Context-Free Language Reachability via Efficient Relation Chaining

CHENGHANG SHI, Institute of Computing Technology, CAS, China and University of Chinese Academy of Sciences, China
HAOFENG LI, Institute of Computing Technology, CAS, China
JIE LU, Institute of Computing Technology, CAS, China
LIAN LI*, Institute of Computing Technology, CAS, China, University of Chinese Academy of Sciences, China, and Zhongguancun Laboratory, China

Context-free language (CFL) reachability is a fundamental framework widely used to model a variety of program analysis tasks, though it often suffers from inherent inefficiency due to its (sub)cubic time complexity.

In this paper, we propose a novel perspective, relation chaining, which interprets CFL-reachability solving as the process of chaining labeled edges representing binary relations. This formulation exposes substantial derivation redundancy (in terms of frequent and repetitive chaining operations) arising from inefficient chaining strategies employed by existing approaches. To address this, we introduce Squid, a new algorithm that incorporates two simple yet effective chaining techniques—adaptive chaining and differential chaining—built upon an enhanced graph representation. We have implemented Squid as a standalone tool and evaluated it against two state-of-the-art CFL-reachability solvers and a leading Datalog solver across three key program analyses: field-sensitive alias analysis and context-sensitive value-flow analysis for C/C++, and field-sensitive points-to analysis for Java. Experimental results show that Squid substantially improves the scalability of CFL-reachability solving by effectively reducing a large portion of redundant chaining operations.

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

Additional Key Words and Phrases: CFL-reachability, program analysis

ACM Reference Format:
Chenghang Shi, Haofeng Li, Jie Lu, and Lian Li. 2026. Context-Free Language Reachability via Efficient Relation Chaining. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 162 (April 2026), 29 pages. https://doi.org/10.1145/3798270

1 Introduction

Context-free language (CFL) reachability is a fundamental program analysis framework that models a wide range of client applications as graph reachability problems [Reps 1998; Yannakakis 1990], including pointer analysis [Sridharan and Bodík 2006; Sridharan et al. 2005; Xu et al. 2009; Zhang et al. 2013; Zheng and Rugina 2008], program slicing [Li et al. 2016; Sridharan et al. 2007; Weiser 1981], type-based flow analysis [Rehof and Fähndrich 2001], data flow analysis [Arzt et al. 2014; He et al. 2019; Li et al. 2024; Reps et al. 1995], sparse value-flow analysis [Li et al. 2011, 2013; Shi et al.

*Corresponding author

Authors' Contact Information: Chenghang Shi, Institute of Computing Technology, CAS, Beijing, China and University of Chinese Academy of Sciences, Beijing, China, shichenghang21s@ict.ac.cn; Haofeng Li, Institute of Computing Technology, CAS, Beijing, China, lihaofeng@ict.ac.cn; Jie Lu, Institute of Computing Technology, CAS, Beijing, China, lujie@ict.ac.cn; Lian Li, Institute of Computing Technology, CAS, Beijing, China and University of Chinese Academy of Sciences, Beijing, China and Zhongguancun Laboratory, Beijing, China, lianli@ict.ac.cn.

ACM 2475-1421/2026/4-ART162
https://doi.org/10.1145/3798270

2018; Sui et al. 2012; Yao et al. 2024], shape analysis [Reps 1998], and specification inference for missing code [Bastani et al. 2015], among others.

Significant research efforts have focused on improving CFL-reachability performance. Notable advancements include transitive redundancy elimination [Lei et al. 2022], bit-vector operations for subcubic performance [Zhang et al. 2014; Zheng and Rugina 2008], disk-assisted parallel computation [Wang et al. 2017], skewed tabulation [Lei et al. 2024], automaton-guided graph folding [Lei et al. 2023], regularization-based graph simplification [Shi et al. 2025], online cycle elimination [Xu et al. 2024], and multi-derivation [Shi et al. 2023, 2024b]. Nevertheless, existing techniques still fall short of satisfying the efficiency requirements of practical client applications.

In this CFL-reachability framework, a context-free language (or equivalently, a context-free grammar, CFG) is used to formalize program analysis problems (e.g., taint analysis). The analyzed program is represented as an edge-labeled graph $G = \langle V, E \rangle$, where V and E denote the sets of vertices and edges, respectively. The analysis task is to determine, given two nodes v_0 and v_n, whether there exists a path $v_0 \xrightarrow{t_1} v_1 \xrightarrow{t_2} v_2 \rightarrow \ldots \rightarrow v_{n-1} \xrightarrow{t_n} v_n$ such that the concatenation of edge labels $t_1 t_2 \ldots t_n$ can is derivable from some grammar symbol X. Such a path is called a *X-reachable path*, and we say that v_0 is *X-reachable* to v_n.

To solve CFL-reachability, the standard algorithm [Melski and Reps 2000] essentially computes a dynamic transitive closure over the edge-labeled graph G. In other words, productions are applied iteratively to summarize reachable paths as new labeled edges in G, thereby making the reachability information explicit. This process continues until a fixed point is reached, i.e., no additional edges can be derived.

To understand the essence of CFL-reachability, consider a production $X ::= Y\ Z$ in a CFG and a node $v \in V$. The solver applies this production by combining each Y-edge ending at v (e.g., $u \xrightarrow{Y} v$) with each Z-edge starting from v (e.g., $v \xrightarrow{Z} w$) to create a new X-edge (e.g., $u \xrightarrow{X} w$) if it does not already exist in G.

Intuitively, this operation corresponds to chaining two sets of labeled edges that share a common node v—the fundamental mechanism underlying CFL-reachability solving. To reason about this process more systematically, we introduce the notion of *relation chaining*, where each labeled edge set represents a binary relation over $V \times V$. From this chaining perspective, the efficiency of a CFL-reachability solver hinges on how effectively it performs relation chaining as new edges are generated on the fly. However, when existing solvers are interpreted in this framework, we observe substantial *derivation redundancy*, manifested as repetitive and unnecessarily frequent chaining operations. To address this inefficiency, we propose Squid, a new CFL-reachability solver based on optimized relation chaining. Squid incorporates two key techniques—*adaptive chaining*, which applies chaining more selectively and intelligently, and *differential chaining*, which avoids redundant chaining through incremental computation. Enabled by an enhanced graph representation, these techniques together eliminate redundant operations, accelerate the saturation process, and substantially improve solving efficiency.

We evaluate Squid in comparison with two representative state-of-the-art CFL-reachability solvers, Skewed [Lei et al. 2024] and Pearl [Shi et al. 2024b], as well as a leading Datalog solver Soufflé [Jordan et al. 2016], across three important program analysis tasks: context-sensitive value-flow analysis [Sui et al. 2012, 2014] and field-sensitive alias analysis [Zhang et al. 2014; Zheng and Rugina 2008] for C/C++, and field-sensitive points-to analysis for Java [Sridharan and Bodík 2006; Sridharan et al. 2005]. Experimental results demonstrate the superior effectiveness and efficiency of Squid.

Compared with Skewed, an optimized standard solver, Squid reduces 97.76%, 84.76%, and 72.05% of chaining operations across the three client analyses, achieving average speedups of 9.30×, 3.08×,

and 1.86×, respectively. Relative to Pearl, a multi-derivation approach based on set-constraints, it reduces 96.78%, 60.30%, and 93.97% of chaining operations, resulting in average speedups of 12.55×, 1.70×, and 8.31×. Compared with Soufflé, a widely used Datalog solver, Squid attains average speedups of 1.87×, 4.93×, and 3.52× on the same analyses. Finally, a case study on taint-style memory leak analysis [Sui et al. 2012, 2014] further demonstrates the practical benefits of Squid in improving CFL-reachability performance.

To summarize, this paper makes the following contributions:

- We introduce *relation chaining*, a new perspective for solving CFL-reachability problems. By interpreting existing CFL-reachability algorithms through this lens, we uncover substantial derivation redundancy arising from their naive chaining strategies.
- We present a novel CFL-reachability algorithm that incorporates two simple yet effective techniques—*adaptive chaining* and *differential chaining*—to accelerate the solving process by reducing redundant chaining operations.
- We conduct extensive experiments demonstrating the effectiveness of Squid, achieving significant speedups over existing algorithms for solving CFL-reachability by reducing a substantial number of chaining operations.

The remainder of this paper is organized as follows. Section 2 provides background on CFL-reachability and presents a motivating example. Section 3 formally defines relation chaining and describes our new approach, Squid, which solves CFL-reachability through efficient relation chaining. Section 4 presents the evaluation results. Section 5 reviews related work in this area. Finally, Section 6 concludes the paper and outlines potential directions for future research.

2 Background

This section begins by introducing the background of CFL-reachability, focusing on the basic definitions and notations used throughout the paper (Section 2.1), followed by the standard algorithm for solving CFL-reachability (Section 2.2). Finally, we present an illustrative example to motivate our research (Section 2.3).

2.1 Language Reachability

Context-free grammar and its normalization. A context-free grammar (CFG) is formally defined as a 4-tuple $\langle \Sigma, N, P, S \rangle$, where Σ and N are disjoint sets of *terminals* (or the *alphabet*) and *variables*, respectively. P is a set of productions of the form $N ::= (\Sigma \cup N)^*$, and $S \in N$ is the start variable.

As is required in CFL-reachability [Melski and Reps 2000], a CFG is first transformed into *weak Chomsky normal form*, in which each production has at most two symbols (either terminals or variables) on the right-hand side. Accordingly, $X ::= \epsilon$ (ϵ denotes an empty string), $X ::= Y$, and $X ::= Y\ Z$ signify *empty productions*, *unary productions*, and *binary productions*, respectively. Without loss of generality, we henceforth assume the input CFG is already in this form.

Language reachability. CFL-reachability is a critical framework for formulating a wide range of program analysis problems [Reps 1998]. In this formulation, a program analysis (e.g., data flow analysis [Reps et al. 1995]) is specified by a CFG. Meanwhile, the program being analyzed is transformed into an edge-labeled graph $G = \langle V, E \rangle$, with V and E being the node set and the edge set, respectively. To precisely encode program semantics (e.g., data flow and control flow), each edge in E is annotated with a terminal in Σ.

A path $v_0 \xrightarrow{t_1} v_1 \xrightarrow{t_2} v_2 \ldots \xrightarrow{t_n} v_n$ is an *X-reachable path* (or *X-path* for short) if its path string $t_1 t_2 \ldots t_n \in \Sigma^*$ is derivable from $X \in N$. In particular, an S-path is also called a *CFL-reachable path*. The goal of CFL-reachability analysis is to compute the set of S-paths as the final solution.

Algorithm 1: The standard CFL-reachability algorithm

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

Output: A set of pairs $\{(u, v) \mid u \xrightarrow{S} v \in E\}$

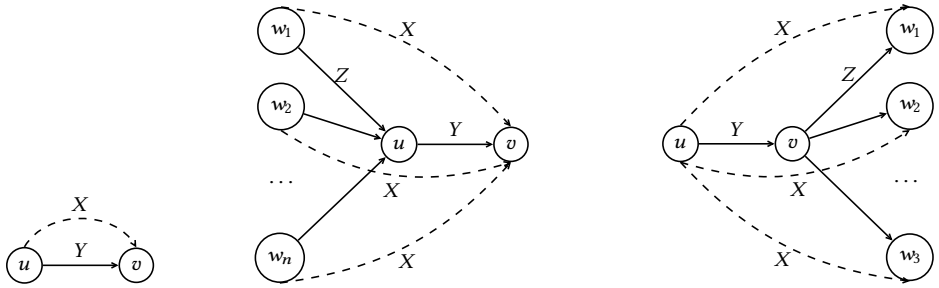
```
1  add E to W
2  for each production X ::= ε ∈ P do
3      for each node v ∈ V do
4          DeriveEdge(v --X--> v)
5  while W ≠ ∅ do
6      u --Y--> v ← Poll(W)
7      for each production X ::= Y ∈ P do
8          DeriveEdge(u --X--> v)
9      for each production X ::= Z Y ∈ P do
10         ΔX ← InE(u, Z) \ InE(v, X)
11         for each w ∈ ΔX do
12             DeriveEdge(w --X--> v)
13     for each production X ::= Y Z ∈ P do
14         ΔX ← OutE(v, Z) \ OutE(u, X)
15         for each w ∈ ΔX do
16             DeriveEdge(u --X--> w)
17 Procedure DeriveEdge (u --X--> v):
18     OutE(u, X) = OutE(u, X) ∪ {v}
19     InE(v, X) = InE(v, X) ∪ {u}
20     add u --X--> v to W
21 return {(u, v) | u --S--> v ∈ E}
```

2.2 Solving CFL-Reachability

Conventionally, CFL-reachability is solved using a dynamic-programming approach [Melski and Reps 2000]. The standard algorithm for CFL-reachability, shown in Algorithm 1, takes as input a CFG and an edge-labeled graph $G = \langle V, E \rangle$, and outputs the set of CFL-reachable paths.

