PEARL: A Multi-Derivation Approach to Efficient CFL-Reachability Solving

Chenghang Shi, Haofeng Li, Yulei Sui, Jie Lu, Lian Li, and Jingling Xue

Abstract—Context-free language (CFL) reachability is a fundamental framework for formulating program analyses. CFL-reachability analysis works on top of an edge-labeled graph by deriving reachability relations and adding them as labeled edges to the graph. Existing CFL-reachability algorithms typically adopt a single-reachability relation derivation (SRD) strategy, i.e., one reachability relation is derived at a time. Unfortunately, this strategy can lead to redundancy, hindering the efficiency of the analysis.
To address this problem, this paper proposes PEARL, a *multi-derivation* approach that reduces derivation redundancy for CFL-reachability solving, which significantly improves the efficiency of CFL-reachability analysis. Our key insight is that multiple edges can be simultaneously derived via batch propagation of reachability relations. We also tailor our multi-derivation approach to tackle transitive relations that frequently arise when solving CFL-reachability. Specifically, we present a highly efficient transitive-aware variant, PEARL^{PG}, which enhances PEARL with *propagation graphs*, a lightweight but effective graph representation, to further diminish redundant derivations. We evaluate the performance of our approach on two clients, i.e., context-sensitive value-flow analysis and field-sensitive alias analysis for C/C++. By eliminating a large amount of redundancy, our approach outperforms two baselines including the standard CFL-reachability algorithm and a state-of-the-art solver POCR specialized for fast transitivity solving. In particular, the empirical results demonstrate that, for value-flow analysis and alias analysis respectively, PEARL^{PG} runs 3.09× faster on average (up to 4.44×) and 2.25× faster on average (up to 3.31×) than POCR, while also consuming less memory.

Index Terms—Program analysis, CFL-reachability, constraint solving, transitive relations

1 INTRODUCTION

Many program analysis problems, such as interprocedural data flow analysis [1], [2], program slicing [3], [4], [5], shape analysis [6], and pointer analysis [7], [8], [9], [10], [11], [12], [13], [14] can be formulated as context-free language (CFL) reachability problems [15]. The CFL-reachability problem extends standard graph reachability to an edge-labeled graph, where Node v is CFL-reachable from Node u if there exists a path from u to v whose edge labels follow a predefined context-free grammar (CFG). Despite its wide applicability, it is well-known that the CFL-reachability solving algorithm has a (sub)cubic time complexity with respect to the number of nodes in the edge-labeled graph [16]. Researchers have developed different performance optimization techniques, including simplifying the graph size via pre-processing [17], [18], [19], applying summary-based techniques for caching [1], [3], [20], and adopting efficient data processing techniques to improve scalability [21], [22]. However, despite all these efforts, CFL-reachability algorithms can still suffer from significant performance loss due to derivation redundancy.

During CFL-reachability solving, an X-reachability relation between source node u and sink node v (i.e., v is X-reachable from u) is explicitly represented as an X-edge $u \xrightarrow{X} v$ in the edge-labeled graph. We use the terms X-edge and X-reachability relation interchangeably, and they are both denoted as $u \xrightarrow{X} v$. Essentially, the edge derivation process of CFL-reachability can be viewed as propagating reachability relations along the edge-labeled graph until a fixed point is reached, i.e., no more new reachability relations can be deduced. During propagation, each newly derived reachability relation (a source-to-sink path) is summarized as a labeled edge and added to the graph, making this reachability relation explicit.

Derivation Redundancy. Existing CFL-reachability algorithms commonly employ a single-reachability relation derivation (SRD or single-derivation) strategy, i.e., one reachability relation is derived at a time. However, this straightforward strategy can lead to considerable derivation redundancy, hindering the efficiency of the analysis. Moreover, the presence of transitive relations (production rule in the form $A := A\ A$) can introduce additional derivation redundancies [23], further impacting overall efficiency.

Our Solution. To address this problem, we propose PEARL, a *multi-derivation* approach to CFL-reachability analysis. Our key insight is that multiple edges can be simultaneously derived via batch propagation of reachability relations, thus significantly reducing derivation redundancy caused by the single-derivation strategy in existing algorithms. In addition, we introduce a simple yet effective graph representation named *propagation graph*, to concisely encode transitive relations which are prevalent in CFL-based program analyses. Propagation graphs can be combined with multi-derivation to further diminish redundant derivations due to transitivity (also known as transitive redundancy [23]), and consequently, boost the performance of

• *Chenghang Shi, Haofeng Li, Jie Lu, and Lian Li are with SKLP, Institute of Computing Technology, CAS, China. Yulei Sui and Jingling Xue are with University of New South Wales, Australia. Chenghang Shi and Lian Li are also with University of Chinese Academy of Sciences, China. Lian Li is also with Zhongguancun Laboratory, China. Lian Li and Haofeng Li are the corresponding authors.*
E-mail:{shichenghang21s,lihaofeng,lujie,lianli}@ict.ac.cn, {y.sui,j.xue}@unsw.edu.au.

CFL-reachability solving. This combination leads to a highly efficient transitive-aware variant, PEARL^{PG}. It is worth noting that a previous work, POCR [23], proposes to eliminate transitive redundancies by keeping track of the derivation order for each transitive relation, at the expense of introducing a spanning tree model. However, this optimization does not fit our multi-derivation approach and thus is not adopted in our implementation, as discussed in Section 3.2.

We have evaluated PEARL and PEARL^{PG} using two popular static analysis clients, context-sensitive value-flow analysis [24], [25] and field-sensitive alias analysis for C/C++ [9]. Experimental results demonstrate that our method significantly outperforms existing approaches. When compared with POCR [23], a state-of-the-art solver with effective optimizations for transitive redundancies, PEARL^{PG} runs 3.09× (up to 4.44×) faster for value-flow analysis and 2.25× (up to 3.31×) faster for alias analysis over POCR with less consumed memory.

To summarize, this paper makes the following contributions:

- We propose a multi-derivation approach that utilizes a batch propagation technique for the rapid derivation of reachability relations. Our approach eliminates repetitive derivations introduced by the single-deriviation strategy in existing CFL-reachability algorithms, thereby enhancing the efficiency of CFL-reachability analysis.
- We introduce *propagation graph*, a lightweight but effective graph representation for transitive relations. Propagation graphs are transitivity-aware subgraphs induced from the original edge-labeled graph on the fly. We have developed efficient algorithms that combine multi-derivation with propagation graphs to effectively reduce redundant derivations caused by transitive relations, which are common in CFL-based program analyses. The approach demonstrates promising performance improvements.
- We apply our technique to two popular CFL-based program analyses: context-sensitive value-flow analysis and field-sensitive alias analysis. Experimental results demonstrate that our approach effectively eliminates a large percentage of derivation redundancy, significantly boosting the performance of CFL-reachability analysis.

The remainder of this paper is structured as follows. Section 2 introduces the background. Section 3 briefly illustrates the core idea of our approach with two motivating examples. We detail our approach in Section 4 and evaluate our tools PEARL and PEARL^{PG} in Section 5. Section 6 surveys related work and Section 7 concludes this paper.

Major Extensions. This paper is an extension of the conference paper [26] published in ASE'23. We have made the following major extensions.

- The multi-derivation algorithm presented in [26] was initially designed to handle productions related to transitivity. In this extension, we have expanded its scope to support general production rules. Consequently, Section 3 and Section 4 have been completely rewritten to introduce a new multi-derivation algorithm for CFL-reachability analysis.
- We reformulate our multi-derivation approach from a constraint-solving perspective (Section 4.1.1) with formally defined rules (Table 1).
- We introduce a practical optimization technique for a specific production pattern in the form of $X ::= a\ Z\ b$, where a and b are terminals. Such kind of production is present in all existing CFL-based analyses (Section 4.1.3). We have successfully adapted this optimization to both the standard algorithm and our multi-derivation algorithm.
- We have added significantly more new empirical experiments to help understand the practical benefits of multi-derivation.

2 BACKGROUND

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

2.1 CFL-reachbility

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

2.2 The Standard Algorithm

In the literature, CFL-reachability is solved by the standard dynamic programming algorithm [27] given in Algorithm 1. The algorithm requires the CFG to be normalized in such a way that the right-hand side of each production has at most two symbols, i.e., productions are in the form $X ::= Y\ Z$, $X ::= Y$ or $X ::= \varepsilon$. Additionally, we introduced an optimization to directly process Dyck-style productions $X ::= a\ Y\ b$ without normalization. This optimizatin is highlighted in Lines 26-29 and its effectiveness will be further discussed in Section 4.1.3. Let W denote a worklist. Algorithm 1 first initializes the worklist with all original edges in the input graph (Line 5) and then adds all self-referencing edges ($v \xrightarrow{X} v$) produced by empty productions $X ::= \varepsilon$ into the graph and worklist (Lines 6-9).

Next, the procedure `Solve` is invoked to iteratively derive new edges until no more edges can be deduced ($W = \emptyset$). Given an edge $u \xrightarrow{Y} v$, the four loops in procedure

`HandleItem` at Line 15, Line 18, Line 22, and Line 26 introduce new edges according to the four distinct forms of production rules $X ::= Y$, $X ::= Y\ Z$, $X ::= Z\ Y$, and $X ::= a\ Y\ b$, respectively. Specifically, each edge $u \xrightarrow{Y} v$ induces an edge $u \xrightarrow{X} v$ by the production rule $X := Y$ (Lines 15 - 17). In addition, each outgoing Z-edge of Node v (Lines 18 - 21) and incoming Z-edge (Lines 22 - 25) of Node u is examined to derive new X edges via the productions $X ::= Y\ Z$ and $X ::= Z\ Y$, respectively. At last, each pair of incoming a-edge of Node u and outgoing b-edge of Node v derives a new X edge according to the production $X ::= a\ Y\ b$ (Lines 26-29). All newly derived edges are added to the graph and the worklist for further processing.

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

2.3 Graph Representation

In CFL-reachability, edge-labeled graphs are typically realized by storing labeled edges in adjacency lists [16]. Given an X-edge $u \xrightarrow{X} v$, we say Node u is an X-predecessor of Node v, and Node v is an X-successor of Node u. Consequently, the *X-predecessor set* of Node v, denoted as $R(X, v) = \{\ u \mid u \xrightarrow{X} v \in E\ \}$, represents all incoming X-edges of Node v (used at Line 23 in Algorithm 1), and the *X-successor set* of Node v, denoted as $S(X, v) = \{\ u \mid v \xrightarrow{X} u \in E\ \}$, represents all outgoing X-edges of Node v (used at Line 19 in Algorithm 1). Specifically, the X-predecessor set of Node v is also called the *X-reachability relation set* of Node v.

In practice, the predecessor/successor sets can be implemented by hash tables (e.g., `std::unordered_set` in C++ *standard template library*) with $O(1)$ amortized time complexity for insertion and lookup operations [28].

2.4 Transitive Production Rules

This subsection provides definitions related to transitivity. Transitive relations (manifest in fully and partially transitive productions) are ubiquitous in CFL-based analyses, often introduced to model data flow or control flows.

