Mutual Refinements of Context-Free Language Reachability

Shuo Ding[0000−0003−0843−0729] and Qirun Zhang[0000−0001−5367−9377]

Georgia Institute of Technology, Atlanta GA 30332, USA
{sding,qrzhang}@gatech.edu

Abstract. Context-free language reachability is an important program analysis framework, but the exact analysis problems can be intractable or undecidable, where CFL-reachability approximates such problems. For the same problem, there could be many over-approximations based on different CFLs $C_1, \ldots, C_n$. Suppose the reachability result of each C_i produces a set P_i of reachable vertex pairs. Is it possible to achieve better precision than the straightforward intersection $\bigcap_{i=1}^{n} P_i$?
This paper gives an affirmative answer: although CFLs are not closed under intersections, in CFL-reachability we can "intersect" graphs. Specifically, we propose *mutual refinement* to combine different CFL-reachability-based over-approximations. Our key insight is that the standard CFL-reachability algorithm can be slightly modified to trace the edges that contribute to the reachability results of C_1, and C_2-reachability only need to consider contributing edges of C_1, which can, in turn, trace the edges that contribute to C_2-reachability, etc. We prove that there exists a unique optimal refinement result (fix-point). Experimental results show that mutual refinement can achieve better precision than the straightforward intersection with reasonable extra cost.

1 Introduction

Context-free language reachability (CFL-reachability) is arguably the best-known graph reachability framework in program analysis [15,19,22,23,26,30]. Typically, the framework consists of a frontend and a backend, where the frontend constructs a graph from the source code, and the backend runs CFL-reachability on the graph to obtain properties of the source code [25]. The graphs and grammars depend on specific analyses, but CFL-reachability has a dynamic programming style (sub)cubic-time algorithm [2,20,25,35] for arbitrary graphs and grammars. Faster algorithms exist on special cases [1,17,36].

However, due to the inherent hardness of program analysis, CFL-reachability may not be able to model the exact formulation of the problem [24, 37]. A typical example is the interleaved-Dyck-reachability formulation [18, 24], which is widely used to simultaneously model function calls/returns [24, 26], field reads/writes [29, 34], locks/unlocks [11], etc. The interleaved-Dyck language is not context-free [9], and the corresponding graph reachability problem is undecidable [24]. In practice, CFL-reachability can over-approximate computationally hard language reachability problems [24]: the idea is to design a context-free language C that over-approximates the non-context-free language L (meaning that C contains more strings than L).

For a computationally hard L-reachability problem, different CFL-reachability-based approaches over-approximate the solution from different angles. We can straightforwardly intersect the results to achieve better precision. Synchronized pushdown systems [27] essentially employ this idea. The linear conjunctive language reachability work [37] also shows this straightforward intersection can improve precision. However, in this case, different CFL-reachability instances are executed independently. On the other hand, CFLs are not closed under intersection [8,9], so in general, we cannot intersect different CFLs to obtain a new CFL for CFL-reachability. For example, the interleaved-Dyck language [18, 37], which is not a CFL, is the intersection of two or more CFLs.

This paper proposes a more synergistic way to "intersect" multiple CFL-reachability-based over-approximations. Specifically, typical CFL-reachability algorithms are of dynamic programming style, which generate "summary edges" from existing graph edges. Our key insight is that when those algorithms generate a summary edge, the existing edges directly contributing to the generation can be recorded. This augmented algorithm can eventually trace a set of original graph edges that contribute to the final reachability results. Therefore, when combining CFL-reachability-based over-approximations based on CFLs $C_1, C_2, \ldots, C_n$, we can run C_2-reachability on the contributing edges of C_1-reachability, as opposed to all edges in the original graph. This process can happen between different C_i's multiple times, which is called *mutual refinement*.

Is the execution order of different CFL-reachability over-approximations important for mutual refinement? We prove in Section 4 that given a set of CFL-reachability over-approximations, there exists a unique fix-point, and any order of executing different CFL-reachabilities will reach the fix-point. This is similar to the fix-point theorem [14] and chaotic-iteration algorithms [4, 12] in dataflow analysis. The soundness of the fix-point and the fact that it is at least as precise as the straightforward intersection are also proved in Section 4. As for the complexity, suppose the CFL is fixed and the number of vertices in the graph is n, then the time complexities of the standard CFL-reachability algorithm [20,25,35] and our augmented algorithm are both $\tilde{O}(n^3)$, and the space complexity is $\tilde{O}(n^2)$ for the standard algorithm and is $\tilde{O}(n^3)$ for our augmented algorithm.

We conduct experiments on two applications: a taint analysis for Java programs obtained from Android apps [10], and a value-flow analysis for LLVM IR programs obtained from the SPEC CPU 2017 benchmark [28]. On average,

compared with the straightforward intersection, mutual refinement achieves a 50.95% precision improvement (measured by the number of reachable pairs) with a 2.65× time increase and a 3.23× space increase on the taint analysis benchmarks, and achieves a 9.37% precision improvement with a 2.55× time increase and a 2.22× space increase on the value-flow analysis benchmarks.

The fast graph simplification algorithm [18] proposed by Li, Zhang, and Reps (abbreviated as the LZR algorithm) also simplify graphs, but the LZR algorithm only works for interleaved-Dyck-reachability while mutual refinement works for any L-reachability preserving CFL-reachability-based over-approximations, and the LZR algorithm is a pre-processing algorithm while mutual refinement is a complete solver. Our taint analysis experiment is interleaved-Dyck reachability, and thus we also evaluate mutual refinement on those graphs simplified by the LZR algorithm: LZR preprocessing can, on average, bring a further precision improvement of 3.27% and reduces the time/space consumption in certain cases. The value-flow analysis experiment is not interleaved-Dyck reachability, so the LZR algorithm is not applicable.

In summary, this paper makes the following main contributions.

- We propose mutual refinement for combining different CFL-reachability over-approximations for hard formal language reachability problems, which can achieve better precision than the straightforward intersection.
- We prove the existence and uniqueness of the fix-point, the soundness, the precision guarantee (being at least as precise as the straightforward intersection), and time/space complexities for mutual refinement.
- We evaluate mutual refinement on two program analysis applications. Experimental results show that mutual refinement can achieve better precision than the straightforward intersection with reasonable extra cost.

This paper is organized as follows. Section 2 gives a motivating example. Section 3 reviews backgrounds and definitions. Section 4 presents mutual refinement and its properties. Section 5 gives experimental results. Section 6 presents discussions. Section 7 surveys related work, and Section 8 concludes.

2 Motivating Example

This section motivates mutual refinement using an example of context-sensitive and field-sensitive taint analysis for C++. The analysis first generates a graph from the source code being analyzed, then the source-sink relation from the source code is reduced to the reachability of two vertices in the graph. This is an extended version of the taint analysis mentioned in the work of Huang *et al* [10]. The original analysis is based on interleaved-Dyck reachability, but our motivating example is not. We compare mutual refinement with the straightforward intersection of two different CFL-reachability-based over-approximations.

