Better Not Together: Staged Solving for Context-Free Language Reachability

Chenghang Shi*
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China

Haofeng Li*†
SKLP, Institute of Computing Technology, CAS
Beijing, China

Jie Lu*
SKLP, Institute of Computing Technology, CAS
Beijing, China

Lian Li*†
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Zhongguancun Laboratory
Beijing, China

Abstract

Context-free language reachability (CFL-reachability) is a fundamental formulation for program analysis with many applications. CFL-reachability analysis is computationally expensive, with a slightly subcubic time complexity concerning the number of nodes in the input graph.

This paper proposes staged solving: a new perspective on solving CFL-reachability. Our key observation is that the context-free grammar (CFG) of a CFL-based program analysis can be decomposed into (1) a smaller CFG, $\mathcal{L}$, for matching parentheses, such as procedure calls/returns, field stores/loads, and (2) a regular grammar, $\mathcal{R}$, capturing control/data flows. Instead of solving these two parts monolithically (as in standard algorithms), staged solving solves $\mathcal{L}$-reachability and $\mathcal{R}$-reachability in two distinct stages. In practice, $\mathcal{L}$-reachability, though still context-free, involves only a small subset of edges, while $\mathcal{R}$-reachability can be computed efficiently with close to quadratic complexity relative to the node size of the input graph. We implement our staged CFL-reachability solver, Stg, and evaluate it using two clients: context-sensitive value-flow analysis and field-sensitive alias analysis. The empirical results demonstrate that Stg achieves speedups of 861.59x and 4.1x for value-flow analysis and alias analysis on average, respectively, over the standard subcubic algorithm. Moreover, we also showcase that staged solving can help to significantly improve the performance of two state-of-the-art solvers, Pocr and Pearl, by 74.82x (1.78x) and 37.66x (1.7x) for value-flow (alias) analysis, respectively.

CCS Concepts

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

Keywords

CFL-rechability, Performance, Staged Analysis

ACM Reference Format:
Chenghang Shi, Haofeng Li, Jie Lu, and Lian Li. 2024. Better Not Together: Staged Solving for Context-Free Language Reachability. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680346

*{shichenghang21s, lihaofeng, lujie, lianli}@ict.ac.cn
†Corresponding author

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680346

1 Introduction

Context-free language reachability (CFL-reachability) serves as a fundamental framework for program analysis [17]. A large number of program analyses, such as pointer analysis [26, 28, 38], interprocedural dataflow analysis [2, 19], program slicing [20, 27], and shape analysis [17] can be formulated as CFL-reachability problems. A CFL-reachability instance consists of (1) an edge-labeled input graph G modeling the program under analysis, and (2) a context-free language L that captures the program properties to be analyzed. Two nodes are considered to be CFL-reachable if there exists a path between them, such that the word obtained by concatenating the labels of the edges on the path is a member of language L.

CFL-reachability analysis is time-intensive. The standard algorithm [15] has a cubic time complexity with respect to the number of nodes in the input graph G, and the complexity can be optimized to subcubic by leveraging fast set operations of bit vectors [5]. How to further reduce the time complexity of general CFL-reachability solving algorithms remains an open question [15]. Nevertheless, for certain CFL-rechability problems, the characteristics of their underlying context-free languages can be exploited to reduce the complexity further. For instance, when L is a regular language, the reachability problem can be solved in $O(mn)$ time [35], where m and n are the numbers of edges and nodes in the input labeled graph G, respectively. In practice, G tends to be sparse, with m being $O(n)$. Hence, for regular language reachability problems, the algorithm in [35] achieves greater efficiency compared to the state-of-the-art subcubic algorithm [5].

Insight. We observe that, for every CFL-based program analysis problem that we are aware of, the corresponding context-free grammar (CFG) comprises two components: (1) a context-free part modeling properly matched parentheses [9, 13], such as procedure

calls and returns [19, 32], field loads and stores [28, 34], or pointer references and dereferences [38], and (2) a regular part capturing control/data flow. In light of this observation, we propose a *staged solving* approach, Stg, to enhance the scalability of CFL-reachability solving. Specifically, Stg decomposes a given CFG into two grammars: a *smaller* CFG $\mathcal{L}$ and a regular grammar $\mathcal{R}$ (where "smaller" implies fewer edges involved). These two grammars are solved separately: $\mathcal{L}$-reachability, which is more complex, derives only a small subset of edges, while $\mathcal{R}$-reachability tackles the majority of the edges and can be computed efficiently in $O(mn)$ time, with $m = O(n)$ in practice. In contrast, the standard algorithm resolves all edges together, resulting in significantly lower efficiency.

Challenges. Two technical challenges arise in staged solving: (1) how to decompose a given CFG into $\mathcal{L}$ and $\mathcal{R}$? This is challenging due to the pervasive *mutual dependency* of productions, and (2) how to solve $\mathcal{L}$-reachability and $\mathcal{R}$-reachability efficiently? To address the first challenge, we introduce a pattern-based grammar rewriting technique. The *context-free pattern* (CFP) is introduced to capture the context-free aspect of a CFG and it is the key to enable staged solving. We identify two frequently encountered CFPs in CFL-based program analyses and apply them to a wide range of applications. To address the second challenge, we devise efficient algorithms for both $\mathcal{L}$- and $\mathcal{R}$-reachability, ultimately improving the effectiveness of CFL-reachability analysis.

We develop an efficient staged CFL-reachability solver, Stg, and apply it to two popular clients. The empirical results show that Stg achieves drastic speedups over state-of-the-art approaches [11, 23]. To sum up, this paper makes the following contributions:

- We propose staged solving, a novel approach to improve the scalability of CFL-reachability analysis. Given a CFL-reachability problem, staged solving aims to decompose the underlying CFG into two grammars: a smaller CFG $\mathcal{L}$ and a regular grammar $\mathcal{R}$, and solve them in two distinct phases.
- We introduce *context-free patterns* (CFPs) to capture the context-free aspect of a CFG, and propose a principled technique called *CFP-based grammar decomposition* to enable staged solving.
- We devise and implement efficient algorithms for solving both $\mathcal{L}$- and $\mathcal{R}$-reachability in staged solving.
- We realize our approach in an efficient solver, Stg, and apply it to context-sensitive value-flow analysis and field-sensitive alias analysis for C/C++. The experimental results demonstrate that staged solving can obtain average speedups of 861.59x and 4.1x for value-flow analysis and alias analysis, respectively, over the standard subcubic algorithm [5]. Furthermore, We also showcase that staged solving can improve the performance of two recent solvers, Pocr [11] and Pearl [23], with speedups of 74.82x (1.78x) and 37.66x (1.7x) for value-flow (alias) analysis, respectively.

The remainder of this paper is organized as follows: Section 2 provides an overview of CFL-reachability and illustrates our core idea using a motivating example. Section 3 elaborates on our staged solving approach, which is assessed in Section 4. Section 5 reviews related works, and Section 6 concludes this paper.

2 Motivation

In this section, we first review the background on CFL-reachability, and then describe the benefits and challenges of staged solving using an example of Dyck-CFL-reachability.

2.1 CFL-Reachability

A CFL-reachability instance consists of: (1) an edge-labeled graph $G = (V, E)$, where V and E are node set and edge set of G, respectively, and (2) a context-free grammar $CFG = (N, \Sigma, P, S)$. In the CFG, N is a finite set called the nonterminals, and Σ is a finite set, disjoint from N, called the terminals; P is a set of production rules, each of which is of the form $\alpha \to \beta$, where $\alpha \in N$ is a nonterminal and $\beta \in (N|\Sigma)^*$ is a string of nonterminals and/or terminals; $S \in N$ is the start variable. By convention, we use lowercase letters to denote terminals, and uppercase letters to represent nonterminals.

Each edge in G is labeled by a symbol $\alpha \in (N \cup \Sigma)$, e.g., $u \xrightarrow{\alpha} v$ denotes an edge from Node u to Node v labeled by α. For each path p in G, a word spelled by p is obtained by concatenating the edge labels along the path in order. A path is termed an *X-path* if its word can be derived from nonterminal X through one or more productions in P. Upon discovering an X-path from u to v during CFL-reachability solving, an X-edge $u \xrightarrow{X} v$ is incorporated into the edge-labeled graph. The standard algorithm [15] iteratively generates edges until a fixed point, by applying productions in P to already existing edges. For example, production $X \to Y\ Z$ derives $u \xrightarrow{X} w$ if there exist two edges $u \xrightarrow{Y} v$ and $v \xrightarrow{Z} w$ in E.

2.2 Motivating Example

We motivate our approach with an example of context-sensitive value flow analysis. Figure 1a and Figure 1b depict the code snippet and its corresponding edge-labeled graph (with an additional summary edge $b \xrightarrow{S} d$), respectively. Following [11], we utilize the Dyck language shown in Figure 1c to model context-sensitive value flow analysis [31]. Here the empty string ϵ represents a self-loop edge. An s-edge $u \xrightarrow{s} v$ signifies an assignment from variable u from v; $(_i$ indicates an assignment from an actual argument to a formal parameter at the i-th call site; $)_i$ denotes an assignment from a return value to its receiver at the i-th call site.

Initially, a self-loop edge $u \xrightarrow{\epsilon} u$ is introduced for every Node u in the input graph. For clarity, self-loop edges and S-edges derived from singleton s-edges are omitted in Figure 1b and Figure 1e.