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

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

Relation X is a *partially transitive relation* if and only if it is on the left side of a partially transitive production. Accordingly, edges of fully (partially) transitive relations are called *fully (partially) transitive edges*. We classify fully transitive edges into two categories [23] by the *first* production that generates it:

- *Secondary edge*. A fully transitive edge is a secondary edge if it is first derived via a fully transitive production;

Algorithm 1: The standard CFL-reachability algorithm

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

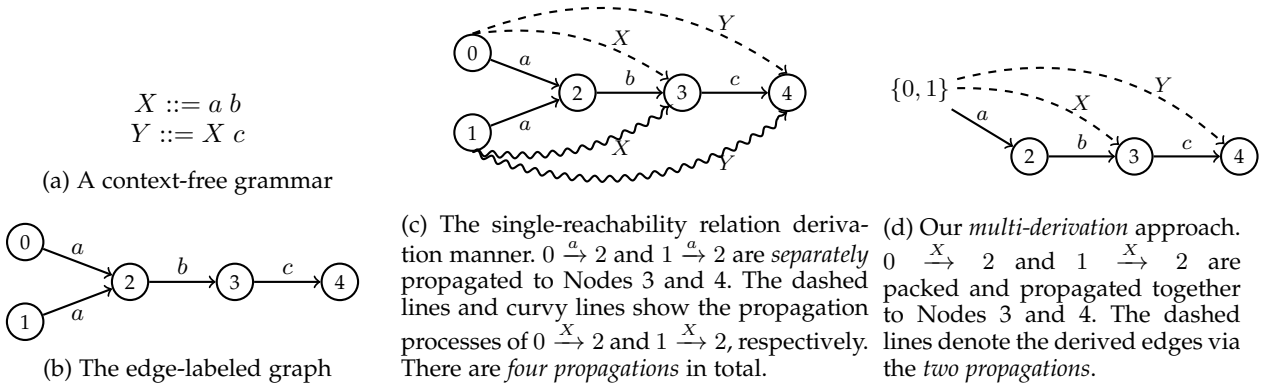
```
1  Function StdCFL(P, G):
2      Init();
3      Solve(P, G);
4  Procedure Init():
5      add E to W;
6      for each production X ::= ε ∈ P do
7          for each node v ∈ V do
8              if v --X--> v ∉ E then
9                  add v --X--> v to E and W;
10 Procedure Solve(P, G):
11     while W ≠ ∅ do
12         pop an edge u --Y--> v from W;
13         HandleItem(P, G, u --Y--> v);
14 Procedure HandleItem(P, G, u --Y--> v):
15     for each production X ::= Y ∈ P do
16         if u --X--> v ∉ E then
17             add u --X--> v to E and W;
18     for each production X ::= YZ ∈ P do
19         for outgoing edge v --Z--> w from node v do
20             if u --X--> w ∉ E then
21                 add u --X--> w to E and W;
22     for each production X ::= ZY ∈ P do
23         for incoming edge w --Z--> u to node u do
24             if w --X--> v ∉ E then
25                 add w --X--> v to E and W;
26     for each production X ::= aY b ∈ P do
27         for each pair of edges w --a--> u and v --b--> w' do
28             if w --X--> w' ∉ E then
29                 add w --X--> w' to E and W;
```

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

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

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

Transitive production rules have a nice property, e.g., production $X ::= XA$ suggests that X-reachability relations



(a) A context-free grammar

(b) The edge-labeled graph

(c) The single-reachability relation derivation manner. $0 \xrightarrow{a} 2$ and $1 \xrightarrow{a} 2$ are *separately* propagated to Nodes 3 and 4. The dashed lines and curvy lines show the propagation processes of $0 \xrightarrow{X} 2$ and $1 \xrightarrow{X} 2$, respectively. There are *four propagations* in total.

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

Fig. 1: A motivating example to show the derivation redundancy due to the single-reachability derivation style.

can be propagated via A-edges while preserving their edge label X.

3 PEARL IN A NUTSHELL

In this section, we briefly illustrate our *multi-derivation* approach with a motivating example. Then we show how to optimize CFL-reachability solving for transitive relations that frequently arise in CFL-based program analyses, by equipping multi-derivation with *propagation graph* representation.

3.1 The Multi-derivation Approach

Figure 1a gives an example context-free grammar (CFG), consisting of the two productions $X ::= a\ b$ and $Y ::= X\ c$. The input edge-labeled graph G, shown in Figure 1b, contains four edges and five nodes. In CFL-reachability solving, new reachability relations (edges) are derived via the propagation of old reachability relations (edges) until a fixed point. For instance, the production $X ::= a\ b$ suggests that an a-edge $u \xrightarrow{a} v$ and a b-edge $v \xrightarrow{b} w$ together can derive an X-edge $u \xrightarrow{X} w$. From another perspective, by propagating the relation $u \xrightarrow{a} v$ along the edge $v \xrightarrow{b} w$, we obtain a new relation $u \xrightarrow{X} w$.

Single-derivation. The single-reachability derivation (SRD) strategy, as exemplified in Algorithm 1, is commonly employed in existing approaches. This strategy deduces only one reachability relation in one propagation step. For instance, given the production $X ::= a\ b$ and the two edges $0 \xrightarrow{a} 2$ and $2 \xrightarrow{b} 3$, a new relation $0 \xrightarrow{X} 3$ will be derived and introduced into the graph. Moreover, since we have $Y ::= X\ c$, propagating the newly generated relation $0 \xrightarrow{X} 3$ along the edge $3 \xrightarrow{c} 4$ results in another new relation $0 \xrightarrow{Y} 4$. The above two newly derived relations, $0 \xrightarrow{X} 3$ and $0 \xrightarrow{Y} 4$, are summarized as dashed lines in Figure 1c. Similarly, the propagation of $1 \xrightarrow{a} 2$ from Node 2 to Node 4 generates the two relations $1 \xrightarrow{X} 3$ and $1 \xrightarrow{Y} 4$, as highlighted by the curvy lines in Figure 1c.

Multi-derivation. In the single-derivation strategy, the two a-reachability relations ($0 \xrightarrow{a} 2$ and $1 \xrightarrow{a} 2$) are separately propagated. This separate propagation process introduces *derivation redundancy*, hindering the efficiency of CFL-reachablity solving. To address this problem and diminish derivation redundancy, we propose a multi-derivation approach – instead of propagating reachability relations separately, we propagate them in batches.

Figure 1d depicts our multi-derivation strategy. The two reachability relations, $0 \xrightarrow{a} 2$ and $1 \xrightarrow{a} 2$, are packed together as one set $\{0, 1\} \xrightarrow{a} 2$. Then the set of a-reachability relations are propagated along the edge $2 \xrightarrow{b} 3$, resulting in the set of two new X-reachability relations ($\{0, 1\} \xrightarrow{X} 3$), produced by the rule $X ::= a\ b$. Next, the set $\{0, 1\} \xrightarrow{X} 3$ is further propagated along c-edge $3 \xrightarrow{c} 4$. As a result, multiple new Y-reachability relations ($\{0, 1\} \xrightarrow{Y} 4$) are derived simulatenously by the production $Y ::= X\ c$.

Compared to the standard single-derivation strategy, our multi-derivation approach can significantly reduce the number of reachability propagations, thereby boosting the performance of CFL-reachability algorithms. In this example, while the standard SRD approach necessitates four propagations to derive all the X- and Y-reachability relations, our multi-derivation approach accomplish the same task in just two propagations.

3.2 Tailoring Multi-derivation for Transitivity

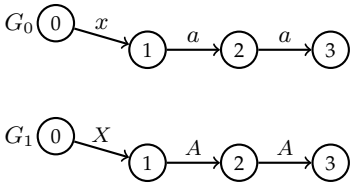
Transitive relations, commonly utilized to represent control and data flows, are prevalent in CFL-based program analyses. However, these relations can lead to redundant propagation, which diminishes the performance of the CFL-reachability solving. Consider the CFG in Figure 2a, where $A ::= A\ A$ is a fully transitive production (Definition 1) and $X ::= X\ A$ is a partially transitive production (Definition 2). In Figure 2b, G_0 is the input graph, while G_1 results from applying the productions $X ::= x$ and $A ::= a$ to G_0. In Figure 2c, the two A-edges $1 \xrightarrow{A} 2$ and $2 \xrightarrow{A} 3$ produces the secondary edge $1 \xrightarrow{A} 3$ (highlighted with a curvy line) based on the fully transitive production $A ::= A\ A$. Subsequently, the X-reachability relation $0 \xrightarrow{X} 1$ is propagated *twice* from Node 1 to Node 3: once via the path $1 \xrightarrow{A} 2 \xrightarrow{A} 3$, and again via the edge $1 \xrightarrow{A} 3$ (the curvy line). Consequently, the edge $0 \xrightarrow{X} 3$ is derived twice, resulting in redundant derivation. Such redundancy is also called *transitive redundancy* [23].

Transitivity-aware Propagation Graph. To reduce transitive redundancy, we introduce a simple yet effective graph

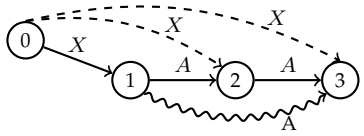
$A ::= AA \mid a$

$X ::= XA \mid x$

(a) A context-free grammar



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



(c) Redundant propagation due to transitivity. The X-reachability relations of Node 1 (e.g., $0 \xrightarrow{X} 1$) are propagated from Node 1 to Node 3 twice. One is via the path $1 \xrightarrow{A} 2 \xrightarrow{A} 3$, and the other is via the "shortcut" edge $1 \xrightarrow{A} 3$ (the curvy line).

Fig. 2: An example to demonstrate the redundant propagation due to transitivity.

Rule Type	Production	Edge(s)	New Edge	Constraints
Terminal		$u \xrightarrow{a} v$		$u \in R(a, v)$
ε	$X ::= \varepsilon$		$\forall v \in V, v \xrightarrow{X} v$	$\forall v \in V, v \in R(X, v)$
Unary	$X ::= Y$	$u \xrightarrow{Y} v$	$u \xrightarrow{X} v$	$\forall v \in V, R(Y, v) \subseteq R(X, v)$
Binary	$X ::= Y\ Z$	$u \xrightarrow{Y} v \wedge v \xrightarrow{Z} w$	$u \xrightarrow{X} w$	$\forall u \xrightarrow{Z} v \in E, R(Y, u) \subseteq R(X, v)$
Dyck	$X ::= a\ Y\ b$	$w \xrightarrow{a} u \wedge u \xrightarrow{Y} v \wedge v \xrightarrow{b} w'$	$w \xrightarrow{X} w'$	$\forall u \xrightarrow{Y} v \in E \wedge v \xrightarrow{b} w \in E, R(a, u) \subseteq R(X, w)$

TABLE 1: Constraint rules. The first rule performs initialization for terminal edges, while the last four specify constraints for ε-production, unary production, binary production, and Dyck-style production, respectively, where $X \in N$, $Y, Z \in (\Sigma \cup N)$, and $a, b \in \Sigma$. V and E denote the vertex set and edge set of the edge-labeled graph, respectively. $R(X, v)$ denotes the X-predecessor set of Node v (Section 2).

representation called *propagation graph* (Definition 4). For each transitive relation A, a propagation graph $PG(A)$ is constructed by selecting primary edges ($1 \xrightarrow{A} 2$ and $2 \xrightarrow{A} 3$) from the original edge-labeled graph. $PG(A)$ is transitive-aware since fully transitive relations are implicitly encoded in $PG(A)$. Partially transitive relations (e.g., $X ::= X\ A$) are solved by batchly propagating X-reachability relations in $PG(A)$. Note that the construction of propagation graphs is non-trivial because secondary edges may be derived by productions other than $A ::= A\ A$. Section 4.2 discusses this in detail.

The POCR Approach. A recent solver POCR [23] eliminates redundant propagation for transitivity by utilizing a spanning tree model. Specifically, given the graph G_1 in Figure 2b, for A-reachability relation, three spanning trees are constructed:

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

Given the reachability relation $0 \xrightarrow{X} 1$, the spanning tree rooted from Node 1 is traversed to derive two reachability relations, $0 \xrightarrow{X} 2$ and $0 \xrightarrow{X} 3$.

There are two reasons why we do not choose the spanning tree as the underlying representation of our approach. First, the A-transitivity information of each node is maintained *separately* in distinct spanning trees, and hence X-reachability relations of different root nodes can not be packed together for efficient multi-derivation. Second, the reachability information is redundantly preserved, e.g., $2 \xrightarrow{A} 3$ is copied into at least two spanning trees mentioned above, which can be expensive in terms of time and space when A-reachability relations are dense, as confirmed in our experiments (Section 5). Lastly, it is worth mentioning that POCR still adopts the single-reachability derivation style like the standard algorithm. We refer to the original paper [23] for readers who are interested in more details of POCR.

4 METHODOLOGY

This section formulates our multi-derivation approach as a constraint-solving framework to solve CFL-reachability (Section 4.1). In Section 4.2, we introduce a simple yet effective *transitivity-aware* graph representation called *propagation graph* and propose an efficient algorithm for constructing propagation graphs during on-the-fly reachability solving. We then demonstrate how to enhance multi-derivation with propagation graph representation to tackle transitive relations, which are ubiquitous in CFL-based program analyses.

4.1 Multi-derivation via Constraint Solving

We implement our multi-derivation approach as a constraint-solving framework for CFL-reachability. Essentially, the constraints specified by the productions and labeled edges are solved by *batch propagation* of reachability relations, thereby reducing the derivation redundancy due to the single-reachability derivation manner in the standard algorithm.

