Taming Transitive Redundancy for Context-Free Language Reachability

YUXIANG LEI, University of Technology Sydney, Australia
YULEI SUI, University of Technology Sydney, Australia
SHUO DING, Georgia Institute of Technology, USA
QIRUN ZHANG, Georgia Institute of Technology, USA

Given an edge-labeled graph, context-free language reachability (CFL-reachability) computes reachable node pairs by deriving new edges and adding them to the graph. The redundancy that limits the scalability of CFL-reachability manifests as redundant derivations, i.e., identical edges can be derived multiple times due to the many paths between two reachable nodes. We observe that most redundancy arises from the derivations involving transitive relations of reachable node pairs. Unfortunately, existing techniques for reducing redundancy in transitive-closure-based problems are either ineffective or inapplicable to identifying and eliminating redundant derivations during on-the-fly CFL-reachability solving.

This paper proposes a scalable yet precision-preserving approach to all-pairs CFL-reachability analysis by taming its transitive redundancy. Our key insight is that transitive relations are intrinsically ordered, and utilizing the order for edge derivation can avoid most redundancy. To address the challenges in determining the derivation order from the dynamically changed graph during CFL-reachability solving, we introduce a hybrid graph representation by combining spanning trees and adjacency lists, together with a dynamic construction algorithm. Based on this representation, we propose a fast and effective partially ordered algorithm Pocr to boost the performance of CFL-reachability analysis by reducing its transitive redundancy during on-the-fly solving. Our experiments on context-sensitive value-flow analysis and field-sensitive alias analysis for C/C++ demonstrate the promising performance of Pocr. On average, Pocr eliminates 98.50% and 97.26% redundant derivations respectively for the value-flow and alias analysis, achieving speedups of 21.48× and 19.57× over the standard CFL-reachability algorithm. We also compare Pocr with two recent open-source tools, Graspan (a CFL-reachability solver) and Soufflé (a Datalog engine). The results demonstrate that Pocr is over 3.67× faster than Graspan and Soufflé on average for both value-flow analysis and alias analysis.

CCS Concepts: • **Theory of computation → Grammars and context-free languages**.

Additional Key Words and Phrases: CFL-reachability, transitive relation, redundancy, performance

ACM Reference Format:
Yuxiang Lei, Yulei Sui, Shuo Ding, and Qirun Zhang. 2022. Taming Transitive Redundancy for Context-Free Language Reachability. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 180 (October 2022), 27 pages. https://doi.org/10.1145/3563343

1 INTRODUCTION

Context-free language (CFL) reachability is a fundamental framework for many static analyses such as shape analysis [Reps 1995], polymorphic flow analysis [Rehof and Fähndrich 2001], data flow analysis [Reps et al. 1995], typestate analysis [Naeem and Lhoták 2008], point-to analysis [Zheng

Authors' addresses: Yuxiang Lei, Yuxiang.Lei@student.uts.edu.au, University of Technology Sydney, Sydney, Australia; Yulei Sui, Yulei.Sui@uts.edu.au, University of Technology Sydney, Sydney, Australia; Shuo Ding, sding@gatech.edu, Georgia Institute of Technology, Atlanta, USA; Qirun Zhang, qrzhang@gatech.edu, Georgia Institute of Technology, Atlanta, USA.

2475-1421/2022/10-ART180
https://doi.org/10.1145/3563343

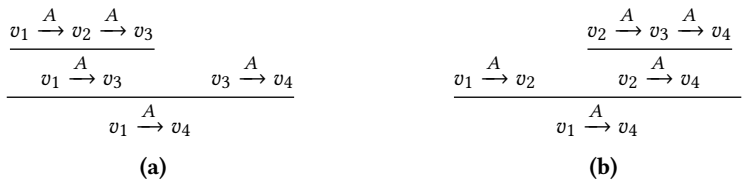


Fig. 1. Transitive redundancy caused by two ways to derive $v_1 \xrightarrow{A} v_4$ from $v_1 \xrightarrow{A} v_2 \xrightarrow{A} v_3 \xrightarrow{A} v_4$.

and Rugina 2008]. It extends standard graph reachability from unlabeled graphs to edge-labeled graphs, and the process of discovering reachability relations is to *derive* and *add* edges to the graphs. Specifically, CFL-reachability represents the (binary) reachability relation A introduced by a path from node u to v by deriving an edge connecting nodes u and v, i.e., $(u, v) \in A$ iff v is A-reachable from u. If the derived edge is not in the graph, CFL-reachability algorithms add it to the graph, making the derived reachability relation explicit.

Transitive relations are ubiquitous in many CFL-reachability-based analyses. Among the aforementioned static analyses, control/data/value flows are usually formalized as transitive relations. Moreover, the widely applicable Dyck-relations [Chatterjee et al. 2018; Zhang et al. 2013] are also transitive. For example, given a transitive relation A and three nodes v_1, v_2 and v_3, if $(v_1, v_2) \in A$ and $(v_2, v_3) \in A$, then $(v_1, v_3) \in A$. For any path comprised of two transitive edges $v_1 \xrightarrow{A} v_2$ and $v_2 \xrightarrow{A} v_3$, CFL-reachability analysis derives a new transitive edge $v_1 \xrightarrow{A} v_3$, which is then added to the graph, indicating a reachability relation or path from v_1 to v_3. New paths are incrementally derived based on the established transitive edges on the dynamic graph until a fixed point is reached.

Transitive Redundancy. CFL-reachability problems are typically solved by a standard worklist algorithm [Melski and Reps 2000] which iteratively derives new edges from every path consisting of transitive edges between two nodes. Transitive relations significantly increase the algorithm's redundancy during CFL-reachability solving because multiple derivations can result in the same edge from node u to v. As a result, only one derivation is needed to make this reachability relation explicit (by adding one edge to the graph), while all other derivations are redundant. Consider a path $v_1 \xrightarrow{A} v_2 \xrightarrow{A} v_3 \xrightarrow{A} v_4$, there are two ways to derive $v_1 \xrightarrow{A} v_4$, as shown in Figure 1. $v_1 \xrightarrow{A} v_4$ only needs to be derived and added to the graph once to make explicit the reachability relation from v_1 to v_4. However, the standard algorithm [Melski and Reps 2000] can derive $v_1 \xrightarrow{A} v_4$ twice, causing one redundant derivation. Such redundancy is further amplified in the presence of paths consisting of more transitive edges derived during on-the-fly CFL-reachability solving. Unfortunately, redundancy can not be eliminated by simply merging nodes connected by a transitive edge because the merging can cause incorrect/imprecise results, especially in a dynamically changing graph. For example, given two established edges $v_i \xrightarrow{A} v_j$, $v_i \xrightarrow{A} v_k$ and a newly derived edge $v_l \xrightarrow{A} v_j$, merging v_i and v_j causes v_l to reach v_k, which is incorrect.

One prevalent approach to reduce transitive redundancy is to reduce the size of the input graph in the preprocessing stage [Fähndrich et al. 1998; Li et al. 2020]. The most popular technique is cycle elimination [Fähndrich et al. 1998; Hardekopf and Lin 2007; Nuutila and Soisalon-Soininen 1994; Pereira and Berlin 2009; Tarjan 1972], which merges cycles consisting of only transitive edges. However, there are many more cases under which the redundant derivations are caused by paths consisting of transitive edges that do not form cycles. These cases cannot be handled by cycle elimination. Besides, applying the technique during CFL-reachability solving is also impractical since it needs to repeatedly find and collapse cycles once the graph is changed, thus impacting the algorithm's performance.

Moreover, albeit the offline preprocessing techniques can alleviate redundancy by reducing the size of the input graph, there are still a large number of redundant derivations that can only be captured and eliminated during the on-the-fly CFL-reachability solving procedure [Bravenboer and Smaragdakis 2009; Jordan et al. 2016b; Wang et al. 2017]. Our empirical study (Section 6) also shows that the standard algorithm still exhibits numerous redundant derivations even after pre-processing the input graph by offline techniques like cycle elimination. To reduce redundant derivations during online solving, a recent technique [Wang et al. 2017] partitions the adjacency list of each node into two parts, marked as "old" and "new", to avoid computing the established edges repeatedly. Although it avoids adding duplicate edges, this approach does not utilize the property of transitive relations to establish an effective derivation order. Therefore, it cannot eliminate redundant derivations that occupy a substantial amount of analysis time.

Our approach differs from the existing ones by dynamically inferring a derivation order to resolve edges involving transitive relations. We observe that transitive edges can be derived from a path in various orders, e.g., from the head to the tail like Figure 1(a), from the tail to the head like Figure 1(b) or from the middle to both ends when the path is longer. Our insight is that maintaining a consistent derivation order can effectively eliminate redundant derivations. For example, maintaining an order that computes edges from the head to the tail avoids the redundant derivation of Figure 1(b), while maintaining an order from the tail to the head avoids the redundant derivation of Figure 1(a).

The challenge is to find the best possible derivation order when computing transitive closures in CFL-reachability analysis. Obviously, getting the best derivation order to avoid redundancy on a simple path (e.g., Figure 1) is trivial. However, a node can reside in multiple paths and/or in cycles on a dynamically updated graph. Identifying the derivation order in such dynamic graphs is non-trivial. Intuitively, retrieving the topological order by traversing the graph can help determine the order and reduce redundancy. However, to maintain the precision and correctness in the presence of resolving dynamic transitive closure, the analysis requires repeatedly computing the topological order, which significantly increases the overheads and defeats the purpose of improving the scalability of CFL-reachability.

To address this challenge, we introduce a hybrid graph representation combining spanning trees and adjacency lists, together with a fast yet effective dynamic construction algorithm, to on-the-fly infer the derivation order during CFL-reachability solving. The acyclic property of spanning trees makes the traversals for determining derivation orders efficient. Based on this representation, we propose a partially ordered CFL-reachability algorithm Pocr, which quickly solves all-pairs reachability analysis by reducing the transitive redundancy. Compared to the standard algorithm, Pocr computes the same solution and is much more efficient.

We have evaluated Pocr using two popular static analyses for C/C++ as our clients: context-sensitive value-flow analysis [Sui et al. 2014] and field-sensitive alias analysis [Zheng and Rugina 2008]. The empirical results show that: (1) Pocr eliminates almost all redundant derivations. On average, the reduction rates of value-flow analysis and alias analysis are 98.50% and 97.26%, respectively, and (2) By eliminating redundant derivations, we can significantly improve the performance of the two CFL-reachability-based clients. On average, Pocr achieves speedups of 21.48× and 19.57× over the standard algorithm [Melski and Reps 2000], respectively, for value-flow analysis and alias analysis. We also compare Pocr with two recent open-source tools Graspan [Wang et al. 2017] (a CFL-reachability solver) and Soufflé [Jordan et al. 2016b] (a Datalog engine). The results show that Pocr is over 3.67× faster than Graspan and Soufflé for both value-flow analysis and alias analysis.

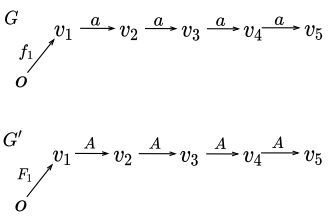
The contributions of this paper are as follows:

$$A ::= A\ A \mid a$$
$$F_i ::= F_i\ A \mid f_i$$

(a) A context-free grammar.

```
struct T{int f1; int f2;}
struct T o; ...
int v1, v2, v3, v4, v5;
v1 = o.f1;
v5 = v4 = v3 = v2 = v1;
```

(b) C code fragment.



(c) G is the graph abstracted from (a). G' is transformed from G via $A ::= a$ and $F_i ::= f_i$.

$$o \xrightarrow{F_1} v_5 \Leftarrow \begin{cases} o \xrightarrow{F_1} v_1 \xrightarrow{A} v_5 \\ o \xrightarrow{F_1} v_2 \xrightarrow{A} v_5 \\ o \xrightarrow{F_1} v_3 \xrightarrow{A} v_5 \\ o \xrightarrow{F_1} v_4 \xrightarrow{A} v_5 \ \text{/* Pocr */} \end{cases}$$

(d) $o \xrightarrow{F_1} v_5$ can be derived in four ways by the standard algorithm whereas our approach reduces the four ways to one (in this example, it is the last one).

Fig. 2. Motivating example.

