobi.ka.gp.wrkshts.app.apk	083d0771826379f85d53892a661591dc3591bdbdb0fa4b8939366b1399acde3a
com.fineapp.yogiyo.apk	a4d16bba29ce7c01990cbd1e07608cf1fb478137d2e0a08ddd4a563137f0d45a
com.thredup.android.apk	1a7f39f229d09b371f9241b097d7c71b120b3363dcd9c5552003a0bd8f08d2a1
kr.co.smartstudy.dinoworld_android_googlemarket.apk	b92f2ee76807e26deec705eb48ea7168bd1dfa2152cb313772209ca7f19ced11
com.opera.touch.apk	4b67fb5161fc1e3b1fc4d900a606d6e59903a6eecebcc2c12a0c96c45860f91e
com.calmid.learnandplay.launcher.apk	7e582912c2f9f12a5e4150b6b7c63df1b70e261d7ef0d57804bb888ec726ad06
com.astepanov.mobile.mathforkids.apk	c08f4f8ca574688175b849881b5e4c97776b1e3830525783b229c080605df6cd
com.microsoft.math.apk	2cedcf2fd59d5b23ec44f6e3c3b715522e92b8bb2c2515fd223cde95af645960
com.citi.citimobile.apk	5ba14ca78a07bdc80f96f4ddfd84c096b58b870e1c9e75b22dddf168452e18d6
com.yy.hiyo.apk	cca85c559c68dbdd65324e83b7a045f006ccf36ac18c9c0c4307e7acc1f56ed2
com.kedronic.cbnvehiclesfree.apk	a7b5ffdf5cd94e2475fbca33effffa7186603f6781c0af4c64a95b0177136ea7
de.etecture.ekz.onleihe.apk	bb44ebf657d97a2236df28efc7d5f0d1f02bc30a75f15652c4267df107882456
com.urbanclap.urbanclap.apk	1bbdbfda6a19b6e2fe241757a140539411a1f36aabe2603901b243305fcd4a30
com.niksoftware.snapseed.apk	e0b85eed2f84aa09bdfe65033cb9a48e9b97edde5eb6d7b5a0e04b13825f0b31
com.kevinbradford.games.pklg2.apk	52b733c7c41b871b4df60cf0989afd4fed144e7534cbed2f931503b8875d653b
ua.insomnia.kenya.newsi.apk	d2c7dfe4bb09e928f3cce393fa18ee7df0371ed5adf652d788b7b58441b05b16
com.microsoft.amp.apps.bingnews.apk	39b24376d9ffdb027135a53e07c5a65ac6296ab77e32a7a53ebd5840f6aebb5d
com.rvappstudios.abc_kids_toddler_tracing_phonics.apk	76d7a1613be00b4bfb4eeb4e647ef5f6f5222ccbf5473ced542e9e07d4b51fd8
com.budgestudios.CrayolaNailParty.apk	44977ccffde8abe8e9775435a8c15fda5ee78a809f3cf00b6474e2d117fd633a
com.bimiboo.puzzles.apk	21b1f93f974429d2c4589de445115cdb590ae9d34c973dd59bd9157b08d3e168
org.pbskids.video.apk	b1024916bae5e592e58f71eb5c3f32162172334bf7bd376b50b29adbe7278621
com.rvappstudios.shapes.colors.toddler.apk	ab434ef8106e24d39e79af477b45a97f40de526d3caed3bfa421957aae3ce34b
com.curse.dndbeyond.apk	38ebf5fd508c05559f02a5094cd7ba4816ebc5bda3a1f67c156104ff78d500b9
com.money91.apk	f10db842b0b8051e8ed752d0eda71cb5110cbce3d0a57bc10bb366a0176411e4
com.foofoo.tracing.apk	8aba701329ecbbd52dd5b6060ff7d1ea6f3fc0f736b79182c2c28237803c49d1
com.cisco.wx2.android.apk	e43d84e5609a1f7cc9c76195c27355abd2634723a43ace192ecbca9e9015a1d3
com.adpdigital.mbs.ayande.apk	a7fb7a5a13128ea90643193b6156ce9c06f1e987e7feac4966119aa9aa04a919
kz.kkb.homebank.apk	aa833049899cf5dfce78a552a8fb115fa837d6a2b813798163277b29fa86c3cb
com.aol.mobile.aolapp.apk	214482468cea54a4d0e85af74ec9a8f554d674cc5f92a5b8527de053dd9cb4f6

App Name	SHA256 Checksum
com.resmed.myair.apk	02dfb457f663a1416d9efe9d829088b56e759430a692ab58977edfd6b0e21a2b
com.sadadpsp.eva.apk	46b990e564c0af762bd983fce9513547d97546d47b0f9002e104e65768c127d7
com.budgestudios.googleplay.HotWheelsUnlimited.apk	4de9cb4ced761b8840fa9dbf071f50f83c50e1c503f403eca5af58cdda3c626c
com.zoho.sheet.android.apk	85ae0ca44350672d0ff45f1ddfaabbef3e066ed41cb32de834fdc8a7319f43c7
com.facebook.mlite.apk	8021599efe390a204262d673f9e62ed05054a59c02d93a61e3300b46d895694c