4.1.1 A Constraint-Solving Perspective

In the literature, the context-free grammar (CFG) is first translated into the normalized form to fit the standard algorithm (Algorithm 1). After normalization, a CFG contains

three forms of production: binary production $X ::= YZ$, unary production $X ::= Y$, ε-production $X ::= \varepsilon$. Additionally, in this paper, Dyck-style productions in the form of $X ::= a\ Y\ b$ are not normalized to enable our optimization technique, which is detailed in Section 4.1.3. Without loss of generality, we discuss our approach based on these kinds of productions.

Columns 2-4 of Table 1 demonstrate how the standard algorithm generates new edges, where $X \in N, Y, Z \in (\Sigma \cup N)$, and $a, b \in \Sigma$:

- Regarding an ε-production $X ::= \varepsilon$, an edge $v \xrightarrow{X} v$ is generated for each node $v \in V$.
- Given a unary production $X ::= Y$, an edge $u \xrightarrow{X} v$ is derived for each $u \xrightarrow{Y} v \in E$.
- Given a binary production $X ::= Y\ Z$, an edge $u \xrightarrow{X} v$ is deduced if there exist two edges $u \xrightarrow{Y} w$ and $w \xrightarrow{Z} v$.
- For a Dyck-style production $X ::= a\ Y\ b$, an edge $w \xrightarrow{X} w'$ is produced if there exists three consecutive edges $w \xrightarrow{a} u$, $u \xrightarrow{Y} v$ and $v \xrightarrow{b} w'$.

The standard algorithm (Algorithm 1) iteratively produces new edges according to the above four rules, until a fixed point is reached where no more new edges can be introduced. The solution to the underlying CFL-reachability problem is then obtained.

In this work, we formalize the CFL-reachability problem as a new set constraint problem, and Table 1 defines the five constraint rules as follows.

- For a terminal edge $u \xrightarrow{a} v$ in the input graph, we have $u \in R(a, v)$.
- An ε-production $X ::= \varepsilon$ indicates that, for any node $v \in V$, we have $v \in R(X, v)$.
- A unary production $X ::= Y$ suggests that, for any node $v \in V$, we have $R(Y, v) \subseteq R(X, v)$.
- Regarding a binary production $X ::= YZ$, a Z-edge $u \xrightarrow{Z} v$ produces the constraint $R(Y, u) \subseteq R(X, v)$.
- Given a Dyck-style production $X ::= a\ Y\ b$, two edges $u \xrightarrow{Y} v$ and $v \xrightarrow{b} w$ collectively generate the constraint $R(a, u) \subseteq R(X, w)$.

For each edge $u \xrightarrow{X} v$ produced by Algorithm 1, we have $u \in R(X, v)$. Specifically, the first constraint rule handles terminal edges (Table 1) introduced at the initialization step of Algorithm 1, and Rows 3-6 processes edges introduced by the four kinds of productions in Algorithm 1, respectively. Consequently, the least solution satisfying constraints specified in Table 1 gives identical results to Algorithm 1.

Thus, CFL-reachability is then solved by finding the least solution satisfying all these introduced set constraints. While previous works have demonstrated the interconvertability between CFL-reachability and set-constraint problems [20], [27], we argue that our constraint rules (Table 1) are more intuitive and considerably simpler. Furthermore, previous works rely on an off-the-shelf general constraint solver. In contrast, we develop a solver specifically for CFL-reachability, with built-in optimizations implemented for transitivity and common production rules such as Dyck-style productions (Section 4.1.3).

Algorithm 2: The multi-derivation CFL-reachability algorithm

```
   Input: Normalized CFG = (Σ, N, P, S),
          edge-labeled directed graph G = (V, E)
   Output: all reachable pairs in G
1  Function ConsCFL(P, G):
2      Init();
3      Solve(P, G);
4  Procedure Init():
5      for each terminal edge u -t-> v ∈ E do
6          let newRel = {u};
7          DiffProp(newRel, t, v);
8      for each production X ::= ε ∈ P do
9          for each node v ∈ V do
10             let newRel = {v};
11             DiffProp(newRel, X, v);
12 Procedure Solve(P, G):
13     while NW ≠ ∅ do
14         pop (Y, v) from NW ;
15         ΔY = ΔR(Y, v);
16         ΔR(Y, v) = ∅;
17         R(Y, v) = R(Y, v) ∪ ΔY;
18         for u ∈ ΔY do
19             S(Y, u) = S(Y, u) ∪ {v}
20         HandleItem(P, G, Y, ΔY, v);
21 Procedure HandleItem(P, G, Y, ΔY, v):
22     for each production X ::= Y ∈ P do
23         DiffProp(ΔY, X, v);
24     for each production X ::= Y Z ∈ P do
25         for each u ∈ S(Z, v) do
26             DiffProp(ΔY, X, u);
27     for each production X ::= Z Y ∈ P do
28         for each node u ∈ ΔY do
29             let newRel = R(Z, u);
30             DiffProp(newRel, X, v);
31     for each production X ::= a Y b ∈ P do
32         for each node u ∈ ΔY do
33             for each w ∈ S(b, v) do
34                 let newRel = R(a, u);
35                 DiffProp(newRel, X, w);
36 Procedure DiffProp(newRel, X, v):
37     ΔR(X, v) = ΔR(X, v) ∪ (newRel \ R(X, v));
38     if ΔR(X, v) changes then
39         add tuple ⟨X, v⟩ into NW;
```

4.1.2 The Multi-derivation Algorithm

Algorithm 2 presents our algorithm for multi-derivation. Similar to the standard algorithm (Algorithm 1), Algorithm 2 is divided into two steps, initialization (`Init` in Lines 4-11) and reachability solving (`Solve` in Lines 12-35).

The algorithm maintains a work list NW of 2-tuples

$\langle X, v\rangle$, where X is an edge label and v is a node. Recall that $R(X, v)$ and $S(X, v)$ denote the predecessor and successor sets of Node v in the input edge-labeled graph G, respectively (Section 2.3). We leverage the widely used *difference propagation* technique [29], [30], [31] to avoid redundant processing, and $\Delta R(Y, v)$ denotes the set of *new* Y-reachability relations of node v to be processed, i.e., $\Delta R(Y, v) = \{u | u \xrightarrow{Y} v$ is a *newly introduced* Y-reachability relation not yet processed$\}$. Initially, $\Delta R(Y, v)$ is $\emptyset$. The algorithm dynamically updates $\Delta R(Y, v)$ with newly introduced Y-reachability relations, and the set is reset to $\emptyset$ after all relations in $\Delta R(Y, v)$ are processed.

The initialization step handles two kinds of edges: edges labeled by a terminal symbol from the input graph (Lines 5-7) and self-referencing edges produced by productions in the form $X ::= \varepsilon$ (Lines 8-11). Given a terminal edge $u \xrightarrow{t} v$, node u is added into $\Delta R(t, v)$ and the pair $\langle t, v\rangle$ is pushed into NW via the procedure `DiffProp` (Lines 36 - 39). Self-referencing edges are handled in a similar fashion, to satisfy constraints indicated by ε-production.

During reachability solving, a pair $\langle Y, v\rangle$ is selected at each iteration (Line 14). The set of newly introduced relation $\Delta R(Y, v)$ is reset to $\emptyset$ after being copied to ΔY for further processing (Lines 15-16). Next, the edge-labeled graph is updated by incorporating the set of new edges $\{u \xrightarrow{Y} v | u \in \Delta Y\}$ (Lines 17-19). In line 20, we invoke the procedure `HandleItem` to process the four kinds of productions involving Y-reachability, as follows:

- For unary production $X ::= Y$, $\Delta R(X, v)$ is updated with ΔY (with $R(X, v)$ excluded), ensuring the unary constraint $R(Y, v) \subseteq R(X, v)$ (Line 22-23).
- Regarding a production $X ::= Y\ Z$, for each outgoing Z-edge $v \xrightarrow{Z} u$ from node v, ΔY is merged into $\Delta R(X, u)$ with $R(X, u)$ excluded (Lines 24-26). Consequently, the constraint $R(Y, u) \subseteq R(X, v)$ is satisfied.
- For a production $X ::= Z\ Y$, for each Y-edge $u \xrightarrow{Y} v$ where $u \in \Delta Y$, the Z-reachability relation of node u ($R(Z, u)$) is merged into $\Delta R(X, v)$ (Lines 27-30), satisfying the constraint $R(Z, u) \subseteq R(X, v)$.
- Finally, given a Dyck-style production $X ::= a\ Y\ b$, for each Y-edge $u \xrightarrow{Y} v$ where $u \in \Delta Y$ and an edge $v \xrightarrow{b} w$, the a-reachability relation of node u ($R(a, u)$) is merged into $\Delta R(X, w)$ (Lines 31-35), with respect to the constraint $R(a, u) \subseteq R(X, w)$.

In the end, all predecessors and successors of each node are computed and the resulting graph is guaranteed to satisfy all constraints in Table 1.

4.1.3 Optimization for Dyck-style Productions

Productions in the form $X ::= a\ Y\ b$ are prevalent in many CFL-reachability based program analyses. They are used to model a range of programming constructs, including matching field load and store [7], [8], [10], [12], pointer deference and reference [9], or procedure call and return [20], [32], [33]. These productions generalize Dyck-CFL reachability [20], which requires X and Y to be the same non-terminal symbol.

$$A ::= A\ B \mid B$$
$$X ::= X B \mid x$$
$$B ::= a$$

Fig. 3: An equivalent context-free grammar rewritten from the CFG in Figure 2a to reduce redundant propagation.

In the standard algorithm [27], the production $X ::= a\ Y\ b$ undergoes normalization into two productions: $Z ::= a\ Y$ and $X ::= Z\ b$. Each of these productions contains at most two symbols on the right-hand side and Z is a newly introduced non-terminal symbol. However, such normalization introduces unnecessary computation of Z-reachability relations, often leading to a significant degradation in overall performance. A Z-reachability relation $u \xrightarrow{Z} v$ is necessary only if there exists an outgoing b-edge from v. However, in practice, Y-relation is often dominant and most Z-relations generated by production $Z ::= a\ Y$ are unnecessary.

To overcome the above drawbacks, we propose to process the production $X ::= a\ Y\ b$ directly without normalization. Thus, given a production $X ::= a\ Y\ b$, we have the constraint $\forall u \xrightarrow{Y} v \in E \wedge v \xrightarrow{b} w \in E, R(a, u) \subseteq R(X, w)$. New X-reachability relations are obtained by computing the *Cartesian product* of Node u's a-predecessors ($R(a, u)$) and Node v's b-successors ($S(b, v)$). Both the standard algorithm and our multi-derivation algorithm can adopt this optimization, as highlighted in Algorithm 1 and Algorithm 2, respectively.

4.2 Reducing Transitive Redundancy with Propagation Graphs

In this section, we enhance our multi-derivation algorithm by incorporating propagation graphs and demonstrate how this improvement helps reduce transitive redundancy.

4.2.1 Propagation Graphs

The key to reducing transitive redundancy is to identify and prune secondary edges, since these edges do not contribute to deriving new edges but only introduce repetitive derivation. In the simple case, secondary edges are derived by fully transitive relations (e.g., $A ::= A\ A$ in Figure 2c), and those secondary edges can be easily identified and removed with syntactical grammar rewriting.

For instance, the context-free grammar (CFG) in Figure 2a can be transformed into an equivalent CFG in Figure 3. This grammar transformation introduces a new non-terminal symbol B, to represent edges deduced by a-edges. Then the original production $X ::= X\ A$ is replaced by $X ::= X\ B$. As a result, X-edges are derived via B-edges only, rather than via secondary A-edges introduced by $A ::= A\ A$ in the original CFG. Note that compared to the number of B-relations, the number of A-relations is significantly larger since it amounts to the transitive closure of B-edges (Section 3.2). Hence, the extra overhead of computing B-relation is usually negligible.

However, secondary edges can also be dynamically generated by non-transitive productions, e.g., $A ::= B\ C$. And

it is not a trivial task to effectively identify and prune such secondary edges. In our approach, we address this challenge by maintaining a *propagation graph*, denoted as $PG(A)$.