- We offer a new perspective, namely reducing transitive redundancy via ordered derivations, and introduce a hybrid graph representation to efficiently infer the derivation order during on-the-fly all-pairs CFL-Reachability analysis.
- We present Pocr, a partially ordered CFL-reachability algorithm, which significantly accelerates CFL-reachability solving where transitive redundancy dominates.
- We apply our technique to a context-sensitive value-flow analysis [Zheng and Rugina 2008] and a field-sensitive alias analysis for C/C++ [Sui et al. 2014]. The experiment results show that our technique eliminates almost all redundant derivations in the two clients and significantly improves the performance.

The remainder of this paper is structured as follows. Section 2 presents a motivating example. Section 3 introduces the background and formulates the research problem. Section 4 elaborates on our approach Pocr. Section 5 discusses the effectiveness of Pocr. Sections 6 details the experiments, followed by related work and the conclusion in Sections 7 and 8.

2 MOTIVATING EXAMPLE

This section gives an example to illustrate the transitive redundancy of CFL-reachability, motivate our approach and show the benefits and the challenges.

The context-free grammar (CFG) in Figure 2(a) is widely applied in field-sensitive analysis [Zheng and Rugina 2008]. In the grammar, A denotes a value flow, F_i denotes the propagation of the value of the i-th field, a denotes an assignment, and f_i denotes the address of the i-the field. The two production rules mean that an A-edge can be generated from an a-edge or two connected A-edges in an edge labeled graph; and an F_i-edge can be generated from an f_i-edge or a path comprised of an F_i-edge and an A-edge. Obviously, A is a transitive relation, and F_i is partially transitive because it can be transited via A relations. In Figure 2(c), graph G is abstracted from the code fragment of Figure 2(b) and G' is transformed from G by applying $A ::= a$ and $F_i ::= f_i$.

Transitive redundancy in CFL-reachability. The standard CFL-reachability algorithm [Melski and Reps 2000] derives new edges from paths consisting of at most two transitive edges. Redundant derivations due to transitive relations can be triggered in various ways. Our motivating example illustrates one case. According to the production rules $A ::= A\ A$ and $F_i ::= F_i\ A$ in Figure 2(a), there will be $v_k \xrightarrow{A} v_5$ for all $k \in \{1, \ldots, 4\}$ and $o \xrightarrow{F_1} v_j$ for all $j \in \{1, \ldots, 5\}$ in the graph after solving reachability. Here we study how $o \xrightarrow{F_1} v_5$ is derived. Due to the rule $F_i ::= F_i\ A$, $o \xrightarrow{F_1} v_5$ can be

derived from $o \xrightarrow{F_1} v_j$ and $v_k \xrightarrow{A} v_5$ whenever $j = k$. There are four paths (consisting of two edges) in total that can generate $o \xrightarrow{F_1} v_5$, as shown in Figure 2(d). The traditional algorithm computes edges in an arbitrary order. To ensure a correct reachability solution, it derives edges from each of the four paths at least once. Thus, there are at least three redundant derivations of $o \xrightarrow{F_1} v_5$.

Our approach and its benefits. Our approach determines the computation order of the F_1-edges based on the order of the nodes on the path $v_1 \xrightarrow{A} v_2 \xrightarrow{A} v_3 \xrightarrow{A} v_4 \xrightarrow{A} v_5$ in G' of Figure 2(c). Specifically, we only derive F_1-edges $o \xrightarrow{F_1} v_{j+1}$ from $o \xrightarrow{F_1} v_j \xrightarrow{A} v_{j+1}$, where $j \in \{1, \ldots, 4\}$, and always derive $o \xrightarrow{F_1} v_{j+1}$ immediately after adding $o \xrightarrow{F_1} v_j$ to the graph. This is a "head-to-tail" derivation order, and it only derives $o \xrightarrow{F_1} v_5$ by the last line in Figure 2(d), avoiding the first three ways causing redundant derivations. Similarly, the redundant derivations of $o \xrightarrow{F_1} v_2$, $o \xrightarrow{F_1} v_3$ and $o \xrightarrow{F_1} v_4$ are eliminated by our approach. Moreover, if $o \xrightarrow{F_1} v_1$ is added to the graph again based on they other ways described in Figure2(d), our approach can avoid the repeated derivation of $o \xrightarrow{F_1} v_k$ where $k \in \{2, \ldots, 5\}$ because we know that such edges have already been added to the graph along with the first time adding $o \xrightarrow{F_1} v_1$. Compared to existing techniques, cycle elimination has no effect in this example because there is no cycle in the graph. The existing edge duplication reduction technique (Section 4.2 in [Wang et al. 2017]) which does not exploit ordered derivations still introduces redundancy that $o \xrightarrow{F_1} v_5$ can be derived from $o \xrightarrow{F_1} v_1 \xrightarrow{A} v_5$ and $o \xrightarrow{F_1} v_2 \xrightarrow{A} v_5$. Our experiment also confirms that the technique still suffers from a lot of redundant derivations during CFL-reachability solving.

Challenges. In this small example, we can easily obtain the computation order along the path $v_1 \xrightarrow{A} v_2 \xrightarrow{A} v_3 \xrightarrow{A} v_4 \xrightarrow{A} v_5$. However, the graphs of real-world CFL-reachability problems are large and complex, where multiple paths and/or cycles may share vertices. Moreover, adding edges when solving CFL-reachability may also change the transitive closure of the graph. Hence, the best derivation order is also changed dynamically. These require an effective representation to maintain and infer the computation order efficiently, and the representation can be updated accordingly with the dynamic graph during CFL-reachability solving.

3 PRELIMINARIES

This section introduces the preliminaries and formulates the research problem of this paper.

3.1 CFL-Reachability

A CFL-reachability instance $Reach\langle CFG, G\rangle$ is comprised of a context-free grammar $CFG = \langle \Sigma, N, P, S\rangle$ and an edge-labeled graph $G = \langle V, E\rangle$. In CFG, $\Sigma = N \cup T$ is an *alphabet* containing two kinds of symbols: N a set of *non-terminals* and T a set of *terminals*; P is a set of *production rules*, and each rule describes an inference from terminals/non-terminals on the right side to the non-terminal on the left side; and $S \in N$ is called the start symbol. In graph G, each edge $v_i \xrightarrow{X} v_j \in E$ from node v_i to v_j is labeled by a symbol $X \in \Sigma$. Initially, G contains only edges labeled by terminals.

In CFL-reachability, a path $v_0 \xrightarrow{Y_1} v_1 \xrightarrow{Y_2} \ldots \xrightarrow{Y_n} v_n$ *implies* an X-relation from v_0 to v_n, if there is a non-terminal $X \in N$ that can be inferred from the string $Y_1 Y_2 \cdots Y_n \in \Sigma^n$ formed by the sequence of edge labels of the path via one or more production rules. For any two nodes $v_i, v_j \in V$, if there is

Algorithm 1: Standard CFL-reachability algorithm.

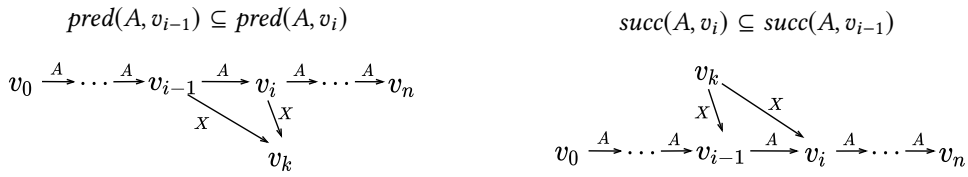
```
1  Function Reach(CFG, G)
2      init();                                                    /* Lines 11–15 */
3      while W ≠ ∅ do
4          select and remove an edge v_i -Y-> v_j from W;
5          for each production X ::= Y ∈ P do
6              if v_i -X-> v_j ∉ E then add v_i -X-> v_j to E and to W;
7          for each production X ::= Y Z ∈ P do
8              CheckSucc(X, Z, v_i, v_j);                         /* Lines 19–21 */
9          for each production X ::= Z Y ∈ P do
10             CheckPred(X, Z, v_i, v_j);                         /* Lines 16–18 */
11 Procedure init()
12     add all edges of E to W;
13     for each production X ::= ε ∈ P do
14         for each node v_i ∈ V do
15             if v_i -X-> v_i ∉ E then add v_i -X-> v_i to E and to W;
16 Procedure CheckPred(X, Z, v_i, v_j);
17     for each edge v_k -Z-> v_i ∈ G do
18         if v_k -X-> v_j ∉ E then add v_k -X-> v_j to E and to W;
19 Procedure CheckSucc(X, Z, v_i, v_j)
20     for each edge v_j -Z-> v_k ∈ G do
21         if v_i -X-> v_k ∉ E then add v_i -X-> v_k to E and to W;
```

a path from v_i to v_j implying an X-relation, v_j is said to be X-reachable from v_i. Generally speaking, CFL-reachability discovers reachability relations between given sources and sinks in the graph.

Solving CFL-reachability involves deriving new edges from existing paths and adding the edges to the graph to make explicit the reachability relations:

- *Edge Derivation*: An X-edge $v_i \xrightarrow{X} v_j$ can be derived from a path $v_i \xrightarrow{Y_1} \cdots \xrightarrow{Y_k} v_j \in G$ if there is a production rule $X ::= Y_1 \cdots Y_k \in P$.
- *Edge Addition*: A newly derived X-edge $v_i \xrightarrow{X} v_j$ should be added to the graph to make explicit the X-relation from v_i to v_j if it is not already in the graph.

There is a standard dynamic-programming algorithm [Melski and Reps 2000] for solving all-pairs CFL-reachability. It is detailed in Algorithm 1 for the convenience of our further discussions. The algorithm computes new edges from paths consisting of at most two edges. Correspondingly, it requires the input CFG to be normalized so that the right-hand side of each production has at most two symbols. It maintains a worklist W holding new edges. Once an edge is added to the graph, it is also added to the worklist (Lines 6, 15, 18, 21). Lines 5–6 derive edges from paths containing only one edge. The two procedures `CheckPred` and `CheckSucc` derive edges from paths consisting of two edges. The process of solving CFL-reachability is to iteratively process the edges in the worklist until a fixed point is reached: no new edge can be added to the graph, i.e., all reachability relations are explicit.



(a) Processing $v_{i-1} \xrightarrow{X} v_k$ and $v_i \xrightarrow{X} v_k$ based on $X ::= A\ X$ leads to $|pred(A, v_{i-1})|$ repeated computations.

(b) Processing $v_k \xrightarrow{X} v_{i-1}$ and $v_k \xrightarrow{X} v_i$ based on $X ::= X\ A$ leads to $|succ(A, v_i)|$ repeated computations.

Fig. 3. Redundant checks in partial transitive derivations, where A is a transitive relation.

3.2 Redundant Derivations and Transitive Redundancy

Adjacency-list graph representation. A standard form of graph representation is to store edges in the adjacency lists of nodes. Specifically, with respect to a label X and a node v_i, the X-predecessors and the X-successors of v_i are stored in two sets $pred(X, v_i) \subseteq V$ and $succ(X, v_i) \subseteq V$ such that

$$\begin{cases} pred(X, v_i) = \{v_i' \mid v_i' \xrightarrow{X} v_i \in E\} \\ succ(X, v_i) = \{v_i' \mid v_i \xrightarrow{X} v_i' \in E\}. \end{cases}$$

The adjacency lists are growing when solving CFL-reachability, e.g., adding an edge $v_i \xrightarrow{X} v_j$ to E means to add v_i to $pred(X, v_j)$ and v_j to $succ(X, v_i)$.

Redundant derivation. In Algorithm 1, redundant derivations mainly occur in `CheckPred` and `CheckSucc`, where edge derivations are implied in the traversal of predecessors/successors of nodes. For example, `CheckPred`(X, Z, v_i, v_j) naturally derives $|pred(Z, v_i)|$ X-edges and each derived edge is checked to determine whether it is already in the graph before addition. Thus, the shared predecessors or successors of nodes is a major contributing factor in redundant derivations.

Example 3.1. While processing an edge $v_i \xrightarrow{Y} v_j$, for any production rule $X ::= Z\ Y \in P$, `CheckPred`(X, Z, v_i, v_j) traverses $pred(Z, v_i)$ to determine whether the derived $v_k \xrightarrow{X} v_j$ is in the graph for all $v_k \in pred(Z, v_i)$. Similarly, while processing another edge $v_i' \xrightarrow{Y} v_j$, `CheckPred`(X, Z, v_i', v_j) traverses $pred(Z, v_i')$ to determine whether the derived $v_k' \xrightarrow{X} v_j$ is in the graph for all $v_k' \in pred(Z, v_i')$. The nodes in $pred(Z, v_i) \cap pred(Z, v_i')$ are repeatedly visited by `CheckPred`(X, Z, v_i, v_j) and `CheckPred`(X, Z, v_i', v_j). Namely, there are at least $|pred(Z, v_i) \cap pred(Z, v_i')|$ repeated edge derivations.