Example Code. Figure 1 shows a C++ code snippet. The analysis goal is to decide whether the value of the variable s could flow into the variable t. Because

```
#include ...

class Pair {
    int first, second;
    Pair(int fi, int se) : first(fi), second(se) {}
}

int getFirst(Pair p1) {
    return p1.first; // represented by ret1
}

int getSecond(Pair p2) {
    return p2.second; // represented by ret2
}

int main() {
    int s = getSecret();
    Pair a(0, s);
    Pair b(0, 0);
    Pair t(0, 0);
    if (getInput() == "first") {
        int x = getFirst(a);
        t.first = x;
    } else {
        send(getSecond(a));
        int y = getSecond(b);
        t.second = y;
    }
    ...
}
```

Fig. 1: A taint analysis example for C++. The goal is to decide whether the value s can flow into t. The fact is that the value of s cannot flow into t.

t can only contain the first field of a or the second field of b, the answer is that the value of s cannot flow into t.

Graph Reachability Formulation. We use vertices to represent variables and use edges to represent values flowing among variables (Figure 2). To achieve context-sensitivity and field-sensitivity, we use parenthesis-labeled edges for function calls/returns, and use bracket-labeled edges for field writes/reads.

The taint analysis decides whether the value of s can flow into t (including t's fields). The answer is "yes" if and only if there is a path whose edge labels can be concatenated to a string that represents the interleaving of matched parentheses, matched brackets, and unmatched open brackets. Formally, given an alphabet Σ, we define the interleaving operator [18] $\odot : \Sigma^* \times \Sigma^* \to \mathcal{P}(\Sigma^*)$ as follows, where s, s_1, s_2 are strings and c_1, c_2 are single characters.

$$\begin{aligned}
\epsilon \odot s &= \{s\} \\
s \odot \epsilon &= \{s\} \\
c_1 s_1 \odot c_2 s_2 &= \{c_1 w \mid w \in (s_1 \odot c_2 s_2)\} \cup \{c_2 w \mid w \in (c_1 s_1 \odot s_2)\}.
\end{aligned}$$

For the taint analysis, we are interested in L_T-reachability problem, where $L_T = \bigcup\{s_1 \odot s_2 \mid s_1 \in P, s_2 \in B\}$ and CFLs P and B are defined as follows. Note that we use P or B to denote both the languages and the starting symbols in

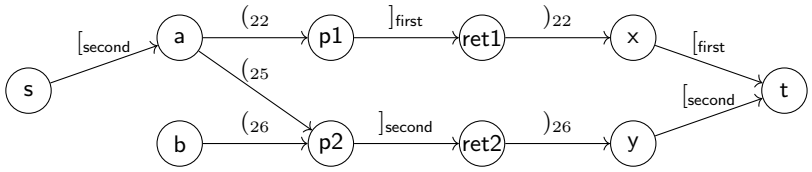


Fig. 2: The taint analysis graph for Figure 1. Vertices are variables and edges model values flowing among variables: $i \xrightarrow{(_c} j$ represents that i flows into j via the function call at line c; $i \xrightarrow{)_c} j$ represents that i flows into j via the function return at line c; $i \xrightarrow{[_f} j$ represents that i flows into the f field of j; $i \xrightarrow{]_f} j$ represents that the f field of i flows into j.

the grammars. L_T is not context-free (which is proved in Section 6). We also extend the definition of the interleaving operator $\odot$ to languages, and thus we write $L_T = P \odot B$.

$$P \to P\ P \mid (_1\ P\ )_1 \mid \ldots \mid (_m\ P\ )_m \mid \epsilon$$

$$B \to D\ B \mid [_1\ B \mid \ldots \mid [_n\ B \mid \epsilon.$$
$$D \to D\ D \mid [_1\ D\ ]_1 \mid \ldots \mid [_n\ D\ ]_n \mid \epsilon.$$

In Figure 2, there are only two possible paths from s to t. None of these paths satisfy our requirement, so the value of s cannot flow into t.

CFL-Reachability-Based Over-Approximations. We devise two CFLs C_P and C_B to over-approximate L_T. C_P considers only parentheses and treats brackets as empty symbols. C_B considers only brackets and treats parentheses as empty symbols. Both algorithms can be implemented using the well-known CFL-reachability algorithm [20, 25, 35]. In Figure 2, both C_P-reachability and C_B-reachability conclude that t is reachable from s. For example, for C_P, t is reachable from s via the path $\mathsf{s} \xrightarrow{[_{\mathsf{second}}} \mathsf{a} \xrightarrow{(_{22}} \mathsf{p1} \xrightarrow{]_{\mathsf{first}}} \mathsf{ret1} \xrightarrow{)_{22}} \mathsf{x} \xrightarrow{[_{\mathsf{first}}} \mathsf{t}$, and for C_B, t is reachable from s via the path $\mathsf{s} \xrightarrow{[_{\mathsf{second}}} \mathsf{a} \xrightarrow{(_{25}} \mathsf{p2} \xrightarrow{]_{\mathsf{second}}} \mathsf{ret2} \xrightarrow{)_{26}} \mathsf{y} \xrightarrow{[_{\mathsf{second}}} \mathsf{t}$. These two conclusions are both false positives.

Straightforward Intersection. One possible method to combine the above two over-approximations is to directly intersect their results, as synchronized pushdown system [27] and linear conjunctive language reachability [37] did. In Figure 2, however, this method still concludes that t is reachable from s, because the two algorithms both conclude that they are reachable.

Mutual Refinement. To further improve the precision, we first run C_B-reachability, which concludes that t is reachable from s, and the edges contributing to all reachable pairs can be shown in Figure 3 (parentheses are treated as empty symbols so edges labeled with parentheses are preserved). Now the graph has been simplified. Running C_P-reachability on the graph in Figure 3 concludes

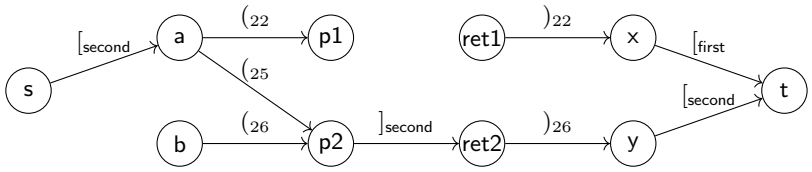


Fig. 3: After running C_B-reachability and tracing only the edges contributing to its results, the graph is simplified. Subsequent execution of C_P-reachability can then conclude that t is not reachable from s.

that t is not reachable from s. Thus C_B-reachability "refines" the subsequent execution of C_P-reachability. This example shows that mutual refinement can achieve better precision compared with the straightforward intersection.

3 Preliminary

We define L-reachability in Section 3.1 and review the standard dynamic programming style algorithm for CFL-reachability in Section 3.2. CFL-reachability is used to over-approximate other L-reachability problems in practice.

3.1 L-Reachability

The L-reachability problem is to find pairs (s, t) of vertices such that there exists a path from s to t, and the edge labels along that path form a string in L.