Standard Algorithm. The standard algorithm generates new edges until it reaches a fixed point, where no additional edges can be derived. For instance, in Figure 1b, the S-edge $a \xrightarrow{S} c$ (omitted in the graph) is deduced from the input edge $a \xrightarrow{s} c$, and $b \xrightarrow{S} d$ is produced by the path $b \xrightarrow{(_i} a \xrightarrow{S} c \xrightarrow{)_i} d$. In addition to its (sub)cubic complexity, the standard algorithm suffers further from a performance loss caused by redundant edge processing. Consider the path $f \xrightarrow{S} b \xrightarrow{S} d \xrightarrow{S} e$ in Figure 1b, the edge $d \xrightarrow{S} e$ may be visited twice: once for $b \xrightarrow{S} d$ and once for $f \xrightarrow{S} d$ (derived by $f \xrightarrow{S} b \xrightarrow{S} d$). Note that such redundancy can be avoided if we follow the topological order and process the edge $d \xrightarrow{S} e$ after both

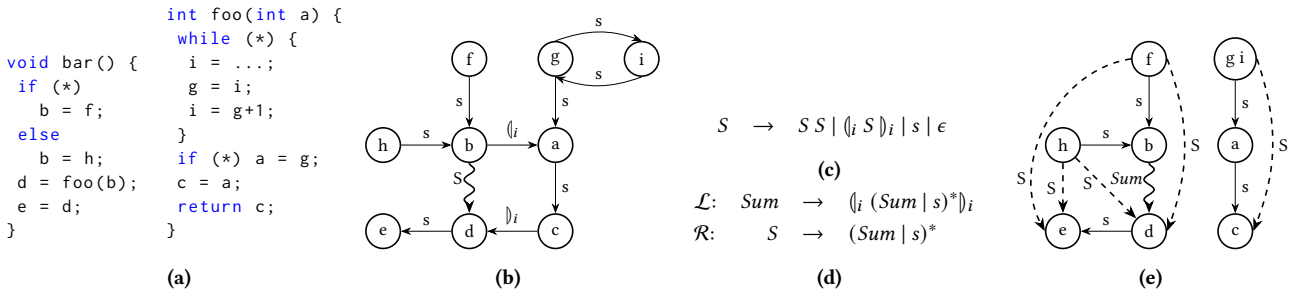

Figure 1: A motivating example. (a) The code snippet. (b) The corresponding input graph, with an additional summary edge $b \xrightarrow{S} d$. (c) The context-free grammar (CFG) of Dyck CFL [9]. (d) The decomposed CFG. (e) Staged solving resolves the *Sum*-edge in Phase$_L$ (the wavy line), and derives *S*-edges in Phase$_R$ (dash lines).

$f \xrightarrow{S} d$ and $b \xrightarrow{S} d$ have been generated. In this way, two edges can be processed in one set operation [5]. However, establishing a proper order is challenging due to dynamically introduced edges such as $b \xrightarrow{S} d$.

Staged Solving. Figure 1d and Figure 1e demonstrate the core idea of staged solving. Figure 1d illustrates the decomposition of the original CFG for Dyck CFL into two components: $\mathcal{L}$ and $\mathcal{R}$. $\mathcal{L}$ captures the context-free nature of the CFG ($\langle\!|_i\ S\ |\!\rangle_i$) to compute summary edges using the production $Sum \rightarrow \langle\!|_i\ (Sum \mid s)^*\ |\!\rangle_i$, and $\mathcal{R}$ is a regular grammar dedicated to computing all remaining reachability edges. $\mathcal{L}$-reachabiltiy and $\mathcal{R}$-reachability are separately addressed in Phase$_L$ and Phase$_R$, respectively. In Phase$_L$, a summary edge $b \xrightarrow{Sum} d$ (the wave line in Figure 1e) is generated. In Phase$_R$, all *S*-edges (the dashed lines in Figure 1e) are derived by efficiently solving a regular language reachability problem.

Compared to the standard algorithm, staged solving offers the following two advantages:
(1) Reduced Complexity. While $\mathcal{L}$ in Figure 1d remains context-free, it focuses solely on a narrow subset of edges (input edges and $b \xrightarrow{Sum} d$ in Figure 1e). The majority of edges are computed during Phase$_R$ (dashed lines in Figure 1e) by solving $\mathcal{R}$-reachability in $O(mn)$ time, where the edge size m is typically linear to the node size n in practice. Consequently, staged solving lowers complexity and drastically improves performance over the standard algorithm.
(2) Reduced Redundancy. The decomposition of the grammar in Figure 1c to that in Figure 1d ensures the prioritization of *Sum*-edges (addressed in Phase$_L$) over *S*-edges (addressed in Phase$_R$). For instance, in Figure 1e, the *Sum*-edge $b \xrightarrow{Sum} d$ is generated before computing *S*-edges in Phase$_R$. Additionally, a topological order can be established with strongly connected components collapsed (e.g., Node *g* and Node *i* are consolidated as one node in Figure 1e) when solving $\mathcal{R}$-reachability. As a result, we can process nodes in the topological ordering of $(f, h, b, d, e, \{gi\}, a, c)$, thereby ensuring that each edge is processed only once and eliminating redundancy.

2.3 Problem Statement

Staged solving shares some similarities with bottom-up interprocedural analysis [6, 16]: in staged solving, the concept of a function summary is generalized to summarize certain reachable paths (e.g., paths derived by the production $S \rightarrow \langle\!|_i\ S\ |\!\rangle_i$ in Figure 1c). An immediate question arises: which paths need to be summarized? We address the question by summarizing paths derived using context-free productions, so that the remaining paths can be expressed in a regular grammar. This involves introducing new nonterminals for context-free productions and, subsequently, rewriting the grammar to decompose it into a context-free part and a regular part.

Given a CFL-reachability analysis problem, staged solving aims to address the following two challenges.

> **Challenge 1**: *Decompose the input CFG into a context-free grammar $\mathcal{L}$ (for summary edges) and a regular grammar $\mathcal{R}$ (for remaining edges), such that $\mathcal{L}$ does not depend on $\mathcal{R}$, which means $\mathcal{L}$-reachability can be solved without concerning $\mathcal{R}$-reachability.*
> **Challenge 2**: *Solve $\mathcal{L}$- and $\mathcal{R}$-reachability in sequence in two distinct phases, and develop efficient algorithms for both phases.*

Our Solution. Determining how to resolve the inherent mutual dependency among different productions, such as $S \rightarrow S\ S$ and $S \rightarrow \langle\!|_i\ S\ |\!\rangle_i$ in Figure 1c, is crucial for addressing challenge 1. In response, we propose a principled, pattern-based grammar rewriting technique (Section 3). To tackle challenge 2, we develop efficient algorithms that incorporate practical optimizations, such as ordered propagation, to solve both $\mathcal{L}$- and $\mathcal{R}$-reachability.

3 Methodology

In Section 3.1, we present the key idea of staged solving, accompanied by the concept of *context-free patterns* (CFPs) and *CFP-based grammar decomposition*. We then introduce two frequently encountered CFPs in CFL-based program analyses, respectively in Section 3.2 and Section 3.3, along with practical applications and optimized algorithms for fast $\mathcal{L}$-reachability solving. At last, we propose an efficient algorithm to compute $\mathcal{R}$-reachability in Section 3.4.

3.1 CFP-Based Grammar Decomposition

We introduce a *pattern-based* approach for grammar decomposition. The approach is applicable to all CFL-based program analyses, to our knowledge.

3.1.1 Context-Free Pattern. We observe that, for all CFL-based program analyses we are aware of, the context-free aspect of a CFG manifests as specific patterns, called *context-free patterns*.

$$
\begin{array}{rcl}
Start & \rightarrow & P\ N \\
P & \rightarrow & S\ P \mid \rangle_i\ P \mid \epsilon \\
N & \rightarrow & S\ N \mid \langle_i\ N \mid \epsilon \\
S & \rightarrow & S\ S \mid \langle_i\ S\ \rangle_i \mid s \mid \epsilon
\end{array}
$$

(a) The original CFG.

$$
\begin{array}{rrcl}
\mathcal{L}: & Sum & \rightarrow & \langle_i\ S\ \rangle_i \\
\mathcal{R}: & Start & \rightarrow & (Sum \mid s \mid \rangle_i)^*(Sum \mid s \mid \langle_i)^* \\
 & P & \rightarrow & (Sum \mid s \mid \rangle_i)^* \\
 & N & \rightarrow & (Sum \mid s \mid \langle_i)^* \\
 & S & \rightarrow & (Sum \mid s)^*
\end{array}
$$

(b) After rewriting $P_{\mathcal{R}}$.

$$
\begin{array}{rrcl}
\mathcal{L}: & Sum & \rightarrow & \langle_i\ (Sum \mid s)^*\ \rangle_i \\
\mathcal{R}: & Start & \rightarrow & (Sum \mid s \mid \rangle_i)^*\ (Sum \mid s \mid \langle_i)^* \\
 & P & \rightarrow & (Sum \mid s \mid \rangle_i)^* \\
 & N & \rightarrow & (Sum \mid s \mid \langle_i)^* \\
 & S & \rightarrow & (Sum \mid s)^*
\end{array}
$$

(c) The decomposed CFG (after rewriting $P_{\mathcal{L}}$).

Figure 2: The context-free grammar (CFG) for Extended Dyck-CFL-reachability [9], *Start* is the start variable. In (c), P, N, and S in $\mathcal{R}$ can be removed because *Start* does not depend on them.

Definition 1. (Context-Free Pattern). Given a $CFG = (N, \Sigma, P, S)$, let $CFP = (S', a, E, b)$ be a *context-free pattern*, where S' is a nonterminal representing *summary edges*, a and b denote a pair of terminals, and E is a regular expression over $N \cup \Sigma$. A context-free pattern is explicitly written as $S' \rightarrow a\ E\ b$.

In essence, a CFP captures the context-free aspect of a CFL-based program analysis by matching terminals a and b (e.g., calls and returns [9], field writes and reads [28]), while modelling transitive control/data flow with expression E. Despite its simplicity, the formulation of a context-free pattern (CFP) here can cover a wide range of CFL-based program analysis problems, including, but not limited to shape analysis [17], slicing [27], interprocedural dataflow analysis [19] and pointer analysis [28, 38].

