Context-, Flow-, and Field-Sensitive Data-Flow Analysis using Synchronized Pushdown Systems

JOHANNES SPÄTH, Fraunhofer IEM, Germany
KARIM ALI, University of Alberta, Canada
ERIC BODDEN, Heinz Nixdorf Institut, Universität Paderborn and Fraunhofer IEM, Germany

Precise static analyses are context-, field- and flow-sensitive. Context- and field-sensitivity are both expressible as context-free language (CFL) reachability problems. Solving both CFL problems along the same data-flow path is undecidable, which is why most flow-sensitive data-flow analyses over-approximate *field-sensitivity* through k-limited access-path, or through access graphs. Unfortunately, as our experience and this paper show, both representations do not scale very well when used to analyze programs with recursive data structures.

Any *single* CFL-reachability problem *is* efficiently solvable, by means of a pushdown system. This work thus introduces the concept of *synchronized pushdown systems* (SPDS). SPDS encode both procedure calls/returns and field stores/loads as separate but "synchronized" CFL reachability problems. An SPDS solves both individual problems precisely, and approximation occurs only in corner cases that are apparently rare in practice: at statements where both problems are satisfied but not along the same data-flow path.

SPDS are also efficient: formal complexity analysis shows that SPDS shift the complexity from $|\mathbb{F}|^{3k}$ under k-limiting to $|\mathbb{S}||\mathbb{F}|^2$, where $\mathbb{F}$ is the set of fields and $\mathbb{S}$ the set of statements involved in a data-flow. Our evaluation using DaCapo shows this shift to pay off in practice: SPDS are almost as efficient as k-limiting with $k = 1$ although their precision equals $k = \infty$. For a typestate analysis SPDS accelerate the analysis up to 83× for data-flows of objects that involve many field accesses but span rather few methods.

We conclude that SPDS can provide high precision and further improve scalability, in particularly when used in analyses that expose rather local data flows.

CCS Concepts: • **Theory of computation → Program analysis**; *Object oriented constructs*; Grammars and context-free languages;

Additional Key Words and Phrases: static analysis, data-flow, aliasing, access paths, pushdown system

ACM Reference Format:
Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow-, and Field-Sensitive Data-Flow Analysis using Synchronized Pushdown Systems. *Proc. ACM Program. Lang.* 3, POPL, Article 48 (January 2019), 29 pages.
https://doi.org/10.1145/3290361

1 INTRODUCTION

Static data-flow analysis helps detect bugs and security vulnerabilities early in the software development process, including semantic properties such as null-pointers [Nanda and Sinha 2009], data races [Kahlon et al. 2009; Yan et al. 2011], and misuses of application programming interfaces

Authors' addresses: Johannes Späth, Fraunhofer IEM, Germany, johannes.spaeth@iem.fraunhofer.de; Karim Ali, University of Alberta, Canada, karim.ali@ualberta.ca; Eric Bodden, Heinz Nixdorf Institut, Universität Paderborn and Fraunhofer IEM, Germany, eric.bodden@uni-paderborn.de.

2475-1421/2019/1-ART48
https://doi.org/10.1145/3290361

(APIs) [Fink et al. 2008; Hovemeyer and Pugh 2004; Krüger et al. 2018], as well as security vulnerabilities such as privacy leaks [Arzt et al. 2014; Grech and Smaragdakis 2017], SQL injection [Livshits and Lam 2005; Martin et al. 2005], and execution of untrusted code [Lerch et al. 2014].

To avoid false positives, static analyses require precise algorithms and abstractions. There are various design dimensions to fine tune the precision of a static analysis. For instance, an analysis can be *context-sensitive*, *field-sensitive*, *flow-sensitive*, or any combination of the previous options. Context-sensitivity distinguishes its approximations by calling contexts, field-sensitivity distinguishes two fields of the same object, and flow-sensitivity distinguishes data-flow results for distinct control-flow paths. Ideally, a highly precise analysis combines all three, but in the past researchers have struggled to design scalable static-analysis algorithms that are context-, field- and flow-sensitive. This is because in essence such a setup entails that one has to store analysis information at every program statement (for flow-sensitivity), for every calling context (context-sensitivity) and for different possible combinations of field accesses (field-sensitivity).

We ourselves have applied static analyses in real-world settings and found that field-sensitivity can pose a particular challenge for code patterns with cyclic field references. These tend to degrade the analysis performance. In Java, such code patterns typically occur in the implementation of standard data structures such as `java.util.LinkedList`, `java.util.HashMap` or `java.util.TreeMap`. The patterns also occur when inner classes are used[1], and when two classes mutually reference each other.

Listing 1 shows an example of a cyclic field reference. It shows an excerpt of the source code from the Java 8 implementation[2] of `TreeMap`. The class contains an inner class `TreeMap.Entry` that lists three fields (`parent`, `right`, and `left`), each of type `TreeMap.Entry`. Method `put()` creates a `TreeMap.Entry` that wraps the inserted element. The `TreeMap.Entry` is then used to balance the tree after insertion (call to `fixAfterInsertion` in line 4). The method `fixAfterInsertion` iterates over all `parent` entries and calls `rotateLeft` to shift around elements within the tree (line 10). The latter method stores to and loads from the fields `parent`, `right`, and `left` of the class `TreeMap.Entry`.

Why is this challenging to analyze? Let us assume a context-sensitive, flow-sensitive, and field-sensitive static taint analysis that tracks the inserted `value`, which is a parameter to the method `put()`. To cope with heap-reachable data-flows, field-sensitive analyses commonly propagate data-flow facts in the form of access paths [Arzt et al. 2014; Balatsouras et al. 2017; Cheng and Hwu 2000; De and D'Souza 2012; Deutsch 1994; Feng et al. 2015; Hauzar et al. 2014; Tripp et al. 2013, 2009]. An access path comprises a local variable followed by a sequence of field accesses, and every field-store statement adds an element to the sequence. The `while`-loop of `fixAfterInsertion` (line 7) in combination with the three field stores (lines 16, 19, and 20) within the method `rotateLeft()` represent a common code pattern that leads to the generation of access paths of all combinations contained in the set $T = \{\texttt{this}.f_1.f_2.\cdots.f_n.\texttt{value} \mid f_i \in \{\texttt{right}, \texttt{left}, \texttt{parent}\}, n \in \mathbb{N}\}$. This is due to the fact that the static analysis is path-insensitive and over-approximates the execution of all paths, hence the inserted `value` is potentially heap-reachable via all these access paths. The data-flow element is correctly propagated only if the correct access path exists in the set T at the call to a map operation that retrieves the inserted `value`, e.g., `get(Object key)` or `iterator()`. [3]

The set of data-flow facts T is unbounded. Because most static data-flow algorithms require a finite data-flow domain, they typically use so-called k-limiting to limit the field-sequence of the access paths to length k [Deutsch 1994]. k-limiting frequently results in analysis imprecision, also

[1]The compiler automatically stores the outer class instance within a field of the inner class.
[2]http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/eab3c09745b6/src/share/classes/java/util/TreeMap.java
[3]One way to avoid such exponential blowup is to use so-called store-based heap models, which model the heap via allocation sites. As we show in related work, however, such models have very profound limitations on their own, for instance precluding concise, intensional procedure summaries [Bodden 2018]. Hence here we focus here on storeless models using access paths.

```
public V put(K key, V value) {
  TreeMap.Entry<K,V> parent = //complex computation done earlier
  TreeMap.Entry<K,V> e = new TreeMap.Entry<>(key, value, parent);
  fixAfterInsertion(e);
}
private void fixAfterInsertion(Entry<K,V> x) {
  while (x != null && x != root && x.parent.color == RED) {
    //removed many branches here...
    x = parentOf(x);
    rotateLeft(parentOf(parentOf(x)));
  }
}
private void rotateLeft(TreeMap.Entry<K,V> p) {
  if (p != null) {
    TreeMap.Entry<K,V> r = p.right;
    p.right = r.left;
    if (l.right != null) l.right.parent = p;
    //removed 8 lines with similar field accesses
    r.left = p;
    p.parent = r;
  }
}
```

Listing 1. Excerpt code example of `TreeMap` which is difficult to analyze statically.

known as "over-tainting", so also in Listing 1: not only will the field `value` of a `TreeMap.Entry` of the map be tainted, but any other field will be tainted as well. For example, any `key` inserted into the map imprecisely receives the taint as `TreeMap.Entry` has a field `key`. Because the set T is infinite, the imprecision occurs for *any* value k.

Access graphs are a way to avoid k-limiting [Geffken et al. 2014; Khedker et al. 2007]. They model the "language" of field accesses using an automaton. Access graphs represent the set T finitely and precisely. However, just as access paths, also access graphs suffer from the state-explosion we show in Listing 1. In the illustrated situation, the analysis must store a set of data-flow facts similar to T, i.e., access paths or graphs, at *every* statement, and potentially *every* context where a variable pointing to the map exists. Given the large size of T, computing the fixed point for all these statements is highly inefficient, even when access graphs are used.

In this work, we thus present *synchronized pushdown systems* (SPDS), a novel abstraction that uses pushdown systems to combine context-sensitivity, field-sensitivity and flow-sensitivity efficiently. First, we show how to encode a flow-sensitive and field-sensitive analysis within a pushdown system, the *field-PDS*, which pushes and pops fields from a stack at field-store and field-load statements. Without resorting to any k-limiting, this field-PDS delivers a concise and precise representation of all sets T for every statement. The field-PDS is inspired by a pushdown system proposed earlier by Reps et al. [2003], the *call-PDS*, which solves a *context-sensitive* and flow-sensitive (but field *in*sensitive) analysis problem by pushing call sites onto its stack. The SPDS then synchronizes both PDS, delivering a *decidable* construction of field- and context-sensitive data-flow results.

The problem of field-sensitive and context-sensitive analysis is undecidable in general [Reps 2000], which forces also SPDS to over-approximate. SPDS, though, are specifically designed to expose false positives *only* in corner cases of which we hypothesize (and for which our evaluation confirms) that they virtually non-existent in practice: situations, in which an improperly matched caller accesses relevant fields in the same ways as the proper caller would.

A formal complexity analysis shows that SPDS shift the complexity from $|\mathbb{F}|^{3k}$ under k-limiting to $|\mathbb{S}||\mathbb{F}|^2$, where $\mathbb{F}$ is the set of fields and $\mathbb{S}$ the set of statements involved in a data-flow. In our experiments, SPDS are scale similar to k-limiting with $k = 1$ although their precision equals $k = \infty$.

We also present two instantiations of the synchronized pushdown systems: a demand-driven pointer analysis and a typestate analysis. Based on those clients, we validate our hypothesis and observe that SPDS theoretical over-approximations are non-existent in practice for the DaCapo benchmark suite. Additionally, SPDS outperforms the access graph abstraction for the dominant types of data-flows, data-flows that involve many fields but span across few methods.

To summarize, the main contributions of this paper are:

- A pushdown-system-based formulation of an unlimited field-sensitive data-flow analysis.
- A synchronization technique of two pushdown systems resulting in a decidable flow-sensitive data-flow analysis that combines context-sensitivity with field-sensitivity.
- A worst-case complexity analysis of the presented synchronized pushdown systems in comparison to an access-path based analysis.
- An empirical evaluation of the proposed solutions on two common data-flow analysis clients: pointer analysis and typestate analysis.

2 A SHORT INTRODUCTION TO PUSHDOWN SYSTEMS

In this section, we briefly introduce the reader to pushdown systems (PDS). Technically, a pushdown system finitely represents a transition system with potentially infinitely many states, called the control locations. PDS are used in model checking [Esparza et al. 2000; Finkel et al. 1997; Lal and Reps 2006] but have also been applied to data-flow analysis [Lal and Reps 2008; Reps et al. 2005]. For model checking, each control location represents a possible program state, whereas in data-flow analysis the control locations resemble data-flow facts.

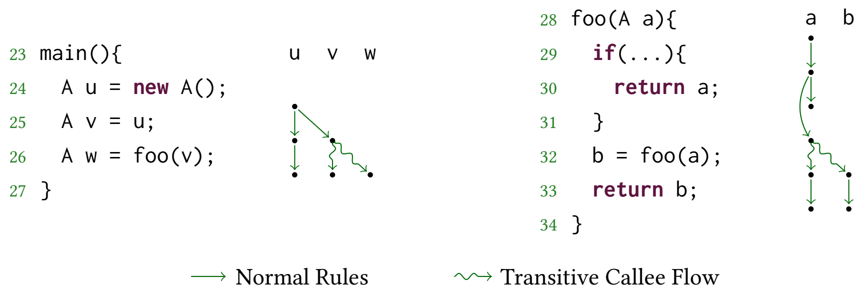
Definition 1. *A* pushdown system *is a triple* $\mathcal{P} = (P, \Gamma, \Delta)$*, where* P *and* Γ *are finite sets called the* control locations *and the* stack alphabet*, respectively. A* configuration *is a pair* $\langle\langle p, w \rangle\rangle$*, such that* $p \in P$ *and* $w \in \Gamma^*$*, i.e., a control location with a sequence of stack elements. The finite set* Δ *is composed of* rules*. A rule has the form* $\langle\langle p, \gamma \rangle\rangle \to \langle\langle p', w \rangle\rangle$*, where* $p, p' \in P$*,* $\gamma \in \Gamma$*, and* $w \in \Gamma^*$*.*

The length of w determines the type of the rule. A rule with $|w| = 1$ is called a *normal rule*, one with length 2 is a *push rule*, and a rule of length 0 is a *pop rule*. If the length of w is larger than 2, the rule can be subdivided into multiple push rules of length 2.

In the remainder of the paper, we refer to a particular pushdown system for context-sensitive and flow-sensitive data-flow analysis as the "*call-PDS*" $\mathcal{P}_\mathbb{S} = (\mathbb{V}, \mathbb{S}, \Delta_\mathbb{S})$. The control locations P are program variables (from the set $\mathbb{V}$) and the stack alphabet Γ is the set of program statements (hereafter $\mathbb{S}$). The rule set Δ models the data-flow effect of a variable at a statement. In standard program-analysis terminology, the rule set Δ corresponds to *flow functions* [Kildall 1973; Naeem et al. 2010; Reps et al. 1995]. Its normal rules model intra-procedural data-flows, whereas the push and pop rules handle inter-procedural data-flows and model context-sensitivity. $\mathcal{P}_\mathbb{S}$ contains multiple push rules for each call site. Each rule maps an argument to the corresponding parameter variable of the callee. The pop rules in $\mathcal{P}_\mathbb{S}$ map escaping data-flow variables at return statements

Table 1. The flow function *assignFlow* for normal rules of assignment statements for $\mathcal{P}_{\mathbb{S}}$.

In		**Out**
Variable ($\mathbb{V}$)	Statement ($\mathbb{S}$)	($\wp(\mathbb{V})$)
x	$x \leftarrow *$	$\varnothing$
y	$x \leftarrow y$	$\{x, y\}$
y	$x.f \leftarrow y$	$\{x, y\}$
y	$x \leftarrow y.f$	$\{x, y\}$



```
23 main(){
24   A u = new A();
25   A v = u;
26   A w = foo(v);
27 }
```

```
28 foo(A a){
29   if(...){
30     return a;
31   }
32   b = foo(a);
33   return b;
34 }
```

Fig. 1. Data-flow example for a simple recursive program.

back to the respective call sites. The normal rules also capture the semantics of the control-flow, which makes $\mathcal{P}_{\mathbb{S}}$ flow-sensitive.