Transitive redundancy. In CFL-reachability, the transitivity of a relation A can either manifest in a doubly recursive rule (Definition 3.1) $A ::= A\ A$ or be implied in other production rules such as $A ::= A^*$, $A ::= A^+$, etc. Any edge $v_i \xrightarrow{A} v_j \in G$ denoting a transitive relation implies a series of reachability relations $v_k \xrightarrow{A} v_k'$ for all $(v_k, v_k') \in pred(A, v_i) \times succ(A, v_j)$.

Besides transitive relations, CFL-reachability also handles partial transitive relations whose edges are derived via singly recursive rules (Definition 3.2). For example, the production rule $F_i ::= F_i\ A$ in our motivating example is left-recursive. Different from transitive relations, partial transitive relations do not benefit from cycle elimination since cycles consisting of partial transitive edges cannot be merged.

Definition 3.1 (*Doubly Recursive Rules*)**.** A doubly recursive rule, also called a left-right-recursive rule, is in form of $A ::= A\ A$.

Definition 3.2 (*Singly Recursive Rules*)**.** The *left-recursive* rules, in the form of $X ::= X\ A$, and the *right-recursive* rules, in the form of $X ::= A\ X$, are collectively called singly recursive rules.

In CFL-reachability, both transitive and partial transitive relations suffer from transitive redundancy. The property of transitive relations implies that for any path $v_0 \xrightarrow{A} v_1 \xrightarrow{A} \cdots \xrightarrow{A} v_n$ where A is transitive, there are $pred(A, v_{i-1}) \subseteq pred(A, v_i)$ and $succ(A, v_i) \subseteq succ(A, v_{i-1})$ for all $i \in \{1, \ldots, n\}$. Therefore, for any $X ::= A\ X \in P$, processing $v_i \xrightarrow{X} v_k$ and $v_j \xrightarrow{X} v_k$ where $i, j \in \{0, \ldots, n\}$ and $i < j$ leads to $|pred(A, v_i)|$ repeated derivation. Similarly, for any $X ::= X\ A \in P$, processing $v_k \xrightarrow{X} v_i$ and $v_k \xrightarrow{X} v_j$ where $i, j \in \{0, \ldots, n\}$ and $i < j$ leads to $|succ(A, v_j)|$ repeated derivation, as illustrated in Figure 3. Such property greatly increase redundancy when the path to be derived is long.

3.3 Problem Formulation

Given the ubiquity of transitive relations in static analyses, this paper aims to improve the scalability of CFL-reachability by reducing the redundant derivations caused by transitive relations.

Our technique benefits the CFL-reachability problems containing transitive relations, which manifest in doubly recursive rules, and partial transitive relations, which manifest in left- or right-recursive rules. Similar to the standard CFL-reachability algorithm, our technique also works on normalized context-free grammars. Specifically, it requires the transitive relations to be explicit in the form of doubly recursive rules, i.e., $A ::= A\ A$.

We formulate the research problem as follows:

> Given a CFL-reachability instance containing transitive relations, eliminate the redundant derivations caused by singly recursive or doubly recursive rules.

4 POCR: PARTIALLY ORDERED CFL-REACHABILITY FOR ELIMINATING TRANSITIVE REDUNDANCY

Our motivating example in Section 2 shows that ordering computations based on the property of transitive relations can effectively reduce redundant derivations. The challenge lies in constructing a proper representation of the transitive relations on top of the dynamically changed graph and correctly updating the representation when solving CFL-reachability. This section details our solution. Section 4.1 introduces a hybrid graph representation to reduce redundant derivations. Section 4.2 provides a dynamic construction algorithm to efficiently update the spanning-tree model in our hybrid graph representation. Section 4.3 proposes the overall solution: a partially ordered CFL-reachability algorithm Pocr for all-pairs CFL-reachability analysis.

4.1 Hybrid Graph Representation for Reducing Redundant Derivations

For any node $v_i \in V$, we can always construct a spanning tree rooted at v_i to represent its predecessors/successors associated with a transitive relation A [Italiano 1986]. Before introducing our spanning-tree model, we first study the following property: In CFL-reachability, a transitive A-edge can be created not only by using a doubly recursive rule $A ::= A\ A$ but also by other rules such as $A ::= a$ or $A ::= B\ C$, etc. We classify the A-edges created via different production rules into two categories: *primary edges* (Definition 4.1) and *secondary edges* (Definition 4.2).

Definition 4.1 (Primary Edges)**.** For a transitive relation A, a primary A-edge is created via a production rule that is not in the form of $A ::= A\ A$.

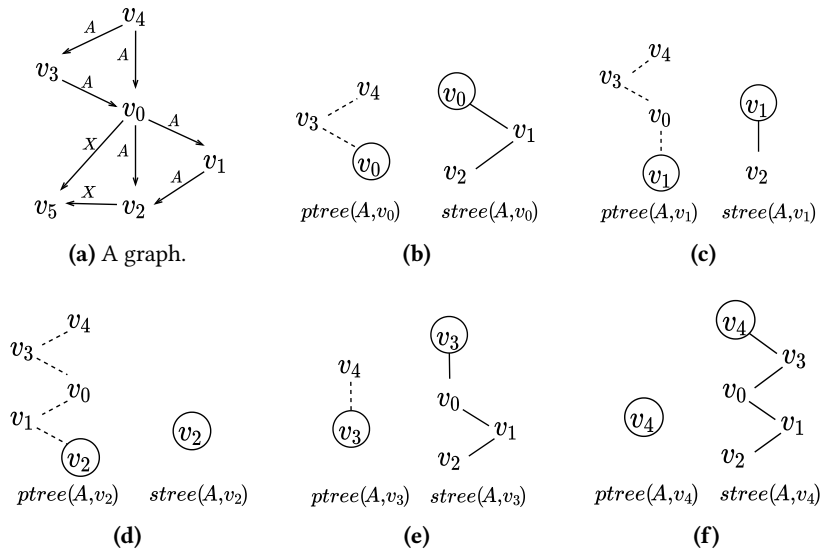


Fig. 4. Predecessor trees and successor trees. The root of each spanning tree is circled.

Definition 4.2 (Secondary Edges)**.** For a transitive relation A, a secondary A-edge is created via the production rule $A ::= A\ A$.

It is important to note that while traversing the graph along the transitive A-edges, whether there are secondary A-edges does not affect the nodes that can be visited. Conversely, the existence of primary A-edges may result in additional nodes which can be visited. Figure 2(c) of our motivating example illustrates this property: adding the A-edges created via $A ::= a$ increases the number of nodes that can be visited by traversing along the A-edges, whereas adding the A-edges created via $A ::= A\ A$ does not. Thus, while constructing spanning trees, we do not consider the secondary edges.

4.1.1 Predecessor Trees and Successor Trees. We use the primary edges to construct spanning trees to determine the computation order. Corresponding to the adjacency lists (Section 3.2), for each transitive relation A, we assign to each node v_i a *predecessor tree* $ptree(A, v_i)$ and a *successor tree* $stree(A, v_i)$. Both $ptree(A, v_i)$ and $stree(A, v_i)$ are rooted at v_i with the following properties:

$ptree(A, v_i)$: (1) for any node $v_j \neq v_i$, $v_j \in ptree(A, v_i)$ iff $v_j \in pred(A, v_i)$;
(2) for any two nodes v_k, v_l such that v_l is a child of v_k in $ptree(A, v_i)$, there is a primary A-edge $v_l \xrightarrow{A} v_k \in E$.

$stree(A, v_i)$: (1) for any node $v_j \neq v_i$, $v_j \in stree(A, v_i)$ iff $v_j \in succ(A, v_i)$;
(2) for any two nodes v_k, v_l such that v_l is a child of v_k in $stree(A, v_i)$, there is a primary A-edge $v_k \xrightarrow{A} v_l \in E$.

The above properties provide an efficient tree traversal to determine the computation order of partial transitive relations. The benefit of the above properties is two-fold. On one hand, the first property of $ptree(A, v_i)$ and $stree(A, v_i)$ ensures that the traversal can touch all the A-predecessors and A-successors of v_i, which ensures complete edge additions. On the other hand, the tree structure provides an efficient traversal to determine computation order. Specifically, traversing the A-predecessors/A-successors of a node can be done in $O(|pred(A, v_i)|)/O(|succ(A, v_i)|)$ time, as the number of edges in a tree is always equal to the number of nodes minus one.

Algorithm 2: Singly recursive edge derivations.

```
1  Function CheckPtree(X, A, v_x, v_y, v_t)
2      for each v_z child of v_x in ptree(A, v_t) do
3          if v_z --X--> v_y ∉ E then
4              add v_z --X--> v_y to E and to W;
5              CheckPtree(X, A, v_z, v_y, v_t);
6  Function CheckStree(X, A, v_x, v_y, v_t)
7      for each v_z child of v_y in stree(A, v_t) do
8          if v_x --X--> v_z ∉ E then
9              add v_x --X--> v_z to E and to W;
10             CheckStree(X, A, v_x, v_z, v_t);
```

Example 4.1 (Predecessor Trees and Successor Trees). Figure 4(a) is a graph where A is transitive and (b)–(f) display the predecessor trees and successor trees of nodes $v_0, \ldots, v_4$. The root of each tree is marked by a circle, and the edges in the predecessor trees are marked by dashed edges. Note that the two secondary edges $v_4 \xrightarrow{A} v_0$ and $v_0 \xrightarrow{A} v_2$ are not included in any of the spanning trees.

4.1.2 Hybrid Graph Representation. We embed the spanning-tree model into the standard adjacency-list graph representation, constructing a *hybrid graph representation*. In our hybrid graph representation, for a transitive relation A and a node v_i, each element $v_j \in \mathit{pred}(A, v_i)$ is maintained as a pointer pointing to the node $v_j \in \mathit{ptree}(A, v_i)$, and each element $v_k \in \mathit{succ}(A, v_i)$ is maintained as a pointer pointing to the node $v_k \in \mathit{stree}(A, v_k)$. Our hybrid graph representation keeps good time efficiency for both lookups and traversals. Specifically, we perform lookups of nodes in adjacency lists[1], and traversals of predecessors and successors of nodes in predecessor trees and successor trees, respectively.

4.1.3 Efficient Singly Recursive Edge Creations. Algorithm 2 performs efficient derivations based on singly recursive rules. Like `CheckPred` and `CheckSucc` in Algorithm 1, `CheckPtree`/`CheckStree` create edges based on $X ::= A\ Y \in P / X ::= Y\ A \in P$ for all transitive relations A. The difference is that `CheckPtree` traverses a predecessor tree $\mathit{ptree}(A, v_t)$ (whose root v_t is input as parameter), instead of traversing the adjacency list $\mathit{pred}(A, v_x)$. Similarly, `CheckStree` traverses $\mathit{stree}(A, v_t)$ instead of traversing $\mathit{succ}(A, v_t)$.

While processing an edge $v_x \xrightarrow{X} v_y$, for each $X ::= A\ X$ where A is transitive, `CheckPtree`(X, A, v_x, v_y, v_x) is called (with v_t specified as v_x) to traverse $\mathit{ptree}(A, v_x)$ from the root to the leaves to create and add the X-edges $v_z \xrightarrow{X} v_y$ to the graph, where $v_z \in \mathit{pred}(A, v_x)$. Similarly, for each $X ::= X\ A$, `CheckStree`(X, A, v_x, v_y, v_y) is called to traverse $\mathit{stree}(A, v_y)$ to create and add the X-edges $v_x \xrightarrow{X} v_z'$ to the graph, where $v_z' \in \mathit{succ}(A, v_y)$.

[1]Implementing the adjacency lists by hash tables can reduce the time complexity of lookups to $O(1)$.

Correctness Property of Algorithm 2. In Algorithm 1, while processing $v_i \xrightarrow{X} v_j$, for all transitive A, replacing $\texttt{CheckPred}(X, A, v_i, v_j)$/$\texttt{CheckSucc}(X, A, v_i, v_j)$ by $\texttt{CheckPtree}(X, A, v_i, v_j, v_i)$ resp. $\texttt{CheckStree}(X, A, v_i, v_j, v_j)$ yields identical results with respect to the standard CFL-reachability solution (Algorithm 1).

