Two Birds with One Stone: Multi-Derivation for Fast Context-Free Language Reachability Analysis

Chenghang Shi*†, Haofeng Li*¶, Yulei Sui§, Jie Lu*, Lian Li*†‡¶, and Jingling Xue§
*SKLP, Institute of Computing Technology, CAS, China
†University of Chinese Academy of Sciences, China
‡Zhongguancun Laboratory, China
§University of New South Wales, Australia
*{shichenghang21s, lihaofeng, lujie, lianli}@ict.ac.cn
§{y.sui, j.xue}@unsw.edu.au

***Abstract*—Context-free language (CFL) reachability is a fundamental framework for formulating program analyses. CFL-reachability analysis works on top of an edge-labeled graph by deriving reachability relations and adding them as labeled edges to the graph. Existing CFL-reachability algorithms typically adopt a single-reachability relation derivation (SRD) strategy, i.e., one reachability relation is derived at a time. Unfortunately, this strategy can lead to redundancy, hindering the efficiency of the analysis.**

To address this problem, this paper proposes PEARL, a *multi-derivation* approach that reduces derivation redundancy for transitive relations that frequently arise when solving reachability relations, significantly improving the efficiency of CFL-reachability analysis. Our key insight is that multiple edges involving transitivity can be simultaneously derived via batch propagation of reachability relations on the transitivity-aware subgraphs that are induced from the original edge-labeled graph. We evaluate the performance of PEARL on two clients, i.e., context-sensitive value-flow analysis and field-sensitive alias analysis for C/C++. By eliminating a large amount of redundancy, PEARL achieves average speedups of 82.73x for value-flow analysis and 155.26x for alias analysis over the standard CFL-reachability algorithm. The comparison with POCR, a state-of-the-art CFL-reachability solver, shows that PEARL runs 10.1x (up to 29.2x) and 2.37x (up to 4.22x) faster on average respectively for value-flow analysis and alias analysis with less consumed memory.

***Index Terms*—CFL-Reachability, transitive relations**

I. Introduction

Many program analysis problems, such as interprocedural data flow analysis [1], [2], program slicing [3], shape analysis [4], and pointer analysis [5]–[12], can be formulated as context-free language (CFL) reachability problems [13]. The CFL-reachability problem extends standard graph reachability to an edge-labeled graph. The CFL-reachability solving algorithm has a (sub)cubic time complexity with respect to the number of nodes in the edge-labeled graph [14]. Researchers have developed different performance optimization techniques, including reducing the graph size via pre-processing [15]–[20], applying summary-based techniques for caching [1], [3], [21], and adopting efficient data processing techniques to improve scalability [22], [23]. However, despite all these efforts, CFL-reachability algorithms can still suffer from significant performance loss due to redundancy when deriving all-pair reachability relations.

During CFL-reachability solving, an X-reachability relation between source node u and sink node v (i.e., v is X-reachable from u) is explicitly represented as an X-edge $u \xrightarrow{X} v$ in the edge-labeled graph. We use the terms X-edge and X-reachability relation interchangeably, and they are both denoted as $u \xrightarrow{X} v$. A CFL-reachability problem can be converted into a set constraint problem [21], [24]. Let v's X-reachability relations be a set $R(X, v) = \{u \mid u \xrightarrow{X} v\}$. Given a production rule $X ::= YZ$, a Z-edge $u \xrightarrow{Z} v$ specifies the constraint $R(Y, u) \subseteq R(X, v)$. Such a constraint can be solved by propagating u's Y-reachability relations via Z-edge $u \xrightarrow{Z} v$ to produce v's X-reachability relations. Essentially, the edge derivation process of CFL-reachability can be viewed as propagating reachability relations along the edge-labeled graph until a fixed point is reached. During propagation, each newly derived reachability relation (a source-to-sink path) is summarized as a labeled edge and added to the graph, making this reachability relation explicit. Existing CFL-reachability algorithms typically adopt a single-reachability relation derivation (SRD) strategy, i.e., one reachability relation is derived at a time, which can cause redundancy, hindering the efficiency of the analysis.

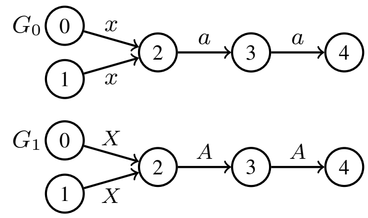
Figure 1 shows an example to illustrate the reachability relation propagations and their redundancy. A context-free grammar (CFG) is given in Figure 1a, where production rules $X ::= x$ and $A ::= a$ suggest that an X-reachability relation and A-reachability relation can be created by a single x-edge and a-edge, respectively, which results in the transformation from input graph G_0 into G_1 in Figure 1b. The other two productions $A ::= AA$ and $X ::= XA$ in the CFG indicate that A-reachability and X-reachability relations can be propagated via an A-edge to produce new A-reachability and X-reachability relations, respectively.

In Figure 1c, Node 1 is omitted from the graph for simplicity, and the A-edge $2 \xrightarrow{A} 4$ (the wavy line) is derived by propagating reachability relation $2 \xrightarrow{A} 3$ via A-edge $3 \xrightarrow{A} 4$.

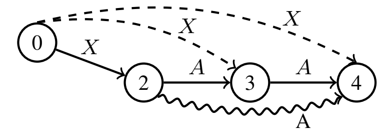
¶ Corresponding author

$$A ::= AA \mid a$$
$$X ::= XA \mid x$$

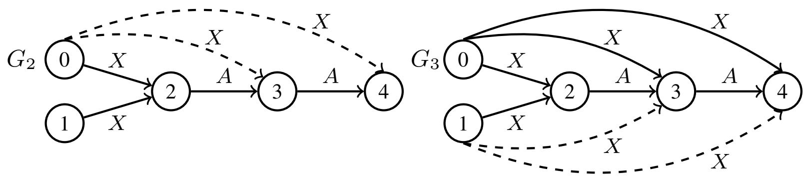
(a) Context-free grammar



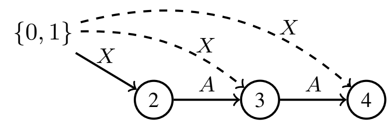
(b) G_0 is the input graph, and G_1 is transformed from G_0 by applying $X ::= x$ and $A ::= a$.



(c) Derivation via propagating reachability relations. Propagating $0 \xrightarrow{X} 2$ along the path $2 \xrightarrow{A} 3 \xrightarrow{A} 4$ derives $0 \xrightarrow{X} 3$ and $0 \xrightarrow{X} 4$ (two dashed lines). Alternatively, $0 \xrightarrow{X} 4$ can also be derived by propagating $0 \xrightarrow{X} 2$ via the edge $2 \xrightarrow{A} 4$ (the wave line).



(d) The Single-reachability Relation Derivation (SRD) manner. $0 \xrightarrow{X} 2$ and $1 \xrightarrow{X} 2$ are *separately* propagated to Nodes 3 and 4. The dashed lines in G_2 and G_3 show the propagation processes of $0 \xrightarrow{X} 2$ and $1 \xrightarrow{X} 2$, respectively.



(e) Our *multi-derivation* approach. $0 \xrightarrow{X} 2$ and $1 \xrightarrow{X} 2$ are packed and propagated together to Nodes 3 and 4. The dashed lines denote the derived edges via the two propagations.

Fig. 1: Motivating Example

Besides, propagating reachability relation $0 \xrightarrow{X} 2$ along path $2 \xrightarrow{A} 3 \xrightarrow{A} 4$ derives two new reachability relations $0 \xrightarrow{X} 3$ and $0 \xrightarrow{X} 4$ (two dashed lines). An alternative derivation of $0 \xrightarrow{X} 4$ is to propagate reachability relation $0 \xrightarrow{X} 2$ via A-edge $2 \xrightarrow{A} 4$.

Motivation. There are two types of *derivation redundancy* during reachability propagation, as follows.

