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 ➔ v in the edge-labeled graph. We use the terms X-edge and X-reachability relation interchangeably, and they are both denoted as u ➔ 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 ➔ 4 (the wavy line) is derived by propagating reachability relation 2 ➔ 3 via A-edge 3 ➔ 4.
A := AA | a
X := XA | 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 sub-graph* 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 reachability 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-reachability

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 $\Sigma$ 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 $u \rightarrow X \rightarrow v$ denotes the edge from Node $u$ to Node $v$ labeled by $X$. Each path in $G$ defines a word over $\Sigma$ 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 $u \rightarrow \ldots \rightarrow v$ 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 $u \rightarrow X \rightarrow v$ into the edge-labeled graph.

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 ($u \rightarrow X \rightarrow v$) produced by productions $X ::= \varepsilon$

---

**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 $\text{StandardCFL}(CFG, G)$:
   2. Initialize ($CFG$, $G$);
   3. Procedure $\text{Init}()$:
      4. add $E$ to $W$;
      5. for each production $X ::= \varepsilon \in P$
         6. for each node $v \in V$
            7. if $v \rightarrow X \varepsilon$ then
               8. add $v \rightarrow X \varepsilon$ to $E$ and $W$
   6. Procedure $\text{Solve}(CFG, G)$:
      7. while $W \neq \emptyset$ do
         8. pop an edge $u \rightarrow Y \rightarrow v$ from $W$;
         9. for each production $X ::= Y \in P$
            10. if $u \rightarrow X \varepsilon$ then
                11. add $u \rightarrow X \varepsilon$ to $E$ and $W$
            12. for each outgoing edge $v \rightarrow Z \in \Delta$ from node $v$
                13. if $u \rightarrow X \varepsilon$ then
                    14. add $u \rightarrow X \varepsilon$ to $E$ and $W$
            15. for each production $X ::= ZY \in P$
                16. if $w \rightarrow X \varepsilon$ then
                    17. add $w \rightarrow X \varepsilon$ to $E$ and $W$
            18. for each production $X ::= ZY \in P$
                19. if $w \rightarrow X \varepsilon$ then
                    20. add $w \rightarrow X \varepsilon$ to $E$ and $W$
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 $S(X,v) = \{ v \mid v \xleftarrow{X} u \in E \}$, 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 $0 \xrightarrow{A} 2$, the spanning tree rooted from Node 2 is traversed to derive two reachability relations, $0 \xrightarrow{A} 3$ and $0 \xrightarrow{A} 4$.

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 $0 \xrightarrow{A} 2$ and $1 \xrightarrow{A} 2$ 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

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 $\text{SOLVE}$ 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 PROPTR and PROPFR, 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 PROPTR at Line 8 and PROPFR 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 $R_{\text{new}}(X,v)$ contains “old” $X$-reachability relations and “new” $X$-reachability relations of Node $v$, respectively. Besides, $P_{nt}$ 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 $R_{\text{new}}(X,v)$, and Node $v$ is added to $R_{\text{new}}(X,u)$ (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):
   for each edge \( u \xrightarrow{X} v \in \text{PSet} \) do
   PackRR(\( u, v \));
   PackRR(\( \overline{X}, v, u \));
   PropRRs();

6 Procedure PropRRs():
   while \( \text{NW} \neq \emptyset \) do
   pop node \( u \) from \( \text{NW} \);
   for each partially transitive relation \( X \) do
   \( R(X, u) \leftarrow R(X, u) \cup R_{\text{new}}(X, u) \);
   \( \Delta R(X, u) \leftarrow R_{\text{new}}(X, u) \);
   for partially transitive production \( X := \overline{X} \in P_{gt} \) do
   for \( u \xrightarrow{A} v \in \text{PG}(A) \) do
   DiffProp(\( X, \overline{R}(X, u) \), v);
   for each node \( v \in \Delta R(X, u) \) do
   PackRR(\( X, u, v \));

18 Procedure DiffProp(\( X, \text{srcSet}, v \)):
   \( R_{\text{new}}(X, v) \leftarrow R_{\text{new}}(X, v) \cup (\text{srcSet} \setminus R(X, v)) \);
   if \( R_{\text{new}}(X, v) \) changes then
   add \( v \) to \( \text{NW} \);

22 Procedure PackRR(\( X, u, v \)):
   \( R_{\text{new}}(X, v) \leftarrow R_{\text{new}}(X, v) \cup \{ u \} \);
   if \( \exists \text{ partially transitive production} \\
   X := \overline{X} \in P_{gt} \) then
   add \( v \) to \( \text{NW} \);

iteration (Line 8), then \( R_{\text{new}}(X, u) \) 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 \( X := \overline{X} A \) (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 \( \text{NW} \) (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.