Observation 1. After resolving reachability related to CFPs in a CFG, we can reformulate the corresponding CFL-reachability problem as a regular language reachability problem.

Staged solving is motivated by Observation 1. Given a CFG, its CFPs can be separately reformulated as a smaller CFG $\mathcal{L}$, making the remaining part a regular grammar $\mathcal{R}$. Unfortunately, it is not straightfoward to decompose an input CFG due to the inherent *mutual dependency between its productions*. For instance, given a CFP $S' \rightarrow a\ E\ b$ where E contain another nonterminal X, S'-reachability and X-reachability may be mutually dependent. How to address such mutual dependency is the key enabling technique for staged solving.

3.1.2 CFP-Based Grammar Decomposition. To tackle mutual dependency, we propose a principled technique called *CFP-based grammar decomposition*. Essentially, given $CFG = (N, \Sigma, P, S)$ and a CFP instance $S' \rightarrow a\ E_c\ b$ (E_c is a concrete string that follows the regular pattern described by the CFP), grammar decomposition consists of the following steps:

(1) *Introduce the production $S' \rightarrow a\ E_c\ b$.* The nonterminal S' is introduced if $S' \notin N$, to summarize paths with sequence $a\ E_c\ b$. This step conceptually divides the production set P of CFG into two parts: the context-free portion $P_{\mathcal{L}}$ containing the newly introduced production for S', and the regular portion $P_{\mathcal{R}}$ consisting of the remaining productions. Despite this separation, mutual dependency persists between productions in $P_{\mathcal{L}}$ and $P_{\mathcal{R}}$.
(2) *Rewrite productions in $P_{\mathcal{R}}$.* In this step, we rewrite each production p in $P_{\mathcal{R}}$, by firstly replacing the pattern $a\ E_c\ b$ to S'. Under Observation 1, this replacement reduces $P_{\mathcal{R}}$ to regular productions. Hence, we reformulate the right-hand size of each production in $P_{\mathcal{R}}$ into a regular expression over $\Sigma \cup \{S'\}$, using standard grammar transformation rules.
(3) *Rewrite production of S' in $P_{\mathcal{L}}$.* For any nonterminal X occurs in production of S' where $X \neq S'$ and $X \rightarrow Exp \in P_{\mathcal{R}}$, we replace X with Exp, which is a regular expression over $\Sigma \cup \{S'\}$ after rewriting $P_{\mathcal{R}}$ (step 2). As a result, the production of S' now contains only symbols in $\Sigma \cup \{S'\}$, effectively eliminating dependency from $P_{\mathcal{L}}$ to $P_{\mathcal{R}}$, which is pivotal for staged solving.
(4) Decompose CFG. The grammar after steps 1-3, $CFG' = (N \cup \{S'\}, \Sigma, P_{\mathcal{L}} \cup P_{\mathcal{R}}, S)$, is decomposed into $\mathcal{L}$ and $\mathcal{R}$, such that (1) $\mathcal{L} = (\{S'\}, \Sigma, P_{\mathcal{L}}, S')$ resolves the input CFP; and (2) $\mathcal{R} = (N \setminus \{S'\}, \Sigma \cup \{S'\}, P_{\mathcal{R}}, S)$ is a regular grammar where S' serves as a terminal in $\mathcal{R}$.

The above grammar decomposition can be easily extend to the general cases with multiple CFP instances by introducing a production for each CFP instance in $P_{\mathcal{L}}$.

3.1.3 Soundness. This subsection discusses the soundness of staged solving.

Lemma 1. (Grammar Equivalence). $CFG = (N, \Sigma, P, S)$ is equivalent to $CFG' = (N \cup \{S'\}, \Sigma, P_{\mathcal{L}} \cup P_{\mathcal{R}}, S)$, which is the CFG after steps 1-3 in CFP-based decomposition.

Proof Sketch. Grammar decomposition employs standard transformation rules, which apply only algebraic substitution to existing productions. Therefore, P is *semantically equivalent* to the union of $P_{\mathcal{L}}$ and $P_{\mathcal{R}}$, i.e., $P_{\mathcal{L}} \cup P_{\mathcal{R}}$. Thus, both grammars generate the same language and produce the same set of S-edges. □

Lemma 2. (Separate Solving). $\mathcal{L}$-reachability can be resolved soundly independent of $\mathcal{R}$-reachability.

Proof Sketch. Productions in $P_{\mathcal{L}}$ does not contain any nonterminals of $\mathcal{R}$. Thus, $\mathcal{R}$ does not produce any edges that contributes to $\mathcal{L}$-reachability. □

Theorem 1. Staged solving soundly solves CFL-reachability.

Proof Sketch. According to Lemma 2, $\mathcal{L}$-reachability can be solved without concerning $\mathcal{R}$-reachability. Hence, by addressing $P_{\mathcal{L}}$ first and then $P_{\mathcal{R}}$, staged solving soundly resolves $P_{\mathcal{L}} \cup P_{\mathcal{R}}$ (corresponding to CFG'). Due to Lemma 1, we conclude that staged solving correctly solves CFL-reachability defined by CFG. □

3.2 The Dyck CFP

The *Dyck CFP* is at the core of Dyck-CFL reachability and takes the form $Sum \rightarrow a\ S^*\ b$. Although this CFP seems simple, it enables

$$
\begin{array}{lcl}
V & \to & \overline{A}^*\ (\overline{f_i}\ V\ f_i \mid M?)\ A^* \\
M & \to & \overline{d}\ V\ d \\
A & \to & a\ M? \\
\overline{A} & \to & M?\ \overline{a}
\end{array}
$$

(a) The original CFG.

$$
\begin{array}{lcll}
\mathcal{L}: & V' & \to & \overline{f_i}\ V\ f_i \\
 & M & \to & \overline{d}\ V\ d \\
\mathcal{R}: & V & \to & (M?\ \overline{a})^*\ (V' \mid M?)\ (a\ M?)^* \\
 & A & \to & a\ M? \\
 & \overline{A} & \to & M?\ \overline{a}
\end{array}
$$

(b) After rewriting $P_{\mathcal{R}}$.

$$
\begin{array}{lcll}
\mathcal{L}: & V' & \to & \overline{f_i}\ (M?\ \overline{a})^*\ (V' \mid M?)\ (a\ M?)^*\ f_i \\
 & M & \to & \overline{d}\ (M?\ \overline{a})^*\ (V' \mid M?)\ (a\ M?)^*\ d \\
\mathcal{R}: & V & \to & (M?\ \overline{a})^*\ (V' \mid M?)\ (a\ M?)^* \\
 & A & \to & a\ M? \\
 & \overline{A} & \to & M?\ \overline{a}
\end{array}
$$

(c) The decomposed CFG (after rewriting $P_{\mathcal{L}}$).

Figure 3: The context-free grammar (CFG) for alias analysis [38]. V or M is the start variable depending on the need of a client. In (c), both productions of V' and M generate summary edges. When M is the start variable, $\mathcal{R}$ can be safely ignore.

staged solving for two significant CFL instances: standard Dyck-CFL-reachability [9] and extended Dyck-CFL-reachability [9, 24].

3.2.1 The Standard Dyck-CFL. Standard Dyck-CFL reachability has been extensively studied [9, 13] and applied to a wide range of applications, including data dependence analysis [4] and unification-based alias analysis for Java [34, 36]. Dyck-CFL reachability restricts the underlying CFL to a Dyck language, which generates properly matched parentheses. A Dyck language is defined by the context-free grammar (CFG) in Figure 1c.

Grammar Decomposition for Standard Dyck-CFL. Grammar decomposition (see Section 3.1.2) can be performed as follows:

(1) Introduce the production $Sum \to (_i\ S\ )_i$ for Dyck CFP. Consequently, we obtain two sets of productions in the input CFG: $P_{\mathcal{L}} = \{\ Sum \to (_i\ S\ )_i\ \}$, and $P_{\mathcal{R}} = \{\ S \to S\ S \mid (_i\ S\ )_i \mid s \mid \epsilon\ \}$.
(2) Rewrite productions in $P_{\mathcal{R}}$:
$$S \to S\ S \mid (_i\ S\ )_i \mid s \mid \epsilon \to S\ S \mid Sum \mid s \mid \epsilon \to (Sum \mid s)^*$$
(3) Rewrite the production $Sum \to (_i\ S\ )_i$ in $P_{\mathcal{L}}$ by replacing S with sequence $(Sum \mid s)^*$. Thus, the dependency from Sum to S is severed.
$$Sum \to (_i\ S\ )_i \to (_i\ (Sum \mid s)^*\ )_i$$
(4) Decompose CFG. Figure 1d shows $\mathcal{L}$ and $\mathcal{R}$ after decomposition.

Staged Solving for Standard Dyck-CFL. As shown in Figure 1d, $\mathcal{L}$-reachability is firstly solved in Phase_L, which computes summary edges (Sum-edges). Next, in Phase_R, $\mathcal{R}$-reachability is solved efficiently by concatenating 0 or more Sum- and/or s-edges to produce S-edges. That is, $\mathcal{R}$-reachability operates on the input graph with all Sum-edges generated in Phase_L.

3.2.2 The Extended Dyck-CFL. The *extended Dyck-CFL* [9] generalizes the standard Dyck-CFL to recognize *partially* matched parentheses. This extension is desirable for modeling interprocedural flow paths whose source and destination nodes are located in distinct methods. In Figure 1b, the path $g \xrightarrow{s} a \xrightarrow{s} c \xrightarrow{)_i} d$ is such an example. Figure 2a defines the grammar for the extended Dyck-CFL. It is worth pointing out that the extended Dyck-CFL reachability is much more time-consuming compared to the standard Dyck-CFL reachability [24].