- *Repetitive Derivation Redundancy (RDR)*. Consider the reachability relation between source node 0 and sink node 4 in Figure 1c, reachability relation $0 \xrightarrow{X} 2$ is propagated to Node 4 twice: (1) one via path $2 \xrightarrow{A} 3 \xrightarrow{A} 4$ with two edges; (2) the other one via edge $2 \xrightarrow{A} 4$ (the wavy line). Therefore, there are three propagations for deriving X-reachability relations, and the number doubles when Node 1 is considered. This repetitive derivation causes redundancy since $0 \xrightarrow{X} 4$ is derived more than once.
- *Redundancy due to SRD*. Consider the reachability relations between source nodes 0,1 and sink nodes 3,4, reachability relations $0 \xrightarrow{X} 2$ and $1 \xrightarrow{X} 2$ in Figure 1d are *separately* propagated along the path $2 \xrightarrow{A} 3 \xrightarrow{A} 4$. G_2 and G_3 in Figure 1d summarize the propagation processes of reachability relations $0 \xrightarrow{X} 2$ and $1 \xrightarrow{X} 2$, respectively. In this derivation strategy by existing CFL-reachability analysis, only one reachability relation is derived in one propagation. In total, there are four propagations for deriving X-reachability relations (four dashed lines in G_2 and G_3), and two of them are duplicate derivations.

The standard algorithm [24] exhibits both types of redundancy. The RDR in this example is due to the unawareness of transitivity introduced by production $A ::= AA$. Edge $2 \xrightarrow{A} 4$ (derived using $A ::= AA$) is the *shortcut* of path $2 \xrightarrow{A} 3 \xrightarrow{A} 4$, thus causing a repetitive derivation. A recent CFL-reachability solver POCR [25] adopts a spanning tree model to eliminate RDR for transitive relations. Unfortunately, POCR still suffers from redundancy due to SRD since it follows the propagation process presented in Figure 1d.

Goal and Challenge. This paper aims to boost the efficiency of CFL-reachability analysis by reducing both types of redundancy for transitive relations that arise frequently when solving reachability relations. The challenge is how to effectively identify and eliminate redundancy while efficiently propagating reachability relations, in a large and complex graph with new edges being dynamically introduced during CFL-reachability analysis.

Insight and Solution. Figure 1e outlines the key idea of our *multi-derivation* approach. Reachability relations $0 \xrightarrow{X} 2$ and $1 \xrightarrow{X} 2$ are packed together (i.e., $\{0, 1\} \xrightarrow{X} 2$), and then propagated via $2 \xrightarrow{A} 3$, with two new reachability relations $0 \xrightarrow{X} 3$ and $1 \xrightarrow{X} 3$ simultaneously derived. Multi-derivation aims to infer multiple reachability relations in one propagation, which is achieved by *batch propagation* of packed reachability relations. The propagation via $3 \xrightarrow{A} 4$ also follows this scheme. Our approach is precision-preserving and needs only two propagations (two dashed lines in Figure 1e) to derive identical X-reachability relations with respect to the standard algorithm, which requires six propagations.

In the above example, $A ::= AA$ is called *fully transitive production* and $X ::= XA$ is called *partially transitive production*. Both fully and partially transitive productions can benefit from our multi-derivation approach, which is based on two key observations:

- A fully transitive production (e.g., $A ::= AA$) derives shortcut edges (e.g., $2 \xrightarrow{A} 4$) that can introduce RDR. To reduce repetitive derivations, a *transitivity-aware subgraph* is induced from the edge-labeled graph by excluding shortcut edges for each fully transitive production.
- Instead of separate propagation, a fully/partially transitive

production can be efficiently solved by batch propagation of reachability relations on the transitivity-aware subgraph, with multiple reachability relations derived in one propagation, hence reducing the duplicate derivations due to SRD.

We propose PEARL, a novel multi-derivation approach to solving transitivity efficiently for all-pairs CFL-reachability analysis. We have evaluated PEARL using two popular static analysis clients, i.e., context-sensitive value-flow analysis [26], [27] and field-sensitive alias analysis for C/C++ [7]. Experimental results demonstrate that PEARL is over 82.73x and 155.26x faster than the standard CFL-reachability algorithm for value-flow analysis and alias analysis, respectively. We have also compared PEARL with a state-of-the-art CFL-reachability solver POCR [25]. The results show that PEARL achieves speedups of 10.1x (up to 29.2x) for value-flow analysis and 2.37x (up to 4.22x) for alias analysis over POCR.

To summarize, this paper makes the following contributions:

- We propose a multi-derivation approach that employs a batch propagation technique for fast deriving reachability relations, thereby boosting the efficiency of CFL-reachability analysis. Our approach eliminates repetitive derivations introduced by the standard CFL-reachability algorithm, while also reducing the redundancy which cannot be eliminated by the state-of-the-art solver POCR.
- We present an efficient algorithm to solve both fully and partially transitive productions in a multi-derivation manner by propagating reachability relations in batch on transitivity-aware subgraphs that are induced from the original edge-labeled graph.
- we apply our technique to two popular static analysis clients for C/C++, context-sensitive value-flow analysis and field-sensitive alias analysis with extensive experiments. The empirical results show that our approach can eliminate a large portion of derivation redundancy and significantly improve the performance of CFL-reachability analysis.

The remainder of this paper is structured as follows. Section II introduces the background. Section III briefly illustrates the core idea of our approach with a motivating example. We detail our approach in Section IV and evaluate our tool PEARL in Section V. Section VI surveys related work and Section VII concludes this paper.

II. BACKGROUND

This section briefly reviews the basic background on CFL-reachability and provides related definitions.

A. CFL-reachbility

We start with the basic notations which will be used throughout the paper. Let $CFG = (\Sigma, N, P, S)$ be a context-free grammar over an alphabet Σ comprised of non-terminals N and terminals T, with the start symbol $S \in N$, and a set of production rules P. Let $G(V, E)$ be a directed graph, where V and E are the vertex set and edge set, respectively. Each edge in G is labeled by a symbol from $\Sigma = T \cup N$, e.g., the edge $u \xrightarrow{X} v$ denotes the edge from Node u to Node v labeled by X. Each path in G defines a word over Σ by concatenating the labels of the edges on the path in order. A path is an X*-path* if its word can be derived from $X \in N$ via one or more productions in P. An X-path $u \to \ldots \to v$ implies that an X-reachability relation holds between Node u and Node v (i.e., v is X-reachable from u). CFL-reachability solving is to make such reachability relation explicit by inserting an X-edge $u \xrightarrow{X} v$ into the edge-labeled graph.

Algorithm 1: The standard CFL-reachability algorithm

Input: Normalized $CFG = (\Sigma, N, P, S)$,
  edge-labeled directed graph $G = (V, E)$
Output: all reachable pairs in G

1 **Function** `StandardCFL(`CFG, G`)`**:**
2   Init();
3   Solve(CFG, G);
4 **Procedure** `Init()`**:**
5   add E to W;
6   **for** *each production* $X ::= \varepsilon \in P$ **do**
7   **for** *each node* $v \in V$ **do**
8   **if** $v \xrightarrow{X} v \notin E$ **then**
9   add $v \xrightarrow{X} v$ to E and W
10 **Procedure** `Solve(`CFG, G`)`**:**
11   **while** $W \neq \emptyset$ **do**
12   pop an edge $u \xrightarrow{Y} v$ from W ;
13   **for** *each production* $X ::= Y \in P$ **do**
14   **if** $u \xrightarrow{X} v \notin E$ **then**
15   add $u \xrightarrow{X} v$ to E and W
16   **for** *each production* $X ::= YZ \in P$ **do**
17   **for** *outgoing edge* $v \xrightarrow{Z} w$ *from node* v **do**
18   **if** $u \xrightarrow{X} w \notin E$ **then**
19   add $u \xrightarrow{X} w$ to E and W
20   **for** *each production* $X ::= ZY \in P$ **do**
21   **for** *incoming edge* $w \xrightarrow{Z} u$ *to node* u **do**
22   **if** $w \xrightarrow{X} u \notin E$ **then**
23   add $w \xrightarrow{X} u$ to E and W

The Standard Algorithm. In the literature, CFL-reachability is solved by the standard dynamic programming algorithm [24] given in Algorithm 1. The algorithm requires the CFG to be normalized in such a way that the right-hand side of each production has at most two symbols, i.e., productions are in the form $X ::= YZ$, $X ::= Y$ or $X ::= \varepsilon$. Let W denote a worklist. Algorithm 1 first initializes the worklist with all original edges (line 5) and then adds all self-referencing edges ($v \xrightarrow{X} v$) produced by productions $X ::= \varepsilon$

into the graph and worklist (Lines 6-9).

Next, the procedure SOLVE is invoked to iteratively derives new edges until no more edges can be deduced ($W = \emptyset$). Given an edge $u \xrightarrow{Y} v$, an edge $u \xrightarrow{X} v$ is derived using the production rule $X := Y$ (lines 13 - 15). In addition, each outgoing Z-edge of Node v (lines 16 - 19) and incoming Z-edge (lines 20 - 23) of Node u is examined to derive new X edges via the productions $X ::= YZ$ and $X ::= ZY$, respectively. All newly derived edges are added to the graph and to the worklist for further processing.