Algorithm 2 holds the above property from two aspects. Here we only discuss CheckPtree as CheckStree is similar. On one hand, with respect to $\texttt{CheckPtree}(X, A, v_x, v_y, v_x)$ which processes $v_x \xrightarrow{X} v_y$, if the traversal on $ptree(A, v_x)$ is never truncated by Line 3, this means that all the nodes $v_z \in ptree(A, v_x)$ are visited and the edges $v_z \xrightarrow{X} v_y$ are added to the graph and to the worklist. This is equivalent to what $\texttt{CheckPred}(X, A, v_x, v_y)$ does. On the other hand, if the traversal on $ptree(A, v_x)$ is truncated by Line 3, this means that the edge $v_z \xrightarrow{X} v_y$ has already been added to the graph. The edge must also have been previously added to the worklist and has been processed (or will be processed) by other calls of CheckPtree, which add all $v'_z \xrightarrow{X} v_y$ to the graph and to the worklist, where v_z is an element of the subtree rooted at $v_z \in ptree(A, v_x)$. Therefore, the final result of $\texttt{CheckPtree}(X, A, v_x, v_y, v_x)$ is also equivalent to what $\texttt{CheckPred}(X, A, v_x, v_y)$ do (also see our supplementary material for detailed proof).

Algorithm 2 stops the redundant derivations at the first stage via the termination criteria at Line 3 and Line 8. Specifically, with respect to CheckPtree, once visiting a node $v_z \in ptree(A, v_t)$ such that the associated edge $v_z \xrightarrow{X} v_y$ is already in the graph, the algorithm stops traversing the subtree rooted at v_z. This avoids the redundant derivations of $v'_z \xrightarrow{X} v_y$, where v'_z is an element of the subtree. Such traversals stop similarly in $stree(A, v_t)$.

Example 4.2 (Singly Recursive Edge Creations). In Figure 4(a), given $X ::= A\ X \in P$, while processing $v_0 \xrightarrow{X} v_5$, $\texttt{CheckPtree}(X, A, v_0, v_5, v_0)$ traverses $ptree(A, v_0)$ (Figure 4(b)) and adds $v_3 \xrightarrow{X} v_5$ and $v_4 \xrightarrow{X} v_5$ to the graph. While processing $v_2 \xrightarrow{X} v_5$, $\texttt{CheckPtree}(X, A, v_2, v_5, v_2)$ traverses $ptree(A, v_2)$ (Figure 4(d)). While visiting v_1, it adds $v_1 \xrightarrow{X} v_5$ to the graph. Then it goes to v_0 child of v_1 in $ptree(A, v_2)$ and stops the traversal. This is because $v_0 \xrightarrow{X} v_5$ is already in the graph (Line 3, Algorithm 2). Thus, the redundant derivations of two edges $v_3 \xrightarrow{X} v_5$ and $v_4 \xrightarrow{X} v_5$ is avoided.

4.2 Dynamic Construction of Spanning Trees

Algorithm 2 utilizes the two properties of the spanning-tree model (listed in Section 4.1.1) to reduce the redundant derivations caused by singly recursive rules. The key step in our CFL-reachability algorithm is to update the spanning trees representing the transitive relations among nodes and maintain the two properties.

Adding an edge $v_i \xrightarrow{A} v_j$, where A is transitive, to the graph changes $succ(A, v_k)$ and $pred(A, v'_k)$ for all $k \in pred(A, v_i) \cup \{v_i\}$ and $k' \in succ(A, v_j) \cup \{v_j\}$ via $succ(A, v_k) = succ(A, v_k) \cup succ(A, v_j)$ and $pred(A, v'_k) = pred(A, v'_k) \cup pred(A, v_i)$. Note that $pred(A, v_i)$ and $succ(A, v_j)$ take constant time during the edge addition. Thus, the updates of the spanning trees are similar to handling singly recursive rules.

We propose Algorithm 3, a dynamic construction algorithm, to update the predecessor and successor trees when solving CFL-reachability. Algorithm 3 consists of a main procedure NewTrEdge and three subprocedures TravPtree, TravStree and Update. Given an edge $v_{pt} \xrightarrow{A} v_{st}$ where A is transitive, TravPtree and TravStree perform a nested traversal with TravPtree that traverses

$ptree(A, v_{pt})$ nested in `TravStree` that traverses $stree(A, v_{st})$. During the nested traversal, `Update` updates the spanning trees and the adjacency lists of the visited nodes simultaneously, ensuring that for all $(v_k, v'_k) \in ptree(A, v_{pt}) \times stree(A, v_{st})$:

(1) $v_k \in ptree(A, v'_k)$ and $v'_k \in stree(A, v_k)$;
(2) $v_k \in pred(A, v'_k)$ and $v'_k \in succ(A, v_k)$.

We first illustrate Algorithm 3 using the following example:

Example 4.3 (Dynamic Construction of Spanning Trees). Adding a primary edge $v_2 \xrightarrow{A} v_3$ to Figure 4(a) means that for all $(v_k, v'_k) \in pred(A, v_2) \times succ(A, v_3)$, there will be $v_k \in ptree(A, v'_k)$ and $v'_k \in stree(A, v_k)$. `NewTrEdge`(A, v_2, v_3) realizes this by traversing $ptree(A, v_2)$ and $stree(A, v_3)$, and updating the spanning trees (and adjacency lists simultaneously) as shown in Figure 5(a)–(d).

Initially, the traversals of $ptree(A, v_2)$ and $stree(A, v_3)$ start at their roots v_2 and v_3 respectively, as marked in red in Figure 5(a). In this step, v_2 is added to $ptree(A, v_3)$ as a child of v_3, and v_3 is added to $stree(A, v_2)$ as a child of v_2, as marked in blue in Figure 5(a). After this, the inner traversal of $ptree(A, v_2)$ visits v_1, a child of v_2, and the outer traversal of $stree(A, v_3)$ stays at v_3. This step adds v_1 to $ptree(A, v_3)$ as a child of v_2 and adds v_3 to $stree(A, v_1)$ as child of v_2, as shown in Figure 5(b). This can be viewed as copying the edge v_2 to v_1 from $ptree(A, v_2)$ to $ptree(A, v_3)$. Then the depth-first traversal of $ptree(A, v_2)$ continues, as shown in Figure 5(c), updating the spanning trees rooted at the visited nodes until the traversal finishes.

After finishing the inner traversal of $ptree(A, v_2)$, the outer traversal of $stree(A, v_3)$ visits v_0, a child of v_3, and starts another inner traversal of $ptree(A, v_2)$, as Figure 5(d). The nested traversal terminates when for all $(v_k, v'_k) \in ptree(A, v_2) \times stree(A, v_3)$, $v_k \in ptree(A, v'_k)$ and $v'_k \in stree(A, v_k)$.

The nested traversals and edge updates of Algorithm 3 work as follows:
Nested depth-first traversal. `TravPtree` and `TravStree` accept seven parameters:

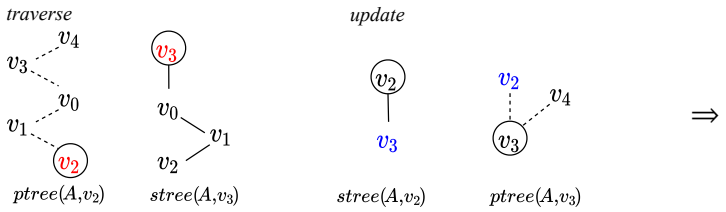
A - the label of the transitive edges;
v_{pt} - the root of $ptree(A, v_{pt})$, i.e., the predecessor tree to be traversed;
v_{st} - the root of $stree(A, v_{st})$, i.e., the successor tree to be traversed;
v_{py} - the node currently visited in $ptree(A, v_{pt})$;
v_{px} - the parent of v_{py} in $ptree(A, v_{pt})$;
v_{sy} - the node currently visited in $stree(A, v_{st})$;
v_{sx} - the parent of v_{sy} in $stree(A, v_{st})$.

Given a new primary edge $v_{pt} \xrightarrow{A} v_{st}$, the traversal of $ptree(A, v_{pt})$ in `TravPtree` starts at its root v_{pt} which does not have an actual parent. So the pseudo parent of v_{pt} is set as v_{st}. Similarly, the traversal of $stree(A, v_{st})$ in `TravStree` starts at v_{st} and the pseudo parent of v_{st} is set as v_{pt}, as shown in Line 2.

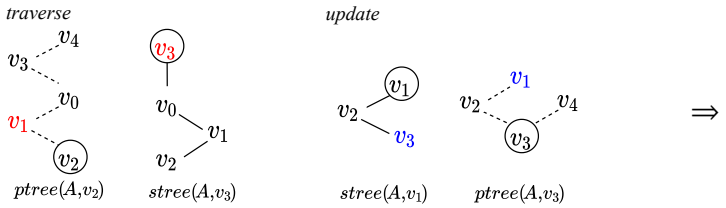
The traversal of $ptree(A, v_{pt})$ is nested in the traversal of $stree(A, v_{st})$, as shown in Line 4. During the traversal of $ptree(A, v_{pt})$, nodes v_{sx} and v_{sy} relevant to the outer traversal remain constant. For each v_{pz} child of v_{py} in $ptree(A, v_{pt})$, the traversal steps from v_{py} to v_{pz} only if $v_{pz} \xrightarrow{A} v_{sy}$ is not in the graph, as shown in Lines 10–12. After finishing the inner traversal of $ptree(A, v_{pt})$, the outer traversal of $stree(A, v_{st})$ steps to v_{sz} a child of v_{sy} in $stree(A, v_{st})$ only if $v_{py} \xrightarrow{A} v_{sz}$ is not in the graph (Lines 5–7), and starts another inner traversal of $ptree(A, v_{pt})$ as shown in Line 4.

The nested depth-first traversal of Algorithm 3 is terminated when any one of the following two constraints is satisfied:

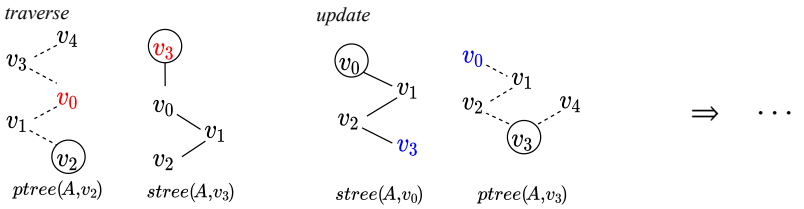
- all the nodes pairs $(v_k, v'_k) \in ptree(A, v_{pt}) \times stree(A, v_{st})$ are visited;
- all the attempts of visiting new tree nodes are stopped by Line 6 or Line 11.



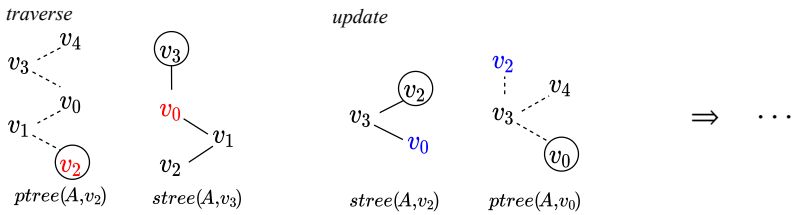
Step (1): `NewTrEdge`(A, v_2, v_3), i.e., Algorithm 3, starts with `TravPtree`$(v_2, v_3, v_2, v_3, v_2, v_3)$, i.e., Line 4, in `TravStree`$(v_2, v_3, v_2, v_3, v_2, v_3)$, i.e., Line 2. v_2 and v_3 are added to $ptree(A, v_3)$ and $stree(A, v_2)$ respectively.



Step (2): `TravPtree`$(v_2, \boldsymbol{v_2}, \boldsymbol{v_1}, v_3, v_2, v_3)$. v_1 and v_3 are added to $ptree(A, v_3)$ and $stree(A, v_1)$ respectively. In particular, v_1 is added as a child of v_2 in $ptree(A, v_3)$, which is the same as in $ptree(A, v_2)$.



Step (3): `TravPtree`$(v_2, \boldsymbol{v_1}, \boldsymbol{v_0}, v_3, v_2, v_3)$. v_0 and v_3 are added to $ptree(A, v_3)$ and $stree(A, v_0)$ respectively. Similar to Step (2), v_0 is added as a child of v_1 in $ptree(A, v_3)$, which is the same as in $ptree(A, v_2)$.



Step (6): `TravPtree`$(v_2, v_3, v_2, v_3, \boldsymbol{v_3}, \boldsymbol{v_0})$. After finishing the inner traversal of $ptree(A, v_2)$, the outer traversal of $stree(A, v_3)$ visits v_0. Similarly, v_0 is added as a child of v_3 in $stree(A, v_2)$, which is the same as in $stree(A, v_3)$.