Grammar Decomposition for Extended Dyck-CFL. Grammar decomposition is performed as follows:

(1) Introduce the production $Sum \to (_i\ S\ )_i$ for Dyck CFP.
(2) Rewrite $P_{\mathcal{R}}$. Same as in Dyck-CFL, we reformulate the production of S to $S \to (Sum \mid s)^*$. Next, the productions of P and N are rewritten by replacing the nonterminal S with the regular expression $(Sum \mid s)^*$, as follows:
$$
\begin{array}{lcll}
P & \to & S\ P \mid)_i\ P \mid \epsilon & \to (S \mid)_i)\ P \mid \epsilon \\
 & \to & (S \mid)_i)^* & \to ((Sum \mid s)^* \mid)_i)^* \\
 & \to & (Sum \mid s \mid)_i)^* & \\
N & \to & S\ N \mid (_i\ N \mid \epsilon & \to (S \mid (_i)\ N \mid \epsilon \\
 & \to & (S \mid (_i)^* & \to ((Sum \mid s)^* \mid (_i)^* \\
 & \to & (Sum \mid s \mid (_i)^* &
\end{array}
$$
Finally, we rewrite the production of *Start* by replacing the nonterminals P and N with corresponding regular expressions: $Start \to (Sum \mid s \mid)_i)^*\ (Sum \mid s \mid (_i)^*$.
(3) Rewrite $P_{\mathcal{L}}$. Same as in Dyck-CFL, this step results in the production $Sum \to (_i\ (Sum \mid s)^*\ )_i$.
(4) Decompose CFG. Figure 2c shows the two decomposed grammars $\mathcal{L}$ and $\mathcal{R}$.

Staged Solving for Extended Dyck-CFL. For the extended Dyck-CFL, $\mathcal{L}$-reachability is computed in the same manner as in the standard Dyck-CFL. However, its $\mathcal{R}$ grammar, depicted in Figure 2c, is more complex. It is noteworthy that the production for the start variable (*Start*) contains only terminal symbols (Sum and s), making the productions of P, N, and S unnecessary for computing $\mathcal{R}$-reachability. Although these productions are no longer needed for solving $\mathcal{R}$-reachability, they are still included for the sake of completeness.

3.2.3 Solving $\mathcal{L}$-Reachability for Dyck CFP. We employ the classic tabulation-based algorithm [19, 20] to solve $\mathcal{L}$-Reachability for Dyck CFP. Here we briefly demonstrate the tabulation-based algorithm using the example in Figure 1b, with the algorithm's details described in [20]. The algorithm deduces the summary edge $b \xrightarrow{Sum} d$ in the following three steps: (1) initialize a self-loop path edge $a \dashrightarrow a$; (2) extend path edges by incorporating s- and Sum-edges until we have $a \dashrightarrow c$; (3) match two parentheses represented by edges $b \xrightarrow{(_i} a$ and $c \xrightarrow{)_i} d$ to obtain the summary edge. Note that a self-loop path edge $u \dashrightarrow u$ is initialized only when there is an incoming edge $w \xrightarrow{(_i} u$.

3.3 The Alias CFP

The Alias CFP is in the form of $X \to a\ A^*\ Y\ B^*\ b$. This pattern is widely used to model structure-transmitted data dependence [18, 28, 38], which is crucial for the formulation of alias analysis [11, 38] and points-to analysis [26, 28]. In Alias CFP, the two terminals a and b emulate the structure-transmitted behaviors, such as pointer reference and deference, as well as field writes and

reads. Meanwhile, the expression $A^*\ Y\ B^*$ models alias relations. Specifically, the nonterminal Y signifies the pointer source (e.g., heap objects allocated by *new* statements), and the *Kleene closures* A^* and B^* represent the transitive flow of the pointer source in different directions. Consequently, the sequence $A^*\ Y\ B^*$ connects two aliased pointer variables.

The Alias CFP rests on the matching of a and b. In the standard algorithm [15], all sequences in the form of $A^*\ Y\ B^*$ are resolved to derive X-edges. However, only those sequences that follow a terminal a and precede a terminal b can derive X-edges, which typically represent only a small subset of such sequences. Furthermore, deriving the sequence $A^*\ Y\ B^*$ easily introduce transitive redundancy [11], which may greatly impact overall performance. Therefore, our staged solving approach takes advantage of this pattern to compute all X-edges from the smallest possible set of sequences of the form $A^*\ Y\ B^*$, resulting in an optimized $\mathcal{L}$-reachability solver for this CFP (Algorithm 1). It should be noted that the derivation of A-, Y-, and B-edges generally depend on the resolution of X-edges, which leads to mutual dependency.

Next, we first demonstrate how Alias CFP can be exploited to enable staged solving for two popular client analyses: alias analysis (Section 3.3.1) and points-to analysis (Section 3.3.2). We then propose an efficient $\mathcal{L}$-reachability solver for this CFP (Section 3.3.3).

3.3.1 Alias Analysis. Figure 3a gives the CFG, as formulated in [38], for alias analysis of C programs. In Figure 3a, terminals d denotes pointer deference, f_i indicates reference to the i-th field, and a represents a direct assignment (i.e., a = b). The nonterminals V, M and A model *value alias*, *memory alias* and value flow, respectively. Alias analysis is conducted on a bidirected program expression graph (PEG) [17, 36], where an $\overline{X}$-edge $u \xrightarrow{\overline{X}} v$ corresponds to an X-edge $v \xrightarrow{X} u$. Thus, we introduce three more terminals $\overline{d}$, $\overline{f_i}$, and $\overline{a}$ and a nonterminal $\overline{A}$. Since both value alias (V) and memory alias (M) are symmetric, there is no need to introduce $\overline{V}$ and $\overline{M}$.

Grammar Decomposition for Alias Analysis. Now, let us examine how to solve CFL-based alias analysis in stages. The CFG is decomposed by rewriting the two patterns $\overline{f_i}\ V\ f_i$ (in production of V) and $\overline{d}\ V\ d$ (in production of M). Grammar decomposition is performed as follows:

(1) Introduce productions for two Alias CFPs:
$$V' \rightarrow \overline{f_i}\ V\ f_i \ (V' \text{ is a new nonterminal})$$
$$M \rightarrow \overline{d}\ V\ d$$
(2) Rewrite $P_{\mathcal{R}}$:
$$V \rightarrow \overline{A}^*\ (\overline{f_i}\ V\ f_i \mid M?)\ A^* \rightarrow \overline{A}^*\ (V' \mid M?)\ A^*$$
$$\rightarrow (M?\ \overline{a})^*\ (V' \mid M?)\ (a\ M?)^*$$
(3) Rewrite $P_{\mathcal{L}}$:
$$V' \rightarrow \overline{f_i}\ V\ f_i \rightarrow \overline{f_i}\ (M?\ \overline{a})^*\ (V' \mid M?)\ (a\ M?)^*\ f_i$$
$$M \rightarrow \overline{d}\ V\ d \rightarrow \overline{d}\ (M?\ \overline{a})^*\ (V' \mid M?)\ (a\ M?)^*\ d$$
(4) Decompose grammar, see Figure 3c.

Stage Solving for Alias Analysis. $\mathcal{L}$ has two intricate and mutually dependent productions for V' and M, respectively. To resolve V'- and M-edges, the standard algorithm needs to derive all such sequences $(M?\ \overline{a})^*\ (V' \mid M?)\ (a\ M?)^*$. As discussed previously, this straightforward approach suffers from a large number of useless sequences since M/V'-edges can only be derived from those sequences surrounded by a matched pair of terminals $\overline{d}/\overline{f_i}$ (on the left-hand side) and d/f_i (on the right-hand side), respectively. An intuitive solution to avoid these unnecessary sequences is to adapt the classic tabulation algorithm [20] for Alias CFP. However, it suffers from transitive redundancy [11]. Hence, in stage solving, we develop a dedicated L-reachability solver for Alias CFP, detailed in Algorithm 1 (Section 3.3.3), to tackle this problem.

For some clients, Phase_{R} can be skipped. For instance, points-to analysis in [38] only concerns M-paths, making $\mathcal{R}$ unnecessary.

3.3.2 Points-To Analysis. Let us study a CFL that formulates points-to analysis for Java [28]. The underlying CFG, reproduced from [28] with some notational changes, is given in Figure 4a. In this CFG, three terminals a, put_f, and get_f denote assignment, field write, and field read, respectively. Production of FT in Figure 4a suggests that an object can *flow to* a variable via zero or many direct (a) or indirect ($put_f\ Alias\ get_f$) assignments.

The nonterminal *Alias* is introduced for the alias relation. An *Alias* path connects two aliased variables p_1 and p_2, when a heap object O flows to both variables, i.e., $O \xrightarrow{FT} p_1 \wedge O \xrightarrow{FT} p_2$. Inverse edges are introduced to connect p_1 and p_2 [17] via the CFL-reachable path $p_1 \xrightarrow{\overline{FT}} O \xrightarrow{FT} p_2$. Interestingly, $\overline{FT}$, as the inverse relation of FT, denotes a points-to relation and serves as the start variable of this CFG. To build $\overline{FT}$-paths, we introduce four inverse terminals: $\overline{new}$, $\overline{a}$, $\overline{put_f}$ and $\overline{get_f}$.