The standard algorithm exhibits a *single-reachability relation derivation* style, i.e., each reachability relation (e.g., $u \xrightarrow{Y} v$) of Node v is separately handled in distinct iterations (Line 12).

Graph Representation. Given an X-edge $u \xrightarrow{X} v$, we say Node u is an X-predecessor of Node v, and Node v is an X-successor of Node u. Consequently, the *X-predecessor set* of Node v, denoted as $R(X, v) = \{ u \mid u \xrightarrow{X} v \in E \}$, represents all incoming X-edges of Node v (used at Line 21 in Algorithm 1), and the *X-successor set* of node v, denoted as $S(X, v) = \{ u \mid v \xrightarrow{X} u \in E \}$, represents all outgoing X-edges of Node v (used at Line 17 in Algorithm 1). The X-predecessor set of Node v is also called the *X-reachability relation set* of Node v.

B. Transitive Production Rule

Definition 1. (Fully Transitive Production). A *fully transitive production* is in the form $A ::= AA$. Relation A is a *fully transitive relation* if and only if it is in a fully transitive production.

Definition 2. (Partially Transitive Production). A *left (right) transitive production* is in the form $X ::= XA$ ($X ::= AX$) where relation A is fully transitive and $X \neq A$. A *partially transitive production* is either left transitive or right transitive.

Relation X is a *partially transitive relation* if and only if it is on the left side of a partially transitive production. Accordingly, edges of fully (partially) transitive relations are called *fully (partially) transitive edges*. We classify fully transitive edges into two categories [25] by the *first* production that generates it:

- *Secondary edge*. A fully transitive edge is a secondary edge if it is first derived via a fully transitive production;
- *Primary edge*. A fully transitive edge is a primary edge if it is first derived via a production that is not fully transitive.

For convenience, fully transitive relations (edges) and partially transitive relations (edges) are collectively denoted as *transitive relations (edges)*.

Definition 3. (Transitive Production Rule). A transitive production rule is either fully transitive or partially transitive. Accordingly, other production rules are defined as *non-transitive production rules*.

Transitive production rules have a nice property, e.g., production $X ::= XA$ suggests that X-reachability relations can be propagated via A-edges while preserving their edge label X.

C. The POCR Approach

A recent solver POCR [25] addresses the repetitive derivation redundancy (RDR) by utilizing a spanning tree model for each fully transitive production. Specifically, given the graph G_1 in Figure 1b, for A-reachability relation, three spanning trees are constructed:

(1) $2 \xrightarrow{A} 3 \xrightarrow{A} 4$ rooted from Node 2;
(2) $3 \xrightarrow{A} 4$ rooted from Node 3;
(3) a tree rooted from Node 4 without children.

Given the reachability relation $0 \xrightarrow{X} 2$, the spanning tree rooted from Node 2 is traversed to derive two reachability relations, $0 \xrightarrow{X} 3$ and $0 \xrightarrow{X} 4$.

There are two reasons why we do not choose this spanning tree as the underlying representation of our approach. First, the A-transitivity information of each node is maintained *individually* in distinct spanning trees, and hence X-reachability relations of different root nodes can not be packed together. Second, the reachability information is redundantly preserved, e.g., $3 \xrightarrow{A} 4$ is copied into at least two spanning trees mentioned above, which can be expensive in terms of time and space when A-reachability relations are dense, as confirmed in our experiments (Section V).

III. PEARL IN A NUTSHELL

In this section, we briefly illustrate how our *multi-derivation* approach can effectively reduce derivation redundancy.

A. Transitivity-aware Propagation Graph

For each fully transitive relation A, our approach maintains a *propagation graph*, denoted as $PG(A)$.

Definition 4. (Propagation Graph). Given an edge-labeled Graph $G(V, E)$, the propagation graph $PG(A)$ for a fully transitive relation A is the subgraph induced from G with primary A-edges, i.e., $PG(A) = (V', E')$, where edge set E' consists of all the primary A-edges in E, and vertex set V' consists of the endpoints of E'.

$PG(A)$ is *transitivity-aware* since it implicitly preserves all A-reachability relations in the original graph G [28], [29]. The spanning tree model of POCR [25] is not suitable for our batch propagation technique because it maintains transitivity for each node separately.

B. Solving Transitive Productions via Multi-derivation

Edge derivations using fully and partially transitive productions are transformed into computing transitive closures of propagation graphs. For example, given a partially transitive production $X ::= XA$, an A-edge $u \xrightarrow{A} v$ specifies the constraint $R(X, u) \subseteq R(X, v)$ [21], [24], which is solved by our *multi-derivation* approach via propagating reachability relations in batch on $PG(A)$.

Next, we highlight how transitive production $X ::= XA$ is solved for our motivating example in Figure 1.

(1) **Propagation Graph Construction**. Given the graph G_1 in Figure 1b, $PG(A)$ is constructed by selecting only primary A-edges, i.e., the A-path $2 \xrightarrow{A} 3 \xrightarrow{A} 4$.
(2) **Packing Reachability Relations**. Two reachability relations $0 \xrightarrow{X} 2$ and $1 \xrightarrow{X} 2$ are packed together as $\{0, 1\} \subseteq R(X, 2)$.
(3) **Multi-derivation via Batch Propagation**. Edges $2 \xrightarrow{A} 3$ and $3 \xrightarrow{A} 4$ indicate constraints $R(X, 2) \subseteq R(X, 3)$ and $R(X, 3) \subseteq R(X, 4)$, respectively, which are solved by propagating reachability $\{0, 1\} \subseteq R(X, 2)$ in batch along edges $2 \xrightarrow{A} 3$ and $3 \xrightarrow{A} 4$ in $PG(A)$.

Reducing Derivation Redundancy. By excluding the secondary edge $2 \xrightarrow{A} 4$ from $PG(A)$, we avoid solving the trivial constraint $R(X, 2) \subseteq R(X, 4)$ that is already implied in two constraints $R(X, 2) \subseteq R(X, 3)$ and $R(X, 3) \subseteq R(X, 4)$, thereby eliminating repetitive derivations. Moreover, our multi-derivation approach solves production $X ::= XA$ via batch propagation of X-reachability relations on $PG(A)$, thus reducing the redundancy due to single-reachability relation derivation.

IV. The Methodology

Algorithm 2: Overall algorithm of Pearl

Input: Normalized $CFG = (\Sigma, N, P, S)$,
edge-labeled directed graph $G = (V, E)$
Output: all reachable pairs in G

1 **Function** `PearlCFL(`CFG`,`G`)`**:**
2   Init(); // Lines 4-9 in Algo. 1
3     $CFG_{nt} \leftarrow CFG$ without transitive productions;
4   **while** $W \neq \emptyset$ **do**
5   Solve(CFG_{nt}, G); // Lines 10-23 in Algo. 1
6   add derived partially transitive edges to $PSet$;
7   add derived fully transitive edges to $FSet$;
8   PropPTR($PSet$); // Algo. 3
9   PropFTR($FSet$); // Algo. 4
10         $PSet$.clear(); $FSet$.clear();
11   add new transitive edges to W;

As shown in Algorithm 2, Pearl maintains non-transitive, fully transitive, and partially transitive relations in 3 distinct worklists (W, $FSet$, and $PSet$) and solves them in different manners. Given an input grammar $CFG = (\Sigma, N, P, S)$, $CFG_{nt} = (\Sigma, N, P_{nt}, S)$ denotes the grammar modified from CFG with all transitive production rules excluded. The initialization phase (Line 2) of Pearl is the same as the standard algorithm (Algorithm 1)

In each loop iteration, non-transitive production rules are firstly solved in the standard single-reachability relation derivation (SRD) manner by applying the same Solve procedure on the modified grammar CFG_{nt} (Lines 5 and 7). After this step, no more new edges can be derived from non-transitive production rules. Next, partially and fully transitive productions are solved in a *multi-derivation* manner by functions PropPTR and PropFTR, respectively (Lines 8,9): new transitive relations derived from non-transitive production rules (Init at Line 2 and Solve at line 5) are propagated in the propagation graphs to discover all other transitive relations. After this stage, no more new edges can be derived from transitive production rules. However, the newly derived transitive edges may enable derivations via non-transitive production rules. Hence, at the end of each iteration, newly derived transitive edges (created by PropPTR at Line 8 and PropFTR at Line 9) in the current iteration are pushed to the worklist W (Line 11), triggering the next iteration of the loop.