Graph representation. During solving, reachable paths are incrementally discovered, summarized into *summary edges*, and inserted into G to make reachability explicit. The labeled edge set E (or the graph G) is stored in a *bidirectional* adjacency list representation [Chaudhuri 2008; Lei et al. 2022; Shi et al. 2024b], that is, $E = \langle \texttt{InE}, \texttt{OutE} \rangle$. Here, we maintain two label-indexed adjacency structures $\texttt{InE}, \texttt{OutE} : V \times (\Sigma \cup N) \to 2^V$, where $\texttt{InE}(v, X)$ (resp., $\texttt{OutE}(v, X)$) denotes the set of incoming (resp., outgoing) neighbors of v via an X-labeled edge. When inserting an edge $u \xrightarrow{X} v$ (via `DeriveEdge` in Algorithm 1), the incoming X-edge set of v is updated by adding node u, i.e., $u \in \texttt{InE}(v, X)$ (line 19). Similarly, the outgoing X-edge set of u is updated by incorporating node v, i.e., $v \in \texttt{OutE}(u, X)$ (line 18). We refer to $u \in \texttt{InE}(v, X)$ and $v \in \texttt{OutE}(u, X)$ as the *head view* and *tail view* of the edge $u \xrightarrow{X} v$, respectively. This bidirectional representation enables efficient traversal during reachability solving, as demonstrated later.

The solving process. Initially, all edges labeled with terminals are inserted into the worklist W and the edge set E. For each empty production $X ::= \varepsilon$, the algorithm inserts self-loops for all nodes into both W and E by invoking `DeriveEdge` (lines 1–4).



(a) $X ::= Y$ (b) $X ::= Z\ Y$ (c) $X ::= Y\ Z$

Fig. 1. Deriving new edges (dashed X-edges) by applying production rules to existing edges (solid Y- and Z-edges) in Algorithm 1.

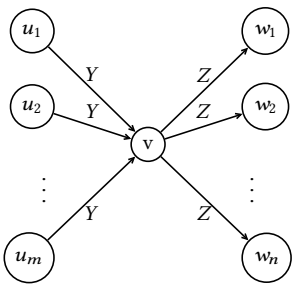


Fig. 2. Illustration of relation chaining based on $X ::= Y\ Z$

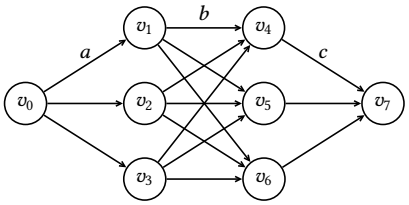
In the main loop, an edge $u \xrightarrow{Y} v$ is extracted and removed from W, and new X-edges are derived using unary productions $X ::= Y$ (lines 6–8). As illustrated in Figure 1a, this yields a new edge $u \xrightarrow{X} v$ (in the dashed line). Next, for each binary production $X ::= Z\ Y$, the incoming Z-edges of node u ($w_i \xrightarrow{Z} u$ for $i \in [1, n]$) are examined, and concatenated with $u \xrightarrow{Y} v$ to produce new X-edges $w_i \xrightarrow{X} v$ (in dashed lines), as depicted in Figure 1b. If an X-edge already exists in G, it is skipped and excluded from the delta set ΔX, as indicated at line 10, $\Delta X = \texttt{InE}(u, Z) \setminus \texttt{InE}(v, X)$. Analogously, for each binary production of the form $X ::= Y\ Z$, the edge $u \xrightarrow{Y} v$ is concatenated with the outgoing Z-edges of node v, that is, $v \xrightarrow{Z} w_i$ for $i \in [1, n]$, to produce new X-edges (Figure 1c). As observed, new edges are stored in the worklist W and later popped to be combined with existing edges in the edge list E. This design enables incremental solving for preventing redundant combinations among old edges. This process is repeated until G becomes saturated, i.e., no new summary edges can be added. As can be seen, E initially contains only terminal edges and is incrementally extended with summary edges, ultimately including the S-edges forming the CFL-reachability solution.

2.3 Motivation

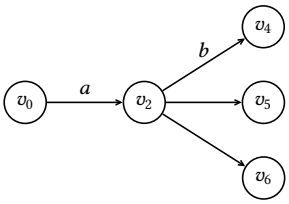
2.3.1 A Relation Chaining Perspective. We begin with a new perspective on solving CFL-reachability, which we term *relation chaining*. Intuitively, solving CFL-reachability (Algorithm 1) fundamentally relies on the application of production rules, as discussed earlier and illustrated in Figure 1. Viewing unary productions as special cases of binary productions, we summarize this process in Figure 2, where consecutive edges whose labels match the patterns specified by the productions (e.g., Y-edges and Z-edges connected via node v) are *chained* to form new summary edges (X-edges). For a label X, treating the X-edge set as a binary relation over $V \times V$ encoding X-reachability for all node

$X ::= a\ b$
$Y ::= b\ c$

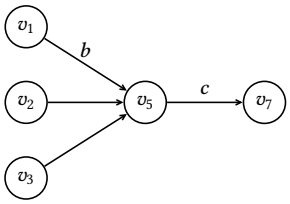
(a) Context-free grammar



(b) G_0: the input graph



(c) G_1: a sub-graph of G_0 illustrating $X ::= a\ b$



(d) G_2: a sub-graph of G_0 illustrating $Y ::= b\ c$

Productions	Graph	Skewed [Lei et al. 2024]	Pearl [Shi et al. 2023]	Squid (Our work)
$X ::= a\ b$	G_1	4	3	1
	G_0	12	9	3
$Y ::= b\ c$	G_2	4	1	1
	G_0	12	3	3
$X ::= a\ b$ $Y ::= b\ c$	G_0	24	12	6

(e) The number of chaining operations for deriving X- and Y-edges

Fig. 3. A motivating example.

pairs, this chaining can be formalized as the natural join of Y and Z on the intermediate node v. Applying the production $X ::= Y\ Z$ gives:

$$\{\, u \xrightarrow{X} w \mid \exists v.\ u \xrightarrow{Y} v \in E \wedge v \xrightarrow{Z} w \in E \,\} \subseteq E.$$

In this paper, we abstract the process of chaining a *pivot edge* with its adjacent edges into a binary relation operation, referred to as *relation chaining* (Theorem 3.1). For example, in Figure 2, choosing each Y-edge as the pivot allows all Z-edges to be chained in a single operation (yielding m operations in total), whereas choosing each Z-edge as the pivot allows all Y-edges to be chained in a single operation (yielding n operations in total).

2.3.2 An Illustrating Example. Figure 3 presents an illustrative example motivating our research. Figure 3a shows a simple CFG with two productions, $X ::= a\ b$ and $Y ::= b\ c$. The edge-labeled graph G_0, depicted in Figure 3b, is slightly more involved: it contains three a-edges from v_0 to v_1, v_2, and v_3; three c-edges from v_4, v_5, and v_6 to v_7; and nine b-edges connecting v_1, v_2, and v_3 to v_4, v_5, and v_6. For readability, only one representative edge of each label is shown, although multiple edges of the same type exist. In practice, derivations of X-edges and Y-edges are usually intertwined and involve more complex productions. Nevertheless, this simplified grammar suffices to highlight the essence of different chaining strategies.

To simplify the presentation, we extract two subgraphs of G_0, G_1 and G_2, shown in Figure 3c and Figure 3d, respectively, to illustrate the relation-chaining process for $X ::= a\ b$ and $Y ::= b\ c$. We then compare our approach, SQUID, with two representative state-of-the-art solvers: SKEWED [Lei et al. 2024], which employs conventional chaining, and PEARL [Shi et al. 2024b], which adopts set-constraint-based chaining. The key distinction among these strategies lies in how they organize relations and perform chaining operations. Our comparison is based on the number of chaining operations, a metric that is agnostic to a solver's internal strategy. Specifically, each chaining operation is defined as the application of a pivot edge to an edge set (Theorem 3.1) and corresponds to a bit-vector operation, which is measured uniformly across all solvers.

Standard chaining. The standard method, though conceptually simple, has been widely adopted in both the classic algorithm [Melski and Reps 2000] and its optimized variants [Chaudhuri 2008; Lei et al. 2022; Xu et al. 2024] including a leading solver SKEWED [Lei et al. 2024]. While its high-level idea has been outlined in Section 2.2, we now use this example to illustrate its inefficiency in detail, with the number of chaining operations under different strategies reported in Figure 3e.

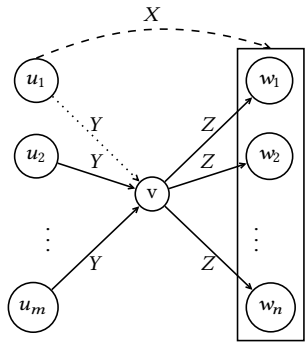
In subgraph $G_1 \subseteq G_0$, applying production $X ::= a\ b$ entails four chaining operations, each using a different edge as the pivot. For example, when the a-edge $v_0 \xrightarrow{a} v_2$ serves as the pivot, it is chained *forward* with the three outgoing b-edges of v_2 ($v_2 \xrightarrow{b} \{v_4, v_5, v_6\}$), yielding three X-edges. Conversely, in the remaining three operations, each b-edge $v_2 \xrightarrow{b} v_i$ ($i \in [4, 6]$) serves as the pivot and is chained *backward* with $v_0 \xrightarrow{a} v_2$, producing $v_0 \xrightarrow{X} v_i$. Notably, each X-edge is derived twice—once with each pivot edge—resulting in one redundant derivation, which introduces redundancy because both edge types appear in W and E.

Extending this reasoning to the entire graph G_0, since v_1 and v_3 have the same incident edges as v_2, deriving all X-edges requires 12 operations (4×3). Similarly, applying $Y ::= b\ c$ incurs 4 operations on G_2 and 12 on G_0, leading to a total of 24 operations when both productions are applied to G_0.

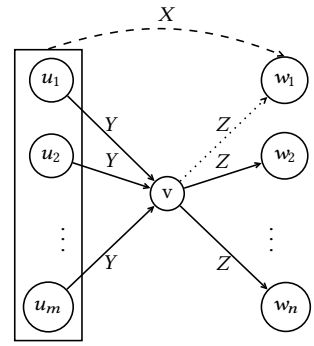
Chaining based on set constraints. A recent approach [Shi et al. 2023, 2024b], PEARL, improves upon this by adopting a multi-derivation strategy under a set-constraint transformation [Kodumal and Aiken 2004; Melski and Reps 2000]. For $X ::= a\ b$, each edge $v_2 \xrightarrow{b} v_i$ with $i \in [4, 6]$ introduces a constraint $R(a, v_2) \subseteq R(X, v_i)$, where $R(A, v) = \{u \mid u \xrightarrow{A} v \in E\}$. Each such constraint corresponds to a single backward chaining operation, yielding 3 operations for G_1 and 9 for G_0 (3×3). Similarly, applying $Y ::= b\ c$ requires 1 operation on G_2 and 3 on G_0. Overall, solving CFL-reachability on G_0 with both productions requires only 12 operations—cutting the total number in half compared to standard chaining.

Insight and our work. A relation-chaining operation may yield varying numbers of newly derived summary edges. Since the total number of summary edges is fixed, deriving more edges per operation can substantially reduce both the number of chaining operations and the fixed-point iterations for faster saturation, thereby improving the efficiency of CFL-reachability solving. This efficiency crucially depends on the choice of pivot edges. For instance, in G_1, selecting $v_0 \xrightarrow{a} v_2$ as the pivot for $X ::= a\ b$ enables all outgoing b-edges of v_2 to be processed in a single operation. Conversely, in G_2, choosing $v_5 \xrightarrow{c} v_7$ as the pivot for $Y ::= b\ c$ allows one operation to handle all three incoming b-edges to v_5. In practice, however, no pivot choice is universally optimal, even for a single production, as summary edges are dynamically inserted into the labeled graph G and edge distributions may vary significantly across nodes.

Building on this insight, we present a novel reachability algorithm, SQUID, which leverages two simple yet effective chaining techniques—adaptive chaining and differential chaining—together



(a) Forward chaining using $u_1 \xrightarrow{Y} v$ as the pivot edge to chain all Z-edges. Other Y-edges can be processed similarly.