Grammar Decomposition for Points-To Analysis. The context free ingredients of this CFG reflect on checking put_f and get_f edges, which are the balanced parentheses for field-sensitivity. The grammar is decomposed as follows:

(1) Introduce productions for A and $\overline{A}$. Here we include $A \rightarrow a$ ($\overline{A} \rightarrow \overline{a}$) only for clarity of presentation.
$$A \rightarrow a \mid put_f\ Alias\ get_f$$
$$\overline{A} \rightarrow \overline{a} \mid \overline{get_f}\ Alias\ \overline{put_f}$$
(2) Rewrite $P_{\mathcal{R}}$:
$$FT \rightarrow new\ (a \mid put_f\ Alias\ get_f)^* \rightarrow new\ A^*$$
$$\overline{FT} \rightarrow (\overline{a} \mid \overline{get_f}\ Alias\ \overline{put_f})^*\ \overline{new} \rightarrow \overline{A}^*\ \overline{new}$$
$$Alias \rightarrow \overline{FT}\ FT \rightarrow \overline{A}^*\ \overline{new}\ new\ A^*$$
(3) Rewrite $P_{\mathcal{L}}$:
$$A \rightarrow a \mid put_f\ Alias\ get_f \rightarrow a \mid put_f\ \overline{A}^*\ \overline{new}\ new\ A^*\ get_f$$
$$\overline{A} \rightarrow \overline{a} \mid \overline{get_f}\ Alias\ \overline{put_f} \rightarrow \overline{a} \mid \overline{get_f}\ \overline{A}^*\ \overline{new}\ new\ A^*\ \overline{put_f}$$
(4) Decompose CFG. See Figure 4c.

Stage Solving for Points-To Analysis. Similar to the process in Alias analysis, $\mathcal{L}$-reachability is solved by Algorithm 1, which optimizes transitive redundancy and computes only the necessary sequences $\overline{A}^*\overline{new}newA^*$ that follow a put_f ($\overline{get_f}$) and precede a get_f ($\overline{put_f}$) to derive A ($\overline{A}$) relations. In solving $\mathcal{R}$-reachability, since $\overline{FT}$ is the start variable, irrelevant productions of FT and *Alias* can be eliminated from $\mathcal{R}$ because $\overline{FT}$ does not depend on them. Thus, grammar decomposition not only eliminates dependencies between $\mathcal{L}$ and $\mathcal{R}$ (e.g., from A to *Alias*), but also within $\mathcal{R}$ itself (e.g., from $\overline{FT}$ to *Alias*).

3.3.3 Solving $\mathcal{L}$-Reachability for Alias CFP. We devise Algorithm 1 to efficiently solve $\mathcal{L}$-reachability for the Alias CFP, which consists of productions in the form of $X \rightarrow a\ A^*\ Y\ B^*\ b$. In a nut shell, the

$$
\begin{array}{rcl}
\overline{FT} & \rightarrow & (\overline{a} \mid \overline{get_f}\ Alias\ \overline{put_f})^*\ \overline{new} \\
FT & \rightarrow & new\ (a \mid put_f\ Alias\ get_f)^* \\
Alias & \rightarrow & \overline{FT}\ FT
\end{array}
$$

(a) The original CFG.

$$
\begin{array}{rcrcl}
\mathcal{L}: & A & \rightarrow & a \mid put_f\ Alias\ get_f \\
 & \overline{A} & \rightarrow & \overline{a} \mid \overline{get_f}\ Alias\ \overline{put_f} \\
\mathcal{R}: & \overline{FT} & \rightarrow & \overline{A}^*\ \overline{new} \\
 & FT & \rightarrow & new\ A^* \\
 & Alias & \rightarrow & \overline{A}^*\ \overline{new}\ new\ A^*
\end{array}
$$

(b) After rewriting $P_{\mathcal{R}}$.

$$
\begin{array}{rcrcl}
\mathcal{L}: & A & \rightarrow & a \mid put_f\ \overline{A}^*\ \overline{new}\ new\ A^*\ get_f \\
 & \overline{A} & \rightarrow & \overline{a} \mid \overline{get_f}\ \overline{A}^*\ \overline{new}\ new\ A^*\ \overline{put_f} \\
\mathcal{R}: & \overline{FT} & \rightarrow & \overline{A}^*\ \overline{new} \\
 & FT & \rightarrow & new\ A^* \\
 & Alias & \rightarrow & \overline{A}^*\ \overline{new}\ new\ A^*
\end{array}
$$

(c) The decomposed CFG (after rewriting $P_{\mathcal{L}}$).

Figure 4: The context-free grammar (CFG) for Java's points-to analysis [28]. The start variable $\overline{FT}$ denotes points-to relations (the reverse version of FT). In (c), FT and $Alias$ can be eliminated because $\overline{FT}$ does not depend on them.

Algorithm 1: Solving $\mathcal{L}$-reachability based on the CFP $X \rightarrow a\ A^*\ Y\ B^*\ b$

```
1  Procedure SolveLReach():
2      initialize A-, Y-, and B-edges;
3      add all Y-edges to FWL;
4      while FWL ≠ ∅ or BWL ≠ ∅ do
5          while FWL ≠ ∅ do
6              pop u → v from FWL;
7              if there exists some b-edge v -b-> w then
8                  add v → u to BWL;
9              for each v -B-> w do
10                 if u → w ∉ FEdges then
11                     add u → w to FWL and FEdges;
12         while BWL ≠ ∅ do
13             pop u → v from BWL;
14             for each x -a-> v do
15                 for each u -b-> y do
16                     add x -X-> y to SumEdges;
17             for each v -Ā-> w do
18                 if u → w ∉ BEdges then
19                     add u → w to BWL and BEdges;
20         generate A-, Y-, and B-edges based on new X-edges;
21         add all new Y-edges to FWL;
22         for each new B-edge u -B-> v do
23             for each w → u ∈ FEdges do
24                 if w → v ∉ FEdges then
25                     add w → v to FWL and FEdges;
26         for each new Ā-edge u -Ā-> v do
27             for each w → u ∈ BEdges do
28                 if w → v ∉ BEdges then
29                     add w → v to BWL and BEdges;
```

algorithm extends a Y-path through forward B-edges, and upon encountering a b-edge, it begins to traverse backward via $\overline{A}$-edges until it reaches an a-edges. Thus, an X-edge is deduced.

Solving Algorithm. We classify the *path edges* (edge sequences) being processed into two categories: *forward path edges* and *backward path edges*. A forward path edge $u \dashrightarrow v$ represents a path from u to v with the label sequence $Y\ B^*$, and a backward path edge $v \dashleftarrow u$ indicates the existence of a reverse path from v to u with the label sequence $A^*\ Y\ B^*$ preceding a b-edge $u \xrightarrow{b} w$.

In Algorithm 1, FWL ($FEdges$) and BWL ($BEdges$) denote worklists (edge sets) of forward and backward path edges, respectively. Lines 2-3 initialize A-, Y-, and B-edges with all Y-edges added to FWL. In Lines 5-11, forward path edges are propagated along B-edges, and a backward path edge is generated when a forward edge meets a b-edge (Lines 7 and 8). In Lines 12-19, we propagate backward path edges via $\overline{A}$-edges, and produce summary edges (X-edges) by matching a-edges with b-edges. Due to mutual dependency, newly introduced X-edges may generate new A-, Y-, and B-edges (Line 20), which are handled by Lines 21, 22-25, and 26-29, respectively.

Example. We use a points-to analysis example to illustrate how Algorithm 1 solves the production $A \rightarrow put_f\ \overline{A}^*\ \overline{new}\ new\ A^*\ get_f$. In Figure 5b, G_0 is abstracted from the code snippet of Figure 5a, and G_1 is transformed from G_0 by applying the two productions $A \rightarrow a$ and $Y \rightarrow \overline{new}\ new$. For clarity, all inverse edges are omitted.

Figure 5c demonstrates the forward and backward propagation using G_2 and G_3, with all path edges represented as dashed lines: (1) *Forward propagation* (G_2): The path edge $b \dashrightarrow b$ is popped from FWL (Line 6), and propagated along $b \xrightarrow{A} c$ and $c \xrightarrow{A} e$ (once per iteration), generating two forward path edges $b \dashrightarrow c$ and $b \dashrightarrow e$ (Lines 9-11). As stated in lines 7-8, a backward path edge $b \dashleftarrow e$ is created, due to the existence of $e \xrightarrow{get_f} i$. (2) *Backward propagation* (G_3): In Lines 12-19, the backward path edge $b \dashleftarrow e$ produces $d \dashleftarrow e$. Subsequently, a summary edge $h \xrightarrow{A} i$ (represented as the curvy line in G_3) is resulted by matching $h \xrightarrow{put_f} d$ with $e \xrightarrow{get_f} i$. (3) Handling new edges: new edges may arise from the newly created summary edge (Lines 22-29). Algorithm 1 terminates when new edges can no longer be produced.

Discussion. Algorithm 1 is specifically designed for Alias CFP, and it may require adjustments for new patterns. In our earlier implementation, we failed to adapt the tabulation-based algorithm [20] for this pattern due to poor performance caused by transitive redundancy [11]. Algorithm 1 circumvents such redundancy by propagating Y-reachability information along A-edges and B-edges. Regarding precision, both algorithms compute the same set of summary edges. Lastly, this optimization is also implemented in the baseline algorithms for a fair comparison in our evaluation (Section 4).

```
b = new O();//O1
c = b;
d = b;
e = c;
d.f = h;
i = e.f;
```

(a)

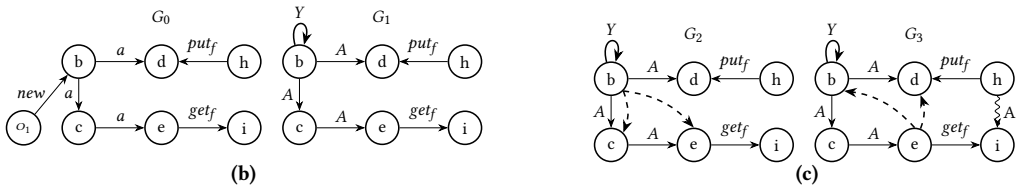

Figure 5: An example of Java's points-to analysis. (a) A code snippet. (b) G_0 is the input graph translated from *(a)*. G_1 is transformed from G_0 via $A \rightarrow a$ and $Y \rightarrow \overline{new}\ new$. (c) G_2 and G_3 show the forward and backward propagation (in dashed lines), respectively. Finally, a summary edge $h \xrightarrow{A} i$ (the curvy line) is generated.