Table 1 depicts the flow function *assignFlow* that encodes the data-flow of a variable x at an assignment statement s. If t is the control-flow successor of s, $\mathcal{P}_{\mathbb{S}}$ has a normal rule $\langle\!\langle x, s\rangle\!\rangle \to \langle\!\langle y, t\rangle\!\rangle$ for each $y \in \mathit{assignFlow}(x, s)$. For example, an assignment statement may overwrite the lefthand-side variable and kill the data-flow. This is indicated in the first row of Table 1 which is read as follows, when a data-flow fact x reaches any assignment statement that overwrites variable x (for instance, $x \leftarrow y$ or $x \leftarrow y.f$, grouped in the table as $x \leftarrow *$), the function *assignFlow* returns $\varnothing$ to kill the data-flow. Statements also generate data-flow facts. *assignFlow* returns the set $\{x, y\}$ when a data-flow y reaches an assignment statement $x \leftarrow y$ (second row in Table 1). Note, the definition for $\mathcal{P}_{\mathbb{S}}$ also describes flows to and from the base variables of field store and field load statements (third and forth rows in Table 1). A data-flow fact y reaching a field store statement $x.f \leftarrow y$ generates the fact x, *assignFlow* ignores the field f which means $\mathcal{P}_{\mathbb{S}}$ models a field-insensitive data-flow analysis. $\mathcal{P}_{\mathbb{S}}$ intentionally disregards fields to keep the set of control locations of the PDS finite. For SPDS to be field-sensitive, the field-PDS models the infinite state space provoked by field stores.

Example 2.1. We discuss an example based on the program given in Figure 1. The program allocates an object that is then passed to a method `foo()` that calls itself recursively (line 32). For the example program, the control locations of $\mathcal{P}_{\mathbb{S}}$ are the local variables of `main()` and `foo()`. The stack alphabet of $\mathcal{P}_{\mathbb{S}}$ is the set of program statements. Throughout our examples, we use line numbers to refer to statements. The rule set $\Delta_{\mathbb{S}}$ depends on the control-flow graph as well as the call graph of the program. The construction of both graphs is straightforward for the example in Figure 1. Table 2 lists all rules within the set $\Delta_{\mathbb{S}}$ of $\mathcal{P}_{\mathbb{S}}$. Normal rules are intra-procedural, in Table 2 we group them by their containing method. The push and pop rules are inter-procedural and their target configurations contain statements of the caller and the callees.

Table 2. Rules of the $\mathcal{P}_{\mathbb{S}}$ for the example in Figure 1.

Normal Rules (`main`)
$\langle\!\langle \mathtt{u}, 24 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{u}, 25 \rangle\!\rangle$
$\langle\!\langle \mathtt{u}, 24 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v}, 25 \rangle\!\rangle$
$\langle\!\langle \mathtt{u}, 25 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{u}, 26 \rangle\!\rangle$

Normal Rules (`foo`)
$\langle\!\langle \mathtt{a}, 28 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{a}, 29 \rangle\!\rangle$
$\langle\!\langle \mathtt{a}, 29 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{a}, 30 \rangle\!\rangle$
$\langle\!\langle \mathtt{a}, 29 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{a}, 31 \rangle\!\rangle$
$\langle\!\langle \mathtt{a}, 32 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{a}, 33 \rangle\!\rangle$
$\langle\!\langle \mathtt{b}, 32 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{b}, 33 \rangle\!\rangle$

Push Rules
$\langle\!\langle \mathtt{v}, 25 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{a}, 28 \cdot 26 \rangle\!\rangle$
$\langle\!\langle \mathtt{a}, 31 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{a}, 28 \cdot 32 \rangle\!\rangle$

Pop Rules
$\langle\!\langle \mathtt{a}, 30 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{a}, \epsilon \rangle\!\rangle$
$\langle\!\langle \mathtt{a}, 30 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v}, \epsilon \rangle\!\rangle$
$\langle\!\langle \mathtt{a}, 33 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{a}, \epsilon \rangle\!\rangle$
$\langle\!\langle \mathtt{a}, 33 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v}, \epsilon \rangle\!\rangle$
$\langle\!\langle \mathtt{b}, 33 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{w}, \epsilon \rangle\!\rangle$
$\langle\!\langle \mathtt{b}, 33 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{b}, \epsilon \rangle\!\rangle$

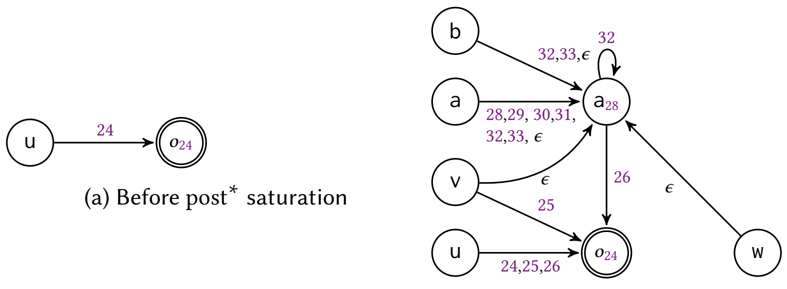
We start by explaining the normal rules. The program's control-flow graph has an edge from statement 24 to 25, because the latter is a control-flow successor of the former. $\mathcal{P}_{\mathbb{S}}$ has two normal rules that match this control-flow edge: $\langle\!\langle \mathtt{u}, 24 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{u}, 25 \rangle\!\rangle$ and $\langle\!\langle \mathtt{u}, 24 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v}, 25 \rangle\!\rangle$. For both rules, the control location of their start configuration is the variable `u`. Both rules describe the flow of the statement in line 25 with respect to that variable. One rule states that the statement transfers data from `u` to `v`, while the other encodes the data-flow that remains within `u`. The other normal rules are straightforward, because they merely follow the control-flow of the program. We visualize the normal rules in the form of a data-flow graph next to the code in Figure 1. For each normal rule, the (start or target) configuration corresponds to a node in the graph, and nodes are plotted in the form of a grid. The configuration's variable can be extracted from the column header. Nodes are drawn between two statements as the associated data-flow holds after a statement (and before the next). Therefore, a rule corresponds to an edge within the graph.

Table 2 additionally lists two push and seven pop rules relevant to the example. The push rule $\langle\!\langle \mathtt{v}, 25 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{a}, 28 \cdot 26 \rangle\!\rangle$ maps the argument variable `v` at the call site in line 26 to the parameter variable `a` of the callee `foo()`. That rule modifies the stack in the following way: it replaces the top most element of the stack (25, the predecessor of the call site in line 26) by the first statement of the called method (line 28). Additionally, the rule pushes the the call site (line 26) onto the stack. The semantics of the push is as follows: when data-flow reaches variable `v` at line 25, the flow continues at the statement in line 28. Whenever a statement is popped from the stack, the flow continues to hold after the call site in line 26.

The pop rules map data-flow from callee to caller. For example, the rule $\langle\!\langle \mathtt{a}, 30 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{b}, \epsilon \rangle\!\rangle$ captures variable `a` returning from the statement in line 30 to `b` at the call site 32.

The rules define a transition relation $\Rightarrow$ between configurations of $\mathcal{P}$: If $\langle\!\langle p, \gamma \rangle\!\rangle \rightarrow \langle\!\langle p', w \rangle\!\rangle$, then $\langle\!\langle p, \gamma w' \rangle\!\rangle \Rightarrow \langle\!\langle p', ww' \rangle\!\rangle$ for all $w' \in \Gamma^*$. Taking the transitive closure of $\Rightarrow$ (denoted by $\Rightarrow^*$) from a starting configuration c constructs a set of reachable configurations called $post^*(c) = \{c' \mid c \Rightarrow^* c'\}$. The set can potentially be infinite, however, the set of configurations is regular and it can be finitely represented by a finite automaton.

Definition 2. *Given a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$, a $\mathcal{P}$-*automaton *is a finite non-deterministic automaton $\mathcal{A} = (Q, \Gamma, \delta, P, F)$ where $Q \supseteq P$ is a finite set of* states, *$\delta \subseteq Q \times \Gamma \times Q$ is the set of transitions and $F \subseteq Q$ are the* final states. *The* initial states *are all control locations P of the pushdown system $\mathcal{P}$. A configuration $\langle\!\langle p, w \rangle\!\rangle$ is* accepted *by $\mathcal{A}$, if the automaton contains a path from state p to some*



(a) Before post* saturation

(b) After post* saturation

Fig. 2. The automaton $\mathcal{A}_{\mathbb{S}}$ before and after post* saturation with the $\mathcal{P}_{\mathbb{S}}$

final state $q \in Q$ such that the word along the path is equal to w. We write $\langle\!\langle p, w \rangle\!\rangle \in \mathcal{A}$ for an accepted configuration.

The $\mathcal{P}$-automaton $\mathcal{A}$ of a pushdown system $\mathcal{P}$ encodes the set $post^*(c)$. There is a standard algorithm [Bouajjani et al. 1997; Esparza et al. 2000; Reps et al. 2007] to compute the automaton efficiently. We refer to this algorithm as post*. The algorithm post* takes as input a $\mathcal{P}$-automaton that accepts the initial configuration c, and applies a saturation process to the automaton. The algorithm saturates the automaton with transitions, i.e., depending on the pushdown system new transitions are added to the automaton until a fixed-point is reached. Upon termination of post*, the automaton represents the set $post^*(c)$.

The $\mathcal{P}$-automaton for the pushdown system $\mathcal{P}_{\mathbb{S}}$ is an automaton whose nodes are variables[4] $x \in \mathbb{V}$ and its transitions are labeled by statements $s \in \mathbb{S}$. Throughout the paper we refer to this automaton as $\mathcal{A}_{\mathbb{S}}$.

Algorithm post* can be consulted to compute data-flow reachability. When $\mathcal{A}_{\mathbb{S}}$ initially accepts a variable $v \in \mathbb{V}$ with a label $s \in \mathbb{S}$, after saturation with post*, $\mathcal{A}_{\mathbb{S}}$ lists all configurations encoding the same variables containing the same data as v at statement s.

Example 2.2. We saturate the $\mathcal{A}_{\mathbb{S}}$ in Figure 2a with the $\mathcal{P}_{\mathbb{S}}$ of Table 2. $\mathcal{A}_{\mathbb{S}}$ initially accepts the configuration $\langle\!\langle \mathsf{u}, 24 \rangle\!\rangle$, which means $\mathcal{A}_{\mathbb{S}}$ tracks the variables that the object allocated in 24 flows to. Therefore, the accepting state of the automaton is labeled by o_{24}, which refers to the object created at this allocation site.

Figure 2b depicts the final post* saturated $\mathcal{A}_{\mathbb{S}}$. The semantics of this automaton is as follows: any variable of any configuration that is accepted within the automaton points-to the object allocated in 24. To keep the figure concise, we visualize transitions between the same states (but with different labels) as a single transition and separate their labels by commas.

We explain the generation of the automaton based on the application of the different rules.

Normal Rules: The configuration that $\mathcal{A}_{\mathbb{S}}$ in Figure 2a initially accepts is the same as the start configuration of the two normal rules $\langle\!\langle \mathsf{u}, 24 \rangle\!\rangle \to \langle\!\langle \mathsf{u}, 25 \rangle\!\rangle$ and $\langle\!\langle \mathsf{u}, 24 \rangle\!\rangle \to \langle\!\langle \mathsf{v}, 25 \rangle\!\rangle$. Since $\mathcal{A}_{\mathbb{S}}$ accepts the rule's start configuration, post* adds a transition such that the target of the rule is also accepted. Therefore, transitions $\mathsf{u} \xrightarrow{25} o_{24}$ and $\mathsf{v} \xrightarrow{25} o_{24}$ are added to $\mathcal{A}_{\mathbb{S}}$. As the start configuration of the rule $\langle\!\langle \mathsf{u}, 25 \rangle\!\rangle \to \langle\!\langle \mathsf{u}, 26 \rangle\!\rangle$ is now also an accepted configuration, the transition $\mathsf{u} \xrightarrow{26} o_{24}$ is added. All edges out of u and v (except the one with label ϵ) are added transitively due to the normal rules in

[4]Except for some intermediate nodes generated by push rules.

`main`. These normal rules propagate data-flow intraprocedurally to the variables at the respective statements of `main`.

Push Rules: $\mathcal{A}_{\mathbb{S}}$ accepts the start configuration of the push rule $\langle\!\langle \texttt{v}, 25\rangle\!\rangle \to \langle\!\langle \texttt{a}, 28 \cdot 26\rangle\!\rangle$. Push rules are treated separately by post* and add two transitions to $\mathcal{A}_{\mathbb{S}}$. For the latter rule, post* adds the two transitions $\texttt{a} \xrightarrow{28} \texttt{a}_{28}$ and $\texttt{a}_{28} \xrightarrow{26} o_{24}$ to render $\langle\!\langle \texttt{a}, 28 \cdot 26\rangle\!\rangle$ accepting. Note, post* introduces the intermediate node $\texttt{a}_{28}$ to $\mathcal{A}_{\mathbb{S}}$ as this state did not exist. The accepting configuration $\langle\!\langle \texttt{a}, 28 \cdot 26\rangle\!\rangle$ carries the semantic that `a` at statement 28 points to o_{24} under calling context 26. Now that $\mathcal{A}_{\mathbb{S}}$ accepts configuration $\langle\!\langle \texttt{a}, 28 \cdot 26\rangle\!\rangle$ the normal rule $\langle\!\langle \texttt{a}, 28\rangle\!\rangle \to \langle\!\langle \texttt{a}, 29\rangle\!\rangle$ is applicable and the data-flow of object o_{24} within method `foo()` is computed.

Method `foo()` recursively calls itself in line 32 and $\mathcal{P}_{\mathbb{S}}$ lists a second push rule, $\langle\!\langle \texttt{a}, 31\rangle\!\rangle \to \langle\!\langle \texttt{a}, 28{\cdot}32\rangle\!\rangle$ that describing that call. The start configuration of the rule is accepted as of transition $\texttt{a} \xrightarrow{31} \texttt{a}_{28}$. Application of the push rule adds the self loop transition $\texttt{a}_{28} \xrightarrow{32} \texttt{a}_{28}$. The latter transition introduces a loop within $\mathcal{A}_{\mathbb{S}}$, and it accepts an infinite set of configurations. $\mathcal{A}_{\mathbb{S}}$ accepts any configuration of the form $\langle\!\langle \texttt{a}, 28 \cdot 32 \cdot \ldots \cdot 32 \cdot 26\rangle\!\rangle$. The recursion of `foo()` is reflected in the infinitely growing calling contexts.