(b) Backward chaining using $v \xrightarrow{Z} w_1$ as the pivot edge to chain all Y-edges. Other Z-edges can be processed similarly.

Fig. 4. Two types of chaining operation based on $X ::= Y\ Z$, continuing from Figure 2.

with an improved graph representation, as existing graph models are unsuitable for Squid. In this motivating example, Squid requires only six chaining operations in total, achieving reductions of 75% and 50% compared to Std and Pearl, respectively. Despite its conceptual simplicity, this approach delivers substantial reductions in chaining operations and significant speedups over prior work, including both optimized standard solvers [Lei et al. 2024] and the set-constraints-based chaining approach Pearl [Shi et al. 2024b], across a range of mainstream program analyses.

3 Methodology

This section elaborates on the design of Squid. Section 3.1 formally defines relation chaining and introduces two chaining techniques: adaptive chaining and differential chaining. Section 3.2 presents Squid, our new algorithm for solving CFL-reachability through efficient chaining. Section 3.3 formally proves the correctness of Squid.

3.1 Relation Chaining

3.1.1 Basic Idea. In this work, we conceptualize production application in CFL-reachability as *relation chaining*, defined as follows:

Definition 3.1 (Relation Chaining)**.** Given a production $X ::= Y\ Z$, where $X \in N$ and $Y, Z \in (\Sigma \cup N)$, a relation chaining operation applies to a *pivot edge* together with a set of adjacent edges to derive X-edges. It can be categorized into $(u, v \in V)$:

- **Forward Chaining**: applied to a pivot edge $u \xrightarrow{Y} v$ and the set of outgoing Z-edges of v;
- **Backward Chaining**: applied to a pivot edge $u \xrightarrow{Z} v$ and the set of incoming Y-edges of u.

Example 3.2. In the illustrative graph of Figure 2, applying $X ::= Y\ Z$ derives the set of output X-edges:

$$\{\, u_i \xrightarrow{X} w_j \mid i \in [1, m],\ j \in [1, n] \,\}.$$

This derivation can be achieved through m forward chaining operations. An example is shown in Figure 4a, where the pivot edge $u_1 \xrightarrow{Y} v$ (the dotted line) is chained forward with v's outgoing Z-edges, $\{v \xrightarrow{Z} w_i \mid i \in [1, n]\}$, effectively deriving n X-edges originating from node u_1, $\{u \xrightarrow{X} w_i \mid$

Algorithm 2: Adaptive relation chaining

Input: $\widetilde{u}$ and $\widetilde{w}$ denote the sets of source and target nodes, respectively, $X \in N$ is a variable

Output: A set of newly derived X-edges$\{u \xrightarrow{X} w \mid u \in \widetilde{u}, w \in \widetilde{w}\}$

```
1 Procedure AdaptiveChain (ũ, w̃, X):
2     if |ũ| ≥ |w̃| then
3         for w ∈ w̃ do
4             DeriveIn (ũ -X-> w)                    /* backward chaining */
5     else
6         for u ∈ ũ do
7             DeriveOut (u -X-> w̃)                   /* forward chaining */
```

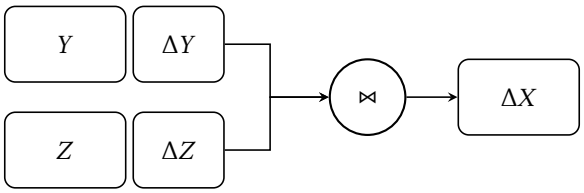


Fig. 5. Relation chaining for $X = YZ$. The operator $\bowtie$ applies $X ::= YZ$ to produce the new incremental relation ΔX.

$i \in [1, n]\}$. Similarly, any other Y-edge $u_i \xrightarrow{Y} v$ for $i \in [2, m]$ can serve as a pivot to trigger another forward chaining operation.

Alternatively, the same result can be obtained via n backward chaining operations. As illustrated in Figure 4b, the pivot edge $v \xrightarrow{Z} w_1$ (the dotted line) is chained backward with u's incoming Y-edges, $\{u_i \xrightarrow{Y} v \mid i \in [1, m]\}$. Likewise, any other Z-edge $v \xrightarrow{Z} w_i$ for $i \in [2, n]$ can serve as a pivot for a backward chaining operation.

3.1.2 Adaptive Chaining. As observed in Theorem 3.2, different chaining strategies result in different numbers of operations, although they eventually yield the same reachability results. Intuitively, performing fewer chaining operations leads to fewer iterations, thereby accelerating the saturation of reachability information. Motivated by this observation, we propose a simple yet effective technique called *adaptive (relation) chaining*, as shown in Algorithm 2.

In procedure `AdaptiveChain`, let v denote the intermediate node connecting $\widetilde{u}$, the set of source nodes, and $\widetilde{w}$, the set of target nodes. Conceptually, this procedure can be viewed as performing a Cartesian product between $\widetilde{u}$ and $\widetilde{w}$. The algorithm determines the chaining direction by simply comparing the sizes of these two endpoint sets. Specifically, if $|\widetilde{u}| \geq |\widetilde{w}|$, each node $w \in \widetilde{w}$ is selected, and its corresponding edge $v \xrightarrow{Y} w$ serves as the pivot edge for backward chaining; otherwise, forward chaining is applied to each node $u \in \widetilde{u}$. In this manner, only $\min(|\widetilde{u}|, |\widetilde{w}|)$ chaining operations are performed.

3.1.3 Differential Chaining. To prevent derivation redundancy, we adopt a *differential chaining* technique, similar to the *difference propagation* technique in pointer analysis [He et al. 2022; Pearce

et al. 2003; Sridharan and Fink 2009] and the *seminaive evaluation* method from the Datalog community [Abiteboul et al. 1995; Jordan et al. 2016].

During the iterative solving of CFL-reachability, summary edges are incrementally derived. To distinguish newly generated information (not processed yet) from existing results, we divide Y-edges into the existing part Y and the incremental part ΔY (and analogously for Z-edges), as illustrated in Figure 5.

Let $\bowtie$ denote the *relation-chaining operator*, which concatenates two sets of edges according to the production rules. For a binary production $X ::= Y\ Z$, the chaining process can be expressed as:

$$\Delta X = (Y \cup \Delta Y) \bowtie (Z \cup \Delta Z) = (Y \bowtie \Delta Z) \cup (\Delta Y \bowtie Z) \cup (\Delta Y \bowtie \Delta Z) \tag{1}$$

This equation explicitly separates the contributions of existing and newly derived edges into three sub-expressions, thereby excluding $Y \bowtie Z$ to avoid redundant chaining between existing Y- and Z-edges, which has already been performed in previous iterations. Hereafter, we define $\bowtie$ to have higher precedence than $\cup$ to improve readability by reducing the need for parentheses. This expression also implicitly demonstrates the distributivity of relation chaining, meaning that each sub-expression can be computed independently.

As an optimization, the sub-expression $\Delta Y \bowtie \Delta Z$ can be omitted in a standard *single-threaded* implementation, since ΔY and ΔZ are processed sequentially. For instance, if ΔY is processed first, its edges are incorporated into Y and subsequently captured by $Y \bowtie \Delta Z$. This optimization mitigates the redundancy in standard chaining discussed in Section 2.3.2.

As will be discussed later, we further encode ΔY, into Δ`InE` and Δ`OutE` to enable efficient incremental derivation during reachability solving. As such, Equation (1) can be rewritten to:

$$\Delta X = \bigcup_{v \in V} \Big(\texttt{InE}(v, Y) \bowtie \Delta\texttt{OutE}(v, Z) \ \cup\ \Delta\texttt{InE}(v, Y) \bowtie \texttt{OutE}(v, Z) \Big). \tag{2}$$

3.1.4 Enhanced Graph Representation. To support the two chaining techniques described above, we introduce an enhanced graph representation that extends the conventional adjacency-list model $E = \langle \texttt{InE}, \texttt{OutE} \rangle$ by incorporating Δ`InE` and Δ`OutE`. These structures capture the incremental (*delta*) portions of the graph, inherently enabling efficient differential chaining.

Different from existing models [Melski and Reps 2000; Shi et al. 2024b], each newly derived Y-edge $u \xrightarrow{Y} v$ is represented from two complementary perspectives: a *head view* ($u \in \Delta\texttt{InE}(v, Y)$) and a *tail view* ($v \in \Delta\texttt{OutE}(u, Y)$), consistent with the representation of old edges (Section 2.2). This dual-view design enables separate handling of the two types of productions involving new Y-edges: for $X ::= Z\ Y$, the operation $\texttt{InE}(v, Z) \bowtie \Delta\texttt{OutE}(v, Y)$ is performed; for $X ::= Y\ Z$, the operation $\Delta\texttt{InE}(v, Y) \bowtie \texttt{OutE}(v, Z)$ is applied. This separation supports adaptive chaining over two sets for improved computational efficiency.

The implementation is shown in Algorithm 3. When a set of incoming X-edges of node v ($\widetilde{u} \xrightarrow{X} v$) is derived and inserted into E via `DeriveIn`, edges already existing in $\texttt{InE}(v, X)$ or $\Delta\texttt{InE}(v, X)$ are excluded (line 2). The remaining newly generated edges $\Delta\widetilde{u} \xrightarrow{X} v$ are then inserted into both Δ`InE` and W_{in} (lines 4–5), as well as into Δ`OutE` and W_{out} (lines 7–8), to ensure bidirectional consistency. Symmetrically, `DeriveOut` performs the corresponding operation for new outgoing X-edges $u \xrightarrow{X} \widetilde{v}$.

It should be noted that existing approaches [Lei et al. 2024; Shi et al. 2024b] also employ incremental derivation. However, they do not distinguish between the two views of newly derived edges, making them unsuitable for our method. In the standard algorithm (Algorithm 1) adopted by Skewed [Lei et al. 2024], each new edge is inserted into the worklist W and subsequently chained with existing edges, either forward or backward, depending on the production rule.

Algorithm 3: Edge derivation and insertion by SQUID

Procedure DeriveIn $(\widetilde{u} \xrightarrow{X} v)$:
- $\Delta\widetilde{u} \leftarrow (\widetilde{u} \setminus \mathtt{InE}(v, X)) \setminus \Delta\mathtt{InE}(v, X)$
- **if** $\Delta\widetilde{u} = \emptyset$ **then return**
- $\Delta\mathtt{InE}(v, X) \leftarrow \Delta\mathtt{InE}(v, X) \cup \Delta\widetilde{u}$
- $W_{in} \leftarrow W_{in} \cup \{\langle v, X\rangle\}$
- **for** $u \in \Delta\widetilde{u}$ **do**
  - $\Delta\mathtt{OutE}(u, X) = \Delta\mathtt{OutE}(u, X) \cup \{v\}$
  - $W_{out} \leftarrow W_{out} \cup \{\langle u, X\rangle\}$

Procedure DeriveOut $(u \xrightarrow{X} \widetilde{v})$:
- $\Delta\widetilde{v} \leftarrow (\widetilde{v} \setminus \mathtt{OutE}(u, X)) \setminus \Delta\mathtt{OutE}(u, X)$
- **if** $\Delta\widetilde{v} = \emptyset$ **then return**
- $\Delta\mathtt{OutE}(u, X) \leftarrow \Delta\mathtt{OutE}(u, X) \cup \Delta\widetilde{v}$
- $W_{out} \leftarrow W_{out} \cup \{\langle u, X\rangle\}$
- **for** $v \in \Delta\widetilde{v}$ **do**
  - $\Delta\mathtt{InE}(v, X) = \Delta\mathtt{InE}(v, X) \cup \{u\}$
  - $W_{in} \leftarrow W_{in} \cup \{\langle v, X\rangle\}$