\[
\overline{X} := \overline{XA}_1 \\quad X := \overline{XA}_2
\]

(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 \( 0 \xrightarrow{A_1} 3 \), which involves two partially transitive productions \( \overline{X} := \overline{XA}_1 \) and \( X := \overline{XA}_2 \), 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: \( 2 \xrightarrow{\overline{X}} 0 \Rightarrow 0 \xrightarrow{X} 2 \) (the wavy line).
3) Propagate reachability relation \( 0 \xrightarrow{X} 2 \) via edge \( 2 \xrightarrow{A_2} 3 \) to derive \( 0 \xrightarrow{X} 3 \).

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):
   for each edge $u \xrightarrow{A} v$ in FSet do
     UpdatePG($A, u, v$);
     if $u \xrightarrow{A} v$ is added into $PG(A)$ then
       for each $X := XA \in PG(A)$ do
         DiffProp$(X, R(X, u), v)$; // Lines 18-21 in Algo. 3

2. **Procedure** UpdatePG($A, u, v$):
   if $u \notin R(A, v)$ then
     add $u \xrightarrow{A} v$ to $PG(A)$;
     add $v \xrightarrow{A} u$ to $PG(A)$;
   srcSet $\leftarrow R(A, u) \cup \{u\}$;
   DFS($A, srcSet, v$);

3. **Procedure** DFS($A, srcSet, u$):
   $\Delta R(A, u) \leftarrow srcSet \setminus R(A, u)$;
   if $\Delta R(A, u) \neq \emptyset$ then
     $R(A, u) \leftarrow R(A, u) \cup \Delta R(A, u)$;
     for $u \xrightarrow{A} v$ in $PG(A)$ do
       DFS($A, \Delta R(A, u), v$);
     for $v \in \Delta R(A, u)$ do
       $R(A, v) \leftarrow R(A, v) \cup \{u\}$;

![Fig. 3: Iterative propagation for production $A := AA$](image)

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

- **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 $0 \xrightarrow{A} 2$ 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 $PG(A)$ for relation $A$ only when relation $A$ 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 $X := X \bar{A}$). Here we only prove the correctness of the case $X := AX$, as the proof for other productions is similar.

Edge $v_0 \xrightarrow{A} v_1$ is derived based on $X := AX$ by Algorithm 1 only if it is formed by an $X$-edge $v_0 \xrightarrow{X} v_1$ and an $A$-edge $v_1 \xrightarrow{A} v_2$. We prove the correctness by two properties:

- **Soundness.** $v_1 \xrightarrow{X} v_2$ is either a primary edge or secondary edge. Either way, $v_2$ 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 $PG(A)$, $v_0 \xrightarrow{X} v_2$ 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 $X := AX$. \[\square\]

### V. Evaluation

We evaluate the performance of PEARL on two practical static analysis clients: context-sensitive value-flow anal-
\[ A ::= A A | \text{call}_i A \ret_i | a | \epsilon \]

(a) Context-free grammar

\[ A ::= A A | CA_i \ret_i | a | \epsilon \]

CA_i ::= \text{call}_i A

(b) Normalized grammar

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

\[ M ::= \overline{A} V d \\
V ::= \overline{A} V A | f_i V f_i | M | \epsilon \\
A ::= A A | \epsilon M? | \epsilon \\
A ::= A A | a M | a | \epsilon \\
A ::= A A | M? \pi | \epsilon \\
(A) Context-free grammar

(b) Normalized grammar

Fig. 5: CFG for field-sensitive alias analysis

\[ VF ::= \overline{A} V A | f_i V f_i | M | \epsilon \]

\[ DV ::= \overline{A} V | V A | FV_i f_i | M | \epsilon \]

\[ M ::= DV d \\
FV_i ::= f_i V \\
\]

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

yasis [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 an 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 \text{call}_i and \text{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 \(v \xleftarrow{X} u\) in PEG. The normalized grammar is shown in Figure 5b.

**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 added edges 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).