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

$PG(A)$ is *transitivity-aware* since it implicitly preserves all A-reachability relations in the original graph G. More precisely, the transitive closure of $PG(A)$ records all A-relations in G. Transitive relations involving A are efficiently propagated in $PG(A)$, avoiding redundant propagation via secondary A-edges.

In the literature, the minimum equivalent graph [34] and transitive reduction [35] – two topics from the graph reachability community – share a similar high-level concept with our propagation graph. Given an input graph G, these two previous techniques aim to find the smallest graph G' such that there is a path from Node u to Node v in G' whenever there is a path from Node u to Node v in G. Analogously, propagation graphs apply this idea to the transitive relations for CFL-reachability. Importantly, propagation graphs work on a dynamic graph (with transitive edges introduced during reachability solving), and do the edge reduction partially (meaning that we do not find the smallest graph) to exploit a sweet spot between efficiency (the time overhead to build propagation graphs) and effectiveness (removing more useless edges to make the graph smaller).

4.2.2 Dynamic Construction of Propagation Graph

Algorithm 3 enhances the multi-derivation algorithm (Algorithm 2) with propagation graphs to optimize the handling of fully and partially transitive relations. The algorithm constructs propagation graphs on the fly during CFL-reachability solving. In a nutshell, $PG(A)$ is dynamically updated with newly introduced primary A-edges until a fixed point where no more new edges can be deduced. This step effectively resolves fully transitive productions $A := AA$. Moreover, partially transitive productions including $X := XA$ and $X := AX$ are handled using only primary A-edges to avoid transitive redundancy.

How do we know whether a newly deduced A-edge is a primary edge or not? We maintain the transitive closure of $PG(A)$, and the notation $R^*(A, v)$ represents all nodes that can *transitively* reach Node v in $PG(A)$. Thus a new A-edge $u \xrightarrow{A} v$ is regarded as a primary edge only if $u \notin R^*(A, v)$. This strategy guarantees soundness by incorporating all primary A-edges in $PG(A)$, it also achieves high efficiency by discarding as many secondary edges as possible. Note that here we use the notation R^* instead of R, indicating that, for a fully transitive relation A, the result set R solely contains primary A-edges (i.e., edges in $PG(A)$). We next show how $PG(A)$ and its transitive closure are constructed.

Computing Transitive Closure of $PG(A)$. During initialization, given a transitive relation A, both $PG(A)$ and $R^*(A, v)$ (for any node $v \in V$) are initialized to $\emptyset$. Hence, we have introduced 4 new lines (Lines 3-6) to the original procedure `ConsCFL` from Algorithm 2. Moreover, the original `DiffProp` procedure is modified to handle fully transitive A-edges, as shown in Lines 10-14 of Algorithm 3. The procedure `UpdatePG` updates $PG(A)$ and $R^*(A, v)$ according to the newly derived primary edge $u \xrightarrow{A} v$ (Line 13). $\Delta R(X, v)$ is updated accordingly. In `UpdatePG` (Lines 20-22), given a new primary edge $u \xrightarrow{A} v$, we first insert the edge into $PG(A)$. Next, the transitive closure of $PG(A)$ is updated by propagating the new reachability relations introduced by $u \xrightarrow{A} v$ (i.e., $R^*(A, u) \cup \{u\}$) in a depth-first manner in the procedure `DFS`.

In procedure `DFS` (Lines 24-26), only new reachability relations ΔA are propagated along $PG(A)$ to avoid redundant propagation. $R^*(A, u)$ is then updated by incorporating ΔA. Importantly, for a fully transitive relation A, the production $A ::= A\ A$ is also handled simultaneously by computing the transitive closure $R^*(A, v)$ when dynamically constructing $PG(A)$. Hence, CFL-reachability can be solved by skipping fully transitive productions from the production set P before invoking procedure `Solve`, as shown in Lines 7-8.

Handling Non-transitive Productions. A-edges derived in procedure `DFS` are the so-called secondary A-edges and they will be discarded in solving partially transitive productions, i.e., $X ::= X\ A$ and $X ::= A\ X$. Nevertheless, secondary edges are still needed for non-transitive productions such as $X ::= A\ Y$ and $X ::= A$. Hence, for newly generated secondary edges, we invoke the procedure `HandleItem` to handle those non-transitive productions (Lines 27-28).

4.2.3 Implementation Strategy

This section discusses some practical trade-offs in dynamically constructing propagation graphs, including the eager propagation strategy, how insertion order affects the sparseness of propagation graphs, and the memory overhead introduced by the propagation graph.

Eager Propagation. The `DFS` procedure eagerly propagates A-reachability relations to compute the transitive closures of $PG(A)$. Alternatively, one can iteratively propagate A-reachability relations using a worklist, as shown in Algorithm 2. However, this worklist-based approach often results in the creation of redundant secondary edges in $PG(A)$, which can degrade performance. Figure 4 illustrates such a scenario.

Example 1. There are three steps in Figure 4, with one edge (the dashed line) deduced at each step, as follows.

- *Step(a)*, $1 \xrightarrow{A} 2$ is deduced and inserted into $PG(A)$. Consequently, we have $\{1\} \subseteq R(A, 2)$.
- *Step(b)*, $0 \xrightarrow{A} 1$ is deduced and inserted into $PG(A)$. As a result, we have $\{0\} \subseteq R(A, 1)$.
- *Step(c)*, the secondary edge $0 \xrightarrow{A} 2$ is deduced via non-transitive productions. However, the worklist-based approach is unaware that 2 is reachable from 0 since such information is not explicit until we propagate $\{0\} \subseteq R(A, 1)$ to Node 2. Hence, the edge $0 \xrightarrow{A} 2$ may be mistakenly inserted to $PG(A)$, causing performance degradation.

Algorithm 3: Enhancing Algorithm 2 with propagaiton grpahs.

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

1 **Function** `ConsCFL`(P, G)**:**
2 Init(); /* Lines 4-11 in Algo. 2 */
3 **for** *each fully transitive relation* A **do**
4 $PG(A) = \emptyset$;
5 **for** *each node* $v \in V$ **do**
6 $R^*(A, v) = \emptyset$;
7 let P' be P without fully transitive productions;
8 `Solve`(P', G); /* Lines 12-20 in Algo. 2 */
9 **Procedure** `DiffProp`$(newRel, X, v)$**:**
10 **if** X *is fully transitive* **then**
11 **for** *each node* u *in* $newRel$ **do**
12 **if** $u \notin R^*(X, v)$ **then**
13 `UpdatePG`(X, u, v);
14 $\Delta R(X, v) = \Delta R(X, v) \cup \{u\}$;
15 **else**
16 $\Delta R(X, v) = \Delta R(X, v) \cup (newRel \setminus R(X, v))$;
17 **if** $\Delta R(X, v)$ *changes* **then**
18 add (X, v) into NW
19 **Procedure** `UpdatePG`(A, u, v)**:**
20 add $u \xrightarrow{A} v$ to $PG(A)$;
21 $srcSet = R^*(A, u) \cup \{u\}$;
22 `DFS`$(A, srcSet, v)$;
23 **Procedure** `DFS`$(A, srcSet, u)$**:**
24 $\Delta A = srcSet \setminus R^*(A, u)$;
25 **if** $\Delta A \neq \emptyset$ **then**
26 $R^*(A, u) = R^*(A, u) \cup \Delta A$;
27 let P^- be P without productions in the form of $X ::= X\ A$, $X ::= A\ X$, and $A ::= A\ A$;
28 `HandleItem`$(P^-, G, A, \Delta A, u)$; /* Lines 21-35 in Algo. 2 */
29 **for** $u \xrightarrow{A} v \in PG(A)$ **do**
30 `DFS`$(A, \Delta A, v)$;

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

Insertion Order. Suppose that in Figure 4, the insertion order is $0 \xrightarrow{A} 2$, $0 \xrightarrow{A} 1$ and $1 \xrightarrow{A} 2$. Then even with eager propagation, $0 \xrightarrow{A} 2$ will cause redundant propagation but is still kept in $PG(A)$. Based on our empirical experience, such redundant edges only occupy a small portion, making them acceptable. Besides, identifying and removing all secondary edges in the presence of cycles and dynamically

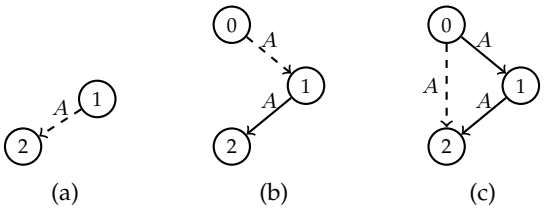


Fig. 4: An example. Three edges are inserted at each step in turn. Edges $1 \xrightarrow{A} 2$ and $0 \xrightarrow{A} 1$ are both primary edges, while $0 \xrightarrow{A} 2$ is a secondary edge and need to be excluded from $PG(A)$ to avoid redundant propagation.

inserted edges, i.e., *online transitive reduction* [36] (which has the same complexity as computing the transitive closure [35]), would also incur additional costs.

In practice, the insertion order is determined by the worklist order of the CFL-reachability algorithm. We have experimented with different ordering strategies (e.g., FIFO or FILO), and the performance differences are negligible.

Memory Overhead of Propagation Graph. Given a fully transitive relation A, when the CFL-reachability algorithm reaches a fixed point, let E_A be the set of all A-edges. According to Definition 4, E_A is the transitive closure of $PG(A)$, hence edges in $PG(A)$ are negligible when compared to E_A.

4.2.4 Multi-derivation for Partially Transitive Productions

Partially transitive relations (e.g., $X ::= X\ A$ where A is fully transitive) are calculated by propagating X-reachability relations via primary A-edges, that is, edges in $PG(A)$. This method effectively avoids redundant propagation when solving partially transitive productions.

Let us revisit the example in Figure 2. There are two primary A-edges in Figure 2c: $1 \xrightarrow{A} 2$ and $2 \xrightarrow{A} 3$. Given the partially transitive production $X ::= X\ A$, each primary A-edge will introduce a constraint, resulting in two new constraints, $R(X, 1) \subseteq R(X, 2)$, and $R(X, 2) \subseteq R(X, 3)$, respectively. The secondary A-edge $1 \xrightarrow{A} 3$ is implicitly encoded in $PG(A)$, instead of being explicitly represented in the labeled graph G ($1 \notin R(A, 3)$, since $1 \xrightarrow{A} 3$ is a secondary A-edge). Hence, it will not generate constraints for X-reachability relations, effectively avoiding redundant propagation.

4.2.5 Correctness

Algorithm 3 optimizes the processing of both the fully and partially transitive relations of our multi-derivation approach. We now prove the correctness of this optimization.

Lemma 1. R^* is the transitive closure of $PG(A)$.

Proof sketch. According to Algorithm 3 (Lines 21, 22, and 25), for each edge $u \to v$ in $PG(A)$, we have two constraints $R^*(A, u) \subseteq R^*(A, v)$ and $u \in R^*(A, v)$. Recall that initially $R^*(A, v)$ (for any node v) is empty, and is calculated using *solely* these two constraint rules. Thus, R^* is the transitive closure of $PG(A)$. □

By Lemma 1, we have the following corollary.

Corollary 1. Node v is reachable from Node u in $PG(A)$ iff $u \in R^*(A, v)$.

Lemma 2. An A-edge $u \xrightarrow{A} v$ exists iff Node v is reachable from Node u in $PG(A)$.

Proof sketch. Regarding the generation of A-edges, Algorithm 3 differs from Algorithm 2 only in handling the production $A ::= AA$.

For each A-edge, say $u \xrightarrow{A} v$, produced by a production other than $A ::= AA$, Algorithm 3 (Lines 12-13) attempts to insert it into $PG(A)$. The edge $u \xrightarrow{A} v$ is not inserted into $PG(A)$ only if condition $u \in R^*(A, v)$ holds, which means that Node v is already reachable from Node u in $PG(A)$ (Corollary 1). In that sense, discarding such an edge does not affect A-reachability on $PG(A)$.

On the other hand, a fully transitive production $A ::= AA$ only generates A-edges that are formed by connecting two or more consecutive A-edges. Thus, excluding these edges does not influence A-reachability of $PG(A)$. This completes our proof. □

Theorem 1. Algorithm 3 correctly derives all A-edges (by computing $R^*(A, v)\ \forall v \in V$).