Pop Rules: Exemplary for the other pop rules, we discuss the application of the rule $\langle\!\langle \texttt{b}, 33\rangle\!\rangle \to \langle\!\langle \texttt{w}, \epsilon\rangle\!\rangle$. $\mathcal{A}_{\mathbb{S}}$ has a transition out of `b` with label 33 that matches the start configuration of the rule. The transition's target state is $\texttt{a}_{28}$, and post* adds the ϵ-transition $\texttt{w} \xrightarrow{\epsilon} \texttt{a}_{28}$. State $\texttt{a}_{28}$ is accepted with label 26 and therefore state `w` is also accepted. After the application of the pop rule, configuration $\langle\!\langle \texttt{w}, 26\rangle\!\rangle$ is accepted, and algorithm post* pops one stack element from the stack. This stack encodes the calling context and makes the data-flow analysis context-sensitive as it only considers *(same-level) realizable* paths [Reps et al. 1995].

3 FIELD PUSHDOWN SYSTEM

Section 2 describes the pushdown-system framework and an instance of it: the *call-PDS* ($\mathcal{P}_{\mathbb{S}}$) which solves a data-flow problem in a context-sensitive and flow-sensitive manner. In this section we present the first contribution of our paper: we show how a field- and flow-sensitive data-flow analysis also can be formulated as a pushdown system.

For that, we introduce another pushdown system, the "*field-PDS*" $\mathcal{P}_{\mathbb{F}}$, whose stack elements are drawn from $\mathbb{F}$, the set of all fields in the analyzed program. The corresponding $\mathcal{P}-$automaton $\mathcal{A}_{\mathbb{F}}$ is a concise and finite representation of the (potentially infinitely many) access paths the analysis propagates. Opposed to standard access-path based approaches that require one access path per statement (and per context), $\mathcal{A}_{\mathbb{F}}$ maintains all access paths at all statements in a single automaton.

In the following, we first provide a formal definition of the system $\mathcal{P}_{\mathbb{F}}$, then we provide examples.

Definition 3. *The* pushdown system of fields *is the pushdown system* $\mathcal{P}_{\mathbb{F}} = (\mathbb{V} \times \mathbb{S}, \mathbb{F}, \Delta_{\mathbb{F}})$. *A control location of this system is a pair of a variable and a statement. We use* $x@s$ *for an element* $(x, s) \in \mathbb{V} \times \mathbb{S}$. *The notation emphasizes the fact that* x *holds* at *statement* s. *The pushdown system pushes and pops elements of* $\mathbb{F}$ *to and from the stack. The stack may also be empty, which is represented by the* ϵ *field.*

The configurations of $\mathcal{P}_{\mathbb{S}}$ are pairs of $\mathbb{V} \times \mathbb{S}^*$, so a configuration of $\mathcal{P}_{\mathbb{F}}$ is an element of $(\mathbb{V} \times \mathbb{S}) \times \mathbb{F}^*$. Both systems are flow-sensitive, but they achieve it in different manners. $\mathcal{P}_{\mathbb{F}}$ models flow-sensitivity by encoding the control-flow within the control location, while $\mathcal{P}_{\mathbb{S}}$ models control-flow within the stack. We write a configuration of $\mathcal{P}_{\mathbb{F}}$ as $\langle\!\langle x@s, f_0 \cdot f_1 \cdot \ldots f_n\rangle\!\rangle$ and it can be read as follows: After statement s, the analysis tracks data flow within the access path $x.f_0 \cdot f_1 \cdot \ldots f_n$. In the following,

Table 3. The function *normalFieldFlow* for $\mathcal{P}_{\mathbb{F}}$. Within the comment column t refers to the input statement.

Variable ($\mathbb{V}$)	**Statement** ($\mathbb{S}$)	**Out** ($\wp(\mathbb{V})$)	**Type**	**Comment**
x	$x \leftarrow *$	$\varnothing$	intra	kill info on x at t
y	$x \leftarrow y$	$\{x, y\}$	intra	y copied to x at t
y	$x.f \leftarrow y$	$\{y\}$	intra	info on y is retained at t
y	$x \leftarrow y.f$	$\{y\}$	intra	info on y is retained at t
p_i	$m(p_1, p_2, \ldots, p_n)$	$\{q_i\}$	inter	p_i copied to formal q_i
q_i	**return**	$\{p_i\}$	inter	q_i copied to actual p_i
x	**return** x	$\{y\}$	inter	x copied to assigned value y

we construct the set of rules $\Delta_{\mathbb{F}}$ as the disjoint union of the sets of normal ($\Delta_{\mathbb{F}}^{normal}$), push ($\Delta_{\mathbb{F}}^{push}$), and pop rules ($\Delta_{\mathbb{F}}^{pop}$).

3.1 Normal Rules ($\Delta_{\mathbb{F}}^{normal}$)

$\mathcal{P}_{\mathbb{F}}$ pushes and pops fields to and from its stack. Unlike $\mathcal{P}_{\mathbb{S}}$, here the statements that push and pop the fields are not call and return statements but store and load statements. All other statements maintain the field stack unchanged and constitute normal rules of $\mathcal{P}_{\mathbb{F}}$.

We construct the normal rules by the help of the function *normalFieldFlow*. The function maps from $\mathbb{V} \times \mathbb{S}$ to $\wp(\mathbb{V})$. Table 3 lists the *normalFieldFlow* function. The first two columns describe the inputs, while the third column contains the respective output set $O \subseteq \mathbb{V}$ of the function.

Assume $\mathcal{P}_{\mathbb{F}}$ accepts a configuration $\langle\!\langle x@s, g_0 \cdot \ldots \cdot g_n \rangle\!\rangle$ and let t be an intraprocedural control-flow successor of s. Assume further $y \in normalFieldFlow(x, t)$, then $\mathcal{P}_{\mathbb{F}}$ has a rule:

$$\langle\!\langle x@s, g_0 \rangle\!\rangle \rightarrow \langle\!\langle y@\tilde{t}, g_0 \rangle\!\rangle \in \Delta_{\mathbb{F}}^{normal}.$$

In the formula it is $\tilde{t} = t$, unless t is a call site or a return statement (cases for which Table 3 lists inter as type). If t is a call site, $\tilde{t}$ is the first statement of the callee. For a return statement t of a method m, $\tilde{t}$ is defined as the call site calling m.

The start configuration and the target configuration of the rules have the same field as the stack location. Therefore, none of these rules add or remove an element from the stack.

As $\mathcal{P}_{\mathbb{S}}$, also $\mathcal{P}_{\mathbb{F}}$ kills data-flows at assignment statements. The first row of Table 3 is equivalent to the formulation in $\mathcal{P}_{\mathbb{S}}$. Function *normalFieldFlow* returns $\varnothing$ for a data-flow x reaching a statement $x \leftarrow *$. For an assignment statement $t{:}\, x \leftarrow y$, the field-PDS generates the normal rules $\langle\!\langle y@s, g_0 \rangle\!\rangle \rightarrow \langle\!\langle y@t, g_0 \rangle\!\rangle$ and $\langle\!\langle y@s, g_0 \rangle\!\rangle \rightarrow \langle\!\langle x@t, g_0 \rangle\!\rangle$. At the successor statement t of s, the data is reachable via variables x and y. Field load and store statements differ from the handling in $\mathcal{P}_{\mathbb{S}}$. These statements do not generate additional normal rules for $\mathcal{P}_{\mathbb{F}}$, but purely add *identity* normal rules. When a fact y holds before the field-store statement $t{:}\, x.f \leftarrow y$, the *normal* rule propagates y simply as $y@t$ to hold after the statement. The handling of the actual assignment to $x.f$ is performed by an additional push rule that we describe below.

The inter-procedural data-flows map to normal rules in $\mathcal{P}_{\mathbb{F}}$. A parameter p_i reaching the call site $m(p_1, p_2, \ldots, p_n)$ is mapped to the $q_i@\tilde{t}$ which refers to the i-th formal parameter at the first statement $\tilde{t}$ of the callee method. At return statements, the parameters q_i are also mapped back to the corresponding arguments p_i at the call site $\tilde{t}$ as defined in the last row of Table 3. Hereby, $\mathcal{P}_{\mathbb{F}}$ is defined to propagate the data-flow at return statements of m to all call sites of m, and, opposed to

```
35 foo(u, v, w){
36   v.f = u;
37   x = w.f;
38   if(...){
39     w.g = v;
40   } else {
41     w.h = v;
42   }
43 }
```

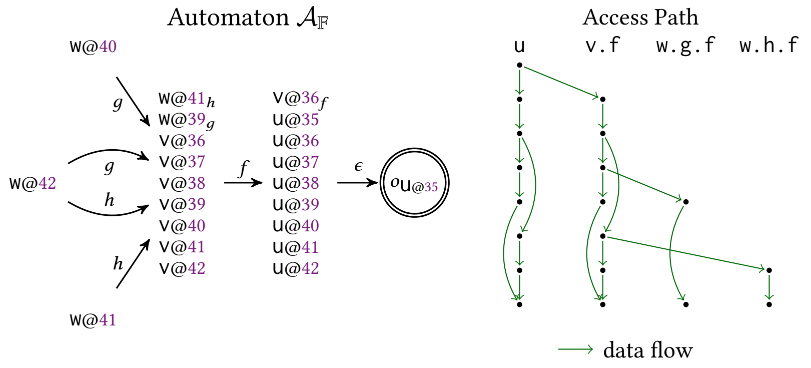


Fig. 3. A post*-saturated $\mathcal{A}_{\mathbb{F}}$ when initialized with the configuration $\langle\langle$u@35, $\epsilon\rangle\rangle$ and saturated with $\mathcal{P}_{\mathbb{F}}$ listed in Table 4. Next to it, the same information represented as standard data-flow graph with an access-path domain.

$\mathcal{P}_{\mathbb{S}}$, $\mathcal{P}_{\mathbb{F}}$ is context-insensitive. A similar rule also handles the return value, which we omit from the table for brevity.

3.2 Push Rules ($\Delta_{\mathbb{F}}^{push}$)

When a configuration $\langle\langle y@s, g_0 \cdot \ldots \cdot g_n\rangle\rangle$ is accepted by $\mathcal{A}_{\mathbb{F}}$ and some successor t of s is a field-store statement, i.e., $t{:}\, x.f \leftarrow y$, $\mathcal{P}_{\mathbb{F}}$ lists the normal rule $\langle\langle y@s, g_0\rangle\rangle \to \langle\langle y@t, g_0\rangle\rangle$. In addition to this normal rule, $\mathcal{P}_{\mathbb{F}}$ lists the following push rule for a field-store statement which pushes a field element onto the stack: $\langle\langle y@s, g_0\rangle\rangle \to \langle\langle x@t, f \cdot g_0\rangle\rangle \in \Delta_{\mathbb{F}}^{push}$. The push rule has the following semantic meaning: Whenever the configuration $\langle\langle y@s, g_0 \cdot g_1 \cdot \ldots \cdot g_n\rangle\rangle$ turns accepting during the computation of post*, the configuration $\langle\langle x@t, f \cdot g_0 \cdot g_1 \cdot \ldots \cdot g_n\rangle\rangle$ is marked as accepting as well. Speaking in terms of an access path: If an access path $y.g_0 \cdot g_1 \cdot \ldots \cdot g_n$ holds before statement t, the access path $x.f \cdot g_0 \cdot g_1 \cdot \ldots \cdot g_n$ holds after statement t, i.e., the field f is prepended to the access path.

3.3 Pop Rules ($\Delta_{\mathbb{F}}^{pop}$)

The pop rules correspond to the runtime semantics of a field-load statement. For an accepting configuration $\langle\langle y@s, f \cdot g_0 \cdot \ldots \cdot g_n\rangle\rangle$ where the successor statement t of s is a load statement $x \leftarrow y.f$, $\mathcal{P}_{\mathbb{F}}$ removes a stack element from the field stack, spoken in the form of its rules, the system has a pop rule: $\langle\langle y@s, f\rangle\rangle \to \langle\langle x@t, \epsilon\rangle\rangle \in \Delta_{\mathbb{F}}^{pop}$.

An accepting configuration $\langle\langle y@s, f \cdot g_0 \cdot \ldots \cdot g_n\rangle\rangle$ induces the configuration $\langle\langle x@t, g_0 \cdot \ldots \cdot g_n\rangle\rangle$. In other terms, when the access path $y.f \cdot g_0 \cdot \ldots \cdot g_n$ holds before t, the analysis continues to propagate the data flow $x.g_0 \cdot \ldots \cdot g_n$ after statement t.

Example 3.1. Figure 3 shows an example program code with three field-store and one field-load statements. $\mathcal{P}_{\mathbb{F}}$ modeling the code's data-flow is shown in the form of the rule set $\Delta_{\mathbb{F}}$ in Table 4. Table 4 lists normal, push, and pop rules for the data-flows in method `foo()`. For example, the normal rule $\langle\langle$u@36, $*\rangle\rangle \to \langle\langle$u@37, $*\rangle\rangle$ encodes that data flows from u@36 to u@37. The kleene-star ($*$) at the stack location of the rule is a wildcard that can be replaced by any field $g \in \mathbb{F}$, i.e., the representation actually bundles multiple rules. The semantics of the rule is that any data stored in any field dereferenced from u at statement 36 is propagated to the successor statement 37, because statement 36 does not modify u.

The rule set $\Delta_{\mathbb{F}}$ contains three push rules, each of which match a field-store statement. For example the push rule $\langle\langle$u@35, $*\rangle\rangle \to \langle\langle$v@36, f $\cdot$ $*\rangle\rangle$ encodes that any data stored in u@35 flows

Table 4. The rule set $\Delta_{\mathbb{F}}$ of $\mathcal{P}_{\mathbb{F}}$ for the code shown in Figure 3.

Normal Rules	
$\langle\langle \mathtt{u}@35, * \rangle\rangle \to \langle\langle \mathtt{u}@36, * \rangle\rangle$	$\langle\langle \mathtt{v}@37, * \rangle\rangle \to \langle\langle \mathtt{v}@38, * \rangle\rangle$
$\langle\langle \mathtt{u}@36, * \rangle\rangle \to \langle\langle \mathtt{u}@37, * \rangle\rangle$	$\langle\langle \mathtt{v}@38, * \rangle\rangle \to \langle\langle \mathtt{v}@39, * \rangle\rangle$
$\langle\langle \mathtt{u}@37, * \rangle\rangle \to \langle\langle \mathtt{u}@38, * \rangle\rangle$	$\langle\langle \mathtt{v}@39, * \rangle\rangle \to \langle\langle \mathtt{v}@42, * \rangle\rangle$
$\langle\langle \mathtt{u}@38, * \rangle\rangle \to \langle\langle \mathtt{u}@39, * \rangle\rangle$	$\langle\langle \mathtt{v}@38, * \rangle\rangle \to \langle\langle \mathtt{v}@40, * \rangle\rangle$
$\langle\langle \mathtt{u}@39, * \rangle\rangle \to \langle\langle \mathtt{u}@42, * \rangle\rangle$	$\langle\langle \mathtt{v}@40, * \rangle\rangle \to \langle\langle \mathtt{v}@41, * \rangle\rangle$
$\langle\langle \mathtt{u}@38, * \rangle\rangle \to \langle\langle \mathtt{u}@40, * \rangle\rangle$	$\langle\langle \mathtt{v}@41, * \rangle\rangle \to \langle\langle \mathtt{v}@42, * \rangle\rangle$
$\langle\langle \mathtt{u}@40, * \rangle\rangle \to \langle\langle \mathtt{u}@41, * \rangle\rangle$	$\langle\langle \mathtt{w}@39, * \rangle\rangle \to \langle\langle \mathtt{w}@42, * \rangle\rangle$
$\langle\langle \mathtt{u}@41, * \rangle\rangle \to \langle\langle \mathtt{u}@42, * \rangle\rangle$	$\langle\langle \mathtt{w}@41, * \rangle\rangle \to \langle\langle \mathtt{w}@42, * \rangle\rangle$
$\langle\langle \mathtt{v}@36, * \rangle\rangle \to \langle\langle \mathtt{v}@37, * \rangle\rangle$	

Push Rules
$\langle\langle \mathtt{u}@35, * \rangle\rangle \to \langle\langle \mathtt{v}@36, \mathtt{f} \cdot * \rangle\rangle$
$\langle\langle \mathtt{v}@38, * \rangle\rangle \to \langle\langle \mathtt{w}@39, \mathtt{g} \cdot * \rangle\rangle$
$\langle\langle \mathtt{v}@40, * \rangle\rangle \to \langle\langle \mathtt{w}@41, \mathtt{h} \cdot * \rangle\rangle$

Pop Rules
$\langle\langle \mathtt{w}@36, \mathtt{f} \rangle\rangle \to \langle\langle \mathtt{x}@37, \epsilon \rangle\rangle$

to v@36 at the same time pushing f to the top of the stack. We also use the kleene-star notation, because the field f is pushed, no matter which field is on the stack.