Algorithm 2: Solving $\mathcal{R}$-reachability via ordered propagation

Input: Regular expression $E = e_1 \cdot e_2 ... e_m$
Output: Reachability results in R

```
1  Procedure SolveReg(e1 · e2...em):
2      for each node n ∈ V do
3          R_old(n) := {n};
4      for i from 1 to m do
5          if e_i = a then
6              for each node n ∈ V do
7                  R(n) := ∅;
8              for each edge u -a-> v ∈ E do
9                  R(v) := R(v) ∪ R_old(u);
10         else if e_i = a* then
11             for each node n ∈ V do
12                 R(n) := R_old(n);
13             for each edge u -a-> v ∈ G_a in topological order do
14                 R(v) := R(v) ∪ R(u);
15         R_old = R;
```

3.4 Solving $\mathcal{R}$-Reachability

Preprocessing. After grammar decomposition (Section 3.1.2), $\mathcal{R}$ is a regular language with each nonterminal represented as a regular expression over $\Sigma \cup N_{\mathcal{L}}$, where Σ is the set of terminals in the original *CFG* and $N_{\mathcal{L}}$ is the set of nonterminals in $\mathcal{L}$. Following the rules in [3, 7], we further normalize the regular expression for each nonterminal in $\mathcal{R}$ into the *disjunctive normal form*:

Definition 2. (Disjunctive Normal Form). The disjunctive normal form is of the form $E_1|E_2|...|E_n$, where each E_i, for $i = 1...n$, is a regular expression using only concatenation $\cdot$ and Kleene star $*$.

For each production $R \rightarrow E_1|E_2|...|E_n$ of $\mathcal{R}$ in disjunctive normal form, we solve each sub-production $R \rightarrow E_i$ separately in Algorithm 2, with the collective results constituting the final solution. Commonly, each E_i is a concatenation of either terminals (a), or Kleene closures of a terminal (a^*), as observed in the four CFGs we studied above. Hence, for clarity, we present Algorithm 2 for this simple form of regular expressions, then discuss how general regular expressions are addressed with a minor extension.

Solving Algorithm. Algorithm 2 solves the input expression $E = e_1 \cdot e_2 ... e_m$ by processing each sub-expression e_i sequentially, from left to right. In the i-th iteration of the algorithm, $R(v)$ (resp. $R_{old}(v)$) signifies nodes reachable to v through a path with the sequence $e_1 \cdot ... e_i$ (resp. $e_1 \cdot ... e_{i-1}$). Lines 5-9 address the scenario where $e_i = a$, which is self-explanatory. When dealing with $e_i = a^*$ (Lines 10-14), we initially set $R(n)$ to $R_{old}(n)$ (considering a^* as ϵ). Subsequently, we propagate $R(n)$ along a-edges following the topological order, which is obtained by building a subgraph G_a consisting of only a-edges with strongly connected components (SCCs) collapsed. This ordered propagation is facilitated by staged solving, and is not applicable in the standard CFL-reachability algorithm [5, 18] due to dynamically inserted edges.

Optimization. For $e_i = a^*$, two optimizations are implemented: (1) Except for the first loop iteration (e_1), we can skip the loop at Lines 11-12 since the contents of R and R_{old} are identical. (2) An SCC is collapsed as a single node, with only one copy of R retained for all nodes within the SCC.

Extension. Algorithm 2 can be easily extended to handle complex expressions, such as $e_i = (ab)^*$ and $e_i = (ab^*)^*$, by conceptually introducing new nonterminals for inner expressions of a Kleene closure. For example, to handle $e_i = (ab^*)^*$, we can introduce a new nonterminal C for the sequence ab^*, thereby simplifying the original expression to C^*.

4 Evaluation

We evaluate staged solving on two applications: context-sensitive value-flow analysis [31] and field-sensitive alias analysis [38]. Both analyses are extensively evaluated in recent works [11, 12, 23, 33].

Value-flow Analysis. The context-sensitive value-flow analysis is conducted on the sparse value-flow graphs (SVFGs) [30, 31], using the CFG of the extended Dyck-CFL as shown in Figure 2a. Previous works [11, 23] have evaluated value-flow analysis employing the standard Dyck-CFL (Figure 1c), while we choose the extended Dyck-CFL, which is more applicable but hard to scale [24].

Alias Analysis. The field-sensitive alias analysis for C++ is conducted on the program expression graphs (PEGs) [38], using the CFG presented in Figure 3a.

4.1 Experimental Setup

Environment. All the experiments were conducted on a server with two 12-core 3.00GHz Intel(R) Xeon(R) Gold 5317 CPUs and 1 TB of physical memory. We conducted the experiments within a time limit of 6 hours and a memory allocation of 512 GB. Each

Table 1: Results of context-sensitive value-flow analysis. #Nodes and #Edges denote the numbers of nodes and edges in the input graph, respectively. Time and Mem signify the time (in seconds) and memory (in gigabytes) consumption. "-" means that an algorithm timeouts in 6 hours; "OoM" indicates that an algorithm runs out of memory (512GB). #SumEdge, #PathEdge, and #TotalEdge denote summary edges, processed path edges in Phase_L, and total derived edges in two phases, respectively.

id	#Nodes	#Edges	Subcubic		Stg		Pocr		Pocr^{Stg}		Pearl		$\text{Pearl}^{\text{Stg}}$		#SumEdge	#PathEdge	#TotalEdge
			Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem			
cactus	544,480	1,007,989	-	-	19.61	15.49	OoM	OoM	555.38	201.42	-	-	974.12	20.70	1,892,558	7,752,527	141,772,144,876
imagick	574,089	842,509	-	-	13.43	14.93	OoM	OoM	116.74	59.07	-	-	76.75	18.62	353,018	2,209,455	44,436,103,006
leela	64,466	89,081	249.24	1.68	0.50	0.39	123.88	20.86	0.94	0.74	17.31	1.06	0.66	0.48	37,342	171,158	313,802,497
nab	55,652	72,366	80.16	0.70	0.55	0.26	164.40	38.67	20.20	7.95	13.70	0.71	0.98	0.37	139,343	499,097	82,655,045
omnetpp	664,358	1,857,831	-	-	19.50	20.94	-	-	49.78	34.28	5695.63	49.98	47.91	22.63	1,055,668	2,397,273	24,485,460,751
parest	299,718	407,343	63.84	3.32	2.45	1.81	33.98	6.34	2.58	2.09	13.03	4.33	2.58	2.12	15,514	113,865	63,953,921
perlbench	697,744	1,662,445	-	-	28.24	26.46	OoM	OoM	614.45	220.51	-	-	494.03	36.63	404,195	11,780,236	234,707,705,380
povray	537,775	1,041,687	-	-	21.63	13.90	OoM	OoM	528.32	209.92	-	-	321.28	19.13	2,839,417	10,664,021	136,347,654,971
x264	207,064	340,217	11576.80	24.74	3.36	3.64	7736.21	469.90	53.68	25.61	560.51	10.05	11.46	4.73	107,811	844,686	8,357,200,739
xz	49,395	62,955	57.65	0.70	0.30	0.22	36.91	7.93	0.48	0.40	4.39	0.59	0.34	0.28	8,594	45,784	66,835,873
Average			2405.54	6.23	10.96	9.8	1619.08	108.74	194.26	76.2	1050.76	11.12	193.01	12.57			

Table 2: Results of field-sensitive alias analysis. The meaning of each column is the same as Table 1.

id	#Nodes	#Edges	Subcubic		Stg		Pocr		Pocr^{Stg}		Pearl		$\text{Pearl}^{\text{Stg}}$		#SumEdge	#PathEdge	#TotalEdge
			Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem			
cactus	93,557	212,478	664.61	2.76	156.83	2.55	353.32	17.99	147.92	18.87	149.87	1.28	71.72	2.68	15,865,822	265,900,612	831,928,173
imagick	119,314	301,846	2973.75	13.54	965.38	6.00	800.12	37.93	756.05	40.30	579.20	2.77	587.03	6.12	176,799,781	1,340,981,966	3,621,222,682
leela	22,186	49,748	13.74	0.20	2.08	0.24	2.96	0.36	1.68	0.50	2.48	0.12	1.32	0.25	589,865	5,697,894	28,249,442
nab	16,261	34,676	2.82	0.11	0.58	0.13	1.10	0.11	0.53	0.15	0.70	0.06	0.51	0.14	692,061	1,869,752	9,758,181
omnetpp	241,916	509,166	4868.77	9.17	1795.77	9.81	3834.51	215.07	3291.23	348.17	1296.51	5.21	938.54	10.12	56,930,084	929,126,781	2,568,491,006
parest	117,500	251,436	525.44	3.15	100.73	2.59	257.57	3.12	107.46	4.51	155.73	1.43	46.57	2.72	8,309,106	126,448,701	566,954,858
perlbench	139,183	348,916	18035.20	47.23	7937.61	11.29	6698.27	401.08	6721.77	406.59	3796.14	9.25	2549.65	11.08	106,395,782	5,892,471,865	12,305,454,819
povray	76,405	174,258	543.43	2.26	110.66	1.97	283.43	13.79	105.82	12.89	117.28	0.99	55.21	2.01	18,392,790	212,514,033	690,099,171
x264	60,956	136,352	39.77	0.58	10.65	1.02	13.39	0.84	8.59	1.40	9.98	0.46	8.26	1.07	6,052,306	20,590,293	84,099,124
xz	12,425	26,468	1.33	0.06	0.39	0.09	0.50	0.06	0.29	0.11	0.34	0.05	0.30	0.10	464,570	1,543,965	5,777,437
Average			2766.89	7.91	1108.07	3.57	1224.52	69.03	1114.13	83.35	610.82	2.16	425.91	3.63			

experiment was run 5 times with the average runtime and memory consumption reported.