Definition 1 (L-Reachability). *Given a formal language L with a finite alphabet Σ and a finite graph $G = (V, E)$, where each edge $e \in E$ is labeled with a character in Σ, vertex $t \in V$ is L-reachable from vertex $s \in V$ if and only if there exists a finite path (with possibly duplicate vertices and edges) $p = s \xrightarrow{l_1} v_1 \xrightarrow{l_2} \dots \xrightarrow{l_{n-1}} v_{n-1} \xrightarrow{l_n} t$ in the graph such that the string $l_1 l_2 \dots l_n$ is in the given formal language. $R(p) = l_1 l_2 \dots l_n$ is the path string of p. The zero-length path from a vertex to itself forms the empty string. (s, t) is an L-reachable pair. The L-reachability problem is to find the set of all L-reachable pairs.*

Fix a formal language L. Any graph $G = (V, E)$ whose edges are labeled with characters in the alphabet of L essentially gives an instance of the L-reachability problem. We denote this instance as $\langle L, (V, E)\rangle$. There also exist other variants of L-reachability, such as the single-pair reachability, which only cares about the reachability between a specific pair of vertices. In this paper, we focus on all-pairs reachability unless otherwise noted.

3.2 CFL-Reachability

In L-reachability, when the formal language L is a context-free language, the problem is a CFL-reachability problem. CFL-reachability exhibits a popular dynamic programming style cubic-time algorithm [20, 25, 35], which is shown in

Algorithm 1 The CFL-Reachability Algorithm

```
1: function CFL-Reachability(⟨C, (V, E)⟩)
2:     W ← emptyWorkList()
3:     W.addAll(E)
4:     for X → ϵ ∈ C do
5:         for v ∈ V do
6:             if X⟨v, v⟩ ∉ E then
7:                 add X⟨v, v⟩ to E and W
8:     while W.nonEmpty() do
9:         Y⟨i, j⟩ ← W.pop()
10:        for X → Y ∈ C do
11:            if X⟨i, j⟩ ∉ E then
12:                add X⟨i, j⟩ to E and W
13:        for X → YZ ∈ C do
14:            for Z⟨j, k⟩ ∈ E do
15:                if X⟨i, k⟩ ∉ E then
16:                    add X⟨i, k⟩ to E and W
17:        for X → ZY ∈ C do
18:            for Z⟨k, i⟩ ∈ E do
19:                if X⟨k, j⟩ ∉ E then
20:                    add X⟨k, j⟩ to E and W
21:    return (V, E)
```

Algorithm 1. Given an instance $\langle C, (V, E)\rangle$ of (all-pairs) CFL-reachability problem, we call CFL-Reachability($\langle C, (V, E)\rangle$) in Algorithm 1 to compute the results. The CFL $C = (\Sigma, N, P, S)$ contains the set of terminal symbols Σ, the set of non-terminal symbols N, the set of productions P, and the start symbol S. All productions are in the forms $X \to YZ$, $X \to Y$, and $X \to \epsilon$, where X, Y, and Z are terminal symbols or non-terminal symbols. Any context-free grammar can be transformed into this form [9]. Algorithm 1 works as follows.

1. Initially, all edges in the original graph are added to the worklist (line 3).
2. Then the productions with empty right-hand sides are applied where new edges are added to both the graph and the worklist (lines 4-7).
3. After that, the algorithm removes an edge $Y\langle i, j\rangle$ (connecting vertices i, j with label Y) from the worklist, trying all productions with Y on the right-hand side, adding newly generated edges to both the graph and the worklist, and repeating this process until the worklist becomes empty (lines 8-20).
4. Finally, the updated graph is returned as the result (line 21).

Algorithm 1 shows the process of generating new edges in the graph according to productions, which we call *production applications*. For example, if there is a production $X \to YZ$ and there are edges $Y\langle i, j\rangle$ and $Z\langle j, k\rangle$ already in the graph, we add a new edge $X\langle i, k\rangle$ to the graph via a production application. Note that when we pop an edge e out from the worklist and process it (lines 8-20), depending on the previously processed edges, some of e's adjacent edges

might not be available in the graph yet, thus the current iteration of lines 8-20 may not add all edges that can be generated from e via production applications. However, it is well-known that eventually, all possible edges will be added, and the order of popping out edges from the worklist does not matter.

Theorem 1 (Algorithm 1's Correctness). *When Algorithm 1 terminates, all edges that can be generated by production applications will be in the graph.*

Proof. We prove it by contradiction. Suppose there is an edge e_n, which could be generated by a finite number of production applications, and which is not in the graph produced by Algorithm 1. It is obvious that e_n cannot be in the original edge set. So e_n could be obtained by finitely and positively many production applications. We can denote this process by a sequence $e_1, e_2, \ldots, e_n$, where for all $i \in \{1, 2, \ldots, n\}$, e_i is either in the original graph's edge set or is obtained by applying a production on a set of edges $\{e_{i_1}, e_{i_2}, \ldots, e_{i_{n_i}}\} \subseteq \{e_1, e_2, \ldots, e_{i-1}\}$. Suppose e_j is the first edge in $e_1, e_2, \ldots, e_n$ that is not in the graph produced by Algorithm 1. There must exist such an e_j because according to our assumption, at least e_n is not in the graph produced by Algorithm 1. Again, e_j cannot be in the original edge set. Suppose e_j can be obtained by applying a production on a set of edges $\{e_{j_1}, e_{j_2}, \ldots, e_{j_{n_j}}\} \subseteq \{e_1, e_2, \ldots, e_{j-1}\}$. Since all edges in $\{e_{j_1}, e_{j_2}, \ldots, e_{j_{n_j}}\}$ are in the graph produced by Algorithm 1, when the last edge was popped out, since all productions were tried, Algorithm 1 must add e_j to the graph since all edges in $\{e_{j_1}, e_{j_2}, \ldots, e_{j_{n_j}}\}$ were available in the graph at that time. This is a contradiction.

We briefly explain our notations for time/space complexity. To make the analysis rigorous, we should also consider the time/space complexity of handling numbers. For example, if there are n vertices in the graph and we use $0, 1, \ldots, n-1$ to represent the vertices, then the number $n-1$ itself requires memory of $O(\log n)$, and arithmetic operations on those numbers are not of constant complexity. To focus on dominating factors, instead of using the big-O notation, we use the $\tilde{O}$ notation [3] to hide logarithm factors: $\tilde{O}(f(n))$ represents $O(f(n)(\log n)^k)$ for some constant natural number k.

Complexity Analysis for Algorithm 1. Suppose the grammar of the CFL is fixed, adding one edge to the graph takes constant time, accessing each graph vertex's adjacent vertices takes linear time, and pushing/popping elements to/from the worklist takes constant time. There could be at most $\tilde{O}(|V|^2)$ edges popped out from the worklist at line 9, and for each edge popped out, there could be at most $\tilde{O}(|V|)$ adjacent edges to try in the main **while** loop. Thus the time complexity is $\tilde{O}(|V|^3)$. The space complexity is $\tilde{O}(|V|^2)$ since there could be at most $\tilde{O}(|V|^2)$ edges in the graph and in the worklist.