Algorithm 4: CFL-reachability via efficient relation chaining

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

Output: A set of pairs $\{(u, v) \mid u \xrightarrow{S} v \in E\}$

- **for** *each edge* $u \xrightarrow{t} v$ **do**
  - $\mathtt{DeriveIn}(\{u\} \xrightarrow{X} v)$
- **for** *each production* $X ::= \varepsilon \in P$ **do**
  - **for** *each node* $v \in V$ **do**
    - $\mathtt{DeriveIn}(\{v\} \xrightarrow{t} v)$
- **while** $W_{in} \neq \emptyset \vee W_{out} \neq \emptyset$ **do**
  - **while** $W_{in} \neq \emptyset$ **do**
    - $\langle v, Y\rangle \leftarrow$ **Poll**(W_{in})
    - $\widetilde{u} \leftarrow \mathtt{FlushIn}(v, Y)$
    - **for** *each production* $X ::= Y \in P$ **do**
      - $\mathtt{DeriveIn}(\widetilde{u} \xrightarrow{X} v)$
    - **for** *each production* $X ::= Y\ Z \in P$ **do**
      - $\mathtt{AdaptiveChain}(\widetilde{u}, \mathtt{OutE}(v, Z), X)$
  - **while** $W_{out} \neq \emptyset$ **do**
    - $\langle u, Y\rangle \leftarrow$ **Poll**(W_{out})
    - $\widetilde{v} \leftarrow \mathtt{FlushOut}(u, Y)$
    - **for** *each production* $X ::= Z\ Y \in P$ **do**
      - $\mathtt{AdaptiveChain}(\mathtt{InE}(u, Z), \widetilde{v}, X)$

Procedure FlushIn (v, X):
- $\Delta\widetilde{u} \leftarrow \Delta\mathtt{InE}(v, X)$
- $\Delta\mathtt{InE}(v, X) \leftarrow \emptyset$
- $\mathtt{InE}(v, X) \leftarrow \mathtt{InE}(v, X) \cup \Delta\widetilde{u}$
- **return** $\Delta\widetilde{u}$

Procedure FlushOut (u, X):
- $\Delta\widetilde{v} \leftarrow \Delta\mathtt{OutE}(u, X)$
- $\Delta\mathtt{OutE}(u, X) \leftarrow \emptyset$
- $\mathtt{OutE}(u, X) \leftarrow \mathtt{OutE}(u, X) \cup \Delta\widetilde{v}$
- **return** $\Delta\widetilde{v}$

return $\{(u, v) \mid u \xrightarrow{S} v \in E\}$

By comparison, PEARL [Shi et al. 2024b] adopts a multi-derivation strategy to mitigate derivation redundancy. Let us consider the process of handling newly generated Y-edges. For a production $X ::= Y\ Z$, all newly derived incoming Y-edges of a node v are grouped and chained forward with each existing outgoing Z-edge of v. However, when v has substantially more outgoing Z-edges

than incoming Y-edges, this strategy incurs a large number of redundant chaining operations, as confirmed by our experimental evaluation (Section 4). The chaining strategy for productions of the form $X ::= Z\ Y$ involving new Y-edges remains essentially identical to the standard approach.

In summary, existing chaining strategies are static, leaving room for further optimization through adaptive chaining.

3.2 The Overall Algorithm

Algorithm 4 presents our new algorithm for solving CFL-reachability. During initialization, similar to Algorithm 1, edges labeled by terminals (lines 1–2) and those produced by empty productions (lines 3–5) are inserted into the graph G.

In the main loop, the algorithm iteratively performs relation chaining until no new edges can be added to G. This fixed-point calculation essentially realizes Equation (2). Two worklists are maintained: W_{in} (resp. W_{out}), where $\langle v, Y\rangle \in W_{in}$ (resp. $\langle u, Y\rangle \in W_{out}$) indicates that some of v's incoming (resp. u's outgoing) Y-edges remain to be processed.

In the first inner loop, new Y-edges ending at node v, i.e., $\widetilde{u} \xrightarrow{Y} v$, are fetched from $\Delta\texttt{InE}$ (the head view of Y-edges; lines 8–9). Subsequently, new X-edges are derived via unary productions $X ::= Y$ (lines 10–11). At lines 12–13, for binary productions $X ::= Y\ Z$, these Y-edges are chained with existing Z-edges outgoing from v using the `AdaptiveChain` operator defined in Algorithm 2.

Analogously, the second inner loop (lines 14–18) processes the tail view of newly derived Y-edges for binary productions $X ::= Z\ Y$.

Finally, we note that this new algorithm shares the same asymptotic cubic time complexity and quadratic space complexity as Algorithm 1. However, it significantly reduces chaining operations through our proposed chaining techniques, as confirmed by our evaluation in Section 4.

3.3 Correctness

This subsection establishes the correctness of Squid.

Lemma 3.3 (Termination)**.** Algorithm 4 terminates.

Proof. The total number of possible edges is bounded by $O(N^2)$. Each edge $u \xrightarrow{X} v$ is inserted into the graph G by adding node u to $\Delta\texttt{InE}(v, X)$ and subsequently flushing it into $\texttt{InE}(v, X)$ (and analogously for `OutE` and Δ`OutE`). Each insertion occurs at most once, as enforced by lines 2 and 10 in Algorithm 3. Consequently, the total number of worklist iterations triggered by edge insertions for both W_{in} and W_{out} is finite, and the algorithm is therefore guaranteed to terminate.

Lemma 3.4 (Reachability Solving)**.** Squid correctly derives all summary edges.

Proof. Solving CFL-reachability requires correctly handling three types of productions: empty, unary, and binary. Empty and unary productions are handled in lines 3–5 and 10–11 of Algorithm 4, similar to Algorithm 1.

Let us prove the correctness for binary productions. Consider a production $X ::= Y\ Z$. We need to show that for any nodes u, v, w, $u \xrightarrow{Y} v \in G \ \wedge\ v \xrightarrow{Z} w \in G \ \Rightarrow\ u \xrightarrow{X} w \in G$.

We first discuss the correctness of differential chaining. Based on the enhanced graph representation, an edge is added to $\Delta\texttt{InE}$ and $\Delta\texttt{OutE}$ upon derivation (Algorithm 3) and later flushed into `InE` and `OutE` during processing (lines 19–28 of Algorithm 4). Equation (1) performs relation chaining differentially by excluding the expression $Y \bowtie Z$, which has already been computed in previous iterations. Now consider Equation (2). Since the two edges are processed sequentially, it suffices to consider two cases based on their processing order. Suppose that $u \xrightarrow{Y} v$ is processed after $v \xrightarrow{Z} w$. In this case, when processing $u \xrightarrow{Y} v$, we have $u \in \Delta\texttt{InE}(v, Y)$ and $w \in \texttt{OutE}(v, Z)$. A

symmetric argument applies when $u \xrightarrow{Y} v$ is processed before $v \xrightarrow{Z} w$, in which case $u \in \texttt{InE}(v, Y)$ and $w \in \Delta\texttt{OutE}(v, Z)$. Hence, the computation of $\Delta\texttt{InE}(v, Y) \bowtie \Delta\texttt{OutE}(v, Z)$ can be safely skipped.

We next consider adaptive chaining (Algorithm 2), which joins the corresponding in- and out-edge sets identified above, namely, $\Delta\texttt{InE}(v, Y) \bowtie \texttt{OutE}(v, Z)$ and $\texttt{InE}(v, Y) \bowtie \Delta\texttt{OutE}(v, Z)$. Adaptive chaining selects either $u \xrightarrow{Y} v$ or $v \xrightarrow{Z} w$ as the pivot edge based on the sizes of the input sets, and performs chaining in the corresponding direction. Regardless of the chosen pivot, adaptive chaining exhaustively enumerates all such joinable pairs and derives the summary edge $u \xrightarrow{X} w$, thereby establishing the correctness of CFL-reachability for binary productions.

Hence, all summary edges required by CFL-reachability are correctly derived.

Theorem 3.5 (Correctness)**.** Squid accurately solves CFL-reachability.

Proof. By combining Theorem 3.3 and Theorem 3.4, we conclude that Squid derives all summary edges upon termination, and thus correctly solves CFL-reachability.

4 Evaluation

We first evaluate Squid on three major clients: field-sensitive alias analysis [Zhang et al. 2014; Zheng and Rugina 2008] and value-flow analysis [Sui et al. 2012, 2014] for C/C++, as well as field-sensitive points-to analysis for Java [Sridharan and Bodík 2006; Sridharan et al. 2005; Xu et al. 2009], all extensively studied in recent work [Lei et al. 2024, 2022; Shi et al. 2024b; Wang et al. 2017; Xu et al. 2024]. In addition, we conduct a case study using taint analysis [Kodumal and Aiken 2004] to detect memory leak issues in C/C++ programs [Sui et al. 2014].

Our evaluation demonstrates that Squid effectively enhances CFL-reachability-based analyses, achieving substantial speedups over two state-of-the-art solvers, Pearl [Shi et al. 2024b] and Skewed [Lei et al. 2024]. Specifically, Skewed, an optimized variant of the standard algorithm [Melski and Reps 2000], employs a *skewed tabulation* technique that integrates static and dynamic skewing. Static skewing is a grammar transformation that rewrites the CFG to avoid redundant edge generation and insertion induced by recursive nonterminals. In contrast, dynamic skewing refrains from inserting certain types of summary edges (called *propagation edges*) into the graph while retaining them in the worklist to preserve soundness. In contrast, Pearl [Shi et al. 2023, 2024b] uses a multi-derivation optimization strategy, as detailed in Section 3.1. For our evaluation, we apply static skewing uniformly to all solvers as a grammar rewriting step, ensuring a fair comparison of their respective online techniques. We also evaluate a popular Datalog solver, Soufflé (version `2.4`) [Jordan et al. 2016], which synthesizes optimized solvers in C++ based on a given Datalog program (converted from normalized CFGs in Figures 6b, 7b and 8b). We enable the OpenMP feature for parallel execution, and use a 8-thread version (with the option "`-j 8`").

Our experiments aim to answer the following research questions (RQs):

- **RQ1:** To what extent does Squid reduce derivation redundancy—in terms of chaining operations—compared with existing CFL-reachability solvers across three representative analyses?
- **RQ2:** Can Squid improve the scalability and efficiency of CFL-reachability analysis over existing solvers?
- **RQ3:** How do adaptive chaining and differential chaining contribute to the chaining process?
- **RQ4:** Can Squid improve the efficiency of memory leak analysis, a key CFL-reachability-based application?
- **RQ5:** Can Squid improve the efficiency of Skewed, an leading CFL-reachability method?

$$
\begin{array}{lcl}
V & ::= & \overline{A}\ V \mid V\ A \mid \overline{f_i}\ V\ f_i \mid M \mid \epsilon \\
M & ::= & \overline{d}\ V\ d \\
A & ::= & a\ M? \\
\overline{A} & ::= & M?\ \overline{a}
\end{array}
$$

(a) Original grammar

$$
\begin{array}{lcl}
V & ::= & \overline{A}\ V \mid \overline{a}\ V \mid V\ A \mid V\ a \\
 & & \mid FV_i\ f_i \mid M \mid \epsilon \\
FV_i & ::= & \overline{f_i}\ V \\
M & ::= & DV\ d \\
DV & ::= & \overline{d}\ V \\
A & ::= & a\ M \\
\overline{A} & ::= & M\ \overline{a}
\end{array}
$$

(b) Normalized grammar

Fig. 6. Original and normalized CFGs formulating alias analysis for C/C++, with V as the start variable.

$$
A \quad ::= \quad (\!|_i\ A\ |\!)_i \mid a)^*
$$

(a) Original grammar

$$
\begin{array}{lcl}
A & ::= & A\ B \mid A\ a \mid a \mid \varepsilon \\
B & ::= & CA_i\ |\!)_i \\
CA_i & ::= & (\!|_i\ A
\end{array}
$$

(b) Normalized grammar

Fig. 7. Original and normalized CFGs formulating value-flow analysis for C/C++, with A as the start variable.

$$
\begin{array}{lcl}
\mathit{FlowsTo} & ::= & \mathit{new} \mid \mathit{FlowsTo}\ \mathit{Assign} \\
\overline{\mathit{FlowsTo}} & ::= & \overline{\mathit{new}} \mid \overline{\mathit{Assign}}\ \overline{\mathit{FlowsTo}} \\
\mathit{Assign} & ::= & a \mid \mathit{put}_f\ \mathit{Alias}\ \mathit{get}_f \\
\overline{\mathit{Assign}} & ::= & \overline{a} \mid \overline{\mathit{get}_f}\ \mathit{Alias}\ \overline{\mathit{put}_f} \\
\mathit{Alias} & ::= & \overline{\mathit{FlowsTo}}\ \mathit{FlowsTo}
\end{array}
$$

(a) Original grammar

$$
\begin{array}{lcl}
\mathit{FlowsTo} & ::= & \mathit{new} \mid \mathit{FlowsTo}\ \mathit{Assign} \\
\mathit{Assign} & ::= & a \mid \mathit{PutAlias}\ \mathit{get}_f \\
\mathit{PutAlias} & ::= & \mathit{put}_f\ \mathit{Alias} \\
\overline{\mathit{FlowsTo}} & ::= & \overline{\mathit{new}} \mid \overline{\mathit{Assign}}\ \overline{\mathit{FlowsTo}} \\
\overline{\mathit{Assign}} & ::= & \overline{a} \mid \overline{\mathit{get}_f}\ \overline{\mathit{PutAlias}} \\
\overline{\mathit{PutAlias}} & ::= & \mathit{Alias}\ \overline{\mathit{put}_f} \\
\mathit{Alias} & ::= & \overline{\mathit{FlowsTo}}\ \mathit{FlowsTo}
\end{array}
$$

(b) Normalized grammar

Fig. 8. Original and normalized CFGs formulating points-to analysis for Java, with *FlowsTo* as the start variable.