Baselines. We have implemented our staged solver, Stg, and compared it against the baseline solver, Subcubic, which implements Chaudhuri's subcubic algorithm [5]. According to a previous study [37], the subcubic algorithm is much more efficient than the standard cubic algorithm [15]. We further optimized Subcubic using a grammar rewriting technique [11] to reduce transitive redundancy. For instance, $S \rightarrow S\ S \mid s \mid \epsilon$ can be rewritten as $S \rightarrow S\ s \mid \epsilon$. Moreover, we evaluated Stg against two state-of-the-art solvers, Pocr [11] and Pearl [23]. All solvers are implemented on top of SVF [29] and LLVM [10], utilizing the sparse bit vector data-structure from LLVM, which implements fast set operations.

Evaluation of Correctness. We have written a script to validate that Stg and Subcubic compute the same set of edges.

Benchmarks. In alignment with previous works [11, 23], we chose 10 SPEC C/C++ programs (listed in Table 1 and Table 2) from the SPEC CPU 2017 benchmarks. We used SVF to extract both SVFGs and PEGs.

Our evaluation aims to answer the following research questions:

- (RQ1). How does Stg compare to Subcubic?
- (RQ2). How does Stg compare to Pocr and Pearl?
- (RQ3). Can staged solving enhance the efficiency of Pocr and Pearl?
- (RQ4). Which phase dominates the time consumption of Stg?

4.2 Comparison against Subcubic

Table 1 and Table 2 present the results of value-flow analysis and alias analysis, respectively. The meanings of each column are explained in the captions. Regarding time consumption, Stg is faster than Subcubic in all evaluated benchmarks.

Performance Improvement. In value-flow analysis, Stgcan analyze 5 benchmarks (out of 10) that Subcubic cannot within a 6-hour timeframe. For those 5 benchmarks that Subcubic fails to analyze in 6 hours, Stg can complete its analysis in less than 30 seconds. For the remaining 5 benchmarks that both Stg and Subcubic can analyze, Stg achieves an average speedup of 861.59x, with the peek speedup reaching 3445.48x for the benchmark *x264*. In alias analysis, the speedup achieved by Stg is more moderate, with an average improvement of 4.1x. Concerning memory consumption, compared to Subcubic, Stg achieves an average decrease of 3.76x for value-flow analysis and 1.37x for alias analysis, respectively.

Discussion. Now let us examine the performance improvements in two parts: (1) the sparsity of the input graphs, and (2) the number of edges involved in Phase_L. As shown in Table 1 and Table 2, both the input SVFGs and PEGs are extremely sparse, with the ratios of #Edges to #Nodes being 2.26x and 1.74x on average, respectively. Additionally, the number of edges processed in Phase_L is significantly lower far less than the number of total edges in each evaluated benchmark. Specifically, in value-flow analysis (Table 1), summary edges (#SumEdge) and path edges (#PathEdge) in Phase_L constitutes only 0.02% and 0.1% of the total edges (#TotalEdge) on

average, respectively. As a result, we observe a drastic performance improvement in value flow analysis. Meanwhile, in alias analysis (Table 2), summary edges and path edges in Phase_L account for 3.84% and 29.67%of the total edges, respectively. Therefore, a more moderate speedup is obtained for alias analysis.

In both analyses, the number of edges involved in $\text{Phase}_\text{L}$is significantly smaller compared to the total number of derived edges, thus allowing Phase_L to be performed quickly. The time complexity of Phase_R is $O(mn)$. We observe that, m, the combined count of initial edges (#Edges) and summary edges (#SumEdge), is bounded by $O(n)$ (with n = #Nodes). Hence, solving regular language reachability via ordered propagation (Algorithm 2) is highly efficient. This is the primary reason for the notable speedups achieved by Stg over Subcubic. For instance, the moderate sized benchmark *x264* requires Subcubic more than 3 hours to analyze, demonstrating that the extended Dyck-CFL-reachability is challenging to scale [24]. In contrast, Stg completes its analysis in seconds.

We have also evaluated Algorithm 2 in solving $\mathcal{R}$-reachability against the algorithm in [35] (referred to as StdReg), which solves a regular language reachability problem by reducing it to a traditional transitive closure. We have implemented StdReg together with common optimization techniques such as SCC collapsing and ordered propagation. In this comparison, Algorithm 2 demonstrates average speedups of 1.42x and 2.17x over StdReg for value-flow analysis and alias analysis, respectively. Due to space limits, the detailed data is not given here.

4.3 Comparison against Pocr and Pearl

In this subsection, we compare Stg against two state-of-the-art CFL-reachability solvers, namely Pocr [11] and Pearl [23]. Pocr employs partially ordered solving for transitive relations, while Pearl features a transitivity-aware multi-derivation approach. The original implementations of the two solvers are based on the standard cubic algorithm [15]. We further optimized their original implementations by incorporating the more efficient subcubic algorithm [5]. This optimization achieves notable speedups: for instance, in alias analysis, the optimized version of Pocr achieves an average speed up of 2.63x compared to the original implementation. Columns 8-9 and 12-13 of Table 1 (Table 2) present the performance results for Pocr and Pearl in value-flow analysis (alias analysis), respectively.

Value-flow Analysis. In value-flow analysis, both Pocr (1.49x) and Pearl (11.79x) demonstrate significant performance improvements over Subcubic. Note that the speedups will be considerably higher if we disable grammar rewriting for transitive redundancy optimization in Subcubic. However, when compared to Stg (861.59x), the speedups achieved by Pocr and Pearl appear minimal. Furthermore, within a 6-hour timeframe, Pocr and Pearl fail to analyze 5 and 4 benchmarks, respectively. In contrast, Stg completes its analysis in just 30 seconds. Additionally, Pocr demands a large amount of memory and exhausts available memory (512GB) in 4 benchmarks. The large memory footprint of Pocr is due to the high maintenance cost of the spanning tree model (consistent with previous works [11, 12, 23]). On the other hand, Pearl is more memory-efficient.

Stg achieves drastic speedups over Pocr and Pearl. This is due to the fact the in value-flow analysis, Phase_L comprises a minimal percentage of total edges (as detailed in Section 4.2). Hence, by solving the majority edges using an efficiently regular reachability algorithm (Algorithm 2), Stg obtains substantial performance enhancements. On the other hand, Pocr and Pearl optimize on the monolithic subcubic algorithm, resulting in only limited speedups.

Alias Analysis. Pocr and Pearl achieve average speedups of 2.63x and 4.35x over Subcubic, respectively, demonstrating comparable performance to Stg (4.1x). This underscores the effectiveness of Pocr and Pearl in optimizing out transitive redundancy, which often cannot be addressed using grammar rewriting techniques [11]. More importantly, in alias analysis, the percentages of edges involved in solving $\mathcal{L}$-reachability is significantly higher (29.67%) than that in value flow analysis (0.1%). Consequently, the speedups achieved by Stg are more modest, with Stg only showing performance similar to that of Pocr and Pearl.

4.4 Combination with Pocr and Pearl

In this experiment, we apply staged solving in conjunction with Pocr and Pearl, resulting in Pocr^{Stg} and $\text{Pearl}^{\text{Stg}}$, respectively. Both Pocr and Pearl target to optimize transitive redundancy, by conceptually computing a closure of transitive relations. Hence, Pocr^{Stg} and $\text{Pearl}^{\text{Stg}}$ address $\mathcal{L}$-reachability using Pocr and Pearl, respectively, while they compute $\mathcal{R}$-reachability using Algorithm 2, which is shown to be much more efficient in solving regular reachability problems than Pocr and Pearl alone, according to our experiments.

In Table 1 and Table 2, the performance metrics of Pocr^{Stg} and $\text{Pearl}^{\text{Stg}}$ are detailed in columns 10-11 and 14-15, respectively. Specifically, in alias analysis (Table 2), Pocr^{Stg} achieves a 1.78x speedup over Pocr, while $\text{Pearl}^{\text{Stg}}$ achieves a 1.7x speedup over Pearl. The performance improvements are even more pronounced in value-flow analysis (Table 1). When compared to their non-staged counterparts, both Pocr^{Stg} and $\text{Pearl}^{\text{Stg}}$ demonstrate scalability across all evaluated benchmarks, registering improvements of 74.82x and 37.66x, respectively.

When comparing Pocr^{Stg} and $\text{Pearl}^{\text{Stg}}$ to Stg, for value-flow analysis (Table 1), both Pocr^{Stg} and $\text{Pearl}^{\text{Stg}}$ exhibit lower overall efficiency than Stg alone, with an average slowdown of 14.3x and 9.89x, respectively. This surprising result is because in value flow analysis, the overhead of computing transitive closures (to facilitate transitive redundancy elimination) outweighs the benefits of reduced redundancy, especially when $\mathcal{L}$ only accounts for an insignificant percentage of total edges. For instance, in *nab*, the size of the transitive closure is around 60 times larger than the number of path edges. As a result, both Pocr^{Stg} and $\text{Pearl}^{\text{Stg}}$ run slower than Stg in value flow analysis.

However, in alias analysis, where transitive redundancy is abundant and transitive closure is relatively small compared to the number of path edges, Pocr^{Stg} and $\text{Pearl}^{\text{Stg}}$ outperform Stg, with average speedups reaching 1.1x and 1.83x, respectively.

In conclusion, staged solving complements existing techniques and significantly enhances their efficiency.