4 Mutual Refinement

This section formalizes mutual refinement. Specifically, Section 4.1 gives an overview; Section 4.2 presents the important definition of *contributing edges*;

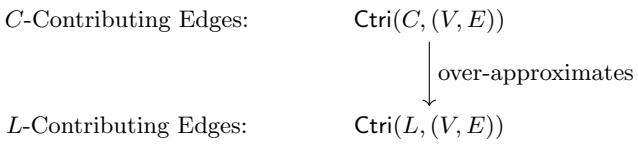


Fig. 4: Two important concepts in mutual refinement. L is a formal language whose reachability problem is computationally hard, and C is a context-free language over-approximating L. The set of C-contributing edges $\mathsf{Ctri}(C,(V,E))$ over-approximates the set of L-contributing edges $\mathsf{Ctri}(L,(V,E))$.

Section 4.3 presents the algorithm used as individual steps in mutual refinements; Section 4.4 presents the complete mutual refinement algorithm.

4.1 Overview

Suppose we have a set of CFL-reachability-based over-approximations using CFLs $C_1, C_2, \ldots, C_m$ for a computationally hard L-reachability problem. For an instance $\langle L,(V,E)\rangle$ of the problem, we first run C_1-reachability. Then we only keep edges that directly or indirectly participated in the construction of S_1 (the start symbol of C_1's grammar) edges. Then we run C_2-reachability and only keep edges participated in the construction of S_2 edges. This process continues until we reach the fix-point: no more edges can be removed. The final reachability result is obtained by executing C_1, C_2, …, C_m-reachability on the minimum graph and taking the intersection.

4.2 Contributing Edges

The key step of mutual refinement is to over-approximate the set of "useful" edges, *i.e.*, the edges contributing to reachable pairs. This is achieved via formal language over-approximations.

Definition 2 (Formal Language Over-Approximation). *Given two formal languages L_1 and L_2, L_1 over-approximates L_2 if and only if $L_1 \supseteq L_2$.*

Definition 3 (L-Contributing Edges). *For a specific formal language L, given an instance $\langle L,(V,E)\rangle$ of the L-reachability problem, an edge $e \in E$ is an L-contributing edge for this instance if and only if there exists a pair of vertices $u, v \in V$, such that e is part of a finite path p connecting u and v and $R(p) \in L$. The set of such L-contributing edges is denoted as $\mathsf{Ctri}(L,(V,E))$.*

In certain undecidable L-reachability problems (*e.g.*, the interleaved-Dyck-reachability), it can be shown that computing the set $\mathsf{Ctri}(L,(V,E))$ is also undecidable in general [18]. So we need to approximate this set. Suppose we have a context-free language C over-approximating L, then it is straightforward to see that $\mathsf{Ctri}(L,(V,E)) \subseteq \mathsf{Ctri}(C,(V,E))$, because every L-path is also a C-path. Figure 4 summarizes the situation.

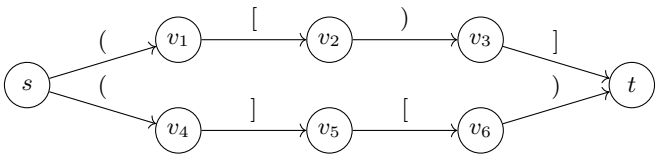


Fig. 5: A Graph illustrating contributing edges.

Example 1 (Contributing Edges). Consider the following example of interleaved-Dyck-reachability, where each string in the interleaved-Dyck language L is an interleaving of two strings from the following two CFLs, respectively.

$$P \to P\ P \mid (\ P\ ) \mid \epsilon$$
$$B \to B\ B \mid [\ B\ ] \mid \epsilon.$$

We use the following context-free language C, which only considers matched parentheses and treats brackets as empty symbols, to over-approximate the interleaved-Dyck language L.

$$C \to C\ C \mid (\ C\ ) \mid [\ ] \mid \epsilon.$$

In the instance of interleaved-Dyck-reachability shown in Figure 5, the set of L-contributing edges is $\{s \xrightarrow{(} v_1, v_1 \xrightarrow{[} v_2, v_2 \xrightarrow{)} v_3, v_3 \xrightarrow{]} t\}$, while the set of C-contributing edges includes all edges in the original graph.

4.3 Tracing Algorithm

The set of CFL-contributing edges is computable via augmenting the standard CFL-reachability algorithm to trace the edges, which results in Algorithm 2. Given an instance $\langle C, (V, E)\rangle$ of a CFL-reachability problem, we first make the function call $\textsc{record}(\langle C, (V, E)\rangle)$ to run the CFL-reachability algorithm and record the meta-information "$\mathsf{metaInfo}$", where "$\mathsf{metaInfo}[e]$" contains all edges that directly contributed to the construction of e. Notice that RECORD is almost the same as the standard CFL-reachability algorithm (Algorithm 1), except that we add the highlighted lines to record the meta-information. Then the original graph's C-contributing edges could be obtained by calling $\textsc{collect}(\mathsf{metaInfo}, E)$, which recursively collects the contributing edges.[1]

The following theorem demonstrates the correctness of Algorithm 2.

Theorem 2 (Tracing Algorithm's Correctness). *For any instance of CFL-reachability $\langle C, (V, E)\rangle$, we have*

$$\textsc{collect}(\textsc{record}(\langle C, (V, E)\rangle)) = \mathsf{Ctri}(C, (V, E)).$$

[1] COLLECT can be implemented using either breadth-first-search or depth-first-search.

Algorithm 2 The Tracing Algorithm

```
1: function RECORD(⟨C, (V, E)⟩)
2:     metaInfo ← emptyMap()
3:     W ← emptyWorkList()
4:     W.addAll(E)
5:     for X → ϵ ∈ CFG do
6:         for v ∈ V do
7:             if X⟨v, v⟩ ∉ E then
8:                 add X⟨v, v⟩ to E and W
9:     while W.nonEmpty() do
10:        Y⟨i, j⟩ ← W.pop()
11:        for X → Y ∈ CFG do
12:            metaInfo[X⟨i, j⟩].add(Y⟨i, j⟩)
13:            if X⟨i, j⟩ ∉ E then
14:                add X⟨i, j⟩ to E and W
15:        for X → YZ ∈ CFG do
16:            for Z⟨j, k⟩ ∈ E do
17:                metaInfo[X⟨i, k⟩].add(Y⟨i, j⟩, Z⟨j, k⟩)
18:                if X⟨i, k⟩ ∉ E then
19:                    add X⟨i, k⟩ to E and W
20:        for X → ZY ∈ CFG do
21:            for Z⟨k, i⟩ ∈ E do
22:                metaInfo[X⟨k, j⟩].add(Z⟨k, i⟩, Y⟨i, j⟩)
23:                if X⟨k, j⟩ ∉ E then
24:                    add X⟨k, j⟩ to E and W
25:    return ((V, E), metaInfo)
26:
27: function COLLECT(((V, E), metaInfo))
28:    visited ← ∅
29:    con ← ∅
30:    function COLLECTDFS(e)
31:        if e ∉ visited then
32:            visited.add(e)
33:            if e's label is a terminal symbol then
34:                con.add(e)
35:            for e′ ∈ metaInfo[e] do
36:                COLLECTDFS(e′)
37:    for e ∈ E do
38:        if e's label is the start symbol then
39:            COLLECTDFS(e)
40:    return con
```