Each field-load statement matches a pop rule. The presented $\mathcal{P}_{\mathbb{F}}$ lists the pop rule $\langle\langle \mathtt{w}@36, \mathtt{f} \rangle\rangle \to \langle\langle \mathtt{x}@37, \epsilon \rangle\rangle$. When variable w reaches statement 36, i.e., w@36 is propagated, and the stack topmost element is the field f (the tracked data is stored at least below field f), the data-flow continues to x@37, and field f is popped from the stack.

Based on $\mathcal{P}_{\mathbb{F}}$, algorithm post* can answer reachability queries over the system described in Table 4. The resulting post*-saturated $\mathcal{P}$-automaton, which we refer to by $\mathcal{A}_{\mathbb{F}}$, contains field-sensitive and flow-sensitive data-flow results[5]. We assume $\mathcal{A}_{\mathbb{F}}$ to initially contain the transition $\mathtt{u}@35 \xrightarrow{\epsilon} o_{\mathtt{u}@35}$. We label the accepting state of the automaton by $o_{\mathtt{u}@35}$, because it refers to the abstract object stored in variable u at the beginning of method foo().

The transition labels of $\mathcal{A}_{\mathbb{F}}$ are elements of $\mathbb{F}$, i.e., fields of the program. The abstract object $o_{\mathtt{u}@35}$ is stored inside field f of variable v at the statement in line 36. The code then branches and in line 39, variable v is stored inside field g of some object pointed-to by w, line 41 stores variable v to field h of w. Therefore, at statement 42, the abstract object $o_{\mathtt{u}@35}$ is transitively accessible either via access path w.g.f or via w.h.f. $\mathcal{A}_{\mathbb{F}}$ encodes this information as it accepts the two words[6] $\mathtt{g} \cdot \mathtt{f} \cdot \epsilon$ and $\mathtt{h} \cdot \mathtt{f} \cdot \epsilon$ starting from node w@42.

Next to the $\mathcal{P}$-automaton, Figure 3 also shows the same data-flow analysis but encoded in a data-flow graph for an access-path based analysis. The example code does not contain a loop and only finitely many access paths are generated. Therefore, both representations encode the same information, and there is a unique transformation between the two. For instance, $\mathcal{A}_{\mathbb{F}}$ accepts configuration $\langle\langle \mathtt{w}@42, g \cdot f \cdot \epsilon \rangle\rangle$. This configuration corresponds to the node with label w.g.f for statement 42 in the access-path based representation on the right.

However, $\mathcal{A}_{\mathbb{F}}$ encodes the same information more concisely. The access-path representation requires an explicit enumeration of the fields, w.g.f and w.h.f are encoded individually. Opposed to that $\mathcal{A}_{\mathbb{F}}$ shares the information that prior to the branch the data-flow is stored in field f. $\mathcal{A}_{\mathbb{F}}$ only needs to store the two transitions labeled g and h out of w@42. The outgoing transition of the target node labeled by f encodes the remaining field of *both* the two access paths w.g.f and w.h.f. The automaton $\mathcal{A}_{\mathbb{F}}$ concisely merges the information sharable between multiple data-flow

[5]For a simpler representation of the automaton, we merged states of transitions with the same field label of the automaton.
[6]An *accepted word* $w = w_1 \cdot w_2 \cdot \dots \cdot w_n \in \mathbb{F}^*$ of $\mathcal{A}_{\mathbb{F}}$ is a path from a some node to the accepting state such that the concatenated transition labels form w.

```
44 foo(a){
45   while(...){
46     b = new B();
47     b.f = a
48     a = b;
49   }
50 }
```

$$\langle\!\langle \texttt{a@46}, * \rangle\!\rangle \rightarrow \langle\!\langle \texttt{b@47}, \texttt{f} \cdot * \rangle\!\rangle$$

$$\langle\!\langle \texttt{a@48}, * \rangle\!\rangle \rightarrow \langle\!\langle \texttt{a@45}, * \rangle\!\rangle$$

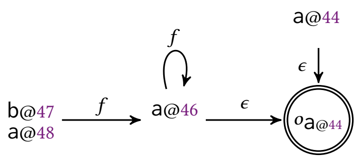


Fig. 4. $\mathcal{P}_{\mathbb{F}}$ and its finite representation of an infinite set of access paths.

paths. It follows that the more branched field-store statements the analyzed code contains, the more efficient the automaton representation is.

In Example 3.1, we have seen how $\mathcal{P}_{\mathbb{F}}$ carries the same information as a data-flow analysis based on access paths. $\mathcal{P}_{\mathbb{F}}$ can, additionally, represent an infinite state space in a finite representation by encoding the access paths in its automaton $\mathcal{A}_{\mathbb{F}}$. Since $\mathcal{A}_{\mathbb{F}}$ represents infinitely many access paths, a transformation into a data-flow graph with finitely many nodes is not possible.

Example 3.2. Figure 4 shows a reduced version of the code excerpt from `TreeMap` in Listing 1, for which a data-flow analysis generates infinitely many access paths. The reduced excerpt also generates infinitely many access path, as shown in the figure. We show how $\mathcal{P}_{\mathbb{F}}$ encodes the infinite number of access paths within a concise $\mathcal{P}$-automaton. On the other hand, an access-path based analysis without k-limiting generates arbitrarily long sequences of access paths: `a.f`, `a.f.f`, `a.f.f.f`, . . .

Figure 4 lists a subset of the rules of $\mathcal{P}_{\mathbb{F}}$, and next to it, the relevant transitions of $\mathcal{A}_{\mathbb{F}}$ that post* generates when tracing the abstract object $o_{\texttt{a@44}}$. Initially, $\mathcal{A}_{\mathbb{F}}$ accepts the configuration $\langle\!\langle \texttt{a@44}, \epsilon \rangle\!\rangle$. Between the statements from line 44 to line 46 variable `a` is not overwritten and configuration $\langle\!\langle \texttt{a@46}, \epsilon \rangle\!\rangle$ becomes accepting. Next, the push rule $\langle\!\langle \texttt{a@46}, * \rangle\!\rangle \rightarrow \langle\!\langle \texttt{b@47}, \texttt{f} \cdot * \rangle\!\rangle$ is applied and yields the accepting configuration $\langle\!\langle \texttt{b@47}, f \cdot \epsilon \rangle\!\rangle$. The statement in line 48 does not overwrite `b` and the configuration flows transitively back to `a@46`, because the rule $\langle\!\langle \texttt{a@48}, * \rangle\!\rangle \rightarrow \langle\!\langle \texttt{a@45}, * \rangle\!\rangle$ encodes a control-flow backward edge from the end of the loop to the entry of the loop in line 45. The normal rules propagate the data-flow back to `a@46` where post* algorithm inserts the self-loop edge with label f for state `a@46`.

Once post* saturates $\mathcal{A}_{\mathbb{F}}$, it encodes all sequences of possible access paths. When required, these can be extracted from $\mathcal{A}_{\mathbb{F}}$ in the form of a regular expression. Assume we would like to know how `a@44` is accessible in line 48 from variable `b`, i.e. from the node `b@48`. Consider this node the initial state of the automaton, then all path(s) to the accepting state are covered by the regular expression `(f)+`. The data may be stored in any access path with base variable `b` and arbitrarily many field accesses `f`.

This example shows that $\mathcal{P}_{\mathbb{F}}$ has a clear benefit over the a k-limited access path-based domain, because it can easily represent infinitely many access paths without introducing imprecisions due to over-approximation.

4 SYNCHRONIZED PUSHDOWN SYSTEMS

In Section 2 we discussed the call-PDS $\mathcal{P}_{\mathbb{S}}$ and in Section 3 the field-PDS $\mathcal{P}_{\mathbb{F}}$. $\mathcal{P}_{\mathbb{S}}$ solves context-sensitive and flow-sensitive data-flow analysis, and $\mathcal{P}_{\mathbb{F}}$ solves flow-sensitive and field-sensitive analysis. However, $\mathcal{P}_{\mathbb{S}}$ is field-*in*sensitive, and $\mathcal{P}_{\mathbb{F}}$ is context-*in*sensitive, i.e., each system has a precision advantage but also disadvantage over the other.

```
51 bar(u, v){
52   v.h = u;               57 foo(p){
53   w = foo(v);            58   q.g = p;
54   x = w.g;               59   return q;
55   y = x.f;               60 }
56 }
```

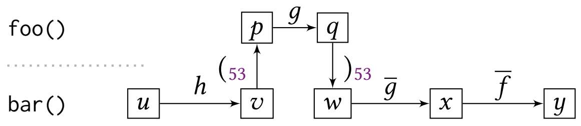


Fig. 5. A code snippet and a labeled graph representation of the code.

This section presents the second contribution of this paper. We combine the results of $\mathcal{P}_{\mathbb{S}}$ and $\mathcal{P}_{\mathbb{F}}$ to construct *one* analysis that combines the precision benefit of each system. Intuitively, a configuration of the more precise analysis is accepted only if $\mathcal{A}_{\mathbb{F}}$ *and* $\mathcal{A}_{\mathbb{S}}$ accept the configuration.

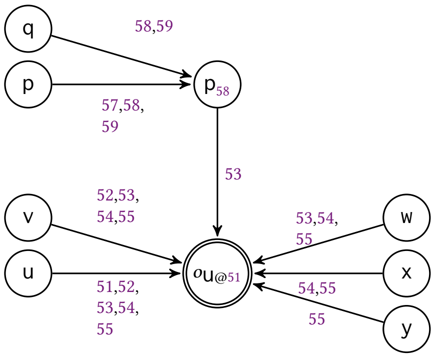
Definition 4. *For the call-PDS $\mathcal{P}_{\mathbb{S}} = (\mathbb{V}, \mathbb{S}, \Delta_{\mathbb{S}})$ and the field-PDS $\mathcal{P}_{\mathbb{F}} = (\mathbb{V} \times \mathbb{S}, \mathbb{F} \cup \{\epsilon\}, \Delta_{\mathbb{F}})$,* synchronized pushdown systems *are a quintuple SPDS = $(\mathbb{V}, \mathbb{S}, \mathbb{F} \cup \{\epsilon\}, \Delta_{\mathbb{F}}, \Delta_{\mathbb{S}})$. A configuration of the SPDS extends from the configuration of each system: A* synchronized configuration *is a triple $(x, s, f) \in \mathbb{V} \times \mathbb{S}^+ \times \mathbb{F}^*$, which we denote as $\langle\!\langle v.f_1 \cdot \ldots \cdot f_m @ s_0^{s_1 \cdot \ldots \cdot s_n} \rangle\!\rangle$ where $s = s_0 \cdot s_1 \cdot \ldots \cdot s_n$ and $f = f_1 \cdot \ldots f_m$. For synchronized pushdown systems we define the set of all reachable synchronized configurations from a start configuration $c = \langle\!\langle v.f_1 \cdot \ldots \cdot f_m @ s_0^{s_1 \cdot \ldots \cdot s_n} \rangle\!\rangle$ to be*

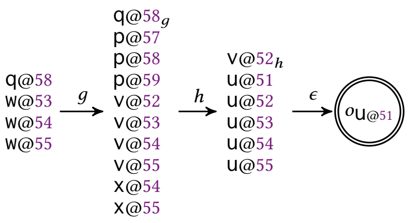
$$post_{\mathbb{S}}^{\mathbb{F}}(c) = \{\langle\!\langle w.g @ t_0^{t_1 \cdot \ldots \cdot t_n} \rangle\!\rangle \mid \langle\!\langle w @ t_0, g \rangle\!\rangle \in post_{\mathbb{F}}^*(\langle\!\langle v @ s_0, f \rangle\!\rangle) \\ \wedge \langle\!\langle w, t \rangle\!\rangle \in post_{\mathbb{S}}^*(\langle\!\langle v, s \rangle\!\rangle)\}.$$

Hence, a synchronized configuration c is accepted if $\langle\!\langle v, s_0 \cdot \ldots \cdot s_n \rangle\!\rangle \in \mathcal{A}_{\mathbb{S}}$ and $\langle\!\langle v @ s_0, f_1 \cdot \ldots \cdot f_m \rangle\!\rangle \in \mathcal{A}_{\mathbb{F}}$ and $post_{\mathbb{S}}^{\mathbb{F}}(c)$ can be represented by the automaton pair $(\mathcal{A}_{\mathbb{S}}, \mathcal{A}_{\mathbb{F}})$, which we refer to as $\mathcal{A}_{\mathbb{S}}^{\mathbb{F}}$.

Example 4.1. Figure 5 shows a code snippet where data flows inter-procedurally and is stored and loaded into a field of an object. Below the snippet, we depict a graph representation of the code to help illustrate the data flow throughout the code. The nodes of the graph represent program variables; horizontal edges between them correspond to field push and pop rules. The edges are labeled with the names of the field. A field label with a line on top, e.g., $\overline{f}$, means the field f is loaded (a pop rule), for field-stores the field is not overlined (push rule). The vertical edges resemble push and pop rules in $\mathcal{P}_{\mathbb{S}}$. We label these edges with opening and closing parentheses. An opening parenthesis "(" matches a push rule, the closing parenthesis ")" corresponds to a pop rule. The line number in the subscript refers to the call site that is pushed to the stack.