taxi.android.client.apk	013b701e2d70c4f3dbc8abc680a059697018dd02c93de491e54b7ca1692e57e8
com.budgestudios.StrawberryShortcakeFoodFair.apk	efe0f27383e12a8c0902cc771c5606e2dbb7711fe6939efa8c08d784a1100463
au.com.auspost.android.apk	42cadeb194c47220a062c2a6e59ea99728b324016794dc67a846a27b18559ebf
com.dating.find_love.apk	748fc4335fcf91c4aa1da9d22b6bf2177eb637676680863325e56aaf20b77770
ma.safe.bnau.apk	2a49e104e9c2aeb4680f0ea4f611bbac38353a825dbbc75b20b3b00c87f6d95b
com.buildium.resident.android.apk	a9f36d1586ac8f6e0e98a7da0c722942a36452047202026f9e98b08496340812
com.affinityapps.blk.apk	71d10693759939057ed7613f19756ced19bcad130503e7d760fa2a381bdcd75f
com.jrtstudio.music.apk	ebbaf09fc37e65af9089bd815b7528ff9c067d9c95e901111974480939bcf0d7
app.quiktrip.com.quiktrip.apk	00f66fdc41dbb684bd122a20cc7106392d8352246998ed249528d60303bb1a48
com.sendo.apk	ae51b595f53d5f9cebef94a404431a38faa8f3c1cb8c9ac4e252193df2fb0934
com.covalent.kippo.apk	6596e1854adf1b70d9983463cb7dedd72c5492e4645fb6351f2e7fd65a88adbf
com.weedmaps.app.android.apk	67e97774f38a5845ad45663342ddd50b4d73749698f4d83ad5e7eba9f78e470f
com.emra.AntibioticGuide.apk	8c9365e2eaf715f44f62ffd8bb1b0fbe1aef8a60355ace7b5821e534b59a71b4
com.onlyoffice.documents.apk	3e7afe11ac7748ff6a7282ab13b17058d73ae22b8c3aa624a499a44607cd2147
com.ak.ta.dainikbhaskar.activity.apk	2cbd59a1c20e63ad7928276ffb0de2c550b138b7a7d901448708d41a65b54dd1
com.cloudmosa.puffinFree.apk	f7e724e3d3dabcbd0c6ad61a849736bdd98a2ed84037826c1a69b9fb69b68acf
com.spinmaster.enterprise.colleggtibles.apk	0a25e9808c34d2a35e51c2da2bc2ec57bef20ef3659b98cbc85124a5aafbd41c
com.finlim.forkapp.apk	9244a6aefb04db4782c335a39ca65dc6d63baedb9ba066e017b679af68b046e7
com.planner5d.swedishhomedesign.apk	4f92722e91e48271b4dacc75cf279779be90a1c795aa14a1a4e9a60ec2ab8acb
net.psyberia.offlinemaps.apk	a7179816079cf58190fb910ac20dfeec028c5dc199b4baa454d6bdc9649929f1
com.nhn.android.webtoon.apk	dcbfced34bf850d89aa081668769169cc3a3a4e7e77047aaea2b868c41407af7
com.ace.android.apk	f648fae084e813040a2dddbdd7ae9272d8dcc23e5106da001b23abc6555528dd
com.lafiya.telehealth.apk	ad7cb0615e066bf8f8339b3e954d680cb930b5e96a0322e9b06900e62eb978a7
com.waitrapp.apk	d346641d19bcbdc82298d24a2a19d4a3d005acb82ffd662339e945128cd64b74
com.match.android.matchmobile.apk	07e6158383c378f3c177ad406539d49eb5eba57c281d3b27c4da528541a723dc
com.clouthub.clouthub.apk	040338ae80e20a9c3b772ebd0ca513607debfc93964e3314fe935424b1522c49
com.belk.android.belk.apk	ee53358cf307bae014c6a85d499e10f02ab9c2a335b47d65c6813f54ad4f8858
com.lyrebirdstudio.face_camera.apk	4b4a20f9569c43f260350518ec9c5e3e3548d52ed8f8b92b9bcd09babb54d904
com.poqstudio.app.platform.boohoo.apk	d8104cfd2d2bbe18d69f9b8550ea6dd5e89927b6ae7000646a45eab59b0535a8
com.foofoo.tracing123.apk	111e78eeed80b86f613aee13b4e93bcb87a3a9287a7564baad7be0200bbf8e74
com.pinger.textfree.apk	db6704e865af504b70488bcbe8143daf790e796afdc5bf4e240777fc2b36f2f0
com.accruehealth.mobile.apk	6a2f15edf4fe9d9b8651d4675088acc70ea3df2a3515b1ca00faa055d1b9fa9f
com.enuma.todomath.apk	1380ebed63a240ade60bbc42f4e581615ca35bf61ae32a3e1b2ffa7f21509077
com.lyrebirdstudio.tbt.apk	e2b83af323cd294ad24905098ff43efcb4d9a7426d830709962996d47b3e1a68