A. Multi-derivation via Propagating Partially Transitive Relations

Unifying Partially Transitive Productions. Without loss of generality, we assume that $X \neq A$ for a given production $X ::= XA$ and a fully transitive production will be given in an explicit form of $A ::= AA$. We have $\overline{A} ::= \overline{A}\ \overline{A}$ ($\overline{A}$ is fully transitive) $\iff$ $A ::= AA$ (A is fully transitive), and right transitive production $X ::= AX$ can be transformed into left transitive production $\overline{X} ::= \overline{X}\ \overline{A}$ by reversing the original production [4]. By adopting this "reversing" transformation, right transitive productions are handled in the same fashion as left transitive productions.

For a relation X and its inverse relation $\overline{X}$, X-successors and $\overline{X}$-predecessors mean exactly the same thing. Hence, we only need to maintain X-predecessor set and $\overline{X}$-predecessor set to represent both relations X and $\overline{X}$. Therefore, when compared to the standard graph representation introduced in Section II-A, inverse relations introduced by reversing transformation do not incur memory overhead.

Propagating Partially Transitive Relations. Algorithm 3 solves partially transitive productions in a multi-derivation manner by batch propagation of reachability relations on the propagation graphs. Our implementation adopts the popular *difference propagation* technique [30]–[32]. In particular, $R(X, v)$ and $R_{new}(X, v)$ contains "old" X-reachability relations and "new" X-reachability relations of Node v, respectively. Besides, P_{gt} denotes the production rule set transformed from the original P by translating right transitive productions into left transitive ones. To avoid confusion with the edge worklist W in Algorithm 1, we use NW to denote *node worklist*. The algorithm involves two steps (Lines 2-5):

(1) *Packing Reachability Relations*. Given a newly derived reachability relation $u \xrightarrow{X} v$, Node u is added to $R_{new}(X, v)$, and Node v is added to $R_{new}(\overline{X}, u)$ (Lines 2-4). Both Nodes u and v are then pushed into the NW (Lines 23-25).
(2) *Propagating Reachability Relations*. The procedure PropRRs (Line 6) propagates new reachability relations in batch along the propagation graph using a standard worklist algorithm [32].

In PropRRs, a Node u is selected from NW at each

Algorithm 3: Multi-derivation via Propagating partially transitive relations

Input: the set of partially transitive relations in $PSet$
Output: partially transitive relations generated via multi-derivation

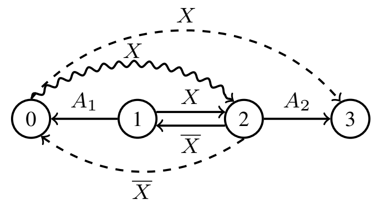
1 **Function** PropPTR($PSet$):
2   **for** *each edge* $u \xrightarrow{X} v \in PSet$ **do**
3   PackRR(X, u, v);
4   PackRR($\overline{X}, v, u$);
5   PropRRs();
6 **Procedure** PropRRs():
7   **while** $NW \neq \emptyset$ **do**
8   pop node u from NW;
9   **for** *each partially transitive relation* X **do**
10             $R(X, u) \leftarrow R(X, u) \cup R_{new}(X, u)$;
11             $\Delta R(X, u) \leftarrow R_{new}(X, u)$;
12             $R_{new}(X, u) \leftarrow \emptyset$;
13   **for** *partially transitive production* $X ::= XA \in P_{gt}$ **do**
14   **for** $u \xrightarrow{A} v \in PG(A)$ **do**
15   DiffProp(X, $\Delta R(X, u)$, v);
16   **for** *each node* $v \in \Delta R(X, u)$ **do**
17   PackRR($\overline{X}$,u, v);
18 **Procedure** DiffProp(X, $srcSet$, v):
19     $R_{new}(X, v) \leftarrow R_{new}(X, v) \cup (srcSet \setminus R(X, v))$;
20   **if** $R_{new}(X, v)$ *changes* **then**
21   add v to NW;
22 **Procedure** PackRR(X, u, v):
23     $R_{new}(X, v) \leftarrow R_{new}(X, v) \cup \{u\}$;
24   **if** $\exists$ *partially transitive production* $X ::= XA \in P_{gt}$ **then**
25   add v to NW;

iteration (Line 8), then $R_{new}(X, u)$ is merged into $R(X, u)$ and flushed (Lines 10-12). Since a partially transitive relation X may be involved in multiple transitive productions, the algorithm visits each left transitive production $X ::= XA$ (Line 13). Then $\Delta R(X, u)$ is propagated via each outgoing primary A-edge of Node u (Lines 14-15) by invoking DIFFPROP. Nodes receiving new reachability relations are then pushed into NW (Lines 19-21). In this way, all reachability relations in $\Delta R(X, u)$ are simultaneously processed in one propagation, while the single-reachability relation derivation strategy in Algorithm 1 separately processed each X-reachability relation of Node u in distinct iterations.

The reversed relation $\overline{X}$ can be also partially transitive. In this case, new X-reachability relations (created at Line 11) are translated into $\overline{X}$-reachability relations (Lines 16-17), which will trigger the propagation of $\overline{X}$-reachability relations.

$$\overline{X} ::= \overline{X} A_1$$
$$X ::= X A_2$$

(a) Context-free grammar



(b) Edge-labeled graph

Fig. 2: Relations X and $\overline{X}$ are both partially transitive

Example 1. Given the CFG in Figure 2a and the input graph in Figure 2b, let both relations A_1 and A_2 be fully transitive. We show the derivation process of edge $0 \xrightarrow{X} 3$, which involves two partially transitive productions $\overline{X} ::= \overline{X} A_1$ and $X ::= X A_2$, as follows.

(1) Propagate reachability relation $2 \xrightarrow{\overline{X}} 1$ via edge $1 \xrightarrow{A_1} 0$ to derive $2 \xrightarrow{\overline{X}} 0$.
(2) *Generate $\overline{X}$-reachability relation.* $2 \xrightarrow{\overline{X}} 0 \implies 0 \xrightarrow{X} 2$ (the wavy line).
(3) Propagate reachability relation $0 \xrightarrow{X} 2$ via edge $2 \xrightarrow{A_2} 3$ to derive $0 \xrightarrow{X} 3$.

B. Multi-derivation via Propagating Fully Transitive Relations

Algorithm 4 updates propagation graphs on the fly and solves fully transitive productions in a multi-derivation manner via batch propagation of fully transitive reachability relations. For each new primary A-edge $u \xrightarrow{A} v$, $PG(A)$ is updated (Line 3) by including the edge if $u \notin R(A, v)$. Consequently, we propagate partially transitive relations $R(X, u)$ to Node v (Line 5-6), by invoking function DIFFPROP in Algorithm 3.

In UPDATEPG, both propagation graphs $PG(A)$ and $PG(\overline{A})$ are updated by including $u \xrightarrow{A} v$ and $v \xrightarrow{\overline{A}} u$, respectively (Lines 9,10). Next, the transitive closures of the $PG(A)$ and $PG(\overline{A})$ are updated by propagating the new reachability relations introduced by $u \xrightarrow{A} v$ (i.e., $\{u\} \cup R(A, u)$ at Line 11) in a depth-first manner (Line 12). Only new reachability relations are propagated along $PG(A)$ (Lines 14-18) to avoid redundant work. The reverse relation $\overline{A}$ is updated together when updating A-edges (Lines 19-20).

Eager Propagation. The procedure DFS eagerly propagates A-reachability relations to compute the transitive closures of $PG(A)$. Alternatively, A-reachability relations can be iteratively propagated using a worklist as in Section IV-A. However, in our earlier implementation, we observe that iterative propagation of fully transitive relations introduces redundant secondary edges in $PG(A)$, causing performance degradation. Let us consider the below example.

Example 2. There are three steps in Figure 3, with an inserted edge (the dashed line) at each step, as follows.