4.5 Time Distribution of Two Phases

As depicted in Figure 6, in value-flow analysis, Stg's temporal distribution allocates averagely 21.92% and 78.08% of the analysis time

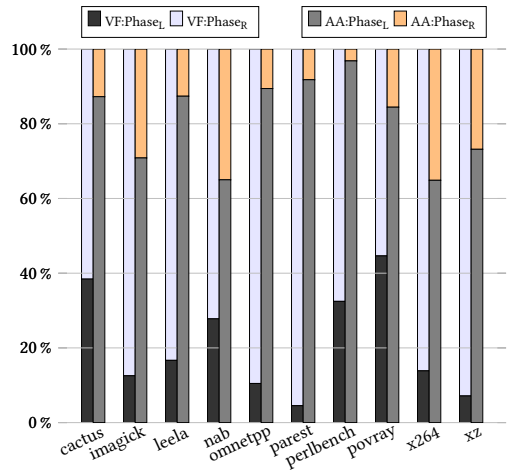

Figure 6: Percentages of time consumption in Phase$_L$ and Phase$_R$ for value-flow analysis (VF) and alias analysis (AA).

to Phase$_L$ and Phase$_R$, respectively. In alias analysis, this distribution is essentially reversed, with Phase$_L$ dominating Stg's time consumption at 81.09%.

We analyze the reason as follows: Grammar $\mathcal{L}$ in alias analysis is more complex, featuring two intricate productions (represented by nonterminals V' and M in Figure 3c) that mutually depend on each other. Consequently, in alias analysis, Phase$_L$ is notably time-consuming, whereas in value-flow analysis, Phase$_L$ can be quickly conducted and summary edges are easily resolved. This complexity gap is also reflected in the portion of edges derived in Phase$_L$, with 0.1% and 29.67% for value-flow analysis and alias analysis, respectively. As a result, compared to value-flow analysis, Stg achieves a more modest performance enhancement over Subcubic in alias analysis, since the time-consuming Phase$_L$ undermines the benefits from efficiently solving $\mathcal{R}$-reachability in Phase$_R$.

4.6 Discussion

Given a CFL-based analysis with the grammar $CFG = (N, \Sigma, P, S)$, staged solving is applicable if, by replacing each context-free expression $a\ E_c\ b$ in P by a symbol S' (with $a, b \in \Sigma$ and E_c is a regular expression over $N \cup \Sigma$), the original grammar *CFG* can be reduced to a regular grammar over $\Sigma \cup \{S'\}$. As such, *CFG* can be decomposed into $\mathcal{L}$ and $\mathcal{R}$ as detailed in Section 3.1.2. For all CFL-based analyses we studied, staged solving is applicable.

Furthermore, staged solving is beneficial if the decomposed grammar $\mathcal{L}$ is smaller (in terms of deduced edges) than the original *CFG*. In general, the smaller $\mathcal{L}$ is, the greater the performance improvements can be achieved since the performance enhancements are primarily attributed to the efficient resolution of $\mathcal{R}$-reachability through Algorithm 2. This is confirmed in in our evaluation: the speedups achieved by Stg over Subcubic is significantly higher in value-flow analysis (861.59x) compared to that in alias analysis (4.1x). This improvement aligns with the reduced size of the grammar $\mathcal{L}$ compared to the original *CFG* in value-flow analysis (0.1%) and in alias analysis (29.67%).

5 Related Work

This work focuses on optimizing CFL-reachability solving, a pivotal framework in program analysis with diverse applications [2, 17, 19, 20, 25–28, 36–38]. The (sub)cubic time complexity of CFL-reachability has prompted the development of practical optimization techniques. Previous works [37, 38] showed that the Four Russian's Trick can yield a subcubic algorithm by utilizing set operations of fast sets. Graspan [33] adopts efficient data processing techniques from the Big Data community. Pocr [11] reduces transitive redundancy by determining a partial derivation order on the fly. Pearl [23] features a multi-derivation approach to eliminate repetitive derivations. In principle, all these techniques for CFL-reachability are orthogonal to staged solving and can be employed in Phase$_L$ of staged solving to further improve the overall efficiency. Graph simplification techniques [12–14, 21], shrinking the size of the input graph, can also benefit both phases of staged solving.

Bottom-up interprocedural analyses [6, 16], focusing on context-sensitivity, share a common high-level concept with staged solving: they both compute summaries first. The *seminaive evaluation algorithm* for Datalog [1, 8] divides the solving process into independent parts but does not address the mutual dependency problem. In the literature, certain program analyses based on graph reachability operate in multiple phases. However, to our knowledge, unlike staged solving, none of these approaches serves as a technique for general CFL-based program analyses. For example, in interprocedural data-flow analysis [17, 24] and slicing [20], researchers proposed a hybrid approach: exhaustively compute function summary edges first and answer queries on demand, while both phases of staged solving are exhaustive. Additionally, the work in [37] introduces a staged algorithm for alias analysis, leveraging specialized properties of the underlying graphs, and thus the proposed techniques do not apply to other program analyses.

6 Conclusion

We propose staged solving, a novel approach to improve the scalability of CFL-reachability analysis by addressing the context-free and regular aspects of the underlying CFG in distinct stages. To the best of our knowledge, staged solving is applicable to all CFL-based analyses in the literature. We have developed an efficient staged CFL-reachability solver, Stg, and applied it to a wide range of CFL-based analyses. Empirical results demonstrate that Stg can achieve drastic speedups over Subcubic, averaging 861.59x for value-flow analysis and 4.1x for alias analysis, respectively. Furthermore, applying staged solving to two state-of-the-art solvers shows significant performance improvements, with 74.82x (1.78x) for Pocr and 37.66x (1.7x) for Pearl in value-flow (alias) analysis, respectively.

7 Data Availability

Our artifact is publicly available at [22].

Acknowledgments

We thank the reviewers for their valuable comments on this work. This work is supported by the National Key R&D Program of China (2022YFB3103900), the National Natural Science Foundation of China (62132020, 62202452, and 62402474), and the China Postdoctoral Science Foundation (2024M753295).

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Vol. 8. Addison-Wesley Reading.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269. https://doi.org/10.1145/2666356.2594299

[3] Anne Brüggemann-Klein. 1993. Regular expressions into finite automata. *Theoretical Computer Science* 120, 2 (1993), 197–213. https://doi.org/10.1016/0304-3975(93)90287-4

[4] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. Optimal Dyck reachability for data-dependence and alias analysis. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30. https://doi.org/10.1145/3158118

[5] Swarat Chaudhuri. 2008. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 159–169. https://doi.org/10.1145/1328438.1328460

[6] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-up context-sensitive pointer analysis for Java. In *Asian Symposium on Programming Languages and Systems*. Springer, 465–484.

[7] John E Hopcroft and Jeffrey D Ullman. 1969. *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc.

[8] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. 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, 422–430. https://doi.org/10.1007/978-3-319-41540-6_23

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

[10] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[11] Yuxiang Lei, Yulei Sui, Shuo Ding, and Qirun Zhang. 2022. Taming transitive redundancy for context-free language reachability. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1556–1582. https://doi.org/10.1145/3563343

[12] Yuxiang Lei, Yulei Sui, Shin Hwei Tan, and Qirun Zhang. 2023. Recursive State Machine Guided Graph Folding for Context-Free Language Reachability. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 318–342. https://doi.org/10.1145/3591233

[13] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2020. Fast graph simplification for interleaved Dyck-reachability. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 780–793. https://doi.org/10.1145/3385412.3386021

[14] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2022. Fast Graph Simplification for Interleaved-Dyck Reachability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44, 2 (2022), 1–28. https://doi.org/10.1145/3492428

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

[16] Erik M Nystrom, Hong-Seok Kim, and Wen-Mei W Hwu. 2004. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Static Analysis: 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004. Proceedings 11*. Springer, 165–180. https://doi.org/10.1007/978-3-540-27864-1_14

[17] Thomas Reps. 1998. Program analysis via graph reachability. *Information and software technology* 40, 11-12 (1998), 701–726. https://doi.org/10.1016/S0950-5849(98)00093-7

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

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

[20] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes* 19, 5 (1994), 11–20. https://doi.org/10.1145/193173.195287

[21] Atanas Rountev and Satish Chandra. 2000. Off-line variable substitution for scaling points-to analysis. *Acm Sigplan Notices* 35, 5 (2000), 47–56. https://doi.org/10.1145/349299.349310

[22] Chenghang Shi, Haofeng Li, Jie Lu, and Lian Li. 2024. Artifact of "Better Not Together: Staged Solving for Context-Free Language Reachability". (2024). https://doi.org/10.6084/m9.figshare.26156944.v4

[23] Chenghang Shi, Haofeng Li, Yulei Sui, Jie Lu, Lian Li, and Jingling Xue. 2023. 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, 624–636. https://doi.org/10.1109/ASE56229.2023.00118

[24] Qingkai Shi, Yongchao Wang, Peisen Yao, and Charles Zhang. 2022. Indexing the extended Dyck-CFL reachability for context-sensitive program analysis. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1438–1468. https://doi.org/10.1145/3563339

[25] Manu Sridharan. 2007. *Refinement-based program analysis tools*. University of California, Berkeley.

[26] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. *ACM SIGPLAN Notices* 41, 6 (2006), 387–400. https://doi.org/10.1145/1133981.1134027

[27] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN conference on programming language design and implementation*. 112–122. https://doi.org/10.1145/1250734.1250748

[28] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. *ACM SIGPLAN Notices* 40, 10 (2005), 59–76. https://doi.org/10.1145/1094811.1094817

[29] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266. https://doi.org/10.1145/2892208.2892235

[30] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 254–264. https://doi.org/10.1145/2338965.2336784

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

[32] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 83–95. https://doi.org/10.1145/2676726.2676997

[33] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 389–404. https://doi.org/10.1145/3037697.3037744

[34] Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *ECOOP*, Vol. 9. Springer, 98–122. https://doi.org/10.1007/978-3-642-03013-0_6

[35] Mihalis Yannakakis. 1990. Graph-theoretic methods in database theory. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 230–242. https://doi.org/10.1145/298514.298576

[36] Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 435–446. https://doi.org/10.1145/2491956.2462159

[37] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. 2014. Efficient subcubic alias analysis for C. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. 829–845. https://doi.org/10.1145/2660193.2660213

[38] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 197–208. https://doi.org/10.1145/1328438.1328464

Received 2024-04-12; accepted 2024-07-03