ir.balad.apk	168d8ccc23bbe32051bb627719efcec6067ae55df5f71fd7c28e21ac7d55cda3
co.brainly.apk	3df6da10ac62569cf7d52c261fba5f4701624f13648f800e9bdce1cedb7bfa97
com.budgestudios.MyLittlePonyHarmonyQuest.apk	b79cbe2fc940147dba81092e39dfa21f545809eab2304def6e91e9c9ee4028bc
com.budgestudios.StrawberryShortcakePocketLockets.apk	232d2ce3364ce1f4d6062043ca7bf0a4df8ed6d90ba261b0e957bbbe68fb099e
ir.nasim.apk	ee51967db6972af6343a58f2c282c5c882261febd1640e6c9e4836c60139428b
photocollage.photoeditor.collagemaker.apk	ec23c5112a67815ec5faf5f3f74f1987f9fcfe17802ef4458354e3389c751cf9
com.app.rondevo.apk	341bc42f3e13038be22298d13faa2f961d7801c135ae0b3d4a556cd54eb6c4cd
com.foofoo.coloring.apk	b8a345a5b777c727296ce6bff3fb56d255c629f9c314865aef919ae8d5555135
com.uba.vericash.apk	8004ff6108f8f0578e70ab4a31fc7d9586df1d4d9529db053f5005016551d8a4
com.rubenmayayo.reddit.apk	d715f10d81b69f8ae970bea4c22e5807eb7a25afc744ae8ef8d6e260eec576f2
com.adobe.reader.apk	2b0f8e9cadc1355398215cdc61f4f4519124587220d964a1b469f18e6117bfe0
com.musicplayer.music.apk	e832ecfba016f7c0a7decc2cd0f5f1f2ba6539dea74074ee06b1cb596f57aabf
com.footba11.results.apk	fccbed1ac44299c09ccb28e4792b43a03e9022b604b42c7a0e78c551bbe93d21
com.budgestudios.StrawberryShortcakeCandyGarden.apk	06c97d3d7f9d0b3644bdd178f2c205be8359fe201359a4b63f65dcb7838ac1c2
com.microsoft.office.onenote.apk	c249ca4011ec76cd328a1ba6c3a3b10535a1df87d1c5a5b4c1e5e8f3b0da3da7
it.casa.app.apk	0006a8a47b457e9bd2a7e64335dbdd951ca67008b22f47223e72fae34b29660c
com.tubitv.apk	d6595ccdf6d351c49eca443e2519ee5aa8590153dfc301b5d90c4f3ed6202200
info.yogantara.utmgeomap.apk	75fe0d4e3470149e7ec591b7927685b53303828130ddfa36b6920fb93eaa1cee
com.thesouledstore.apk	c57c26eb55a6d0618484a1e35f25ef9f9fc642ce2906d73a7f1fee42f5b9c467
com.duckduckmoosedesign.cpkids.apk	915a3b700fa63cf3e8770e2e0346de186a039fedf4b010da090b5e4a0519ead8
com.dazz.hoop.apk	4b120cd31fc522d4efbfcd65e5aa3d4ec3c543b6af5d2f1ab849bcb33f4dbb78
ir.tgbs.peccharge.apk	c16b725a5ebd1734f665f0cfbeca268058949a608aa9b5701f7576f56ff70fd8
com.sinyee.babybus.engineering.apk	2d1cf94bf57e039f12463d4365152c199a14410a5faa9a5372f76155724f0f1b
com.meesho.supply.apk	a63f721b81514caefed3157df8aedbc18d09a53b944cc6c4fa35087196913dee
de.wonderkind.flightschool.apk	ce69c9d93a2c703f4339efc1b87874c80512d505a937e7c3f06c6673a94f1131
fr.cnaf.mobile.moncompte.apk	1fa6fde595d41916ce246ca16bbedce5f8712e6a55b0a77f08c6d40bfb37df79
com.marlustudio.englishforkids.apk	ba0cb61dc4e913c8ca61c9ad83ed9bc5fd98202ab7f200cdd76068c8544b0217
com.Slack.apk	200de82bccde54d4db7d09e0c887ca39015bad5a2877d01250bf873472208dfc
com.fuib.android.spot.online.apk	066fbef97444bbb17a71688a817e56cf86ee338bc7fb667ce328474edc47ea75
com.playtoddlers.urbancitystories.free.apk	64581c42741a40ab76b404acbc2df39845193716cb5f6e397571021e1b758548
camsurf.com.apk	3b35f57e003092271644fb99aca3aea7cd24f3017d28a7f77742bf3853c8c4e8
com.lego.friends.heartlake.apk	c8539ee963bcd42c6527da0c7c433773a32a8c61df832d6d96e2c079f220bc29
kidzooly.rhymes.apk	42b0e6172af6ae1204582c85807ca04177aeb1b3915c7c6bbb7da27dbe2a8603

App Name	SHA256 Checksum
police.scanner.radio.broadcastify.citizen.apk	ba23252a676054d0f9f02a2a72a670ad943b7d68aec1536721906b6f53c6ac98

Table A.1.: App Set