4.1 Evaluated Grammars

Field-sensitive alias analysis for C/C++. Figure 6a shows the standard CFG defining alias relations between program expressions in C/C++ [Zhang et al. 2014; Zheng and Rugina 2008]. This analysis is performed over program expression graphs (PEGs). In this grammar, the terminals a, d, f_i, and ϵ represent assignment, pointer dereferencing, the i-th object field, and the empty string, respectively. The variables include the start variable V (value alias), indicating expressions that may evaluate to the same pointer address; M (memory alias), denoting expressions that may refer to the same memory location; and A, which captures both direct ($A ::= a$) and indirect ($A ::= a\ M$) assignments. The PEG is bidirectional [Reps 1998]: for any edge $u \xrightarrow{t} v$, a corresponding reversed edge $v \xrightarrow{\overline{t}} u$ exists, introducing symbols such as $\overline{a}$, $\overline{d}$, $\overline{f_i}$, and $\overline{A}$. From the normalized grammar Figure 6b, FV_i and DV are two variables introduced for normalization.

Context-sensitive value-flow analysis for C/C++. Figure 7 presents the original and normalized CFGs for value-flow analysis in C/C++ [Li et al. 2011; Shi et al. 2018; Sui et al. 2012; Yao et al. 2024], equivalent to the extensively studied Dyck language [Xu et al. 2009; Zhang et al. 2013; Zhang and Su 2017]. This analysis uses a sparse value-flow graph (SVFG), where each edge is annotated with three types of terminals: a for assignment, and $(\!|_i$ and $|\!)_i$ for the i-th call site and return site,

Table 1. Benchmark statistics across three analyses.

(a) The number of nodes and edges in PEG and SVFG for each C/C++ benchmark

Bench.	PEG		SVFG	
	#Nodes	#Edges	#Nodes	#Edges
cactus	97,996	210,082	75,082	203,827
imagick	81,157	216,072	62,149	129,822
leela	15,426	35,950	14,208	29,824
nab	11,054	24,518	5,172	8,130
omnetpp	153,180	328,398	165,730	1,007,721
parest	73,197	159,668	62,049	123,803
perlbench	99,273	264,522	307,020	1,008,534
povray	50,792	121,774	97,148	278,969
x264	47,432	111,362	21,019	47,460
xz	7,928	17,272	4,489	8,102

(b) The number of classes, reachable methods, nodes, and edges for each Java benchmark

Bench.	#Classes	#RMethods	#Nodes	#Edges
avrora	8,737	12,643	51,747	233,042
batik	11,858	29,378	119,467	706,254
eclipse	10,134	15,992	68,253	365,888
fop	12,871	49,629	221,766	1,476,010
h2	10,712	21,343	91,950	498,916
luindex	8,433	16,994	71,741	319,478
lusearch	8,433	8,719	36,619	169,852
pmd	9,954	24,918	110,558	743,410
sunflow	7,399	16,626	69,452	322,288
tomcat	8,433	9,486	38,816	180,566
tradebeans	9,954	10,330	44,222	202,724
tradesoap	9,954	10,330	44,222	202,724
xalan	10,841	15,062	72,513	350,710

respectively. The start variable A indicates same-level value flows. Two intermediate variables, B and CA_i, are introduced through grammar normalization, demonstrated in Figure 7b.

Field-sensitive points-to analysis for Java. Figure 8b shows the original CFG used for specifying points-to analysis over the pointer assignment graph (PAG) in Java [He et al. 2022; Sridharan et al. 2005]. The alphabet Σ consists of four terminals: new, a, put_f, and get_f, representing heap object allocation, assignment, field write, and field read, respectively. The PAG is bidirectional as well, introducing corresponding inverse terminals $\overline{new}$, $\overline{a}$, $\overline{put_f}$, and $\overline{get_f}$. Among the variables, *FlowsTo* denotes flows-to relations, representing allocated objects flowing to pointers; *Assign* captures direct or indirect assignments; and *Alias* indicates alias relations between two pointers. In the normalized grammar Figure 8b, the variable *PutAlias* is introduced by CFG normalization.

4.2 Evaluation Setup

Experimental settings. All experiments were conducted on a server equipped with dual 12-core Intel(R) Xeon(R) CPUs at 3.00 GHz and 2 TB of memory. Each experiment was executed three times with a 6-hour time limit. We report the mean (M) time and memory usage over three runs

and compute the standard deviation (SD). The maximum coefficient of variation (SD/M) is 4.87%, indicating minimal variance [Westgard nd]. We measure solver performance in terms of time and memory. Execution time is computed using the `C` function `clock()`, which reports the approximate processor time. Memory usage is read from the `VmRSS` field in `/proc/self/status`, which reflects the physical memory occupied at runtime.

Implementation. All solvers, including both the baselines and our approach Squid, are implemented in C++ on top of LLVM 14.0.0 [Lattner and Adve 2004]. We leverage `llvm::SparseBitVector` to build a bidirectional adjacency list representation for storing labeled edges and employ bit-vector set operations to achieve subcubic performance [Chaudhuri 2008]. This implementation design follows prior work [Lei et al. 2024, 2022], ensuring a fair comparison.

Benchmark selection and graph collection. Our benchmark selection follows prior work. For field-sensitive alias analysis and context-sensitive value-flow analysis in C/C++, we selected 10 programs from the SPEC 2017 suite, following established practice [Lei et al. 2022; Shi et al. 2023, 2024b]. These programs are compiled with Clang and linked into LLVM bitcode using `wllvm` [1]. The corresponding PEG and SVFG are then extracted using SVF [Sui and Xue 2016], a widely used static analysis framework for C/C++ programs. Detailed statistics on the number of nodes and edges for PEG and SVFG in each program are provided in Table 1a.

For Java points-to analysis, following recent work [He et al. 2024; Shi et al. 2025; Thiessen and Lhoták 2017], we selected 13 programs from the well-known DaCapo benchmark suite [Blackburn et al. 2006] (version `6cf0380`), excluding the `jython` benchmark due to overly conservative reflection logs, together with the Java library `JRE1.8.0_31` and TamiFlex [Bodden et al. 2011] to handle reflection. Table 1b reports, for each program, the number of classes and reachable methods, as well as the number of nodes and edges in the corresponding PAG, as extracted by a static analysis framework for Java programs [Tan and Li 2023].

Verification of correctness. We empirically validated the correctness of Squid on all C/C++ and Java benchmarks for which baseline solvers are scalable, as shown in Tables 2 to 4. Specifically, we confirmed that all solvers produce identical sets of CFL-reachable paths.

4.3 RQ1: Reduction of Chaining Operations

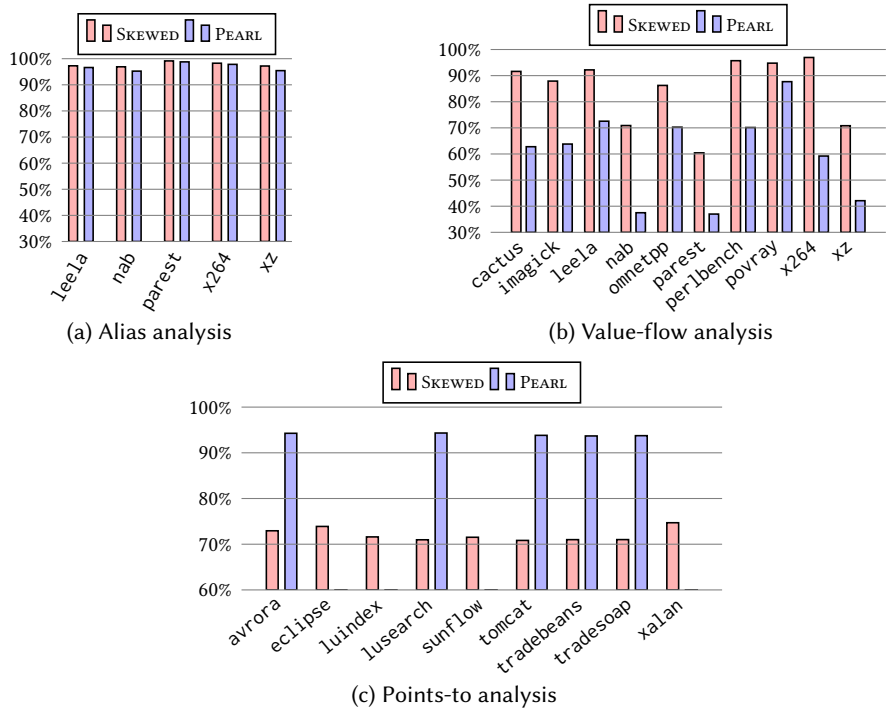
Figure 9 presents the reduction rates of relation chaining operations achieved by Squid for the programs successfully solved by both Squid and the baseline solvers.

Alias analysis. As depicted in Figure 9a, in alias analysis, Squid reduces chaining operations by 97.76% and 96.78% relative to Skewed and Pearl on average, respectively. The reduction rates are also consistent across benchmarks for both baselines.

Value-flow analysis. In value-flow analysis (Figure 9b), Squid eliminates 84.76% and 60.30% chaining operations over Skewed and Pearl, respectively. A closer examination reveals that the reduction rates against Pearl are generally lower than those against Skewed. For example, on `nab` and `parest`, Squid reduces only 37.50% and 36.99% of operations compared with Pearl. This is because Pearl 's multi-derivation technique, which propagates reachability information in batches, already mitigates a large portion of redundant chaining.

Points-to analysis. In points-to analysis (Figure 9c), Squid achieves reduction rates of 72.05% and 93.97% relative to Skewed and Pearl, respectively. Notably, Pearl fails to complete within 6 hours on `eclipse`, `luindex`, `sunflow`, and `xalan`, whereas Skewed successfully solves these benchmarks. For programs successfully solved by both Pearl and Skewed, Pearl performs more redundant chaining operations due to production rules such as $\overline{FlowsTo} ::= \overline{Assign}\ \overline{FlowsTo}$, where

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



(a) Alias analysis

(b) Value-flow analysis

(c) Points-to analysis

Fig. 9. The operation reduction rates of Squid over Skewed and Pearl.

Table 2. Runtime (s) and memory usage (MB) of all solvers on C/C++ alias analysis. A "-" indicates that the analysis did not complete within 6 hours. Bold numbers highlight cases where a baseline outperforms Squid.

Bench.	Skewed		Pearl		Soufflé		Squid	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem
cactus	-	-	-	-	10,242.44	54,885.64	9,749.41	23,699.50
imagick	-	-	-	-	**10,932.20**	99,659.16	14,682.20	63,983.30
leela	208.30	431.13	345.41	500.46	69.14	691.44	24.96	509.90
nab	53.17	258.70	40.16	236.27	18.95	277.46	6.29	239.57
omnetpp	-	-	-	-	-	-	17,304.00	35,553.50
parest	15,631.90	7,292.43	16,915.70	6,138.23	1,503.32	9,957.27	900.78	6,222.68
perlbench	-	-	-	-	-	-	-	-
povray	-	-	-	-	4,754.56	20,149.06	2,940.54	11,093.50
x264	1,480.06	1,833.37	3,541.87	4,662.21	215.46	3,517.77	173.72	4,675.74
xz	12.14	98.67	10.68	140.40	9.03	165.71	3.18	143.33

the cardinality of $\overline{Assign}$ is much smaller than that of $\overline{FlowsTo}$, as discussed in Section 3.1.4. Additionally, the dynamic skewing technique in Skewed helps reduce the insertion of summary edges. As a result, Squid achieves drastic reductions over both Skewed and Pearl, with the improvement being more pronounced relative to Pearl.

4.4 RQ2: Improved Scalability and Efficiency

In this experiment, we evaluate the effectiveness of Squid in improving the scalability and efficiency of CFL-reachability analysis. Tables 2 to 4 report the detailed performance results in terms of runtime

Table 3. Runtime (s) and memory (MB) for four solvers on value-flow analysis for C/C++.