Fig. 5. Processing a new edge $v_2 \xrightarrow{A} v_3$ added to Figure 4(a) by `NewTrEdge`(A, v_2, v_3), which traverses $ptree(A, v_2)$ and $stree(A, v_3)$, and updates the *ptrees* and *strees* of the visited nodes. In each step, the nodes being visited are marked in red, and the nodes newly added to the spanning trees are marked in blue.

Algorithm 3: Dynamic construction of the spanning-tree model.

```
1  Function NewTrEdge(A, v_pt, v_st)
2      TravStree(A, v_pt, v_st, v_pt, v_st, v_pt, v_st);
3  Procedure TravStree(A, v_pt, v_px, v_py, v_st, v_sx, v_sy)
4      TravPtree(A, v_pt, v_px, v_py, v_st, v_sx, v_sy);
5      for each v_sz child of v_sy in stree(A, v_st) do
6          if v_py -A-> v_sz ∉ E then
7              TravStree(A, v_pt, v_px, v_py, v_st, v_sy, v_sz);
8  Procedure TravPtree(A, v_pt, v_px, v_py, v_st, v_sx, v_sy)
9      Update(A, v_px, v_py, v_sx, v_sy);
10     for each v_pz child of v_py in ptree(A, v_pt) do
11         if v_pz -A-> v_sy ∉ E then
12             TravPtree(A, v_pt, v_py, v_pz, v_st, v_sx, v_sy);
13 Procedure Update(A, v_px, v_py, v_sx, v_sy)
14     add v_py -A-> v_sy as a secondary edge to E and to W;
15     if v_py ≠ v_sy and v_py ∉ ptree(A, v_sy) then
16         add a new node v_py pointed to by v_py ∈ pred(A, v_sy) to ptree(A, v_sy) as a child of v_px;
17         add a new node v_sy pointed to by v_sy ∈ succ(A, v_py) to stree(A, v_py) as a child of v_sx;
```

Updating spanning trees and adjacency lists. When creating a new edge $v_{py} \xrightarrow{A} v_{sy}$ not in the graph, Algorithm 3 calls `Update`$(A, v_{px}, v_{py}, v_{sx}, v_{sy})$ at Line 9 to update the spanning trees and the adjacency lists simultaneously. `Update` first adds $v_{py} \xrightarrow{A} v_{sy}$ to the graph at Line 14. In particular, the edge $v_{py} \xrightarrow{A} v_{sy}$ is marked as a "secondary" edge so that it will not be processed by the future calls of `NewTrEdge`. The updates of $ptree(A, v_{sy})$ and $stree(A, v_{py})$ by Lines 15–17 can be viewed as follows: (1) copying the edge from v_{px} to v_{py} in $ptree(A, v_{pt})$ and attaching the copy to the node v_{px} in $ptree(A, v_{sy})$, and (2) copying the edge from v_{sx} to v_{sy} in $stree(A, v_{st})$ and attaching the copy to the node v_{sx} in $stree(A, v_{py})$. To avoid redundant tree edges, the above two steps are performed only if $v_{py} \neq v_{sy}$ and $v_{py} \notin ptree(A, v_{sy})$. The simultaneous updates of spanning trees and adjacency lists of `Update` maintains the two properties of the spanning-tree model (Section 4.1.1).

Executing Algorithm 3 can be viewed as updating $stree(A, v_k)$ and $ptree(A, v'_k)$ for all $(v_k, v'_k) \in pred(A, v_{pt}) \times succ(A, v_{st})$ using the following two steps: (1) pruning a copy of $stree(A, v_{st})/ptree(A, v_{pt})$ by eliminating the nodes already in $stree(A, v_k)/ptree(A, v'_k)$; and (2) attaching the pruned copy to the node v_{pt}/v_{st} in $stree(A, v_k)/ptree(A, v'_k)$. It is almost obvious that using Algorithm 3 to process the primary A-edges maintains the transitive closure of the relation A. Similar to Algorithm 2, we have the following property for Algorithm 3:

Correctness Property of Algorithm 3. In Algorithm 1, replacing $\texttt{CheckPred}(A, A, v_i, v_j)$ and $\texttt{CheckSucc}(A, A, v_i, v_j)$ by $\texttt{NewTrEdge}(A, v_i, v_j)$ for all primary edges $v_i \xrightarrow{A} v_j$ and omitting $\texttt{CheckPred}(A, A, v_i, v_j)$ and $\texttt{CheckSucc}(A, A, v_i, v_j)$ for all secondary edges $v_i \xrightarrow{A} v_j$ yields identical results with respect to the standard CFL-reachability solution (Algorithm 1).

Notably, $\texttt{CheckPred}(A, v_i, v_j)$ and $\texttt{CheckSucc}(A, v_i, v_j)$ can be omitted while processing a secondary edge $v_i \xrightarrow{A} v_j$ because all the secondary edges are processed in the calls of NewTrEdge for processing primary edges.

Algorithm 3 avoids redundancy by the termination criteria at Line 6 and Line 11. In Line 6, $v_{py} \xrightarrow{A} v_{sz} \in E$ avoids the repeated traversal of v'_{sz} descendants of v_{sz} in $stree(A, v_{st})$. Similarly, in Line 11, $v_{pz} \xrightarrow{A} v_{sy} \in E$ avoids the repeated traversal of v'_{pz} descendants of v_{pz} in $ptree(A, v_{pt})$.

4.3 PocR: A Fast Partially Ordered CFL-Reachability Algorithm for All-Pairs Analyses

With Algorithms 2 and 3 running on top of the hybrid graph representation discussed in Section 4.1, we propose PocR, a fast partially ordered CFL-reachability algorithm for all-pairs analysis, as given in Algorithm 4.

PocR consists of two parts: initialization (Lines 2–5) and solving reachability (Lines 6–20). Initialization first follows the initialization scheme of the standard algorithm (Lines 11–15, Algorithm 1) to initialize the graph and the worklist. Then, it initializes the predecessor tree and successor tree for each node $v_i \in V$ (Lines 3–5). Solving reachability also follows the strategy of the standard algorithm, i.e., iteratively solving the edges in the worklist and adding new edges to the graph and to the worklist until no new edges can be added to the graph.

PocR differs from the standard algorithm in handling edge derivations based on singly or doubly recursive rules. It uses NewTrEdge to deal with edge derivations based on doubly recursive rules (Line 9) and replaces CheckPred/CheckSucc by TravPtree/TravStree to create edges based on singly recursive rules when the derivation needs to traverse the predecessors or successors of nodes associated with transitive relations (Lines 15 and 19). In particular, all transitive edges created during CFL-reachability solving are marked "**primary**" by default except for those created in NewTrEdge (Line 14, Algorithm 3), and NewTrEdge only accepts primary edges. The secondary edges created in NewTrEdge are further used to create new edges based on non-doubly recursive rules in the subsequent procedures (Lines 10–20, Algorithm 4).

In summary, when processing an edge $v_i \xrightarrow{Y} v_j$, PocR only differs from the standard algorithm in the following two aspects:

(1) replacing $\texttt{CheckPred}(X, A, v_i, v_j)$/$\texttt{CheckSucc}(X, A, v_i, v_j)$ with $\texttt{CheckPtree}(X, A, v_i, v_j, v_i)$/$\texttt{CheckStree}(X, A, v_i, v_j, v_j)$ when A is transitive and $X = Y$;

(2) replacing $\texttt{CheckPred}(Y, Y, v_i, v_j)$/$\texttt{CheckSucc}(Y, Y, v_i, v_j)$ with $\texttt{NewTrEdge}(Y, v_i, v_j)$ if $v_i \xrightarrow{Y} v_j$ is primary and omitting $\texttt{CheckPred}(Y, Y, v_i, v_j)$ /$\texttt{CheckSucc}(Y, Y, v_i, v_j)$ if $v_i \xrightarrow{Y} v_j$ is secondary.

According to the properties of Algorithms 2 and 3, these replacements do not change the CFL-reachability solution. Namely, for a CFL-reachability problem, PocR produces an identical solution to the standard algorithm.

5 DISCUSSION: EFFECTIVENESS OF POCR

As demonstrated in Section 4, PocR uses ordered derivations based on the spanning trees to reduce redundant derivations for singly and doubly recursive rules. It is called "partially ordered" because

Algorithm 4: Pocr: partially ordered CFL-reachability algorithm.

```
1  Function ODCR(CFG, G)
2      init();                                              /* Lines 11–15, Algorithm 1 */
3      for each transitive relation A do
4          for each node v_i ∈ V do
5              set both ptree(A, v_i) and stree(A, v_i) a tree containing a single node v_i as the root.
6      while W ≠ ∅ do
7          select and remove an element v_i -Y-> v_j from W;
8          if v_i -Y-> v_j is a primary edge then
9              NewTrEdge(Y, v_i, v_j);                      /* Algorithm 3 */
10         else
11             for each production X ::= Y ∈ P do
12                 if v_i -X-> v_j ∉ E then add v_i -X-> v_j to E and to W;
13             for each X ::= Y Z ∈ P and ¬(X = Y = Z) do
14                 if Z is transitive and X = Y then
15                     CheckStree(X, Z, v_i, v_j, v_j);     /* Algorithm 2 */
16                 else CheckSucc(X, Z, v_i, v_j);          /* Lines 19–21, Algorithm 1 */
17             for each X ::= Z Y ∈ P and ¬(X = Y = Z) do
18                 if Z is transitive and X = Y then
19                     CheckPtree(X, Z, v_i, v_j, v_i);     /* Algorithm 2 */
20                 else CheckPred(X, Z, v_i, v_j);          /* Lines 16–18, Algorithm 1 */
```

it does not order the edge derivations through the production rules where the right-hand side has no transitive relation. Hence, the redundant derivations caused by such production rules are not eliminated by Pocr. In particular, given $X ::= Y\ Z \in P$, two paths $v_1 \xrightarrow{Y} v_2 \xrightarrow{Z} v_3$ and $v_1 \xrightarrow{Y} v_4 \xrightarrow{Z} v_3$ cause one redundant derivation of $v_1 \xrightarrow{X} v_3$, which is not transitive redundancy and is not handled by Pocr. The degree/effectiveness of Pocr's redundancy elimination is also related to the CFG of a particular CFL-reachability problem. We further discuss two aspects of this issue as below.

5.1 Grammars Benefiting from Pocr

A comparison of Algorithm 4 and Algorithm 1 shows that for a CFL-reachability problem where there is no transitive relation, the part of Pocr which solves reachability is identical to that of Algorithm 1, because Lines 8–9, 14–15 and 18–19 of Algorithm 4 will never be executed. Therefore, Pocr does not benefit the problems where there are no transitive relations in the CFG.

However, in some CFL-reachability problems, the transitive relations are implicit. We can rewrite the input grammars in CFL-reachability to utilize Pocr. For example, the following grammar

$$A ::= a\ B \mid \varepsilon$$
$$B ::= b\ A$$

can be rewritten as $A ::= A\ A \mid a\ b \mid \varepsilon$ to be used by Pocr. Conversely, for a CFL-reachability problem that has no transitive relation, i.e., its CFG cannot be rewritten to obtain any doubly recursive rule, we assume that it does not have transitive redundancy.

Moreover, some grammars containing doubly recursive rules can be rewritten using only left- or right-recursive rules. For example, the CFG in Figure 2(a) can be rewritten into

$$F_i ::= F_i\ a \mid f_i.$$

Given this modified grammar, there will be no A-edge in the graph, and we will not have the redundant derivations caused by the four cases in Figure 2(d). However, transitive relations are prevalent in many popular real-world CFL-reachability problems, and the input graphs are typically large. Existing techniques [Melski and Reps 2000; Wang et al. 2017] still suffer from a large number of redundant derivations while solving with such modified grammars because they do not exploit an effective computation order. This is also confirmed in our experiment (Section 6.4).

5.2 Grammar-Driven Redundancy Reduction via Optimized Pocr

Here we provide an optimization of Pocr to further improve its efficiency given any grammar with the following three properties:

(1) the grammar has one or more transitive relations;
(2) for any production rule $X ::= Y\ A \in P$ and $X ::= A\ Y \in P$ where A is transitive, $X = Y$;
(3) there is no $X ::= A \in P$ where A is transitive and $X \neq A$.