Assume a context-, flow- and field-sensitive data-flow analysis to track the object pointed to by u@51. We refer to this abstract object by $o_{\texttt{u@51}}$. Additionally, assume we want to infer whether $o_{\texttt{u@51}}$ is accessible by y@55. The actual data-flow is best understood within the graph representation which contains a path from u to y. The labels along this path concatenate to form the sequence (or word) $h \cdot (_{53} \cdot g \cdot)_{53} \cdot \overline{g} \cdot \overline{f}$. The parentheses $(_{53}$ and $)_{53}$ are properly matched. Therefore, the path is realizable in terms of context-sensitivity, i.e., a valid execution path. However, the path is not feasible in terms of field accesses. The field store g is properly matched against the load $\overline{g}$, but the field store h does not match the load of $\overline{f}$. In other words, there is no data-flow connection between





q@58 w@53 w@54 w@55
g
q@58g p@57 p@58 p@59 v@52 v@53 v@54 v@55 x@54 x@55
h
v@52h u@51 u@52 u@53 u@54 u@55
ϵ
ou@51

Fig. 6. The post* saturated $\mathcal{A}_{\mathbb{S}}$ (left) and $\mathcal{A}_{\mathbb{F}}$ (right) for the example in Figure 5.

u@51 and y@55, which means the latter does not point to $o_{\texttt{u@51}}$. Synchronized pushdown systems prove the same as $\langle\langle y.\epsilon@55^{\epsilon}\rangle\rangle \notin post_{\mathbb{S}}^{\mathbb{F}}(\langle\langle u.\epsilon@51^{\epsilon}\rangle\rangle)$.

For the data-flow analysis we construct $\mathcal{A}_{\mathbb{S}}^{\mathbb{F}} = (\mathcal{A}_{\mathbb{S}}, \mathcal{A}_{\mathbb{F}})$ such that the automaton accepts the configuration $\langle\langle \texttt{u}.\epsilon@51^{\epsilon}\rangle\rangle$. Therefore, $\langle\langle \texttt{u}, 51\rangle\rangle \in \mathcal{A}_{\mathbb{S}}$ and $\langle\langle \texttt{u@51}, \epsilon\rangle\rangle \in \mathcal{A}_{\mathbb{F}}$. The set $post_{\mathbb{S}}^{\mathbb{F}}(\langle\langle u.\epsilon@51^{\epsilon}\rangle\rangle)$ is then represented by the two automata depicted in Figure 6. $\mathcal{A}_{\mathbb{S}}$ accepts the configuration $\langle\langle \texttt{x}, 54\rangle\rangle$ and $\mathcal{A}_{\mathbb{F}}$ accepts $\langle\langle \texttt{x@54}, h\cdot\epsilon\rangle\rangle$. Therefore, the synchronized configuration $\langle\langle \texttt{x}.h\cdot\epsilon@54^{\epsilon}\rangle\rangle$ is accepted. Since $\mathcal{A}_{\mathbb{S}}^{\mathbb{F}}$ is constructed based on the initial synchronized configuration $\langle\langle \texttt{u}.\epsilon@51^{\epsilon}\rangle\rangle$, accessing field h of x (line 55) retrieves the same object as stored in variable u at statement 51. In other words, object $o_{\texttt{u@51}}$ is accessible via access path x.h after statement 54. The next line, statement 55, loads the field f of variable x. Due to the field load statement, $\mathcal{P}_{\mathbb{F}}$ lists the pop rule $\langle\langle \texttt{x@54}, \texttt{f}\rangle\rangle \to \langle\langle \texttt{y@55}, \epsilon\rangle\rangle$. $\mathcal{A}_{\mathbb{F}}$ does not contain a transition out of state x@54 with label f. Therefore, the pop rule cannot be applied, consequently y@55 does not become a state of $\mathcal{A}_{\mathbb{F}}$.

Despite $\mathcal{A}_{\mathbb{S}}$ accepting the configuration $\langle\langle \texttt{y}, 55\rangle\rangle$, $\langle\langle \texttt{y@55}, \epsilon\rangle\rangle$ is not an accepted configuration for $\mathcal{A}_{\mathbb{F}}$. In turn, $\langle\langle y.\epsilon@55^{\epsilon}\rangle\rangle \notin post_{\mathbb{S}}^{\mathbb{F}}(\langle\langle u.\epsilon@51^{\epsilon}\rangle\rangle)$.

4.1 Undecidability and Required Approximations

The "synchronized" combination of the two automata, as we presented it above, raises the question whether a tighter integration of both automata would not be possible and beneficial. Unfortunately, as Reps [2000] showed, context-sensitive data-dependence analysis is generally undecidable: it can be mapped to a reachability problem on a graph with two interleaved context-free languages (CFL), which means a word formed along *one* path in the graph must form a correct word in *both* CFLs. In Example 4.1 we have seen that a context- and field-sensitive data-flow analysis is equivalent to a reachability problem of two CFLs: one language ($L_{\mathbb{F}}$) for field stores and loads (e.g., f and $\overline{f}$), and a second one ($L_{\mathbb{S}}$) matching call and return flows (e.g., $(_{53}$ and $)_{53}$).

An SPDS presents a decidable construction (conjunction of the sets $post_{\mathbb{S}}^{*}$ and $post_{\mathbb{F}}^{*}$) and therefore must over-approximate the fully precise context-sensitive and field-sensitive analysis solution. The SPDS computes both sets along potentially *different* control-flow paths which introduces an over-approximation in cases in which improperly matched calls in a target program induces a properly matched field accesses. The following example demonstrates the potential precision loss.

Example 4.2. Figure 7 extends the code snippet provided in Figure 5 with two new methods. The method baz(), similar to bar(), calls foo() after storing a field, and the method qux() that calls both methods baz() and bar() (lines 62 and 63). The first parameter of both calls from qux() to baz() and bar() is the same variable a. Below the code, we also show the complete and updated

```
61 qux(a, b, c){          65 baz(r, s){
62   bar(a, b);           66   s.f = r;
63   baz(a, c);           67   t = foo(s);
64 }                      68 }
```

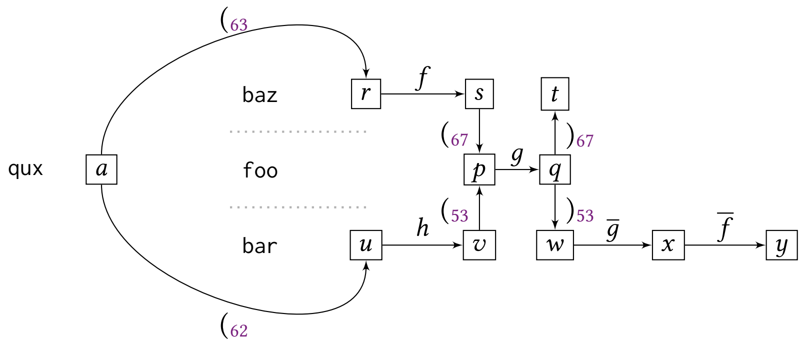


Fig. 7. Code snippet that extends the Example 4.1 from Figure 5 and the updated graph representation.

graphical representation for the code of Figure 7. The earlier representation is extended with the variable nodes for `baz()` and `qux()` and the respective edges.

Assume we want to know if there is a data-flow path from a to y. The graph contains two paths between the nodes. One path that contains node u generates the word $w_1 = (_{62}{\cdot}h \cdot (_{53}{\cdot}g{\cdot})_{53} \cdot \overline{g} \cdot \overline{f}$. The sequence of labels along the other path forms $w_2 = (_{63}{\cdot}f \cdot (_{67}{\cdot}g{\cdot})_{53} \cdot \overline{g} \cdot \overline{f}$. In combination, both paths introduce imprecision into the analysis, because they make the analysis report a flowing to y, despite it being impossible at runtime.

Along the path of the word w_1, and since more opening parentheses are acceptable, the call parenthesis are properly matched, $(_{62}(_{53})_{53}$. However, the field stores and loads are not properly matched, $h \cdot g \cdot \overline{g} \cdot \overline{f}$. For the second path with the word w_2, the situation is the other way around. The field stores and loads are properly matched, $h \cdot g \cdot \overline{g} \cdot \overline{h}$, while the call parentheses of the word do not match, $(_{63}{\cdot}(_{67}{\cdot})_{53}$.

To conclude, the set $post^*_{\mathbb{S}}$ contains all nodes m reachable from an entry node n of the graph such that the word on a path p_1 between n and m forms a word in $L_{\mathbb{S}}$. Further, $post^*_{\mathbb{F}}$ contains all nodes m such that a path p_2 from n to m forms a word in $L_{\mathbb{F}}$. However, the path p_1 and p_2 may differ.

As we showed, it is possible to construct examples where synchronized pushdown systems do not precisely solve the data-flow problem. Yet, our empirical evaluation reveals *no* practical occurrence of this over-approximation. Therefore, we are confident that our hypothesis is true: An improperly matched call site does not induce a properly matched field access (and vice versa).

4.2 Worst-Case Complexity Analysis

We next discuss the worst-case complexity for the computation of $post^{\mathbb{F}}_{\mathbb{S}}(c)$ for SPDS, and compare it to a context-sensitive and flow-sensitive analysis that uses a k-limited access-path representation. For the comparison we encode the latter analysis as an analysis based on a single pushdown system. For a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$, algorithm post* constructs the $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \delta, P, F)$ with a complexity of $\mathcal{O}(|P|\,|\Delta|\,(|Q| + |\Delta|) \;+\; |P|\,|\delta|)$ for both time and space [Esparza et al. 2000].

Synchronized Pushdown Systems. A SPDS computes two independent *post** sets for $\mathcal{P}_\mathbb{F}$ and $\mathcal{P}_\mathbb{S}$, hence the worst-case complexity is the maximum of any of the two post* computations. The control locations of $\mathcal{P}_\mathbb{S}$ are the program variables involved in queried data flows, an upper bound of which is $|P| = |\mathbb{V}|$. The out-set of a data-flow at an assignment statement has at most two[7] variables. For every other statement, the out-set contains one or zero elements. There is a data-flow for every edge (at most $|\mathbb{S}|^2$) in the inter-procedural control-flow graph, and the number of rules can be approximated by $|\Delta| = 2|\mathbb{V}||\mathbb{S}|^2$. $\mathcal{A}_\mathbb{S}$ has one state per variable and an intermediate state for each variable that flows at a call site to a callee, hence the number of $\mathcal{A}_\mathbb{S}$ states $|Q| = |\mathbb{V}| + |\mathbb{V}||\mathbb{S}| \leq 2|\mathbb{V}||\mathbb{S}|$. Each transition of $\mathcal{A}_\mathbb{S}$ is labeled by a statement, and $\mathcal{A}_\mathbb{S}$ has at most $|\delta| = 4|\mathbb{V}|^2|\mathbb{S}|^3$ edges and computing post* for $\mathcal{P}_\mathbb{S}$ has a worst-case complexity of

$$\mathcal{O}(|\mathbb{V}|^2|\mathbb{S}|^2(|\mathbb{V}||\mathbb{S}| + |\mathbb{V}||\mathbb{S}|^2) + |\mathbb{V}|^3|\mathbb{S}|^3) = \mathcal{O}(|\mathbb{V}|^3|\mathbb{S}|^4).$$

The control locations of $\mathcal{P}_\mathbb{F}$ are pairs of variables and statements, and we approximate $|P| = |\mathbb{V}||\mathbb{S}|$. In practice, the variable of a control location of $\mathcal{P}_\mathbb{F}$ must be local to the method of the statement of the control location, which greatly reduces the size of the set P. The number of rules of $\mathcal{P}_\mathbb{F}$ is bounded by $|\Delta| = 2|\mathbb{V}||\mathbb{S}|^2|\mathbb{F}|$, because at an assignment statement the analysis applies at most two rules for every field. In the worst case, for each variable at each statement, a push rule creates an intermediate state which bounds the states of $\mathcal{A}_\mathbb{F}$ by $|Q| = |\mathbb{V}||\mathbb{S}||\mathbb{F}|$. The size of the transitions set of $\mathcal{A}_\mathbb{F}$ can be approximated by $|\delta| = 4|\mathbb{V}|^2|\mathbb{S}|^2|\mathbb{F}|^3$, because, between each of the states, there can be a transition labeled by a field. From these approximations, the complexity of the computation of post* for $\mathcal{P}_\mathbb{F}$ evaluates to $\mathcal{O}(|\mathbb{V}|^2|\mathbb{S}|^3|\mathbb{F}|(|\mathbb{V}||\mathbb{S}||\mathbb{F}| + |\mathbb{V}||\mathbb{S}|^2|\mathbb{F}|) + |\mathbb{V}|^3|\mathbb{S}|^3|\mathbb{F}|^3)$ which reduces to

$$\mathcal{O}(|\mathbb{V}|^3|\mathbb{S}|^5|\mathbb{F}|^2 + |\mathbb{V}|^3|\mathbb{S}|^3|\mathbb{F}|^3). \tag{1}$$

This complexity dominates the complexity for $\mathcal{P}_\mathbb{S}$, therefore, the worst-case complexity for SPDS is the same as of $\mathcal{P}_\mathbb{F}$.

Access Paths with k-limiting. For comparison, we assume the k-limited access-path based analysis (AP^k) to be encoded as a pushdown system similarly to $\mathcal{P}_\mathbb{S}$, i.e., call sites correspond to push-rules and return statements to pop-rules of the system. Instead of using variables ($\mathbb{V}$) as control locations, the control locations for the k-limited analysis are access paths, i.e., a local variable followed by a k-limited sequence of fields. Hence it is $|P| = |\mathbb{V}||\mathbb{F}|^k$. The size of the rule set is at most $|\Delta| = 2|\mathbb{V}||\mathbb{F}|^k|\mathbb{S}|^2$ because, for every edge of the control flow graph, an access path is mapped to at most two access paths.[8] The pushdown system's stack alphabet is $\mathbb{S}$, which limits the size of the state set of the $\mathcal{P}$-automaton to $|Q| = 2|\mathbb{V}||\mathbb{F}|^k|\mathbb{S}|$, and the size of the transitions set to $|\delta| = 4|\mathbb{V}|^2|\mathbb{S}|^3|\mathbb{F}|^{2k}$. For AP^k it results a worst-case complexity of $\mathcal{O}(|\mathbb{V}|^2|\mathbb{S}|^2|\mathbb{F}|^{2k}(|\mathbb{V}||\mathbb{S}||\mathbb{F}|^k + |\mathbb{V}||\mathbb{S}|^2|\mathbb{F}|^k) + |\mathbb{V}|^3|\mathbb{S}|^3|\mathbb{F}|^{3k})$ which simplifies to

$$\mathcal{O}(|\mathbb{V}|^3|\mathbb{S}|^4|\mathbb{F}|^{3k} + |\mathbb{V}|^3|\mathbb{S}|^3|\mathbb{F}|^{3k}). \tag{2}$$

We now compare the analysis complexity of AP^k (2) to the complexity of SPDS (1). The complexities differ in two parts. First, AP^k multiplies the exponent of all $|\mathbb{F}|$ factors by the value k. Second, SPDS increases $|\mathbb{S}|^4$ to $|\mathbb{S}|^5$. The additional factor $|\mathbb{S}|$ is introduced by automaton $\mathcal{A}_\mathbb{F}$, as its states refer to statements in addition to variables.