Proof sketch. Applying Corollary 1 and Lemma 2 together, an A-edge $u \xrightarrow{A} v$ exists iff $u \in R^*(A, v)$. □

Theorem 2. Algorithm 3 correctly handles partially transitive productions $X ::= X\ A$ (resp. $X ::= A\ X$).

Proof sketch. When solving partially transitive relations, Algorithm 3 only uses primary A-edges (edges in $PG(A)$) to handle production $X ::= X\ A$ (resp. $X ::= A\ X$).

According to Lemma 2, an A-edge $u \xrightarrow{A} v$ exists iff there is a path from Node u to Node v in $PG(A)$. Concerning production $X ::= X\ A$ (resp. $X ::= A\ X$), X-edges can be *transited* via consecutive A-edges to derive new X-edges. This transitivity ensures that all X-edges can be exactly derived using only edges in $PG(A)$. Therefore, Algorithm 3 correctly solves partially transitive productions. □

4.2.6 Time and Space Complexity

The theoretical time and space complexity of PEARL (the multi-derivation algorithm) is the same as that of the single-reachability derivation (SRD) approach, including POCR and the standard algorithm. Consider a production $X ::= YZ$ with multiple Y-edges ($\{u_0, u_1, ..., u_k\} \xrightarrow{Y} v$) and a Z-edge $v \xrightarrow{Z} w$. The multi-derivation algorithm process the set of k Y-edges in batches, which significantly reduces the number of iterations but does not impact the theoretical complexity. Nevertheless, this batching processing approach brings significant practical benefits, as evident in Table 4, Table 5 and Table 6.

5 EVALUATION

We evaluate the performance of our multi-derivation approach on two practical static analysis clients: context-sensitive value-flow analysis [24], [25] and field-sensitive alias analysis [9] for C/C++.

5.1 Experimental Setup

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

Value-flow Analysis. We perform context-sensitive value-flow analysis on the sparse value-flow graphs (SVFG) [24], [25]. Figure 5a shows the context-free grammar (CFG) for value-flow analysis, where $call_i$, ret_i and a are terminals, and A is a non-terminal symbol and also the start symbol. In particular, $call_i$ and ret_i denote parameter passing and return flow at the i-th call site respectively, a denotes an assignment, and A denotes an intraprocedural/interprocedural value flow. In fact, this CFG is a specialization of the popular Dyck-CFL grammar [10], [20], which has been extensively studied in recent years. The analysis is also field-sensitive, since each field object is represented as a single node in the SVFG. The normalized grammar is listed in Figure 5b.

Alias Analysis. The field-sensitive alias analysis for C++ is conducted on the program expression graph (PEG) [9]. Figure 6a presents the CFG, where a denotes an assignment statement, d denotes a pointer dereference, f_i denotes the address of i-th field, A denotes a direct/indirect value flow, M denotes memory alias, and V denotes value alias. $M?$ means that M is optional. The PEG is bidirected [10], [11], [32], i.e., for an edge $u \xrightarrow{X} v$ in PEG, there is a reverse edge $v \xrightarrow{\overline{X}} u$ in PEG. The normalized grammar is shown in Figure 6b.

Setup and Benchmarks. We use the benchmarks[1] provided by POCR [23]. These benchmarks contain SVFG and PEG of 10 SPEC 2017 C/C++ programs. Following POCR, SVFG and PEG are pre-processed by cycle elimination [37] to collapse cycles of a-edges and variable substitution [38] to compact particular a-edges. In Table 2, columns 2-5 and columns 6-9 list the numbers of nodes and edges of SVFG and PEG before and after offline preprocessing on each benchmark, respectively.

Baselines. We compare our approach with two baselines: the standard algorithm (Algorithm 1), denoted as STD, and a state-of-the-art solver, POCR [23], which has been open-sourced on Github[2]. POCR reduces derivation redundancy due to transitivity via ordered derivations.

For illustration, we mainly compare two groups of algorithms: (1) transitivity-unaware group, namely, STD and PEARL (Algorithm 2). For a fair comparison, we rewrite the input CFGs, as shown in Figure 5c and Figure 6c, to reduce part of the transitive redundancy syntactically [23]. (2) transitive-aware group, that is, PEARL^{PG}, POCR, and POCR^{PG}. PEARL^{PG} (Algorithm 3) enhances PEARL with propagation graphs. POCR^{PG}– an ablation of POCR designed by us – replaces the default spanning tree model of POCR with our propagation graph, while keeping the other implementation parts of POCR unchanged. We use the CFGs in Figure 5b and Figure 6b. All algorithms (including two base-

1. https://github.com/kisslune/CPU17-graphs
2. https://github.com/kisslune/POCR

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

(a) Context-free grammar

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

(b) Normalized grammar

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

(c) Rewritten grammar

Fig. 5: CFG for context-sensitive value-flow analysis

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

(a) Context-free grammar

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

(b) Normalized grammar

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

(c) Rewritten grammar

Fig. 6: CFG for field-sensitive alias analysis

TABLE 2: Benchmark statistics of value-flow analysis and alias analysis. Columns 2-5 and Columns 6-9 give the numbers of nodes and edges in the input graph before and after preprocessing of SVFG and PEG, respectively.

Benchmark	SVFG (Value-Flow Analysis)				PEG (Alias Analysis)			
	Before Preprocessing		After Preprocessing		Before Preprocessing		After Preprocessing	
	#Nodes	#Edges	#Nodes	#Edges	#Nodes	#Edges	#Nodes	#Edges
cactus	544480	1007989	223046	616399	93557	212478	65232	153470
imagick	574089	842509	165096	319141	119314	301846	73499	196730
leela	64466	89081	21711	40409	22186	49748	14371	33326
nab	55652	72366	15415	23736	16261	34676	8794	19218
omnetpp	664358	1857831	237854	1277123	241916	509166	146049	311980
parest	299718	407343	114099	199793	117500	251436	67949	148286
perlbench	697744	1662445	321778	1122795	139183	348916	72231	192994
povray	537775	1041687	213130	621400	76405	174258	45622	110732
x264	207064	340217	66417	162595	60956	136352	40625	94110
xz	49395	62955	15072	23002	12425	26468	7130	15228

lines) are enhanced with the Dyck optimization discussed in Section 4.1.3.

During the evaluation, we identified a performance issue of POCR due to unnecessary operations in value-flow analysis, which has been fixed by us. Consequently, POCR appears to perform better in terms of efficiency than in the previous results of our conference paper [26]. Moreover, the STD solver here runs noticeably faster than the one in our conference paper [26] due to both grammar rewriting and Dyck optimization.

Implementation. All codes including baselines and our approach are implemented on top of a popular static analysis framework SVF [39] in C++. To ensure fair comparisons, all data structures are implemented consistently across distinct baselines and our approach. Specifically, the predecessor set $R(X,u)$ is implemented as a hash map (`std::unordered_map`), which maps a pair of (X,u) to a hash set (`std::unordered_set`) that stores all the X-predeccessors of Node u. The successor set $S(X,u)$ and transitive closure $R^*(X,u)$ are implemented in a similar manner. As an optimization, the delta (predecessor) set $\Delta R(X,u)$ is implemented using a hash map, mapping a pair of (X,u) to an array (`std::vector`) instead of a hash set, since no membership checking is necessary.

Evaluation of Correctness. The correctness of the proposed algorithms in this paper is verified practically by the fact that our solvers compute the same set of reachable pairs as the standard algorithm (if scalable) in our experiments.

Our evaluation aims to answer the following research questions:

- **(RQ1).** How does PEARL compare to the standard algorithm?
- **(RQ2).** How does PEARL^{PG} compare to the state-of-the-art solver POCR?
- **(RQ3).** How extensive are fully and partially transitive edges in real-world CFL-reachability problems?
- **(RQ4).** Is propagation graph representation effective in reducing redundant propagation?

TABLE 3: Speedups by applying grammar rewriting on STD on two clients. "VF" and "AA" denote value-flow analysis and alias analysis, respectively. With grammar rewriting, STD successfully analyzes 1 and 3 more benchmarks for value-flow analysis and alias analysis, respectively. "-" means that STD (either without or with GR) exceeds the time or memory limit, and the speedup number cannot be obtained

id	VF	AA
cactus	39.08x	-
imagick	32.76x	-
leela	3.08x	2.5x
nab	49.32x	1.05x
omnetpp	3.28x	-
parest	1.21x	1.63x
perlbench	-	-
povray	28.21x	-
x264	17.53x	1.29x
xz	2.66x	1.09x
Avg.	19.68x	1.51x

5.2 RQ1. Comparison with STD

Grammar Rewriting. As shown in Table 3, the application of grammar rewriting enables STD to achieve average speedups of 19.68× and 1.51× for value-flow analysis and alias analysis, respectively . Furthermore, grammar rewriting allows STD to analyze 1 more benchmark (perlbench) for value-flow analysis and 3 more benchmarks (cactus, imagick, and povray) for alias analysis, all within the constraints of 6 hours and 512 GB of memory. However, it is important to note that, even with grammar rewriting, some benchmarks (omnetpp and perlbench in alias analysis) remain unanalyzed by STD, highlighting the need for more advanced optimizations.

In value-flow analysis, grammar rewriting results in speedups range from 1.21× for parest to 49.32× for cactus. The effectiveness of grammar rewriting is directly related to the reduction of redundant derivations. For an A-path $v_1 \xrightarrow{A} v_2 \xrightarrow{A} \ldots \xrightarrow{A} v_{n-1} \xrightarrow{A} v_n$ with n primary A-edges, the original production $A ::= A\ A$ may require $n-2$ derivations to derive the edge $v_1 \xrightarrow{A} v_n$ ($v_1 \xrightarrow{A} v_i \wedge v_i \xrightarrow{A} v_n$ for $2 <= i <= n-1$). However, as shown in Figure 5c, with grammar rewriting, the number of derivations needed to obtain the edge $v_1 \xrightarrow{A} v_n$ is reduced to 1 ($v_1 \xrightarrow{A} v_{n-1} \wedge v_{n-1} \xrightarrow{B} v_n$) using the production $A ::= A\ B$, thereby eliminating redundant derivations.

To estimate the number of eliminated redundant derivations, we calculate the ratio of A-edges to B-edges. The larger the ratio, the more redundant derivations are potentially optimized. In the case of parest and cactus, the number of A-edges is 4.46× and 71.04× higher than the number of B-edges, respectively, leading to the observed disparity in speedups.

Although grammar rewriting can achieve substantial performance improvements, there still exists a large amount of transitive redundancy during on-the-fly reachability solving. Hereafter, we evaluate all algorithms with grammar rewriting, to demonstrate the effectiveness of propagation graphs.

Performance Improvement. Table 4 and Table 5 give the efficiency results of value-flow analysis and alias analysis, respectively. To evaluate the benefit of multi-derivation, we compare PEARL with STD in terms of performance. Regarding efficiency, PEARL runs 5.64× and 4.72× faster than STD for value-flow analysis and alias analysis, respectively. During reachability solving, PEARL adopts a muti-derivation manner via batch propagation to reduce propagation efforts, thereby boosting the overall performance.

Reduced Propagation. To quantify the benefit achieved by multi-derivation, We compare STD with PEARL to show how many propagations are pruned by batch propagation. We define the number of propagations during solving reachability as P. Thus, the reduction rate achieved by batch propagation can then be obtained via $(P_{\text{STD}} - P_{\text{PEARL}})/P_{\text{STD}}$. Table 6 shows the reduction rates achieved by PEARL over STD, with average reductions of 80.96% (up to 96.73%) and 79.72% (up to 88.73%) for value-flow analysis and alias analysis, respectively.

Notably, PEARL only reduces 33.75% of propagation for nab in value-flow analysis. This outlier is attributed to an optimization we implemented for STD: newly derived edges are inserted into W first, and they are added to E only when popped out from W. Compared to the default algorithm that simultaneously introduces newly derived edges into W and E (e.g., line 17 in Algorithm 1), this optimization reduces redundant propagation (derivation checks) caused by newly derived edges. Consider the production $A ::= BC$ with two newly produced edges $u \xrightarrow{B} v$ (B-edge) and $v \xrightarrow{C} w$ (C-edge). If both edges are directly added to W and E, there will be two propagation triggered respectively by popping the B-edge and the C-edge from W. In contrast, with the optimization, the redundant propagation triggered by the firstly popped B-edge can be avoided since the C-edge has not been added to the edge set E yet. Note that this optimization reduces redundant propagation at the cost of performing extra checks in W. In our experiments, the optimization can improve performance by up to 20%.