<table>
<thead>
<tr>
<th>id</th>
<th>Before Prep.</th>
<th>After Prep.</th>
<th>STD</th>
<th>POCR</th>
<th>PEARL</th>
<th>PEARL-WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Nodes #Edges</td>
<td>#Nodes #Edges</td>
<td>Time</td>
<td>Mem</td>
<td>SPU</td>
<td>Mem</td>
</tr>
<tr>
<td>cactus</td>
<td>544480 1007989</td>
<td>232046 616599</td>
<td>3408.36</td>
<td>3.46</td>
<td>604.10</td>
<td>5.6x</td>
</tr>
<tr>
<td>imagick</td>
<td>574089 842509</td>
<td>165096 319141</td>
<td>583.71</td>
<td>0.43</td>
<td>591.3</td>
<td>9.9x</td>
</tr>
<tr>
<td>leela</td>
<td>64466 89081</td>
<td>21711 40409</td>
<td>1.58</td>
<td>0.02</td>
<td>0.47</td>
<td>3.4x</td>
</tr>
<tr>
<td>nab</td>
<td>55652 72366</td>
<td>15415 23736</td>
<td>55.51</td>
<td>0.50</td>
<td>16.59</td>
<td>3.3x</td>
</tr>
<tr>
<td>omnetpp</td>
<td>664358 1857831</td>
<td>237854 1277123</td>
<td>229.26</td>
<td>1.08</td>
<td>15.49</td>
<td>14.8x</td>
</tr>
<tr>
<td>parest</td>
<td>299718 407343</td>
<td>114099 199793</td>
<td>2.40</td>
<td>0.07</td>
<td>0.67</td>
<td>3.6x</td>
</tr>
<tr>
<td>perlbench</td>
<td>697744 1662445</td>
<td>321778 1122795</td>
<td>16366.80</td>
<td>6.35</td>
<td>1520.19</td>
<td>10.8x</td>
</tr>
<tr>
<td>povray</td>
<td>537775 1014167</td>
<td>213130 621400</td>
<td>5834.13</td>
<td>5.05</td>
<td>655.14</td>
<td>8.9x</td>
</tr>
<tr>
<td>x264</td>
<td>207064 340217</td>
<td>66417 162595</td>
<td>194.16</td>
<td>0.70</td>
<td>34.77</td>
<td>5.6x</td>
</tr>
<tr>
<td>xz</td>
<td>49395 62955</td>
<td>15072 23002</td>
<td>0.54</td>
<td>0.01</td>
<td>0.16</td>
<td>3.4x</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>id</th>
<th>Before Prep.</th>
<th>After Prep.</th>
<th>STD</th>
<th>POCR</th>
<th>PEARL</th>
<th>PEARL-WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Nodes #Edges</td>
<td>#Nodes #Edges</td>
<td>Time</td>
<td>Mem</td>
<td>SPU</td>
<td>Mem</td>
</tr>
<tr>
<td>cactus</td>
<td>93537 212478</td>
<td>65232 153470</td>
<td>- -</td>
<td>- -</td>
<td>191.27</td>
<td>11.62</td>
</tr>
<tr>
<td>imagick</td>
<td>119314 301846</td>
<td>73499 196570</td>
<td>- -</td>
<td>- -</td>
<td>554.13</td>
<td>42.55</td>
</tr>
<tr>
<td>leela</td>
<td>22186 49748</td>
<td>14371 33326</td>
<td>312.28</td>
<td>0.31</td>
<td>3.40</td>
<td>91.8x</td>
</tr>
<tr>
<td>nab</td>
<td>16261 34676</td>
<td>8794 19218</td>
<td>7.12</td>
<td>0.10</td>
<td>0.76</td>
<td>9.4x</td>
</tr>
<tr>
<td>omnetpp</td>
<td>241916 509166</td>
<td>146049 311980</td>
<td>- -</td>
<td>- -</td>
<td>410.79</td>
<td>17.96</td>
</tr>
<tr>
<td>parest</td>
<td>117500 251436</td>
<td>67949 148286</td>
<td>- -</td>
<td>- -</td>
<td>92.77</td>
<td>4.79</td>
</tr>
<tr>
<td>perlbench</td>
<td>139183 348916</td>
<td>72231 192994</td>
<td>- -</td>
<td>- -</td>
<td>1733.42</td>
<td>110.84</td>
</tr>
<tr>
<td>povray</td>
<td>76405 147258</td>
<td>45622 110732</td>
<td>14699.10</td>
<td>3.24</td>
<td>160.97</td>
<td>91.3x</td>
</tr>
<tr>
<td>x264</td>
<td>60956 136352</td>
<td>40625 94110</td>
<td>1056.20</td>
<td>1.31</td>
<td>11.13</td>
<td>94.9x</td>
</tr>
<tr>
<td>xz</td>
<td>12425 26468</td>
<td>7130 15228</td>
<td>6.67</td>
<td>0.05</td>
<td>0.42</td>
<td>15.9x</td>
</tr>
</tbody>
</table>

Fig. 7: Computational redundancy of value-flow 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. Compared 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 multi-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
relations, thereby pruning more propagations. As a result, Iterative propagation appears to accumulate more reachability partially transitive production (dominating in alias analysis), and iterative propagation 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