Proof. Suppose $e \in \textsc{collect}(\textsc{record}(\langle C, (V, E)\rangle))$. According to Algorithm 2, we have $e \in E$, and there exists a finite sequence of edges $e_1 = e, e_2, \ldots, e_n$ (we

can move e to the beginning), where each edge is either from the original edge set or obtained from applying one production in C to some preceding edges, e directly or indirectly contributes to e_n, and e_n's label is C's start symbol. So $e \in \mathsf{Ctri}(C, (V, E))$. Conversely, suppose $e \in \mathsf{Ctri}(C, (V, E))$, then $e \in E$ and there exists a finite sequence of edges $e_1 = e, e_2, \ldots, e_n$, where each edge is either from the original edge set or obtained by applying one production in C to some preceding edges, e directly or indirectly contributes to e_n, and e_n's label is C's start symbol. According to Theorem 1, every edge in this sequence must be added to the edge set by Algorithm 1 (and thus also by Algorithm 2). Now let us consider e_n. It is not in the original edge set because its symbol is a non-terminal symbol. Thus e_n can only be obtained by applying one production on some previous edges $\{e_{i_1}, e_{i_2}, \ldots, e_{i_{n_i}}\}$ (which are called the dependency edges of e_n). Since all dependency edges of e_n were added by the algorithm, one edge must be the last one added, and when that one was added, all of e_n's dependency edges were added to the meta-information of e_n (*i.e.*, $\mathsf{metaInfo}(e_n)$). Thus the COLLECTDFS procedure can visit all dependency edges of e_n. Such dependency edges can be reasoned similarly in a depth-first-search manner, and because e_1 directly or indirectly contributes to e_n, eventually COLLECTDFS visits the edge $e_1 = e$. So $e \in \textsc{collect}(\textsc{record}(\langle C, (V, E)\rangle))$.

Complexity Analysis for Algorithm 2. For RECORD, suppose the grammar of the CFL is fixed, the graph supports constant time edge addition and linear time adjacent vertices traversal, the worklist supports constant time pushing/popping, and the operations on metaInfo and the edge set corresponding to each key have logarithmic time complexity and linear space complexity with respect to the number of elements. There could be at most $\tilde{O}(|V|^2)$ edges popped out from the worklist at line 10, and for each edge popped out, there could be at most $\tilde{O}(|V|)$ adjacent edges to try in the main **while** loop. The size of metaInfo is bounded by $\tilde{O}(|V|^2)$ and the size of the edge set corresponding to each key is bounded by $\tilde{O}(|V|)$, so each addition operation to metaInfo has a complexity of $\tilde{O}(\log(|V|^2)+\log|V|) = \tilde{O}(\log|V|)$, which can be hidden by our $\tilde{O}$ notation. Thus the time complexity of RECORD is $\tilde{O}(|V|^3)$. The space complexity of RECORD is $\tilde{O}(|V|^3)$ since metaInfo dominates the space complexity and there could be at most $\tilde{O}(|V|^3)$ edge additions to metaInfo. For COLLECT, suppose visited and con also supports logarithm time operations. Consider a new graph where vertices are edges in E, and if $e_2 \in \mathsf{metaInfo}[e_1]$ then we have a "super edge" from e_1 to e_2. It is easy to see that in this new graph, there are at most $\tilde{O}(|V|^2)$ vertices, and the in-degree and out-degree of each vertex are both bounded by $\tilde{O}(|V|)$. Then COLLECT essentially did a depth-first search on this new graph, whose time complexity is determined by the maximum number of "super edges" in this new graph: $\tilde{O}(|V|^2 \times |V| \times c \cdot \log|V|) = \tilde{O}(|V|^3)$. The logarithm factor is due to operations on visited, metaInfo, and con, but is hidden by our notation $\tilde{O}$. The space complexity is bounded by the sizes of the graph ($\tilde{O}(|V|^2)$), visited ($\tilde{O}(|V|^2)$), con ($\tilde{O}(|V|^2)$), metaInfo ($\tilde{O}(|V|^3)$), and the maximum depth of recursive calls ($\tilde{O}(|V|^2)$). Therefore, the space complexity of COLLECT is $\tilde{O}(|V|^3)$.

Table 1: Time/space complexities of Algorithm 1 and Algorithm 2. The CFL size is assumed to be a constant, and the input graph is $G = (V, E)$.

Algorithm	Time Complexity	Space Complexity
Algorithm 1 (The Standard Algorithm)	$\tilde{O}(\|V\|^3)$	$\tilde{O}(\|V\|^2)$
Algorithm 2 (Our Tracing Algorithm)	$\tilde{O}(\|V\|^3)$	$\tilde{O}(\|V\|^3)$

Table 1 compares the time/space complexities of the standard CFL-reachability algorithm (Algorithm 1) and our tracing version (Algorithm 2). In practice, however, the running time and space also depend on the constant and logarithm factors, the computer architecture, etc.

4.4 Mutual Refinement Algorithm

This section precisely defines the mutual refinement algorithm for a computationally hard L-reachability problem with multiple CFL approximations.

Definition 4 (Refinement Sequence). *Given an instance $\langle L, (V, E)\rangle$ of L-reachability problem and m different CFLs $C_1, \ldots, C_m$ ($m \geq 2$), each of which over-approximates L, a refinement sequence is a finite sequence of sets of edges $E_1, \ldots, E_n$, such that $E_1 = E$ and for all $i \geq 2$, E_i is either $\mathsf{Ctri}(C_j, (V, E_k))$ where $j \in \{1, 2, \ldots, m\}$ and $k \in \{1, 2, \ldots, i-1\}$, or $E_j \cap E_k$ where $j, k \in \{1, 2, \ldots, i-1\}$.*

There exists a global minimum edge set (with respect to the set inclusion relation) that can be computed via a fix-point algorithm. This is due to the following monotonicity property of contributing edges.

Lemma 1 (Monotonicity of Contributing Edges). *Given a formal language L, the following formula holds for all problem instances.*

$$E_1 \subseteq E_2 \implies \mathsf{Ctri}(L, (V, E_1)) \subseteq \mathsf{Ctri}(L, (V, E_2))$$

Proof. This is because any L-path in E_1 is also an L-path in E_2.

Theorem 3 (Minimum Edge Set). *Given an instance $\langle L, (V, E)\rangle$ of the L-reachability problem and m different CFLs $C_1, C_2, \ldots, C_m$ ($m \geq 2$), each of which over-approximates L, there exists an edge set $E_{\min}$, which could be obtained by a specific refinement sequence, and which is a subset of all edge sets in all refinement sequences.*

Proof. Consider the process of applying m algorithms successively in an arbitrary order to refine the edge sets: apply C_{i_1}-reachability on E to get E_1, apply C_{i_2}-reachability on E_1 to get E_2, ..., apply C_{i_m}-reachability on E_{m-1} to get E_m, where $C_{i_1}, C_{i_2}, \ldots, C_{i_m}$ is an arbitrary permutation of $C_1, C_2, \ldots, C_m$. For simplicity, we still denote this order as $C_1, C_2, \ldots, C_m$. This is called one round.