- *Step(a)*, insert $1 \xrightarrow{A} 2$, we have $\{1\} \subseteq R(A, 2)$.
- *Step(b)*, insert $0 \xrightarrow{A} 1$, we have $\{0\} \subseteq R(A, 1)$. Additionally, Node 2 is A-reachable from Node 0, but this infor-

Algorithm 4: Multi-derivation via Propagating fully reachability relations

Input: the set of fully transitive relations in $FSet$
Output: fully transitive relations generated via multi-derivation

1 **Function** `PropFTR(`$FSet$`)`**:**
2   **for** *each edge* $u \xrightarrow{A} v \in FSet$ **do**
3   UpdatePG(A, u, v);
4   **if** $u \xrightarrow{A} v$ *is added into* $PG(A)$ **then**
5   **for** *each* $X ::= XA \in P_{gt}$ **do**
6   DiffProp($X, R(X, u), v$); // Lines 18-21 in Algo. 3
7 **Procedure** `UpdatePG(`A, u, v`)`**:**
8   **if** $u \notin R(A, v)$ **then**
9   add $u \xrightarrow{A} v$ to $PG(A)$;
10   add $v \xrightarrow{\overline{A}} u$ to $PG(\overline{A})$;
11     $srcSet \leftarrow R(A, u) \cup \{u\}$;
12   DFS(A, $srcSet$, v);
13 **Procedure** `DFS(`$A, srcSet, u$`)`**:**
14   $\Delta R(A, u) \leftarrow srcSet \setminus R(A, u)$;
15   **if** $\Delta R(A, u) \neq \emptyset$ **then**
16     $R(A, u) \leftarrow R(A, u) \cup \Delta R(A, u)$;
17   **for** $u \xrightarrow{A} v \in PG(A)$ **do**
18   DFS($A, \Delta R(A, u), v$);
19   **for** $v \in \Delta R(A, u)$ **do**
20       $R(\overline{A}, v) \leftarrow R(\overline{A}, v) \cup \{u\}$;

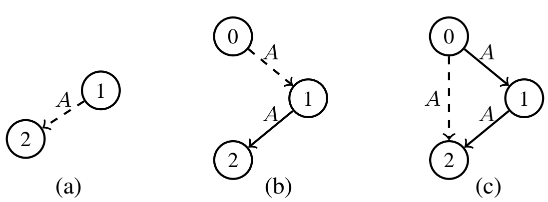


Fig. 3: Iterative propagation for production $A ::= AA$

mation is not explicit until we propagate $\{0\} \subseteq R(A, 1)$ to Node 2.

- *Step(c)*, the secondary edge $0 \xrightarrow{A} 2$ is mistakenly inserted to $PG(A)$ because $0 \notin R(A, 2)$ (Line 8 in Algorithm 4).

Unlike iterative propagation (which tends to collect more reachability relations before propagation), eager propagation appears to propagate fewer reachability relations in one batch. However, it keeps $PG(A)$ sparse by excluding secondary edges. In this example, eagerly propagating $\{0\} \subseteq R(A, 1)$ to Node 2 at *step(b)* avoids inserting $0 \xrightarrow{A} 2$ into $PG(A)$ *at step(c)*.

Memory Overhead of Propagation Graph. Given a fully transitive relation A, when the CFL-reachability algorithm reaches a fixed point, let E_A the set of all A-edges. According to Definition 4, E_A is the transitive closure of $PG(A)$, hence edges in $PG(A)$ are negligible when compared to E_A. Additionally, we construct $PG(\overline{A})$ for relation $\overline{A}$ only when relation $\overline{A}$ occurs in a partially transitive production or a fully transitive production.

Insertion Order. Suppose that in Figure 3, the insertion order is $0 \xrightarrow{A} 2$, $0 \xrightarrow{A} 1$ and $1 \xrightarrow{A} 2$. Then even with eager propagation, $0 \xrightarrow{A} 2$ will cause redundant propagation but is still kept in $PG(A)$. Based on our experience, such redundant edges only occupy a small portion, making them acceptable. Besides, identifying such edges, i.e., online transitive reduction [28], would also incur additional costs.

Completeness. For illustration, Algorithm 3 and Algorithm 4 do not consider the corner case when a reachability relation is both fully transitive and partially transitive, e.g., relation X involves both fully transitive production $X ::= XX$ and partially transitive production $X ::= XA$. Although such case rarely occurs in CFL-based program analyses, we propose two options to complete our algorithm:

- Option 1: Rewrite the production $X ::= XA$ to three new productions: $X' ::= X'A$, $X' ::= X$, and $X ::= X'$, which introduces additional X'-edges.
- Option 2: Update $PG(X)$ using new X-edges created via $X ::= XA$ (Line 11 in Algorithm 3); and propagate new X-reachability relations created via $X ::= XX$ (Line 14 in Algorithm 4) on $PG(A)$.

C. Correctness

Theorem 1. PEARL yields the identical analysis result with respect to the standard CFL-reachability algorithm (Algorithm 1).

Proof sketch. PEARL differs from the standard algorithm in handling transitive productions, manifest in three forms: $A ::= AA$, $X ::= XA$ and $X ::= AX$ (transformed into $\overline{X} ::= \overline{X}\ \overline{A}$). Here we only prove the correctness of the case $X ::= XA$, as the proof for other productions is similar.

Edge $v_0 \xrightarrow{X} v_n$ is derived based on $X ::= XA$ by Algorithm 1 only if it is formed by an X-edge $v_0 \xrightarrow{X} v_1$ and an A-edge $v_1 \xrightarrow{X} v_n$. We prove the correctness by two properties:

- *Soundness*. $v_1 \xrightarrow{X} v_n$ is either a primary edge or secondary edge. Either way, v_n is reachable from v_1 in $PG(A)$ since all A-edges can be connected by one or more primary edges. Therefore, by propagating reachability relation $v_0 \xrightarrow{X} v_1$ along $PG(A)$, $v_0 \xrightarrow{X} v_n$ is derived when PEARL obtains a fixed point.
- *Completeness*. $PG(A)$ contains no spurious A-edges, so a X-reachable path in PEARL is always X-reachable in Algorithm 1.

Thus, PEARL and the standard algorithm computes identical X-edge set for $X ::= XA$. □

V. EVALUATION

We evaluate the performance of PEARL on two practical static analysis clients: context-sensitive value-flow anal-

$$A ::= A\ A \mid call_i\ A\ ret_i \mid a \mid \varepsilon$$

(a) Context-free grammar

$$A ::= A\ A \mid CA_i\ ret_i \mid a \mid \varepsilon$$
$$CA_i ::= call_i\ A$$

(b) Normalized grammar

Fig. 4: CFG for context-sensitive value-flow analysis

$$M ::= \overline{d}\ V\ d$$
$$V ::= \overline{A}\ V\ A \mid \overline{f_i}\ V\ f_i \mid M \mid \varepsilon$$
$$\underline{A} ::= \underline{A}\ \underline{A} \mid a\ M? \mid \varepsilon$$
$$\overline{A} ::= \overline{A}\ \overline{A} \mid M?\ \overline{a} \mid \varepsilon$$

(a) Context-free grammar

$$DV ::= \overline{d}\ V$$
$$M ::= DV\ d$$
$$FV_i ::= \overline{f_i}\ V$$
$$V ::= \overline{A}\ V \mid V\ A \mid FV_i\ f_i \mid M \mid \varepsilon$$
$$\underline{A} ::= \underline{A}\ \underline{A} \mid a\ M \mid a \mid \varepsilon$$
$$\overline{A} ::= \overline{A}\ \overline{A} \mid M\ \overline{a} \mid \overline{a} \mid \varepsilon$$

(b) Normalized grammar

Fig. 5: CFG for field-sensitive alias analysis

ysis [26], [27] and field-sensitive alias analysis (extended from [7]) for C/C++.

Baselines. The baseline of our experiment is a state-of-the-art solver, POCR [25], which has been open-sourced on Github[1]. For completeness, we have also included performance statistics of the standard CFL-reachability algorithm(Algorithm 1) for reference.

Implementation. We have implemented PEARL on top of LLVM-14.0.0 and SVF [33], a popular static analysis framework. All codes including baselines are compiled using gcc-12.2.0 with the commonly used "-O2" optimization flag.

Our evaluation aims to answer the following research questions:

- **(RQ1).** How extensive are fully and partially transitive edges in real-world CFL-reachability problems based on the two clients?
- **(RQ2).** How about the overall performance of PEARL when comparing it with existing approaches?
- **(RQ3).** Is propagation graph representation effective in reducing repetitive derivation redundancy?
- **(RQ4).** Is batch propagation effective in reducing redundancy due to single-reachability relation derivation?

A. *Experimental Setup*

Environment. All the experiments are conducted on a machine with a Intel(R) Xeon(R) Gold 5317 CPU @ 3.00GHz and 1 TB of physical memory. We run the experiments with a time limit of 6 hours and a memory limit of 512 GB.