For such type of grammars, while processing a primary edge $v_i \xrightarrow{A} v_j$ (Line 9, Algorithm 4), we do not add the secondary edges created in `NewTrEdge` to the worklist as in Line 14 of Algorithm 3 (we still add such edges to the graph). Instead, we only add $v_i \xrightarrow{A} v_j$ itself as a secondary edge to the worklist. The insight is to use `CheckPtree` and `CheckStree` to deal with derivations based on as many singly recursive rules as possible.

We briefly demonstrate the feasibility of the optimized Pocr in handling grammars with the aforementioned three properties. For singly recursive rules, we only discuss $X ::= X\ A$ because $X ::= A\ X$ is similar. In a CFL-reachability instance where $X ::= X\ A \in P$ and A is transitive, adding a primary A-edge $v_i \xrightarrow{A} v_j$ to the graph results in $v_l \xrightarrow{X} v_k \in E$ for all $(v_l, v_k) \in \mathit{pred}(X, v_i) \times \mathit{stree}(A, v_j)$. In the original Pocr, $\texttt{NewTrEdge}(A, v_i, v_j)$ is first called to create and add all the new secondary edges $v'_k \xrightarrow{A} v_k$ to the graph, where $(v'_k, v_k) \in \mathit{ptree}(A, v_i) \times \mathit{stree}(A, v_j)$. Then $\texttt{CheckPred}(X, X, v'_k, v_k)$ is called to process each $v'_k \xrightarrow{A} v_k$ in the worklist (Line 20, Algorithm 4) so that for all $v_l \in \mathit{pred}(X, v'_k)$, $v_l \xrightarrow{X} v_k \in E$. Because $X ::= X\ A$, for all $v_l \in \mathit{pred}(X, v'_k)$, $v_l \in \mathit{pred}(X, v_i)$. Hence, all $v_l \xrightarrow{X} v_k$ such that $(v_l, v_k) \in \mathit{pred}(X, v_i) \times \mathit{stree}(A, v_j)$ are added to the graph.

The optimized Pocr handles this in a different way. First, $\texttt{NewTrEdge}(Y, v_i, v_j)$ only adds $v_i \xrightarrow{A} v_j$ as a secondary edge to the graph and to the worklist. Then for each $X ::= X\ A \in P$, $\texttt{CheckPred}(X, X, v_i, v_j)$ processes $v_i \xrightarrow{A} v_j$, adding all the new $v_l \xrightarrow{X} v_j$ such that $v_l \in \mathit{pred}(X, v_i)$ to the graph and to the worklist. Processing all the $v_l \xrightarrow{X} v_j$ in the worklist via $\texttt{CheckStree}(X, A, v_l, v_j, v_j)$ results in $v_l \xrightarrow{X} v_k \in E$ for all $(v_l, v_k) \in \mathit{pred}(X, v_i) \times \mathit{stree}(A, v_j)$.

The original Pocr and the optimized Pocr obtain the same results when dealing with $X ::= X\ A$. Similarly, they also obtain the same results when dealing with $X ::= A\ X$. Furthermore, the three properties listed at the beginning of the subsection mean that there is no need to process $X ::= A$, $X ::= Y\ A$ and $X ::= A\ Y$ where $X \neq Y$ for all transitive A. Therefore, the optimized Pocr is applicable to CFL-reachability whose grammar has the three properties.

Unlike the original Pocr, `CheckPred` and `CheckSucc` in the optimized Pocr are only called to process the input primary edges of `NewTrEdge`. Most of the derivations based on singly recursive rules are done by `CheckPtree` and `CheckStree`, instead of `CheckPred` and `CheckSucc`. Note that `CheckPtree` and `CheckStree` reduce redundant derivations whereas `CheckPred` and `CheckSucc` do not. So the optimized Pocr further improves the efficiency of solving CFL-reachability. It is also

interesting to note that there is a large variety of real-world static analyses that can be formulated into CFL-reachability problems whose grammars have the aforementioned three properties, e.g., dataflow/valueflow analysis [Reps et al. 1995], typestate analysis [Wang et al. 2020b], alias analysis [Zheng and Rugina 2008], etc. Thus, this grammar-driven optimization is worth incorporating.

6 EXPERIMENTS

In this section, we evaluate the performance of PoCR by applying it to two popular static analyses for C/C++: context-sensitive value-flow analysis [Sui et al. 2014] and field-sensitive alias analysis [Zheng and Rugina 2008], where transitive redundancy dominates. In our experiment, we use cycle elimination [Nuutila and Soisalon-Soininen 1994] and variable substitution [Rountev and Chandra 2000] for offline processing of the graphs abstracts from the benchmark programs. The baseline of our experiment is the standard CFL-reachability algorithm [Melski and Reps 2000] accepting the preprocessed input graphs. We also compare our approach with two recent open-source CFL-reachability/Datalog tools, Graspan [Wang et al. 2017] and Soufflé [Jordan et al. 2016b]. We perform all-pairs CFL-reachability analysis in PoCR and all the baselines for both clients.

Our experiments aim to answer the following research questions:

RQ 1. How many redundant derivations can PoCR reduce in real-world CFL-reachability problems based on the two popular clients?
RQ 2. How is the performance of CFL-reachability improved by eliminating transitive redundancy via PoCR?
RQ 3. How about the performance of PoCR when comparing it with the grammar rewriting method which removes doubly recursive rules from the grammar?

We summarize our experimental results as follows: (1) PoCR is highly effective in taming transitive redundancy with 98.50% and 97.26% of redundant derivations being eliminated for context-sensitive value-flow analysis and field-sensitive alias analysis, respectively. (2) By eliminating redundant derivations, PoCR significantly accelerates the standard algorithm by 21.48× and 19.57× respectively for the value-flow and alias analyses. (3) Though grammar rewriting can reduce some redundancy by removing doubly recursive rules, PoCR is still much more effective in reducing redundant derivations than grammar rewriting.

6.1 Experimental Setup

We have conducted our experiment on a platform consisting of an eight-core 2.60GHz Intel Xeon CPU with 128 GB memory, running Ubuntu 18.04.

Value-flow analysis. Our context-sensitive value-flow analysis is conducted on the sparse value-flow graphs (SVFG) [Sui et al. 2014]. We use the context-free grammar (CFG) in Figure 6 for the value-flow analysis, where "$call_i$" and "ret_i" denote, respectively, a call and a return with a callsite index i, "a" denotes an assignment instruction, and "A" denotes a value flow. Note that the grammar in Figure 6 only considers context-sensitivity. However, each field object is represented as a single node in the field-sensitive SVFG, so the analysis is also field-sensitive.

Alias analysis. The CFG for C/C++ field-sensitive alias analysis is listed in Figure 7, which is from [Zheng and Rugina 2008]. In the grammar, a denotes an assignment, d denotes a pointer dereference, f_i denotes the address of the i-th field, A denotes a value flow, M denotes memory aliasing, and V denotes value aliasing. The alias analysis is performed on the program expression graph (PEG), which is bi-directed, i.e., for each edge $v_i \xrightarrow{X} v_j \in E$, there is a reverse edge $v_j \xrightarrow{\overline{X}} v_i \in E$.

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

(a) Context-free grammar.

$$\begin{aligned} A &::= A\ A \mid CA_i\ ret_i \mid a \mid \varepsilon \\ CA_i &::= call_i\ A \end{aligned}$$

(b) Normalized grammar.

Fig. 6. CFG for context-sensitive value-flow analysis.

$$\begin{aligned} M &::= \overline{d}\ V\ d \\ V &::= \overline{A}\ V\ A \mid \overline{f_i}\ V\ f_i \mid M \mid \varepsilon \\ A &::= A\ A \mid a\ M? \mid \varepsilon \\ \overline{A} &::= \overline{A}\ \overline{A} \mid M?\ \overline{a} \mid \varepsilon \end{aligned}$$

(a) Context-free grammar.

$$\begin{aligned} M &::= DV\ d \\ DV &::= \overline{d}\ V \\ V &::= \overline{A}\ V \mid V\ A \mid FV_i\ f_i \mid M \mid \varepsilon \\ FV_i &::= \overline{f_i}\ V \\ A &::= A\ A \mid a\ M \mid a \mid \varepsilon \\ \overline{A} &::= \overline{A}\ \overline{A} \mid M\ \overline{a} \mid \overline{a} \mid \varepsilon \end{aligned}$$

(b) Normalized grammar.

Fig. 7. CFG for field-sensitive alias analysis.

Setup and Benchmarks. The SVFG and PEG of each program are constructed from the bitcode files compiled by Clang-12.0.0 and linked via `wllvm`[2] for whole-program all-pairs CFL-reachability analysis. The SVFG and PEG are preprocessed by cycle elimination [Tarjan 1972] which merges cycles comprised of a-edges and variable substitution [Rountev and Chandra 2000] which contracts particular a-edges. The preprocessing is to make sure the input graph is compacted after applying the existing offline approach to make sure the input does not favor our online approach undesirably. We used 10 SPEC 2017 C/C++ programs for our evaluation. We did not include three small programs (`lbm`, `mcf` and `deepsjeng` whose sizes are less than 1 MB). The other three C/C++ programs in SPEC 2017, i.e., `xalancbmk`, `gcc` and `blender`, failed to be linked by `wllvm`. Table 1 lists the size and graph statistics of each program.

Implementation. We have implemented our Pocr on top of the LLVM compiler, and the sub-project SVF [Sui and Xue 2016]. To study the relationship between the improvement of performance and the reduction of redundant derivations, we also implement the standard algorithm [Melski and Reps 2000] and the edge-reduction technique described in the paper of Graspan (Section 4.2 in [Wang et al. 2017]) on top of SVF. Tables 2 and 3 list the main results of our experiments, where "Base" and "GSA" denote, respectively, the standard algorithm and the edge-reduction algorithm of Graspan. "Pocr" denotes our approach with optimization (Section 5.2) enabled. The columns "Graspan" and "Soufflé" list the results of the open-source tools Graspan [Wang et al. 2020a] and Soufflé [Jordan et al. 2016a], running with their default configurations.[3] The cases which failed to obtain results due to the time constraint (24 hours) are annotated with "–".

6.2 RQ 1: Reduction of Redundant Derivations

The numbers of edges added to the graph and the numbers of edges created during CFL-reachabilty solving of each benchmark are listed in Column 1 and Columns 2–4 of Tables 2 and 3. The number of redundant derivations is obtained via (#Deriv - #Add). The proportion of redundant derivations is computed via (#Deriv - #Add) / #Deriv.

In general, with regard to the standard CFL-reachability algorithm (listed in Column 2 of Tables 2 and 3), redundant derivations are prevalent in both value-flow analysis and alias analysis. The average proportions for redundant derivations out of all derivations are 98.03% and 97.89%, respectively, in value-flow analysis and alias analysis when using the standard algorithm.

A comparison of Columns 3-4 with Column 2 in Tables 2 and 3 shows that both GSA (the edge-reduction technique of Graspan) and our Pocr reduce redundant derivations but Pocr is more effective. The reduction rates of the redundant derivations are listed in Columns 5 and 6 of the

[2] https://github.com/travitch/whole-program-llvm.

[3] We refer to [Wang et al. 2017] and https://souffle-lang.github.io/build#cmake-configuration-options for the default configurations of Graspan and Soufflé, respectively.

Table 1. Benchmark info. #Node and #Edge denote the number of nodes and edges in the initial graphs.

Bench.	Size/MB	SVFG		PEG		Description
		#Node	#Edge	#Node	#Edge	
xz	1.24	51666	65235	12425	26468	General data compression
nab	1.41	59253	76105	16261	34676	Molecular dynamics
leela	2.93	68250	92865	22186	49748	Monte Carlo tree search (Go)
x264	4.68	213943	347142	60956	136352	Video compression
cactus	5.88	563208	1026726	93557	212478	Physics: relativity
povray	7.38	555807	1059724	76405	174258	Ray tracing
imagick	13.68	601687	870107	119314	301846	Image manipulation
parest	16.20	325592	433217	117500	251436	Biomedical imaging
perlbench	18.69	1031348	2203010	156664	388564	Perl interpreter
omnetpp	21.81	703952	1897474	241916	509166	Discrete Event simulation

Table 2. Result of context-sensitive value-flow analysis. #Add/k and #Deriv/k denotes the number of edges added to the graph and created when solving CFL-reachability, measured in thousands. Reduction/% denotes the reduction rate of redundant derivations of GSA and Pocr. Time/s denotes the runtime of each approach, measured in seconds. The baselines of both Reduction/% and speedup are the columns "Base".