Algorithm 3 The Mutual Refinement (MR) Algorithm

```
1: function MR((V, E), {C_1, ..., C_m})
2:     G ← (V, E)
3:     while true do
4:         s ← G.E.size()
5:         for C_i ∈ {C_1, ..., C_m} do
6:             G.E ← COLLECT(RECORD(⟨C_i, G⟩))
7:         if G.E.size() == s then
8:             return G
```

By doing such rounds multiple times until a state where applying another round does not change the edge set, we get the set $E_{\min}$.

First, we prove its termination: after each round, the size of E either decreases or remains the same, but the size of E cannot decrease indefinitely.

Second, we show that $E_{\min}$ is indeed the globally minimal set with respect to the subset relation: given any refinement sequence $E_1, \ldots, E_n$, since it is non-increasing with respect to the set inclusion relation, we just need to prove $E_{\min} \subseteq E_n$. We prove this by induction on the lengths of refinement sequences. First, there is only one possible refinement sequence of length 1, which is E itself, and it is obvious that $E_{\min} \subseteq E$. Now suppose that for all refinement sequences of length at most n, $E_{\min}$ is a subset of the last edge set in the sequence. Consider an arbitrary refinement sequence of length $n+1$: $E_1, \ldots, E_{n+1}$. Here E_{n+1} is either the intersection of two previous edge sets or obtained by applying C_j-reachability ($j \in \{1, \ldots, m\}$) to one of the previous edge sets. In the first case, by the induction hypothesis, the two previous edge sets all contain $E_{\min}$, so E_{n+1} also contains $E_{\min}$. In the second case, suppose C_j-reachability is applied to E_i ($1 \le i \le n$). By the induction hypothesis, we have $E_{\min} \subseteq E_i$, and because the set of contributing edges is monotonic in the sense described in Lemma 1, we further have $E_{\min} = \mathsf{Ctri}(C_j, (V, E_{\min})) \subseteq \mathsf{Ctri}(C_j, (V, E_i)) = E_{n+1}$, where the first equality holds according to the definition of $E_{\min}$.

Algorithm 3 (mutual refinement) gives the complete procedure for finding the minimum edge set described in Theorem 3. It keeps iterating over the given set of algorithms (line 5) until the edge set's size does not change (line 7). Due to Theorem 3, this algorithm is guaranteed to terminate and produce the optimal result among all possible refinement sequences.

Theorem 4 (Mutual Refinement Soundness). *If every CFL in $\{C_1, \ldots, C_m\}$ over-approximates L, then Algorithm 3 does not miss any L-contributing edges.*

Proof. This is immediate from Theorem 2 and Theorem 3.

The final reachability result can be obtained by executing C_1, C_2, …, C_m-reachability on the minimum graph produced by Algorithm 3, and reporting the pairs that all those CFL-reachability executions report as reachable. In fact, the last iteration of the while loop in Algorithm 3 already does this.

Theorem 5 (Precision Guarantee). *Suppose we have an instance $\langle L, (V, E)\rangle$ of L-reachability problem and m different CFLs $C_1, \ldots, C_m$, each of which over-approximates L. Let E_{min} be the set of edges obtained by executing Algorithm 3 on (V, E) and $C_1, \ldots, C_m$. Suppose $P_1, \ldots, P_m$ are the sets of reachable pairs obtained by executing $C_1, \ldots, C_m$ reachabilities on (V, E), and $Q_1, \ldots, Q_m$ are the sets of reachable pairs obtained by executing $C_1, \ldots, C_m$ reachabilities on (V, E_{min}). Then $\bigcap_{i=1}^{m} Q_i \subseteq \bigcap_{i=1}^{m} P_i$.*

Proof. Since $E_{\mathsf{min}} \subseteq E$, it is immediate that $\forall i \in \{1, \ldots, m\}, Q_i \subseteq P_i$, because any C_i-path connecting two vertices in (V, E_{min}) is also present in (V, E).

The time and space complexities of Algorithm 3 depend on the number of iterations, which highly depends on the graph structure and the CFLs. $O(|E|)$ is a very loose upper bound of the number of iterations because in each iteration before the last one, we remove at least one edge. Our evaluation (Section 5) shows that for specific program analysis problems and graphs with edge sizes up to 184k, the number of iterations can still be within five.

Example 2 (Mutual Refinement Example). If we apply mutual refinement to the motivating example discussed in Section 2, where in each round we apply C_B-reachability and C_P-reachability in sequence, then after two rounds, the graph stabilizes. Figure 6 shows this process.

5 Experiments

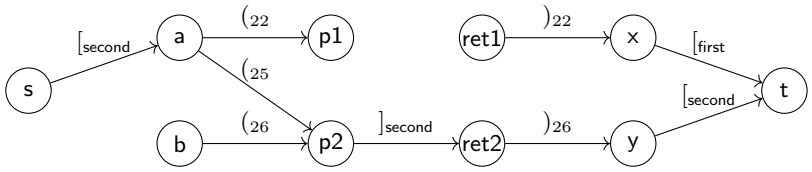
We evaluate mutual refinement on two applications: a taint analysis for Java programs obtained from Android apps [10], and a value-flow analysis for LLVM IR programs obtained from the SPEC CPU 2017 benchmark [28].

When processing experimental data, we use arithmetic means $(\frac{1}{n}\sum_{i=1}^{n} x_i)$ for the average of absolute numbers, and use geometric means $(\sqrt[n]{\prod_{i=1}^{n} x_i})$ for the average of ratios [7]. Also, for the measurement of precision (the number of reachable pairs), we exclude trivial pairs (u, u) and only consider pairs (u, v) where $u \neq v$.

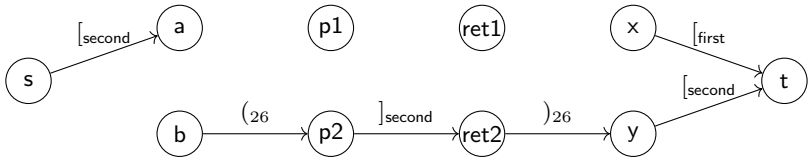
5.1 Experimental Setup

Taint Analysis. We apply our approach to a context-sensitive field-sensitive taint analysis for Java programs obtained from Android apps [10]. The analysis goal is to determine all pairs of variables (s, t) where sensitive information from variable s can flow into variable t. Parentheses model context-sensitivity and brackets model field-sensitivity. A valid path string is an arbitrary interleaving of two strings derived from the two CFLs P and B shown in Figure 7a. This is the interleaved-Dyck reachability problem, which is undecidable [24].