In the case of nab, this benchmark has a dense graph where a large number of edges can be derived repeatedly, but only one copy is kept in W. Hence, a large portion of propagation is filtered out beforehand due to the above optimization. On the other hand, by keeping the newly derived edges in the delta set, PEARL does not require such an optimization to reduce redundant derivations. As a result, the reduction rate of nab is relatively smaller. In fact, the reduction rate of nab without the above optimization is much higher, reaching 97.54%.

Optimization for Dyck-style productions. To study the benefits of the Optimization for Dyck-style productions introduced in Section 4.1.3, we also implement a variant of PEARL without this optimization. Table 7 gives the performance improvement and memory reduction of applying propagation graph on top of PEARL. Specifically, Optimization for Dyck-style productions obtains average speedups of 3.86× and 4.2× on two clients. Furthermore, Optimization for Dyck-style productions also reduces memory consumption because no intermediate reachability relation is needed.

TABLE 4: Runtime statistics (in seconds) of context-sensitive value-flow analysis. . STD denotes the standard algorithm and PEARL represents the multi-derivation algorithm; PEARL^{PG} enhances PEARL with propagation graph, POCR signifies the recent solver in [23], while POCR^{PG}– an ablation of POCR– replaces the default spanning tree model of POCR with our propagation graph. All algorithms are equipped with the Optimization for Dyck-style productions technique.

id	STD	PEARL	PEARL^{PG}	POCR	POCR^{PG}
cactus	356.44	34.73	22.39	89.85	36.74
imagick	44.62	7.03	4.12	11.24	5.70
leela	0.98	0.30	0.14	0.30	0.15
nab	10.25	2.56	2.17	7.90	5.83
omnetpp	19.84	5.68	2.82	7.17	3.35
parest	1.85	0.90	0.27	0.38	0.26
perlbench	496.70	59.41	48.70	171.31	99.42
povray	380.21	41.19	27.90	123.83	48.45
x264	32.16	5.21	3.26	12.24	6.60
xz	0.32	0.10	0.04	0.11	0.05

TABLE 5: Runtime statistics (in seconds) of field-sensitive alias analysis. "-" means exceeding the time limit(6 hours). The meanings of column headings are the same as Table 4.

id	STD	PEARL	PEARL^{PG}	POCR	POCR^{PG}
cactus	2669.86	441.67	97.65	196.37	185.41
imagick	6698.93	1451.08	372.50	642.60	637.43
leela	34.92	7.69	2.21	3.37	3.22
nab	2.56	0.71	0.26	0.86	0.80
omnetpp	-	8605.24	1414	3507.18	5352.42
parest	1238.54	211.80	50.63	124.64	118.36
perlbench	-	-	1962.87	4080.55	3608.89
povray	2750.20	481.12	98.97	238.50	234.73
x264	62.68	16.34	6.08	14.91	14.04
xz	1.90	0.53	0.23	0.48	0.43

TABLE 6: Reduction rates in propagations of reachability relations achieved by PEARL over STD. "VF" and "AA" denote value-flow analysis and alias analysis, respectively. For alias analysis, STD timeouts in two benchmarks (omnetpp and perlbench) while PEARL timeouts in perlbench.

id	VF	AA
cactus	96.14%	88.73%
imagick	92.57%	76.92%
leela	76.78%	72.32%
nab	33.75%	73.42%
omnetpp	76.48%	-
parest	72.23%	87.17%
perlbench	96.73%	-
povray	93.97%	85.79%
x264	89.9%	74.58%
xz	81.08%	78.8%
Avg.	80.96%	79.72%

TABLE 7: Speedups and memory reduction (in terms of memory saved) when applying Dyck optimization with PEARL. "-" means at least one variant timeout. "VF" and "AA" denote value-flow analysis and alias analysis, respectively. "Spu." and "Red." represent speedup and memory reduction, respectively.

id	VF		AA	
	Spu.	Red.	Spu.	Red.
cactus	2.64x	21.34%	2.74x	50.32%
imagick	2.72x	25.56%	7.69x	52.9%
leela	2.07x	20.0%	2.97x	47.5%
nab	1.45x	10.87%	2.03x	27.78%
omnetpp	15.62x	50.38%	1.36x	32.1%
parest	1.69x	14.29%	1.78x	34.91%
perlbench	3.52x	23.8%	-	-
povray	4.71x	39.32%	2.0x	39.97%
x264	2.13x	26.92%	14.42x	61.56%
xz	2.0x	20.0%	2.83x	44.44%
Avg.	3.86 x	25.25 %	4.2 x	43.5 %

In consequence, the reduced memory usages are 25.25% and 43.5% in value-flow analysis and alias analysis, respectively.

Conclusion. By performing batch propagation, PEARL eliminates a substantial number of propagations of reachability relations over STD during reachability solving, thereby obtaining considerable performance improvement. Applying Dyck optimization on top of the multi-derivation algorithm can also achieve promising speedups with considerable memory reduction.

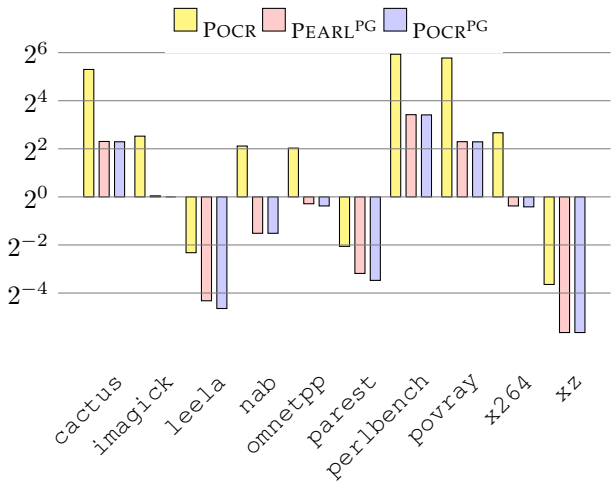


Fig. 7: Memory usage (in GB) in value-flow analysis.

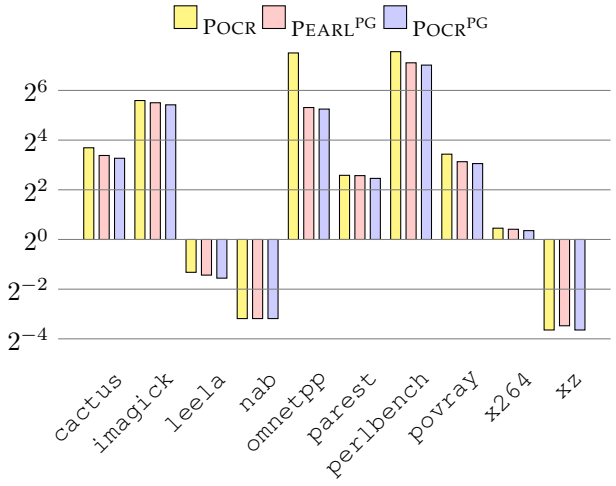


Fig. 8: Memory usage (in GB) in alias analysis.

5.3 RQ2. Comparison with POCR

A recent solver POCR [23] utilizes the spanning tree model to reduce redundant propagation for transitive relations, which is briefly described in Section 3.2. Here we evaluate the performance of our solver PEARL^{PG}, which enhances multi-derivation with propagation graph representation, against POCR.

For a client analysis, POCR provides two options: (1) a general solver, which accepts a CFG defined by a grammar file, and (2) a built-in solver, which "inlines" the grammar into the algorithm. A built-in solver typically outperforms its general counterpart, since there are more optimization opportunities for a specific client, e.g., symmetry. Following POCR [23], we choose built-in solvers to evaluate the performance for a fair comparison. To this end, we implement our built-in solvers to compare with POCR.

Result. Table 4 and Table 5 display the performance of PEARL^{PG} and POCR in value-flow analysis and alias analysis, respectively. In terms of efficiency, PEARL^{PG} outperforms POCR on all benchmarks. We also give the memory consumption of PEARL^{PG} and POCR of two clients in Figure 7 and Figure 8.

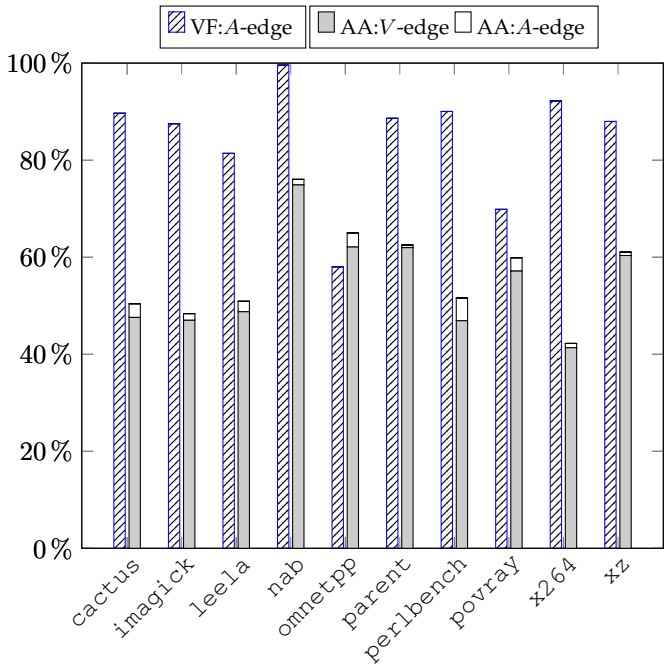


Fig. 9: The percentages of fully and partially transitive edges among all inserted edges for value-flow analysis and alias analysis, respectively. "VF:A-edge" denotes fully transitive A-edges in value-flow analysis, and "AA:V-edge" and "AA:A-edge" represents partially transitive V-edges and fully transitive A-edges in alias analysis, respectively.

Time consumption. In value-flow analysis (Table 4), the comparison to POCR shows that PEARL^{PG} achieves an average speedup of 3.09× over POCR, with a maximum improvement of 4.44× for `povray`. It is worth noting that PEARL^{PG} solves each benchmark for value-flow analysis within one minute. In alias analysis (Table 5), compared to POCR, PEARL^{PG} achieves a performance improvement of 2.25× over POCR on average (up to 3.31×).

Memory usage. Figure 7 and Figure 8 demonstrate that PEARL^{PG} consumes less memory than POCR for almost all benchmarks. In value flow analysis (Table 4), particularly, where the fully transitive edges dominate, PEARL^{PG} achieves significant memory savings compared to POCR. For instance, in the `cactus` benchmark, POCR utilizes nearly 40 GB of memory, whereas PEARL^{PG} only requires less than 5 GB of memory.

Discussion. By combining multi-derivation with propagation graph, PEARL^{PG} obtains promising speedups over POCR for both clients.

5.4 RQ3. Transitive Edges

Transitive relation plays an important role in CFL-based program analyses. For example, both data flow and control flow are inherently transitive. We quantify the importance of transitive relations by measuring the portion of partially and fully transitive edges.

Figure 9 demonstrates the percentages of fully/partially transitive edges among all edges added to the edge-labeled graph during analysis. It is worthy pointing out that derived edges are typically orders of magnitude greater than

terminal edges, i.e., the labeled edges in the input graph; hence, the number of added edges is very close to the total number of edges at the conclusion. For instance, the difference is less than 1% for `perlbench` in value-flow analysis. These statistics are collected after the CFL-reachability analysis reaches a fixed point. "VF:A-edge" denotes fully transitive A-edges in value-flow analysis, and "AA:V-edge" and "AA:A-edge" represents partially transitive V-edges and fully transitive A-edges in alias analysis, respectively.

Result. In value-flow analysis, fully transitive edges (A-edges) account for a percentage of 84.49% on average and there are no partially transitive edges. In alias analysis, fully transitive edges (A-edges) represent a negligible proportion (2.0%) while partially transitive edges (V-edges) occupy 54.81% of total added edges on average.