[7] At a field-store statement $x.f \leftarrow y$, we assume y to flow to x only but not to any alias of x. We discuss aliasing in Section 5.
[8] As for SPDS, we also ignore aliasing here.

```
69 foo(){
70   u = new;
71   v = u;
72   x = new;
73   u.f = x;
74   y = v.f;
75 }
```

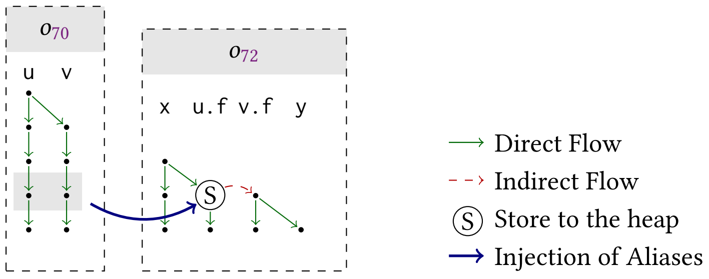


Fig. 8. Over-coming non-distributivity of points-to analysis using points of aliasing.

It is expected that for some $k > 0$ SPDS is more performant than AP^k for data-flows that are assigned to many fields and at the same time reach few statements. Additionally, the complexity estimates show that the larger k the more time and space AP^k requires. In Section 6 we show that in practice SPDS are almost as efficient as AP^k when $k = 1$, although SPDS deliver results as precise as AP^k with $k = \infty$.

5 INSTANTIATIONS OF SYNCHRONIZED PUSHDOWN SYSTEMS

To showcase that SDPS are applicable in a broader context, we present two instantiations, one for points-to and one for typestate analysis. Both instantiations track pointers throughout the program, and both are based on prior work that uses an access-graph based representation [Padhye and Khedker 2013; Späth et al. 2016]. We here present their corresponding synchronized pushdown systems.

5.1 Points-To Analysis

Points-to analysis is a fundamental form of static data-flow analysis that is required by many standard analyses such as call graph construction, taint analysis, and typestate analysis. A points-to analysis computes points-to sets for program variables, where a points-to set element is an allocation site a, i.e., a statement of the form $a : x \leftarrow new$. Those allocation sites abstract the potential objects that the variable may point to at runtime.

Precise points-to information is challenging to compute. In particular, it is *non-distributive* [Padhye and Khedker 2013; Späth et al. 2016]: At field-store statements of the form $x.f \leftarrow y$, not only is the content of $x.f$ changed, but the statement can also change the contents of other variable's fields, for instance, $z.f$ if x and z alias. This aliasing decision requires access to pointer information for both x and z, which leads to non-distributive flow functions.

Späth et al. [2016] present a way to model large parts of the pointer analysis within the distributive framework IFDS [Reps et al. 1995]. The same concept is also applicable to SPDS which uses distributive flow functions, and consequently, one can compute points-to information using SPDS.

Example 5.1. Figure 8 presents a minimal example showcasing non-distributivity of pointer information. The code snippet allocates two objects o_{70} and o_{72} in lines 70 and 72, respectively. At runtime, variables x and y point to the same object o_{72}.

The trick in using SPDS to solve a pointer-analysis problem is to use not one single SPDS but one SPDS instance per allocation site. The figure thus depicts the data-flows computed by $post^{\mathbb{F}}_{\mathbb{S}}$ for *two* SPDS, in the boxes that are labeled with the corresponding object. Points-to information can be extracted from the data-flow graphs nodes for o_{70}. For example, the graph for object o_{70} contains nodes u and v at the statement in line 73. This means that all these variables at the respective statement point to the object o_{70}.

The SPDS for o_{72} encodes only the object's direct assignments, e.g., that o_{72} flows to variable x and access path u.f. To deal with possible aliasing at the heap assignment in line 73, the SPDS for o_{72} thus interacts with the one for o_{70}, because that SPDS contains the base variable of the store (u). Through the SPDS for o_{70}, we learn that at the same statement v may accesses the same object. This alias information is then passed to the field-store statement ⓢ within o_{72}, causing the analysis to add an indirect data-flow edge from u.f to v.f at that statement. Due to the indirect-flow edge, the data-flow graph for o_{72} then encodes a path to variable y, indicating that after statement 74 variable y points to o_{72}.

Encoding pointer-analysis in this distributive manner has one big advantage, no matter whether one uses IFDS or SPDS: Non-distributive analyses propagate *sets* of access paths (or graphs), which can cause an exponential growth in the abstract domain. The distributive encoding avoids this growth by instead tracking allocation sites individually. The use of SPDS can improve over the use of IFDS by the PDS-based encoding of field-sensitivity, as explained in Section 4.2. Our evaluation in Section 6 shows that this encoding, in practice, can greatly benefit performance.

5.2 Weighted Pushdown System for Typestate Analysis

A *typestate analysis* reasons about the states of an object of a particular type by following the data-flow path of an object and discovering paths along which the application programming interface (API) of the object is not used according to its specification [Strom and Yemini 1986]. Typical examples are files that remain open and leak resources or collection APIs throwing runtime exceptions when a developer accesses an element prior to inserting any.

Weighted pushdown systems (WDPS) extend pushdown systems by weights. A *weight* is an element of an idempotent semi-ring $(W, \otimes, \oplus)$. In a weighted pushdown system $\mathcal{P}$, each rule carries a weight and, for the weighted version of a system, the $\mathcal{P}$-automaton encoding the set $post^*$ is lifted to a weighted automaton, where each transitions maps to a weight. The binary operator "*combine*" ($\oplus : W \times W \to W$) of the semi-ring is used when two weights are joined at control-flow join points, while "*extend*" ($\otimes : W \times W \to W$) composes the weights along the data-flow paths.

Some problems long known to be computable by WPDS include linear constant propagation or computing a shortest witness path alongside the data-flow [Reps et al. 2007]. Recently, Späth et al. [2017] showed that *typestate properties* can be solved using Interprocedural Distributive Environments (IDE) [Sagiv et al. 1996] and those can be mapped to weighted pushdown systems. For a typestate analysis, it is crucial to trace *all* aliases to the tracked object, as this allows to compute strong updates. In their framework called IDE^{al}, Späth et al. [2017] propose to track all pointers to an object using access graphs. The access-graph formulation in their work can be replaced by SPDS. The typestate information of the object is then encoded as weights of any of the two pushdown systems, we chose to propagate weights along the call-PDS $\mathcal{P}_{\mathbb{S}}$. Our evaluation in Section 6 uses this data-flow analysis client.

5.3 Backward and Demand-Driven Analyses

In this paper, we discuss synchronized pushdown system as a forward-directed analysis that computes the variables and the access paths an object is propagated to. Synchronized pushdown systems can easily be applied to perform a backward analysis that computes field-sensitive information where the content of a variable originates from.

There are two ways to reverse the direction of the data-flow. One option is to reverse the direction of the control-flow graph and to invert the rules of $\mathcal{P}_{\mathbb{S}}$ and $\mathcal{P}_{\mathbb{F}}$ as Bodden [2012] suggests. An alternative solution is to replace post* with an algorithm called pre* [Esparza et al. 2000] which computes a $\mathcal{P}$-automaton representing the set $pre^*(c) = \{c' \mid c' \Rightarrow^* c\}$.

Precise context-, flow and field-sensitive points-to analyses are frequently too inefficient to compute for the whole program. As client analyses (e.g., taint or typestate) frequently require points-to information only for some variables, a demand-driven points-to analysis can be achieved by computing the "demand" for pointer information using a SPDS-based backward analysis, and then answering those on-demand queries using a forward analysis [Bodden 2018; Späth et al. 2016]. For our evaluation, we also implemented a demand-driven points-to analysis (based on inverting control-flow and the rules) and compare it to an existing demand-driven one which is based on access graphs.

6 EVALUATION

We have used SPDS to implement a demand-driven pointer analysis and a typestate analysis, as sketched in Section 5. The implementation constructs $\mathcal{P}_{\mathbb{F}}$ and $\mathcal{P}_{\mathbb{S}}$ on-the-fly by successively adding rules as required. Also the set $post^{\mathbb{F}}_{\mathbb{S}}$ is constructed dynamically. The implementations of the pointer analysis, the typestate, and our artifact are publicly available.[9]

In our evaluation, we compare the version implemented as SPDS to the state-of-the-art analyses Boomerang [Späth et al. 2016] and IDE^{al} [Späth et al. 2017]. All analyses are implemented based on top of Soot [Lam et al. 2011], allowing us to compare them on equal grounds. IDE^{al} models the heap using access graphs. Boomerang can be configured to use k-limiting or access graphs. Our evaluation compares these abstractions to SPDS through the following research questions:

- **RQ1**: How does the number of field accesses on the data-flow path of a demand-driven pointer query relate to the time taken to compute the query's results?
- **RQ2**: In terms of performance and precision, how does a SPDS-based typestate analysis perform in comparison to an IDE^{al}-based typestate analysis?
- **RQ3**: How does the number of methods and field-stores, i.e., push-rule applications, along a data-flow path influence the typestate-analysis time for an abstract object?

6.1 RQ1: Scalability with Respect to Field Accesses

In this experiment, we compare a demand-driven pointer analysis based on SPDS (Section 5) to Boomerang in a controlled lab environment. Boomerang can be configured to use either k-limiting or access graphs as the field abstraction. We control the number of field accesses along the data-flow path of a pointer query to demonstrate the differences between the field abstractions when the number of field accesses increases. To complement the worst-case complexity analysis in Section 4, we measure how an increase in the number of field accesses affects the query's analysis time in practice.

Experimental Setup. In Section 1, we show an example using `java.util.TreeMap`, in which a data-flow analysis generates all combination of access paths $T = \{\texttt{this}.f_1.f_2.\cdots.f_n.\texttt{value} \mid f_i \in \{\texttt{right}, \texttt{left}, \texttt{parent}\}, n \in \mathbb{N}\}$. We extracted a minimal code pattern from the `TreeMap` implementation that provokes a similar state explosion in the set of generated access paths. We also made the number of field accesses along the data-flow parametrizable and arrived at the code shown for method `stateExplosion1()` in Figure 9. The code is designed to provoke a state explosion for the static data-flow analysis when points-to information for variable `t` is queried after line 86. The backward analysis outputs the three allocations in line 77 as allocation sites for `t`. From there, a

[9]https://github.com/CROSSINGTUD/WPDS and https://github.com/secure-software-engineering/SPDS-experiments

```
76 stateExplosion1() {
77   Node x,p,t = new Node();
78   while(...){
79     if(...){
80       x.a1 = p;
81     }
82     p = x;
83   }
84   if(...){
85     t = x.a1;
86   }
87 }
```

```
88  stateExplosion2() {
89    Node x,p,t = new Node();
90    while(...){
91      if(...){
92        x.a1 = p;
93      } else if(...){
94        x.a2 = p;
95      }
96      p = x;
97    }
98    if(...){
99      t = x.a1;
100   } else if(...){
101     t = x.a2;
102   }
103 }
```

Fig. 9. Example code for EXPL_1 and EXPL_2 that provokes state explosions for access-path based domains. EXPL_1 contains a single field-store and a single field-load of the field a1, EXPL_2 contains additional accesses to a2.

precise forward analysis generates all access paths in the set $\{\texttt{x}.f_1.f_2.\cdots.f_m \mid f_i \in \{\texttt{a1}\}, m \in \mathbb{N}\}$. This result is due to the fact that the while-loop assigns x to p but also stores p in a field of x.

The code snippet allows one to parametrize the number of field accesses. By duplicating both if blocks (lines 79–81 and lines 84–86) and replacing the field a1 of the field-store and load by another field, for example a2, the complexity of the data-flow increases as data flows to all access paths within the set $\{\texttt{x}.f_1.f_2.\cdots.f_m \mid f_i \in \{\texttt{a1}, \dots, \texttt{an}\}, m \in \mathbb{N}\}$. We call the program with n field accesses EXPL_n. In Figure 9, the code for EXPL_2 is depicted as method stateExplosion2().

For this performance experiment, we stepwise scale the number of fields accesses n in the program EXPL_n and trigger points-to queries to Boomerang and the points-to analysis based on SPDS. We run the Boomerang queries with access graphs and with k-limited access paths with values $k = 1, \dots, 5$. Each analysis query computes the points-to set of variable t after statement 86. We measured the analysis time for each query.

Results. The chart in Figure 10 plots the results for this experiment. The number of fields of EXPL_n on the x-axis is plotted against the analysis time on the y-axis. The chart depicts 7 line plots, one for the demand-driven pointer analysis based on SPDS, one for Boomerang using access graphs and 5 lines for AP^k with $k = 1, \dots, 5$. A first observation is that the analysis times of access graphs increases exponentially when more than 5 fields occur along the data-flow. With $n = 5$, the access-graph-based analysis already takes 17 seconds to terminate. For $n = 6$, the queries hit the budget of 50 seconds.

We next compare the access-path-based analyses to SPDS. $AP^{k=1}$ is slightly more efficient than SPDS but generally also less precise when data flows through more than a single field. Lastly, $AP^{k=3}$

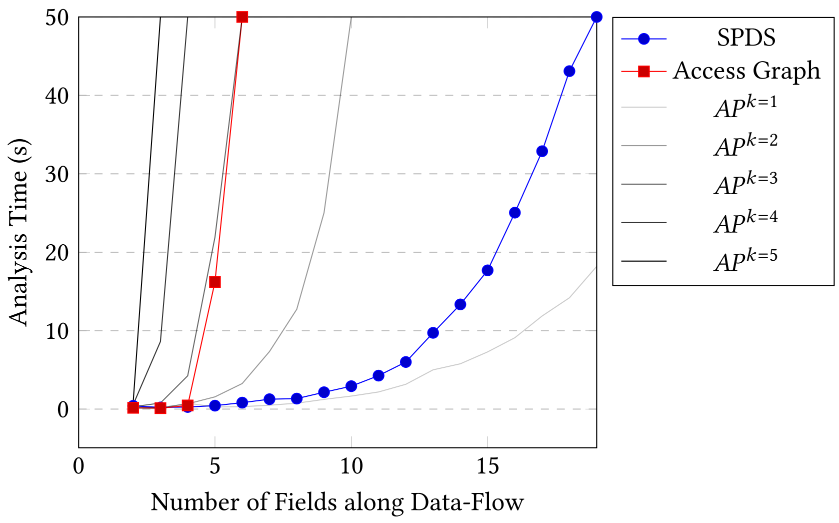


Fig. 10. The number of relevant field accesses for a data-flow analysis and its effect on the analysis time.

is roughly as performant as the one with access graphs. However, an increase of k to 4 and 5 yields to analyses with worse performance.

SPDS also clearly outperforms the access-graph model in this experiment. The line for SPDS shows a quadratic growth when using SPDS, and, even for $n = 18$, the data-flow query finishes in 43 seconds.

This experiment shows the benefit of using SPDS opposed to access graphs or k-limiting when data flows through a sequence of fields-stores. This experiment showcases that the explosion of the size of the data-flow domain directly relates to the analysis time. Compared to both k-limiting and access graphs, for a code pattern that generates loops within the access paths, SPDS always pays off.

Summary. When data flows through five or more nested field-stores, SPDS are more efficient than representations using access paths or access graphs. SPDS show a performance close to k-limited access paths with $k = 1$ although their precision corresponds to $k = \infty$.

6.2 RQ2: Precision and Performance of a Typestate Analysis Client

The next experiment compares an SPDS-based typestate analysis to a typestate analysis based on IDE^{al} on a realistic benchmark. IDE^{al} propagates access graphs, and we want to understand how often the cases discussed in **RQ1** occur in practice. We re-implemented IDE^{al} using SPDS and refer to this variant as IDE$^{al}_{\mathcal{P}}$.