Value-flow Analysis. We perform context-sensitive value-flow analysis on the sparse value-flow graphs (SVFG) [26], [27]. Figure 4a shows the context-free grammar (CFG) for value-flow analysis, where $call_i$ and ret_i denote parameter passing and return flow at the i-th callsite respectively, a denotes an assignment, and A denotes an intraprocedural/interprocedural value flow. The analysis is also field-sensitive since each field object is represented as a single node in the SVFG. The normalized grammar is listed in Figure 4b.

Alias Analysis. The field-sensitive alias analysis for C++ is conducted on the program expression graph (PEG) [7]. Figure 5a presents the CFG, where a denotes an assignment statement, d denotes a pointer dereference, f_i denotes the address of i-th field, A denotes a value flow, M denotes memory alias, and V denotes value alias. PEG is bidirected [8], [34], i.e., for an edge $u \xrightarrow{X} v$ in PEG, there is a reverse edge $v \xrightarrow{\overline{X}} u$ in PEG. The normalized grammar is shown in Figure 5b.

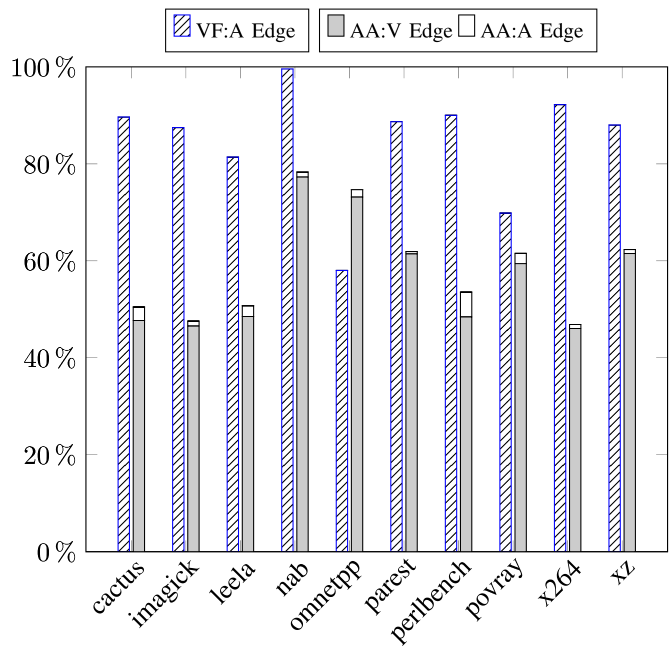


Fig. 6: The percentages of (partially) transitive edges among all inserted edges

Setup and Benchmarks. We use the benchmarks[2] provided by POCR [25]. These benchmarks contain SVFG and PEG of 10 SPEC 2017 C/C++ programs. Following POCR, SVFG and PEG are preprocessed by cycle elimination [15] to collapse cycles of a-edges and variable substitution [16] to compact particular a-edges. In Table I and Table II, columns 2-5 list the numbers of nodes and edges of SVFG and PEG before and after offline preprocessing in each benchmark.

Evaluation of Correctness. The correctness of Theorem 1 is verified practically by the fact that PEARL and the standard algorithm (if scalable) compute the same set of reachable pairs in our experiments.

B. *RQ1. Transitive Edges*

Figure 6 illustrates the percentages of fully/partially transitive edges among all edges added to the edge-labeled graph

[1] https://github.com/kisslune/POCR

[2] https://github.com/kisslune/CPU17-graphs

TABLE I: Result of value-flow analysis. Column 2-5 gives the numbers of nodes and edges before and after preprocessing. “SPU” stands for speedup. Column 9 shows the speedups of POCR over STD, and column 12 shows the speedups of PEARL over POCR. The remaining columns give the time and memory consumption of evaluated algorithms. Time in seconds, memory in GB. “-” means exceeding the time limit (6 hours).

id	Before Prep.		After Prep.		STD		POCR			**PEARL**			**PEARL-WB**	
	#Nodes	#Edges	#Nodes	#Edges	Time	Mem	Time	SPU	Mem	Time	SPU	Mem	Time	Mem
cactus	544480	1007989	223046	616399	3408.36	3.46	604.10	5.6x	40.26	28.26	21.4x	4.74	38.53	4.73
imagick	574089	842509	165096	319141	583.71	0.43	59.13	9.9x	5.87	5.18	11.4x	0.74	6.33	0.73
leela	64466	89081	21711	40409	1.58	0.02	0.47	3.4x	0.19	0.16	2.9x	0.02	0.17	0.02
nab	55652	72366	15415	23736	55.51	0.50	16.59	3.3x	4.34	3.27	5.1x	0.32	5.99	0.32
omnetpp	664358	1857831	237854	1277123	229.26	1.08	15.49	14.8x	3.99	3.62	4.3x	0.53	3.57	0.51
parest	299718	407343	114099	199793	2.40	0.07	0.67	3.6x	0.19	0.38	1.8x	0.07	0.33	0.06
perlbench	697744	1662445	321778	1122795	16366.80	6.35	1520.19	10.8x	63.57	52.06	29.2x	10.42	106.18	10.39
povray	537775	1041687	213130	621400	5834.13	5.05	655.14	8.9x	55.84	43.91	14.9x	4.72	51.08	4.70
x264	207064	340217	66417	162595	194.16	0.70	34.77	5.6x	6.46	4.71	7.4x	0.67	7.06	0.66
xz	49395	62955	15072	23002	0.54	0.01	0.16	3.4x	0.06	0.06	2.7x	0.01	0.06	0.01

TABLE II: Result of alias analysis. Column 2-5 gives the numbers of nodes and edges before and after preprocessing. “SPU” stands for speedup. Column 9 shows the speedups of POCR over STD, and column 12 shows the speedups of PEARL over POCR. The remaining columns give the time and memory consumption of evaluated algorithms. Time in seconds, memory in GB. “-” means exceeding the time limit (6 hours).

id	Before Prep.		After Prep.		STD		POCR			**PEARL**			**PEARL-WB**	
	#Nodes	#Edges	#Nodes	#Edges	Time	Mem	Time	SPU	Mem	Time	SPU	Mem	Time	Mem
cactus	93557	212478	65232	153470	-	-	191.27	-	11.62	96.59	2.0x	9.28	170.67	8.56
imagick	119314	301846	73499	196730	-	-	554.13	-	42.55	334.76	1.7x	41.41	544.86	39.21
leela	22186	49748	14371	33326	312.28	0.31	3.40	91.8x	0.39	2.24	1.5x	0.36	3.30	0.33
nab	16261	34676	8794	19218	7.12	0.10	0.76	9.4x	0.10	0.18	4.2x	0.09	0.68	0.10
omnetpp	241916	509166	146049	311980	-	-	410.79	-	17.96	195.77	2.1x	17.08	369.44	16.38
parest	117500	251436	67949	148286	-	-	92.77	-	4.79	42.10	2.2x	4.69	87.67	4.40
perlbench	139183	348916	72231	192994	-	-	1733.42	-	110.84	978.29	1.8x	80.30	1673.79	75.51
povray	76405	174258	45622	110732	14699.10	3.24	160.97	91.3x	6.60	58.64	2.7x	5.55	146.14	5.41
x264	60956	136352	40625	94110	1056.20	1.31	11.13	94.9x	1.05	3.39	3.3x	1.00	9.59	0.99
xz	12425	26468	7130	15228	6.67	0.05	0.42	15.9x	0.07	0.19	2.2x	0.07	0.38	0.07