Bench.	Skewed		Pearl		Soufflé		Squid	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem
cactus	90.08	662.48	51.04	834.30	123.80	1,107.27	34.48	872.23
imagick	17.80	344.82	11.83	506.56	38.40	345.75	8.50	521.45
leela	3.04	59.77	1.02	57.47	6.91	80.97	0.58	60.11
nab	0.20	24.81	**0.15**	25.72	0.25	12.72	0.16	27.56
omnetpp	80.84	1,248.25	56.31	1,800.38	111.35	919.37	39.69	2,023.14
parest	2.46	208.91	**1.69**	225.10	2.42	58.43	2.05	235.32
perlbench	5,455.85	7,687.47	2,852.32	7,329.29	5,615.77	31,295.55	1,414.24	7,816.26
povray	282.43	1,498.46	304.41	1,522.96	311.69	3,368.84	73.14	1,835.61
x264	21.77	403.68	5.98	216.44	40.88	464.93	2.96	221.50
xz	0.23	17.67	0.16	21.82	0.29	11.59	0.16	22.75

Table 4. Runtime (s) and memory (MB) for four solvers on points-to analysis for Java.

Bench.	Skewed		Pearl		Soufflé		Squid	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem
avrora	2,333.22	3,590.88	9,854.49	3,566.13	5,503.19	6,118.73	1,364.22	3,832.10
batik	-	-	-	-	-	-	-	-
eclipse	10,280.80	9,364.15	-	-	-	-	6,143.41	9,447.50
fop	-	-	-	-	-	-	-	-
h2	-	-	-	-	-	-	14,597.10	17,134.80
luindex	8,328.15	8,744.03	-	-	16,297.94	14,795.18	4,379.09	7,945.93
lusearch	1,014.93	2,144.48	4,824.66	2,023.43	1,857.87	3,634.74	512.48	2,171.61
pmd	-	-	-	-	-	-	-	-
sunflow	7,998.40	7,063.25	-	-	15,947.05	14,964.98	4,102.73	7,584.37
tomcat	1,204.75	2,331.02	5,418.31	2,248.21	2,201.39	4,048.05	640.99	2,418.41
tradebeans	1,664.41	3,275.63	6,891.97	2,833.34	2,854.14	5,073.34	894.31	3,065.70
tradesoap	1,644.78	3,279.55	7,310.92	2,829.04	2,797.14	5,075.18	833.96	3,059.43
xalan	5,472.60	5,955.54	-	-	8,692.35	10,085.86	2,955.51	6,683.86

(in seconds) and peak memory usage (in megabytes), and Figure 10 presents the corresponding speedups, for three evaluated analyses.

Alias analysis. Table 2 presents the detailed performance statistics for the four solvers. Squid consistently outperforms all baselines across the 10 benchmarks, except for `imagick`, where it shows a slowdown of 0.74× compared with Soufflé. Notably, Squid significantly improves the scalability of CFL-reachability, with only one challenging benchmark, `perlbench`, failing to complete within the time limit. In contrast, Skewed, Pearl, and Soufflé fail to solve 5, 5, and 2 benchmarks, respectively, underscoring the scalability of Squid. Figure 10a reports the speedups achieved by Squid relative to each baseline solver on the programs successfully analyzed by both. On average, Squid achieves speedups of 9.30×, 12.55×, and 1.87× over Skewed, Pearl, and Soufflé, respectively, demonstrating the effectiveness of the proposed chaining techniques.

Value-flow analysis. Table 3 summarizes the results for value-flow analysis. In `nab` and `parest`, Squid performs slightly slower than Pearl due to the noticeably lower reduction rates in chaining operations, as discussed in Section 4.3. Furthermore, in the corresponding CFGs shown in Figure 7b,

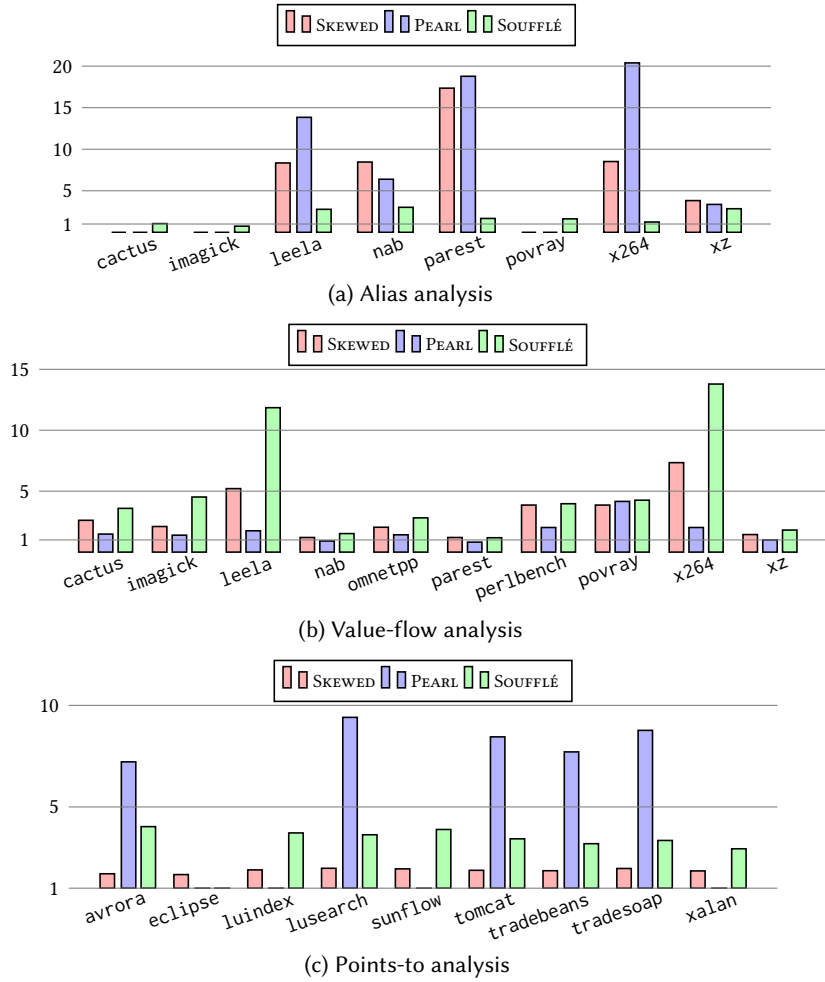


Fig. 10. Speedups of Squid over Skewed, Pearl, and Soufflé.

A-edges constitute the majority of summary edges and require only head views, since A appears only in productions $A ::= A\ B$ and $A ::= A\ a$. As a result, the view separation mechanism (Section 3.1.4) adopted by Squid introduces minor overhead without tangible benefit. Nevertheless, as shown in Figure 10b, Squid still achieves substantial speedups—3.08×, 1.70×, and 4.93× on average—over Skewed, Pearl, and Soufflé, respectively.

Points-to analysis. Lastly, Table 4 reports the performance results for points-to analysis for Java. Among the 13 evaluated programs, Skewed, Pearl, Soufflé, and Squid fail to analyze 4, 7, 5, and 3 benchmarks within 6 hours, respectively, highlighting the strong scalability of Squid. As shown in Figure 10c, Squid achieves speedups of 1.86×, 8.31×, and 3.52× over Skewed, Pearl, and Soufflé, respectively.

These results demonstrate the generality of Squid across diverse CFL-reachability analyses and highlight its substantial benefits in improving solver efficiency. Notably, Squid consistently achieves significant improvements across all three analyses, regardless of their underlying grammars or graph characteristics. This consistency highlights the wide applicability of Squid 's innovative chaining techniques as a general performance optimization for CFL-reachability solvers.

Table 5. Runtime (s) and memory (MB) for four solvers on *context-insensitive* value-flow analysis for C/C++.

Bench.	Skewed		Pearl		Soufflé		Squid	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem
leela	219.14	676.71	172.01	314.02	94.82	2,727.30	125.32	320.70
nab	0.54	29.42	0.25	31.11	0.50	25.10	0.24	34.22
parest	36.02	500.45	22.50	512.62	12.84	388.79	20.94	541.16
x264	2,003.30	3,196.91	1,131.08	1,179.63	593.59	14,379.34	939.13	1,189.64
xz	22.14	116.49	10.74	63.52	13.96	386.23	7.52	66.47

Memory consumption. In terms of memory consumption, Squid is comparable to Pearl. On average, it consumes only 1.39×, 1.18×, and 1.01× of Skewed 's memory usage for alias, value-flow, and points-to analyses, respectively. First, Squid's differential graph representation introduces extra memory overhead. Second, Skewed achieves lower memory consumption by removing many summary edges via dynamic skewing [Lei et al. 2024], a technique orthogonal to our approach that could be integrated into Squid, which is studied in Section 4.7. Compared with Soufflé, Squid uses on average 0.76×, 1.47×, and 0.59× of its memory consumption for the same analyses. Given the substantial speedups achieved by Squid, this level of memory overhead is acceptable.

To further evaluate the generality of Squid, we also performed a realistic, context-insensitive value-flow analysis using a grammar that is neither Dyck-like nor bidirected (unlike the grammars in Figures 6 to 8), yet contains highly skewed productions:

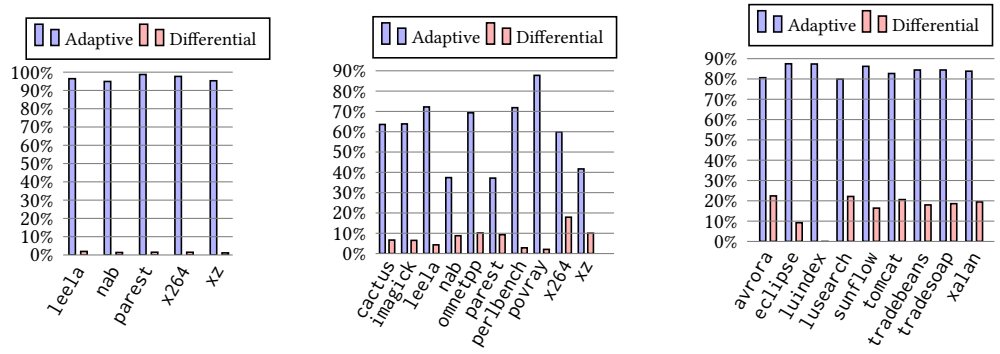
$$A ::= \epsilon \mid A\, a \mid A \,(\!|_{i_i} \mid A\, |\!)_{i_i}.$$

Because context insensitivity introduces substantial spurious value flows, this analysis is more time-consuming. We therefore conducted a small-scale experiment on five benchmarks: `leela`, `nab`, `parest`, `x264`, and `xz`, with the results reported in Table 5. Overall, Squid achieves speedups of 2.16×, 1.23×, and 1.19× and reduces memory usage by 26.77%, −4.64%, and 37.44% relative to Skewed, Pearl, and Soufflé, respectively, demonstrating its generality. Notably, Soufflé outperforms Squid on three benchmarks; while all solvers avoid recursive parenthesis matching for context sensitivity, such recursion would induce dependencies that hinder parallel execution, making its absence especially advantageous for Soufflé.

4.5 RQ3: Ablation Study

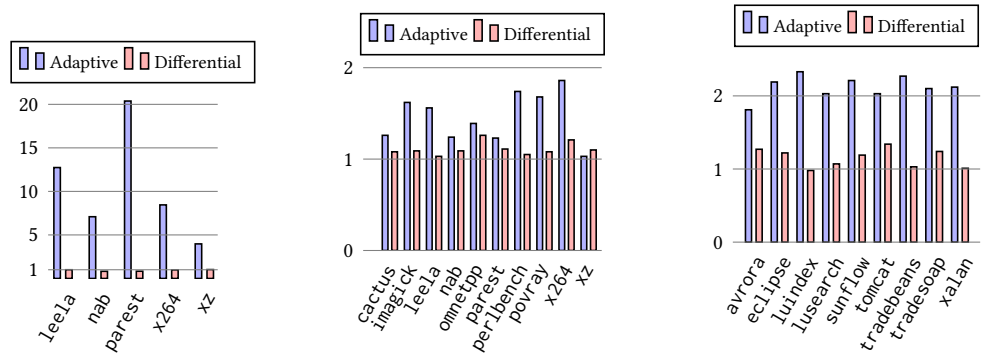
In this RQ, we evaluate the contribution of adaptive chaining and differential chaining via an ablation study. To this end, we design two variants of Squid: Squid$^{\text{-adapt}}$ and Squid$^{\text{-adapt-diff}}$. Specifically, Squid$^{\text{-adapt}}$ disables the adaptive chaining strategy. Implementation-wise, Squid$^{\text{-adapt}}$ always performs backward chaining, regardless of the sizes of $\widetilde{u}$ and $\widetilde{w}$ in Algorithm 2. Alternatively, fixing the chaining direction to forward chaining would exhibit similar behavior. Differential chaining, on the other hand, is tightly coupled with our enhanced graph representation. Consequently, we design Squid$^{\text{-adapt-diff}}$ as a variant of Squid$^{\text{-adapt}}$ that further disables the small optimization illustrated in Section 3.1.3, but retains delta propagation.