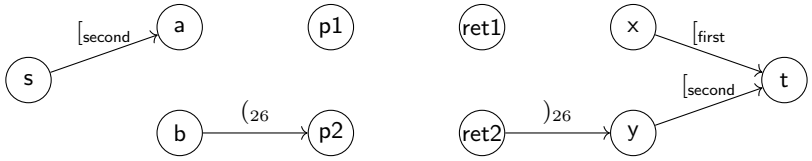
Unlike the example in Section 2, which considers sources/sinks within one function (matched parentheses) and counts flowing into fields as leaks (unmatched brackets), in our experiments, we only count leaks from variables to



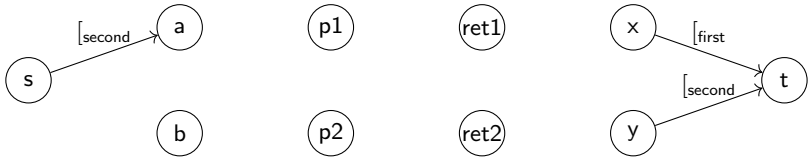
(a) After running C_B-reachability in the first round



(b) After running C_P-reachability in the first round



(c) After running C_B-reachability in the second round



(d) After running C_P-reachability in the second round

Fig. 6: Mutual refinement's iteration process on the motivating example discussed in Section 2. It takes two rounds to converge. If we only consider (s, t)-reachability, then the iteration can stop after the second iteration.

variables, thus disallowing unmatched brackets. One reason is that this is the formulation in the original work [10], and the other reason is that we also evaluate the LZR algorithm [18] on these benchmarks, which only supports matched brackets. In general, the grammar can be adjusted according to needs.

To use mutual refinement, we choose two CFLs (C_P, C_B) over-approximating the interleaved-Dyck language L, where C_P models parentheses matching and C_B models brackets matching. Their grammars are shown in Figure 7b. The execution order is C_P, C_B in each round of mutual refinement.

The benchmarks are selected from the original paper [10]. Specifically, we chose the Contagio malware apps and used the implementation of the client

$$P \to P\ P \mid (_1\ P\ )_1 \mid \ldots \mid (_k\ P\ )_k \mid \epsilon$$

$$B \to B\ B \mid [_1\ B\ ]_1 \mid \ldots \mid [_l\ B\ ]_l \mid \epsilon$$

$$L = P \odot B$$

(a) The taint analysis is formulated as the well-known interleaved-Dyck-reachability problem.

$$C_P \to C_P\ C_P \mid (_1\ C_P\ )_1 \mid \ldots \mid (_k\ C_P\ )_k \mid I \mid \epsilon$$
$$I \to [_1\ ]_1 \mid \ldots \mid [_l\ ]_l$$

$$C_B \to C_B\ C_B \mid [_1\ C_B\ ]_1 \mid \ldots \mid [_l\ C_B\ ]_l \mid J \mid \epsilon$$
$$J \to (_1\ )_1 \mid \ldots \mid (_k\ )_k \mid \epsilon$$

(b) We use the above two CFLs C_P and C_B to over-approximate the interleaved-Dyck language.

Fig. 7: The taint analysis formulation and approximation.

analysis to obtain the graphs.[2] We excluded benchmarks that our tool or the original reference's tool failed to handle. Finally, we got 15 benchmarks, and the size information of the APK files and the graphs is shown in Table 2a.

Value-Flow Analysis. We also apply our approach to a context-sensitive value-flow analysis for LLVM IR programs obtained from the SPEC CPU 2017 benchmark [28]. The analysis goal is to determine all pairs of store/load instructions ($\mathsf{store}\ v_1\ \mathsf{to}\ p_1, \mathsf{load}\ v_2\ \mathsf{from}\ p_2$) where the value of v_1 can flow into v_2 via intermediate assignments and loads/stores. In this case, context-sensitivity is modeled using matched parentheses; memory stores and loads are modeled using matched brackets (in this case, there is only one type of brackets); normal copies of values are modeled using edges with a special label n. Furthermore, since we are interested in store/load pairs, the first edge in the path string must be a memory store, and the last edge must be a memory load. A valid path string is an interleaving of three strings derived from the three CFLs P, B, and N shown in Figure 8a where the first symbol must be [and the last symbol must be]. We denote this formal language as L_V. Section 6 shows the existing $D_1 \odot D_k$-reachability problem, whose decidability is currently open [13], is reducible to the L_V-reachability problem.

In order to use mutual refinement, we choose three CFLs (C_P, C_B, and C_E) over-approximating the underlying problem, where C_P models parentheses matching, C_B models brackets matching, and C_E enforces that the first and last edges of the path must be an open bracket and a closed bracket, respectively. Specifically, they have the three grammars shown in Figure 8b. The execution order is C_P, C_B, C_E in each round of mutual refinement.

The benchmarks are compiled using Clang version 12 [16] to bitcode files. The graphs are generated by the open-source static value-flow analysis framework SVF [31]. We did not include small programs (with bitcode file sizes $<$ 1MB) or programs that failed to be compiled or linked. Finally, we got 10 benchmarks, and the size information of the bitcode files and the graphs is shown in Table 2b.

[2] https://github.com/proganalysis/type-inference.

$$
\begin{aligned}
P &\to P\ P \mid (_1\ P\ )_1 \mid \ldots \mid (_k\ P\ )_k \mid \epsilon \\
B &\to B\ B \mid [\ B\ ] \mid \epsilon \\
N &\to n\ N \mid \epsilon \\
L_V &= ((P \odot B) \odot N) \cap \{s \mid s = [*]\}
\end{aligned}
$$

(a) The value-flow analysis is formulated as an L_V-reachability problem.

$$
\begin{aligned}
C_P &\to C_P\ C_P \mid (_1\ C_P\ )_1 \mid \ldots \mid (_k\ C_P\ )_k \mid I \mid \epsilon \\
I &\to [|\ ] \mid n \\
C_B &\to C_B\ C_B \mid [\ C_B\ ] \mid J \mid \epsilon \\
J &\to (_1|\ )_1 \mid \ldots \mid (_k|\ )_k \mid n \\
C_E &\to [\ K\ ] \\
K &\to K\ K \mid [|\ ] \mid (_1|\ )_1 \mid \ldots \mid (_k|\ )_k \mid n
\end{aligned}
$$

(b) We use the above three CFLs C_P, C_B, and C_E to over-approximate L_V.

Fig. 8: The value-flow analysis formulation and approximation.

Benchmark	APK Size (M)	Graph Size ($\lvert V\rvert, \lvert E\rvert$)
backflash	0.75	(544, 2048)
batterydoc	0.51	(1674, 4790)
droidkongfu	0.08	(734, 1983)
fakebanker	5.17	(434, 1103)
fakedaum	0.14	(1144, 2603)
faketaobao	0.44	(222, 450)
jollyserv	0.42	(488, 998)
loozfon	0.04	(152, 323)
phospy	0.18	(4402, 15660)
roidsec	0.03	(553, 2026)
scipiex	0.31	(1809, 5820)
simhosy	1.43	(4253, 13768)
skullkey	6.63	(18862, 69599)
uranai	0.07	(568, 1246)
zertsecurity	0.10	(281, 710)

(a) Taint Analysis Graphs.