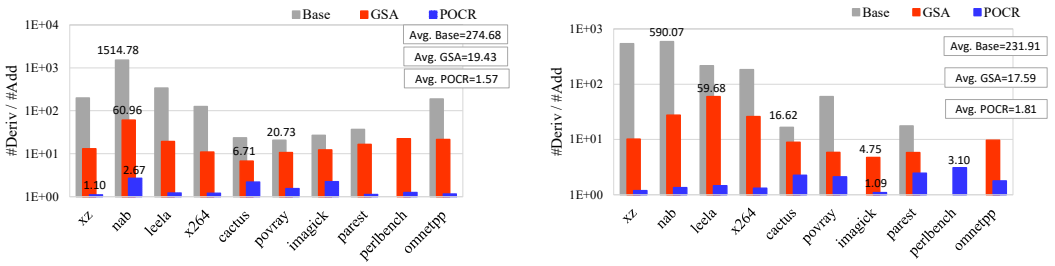
Bench.	#Add/k	#Deriv/k			Reduction/%		Time/s			
		Base	GSA	Pocr	GSA	Pocr	Base	Graspan	Soufflé	Pocr
xz	732	145140	9544	807	93.90	99.95	11.86	2.43	4.31	0.88
nab	1341	2031555	81757	3582	96.04	99.89	662.73	69.92	71.78	39.60
leela	1518	513825	29330	1846	94.57	99.94	30.22	6.62	12.58	2.04
x264	60441	7537094	657053	72668	92.02	99.84	4915.96	656.69	1495.35	299.29
cactus	105114	2465484	705243	229347	74.57	94.74	42121.28	6285.94	6714.28	1754.60
povray	182537	3783718	1943433	281611	51.10	97.25	77651.60	9149.52	12796.00	3893.08
imagick	55561	1493419	683175	123044	56.35	95.31	31210.60	3645.31	5634.66	1043.20
parest	29749	1090633	491326	33441	56.49	99.65	14419.60	3746.12	2265.00	433.20
perlbench	834251	-	18662423	1042244	-	-	-	63737.24	-	13962.88
omnetpp	262480	49378518	5648505	302842	89.03	99.92	58454.40	9630.63	5496.90	2367.20
Average					78.23	98.50			*Speedup*	21.48×

two tables, showing that, on average, Pocr reduces 98.50% and 97.26% of redundant derivations respectively for value-flow analysis and alias analysis, which is much more than GSA. We also observe that the redundant derivations of Pocr are only 4.67% and 9.68% that of GSA respectively for value-flow analysis and alias analysis. Namely, the redundant derivations of Pocr are much fewer than GSA. Interestingly, the set of redundant derivations captured by Pocr is not always a superset of those captured by Graspan because Graspan can also cover some non-transitive relations. However, we can infer from the experiment results that the redundancy caused by non-transitive relations has only a marginal impact on performance. Our approach, which exploits a proper derivation order based on the property of transitive relations, is much more effective in reducing redundant derivations than Graspan. Besides, we did not include the numbers of derivations of Soufflé in the tables because it does not provide an option to collect those numbers directly.

We show the computational redundancy of the three approaches in Figure 8, which is based on the numbers in Columns 1–4 of Tables 2 and 3. Specifically, the redundancy of each approach is represented by (#Deriv / #Add). The data shows how many derivations are needed to actually add an edge to the graph on average. From Figure 8(a) and Figure 8(b), we can see that there is a

Table 3. Result of field-sensitive alias analysis. The meanings of column headings are the same as Table 2.

Bench.	#Add/k	#Deriv/k			Reduction/%		Time/s			
		Base	GSA	Pocr	GSA	Pocr	Base	Graspan	Soufflé	Pocr
xz	427	230356	4332	507	98.30	99.97	2.50	1.71	2.04	0.24
nab	472	278736	13017	635	95.49	99.94	3.40	1.47	3.00	0.38
leela	8797	1904077	524949	12875	72.77	99.78	340.52	38.26	41.72	13.96
x264	13167	2413368	341681	17290	86.31	99.83	537.85	140.11	112.12	25.65
cactus	81832	1360390	727621	184821	49.49	91.94	11156.20	1479.63	1039.05	591.39
povray	53698	3214220	313985	112872	91.76	98.13	17601.30	1393.13	1238.86	631.86
imagick	422916	-	2007956	462378	-	-	-	5733.14	4076.94	1309.51
parest	83800	1472666	485888	205702	71.05	91.22	15337.50	943.52	1713.89	604.04
perlbench	1226586	-	24686471	3797919	-	-	-	29548.05	15536.13	5400.19
omnetpp	485066	-	4721535	866541	-	-	-	14660.59	11235.29	1842.43
Average					80.74	97.26			*Speedup*	19.57×



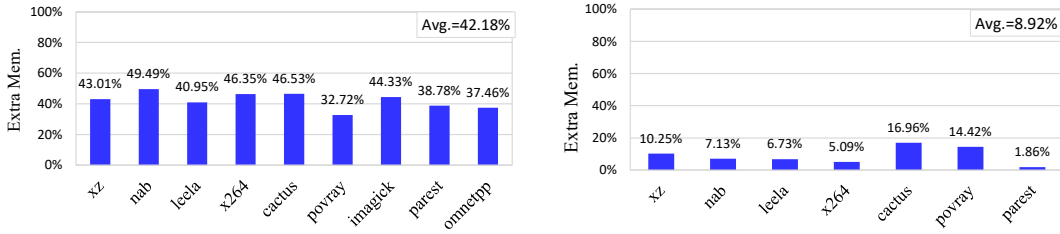
(a) Value-flow analysis. **(b)** Alias analysis.

Fig. 8. The computational redundancy of the three approaches in solving the two clients. The value is computed by (#Deriv / #Add). The vertical axis is logarithmic. The peak, valley and average values of each approach are marked in the charts.

substantial amount of redundancy in the standard algorithm in both value-flow analysis and alias analysis. The redundancy in the value-flow analysis is more significant as most of its reachability relations are transitive (i.e., the A-relations in the grammar of Figure 6).

A comparison between Pocr and GSA in Figure 8 demonstrates that: (1) Pocr is much more effective than GSA in reducing computational redundancy. The average values of the redundancy are only 1.57 and 1.81, respectively, in value-flow analysis and alias analysis, which is much smaller than that of GSA. (2) Even in the worst cases, Pocr keeps the computational redundancy in very low values, with the largest ones being only 1.54 and 3.10, respectively, in value-flow analysis and alias analysis. A comparison of Figure 8(a) and Figure 8(b) shows that the performance of Pocr is slightly better for the value-flow analysis than the alias analysis. This is because there are more non-transitive relations in alias analysis than those in value-flow analysis.

Another interesting observation is the significance of the grammar-based optimization in Pocr. We also run the original Pocr on the two clients and compare the results with the optimized ones in Tables 2 and 3. We find that compared with the original Pocr, the optimized Pocr further reduces 78.72% and 83.56% redundant derivations, respectively, for the two clients on average. Therefore, when handling real-world problems, establishing a proper optimization (e.g. Section 5.2) to exploit the benefit of spanning trees as much as possible is important to boost the performance of Pocr.



(a) Value-flow analysis. **(b)** Alias analysis.

Fig. 9. Extra memory overhead of Pocr over the standard algorithm. Only the benchmarks successfully solved by the standard algorithm are considered.

6.3 RQ 2: Speedups Over Baselines

Columns 7–10 of Tables 2 and 3 list the runtime of the four approaches for the two clients. We first focus on columns 7 and 10. A comparison between Base and Pocr shows that Pocr significantly accelerates the analyses for both value-flow analysis and alias analysis by eliminating transitive redundancy. The largest speedups of Pocr over the standard algorithm occur in `parest` and `povray`, which accords with relatively large reduction rates of computational redundancy in Figure 8. The largest speedups do not occur in the programs with the largest reduction rates of derivations because the preprocessing time (e.g., constructing graphs and performing graph simplification) is also included in the total runtime of Pocr. Such preprocessing time can take a large percent of runtime when handing small programs where the largest reduction rates of derivations occur.

By reducing a large portion of redundant derivations, Pocr successfully solves the value-flow analysis for `perlbench` and the alias analyses for `imagick`, `perlbench` and `omnetpp`, where the standard algorithm fails to solve within the time limit (24 hours). Therefore, taming redundant derivations plays an important role in scaling CFL-reachability analysis.

It is also interesting to note that the time consumed by an analysis depends not only on the program size but also on the features of the graph abstracted from the program. We found that `perlbench`, though not the largest graph, has a complex graph structure, including extremely long value-flow chains and dynamically/incrementally formed large transitive cycles. For example, in the PEG of `perlbench` after CFL solving, 83% of the total nodes are in cycles consisting of "A"-edges, with the largest cycle containing over 43k nodes and the longest simple path (without any cycles) containing 3k nodes, which are larger than those of other programs. This feature makes other CFL-reachability solvers incur much more transitive redundancy while solving, which makes `perlbench` arguably the most challenging program to solve.

To compare the analysis time, we further perform the experiments by running the open-source Graspan and Soufflé tools using their default configurations. Their results are listed in Columns 8 and 9 of Table 2 and Table 3. A comparison of Columns 8–10 of Tables 2 and 3 shows that Graspan and Soufflé can effectively accelerate the two clients. Moreover, Pocr is much more efficient. On average, Pocr is 3.67× and 4.10× faster than Graspan and Soufflé for value-flow analysis, and is 3.73× and 4.19× faster than Graspan and Soufflé for alias analysis, respectively.

As Pocr uses spanning trees to store transitive edges, we study the extra memory overhead of Pocr over the standard algorithm as shown in Figure 9. A comparison between Figure 9(a) and Figure 9(b) shows that the extra memory overheads of Pocr in value-flow analysis are much larger than those in alias analysis. This reflects the characteristics of the two clients: in value-flow analysis, most reachability relations are transitive relations (the A-relations in Figure 6). In alias

$$
\begin{aligned}
A &::= A\ a \mid A\ B \mid a \mid \varepsilon \\
B &::= CA_i\ ret_i \\
CA_i &::= call_i\ A
\end{aligned}
$$

(a) Modified grammar for context-sensitive value-flow analysis.

$$
\begin{aligned}
M &::= DV\ d \\
DV &::= \overline{d}\ V \\
V &::= \overline{A}\ V \mid V\ A \mid FV_i\ f_i \mid M \mid \varepsilon \\
FV_i &::= \overline{f_i}\ V \\
A &::= a\ M \mid a \mid \varepsilon \\
\overline{A} &::= M\ \overline{a} \mid \overline{a} \mid \varepsilon
\end{aligned}
$$

(b) Modified grammar for field-sensitive alias analysis.

Fig. 10. CFG modified from Figure 7 by removing the doubly recursive rules.

analysis, there is a smaller proportion of transitive relations (the A-relations in Figure 7), whereas the alias relations (the V-relations in Figure 7) are more prevalent.

6.4 RQ 3: Pocr vs. Grammar Rewriting

As addressed in Section 5.1, some grammars can be rewritten to eliminate transitive relations while maintaining correct solutions. The grammars in Figure 6 and Figure 7 of our example can be rewritten into Figure 10(a) and Figure 10(b) with the doubly-recursive rules $A ::= A\ A$ and $\overline{A} ::= \overline{A}\ \overline{A}$ removed. The modified grammar still has the ability to compute the required relations (i.e., A for value-flow analysis and V for alias analysis).

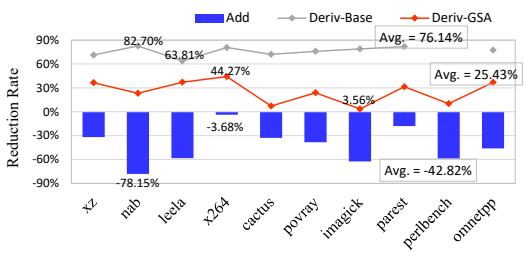
We study the performance impact of such grammar rewriting and compare it with Pocr. The experimental results of the modified grammars are shown in Figures 11 and 12, in which the reduction rate and speedup of each approach are computed based on the result obtained from the original grammar. Pocr is not taken into consideration as it does not benefit the modified grammar.

With respect to value-flow analysis (Figure 11), the values of the reduction rate of the added edges are negative. This is because the modified grammar in Figure 10(a) introduces an extra non-terminal B, leading to extra edges added to the graphs. However, the reduction rate of derivations of the standard algorithm is large because of the removed doubly-recursive rules. In contrast, the reduction rate of derivations of GSA is low because it has already reduced redundant derivations for the original grammar. Accordingly, the standard algorithm is accelerated much more than the other two techniques by grammar rewriting as shown in Figure 11(b). Regarding alias analysis (Figure 12), the total numbers of added edges are reduced because the modified grammar in Figure 10(b) removes doubly recursive rules and does not introduce any extra non-terminal. However, the reduction rates of derivations of both the standard algorithm and GSA are low. This confirms with the aforementioned characteristic that A- and $\overline{A}$-edges only take a small proportion in alias analysis. A comparison of Figure 11(b) and Figure 12(b) shows that the accelerations brought by modifying the grammar are much smaller for all three techniques in alias analysis.