Both Squid$^{\text{-adapt}}$ and Squid$^{\text{-adapt-diff}}$ cannot complete five programs that Squid successfully solves—`cactus`, `imagick`, `omnetpp`, and `povray` in alias analysis, as well as `h2` in points-to analysis—showcasing the effectiveness of adaptive chaining and differential chaining in enhancing scalability. For the benchmarks successfully analyzed by both solvers, we quantify the reduction in chaining operations achieved by our techniques and the corresponding speedups, as shown in Figures 11 and 12. Compared with Squid$^{\text{-adapt}}$, Squid reduces the number of chaining operations



(a) Alias analysis (b) Value-flow analysis (c) Points-to analysis

Fig. 11. Operation reduction rates by adaptive chaining (Squid vs. Squid^{-adapt}) and differential chaining (Squid^{-adapt} vs. Squid$^{-adapt-diff}$).



(a) Alias analysis (b) Value-flow analysis (c) Points-to analysis

Fig. 12. Speedups from adaptive chaining (Squid vs. Squid^{-adapt}) and differential chaining (Squid^{-adapt} vs. Squid$^{-adapt-diff}$).

by 96.61%, 60.47%, and 84.11% on average for alias, value-flow, and points-to analyses, respectively, yielding average speedups of 10.53×, 1.46×, and 2.12×. Likewise, compared with Squid$^{-adapt-diff}$, Squid^{-adapt} reduces chaining operations by 1.48%, 7.85%, and 16.27% on average and achieves corresponding speedups of 0.93×, 1.11×, and 1.15×. For a few benchmarks such as `nab` and `parest` in alias analysis, differential chaining affects only a negligible fraction of chaining operations and may, in some cases, introduce slightly more worklist iterations, resulting in minor performance degradation (Figure 12a). Interestingly, for `luindex` in points-to analysis, differential chaining unexpectedly introduces about 0.5% additional chaining operations. This may be due to changes in the worklist processing order, slightly affecting performance observed in Figure 12c.

Overall, both adaptive and differential chaining generally boost efficiency (sometimes dramatically) by reducing chaining operations, with adaptive chaining providing the larger benefit. Their impact on worklist iterations is minimal, averaging under 2%. We did not measure worklist iteration reductions relative to Pearl and Skewed because the derivation per iteration differs significantly across solvers.

4.6 RQ4: Application on Memory Leak Detection

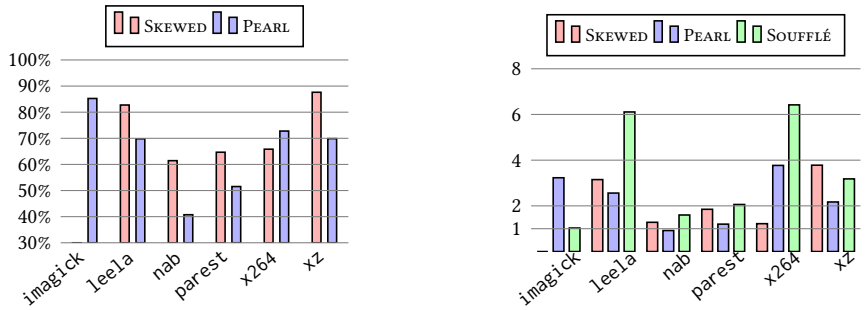
In this experiment, we demonstrate the practical utility of Squid through a taint-style memory leak analysis. The analysis adopts the taint analysis grammar from [Kodumal and Aiken 2004],

$$
\begin{aligned}
S &::= P \mid S\ A \mid S\ (_i \\
P &::= A\ P\ |)_i\ P \mid \varepsilon \\
A &::= A\ A \mid CA_i\ )_i \mid a \mid \varepsilon \\
CA_i &::= (_i\ A
\end{aligned}
$$

Fig. 13. Normalized CFG for Taint analysis

Table 6. Benchmark statistics for taint analysis in terms of nodes and edges after graph simplification, as well as sources and sinks.

	cactus	imagick	leela	nab	omnetpp	parest	perlbench	povray	x264	xz
#Nodes	104595	52442	9306	369	104896	9601	471504	158683	20033	2784
#Edges	218727	92479	16845	461	591858	12815	1155091	338845	35212	4505
#Souces	546	790	69	145	44	503	1015	844	160	42
#Sinks	509	9	428	63	48	1809	46	478	336	92



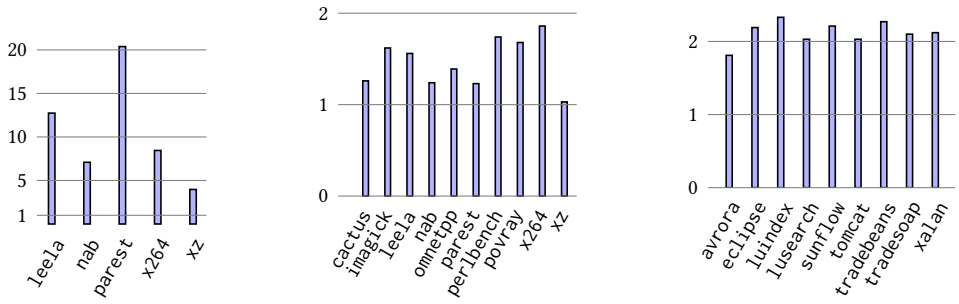
(a) Reduction of operations (b) Speedups

Fig. 14. Operation reduction and speedups of Squid over baseline solvers in taint analysis.

as illustrated in Figure 13. The corresponding CFG extends Figure 7 to capture data-flow paths with unbalanced calls and returns, a widely used abstraction in precise interprocedural data-flow analysis [Arzt and Bodden 2014; Reps et al. 1995; Shi et al. 2022]. For each program, we use the SVFG shown in Table 1a as the input graph. Memory allocation and deallocation sites are modeled as sources and sinks, respectively; a memory leak is reported when a source has no reachable sink.

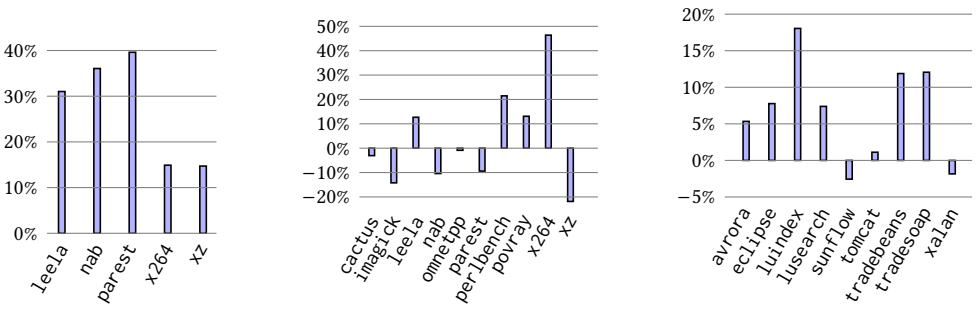
This taint-style analysis is known to be challenging to solve [Lei et al. 2024; Shi et al. 2022]. To enhance solving efficiency, we preprocess the graph by intersecting the forward slices (from sources) and the backward slices (from sinks). Both slicing procedures are performed in a context-insensitive manner [Weiser 1981]. This preprocessing safely eliminates edges not involved in any source-to-sink paths without affecting the reachability results, following the approach in [Shi et al. 2025]. The detailed graph statistics are presented in Table 6.

On average, Squid reduces 72.46% and 65.00% of chaining operations compared to Skewed and Pearl, achieving speedups of 2.26× and 2.31×, respectively, as shown in Figure 14. Revisiting the taint analysis grammar (Figure 13), the transitive relation A (arising from the production $A ::= A\ A$) can introduce transitive redundancy [Lei et al. 2022]. Notably, Skewed incorporates the technique from [Lei et al. 2022] to eliminate such redundancy. Without this technique, Squid could achieve even greater speedups by reducing more operations. Compared to Soufflé, Squid achieves a speedup of 3.40×. These results further underscore Squid's robustness in improving CFL-reachability performance.



(a) Alias analysis (b) Value-flow analysis (c) Points-to analysis

Fig. 15. The speedups of Skewed$^+$ against Skewed.



(a) Alias analysis (b) Value-flow analysis (c) Points-to analysis

Fig. 16. The memory reduction rates by Skewed$^+$ over Skewed.

4.7 RQ5: Combination with Skewed

In this experiment, we evaluate whether Squid can further improve the performance of Skewed. Since static skewing has been applied to rewrite CFGs for all solvers, we design a new solver, Skewed$^+$, which integrates dynamic skewing into Squid by retaining propagation edges in ΔInE and ΔOutE and discarding them after processing, without flushing them into InE and OutE.

In Figures 15 and 16, Skewed$^+$ generally outperforms Skewed, achieving speedups of 8.25×, 2.85×, and 2.26×, together with memory reductions of 27.26%, 3.38%, and 6.58%. These results suggest that Squid is largely orthogonal to Skewed. However, for some benchmarks in value-flow and points-to analyses, Skewed$^+$ incurs higher memory consumption than Skewed. Compared with Skewed, Skewed$^+$ may introduce additional memory overhead due to its differential graph representation. At the same time, this representation can reduce the memory required to store worklist items, since edges are organized more compactly—grouped by endpoint and edge label rather than stored individually in the conventional single-edge manner employed by Skewed. As a result, the overall memory usage of Skewed$^+$ may be either higher or lower than that of Skewed, depending on the characteristics of the benchmark. For example, in value-flow analysis, Skewed maintains 22.97× and 2.35× more worklist items than Skewed$^+$ on x264 and xz, respectively, which helps explain the observed variations in memory reduction rates shown in Figure 16b.

5 Related Work

CFL-reachability was initially developed in the database community [Hellings 2014; Yannakakis 1990] and has since become a foundational framework for expressing a wide range of program analysis problems [Reps 1998]. The classical CFL-reachability algorithm is computationally expensive, with a cubic time complexity of $O(|V|^3)$ [Melski and Reps 2000]. Chaudhuri [2008] proposed a subcubic CFL-reachability algorithm based on bit-vector operations (adopted in our evaluation), improving the cubic bound by a logarithmic factor to $O(|V|^3/\log|V|)$. However, achieving a truly subcubic bound of $O(|V|^{3-\varepsilon})$, where $\varepsilon > 0$, for general CFL-reachability remains an open problem [Chatterjee et al. 2017; Melski and Reps 2000]. Efficient algorithms are also proposed for special cases, such as bidirected Dyck-CFL-reachability [Xu et al. 2009; Zhang et al. 2013], graph reachability with bounded treewidth [Chatterjee et al. 2017], or particular forms of context-free grammars [Shi et al. 2024a, 2022; Zhang et al. 2014].

Practical techniques for improving the efficiency of online CFL-reachability solvers have been extensively explored. Pocr [Lei et al. 2022] introduces a partially ordered CFL-reachability algorithm to mitigate transitive redundancy caused by transitive relations. Iea [Xu et al. 2024] employs an iterative epoch method to eliminate cycles formed by transitive relations. Skewed [Lei et al. 2024] reduces redundant summary edges during solving through a *skewed tabulation* scheme. Graspan [Wang et al. 2017] reformulates static analysis as disk-assisted Big Data computation to enhance scalability on a single machine. CFL-reachability was shown to be interconvertible with a class of set-constraint problems [Melski and Reps 2000], followed by a specialized reduction technique tailored for extended Dyck-CFL-reachability. More recently, Pearl [Shi et al. 2023, 2024b] employs a set-constraint-based multi-derivation approach to propagate reachability information in batches. Different from this line of work, Squid solves CFL-reachability via efficient relation chaining, accelerating the solving process from a fundamentally different perspective.