Discussion. As shown in columns 2-5 and 6-9 of Table 2, offline preprocessing has already pruned a large number of fully transitive edges. However, transitive edges still make up a significant proportion of all edges added during reachability solving in value-flow analysis (84.49%) and alias analysis (56.81%). Thus, it is essential to accelerate edge derivations involving transitive edges to efficiently solve CFL-reachability problems. It is also worth mentioning that we use normalized grammar for calculation, which includes the intermediate reachability relations introduced during grammar normalization. Without this normalization step, the percentages of transitive edges would be higher.

5.5 RQ4. Effectiveness of Propagation Graph

To show the effectiveness of propagation graph, we mainly conduct two experiments: (1) compare PEARL^{PG} with PEARL; (2) compare propagation graph with the spanning tree model on top of POCR.

5.5.1 PEARL^{PG} *vs.* PEARL

Speedups. As demonstrated in Table 4 and Table 5, when applying propagation graph on top of multi-derivation, PEARL^{PG} obtains speedups of 1.87× and 3.86× over PEARL in two clients. Recall that we rewrite the input CFGs to remove part of transitive redundancy (Figure 5c and Figure 6c), otherwise, the performance gap would be more significant.

Discussion. According to our empirical experience, there are two kinds of transitive redundancy: (1) *syntactic redundancy*, which can be eliminated easily by grammar rewriting, and (2) *dynamic redundancy*, which can only be detected on the fly. In principle, both kinds of redundancy can be diminished by PEARL^{PG} (or POCR), while we perform grammar rewriting for PEARL and STD to remove syntactic redundancy beforehand in our experiments.

We observe that grammar rewriting can eliminate a significant portion of secondary edges in value-flow analysis, but this is not the case for alias analysis. In alias analysis, there is still a large amount of dynamic redundancy, which can not be reduced by grammar rewriting. As a result, there is more room for optimization and, thus propagation graph performs better in alias analysis, obtaining a larger performance improvement than in value-flow analysis.

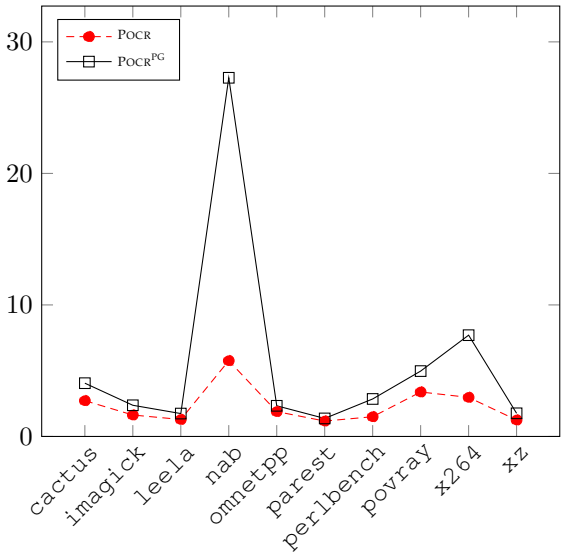


Fig. 10: Computational redundancy of value-flow analysis

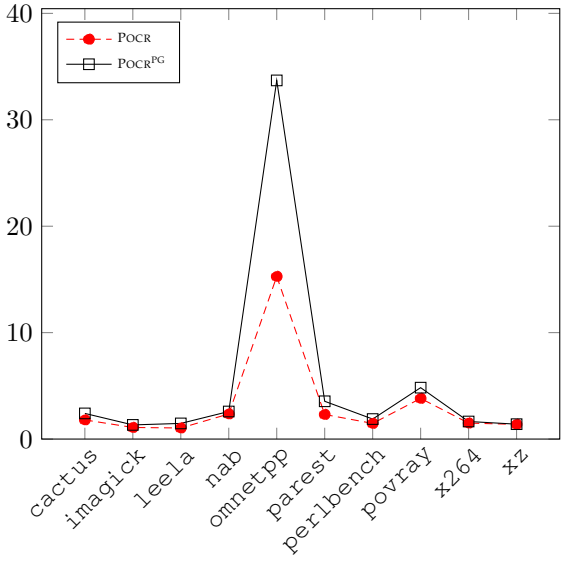


Fig. 11: Computational redundancy of alias analysis

5.5.2 *Spanning Tree vs. Propagation Graph*

To evaluate the effectiveness of the propagation graph representation in eliminating redundant derivations, we design an ablation of POCR, namely POCR^{PG}. In this ablation, we replace only the spanning tree model of POCR with the propagation graph representation, leaving all other components of POCR untouched. We compare POCR^{PG} with POCR in terms of the reduced derivations and overall performance. Recall that both algorithms follow the single-reachability relation derivation (SRD) manner, so the number of derivations equals the number of propagations.

Computational Redundancy. Figure 10 and Figure 11 evaluate the *computational redundancy* defined by D/A, where D and A are the number of total derivations and the number of edges added to the graph. Computational redundancy measures how many derivations are needed for an actual edge addition on average. Lower computa-

tional redundancy means that more repetitive derivations are eliminated during the solving process. The computational redundancy of POCR^{PG} and POCR are close for most benchmarks. On average, the redundancy values of POCR are 2.36 and 3.2 in value-flow analysis and alias analysis, respectively. The average redundancy values of POCR^{PG} are 5.64 in value-flow analysis and 5.48 in alias analysis.

Discussion. POCR maintains two spanning trees, a predecessor tree and a successor tree for each node (as the tree root) [23], [40]. Given a fully transitive relation A, the spanning tree model ensures that each node in a tree is reachable from the root node via only one path. On the other hand, POCR^{PG} retains a global propagation graph for all nodes, where a node pair can be connected via multiple reachable A-paths. As a result, POCR eliminates more repetitive derivations than POCR^{PG} commonly. However, we find that reducing more derivations does not necessarily result in better performance since it can also entail maintenance costs.

Overall Performance. The performance statistics of POCR^{PG} for value-flow and alias analysis are respectively listed in Table 4 and Table 5. POCR^{PG} achieves an average speedup of 1.97x over POCR for value-flow analysis (Table 4). For alias analysis (Table 5), POCR^{PG} runs slightly faster (1.02x) than POCR. Furthermore, POCR^{PG} outperforms POCR in almost every benchmark.

Discussion. We notice that POCR takes a non-trivial amount of work to maintain spanning trees in our experiments, especially when fully transitive edges are dense. This is because an edge is usually copied multiple times by the spanning tree model, which incurs both computation and storage overhead. The larger speedup in value-flow analysis than in alias analysis achieved by POCR^{PG} over POCR, confirms the aforementioned statistics that fully transitive relations dominate in value-flow analysis (Figure 9). In addition, POCR^{PG} saves a lot of memory compared to POCR, particularly in value-flow analysis (Figure 7). This is because our propagation graph representation is conceptually simple and cheap to update on the fly. For the `povary` benchmark, POCR^{PG} consumes approximately 5 GB consumed memory while POCR uses nearly 55 GB consumed memory. This emphasizes that exhaustively diminishing computational redundancy by POCR does not necessarily result in improved overall performance, because it can entail additional costs to maintain the acyclic property of spanning trees. In alias analysis, the performance of POCR^{PG} and POCR are comparable because fully transitive edges only take a small proportion and the representation maintenance cost is negligible compared to overall solving time. POCR^{PG} is slower than POCR in only one benchmark, i.e., `omnetpp`, for alias analysis. This discrepancy is primarily due to a considerable number of erroneously introduced secondary edges in the propagation graph due to insertion order (Section 4.2.3).

To sum up, propagation graph representation is effective (keeping reasonable computational redundancy) and lightweight (cheap to maintain) for both two clients, achieving a promising overall performance. Moreover, when compared to the spanning tree model, propagation graph is conceptually simple, making it easy to implement.

6 Related Work

This work is relevant to improving the efficiency of CFL-reachability analysis. Initially proposed in the context of database theory [15], the CFL-reachability framework becomes popular in formulating many program analysis problems [6]. Since then, CFL-reachability has been studied in various contexts such as recursive state machine [23], [41], pushdown system [42], [43], [44], set constraint [20], [27] and visibly pushdown languages [45] (a subset of deterministic CFL). It is commonly known that CFL-reachability-based algorithms have a cubic worst-case complexity. Previous work [16] showed that the Four Russians' Trick could yield a subcubic algorithm by utilizing set operations of bit-vectors at low amortized time, which is orthogonal to our approach. So far, Significant progress has been made for specific clients, such as bidirected Dyck-reachability [10], [32], [46], IFDS-based analysis [47], [48], [49], [50], [51], [52], [53], pointer analysis [7], [8], [9], [12], [13], [14], [54], to just name a few. In such cases, algorithms are typically designed for a predefined context-free grammar with specialized optimizations. In principle, our techniques can be applied on top of them to yield better performance.

A prevalent solution to avoid derivation redundancy is to construct summary edges for common paths [1], [3], [12], [13], [14], [20], [55], known as summarization. Sparse analysis [24], [25], [50], [55], [56], [57], [58], [59] adopts a similar idea by summarizing data dependencies to skip unnecessary paths. However, summarization does not fundamentally address the derivation redundancy due to the single-reachability derivation manner and transitivity solving, and thus its effectiveness is limited. Reducing the graph size by offline preprocessing techniques [17], [18], [19], [38] can also alleviate redundancy. Nevertheless, a large amount of redundancy can only be captured during online CFL-reachability solving because edges are dynamically derived and inserted into the graph.

Factorized databases [60] also exploit a similar "batching" idea, known as factorization, to evaluate join and aggregation queries in relational databases (without support for *recursion*). For *acyclic* conjunctive query evaluation, efficient algorithms such as the Yannakakis algorithm [61] have also been proposed to avoid computation redundancy. Nonetheless, these algorithms cannot be directly applied to CFL-reachability analysis. To diminish unnecessary computations, Graspan [21] utilizes a few data processing techniques from a novel "Big Data" perspective. Datalog engines such as Souffle [22], [62] adopt the *semi-naive evaluation* strategy. However, these general frameworks are unaware of the graph features (e.g., transitivity) and the program analysis workloads (e.g., Dyck-style productions), and thus still exhibit a substantial amount of derivation redundancy, as shown in previous work [23].

More recently, POCR [23] accelerates CFL-reachability solving by taming transitive redundancy via partially ordered CFL-reachability solving. As confirmed by both previous work [23] and our experiments, the grammar rewriting technique can also contribute to the reduction of redundancy due to transitivity. Different from existing techniques, our multi-derivation approach effectively reduces derivation redundancy by propagating reachability relations in

batches on sparse constraint graphs.

A class of set constraints and CFL-reachability were also shown to be interconvertible [27]. Later, a practical work [20] described a set constraint reduction for Dyck-CFL-reachability. In those works, a CFL-reachability instance is first transformed into a set constraint problem and is then solved by an off-the-shelf constraint solver. Unfortunately, constraint solvers are typically not specialized for CFL-reachability problems. While our work also solves CFL-reachability from a constraint-solving perspective, we argue that our constraint rules are considerably simpler and easier to follow. Moreover, we propose a general multi-derivation CFL-reachability framework, which can leverage the features of CFL-reachability problems to accelerate the CFL-reachability solving process, e.g., accelerating transitivity solving by employing the propagation graph representation, and optimizing the solving process of Dyck-style productions. We hope that our work can pave the way to further exploit the practical side of CFL-reachability solving.

7 Conclusion and Future Work

This paper has proposed PEARL, a *multi-derivation* approach via efficient constraint-solving, which reduces considerable derivation redundancy introduced by the conventional single-reachability derivation manner. Our key insight is that multiple edges can be simultaneously derived via batch propagation of reachability relations. We also introduce a simple yet effective propagation graph representation on top of the multi-derivation framework to further reduce the redundancy due to transitive relations, which are ubiquitous in CFL-based program analyses. This results in a highly efficient transitive-aware variant, PEARL^{PG}. We evaluate our approach by conducting extensive experiments on two popular clients, context-sensitive value-flow analysis and field-sensitive alias analysis for C/C++. The empirical results demonstrate that, by eliminating a large amount of derivation redundancy, our method outperforms existing approaches with promising speedups. In particular, the comparison with a state-of-the-art solver POCR (designed for fast transitivity solving), shows that PEARL^{PG} runs 3.09× (up to 4.44×) and 2.25× (up to 3.31×) faster than POCR on average, respectively for value-flow analysis and alias analysis, while also reducing memory consumption.