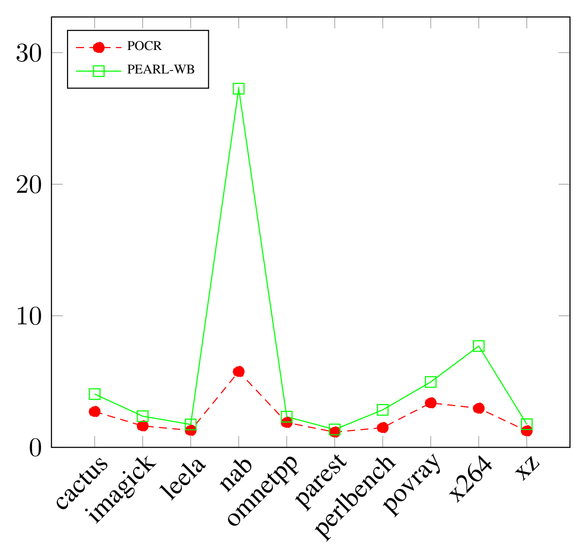


Fig. 7: Computational redundancy of value-flow analysis

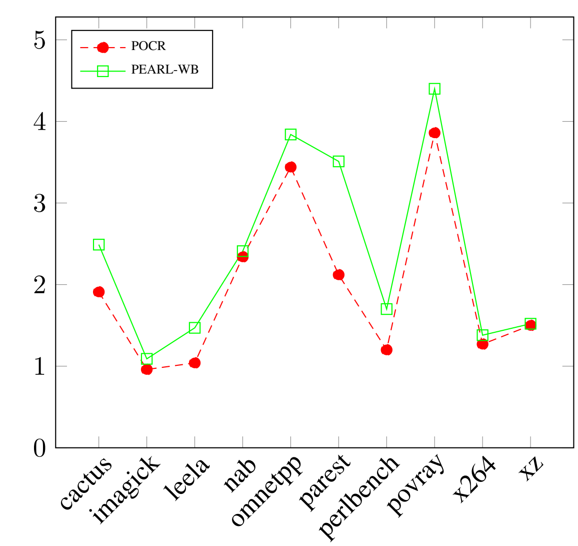


Fig. 8: Computational redundancy of alias analysis

during analysis. “VF:A Edge” represents A-edges in value-flow analysis, while “AA:A Edge” and “AA:V Edge” represent A-edges and V-edges in alias analysis, respectively.

Result. In value-flow analysis, fully transitive edges (A-edges) account for a percentage of 84.51% on average and there are no partially transitive edges. In alias analysis, fully transitive edges (A-edges) represent a negligible proportion (1.8%) while partially transitive edges (V-edges) occupy 57.01% of total added edges on average.

Discussion. As shown in columns 2-5 of Table I and II, offline preprocessing has already pruned a large number of fully transitive edges. However, transitive edges still make up a significant proportion of all edges added during reachability solving in value-flow analysis (84.51%) and alias analysis

(58.81%). Thus, it is essential to accelerate edge derivations involving transitive edges to efficiently solve CFL-reachability problems.

C. RQ2. Performance Evaluation

Table I and Table II display the performance of three algorithms in value-flow analysis and alias analysis, respectively. In both tables, "STD", "POCR" and "PEARL" denote the standard algorithm [24], POCR [25], and our multi-derivation approach, respectively. Besides, column 9 shows the speedup of POCR over STD, and column 12 shows the speedup of PEARL over POCR. The time (measured in seconds) and memory (measured in GB) consumption of each algorithm are shown in the corresponding sub-columns. "-" indicates that the algorithm exceeds the time limit (6 hours).

Result. In terms of performance, PEARL outperforms two baselines on all benchmarks.

Value-flow analysis. In Table I, PEARL is over 82.73x faster than STD (the standard algorithm) on average. Comparison to POCR shows that PEARL achieves an average speedup of 10.1x over POCR, with a maximum improvement of 29.2x for *perlbench*. It is worth noting that PEARL solves each benchmark for value-flow analysis within one minute.

Alias analysis. In Table II, STD timeouts for 5 benchmarks. PEARL runs 155.26x faster than the standard algorithm on average for the 5 completed benchmarks. Compared to POCR, PEARL achieves a performance improvement of 2.37x over POCR on average.

Memory usage. Table I and Table II demonstrate that PEARL only introduces moderate memory overhead over STD. On the other hand, PEARL consumes less memory than POCR for almost all benchmarks. In value flow analysis (Table I), where the fully transitive edges dominate, PEARL achieves significant memory savings compared to POCR. For instance, in the *cactus* benchmark, POCR utilizes nearly 40 GB of memory, whereas PEARL only requires less than 5 GB of memory.

Discussion. By efficiently solving transitivity in a multi-derivation manner, PEARL obtains promising speedups over the standard algorithm and POCR for both clients.

D. RQ3. Effectiveness of Propagation Graph Representation

To evaluate the effectiveness of the propagation graph representation in eliminating repetitive derivations, we design an ablation PEARL-WB, which employs the propagation graph representation but *without* our batch propagation technique. We compare PEARL-WB with POCR, which adopts a spanning tree model, in terms of the reduced derivations and overall performance.

Reduced Derivations. The percentage of repetitive derivations is computed by $(D - A)/D$, where D and A are the number of total derivations and the number of edges added to the graph. The standard algorithm's repetitive derivations account for 99.31% and 99.95% in value-flow analysis and alias analysis, respectively. On average, POCR and PEARL-WB eliminate 99.84% and 99.59% of the repetitive derivations over the standard algorithm for two clients, respectively. Additionally, Figure 7 and Figure 8 evaluate the *computational redundancy* defined by D/A, measuring how many derivations are needed for an actual edge addition on average. The computational redundancy of PEARL-WB and POCR are close for most benchmarks. On average, the redundancy values of POCR are 2.35 and 1.96 in value-flow analysis and alias analysis, respectively. The average redundancy values of PEARL-WB are 5.64 in value-flow analysis and 2.38 in alias analysis.

Discussion. Given a fully transitive relation A, the spanning tree model ensures that each node in the tree is reachable from the root node via only one path; PEARL-WB retains a global propagation graph for all nodes, where a node pair can be connected via multiple reachable A-paths. As a result, POCR eliminates more repetitive derivations than PEARL-WB. However, we find that reducing more derivations do not necessarily result in better performance since it can entail maintenance cost.

Overall Performance. The performance statistics of PEARL-WB for value-flow and alias analysis are respectively listed in Table I and Table II. PEARL-WB achieves a dramatic speedup of 7.17x over POCR for value-flow analysis (Table I). For alias analysis (Table II), PEARL-WB runs slightly faster (1.09x) than POCR.

Discussion. We notice that POCR takes a non-trivial amount of work to maintain spanning trees in our experiments, especially when fully transitive edges are dense. The significant speedup achieved by PEARL-WB over POCR for value-flow analysis, confirms the aforementioned statistics that fully transitive relations dominate in value-flow analysis(Figure 6). In addition, PEARL-WB saves a lot of memory compared to POCR. This is because our propagation graph representation is conceptually simple and cheap to update on the fly. For the *perlbench* benchmark in Table I, POCR solves within over 25 minutes with 63-GB memory consumption, while PEARL-WB takes only around 2 minutes with 10-GB consumed memory. This emphasizes that exhaustively diminishing computational redundancy by POCR does not necessarily result in improved overall performance, because it can entail additional costs to maintain spanning trees. In alias analysis, the performance of PEARL-WB and POCR are comparable because fully transitive edges only take a small proportion and the representation maintenance cost is negligible compared to overall solving time.

To sum up, propagation graph representation is effective (eliminating most repetitive derivations) and lightweight (cheap to maintain) for both two clients, achieving a promising overall performance.

E. RQ4. Effectiveness of Batch Propagation

PEARL adopts a muti-derivation manner via batch propagation to reduce propagation efforts. To quantify the benefit, We compare PEARL with PEARL-WB to show how many propagations are pruned by batch propagation and the offered speedups. We define the number of propagations during solving transitive production rules as PT. Thus, the reduction

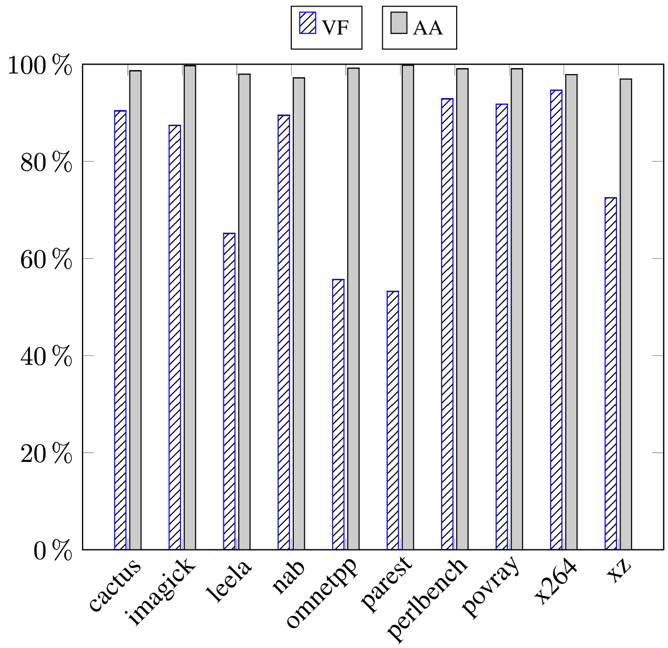


Fig. 9: Reduction rates in propagations of transitive relations

rate achieved by batch propagation can then be obtained via $(PT_{\text{PEARL-WB}} - PT_{\text{PEARL}})/PT_{\text{PEARL-WB}}$.