Offline graph simplification techniques enhance CFL-reachability performance by reducing the input graph size before solving. In pointer analysis, a significant CFL-reachability client, numerous such techniques have been proposed, many exploiting pointer equivalence [Hardekopf and Lin 2007b], including cycle elimination [Hardekopf and Lin 2007a; Pereira and Berlin 2009]—arguably the most extensively studied—and offline variable substitution [Rountev and Chandra 2000]. Guided by recursive state machines, an equivalent formulation of context-free languages, Gf examines state-transition rules to contract trivial edges [Alur et al. 2005; Lei et al. 2023]. This approach generalizes offline variable substitution from pointer analysis to more general CFL-reachability settings. For client-driven CFL-reachability, where only reachability between specific sources and sinks is of interest, Moye [Shi et al. 2025] approximates the target context-free language with a regular one via the well-known MN transformation [Mohri and Nederhof 2001] and leverages the resulting deterministic finite automaton to eliminate useless edges. These simplification techniques are orthogonal to Squid and can serve as preprocessing steps to further improve solving efficiency.

Interleaved Dyck-CFL-reachability [Li et al. 2020, 2022, 2023; Reps 2000; Zhang and Su 2017], a non-context-free problem, can be approximated by intersecting two independent CFL-reachability instances—either by combining their results after saturation [Späth et al. 2019] or by iteratively intersecting underlying graphs to prune non-contributing edges [Ding and Zhang 2023]. Thanks to its superior efficiency, Squid stands out as a promising alternative to existing solvers for such tasks.

Pointer analysis [Reps 1998; Sridharan et al. 2005], an adjacent field to CFL-reachability, has developed various optimization techniques to reduce the number of bit-vector operations (i.e., points-to set propagations) [Hardekopf and Lin 2007a; Pearce et al. 2003; Pereira and Berlin 2009]. A representative technique closely related to Squid is worklist prioritization [Sridharan and Fink 2009], which reduces propagation count by exploiting a dynamically maintained topological order. In

pointer analysis, points-to sets are propagated along a unidirectional, unlabeled pointer-assignment graph, where the propagation (join) direction is fixed. In contrast, adaptive chaining is designed for CFL-reachability on bidirectional, edge-labeled graphs: it operates on two edge sets and dynamically selects a more efficient chaining direction based on local information (i.e., input set sizes) rather than relying on a global graph topology. Extending worklist prioritization to CFL-reachability would require reasoning jointly about node positions and edge labels, making this a nontrivial but promising direction for future work.

Datalog is a well-known framework for expressing program analysis problems [Bravenboer and Smaragdakis 2009], including context-sensitive pointer analysis for Java and security analysis for smart contracts [Antoniadis et al. 2025; Bravenboer and Smaragdakis 2009; Smaragdakis et al. 2021]. Problem instances in Datalog can exploit advanced evaluation strategies—such as semi-naive evaluation and magic set transformations [Abiteboul et al. 1995; Bancilhon et al. 1985; Hollingum and Scholz 2015]—to mitigate redundant computations. In contrast, our work enhances the efficiency of CFL-reachability solving itself, without depending on such specialized evaluation techniques. It is worth noting that differential chaining is conceptually related to seminaive evaluation, but it is specialized for chaining labeled edges and works jointly with our enhanced graph representation.

6 Conclusion

This paper reformulates CFL-reachability analysis as an iterative process of applying relation chaining over labeled edges. This perspective reveals substantial redundancy arising from the inefficient chaining strategies used by representative state-of-the-art methods, including the optimized standard solver Skewed and Pearl, a multi-derivation approach based on set constraints.

To address this inefficiency, we propose Squid, a novel CFL-reachability algorithm featuring two carefully designed chaining techniques, enabled by an enhanced graph representation. We implemented Squid and evaluated it against two leading solvers, Skewed and Pearl, on three widely studied analysis tasks: field-sensitive alias analysis and context-sensitive value-flow analysis for C/C++, and field-sensitive points-to analysis for Java. Empirical results demonstrate that Squid significantly improves performance by eliminating a large fraction of redundant chaining operations. Across the three client analyses, Squid reduces 97.76%, 84.76%, and 72.05% of chaining operations compared with Skewed, achieving average speedups of 9.30×, 3.08×, and 1.86×, respectively. Relative to Pearl, Squid eliminates 96.78%, 60.30%, and 93.97% of chaining operations, yielding average speedups of 12.55×, 1.70×, and 8.31×. When compared with Soufflé, a widely used Datalog solver, Squid achieves average speedups of 1.87×, 4.93×, and 3.52× across the three analyses.

We envision that this chaining perspective can open new opportunities for further boosting CFL-reachability. Future work may explore novel relation representations tailored for more efficient chaining, beyond the conventional or enhanced bidirectional adjacency-list modeling. We also see parallelism as a promising direction for further performance gains. Similar to Pearl and Skewed, Squid currently relies on sequential worklist processing. However, our fine-grained representation may be more amenable to parallelization, as the head and tail views of an edge typically participate in different productions and thus introduce fewer tight dependencies. In addition, combining our approach with techniques beyond Skewed remains an interesting direction for future work.

Acknowledgments

We thank all reviewers for their valuable feedback. This work is supported by the National Natural Science Foundation of China (62402474, 62132020, and 62202452), and the China Postdoctoral Science Foundation (2024M753295).

Data Availability

Our artifact is available at [Shi et al. 2026].

References

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

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

Anastasios Antoniadis, Ilias Tsatiris, Neville Grech, and Yannis Smaragdakis. 2025. Universal Scalability in Declarative Program Analysis (with Choice-Based Combination Pruning). *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 351 (Oct. 2025), 28 pages. doi:10.1145/3763129

Steven Arzt and Eric Bodden. 2014. 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*. 288–298.

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. doi:10.1145/2666356.2594299

Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Cambridge, Massachusetts, USA) *(PODS '86)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/6012.15399

Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Specification Inference Using Context-Free Language Reachability. *SIGPLAN Not.* 50, 1 (Jan. 2015), 553–566. doi:10.1145/2775051.2676977

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. *SIGPLAN Not.* 41, 10 (oct 2006), 169–190. doi:10.1145/1167515.1167488

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, USA, 241–250. doi:10.1145/1985793.1985827

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. Association for Computing Machinery, New York, NY, USA, 243–262. doi:10.1145/1640089.1640108

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. doi:10.1145/3158118

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. doi:10.1145/1328438.1328460

Shuo Ding and Qirun Zhang. 2023. Mutual Refinements of Context-Free Language Reachability. In *Static Analysis*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer Nature Switzerland, Cham, 231–258. doi:10.1007/978-3-031-44245-2_12

Ben Hardekopf and Calvin Lin. 2007a. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 290–299. doi:10.1145/1273442.1250767

Ben Hardekopf and Calvin Lin. 2007b. Exploiting pointer and location equivalence to optimize pointer analysis. In *Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings 14*. Springer, 265–280.

Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. 2019. 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, 267–279.

Dongjie He, Jingbo Lu, and Jingling Xue. 2022. Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:29. doi:10.4230/LIPIcs.ECOOP.2022.30

Dongjie He, Jingbo Lu, and Jingling Xue. 2024. A CFL-Reachability Formulation of Callsite-Sensitive Pointer Analysis with Built-In On-The-Fly Call Graph Construction. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2024.18

Jelle Hellings. 2014. Conjunctive Context-Free Path Queries.. In *ICDT*. 119–130.

Nicholas Hollingum and Bernhard Scholz. 2015. Towards a scalable framework for context-free language reachability. In *Compiler Construction: 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24*. Springer, 193–211.

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. doi:10.1007/978-3-319-41540-6_23

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

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. doi:10.1109/CGO.2004.1281665

Yuxiang Lei, Camille Bossut, Yulei Sui, and Qirun Zhang. 2024. Context-Free Language Reachability via Skewed Tabulation. *Proc. ACM Program. Lang.* 8, PLDI, Article 221 (jun 2024), 24 pages. doi:10.1145/3656451

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. doi:10.1145/3563343

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. doi:10.1145/3591233

Haofeng Li, Chenghang Shi, Jie Lu, Lian Li, and Jingling Xue. 2024. Boosting the Performance of Alias-Aware IFDS Analysis with CFL-Based Environment Transformers. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 364 (Oct. 2024), 29 pages. doi:10.1145/3689804

Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. 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*. Association for Computing Machinery, New York, NY, USA, 343–353. doi:10.1145/2025113.2025160

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

Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program tailoring: Slicing by sequential criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2016.15

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. doi:10.1145/3385412.3386021

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. doi:10.1145/3492428

Yuanbo Li, Qirun Zhang, and Thomas Reps. 2023. Single-Source-Single-Target Interleaved-Dyck Reachability via Integer Linear Programming. *Proc. ACM Program. Lang.* 7, POPL, Article 35 (Jan. 2023), 24 pages. doi:10.1145/3571228

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. doi:10.1016/S0304-3975(00)00049-9

Mehryar Mohri and Mark-Jan Nederhof. 2001. *Regular Approximation of Context-Free Grammars through Transformation*. Springer Netherlands, Dordrecht, 153–163. doi:10.1007/978-94-015-9719-7_6

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

Fernando Magno Quintao Pereira and Daniel Berlin. 2009. Wave propagation and deep propagation for pointer analysis. In *2009 International Symposium on Code Generation and Optimization*. IEEE, 126–135.

Jakob Rehof and Manuel Fähndrich. 2001. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (London, United Kingdom) *(POPL '01)*. Association for Computing Machinery, New York, NY, USA, 54–66. doi:10.1145/360204.360208

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

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

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

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

Chenghang Shi, Dongjie He, Haofeng Li, Jie Lu, Lian Li, and Jingling Xue. 2025. Fast Client-Driven CFL-Reachability via Regularization-Based Graph Simplification. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 287 (Oct. 2025), 28 pages.

doi:10.1145/3763065

Chenghang Shi, Haofeng Li, Jie Lu, and Lian Li. 2024a. Better Not Together: Staged Solving for Context-Free Language Reachability. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, New York, NY, USA, 1112–1123. doi:10.1145/3650212.3680346

Chenghang Shi, Haofeng Li, Jie Lu, and Lian Li. 2026. Artifact of "Context-Free Language Reachability via Efficient Relation Chaining". (2026). doi:10.6084/m9.figshare.30330334.v3

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. doi:10.1109/ASE56229.2023.00118

Chenghang Shi, Haofeng Li, Yulei Sui, Jie Lu, Lian Li, and Jingling Xue. 2024b. PEARL: A Multi-Derivation Approach to Efficient CFL-Reachability Solving. *IEEE Transactions on Software Engineering* (2024). doi:10.1109/TSE.2024.3437684

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. doi:10.1145/3563339

Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. 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*. Association for Computing Machinery, New York, NY, USA, 693–706. doi:10.1145/3192366.3192418

Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. 2021. Symbolic value-flow static analysis: deep, precise, complete modeling of Ethereum smart contracts. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 163 (Oct. 2021), 30 pages. doi:10.1145/3485540

Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

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

Manu Sridharan and Stephen J Fink. 2009. 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, 205–221.

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. doi:10.1145/1250734.1250748

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. doi:10.1145/1094811.1094817

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. doi:10.1145/2892208.2892235

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. doi:10.1145/2338965.2336784

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. doi:10.1109/TSE.2014.2302311

Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1093–1105. doi:10.1145/3597926.3598120

Rei Thiessen and Ondřej Lhoták. 2017. Context transformations for pointer analysis. *ACM SIGPLAN Notices* 52, 6 (2017), 263–277. doi:10.1145/3062341.3062359

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. doi:10.1145/3037697.3037744

Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, USA) *(ICSE '81)*. IEEE Press, 439–449.

James O. Westgard. n.d.. Lesson 34: What is an acceptable CV? https://westgard.com/lessons/z-stats-basic-statistics/lesson34.html Accessed: 2025-01-12.

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. doi:10.1007/978-3-642-03013-0_6

Pei Xu, Yuxiang Lei, Yulei Sui, and Jingling Xue. 2024. Iterative-Epoch Online Cycle Elimination for Context-Free Language Reachability. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 145 (April 2024), 26 pages. doi:10.1145/3649862

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. doi:10.1145/298514.298576

Peisen Yao, Jinguo Zhou, Xiao Xiao, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2024. Falcon: A Fused Approach to Path-Sensitive Sparse Data Dependence Analysis. *Proc. ACM Program. Lang.* 8, PLDI, Article 170 (June 2024), 26 pages. doi:10.1145/3656400

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. doi:10.1145/2491956.2462159

Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 344–358. doi:10.1145/3009837.3009848

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. doi:10.1145/2660193.2660213

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. doi:10.1145/1328438.1328464

Received 2025-10-10; accepted 2026-02-17