Experimental Setup. We conducted our experiments using the DaCapo 2006 benchmark suite [Blackburn et al. 2006]. DaCapo is widely used in static-analysis benchmarking. The setup for IDE^{al} and IDE$^{al}_{\mathcal{P}}$ is identical. We *include all libraries*, which includes the Java Runtime Environment in version 1.8.0_162. DaCapo contains 11 Java programs of reasonably large size, varying between 1,725 and 5,923 call-graph reachable methods (as computed by SPARK [Lhoták and Hendren 2003]).

The implementation of IDE^{al} ships with a set of typestate specifications for `Vector`, `Iterator`, and `IO`. The specification for `Vector` checks that an element is not accessed when the vector is empty. The specification for `Iterator` detects calls to `next` without first checking the existence of

Table 5. Statistics of a typestate analysis performed in IDE^{al} (▨) and IDE$^{al}_{\mathcal{P}}$ (🟥) on the DaCapo 2006 benchmark programs. A row containing a dash in column *Objects* means no object (allocation site) was found in the program and no statistics are collected. Note: the analysis includes all libraries.

	IO						Vector						Iterator					
	Objects	Visited Methods	Nesting Depth	Total Time (s)	Timeouts	Rel. Timeouts (%)	Objects	Visited Methods	Nesting Depth	Total Time (s)	Timeouts	Rel. Timeouts (%)	Objects	Visited Methods	Nesting Depth	Total Time (s)	Timeouts	Rel. Timeouts (%)
ANTLR	17	12	4	4,023	5	29.4	2	178	3	150	0	0	-	-	-	-	-	0
				15	0	0				601	1	50				-	-	0
BLOAT	36	234	9	15,604	26	72.2	9	182	6	1,203	2	22.2	288	23	9	72,062	105	36.5
				3,491	3	8.3				119	0	0				6,104	3	1.0
CHART	18	10	4	3,602	6	33.3	4	18	1	1	0	0	21	5	4	6,601	11	52.4
				22	0	0				5	0	0				4	0	0
ECLIPSE	24	14	5	6,363	10	41.7	24	62	10	6,001	10	41.7	5	7	8	660	1	20
				40	0	0				1,210	2	8.3				2	0	0
FOP	16	14	4	4,218	7	43.8	4	29	6	601	1	25.0	4	5	1	1	0	0
				24	0	0				143	0	0				2	0	0
HSQLDB	54	248	9	27,483	45	83.3	1	17	1	1	0	0	-	-	-	-	-	0
				10,502	15	27.8				2	0	0				-	-	0
JYTHON	69	157	11	25,593	41	59.4	36	659	13	15,338	24	66.7	5	7	9	180	0	0
				6,184	10	14.5				13,370	14	38.9				3	0	0
LUINDEX	15	12	3	3,604	6	40	15	30	4	2,193	3	20	6	5	1	31	0	0
				14	0	0				28	0	0				2	0	0
LUSEARCH	15	13	5	3,243	5	33.3	14	134	13	4,471	7	50	11	5	10	2,044	3	27.3
				19	0	0				3,019	5	35.7				3	0	0
PMD	18	12	4	3,736	5	27.8	13	282	5	6,000	10	76.9	54	5	8	10,631	17	31.5
				18	0	0				2,094	3	23.1				7	0	0
XALAN	16	11	3	2,582	4	25.0	3	20	1	0	0	0	2	5	7	62	0	0
				18	0	0				3	0	0				1	0	0

the next element using `hasNext`. The typestate specification `IO` concerns `Input` and `Output` streams which must be correctly closed after usage.

Using these typestate specifications, we applied both IDE^{al} and IDE$^{al}_{\mathcal{P}}$ to each program in DaCapo. Both analyses generate a context-insensitive call-graph using Spark [Lhoták and Hendren 2003], determine all abstract objects, i.e., allocation sites, of type `Vector`, `Iterator`, and `IO`, and execute a static analysis per abstract object. Aliasing at heap accesses and strong updates of typestate information are resolved through Boomerang-based points-to queries.

We ran this experiment on a 2.3 GHz Intel Core i7 machine, and we granted 12 GB of heap memory to the JVM. During the computation, we record two statistics about the data-flow of each abstract object. First, the number of *Visited Methods*: a method is *visited* if at least one data-flow fact is generated in this method. Second, the *Nesting Depth* of an object which is the maximal number of fields the object is stored in and is computed as the length of the longest acyclic path in $\mathcal{A}_{\mathbb{F}}$. If $\mathcal{A}_{\mathbb{F}}$ does not contain a cycle, this number corresponds to the minimal value for k-limiting that avoids approximation. To limit the total analysis time to an acceptable time budget for programs using many abstract objects, we limit the computation of the data-flow for each abstract object to 10 minutes.

Precision Results. SPDS is based on the hypothesis that an improperly matched call site does not induce a properly matched field access and vice versa. SPDS over-approximates when the target program contains two distinct paths d_1 and d_2 such that d_1 properly matches one language ($L_{\mathbb{F}}$ or $L_{\mathbb{S}}$) but does not match in the second language and conversely for path d_2 (see Section 4.1). For the typestate analysis IDE$^{al}_{\mathcal{P}}$, this over-approximation would lead to an additionally reported finding (false positive) in comparison to IDE^{al}, because it would cause the analysis to construct an invalid data-flow path. Yet, for all objects for which IDE^{al} and IDE$^{al}_{\mathcal{P}}$ terminate (464 out of 819 objects), both analyses report the *same results*. This evidence shows that our hypothesis is true in practice. Späth et al. [2017] has a more detailed discussion of the general precision of IDE^{al}.

Performance Results. Table 5 lists the results of the typestate analysis grouped by the three typestate properties (`IO`, `Vector`, and `Iterator`) on the DaCapo benchmark suite. Each row of the table corresponds to one program. For each property, column *Objects* lists how many `Vector`, `Iterator`, or `IO` allocation sites the program contains in the pre-computed call-graph. Column *Visited Methods* shows the average number of methods visited when computing the data-flows for all objects. The column *Nesting Depth* represents the average nesting depth of all objects. The last three columns reflect the analysis time: The column *Total Time* lists the complete analysis time on the benchmark (excluding call-graph construction time). The column *Timeouts* shows the *number* of objects for which the data-flow analysis exceeded the budget of 10 minutes, the last column, *Rel. Timeouts* shows the *fraction* of those objects over all analyzed ones. The last three columns, *Total Time*, *Timeouts*, and *Rel. Timeouts* are split horizontally into two rows per program. For each program, the upper row contains data for the original IDE^{al} implementation, and the lower row contains data for IDE$^{al}_{\mathcal{P}}$.

For example, the program Antlr allocates a total of 17 objects related to `IO`, e.g., of type `FileInputStream` or `FileOutputStream`. Across these 17 objects, on average, the data-flow path visits 12 methods and stores the object within 4 unique fields, indicating that an access path of length at least 4 is required for a precise analysis. For 5 of the 18 analyzed objects, IDE^{al} times out, whereas IDE$^{al}_{\mathcal{P}}$ does not time out on any object. This leads to a total analysis time of 4,023 seconds for IDE^{al} opposed to only 15 seconds for IDE$^{al}_{\mathcal{P}}$.

The results show that IDE$^{al}_{\mathcal{P}}$ outperforms IDE^{al} for the typestate properties `IO` and `Iterator` on all DaCapo benchmarks in terms of the total analysis time and the number of timeouts. On average, IDE$^{al}_{\mathcal{P}}$ is 83× more efficient than IDE^{al} for the property `Iterator`. For the `IO` property, IDE$^{al}_{\mathcal{P}}$ is 64× more efficient. For the typestate property `Vector`, IDE$^{al}_{\mathcal{P}}$ outperforms IDE^{al} only by a factor of 1.8×. We discuss the large difference in these factors in more details in **RQ3**.

The timeouts dominate the overall analysis time. Switching from IDE^{al} to IDE$^{al}_{\mathcal{P}}$ reduces the timeouts from 160 to 28 for all 298 `IO` objects, and from 57 to 25 for the 125 `Vector` objects. For the 396 `Iterator` object data-flows, the difference is most significant: with IDE^{al}, a total of 137 data-flows time out, while only 3 time out with IDE$^{al}_{\mathcal{P}}$.

In **RQ1**, we constructed the programs EXPL$_n$ that provoke the state explosion on the data-flow domains by nesting an object in up to n fields. The values for nesting depth in Table 5 range up to $k = 13$ and indicate that nesting an object in up to 13 fields, i.e., EXPL$_{13}$, is realistic in the case of a typestate analysis.

Summary. The over-approximations introduced by SPDS do not occur for the typestate analysis IDE$^{al}_{\mathcal{P}}$ on the DaCapo benchmark programs, and IDE$^{al}_{\mathcal{P}}$ outperforms the access-graph-based analysis IDE^{al} in terms of analysis time, while timing out on only 6.8% (56 out of 819) object data-flows.

6.3 RQ3: Influencing Dimensions for IDE^{al} and IDE$^{al}_{\mathcal{P}}$

We next seek to relate the theoretic worst-case complexity results from Section 4 to the practical performance results we obtained. For access-path and access-graph-based analyses, the more field stores are involved in a data-flow path of an object, the higher the expected analysis time for the object is. While the time also increases for SPDS, due to the concise representation of $\mathcal{A}_{\mathbb{F}}$, it is expected that the time is affected less heavily. On the other hand, for data-flows that reach more statements (or equivalent methods), SPDS is expected to have a higher complexity.

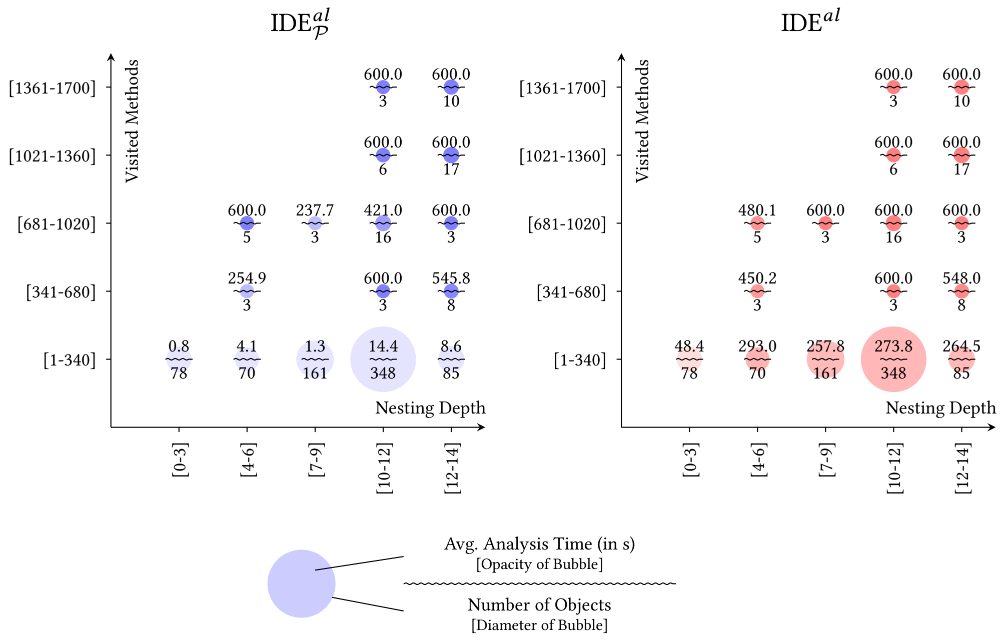


Fig. 11. Relating the analysis times to the number of visited methods and the nesting depth along objects data-flow for $\text{IDE}^{al}_{\mathcal{P}}$ and IDE^{al}.

Experimental Setup. In addition to the experimental setup discussed in **RQ2**, we measured the analysis time per object data-flow to evaluate this influence in practice. Measuring the analysis time for individual abstract objects allows us to cluster based on other recorded statistics (visited methods and nesting depth) of the object data-flows.

Results. First, we explain the variance in the performance reported in Table 5 across the typestate properties by relating the *Timeout* column to the columns *Visited Methods* and *Nesting Depth*.

$\text{IDE}^{al}_{\mathcal{P}}$ timed out in only 3 of the 396 `Iterator` objects, and, in total, timed out in only 42 cases. This is a significant reduction, which we explain as follows. For all benchmarks, the number of visited methods for the typestate `Iterator` is relatively low. On average, an `Iterator` object is alive across 5–23 methods. These methods include the factory calls where the `Iterator` is allocated, as well as their constructors. Ignoring loops, `Iterator` objects are nested in other object's fields in a depth between 1 and 10. The access paths required for `Iterator` are also cyclic. For example, for the benchmark program Eclipse, the following access path is required to be tracked

```
config.this$0.ig.nodes.map.tail.parent.next.prev.right
```

where the part `tail.parent.next.prev.right` can occur in any arbitrary order. As the automaton for an access graph may slightly vary at each statement, IDE^{al} must store the automaton at every statement and eventually times out. On the other hand, in $\text{IDE}^{al}_{\mathcal{P}}$, the automaton $\mathcal{A}_{\mathbb{F}}$ represents all these access paths at every statement concisely in a single automaton.

The `Vector` typestate property is the other extreme. The performance gains through $\text{IDE}^{al}_{\mathcal{P}}$ are less significant. As data containers, `Vector` objects are expected to have a longer lifetime than `Iterator` objects. In Table 5, the number of visited methods for `Vector` objects ranges from 17 to 659.

This motivates a second representation of the data set. In Figure 11, we group all objects, regardless of their type, into buckets, depending on the nesting depth and the number of visited methods along their data flows. The visited methods are plotted along the y-axis, and the number of fields along the x-axis. We subdivide both axes into five equally long ranges which generates a total of 25 buckets.

For each bucket, we report two statistics. The first is the number of objects contained in the bucket, indicated by the value below the meandered line in the diagram. The second statistic is the average analysis time of the objects within this bucket, which is indicated by the number above the meandered line. Timeouts are included in the average with their 10 minutes. We also visualize these statistics as circles associated to the bucket. The diameter of the circle corresponds to the number of objects contained in a bucket. The more opaque a circle is, the more time the analysis took on average across the objects. Figure 11 shows a diagram for $\text{IDE}^{al}_{\mathcal{P}}$ on the left and for IDE^{al} on the right.

The diagram shows two important characteristics. First, for the vast majority of objects, $\text{IDE}^{al}_{\mathcal{P}}$ significantly reduces the analysis times compared to IDE^{al}. Second, the more *Visited Methods* a data-flow spans, the larger the time budget the analysis requires. While this holds for IDE^{al} and $\text{IDE}^{al}_{\mathcal{P}}$, the number of field-stores along the data-flow paths does influence the analysis times more heavily for IDE^{al}.

$\text{IDE}^{al}_{\mathcal{P}}$ shows the largest speedups for the bottom part of Figure 11. In other words, switching from access graphs to SPDS benefits data-flows which span few methods, no matter how deeply the object is nested. $\text{IDE}^{al}_{\mathcal{P}}$ effectively reduces the analysis time for all buckets whose visited method range is [1-340]. Figure 11 shows that the majority of data-flows fall into this range, which contains 742 out of all 819 objects and require only a fraction of all call-graph reachable methods for the analysis.