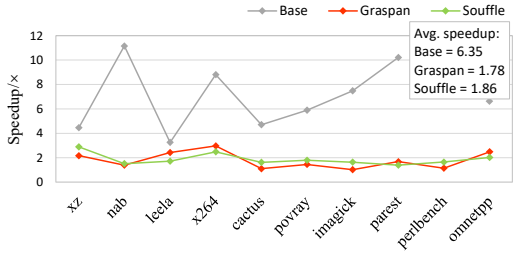
Additionally, it can be computed from Figure 11(b) and 12(b) that all three accelerated techniques through the modified grammars are still slower than Pocr. We analyze the reason as follows. Although the modified grammars appear to have the "head-to-tail" derivations illustrated in Section 2, both the standard worklist algorithm and GSA do not strictly follow this derivation order. Hence, Pocr is faster than the three techniques in the presence of grammar rewriting.

7 RELATED WORK

This work is relevant to improving the performance of CFL-reachability. Reducing the cubic time complexity of general CFL-reachability is difficult. To the best of our knowledge, the fastest

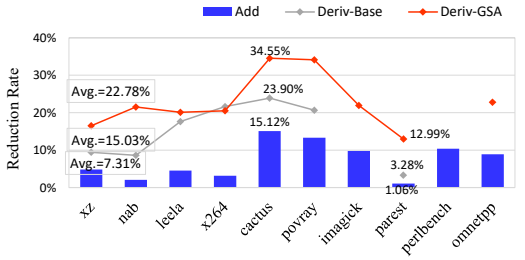


(a) Reduction rates of #Add and #Deriv.

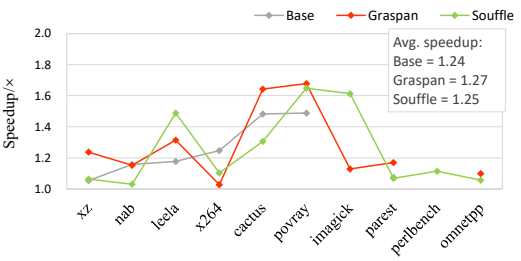


(b) Speedups comparing to the original grammar.

Fig. 11. Results of context-sensitive value-flow analysis using the modified grammar in Figure 10(a).



(a) Reduction rates of #Add and #Deriv.



(b) Speedups comparing to the original grammar.

Fig. 12. Results of field-sensitive alias analysis using the modified grammar in Figure 10(b).

algorithm [Chaudhuri 2008] treats CFL-reachability as an equivalent representation called recursive-state-machine reachability (RSM-reachability) and exhibits a slightly subcubic $O(n^3/\log^2 n)$ time complexity for bounded-stack RSMs, where n denotes the number of nodes of the input graph. So far, significant progress has only been made in handling special cases. One of the extensively studied relation in CFL-reachability is the Dyck relation [Chatterjee et al. 2018; Yuan and Eugster 2009; Zhang et al. 2013]. A recent work has reduced the complexity to nearly linear with respect to the input graph size [Chatterjee et al. 2018]. However, these techniques are only applicable to Dyck reachability on bidirected graphs. Recently, a specialized parallel data structure was proposed for scalable equivalence relations and provides a quadratic worst-case speedup [Nappa et al. 2019]. However, the transitive relations handled in this paper are not equivalence relations, thus their technique is not directly applicable.

Due to the difficulty in reducing time complexity, many existing approaches improve the performance of CFL-reachability from more practical perspectives. One prevalent perspective is graph simplification, including cycle elimination [Hardekopf and Lin 2007; Kodumal and Aiken 2004; Lei and Sui 2019; Pereira and Berlin 2009; Su et al. 2014; Xu et al. 2009], variable substitution [Rountev and Chandra 2000], InterDyck edge reduction [Li et al. 2020] and other techniques [Aho et al. 1972; Hsu 1975; Moyles and Thompson 1969]. It is interesting to point out that the InterDyck algorithm [Li et al. 2020] can improve the precision of the undecidable interleaved-Dyck problems [Reps 2000] by removing the non-Dyck-contributing edges, whereas Pocr focuses on handling standard CFL-reachability, which is decidable and can be solved precisely in polynomial time. Notably, all the aforementioned offline graph simplification techniques are orthogonal to our online CFL-reachability solver Pocr. Hence they are well-compatible to further improve scalability.

Recently, researchers have also studied the simplification of CFL-reachability from the perspective of its alternative forms, such as recursive state machines (RSMs) [Alur et al. 2005; Chaudhuri 2008] and pushdown automata (PDA) [Gauwin et al. 2019; Heizmann et al. 2017]. Different from graph simplification, these works tried to scale CFL-reachability by simplifying the automata (i.e., the context-free grammar) rather than the input graphs.

Reducing edge redundancy on-the-fly has also been incorporated into many existing techniques. Fähndrich et al. [1998] and Su et al. [2000] arbitrarily ordered the computations based on node indices in order to reduce repeated computations. A recent technique Graspan [Wang et al. 2017] use auxiliary structures to classify "new" and "old" edges to avoid the need to recompute the established edges. Our approach, different from the above techniques, exploits an effective edge derivation order based on the property of transitive relations so that almost all edge redundancy caused by singly or doubly recursive rules is avoided.

8 CONCLUSION

This paper has proposed POCR, a fast yet precision-preserving approach to CFL-reachability analysis by taming its transitive redundancy. The empirical results show that POCR significantly accelerates the solving of CFL-reachability problems where transitive redundancy dominates. POCR is over 3.67× faster over recent open-source tools Graspan and Soufflé for both clients. POCR obtains speedups of 21.48× and 19.57× over the standard algorithm by eliminating 98.50% and 97.26% redundant derivations for value-flow analysis and alias analysis, respectively.

DATA AVAILABILITY STATEMENT

Materials for our evaluation are publicly available [Lei et al. 2022] and can be used to reproduce the data of our experiment. The code is sourced from the project https://github.com/kisslune/POCR.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for valuable feedback on earlier drafts of this paper, which helped improve its presentation. We thank Camille Bossut for helpful comments. This research is supported by Australian Research Grants DP200101328 and DP210101348; by Amazon under an Amazon Research Award in automated reasoning; by the United States National Science Foundation (NSF) under grants No. 1917924 and No. 2114627; and by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-21-C-4024. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

REFERENCES

Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. 1972. The transitive reduction of a directed graph. *SIAM J. Comput.* 1, 2 (1972), 131–137. https://doi.org/10.1137/0201008

Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. 2005. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 4 (2005), 786–818. https://doi.org/10.1007/3-540-44585-4_18

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 243–262. https://doi.org/10.1145/1640089.1640108

Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2018. Optimal Dyck reachability for data-dependence and alias analysis. *Proc. ACM Program. Lang.* 2, POPL (2018), 30:1–30:30. https://doi.org/10.48550/arXiv.1910.00241

Swarat Chaudhuri. 2008. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 159–169. https://doi.org/10.1145/1328897.1328460

Manuel Fähndrich, Jeffrey S Foster, Zhendong Su, and Alexander Aiken. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 85–96. https://doi.org/10.1145/277652.277667

Olivier Gauwin, Anca Muscholl, and Michael Raskin. 2019. Minimization of visibly pushdown automata is NP-complete. *arXiv preprint arXiv:1907.09563* (2019). https://doi.org/10.48550/arXiv.1907.09563

Ben Hardekopf and Calvin Lin. 2007. 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*. 290–299. https://doi.org/10.1145/1273442.1250767

Matthias Heizmann, Christian Schilling, and Daniel Tischner. 2017. Minimization of visibly pushdown automata using partial Max-SAT. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 461–478. https://doi.org/10.1007/978-3-662-54577-5_27

Harry T Hsu. 1975. An algorithm for finding a minimal equivalent graph of a digraph. *Journal of the ACM (JACM)* 22, 1 (1975), 11–16. https://doi.org/10.1145/321864.321866

Giuseppe F. Italiano. 1986. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science* 48 (1986), 273–281. https://doi.org/10.1016/0304-3975(86)90098-8

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016a. Soufflé. https://github.com/souffle-lang/souffle

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016b. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*. Springer, 422–430. https://doi.org/10.1007/978-3-319-41540-6_23

John Kodumal and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. *ACM Sigplan Notices* 39, 6 (2004), 207–218. https://doi.org/10.1145/996893.996867

Yuxiang Lei and Yulei Sui. 2019. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In *International Static Analysis Symposium*. Springer, 27–47. https://doi.org/10.1007/978-3-030-32304-2_3

Yuxiang Lei, Yulei Sui, Shuo Ding, and Qirun Zhang. 2022. Artifact of "Taming Transitive Redundancy for Context-Free Language Reachability". https://doi.org/10.5281/zenodo.7066401

Yuanbo Li, Qirun Zhang, and Thomas Reps. 2020. Fast graph simplification for interleaved Dyck-reachability. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 780–793. https://doi.org/10.1145/3492428

David Melski and Thomas Reps. 2000. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science* 248, 1-2 (2000), 29–98. https://doi.org/10.1016/S0304-3975(00)00049-9

Dennis M Moyles and Gerald L Thompson. 1969. An algorithm for finding a minimum equivalent graph of a digraph. *Journal of the ACM (JACM)* 16, 3 (1969), 455–460. https://doi.org/10.1145/321526.321534

Nomair A Naeem and Ondrej Lhoták. 2008. Typestate-like analysis of multiple interacting objects. *ACM Sigplan Notices* 43, 10 (2008), 347–366. https://doi.org/10.1145/1449955.1449792

Patrick Nappa, David Zhao, Pavle Subotić, and Bernhard Scholz. 2019. Fast parallel equivalence relations in a datalog compiler. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 82–96. https://doi.org/10.1109/PACT.2019.00015

Esko Nuutila and Eljas Soisalon-Soininen. 1994. On finding the strongly connected components in a directed graph. *Inform. Process. Lett.* 49, 1 (1994), 9–14. https://doi.org/10.1016/0020-0190(94)90047-7

Fernando Magno Quintao Pereira and Daniel Berlin. 2009. Wave propagation and deep propagation for pointer analysis. In *2009 International Symposium on Code Generation and Optimization*. IEEE, 126–135. https://doi.org/10.1109/CGO.2009.9

Jakob Rehof and Manuel Fähndrich. 2001. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. *ACM SIGPLAN Notices* 36, 3 (2001), 54–66. https://doi.org/10.1145/360204.360208

Thomas Reps. 1995. Shape analysis as a generalized path problem. In *Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. 1–11. https://doi.org/10.1145/215465.215466

Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 162–186. https://doi.org/10.1145/345099.345137

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61. https://doi.org/10.1145/199448.199462

Atanas Rountev and Satish Chandra. 2000. Off-line variable substitution for scaling points-to analysis. *Acm Sigplan Notices* 35, 5 (2000), 47–56. https://doi.org/10.1145/358438.349310

Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel pointer analysis with CFL-reachability. In *2014 43rd International Conference on Parallel Processing*. IEEE, 451–460. https://doi.org/10.1109/ICPP.2014.54

Zhendong Su, Manuel Fähndrich, and Alexander Aiken. 2000. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 81–95. https://doi.org/10.1145/325694.325706

Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *CC '16*. 265–266. https://doi.org/10.1145/2892208.2892235

Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122. https://doi.org/10.1109/TSE.2014.2302311

Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160. https://doi.org/10.1109/SWAT.1971.10

Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020b. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *42nd International Conference on Software Engineering*. ACM. https://doi.org/10.1145/3377811.3380386

Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 389–404. https://doi.org/10.1145/3093336.3037744

Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2020a. Graspan-cpp. https://github.com/Graspan/graspan-cpp

Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *European Conference on Object-Oriented Programming*. Springer, 98–122. https://doi.org/10.1007/978-3-642-03013-0_6

Hao Yuan and Patrick Eugster. 2009. An efficient algorithm for solving the dyck-cfl reachability problem on trees. In *European Symposium on Programming*. Springer, 175–189. https://doi.org/10.1007/978-3-642-00590-9_13

Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. 2013. 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*. 435–446. https://doi.org/10.1145/2491956.2462159

Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 197–208. https://doi.org/10.1145/1328897.1328464