Result. Figure 9 shows the reduction rates achieved by PEARL, with average reductions of 79.31% and 98.53% for value-flow analysis and alias analysis, respectively. Consequently, PEARL runs 1.3x and 2.17x faster than PEARL-WB for value-flow analysis (Table I) and alias analysis (Table II), respectively.

Discussion. By performing batch propagation, PEARL eliminates a substantial number of propagations for transitive relations over PEARL-WB. As discussed in Section IV, we adopt eager propagation for fully transitive production (dominating in value-flow analysis), and iterative propagation for partially transitive production (dominating in alias analysis). Iterative propagation appears to accumulate more reachability relations, thereby pruning more propagations. As a result, PEARL achieves a larger speedup over PEARL-WB in alias analysis than value-flow analysis.

VI. Related Work

This work is relevant to improving the efficiency of CFL-reachability analysis. CFL-reachability framework was initially proposed in [13] and has been used to formulate many program analysis problems [4]. CFL-reachability was also studied in various contexts such as recursive state machine [35] and visibly pushdown languages [36]. A class of set constraints and CFL-reachability were also shown to be interconvertible [24]. Later, a practical work [21] described a specialized set constraint reduction for Dyck-CFL-reachability. It is commonly known that CFL-reachability-based algorithms have a cubic worst-case complexity. Previous work [14] showed that the Four Russians' Trick could yield a subcubic algorithm, which is orthogonal to our approach. So far, Significant progress has been made for specific clients, such as bidirected Dyck-reachability [8], [34], [37], IFDS-based analysis [38]–[43], pointer analysis [5]–[7], [10]–[12], [44], to just name a few. However, these algorithms are designed for predefined context-free grammars and typically do not work for other grammars, e.g., the IFDS framework [1] is not applicable to the alias analysis evaluated in our experiments.

A prevalent solution to avoid derivation redundancy is to construct summary edges for common paths [1], [3], [10]–[12], [21], [45], known as summarization. Sparse analysis [26], [27], [39], [45]–[49] adopts a similar idea by summarizing data dependencies to skip unnecessary paths. However, paths consisting of transitive edges have already been summarized as secondary edges by the standard algorithm [24], which exhibits a large amount of redundancy and poor scalability. Reducing the graph size by offline preprocessing techniques [16], [18]–[20] can also alleviate redundancy. Nevertheless, a large amount of redundancy can only be captured during the analysis. To diminish unnecessary computations, Graspan [22] utilizes a few data processing techniques from a novel "Big Data" perspective, and Datalog engine Soufflé [23] adopts the *semi-naive evaluation* strategy. However, these general frameworks do not utilize transitivity, and there is still a substantial amount of derivation redundancy [25]. POCR [25] accelerates CFL-reachability solving by reducing repetitive derivations. It also shows that removing transitive relations by grammar rewriting has limited effectiveness in reducing redundancy. Different from existing techniques, our multi-derivation approach effectively reduces derivation redundancy by propagating reachability relations in batch on sparse constraint graphs.

VII. Conclusion

This paper has proposed PEARL, a fast multi-derivation approach that efficiently solves transitivity for CFL-reachability by reducing derivation redundancy. Our experiments demonstrate that PEARL significantly accelerates CFL-reachability solving, achieving average speedups of 82.73x for value-flow analysis and 155.26x for alias analysis over the standard CFL-reachability algorithm. When compared with POCR, a state-of-the-art CFL-reachability solver, PEARL is 10.1x and 2.37x faster for value-flow and alias analysis, respectively.

Acknowledgment

We thank all anonymous reviewers for their valuable feedback. This work is supported by the National Key R&D Program of China (2022YFB3103900), the National Natural Science Foundation of China (NSFC) under grant number 62132020 and 62202452.

Data Availability Statement

We have provided an artifact to reproduce our experimental results in Section V. The artifact is publicly available at https://doi.org/10.6084/m9.figshare.23702271.

References

[1] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[3] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes*, 19(5):11–20, 1994.

[4] Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998.

[5] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. *ACM SIGPLAN Notices*, 40(10):59–76, 2005.

[6] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. *ACM SIGPLAN Notices*, 41(6):387–400, 2006.

[7] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 197–208, 2008.

[8] Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 435–446, 2013.

[9] Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*, volume 9, pages 98–122. Springer, 2009.

[10] Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 155–165, 2011.

[11] Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 264–274, 2012.

[12] Yu Su, Ding Ye, and Jingling Xue. Parallel pointer analysis with cfl-reachability. In *2014 43rd International Conference on Parallel Processing*, pages 451–460. IEEE, 2014.

[13] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 230–242, 1990.

[14] Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 159–169, 2008.

[15] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[16] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. *Acm Sigplan Notices*, 35(5):47–56, 2000.

[17] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, 2007.

[18] Yuanbo Li, Qirun Zhang, and Thomas Reps. Fast graph simplification for interleaved dyck-reachability. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 780–793, 2020.

[19] Yuanbo Li, Qirun Zhang, and Thomas Reps. Fast graph simplification for interleaved-dyck reachability. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(2):1–28, 2022.

[20] Yuxiang Lei, Yulei Sui, Shin Hwei Tan, and Qirun Zhang. Recursive state machine guided graph folding for context-free language reachability. *Proceedings of the ACM on Programming Languages*, 7(PLDI):318–342, 2023.

[21] John Kodumal and Alex Aiken. The set constraint/cfl reachability connection in practice. *ACM Sigplan Notices*, 39(6):207–218, 2004.

[22] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. *ACM SIGARCH Computer Architecture News*, 45(1):389–404, 2017.

[23] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, pages 422–430. Springer, 2016.

[24] David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1-2):29–98, 2000.

[25] Yuxiang Lei, Yulei Sui, Shuo Ding, and Qirun Zhang. Taming transitive redundancy for context-free language reachability. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1556–1582, 2022.

[26] Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 254–264, 2012.

[27] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.

[28] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[29] Dennis M Moyles and Gerald L Thompson. An algorithm for finding a minimum equivalent graph of a digraph. *Journal of the ACM (JACM)*, 16(3):455–460, 1969.

[30] Christian Fecht and Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. *Nord. J. Comput.*, 5(4):304–329, 1998.

[31] David J Pearce, Paul HJ Kelly, and Chris Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 3–12. IEEE, 2003.

[32] Manu Sridharan and Stephen J Fink. The complexity of andersen's analysis in practice. In *Static Analysis: 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings 16*, pages 205–221. Springer, 2009.

[33] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[34] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. Optimal dyck reachability for data-dependence and alias analysis. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

[35] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):786–818, 2005.

[36] Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211, 2004.

[37] Yuanbo Li, Kris Satya, and Qirun Zhang. Efficient algorithms for dynamic bidirected dyck-reachability. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022.

[38] Nomair A Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the ifds algorithm. In *Compiler Construction: 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*, pages 124–144. Springer, 2010.

[39] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. Performance-boosting sparsification of the ifds algorithm with applications to taint analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 267–279. IEEE, 2019.

[40] Haofeng Li, Haining Meng, Hengjie Zheng, Liqing Cao, Jie Lu, Lian Li, and Lin Gao. Scaling up the ifds algorithm with efficient disk-assisted computing. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 236–247. IEEE, 2021.

[41] Steven Arzt and Eric Bodden. Reviser: efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*, pages 288–298, 2014.

[42] Steven Arzt. Sustainable solving: Reducing the memory footprint of ifds-based data flow analyses using intelligent garbage collection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1098–1110. IEEE, 2021.

[43] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[44] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. Efficient subcubic alias analysis for c. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 829–845, 2014.

[45] Lian Li, Cristina Cifuentes, and Nathan Keynes. Precise and scalable context-sensitive pointer analysis via value flow graph. *ACM SIGPLAN Notices*, 48(11):85–96, 2013.

[46] Lian Li, Cristina Cifuentes, and Nathan Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 343–353, 2011.

[47] Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 460–473, 2016.

[48] Yulei Sui and Jingling Xue. Value-flow-based demand-driven pointer analysis for c and c++. *IEEE Transactions on Software Engineering*, 46(8):812–835, 2018.

[49] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 693–706, 2018.