Additionally, Figure 11 shows that $\text{IDE}^{al}_{\mathcal{P}}$ times out slightly more often when the number of visited methods increases. This observation aligns with the worst-case complexity analysis. Scaling into this direction is an orthogonal challenge to the work presented in this paper. However, pushdown systems enable a range of optimizations, for example *summarization* [Lal and Reps 2008] where sub-automata of $\mathcal{A}_{\mathbb{S}}$ (or $\mathcal{A}_{\mathbb{F}}$) are shared across multiple post* computations. Orthogonal to these optimizations, we are currently investigating how demand-driven call-graph refinement [Sridharan and Bodík 2006] may help reduce the remaining timeouts.

Summary. The number of visited methods and the number of fields participating in the data-flow are *the influencing factors* for the analysis times for $\text{IDE}^{al}_{\mathcal{P}}$. However, the number of visited methods has a higher impact and SPDS is most advantageous in situations where the data-flow spans few methods but flows through many fields.

7 RELATED WORK

There are various approaches for encoding context-sensitive and field-sensitive (but mostly flow-insensitive) alias or points-to analyses as two CFL-reachability (or Dyck-reachability) problems [Chatterjee et al. 2018; Sridharan and Bodík 2006; Sridharan et al. 2005; Xu et al. 2009; Zhang et al. 2013]. To guarantee decidability, all the CFL-formulations over-approximate either the CFL for field stores/loads or the CFL for call/returns. We refer to surveys on alias analysis and respective heap abstractions [Kanvar and Khedker 2016; Sridharan et al. 2013] for more detailed comparisons. Analyses that are flow-insensitive are expected to be more memory-efficient, because persisting a single data-flow fact per method suffices. For flow-sensitive data-flow clients, e.g., a typestate analysis that performs strong updates, the results are insufficiently precise [Fink et al. 2008; Späth et al. 2017].

In recent work, Zhang and Su [2017] introduce linear conjunctive language (LCL) reachability and show how the interleaved matching-parentheses problem of field-sensitive and context-sensitive data-flow analysis can be over-approximated by a LCL. Their work presents a new algorithm to solve LCL-reachability, instead, we show that we can formulate the problem in two pushdown systems and rely on existing algorithms and improvements [Esparza et al. 2000; Lal et al. 2005; Reps et al. 2016]. Zhang et al. also base their approach on the hypothesis that we discuss in Section 4.1. Hence, our evaluation also delivers additional evidence for their work.

Control-flow analysis is a technique to analyze languages that allow higher-order functions as first class elements of the language. Data-flow analyses of target programs using higher-order functions require constructing control-flow (i.e., call graph) dynamically to be precise. CFA2 [Vardoulakis and Shivers 2010], PDCFA [Earl et al. 2012], and P4F [Gilray et al. 2016] are analyses targeted at dynamic exploration of control-flow. In contrast to these approaches, SPDS relies on a precomputed call graph. While Java supports limited higher-order functions through its object-orientation [Might et al. 2010], current call-graph algorithms for Java can most often properly resolve function calls through pointer analysis. Subsequently, we do not expect higher-order functions to have a significant impact on the analysis precision in Java. A second major difference to CFA2, PDCFA, and P4F is that SPDS uses two instead of one pushdown systems to model the heap.

We showcased SPDS based on a typestate analysis. A typestate analysis requires flow-sensitive information. Prior research on data-flow analyses that are flow-sensitive, context-sensitive, and also field-sensitive is rare. Andromeda [Tripp et al. 2013] and FlowDroid [Arzt et al. 2014] are two precise taint analyses of these dimensions, and both use k-limiting. From the results obtained in the evaluation, we assume that lifting FlowDroid to SPDS, tainted data-flows can be computed more efficiently.

IFDS-APA [Lerch et al. 2015] encodes a context-sensitive, flow-sensitive, and field-sensitive analysis. Such configuration is closely related to SPDS. This formulation does not use pushdown systems, instead the authors' approach requires that either the language of field stores and loads or the language of matching call and returns must be over-approximated by a regular language, an additional (and lossy) computation step not required in SPDS.

In Section 1, we have discussed the related work on access graphs [Geffken et al. 2014; Khedker et al. 2007; Späth et al. 2016], which is a similar representation to $\mathcal{A}_{\mathbb{F}}$ of $\mathcal{P}_{\mathbb{F}}$. Access graphs allow a finite representation of the potentially infinite number of access paths. However, to be flow-sensitive, these previous approaches maintained access graphs per statement. Using SPDS, the single automaton $\mathcal{A}_{\mathbb{F}}$ encodes all field accesses at all statements. Similarly, also *alias graphs* [Kastrinis et al. 2018], a field abstraction proposed as an efficient data-flow model for must-aliasing access paths, must store information statement-wise. While alias graphs compute *must*-alias information, our solution computes *may*-alias information.

8 CONCLUSION

We have presented synchronized pushdown systems (SPDS), a new concept to context-sensitive, flow-sensitive, and field-sensitive data-flow analysis. An SPDS is a combination of two flow-sensitive pushdown systems, one matching calls and returns, the other matching field stores and loads. The combination of two resulting $\mathcal{P}$-automata then computes context-sensitive, flow-sensitive, and field-sensitive data-flow information.

SPDS pushes its over-approximations into corner cases where a context-insensitive data-flow path occurs simultaneously with a field-sensitive path or vice versa. While we we showed that this causes imprecision on synthetic examples, our experiments using a pointer-tracking typestate analysis confirm that those situations do not arise in practice, allowing SPDS to yield an analysis that is fully field- and context-sensitive in most practical situations.

This paper also presents a worst-case complexity analysis of SPDS which unravels that the SPDS shifts the complexity in comparison to the same analysis implemented with k-limiting in an often favorable way. Our practical evaluation confirms these results: In particular for the predominant types of data-flows, flows that span a small number of methods, but require tracking a large number of field accesses, using SPDS outperforms existing solutions based on access paths or access graphs.

ACKNOWLEDGMENTS

This research was supported by a Fraunhofer Attract grant as well as the Heinz Nixdorf Foundation. This material is also based upon work supported by the National Sciences and Engineering Research Council of Canada.

REFERENCES

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Programming Language Design and Implementation (PLDI)*. 259–269.

George Balatsouras, Kostas Ferles, George Kastrinis, and Yannis Smaragdakis. 2017. A Datalog Model of Must-Alias Analysis. In *International Workshop on State Of the Art in Java Program analysis, (SOAP)*. 7–12. https://doi.org/10.1145/3088515.3088517

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 169–190.

Eric Bodden. 2012. Interprocedural Data-Flow Analysis with IFDS/IDE and Soot. In *International Workshop on State Of the Art in Java Program analysis, (SOAP)*. 3–8. https://doi.org/10.1145/2259051.2259052

Eric Bodden. 2018. The Secret Sauce in Efficient and Precise Static Analysis. In *International Workshop on State Of the Art in Java Program analysis, (SOAP)*. 84–92. http://bodden.de/pubs/bodden18secret.pdf

Ahmed Bouajjani, Javier Esparza, and Oded Maler. 1997. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *International Conference on Concurrency Theory (CONCUR)*. 135–150. https://doi.org/10.1007/3-540-63141-0_10

Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2018. Optimal Dyck Reachability for Data-Dependence and Alias Analysis. In *Symposium on Principles of Programming Languages (POPL)*. 30:1–30:30. https://doi.org/10.1145/3158118

Ben-Chung Cheng and Wen-mei W. Hwu. 2000. Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Evaluation. In *Programming Language Design and Implementation (PLDI)*. 57–69. https://doi.org/10.1145/349299.349311

Arnab De and Deepak D'Souza. 2012. Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates. In *European Conference on Object-Oriented Programming (ECOOP)*. 665–687. https://doi.org/10.1007/978-3-642-31057-7_29

Alain Deutsch. 1994. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Programming Language Design and Implementation (PLDI)*. 230–241. https://doi.org/10.1145/178243.178263

Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. 2012. Introspective Pushdown Analysis of Higher-Order Programs. *CoRR* abs/1207.1813 (2012). arXiv:1207.1813 http://arxiv.org/abs/1207.1813

Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. 2000. Efficient Algorithms for Model Checking Pushdown Systems. In *International Conference on Computer Aided Verification (CAV)*. 232–247. https://doi.org/10.1007/10722167_20

Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-Up Context-Sensitive Pointer Analysis for Java. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 465–484. https://doi.org/10.1007/978-3-319-26529-2_25

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective Typestate Verification in the Presence of Aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2008).

Alain Finkel, Bernard Willems, and Pierre Wolper. 1997. A Direct Symbolic Approach to Model Checking Pushdown Systems. *Electronic Notes in Theoretical Computer Science* (1997), 27–37.

Manuel Geffken, Hannes Saffrich, and Peter Thiemann. 2014. Precise Interprocedural Side-Effect Analysis. In *International Colloquium on Theoretical Aspects of Computing (ICTAC)*. 188–205.

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown Control-Flow Analysis for Free. In *Symposium on Principles of Programming Languages (POPL)*. 691–704. https://doi.org/10.1145/2837614.2837631

Neville Grech and Yannis Smaragdakis. 2017. P/Taint: Unified Points-To and Taint Analysis. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 102:1–102:28.

David Hauzar, Jan Kofron, and Pavel Bastecký. 2014. Data-Flow Analysis of Programs with Associative Arrays. In *International Workshop on Engineering Safety and Security Systems (ESSS)*. 56–70. https://doi.org/10.4204/EPTCS.150.6

David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 132–136. https://doi.org/10.1145/1028664.1028717

Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. 2009. Static Data Race Detection for Concurrent Programs with Asynchronous Calls. In *International Symposium on Foundations of Software Engineering (FSE)*. 13–22. https://doi.org/10.1145/1595696.1595701

Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Computing Surveys (CSUR)* (2016), 29:1–29:47.

George Kastrinis, George Balatsouras, Kostas Ferles, Nefeli Prokopaki-Kostopoulou, and Yannis Smaragdakis. 2018. An Efficient Data Structure for Must-Analysis. In *Compiler Construction (CC)*. 48–58.

Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. 2007. Heap Reference Analysis Using Access Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2007).

Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Symposium on Principles of Programming Languages (POPL)*. 194–206. https://doi.org/10.1145/512927.512945

Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2018. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *European Conference on Object-Oriented Programming (ECOOP)*. 10:1–10:27. https://doi.org/10.4230/LIPIcs.ECOOP.2018.10

Akash Lal and Thomas W. Reps. 2006. Improving Pushdown System Model Checking. In *International Conference on Computer Aided Verification (CAV)*. 343–357.

Akash Lal and Thomas W. Reps. 2008. Solving Multiple Dataflow Queries Using WPDSs. In *International Symposium on Static Analysis (SAS)*. 93–109. https://doi.org/10.1007/978-3-540-69166-2_7

Akash Lal, Thomas W. Reps, and Gogul Balakrishnan. 2005. Extended Weighted Pushdown Systems. In *International Conference on Computer Aided Verification (CAV)*. 434–448. https://doi.org/10.1007/11513988_44

Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*. http://www.bodden.de/pubs/lblh11soot.pdf

Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. 2014. FlowTwist: Efficient Context-Sensitive Inside-Out Taint Analysis for Large Codebases. In *International Symposium on Foundations of Software Engineering (FSE)*. 98–108. https://doi.org/10.1145/2635868.2635878

Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. 2015. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths. In *International Conference on Automated Software Engineering (ASE)*. 619–629. https://doi.org/10.1109/ASE.2015.9

Ondrej Lhoták and Laurie J. Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Compiler Construction (CC)*. 153–169.

V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*. https://www.usenix.org/conference/14th-usenix-security-symposium/finding-security-vulnerabilities-java-applications-static

Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. 2005. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 365–383. https://doi.org/10.1145/1094811.1094840

Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. Object-Oriented Program Analysis. In *Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 305–315. https://doi.org/10.1145/1806596.1806631

Nomair A. Naeem, Ondrej Lhoták, and Jonathan Rodriguez. 2010. Practical Extensions to the IFDS Algorithm. In *Compiler Construction (CC)*. 124–144.

Mangala Gowri Nanda and Saurabh Sinha. 2009. Accurate Interprocedural Null-Dereference Analysis for Java. In *International Conference on Software Engineering (ICSE)*. 133–143. https://doi.org/10.1109/ICSE.2009.5070515

Rohan Padhye and Uday P. Khedker. 2013. Interprocedural Data Flow Analysis in Soot Using Value Contexts. In *International Workshop on State Of the Art in Java Program analysis, (SOAP)*. 31–36.

Thomas W. Reps. 2000. Undecidability of Context-Sensitive Data-Independence Analysis. *ACM Transactions on Programming Languages and Systems* (2000), 162–186. https://doi.org/10.1145/345099.345137

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Symposium on Principles of Programming Languages (POPL)*. 49–61.

Thomas W. Reps, Akash Lal, and Nicholas Kidd. 2007. Program Analysis Using Weighted Pushdown Systems. In *International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. 23–51.

Thomas W. Reps, Stefan Schwoon, and Somesh Jha. 2003. Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis. In *International Symposium on Static Analysis (SAS) (Lecture Notes in Computer Science)*, Vol. 2694. Springer, 189–213.

Thomas W. Reps, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis. *Science of Computer Programming* (2005), 206–263. https://doi.org/10.1016/j.scico.2005.02.009

Thomas W. Reps, Emma Turetsky, and Prathmesh Prabhu. 2016. Newtonian Program Analysis via Tensor Product. In *Symposium on Principles of Programming Languages (POPL)*. 663–677. https://doi.org/10.1145/2837614.2837659

Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theoretical Computer Science* (1996), 131–170.

Johannes Späth, Karim Ali, and Eric Bodden. 2017. IDE^{al}: Efficient and Precise Alias-Aware Dataflow Analysis. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 99:1–99:27.

Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *European Conference on Object-Oriented Programming (ECOOP)*. 22:1–22:26.

Manu Sridharan and Rastislav Bodík. 2006. Refinement-Based Context-Sensitive Points-To Analysis for Java. In *Programming Language Design and Implementation (PLDI)*. 387–400. https://doi.org/10.1145/1133981.1134027

Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Alias Analysis for Object-Oriented Programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. 196–232.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-Driven Points-To Analysis for Java. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 59–76.

Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *Transactions on Software Engineering (TSE)* (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*. 210–225.

Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Programming Language Design and Implementation (PLDI)*. 87–97. https://doi.org/10.1145/1542476.1542486

Dimitrios Vardoulakis and Olin Shivers. 2010. CFA2: A Context-Free Approach to Control-Flow Analysis. In *European Symposium on Programming (ESOP)*. 570–589. https://doi.org/10.1007/978-3-642-11957-6_30

Guoqing (Harry) Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-Reachability-Based Points-to Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *European Conference on Object-Oriented Programming (ECOOP)*. 98–122.

Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. 2011. Demand-Driven Context-Sensitive Alias Analysis for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*. 155–165.

Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis. In *Programming Language Design and Implementation (PLDI)*. 435–446.

Qirun Zhang and Zhendong Su. 2017. Context-Sensitive Data-Dependence Analysis via Linear Conjunctive Language Reachability. In *Symposium on Principles of Programming Languages (POPL)*. 344–358. http://dl.acm.org/citation.cfm?id=3009848