We also discuss the practical side of CFL-reachability analysis, such as optimization for Dyck-style productions. We hope that more opportunities can be explored on top of our multi-derivation framework.

Acknowledgment

We thank the reviewers for their constructive comments. This work is supported by the National Key R&D Program of China (2022YFB3103900), the National Natural Science Foundation of China (NSFC) under grant number 62132020 and 62202452, and the China Postdoctoral Science Foundation under grant number 2024M753295.

References

[1] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995, pp. 49–61.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[3] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay, "Speeding up Slicing," *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 5, pp. 11–20, 1994.

[4] M. Sridharan, S. J. Fink, and R. Bodik, "Thin Slicing," in *Proceedings of the 28th ACM SIGPLAN conference on programming language design and implementation*, 2007, pp. 112–122.

[5] Y. Li, T. Tan, Y. Zhang, and J. Xue, "Program Tailoring: Slicing by Sequential Criteria," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[6] T. Reps, "Program Analysis via Graph Reachability," *Information and software technology*, vol. 40, no. 11-12, pp. 701–726, 1998.

[7] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven Points-to Analysis for Java," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 59–76, 2005.

[8] M. Sridharan and R. Bodík, "Refinement-based Context-sensitive Points-to Analysis for Java," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 387–400, 2006.

[9] X. Zheng and R. Rugina, "Demand-driven Alias Analysis for C," in *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2008, pp. 197–208.

[10] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su, "Fast Algorithms for Dyck-CFL-reachability with Applications to Alias Analysis," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 435–446.

[11] G. Xu, A. Rountev, and M. Sridharan, "Scaling CFL-reachability-based Points-to Analysis using Context-sensitive Must-not-alias Analysis," in *ECOOP*, vol. 9. Springer, 2009, pp. 98–122.

[12] D. Yan, G. Xu, and A. Rountev, "Demand-driven Context-sensitive Alias Analysis for Java," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 155–165.

[13] L. Shang, X. Xie, and J. Xue, "On-demand Dynamic Summary-based Points-to Analysis," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012, pp. 264–274.

[14] Y. Su, D. Ye, and J. Xue, "Parallel Pointer Analysis with CFL-reachability," in *2014 43rd International Conference on Parallel Processing*. IEEE, 2014, pp. 451–460.

[15] M. Yannakakis, "Graph-theoretic Methods in Database Theory," in *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1990, pp. 230–242.

[16] S. Chaudhuri, "Subcubic Algorithms for Recursive State Machines," in *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2008, pp. 159–169.

[17] Y. Li, Q. Zhang, and T. Reps, "Fast Graph Simplification for Interleaved Dyck-reachability," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 780–793.

[18] ——, "Fast Graph Simplification for Interleaved-Dyck Reachability," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 44, no. 2, pp. 1–28, 2022.

[19] Y. Lei, Y. Sui, S. H. Tan, and Q. Zhang, "Recursive State Machine Guided Graph Folding for Context-free Language Reachability," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 318–342, 2023.

[20] J. Kodumal and A. Aiken, "The Set Constraint/CFL Reachability Connection in Practice," *ACM Sigplan Notices*, vol. 39, no. 6, pp. 207–218, 2004.

[21] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani, "Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 389–404, 2017.

[22] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On Synthesis of Program Analyzers," in *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*. Springer, 2016, pp. 422–430.

[23] Y. Lei, Y. Sui, S. Ding, and Q. Zhang, "Taming Transitive Redundancy for Context-free Language Reachability," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 1556–1582, 2022.

[24] Y. Sui, D. Ye, and J. Xue, "Static Memory Leak Detection using Full-sparse Value-flow Analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 254–264.

[25] ——, "Detecting Memory Leaks Statically with Full-sparse Value-flow Analysis," *IEEE Transactions on Software Engineering*, vol. 40, no. 2, pp. 107–122, 2014.

[26] C. Shi, H. Li, Y. Sui, J. Lu, L. Li, and J. Xue, "Two Birds with One Stone: Multi-Derivation for Fast Context-free Language Reachability Analysis," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2023, pp. 624–636.

[27] D. Melski and T. Reps, "Interconvertibility of a Class of Set Constraints and Context-free-language Reachability," *Theoretical Computer Science*, vol. 248, no. 1-2, pp. 29–98, 2000.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT press, 2022.

[29] C. Fecht and H. Seidl, "Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems," *Nord. J. Comput.*, vol. 5, no. 4, pp. 304–329, 1998.

[30] D. J. Pearce, P. H. Kelly, and C. Hankin, "Online Cycle Detection and Difference Propagation for Pointer Analysis," in *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 2003, pp. 3–12.

[31] M. Sridharan and S. J. Fink, "The Complexity of Andersen's Analysis in Practice," in *Static Analysis: 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings 16*. Springer, 2009, pp. 205–221.

[32] K. Chatterjee, B. Choudhary, and A. Pavlogiannis, "Optimal Dyck Reachability for Data-Dependence and Alias Analysis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–30, 2017.

[33] Q. Shi, Y. Wang, P. Yao, and C. Zhang, "Indexing the Extended Dyck-CFL Reachability for Context-sensitive Program Analysis," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 1438–1468, 2022.

[34] D. M. Moyles and G. L. Thompson, "An Algorithm for Finding a Minimum Equivalent Graph of a Digraph," *Journal of the ACM (JACM)*, vol. 16, no. 3, pp. 455–460, 1969.

[35] D. Gries, A. J. Martin, J. L. A. v. d. Snepscheut, and J. T. Udding, "An Algorithm for Transitive Reduction of an Acyclic Graph," *Sci. Comput. Program.*, vol. 12, no. 2, pp. 151–155, 1989.

[36] A. V. Aho, M. R. Garey, and J. D. Ullman, "The Transitive Reduction of a Directed Graph," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 131–137, 1972.

[37] R. Tarjan, "Depth-first Search and Linear Graph Algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[38] A. Rountev and S. Chandra, "Off-line Variable Substitution for Scaling Points-to analysis," *Acm Sigplan Notices*, vol. 35, no. 5, pp. 47–56, 2000.

[39] Y. Sui and J. Xue, "SVF: Interprocedural Static Value-flow Analysis in LLVM," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.

[40] G. F. Italiano, "Amortized Efficiency of a Path Retrieval Data Structure," *Theoretical Computer Science*, vol. 48, pp. 273–281, 1986.

[41] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis, "Analysis of Recursive State Machines," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 4, pp. 786–818, 2005.

[42] T. Reps, S. Schwoon, and S. Jha, "Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis," in *International Static Analysis Symposium*. Springer, 2003, pp. 189–213.

[43] T. Reps, S. Schwoon, S. Jha, and D. Melski, "Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis," *Science of Computer Programming*, vol. 58, no. 1-2, pp. 206–263, 2005.

[44] J. Späth, K. Ali, and E. Bodden, "Context-, Flow-, and Field-sensitive Data-flow Analysis using Synchronized Pushdown Systems," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[45] R. Alur and P. Madhusudan, "Visibly Pushdown Languages," in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, 2004, pp. 202–211.

[46] Y. Li, K. Satya, and Q. Zhang, "Efficient Algorithms for Dynamic Bidirected Dyck-reachability," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–29, 2022.

[47] N. A. Naeem, O. Lhoták, and J. Rodriguez, "Practical Extensions to the IFDS Algorithm," in *Compiler Construction: 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*. Springer, 2010, pp. 124–144.

[48] S. Arzt and E. Bodden, "Reviser: Efficiently Updating IDE-/IFDS-based Data-flow Analyses in Response to Incremental Program Changes," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 288–298.

[49] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven Flow-and Context-sensitive Pointer Analysis for Java," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[50] D. He, H. Li, L. Wang, H. Meng, H. Zheng, J. Liu, S. Hu, L. Li, and J. Xue, "Performance-boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 267–279.

[51] S. Arzt, "Sustainable Solving: Reducing the Memory Footprint of IFDS-based Data Flow Analyses using Intelligent Garbage Collection," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1098–1110.

[52] H. Li, H. Meng, H. Zheng, L. Cao, J. Lu, L. Li, and L. Gao, "Scaling up the IFDS Algorithm with Efficient Disk-assisted Computing," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 236–247.

[53] H. Li, J. Lu, H. Meng, L. Cao, L. Li, and L. Gao, "Boosting the Performance of Multi-solver IFDS Algorithms with Flow-Sensitivity Optimizations," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2024, pp. 296–307.

[54] Q. Zhang, X. Xiao, C. Zhang, H. Yuan, and Z. Su, "Efficient Subcubic Alias Analysis for C," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 829–845.

[55] L. Li, C. Cifuentes, and N. Keynes, "Precise and Scalable Context-sensitive Pointer Analysis via Value Flow Graph," *ACM SIGPLAN Notices*, vol. 48, no. 11, pp. 85–96, 2013.

[56] ——, "Boosting the Performance of Flow-sensitive Points-to Analysis using Value Flow," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 343–353.

[57] Y. Sui and J. Xue, "On-demand Strong Update Analysis via Value-flow Refinement," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 460–473.

[58] ——, "Value-flow-based Demand-driven Pointer Analysis for C and C++," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 812–835, 2018.

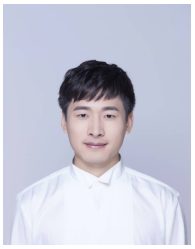
[59] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 693–706.

[60] D. Olteanu and M. Schleich, "Factorized Databases," *ACM SIGMOD Record*, vol. 45, no. 2, pp. 5–16, 2016.

[61] M. Yannakakis, "Algorithms for acyclic database schemes," in *VLDB*, vol. 81, 1981, pp. 82–94.

[62] M. Bravenboer and Y. Smaragdakis, "Strictly Declarative Specification of Sophisticated Points-to Analyses," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2009, pp. 243–262.

Chenghang Shi Chenghang Shi received his B.Eng degree in Software Engineering from Tongji University in 2021. He is currently pursuing his Ph.D. degree at Institute of Computing Technology, Chinese Academy of Sciences, and University of Chinese Academy of Sciences. His research interests are in the areas of programming languages and software engineering, specifically focusing on program analysis techniques.



Haofeng Li Haofeng Li received his BS degree from Shandong University in 2017 and his PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences in 2023. He is currently an assistant professor in the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include program analysis, pointer analysis and bug detection.

Yulei Sui Yulei Sui is an Associate Professor at School of Computer Science and Engineering, University of New South Wales (UNSW). He is broadly interested in Program Analysis, Secure Software Engineering and Machine Learning. In particular, his research focuses on building open-source frameworks for static analysis and verification techniques to improve the reliability and security of modern software systems. His recent interest lies at the intersection of programming languages, natural languages and code LLMs. Specifically, his current research projects include analysis and verification for software systems and AI models.

Jie Lu Jie Lu received his BS degree in Computer Science from Sichuan University in 2014 and his PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences in 2020. He is currently an associate professor in the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include program analysis, log analysis and distributed systems.

Lian Li Lian Li received his BSc degree in Engineering physics from Tsinghua University in 1998 and his PhD degree from University of New South Wales in 2007. He is currently a professor in the Institute of Computing Technology, Chinese Academy of Sciences, where he leads the Program Analysis Group.

Lian Li's main research interest focuses on program analysis, more specifically, on program analysis techniques and practical tools for improving software safety and security.

Jingling Xue (IEEE Fellow) Jingling Xue is a Scientia Professor at UNSW Sydney, leading the Programming Languages and Compilers group in the School of Computer Science and Engineering. He received his B.Eng and M.Eng degrees in Computer Science and Engineering from Tsinghua University in 1984 and 1987, respectively, and his PhD degree in Computer Science and Engineering from Edinburgh University in 1992.

His research focuses on programming languages, compiler technology, and program analysis. He has supervised 31 PhD students and shares his research outcomes through open-source tools like SVF (https://svf-tools.github.io/SVF), Qilin (https://qilinpta.github.io/Qilin), and RuPTA (https://rustanlys.github.io/rupta/). He has received multiple awards at CGO, ECOOP, ICSE, FSE, ASE, and ISSTA.

Jingling Xue has served as PC and GC Chair at LCTES'13, CC'18, CGO'20, and CGO'24, and as a PC member at over 200 international conferences.