Benchmark	Bitcode Size (M)	Graph Size ($\lvert V\rvert, \lvert E\rvert$)
cactus	5.61	(101325, 114805)
imagick	13.68	(103594, 131707)
leela	2.93	(16134, 19110)
nab	1.41	(12727, 13605)
omnetpp	20.80	(171502, 184601)
parest	16.20	(84355, 93493)
perlbench	11.88	(125345, 160958)
povray	7.38	(61802, 71892)
x264	4.68	(49806, 56376)
xz	1.24	(9918, 10767)

(b) Value-flow Analysis Graphs.

Table 2: Benchmark statistics.

Research Questions. Our experiments aim to answer the following questions.

- *RQ1:* Can mutual refinement achieve better precision compared with the straightforward intersection (baseline) on the two applications?
- *RQ2:* What is the time/space overhead that mutual refinement incurs compared with the straightforward intersection (baseline), and how many rounds does mutual refinement take to converge?
- *RQ3:* Can the LZR graph simplification algorithm improve the precision/performance of mutual refinement on the taint analysis application?

Implementation and Experiment Execution. We implemented mutual refinement in C++17.[3] All experiments were performed on a machine running Ubuntu 20.04.2 LTS. We set a timeout of 4 hours and a space limit of 128 GB for each algorithm's execution on each benchmark item. For *RQ3*, we used the original implementation of the LZR algorithm available online.[4] Since the LZR algorithm is fast enough, we did not set time/space limits on its executions.

5.2 RQ1: Precision Improvement

According to Theorem 5, mutual refinement's precision is at least as good as the straightforward intersection. We define the precision improvement as $(P_{\mathsf{Baseline}}/P_{\mathsf{MR}}) - 1$, where P_{Baseline} and P_{MR} represent the number of reachable pairs computed by the straightforward intersection (baseline) and mutual refinement, respectively. Table 3 shows that, on average, mutual refinement achieves 50.95% precision improvement on the taint analysis benchmarks and 9.37% precision improvement on the value-flow analysis benchmarks. Note that the improvement greatly depends on specific applications and benchmarks.

Summary: On the two program analysis applications, mutual refinement can achieve visibly better precision compared with the straightforward intersection.

5.3 RQ2: Performance Overhead

Mutual refinement traces the sets of contributing edges, which can cost more time and space. Also, mutual refinement might need several rounds to converge. As shown in Table 3 and Figure 9, on average, on the taint analysis benchmarks, mutual refinement takes 2.93 rounds to converge and consumes 2.65× time and 3.23× memory compared with the baseline; on the value-flow analysis benchmarks, mutual refinement takes 3.13 rounds to converge and consumes 2.55× time and 2.22× memory compared with the baseline. In some cases, mutual refinement's space consumption can be much higher. For example, mutual refinement incurs a 80.58× space increase on the phospy benchmark in Table 3a. And there is a trend that larger graphs result in larger differences in memory consumption, which reflects the space complexity difference ($\tilde{O}(|V|^2)$ and $\tilde{O}(|V|^3)$). Section 6.4 discusses mutual refinement's memory cost in detail.

However, mutual refinement can also simplify the graph during the execution of each CFL-reachability, while the straightforward intersection cannot. This could lead mutual refinement to consume less resources in certain cases. For example, on the cactus benchmark in Table 3b, mutual refinement consumes less time than the straightforward intersection.

Summary: Mutual refinement typically needs more time and space, but the average time/space increase on the two program analysis applications is within 5×, and the number of iterations needed to converge is within five.

[3] The implementation is available on GitHub (https://github.com/sdingcn/mutual-refinement) and Zenodo (https://doi.org/10.5281/zenodo.8191389). Certain low-level data structure optimizations were used.

[4] https://github.com/yuanboli233/interdyck_graph_reduce.

Benchmark	Iterations	Precision (Pairs)		Time (Seconds)		Space (MB)	
		Baseline	MR	Baseline	MR	Baseline	MR
backflash	2	6080	2870	0.42	0.67	16.06	36.70
batterydoc	3	8386	5484	0.93	6.06	31.43	133.76
droidkongfu	3	6471	5442	0.32	4.55	16.93	90.06
fakebanker	3	1407	1172	0.04	0.12	9.17	11.96
fakedaum	3	3507	3243	0.28	1.07	18.16	37.40
faketaobao	3	398	328	0.01	0.02	6.98	7.23
jollyserv	3	562	303	0.10	0.14	10.61	15.62
loozfon	2	441	424	0.01	0.04	6.74	9.62
phospy	3	103961	81925	1309.64	9436.26	1202.29	96882.00
roidsec	2	17301	16425	1.79	5.94	28.85	168.93
scipiex	4	20542	10210	69.96	266.33	209.97	3471.78
simhosy	3	100552	41992	102.10	110.78	363.91	1669.46
skullkey	-	-	-	-	-	-	-
uranai	4	353	148	0.11	0.08	10.54	10.56
zertsecurity	3	1969	1110	0.26	0.16	11.24	13.93

(a) Taint Analysis Results.

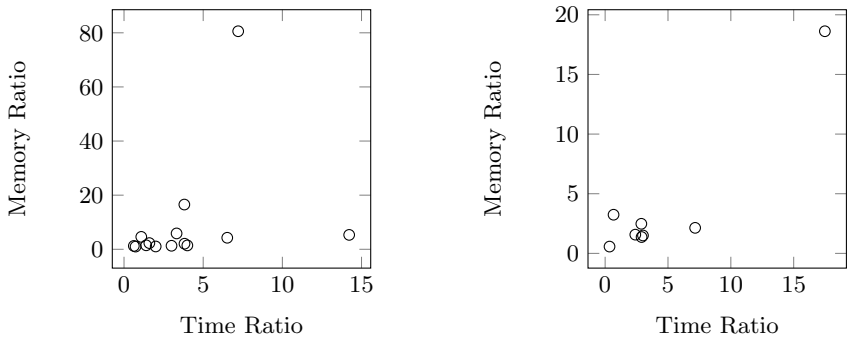
Benchmark	Iterations	Precision (Pairs)		Time (Seconds)		Space (MB)	
		Baseline	MR	Baseline	MR	Baseline	MR
cactus	4	46502	46421	621.36	419.39	2888.79	9373.77
imagick	-	22091	-	14088.50	-	12211.74	-
leela	3	392	392	0.81	1.94	78.01	122.45
nab	3	1958	1788	0.30	0.87	51.83	70.76
omnetpp	4	90412	50568	76.70	221.21	1769.57	4396.33
parest	3	4571	4571	2.89	8.70	243.79	364.86
perlbench	-	-	-	-	-	-	-
povray	3	7453	7230	18.18	6.43	455.19	260.73
x264	3	61577	60792	47.01	821.81	571.96	10650.62
xz	2	211	211	0.32	2.29	41.10	87.94

(b) Value-flow Analysis Results.

Table 3: Precision and performance results. We present the number of rounds that mutual refinement takes to converge, as well as the comparison of precision/time/space between the straightforward intersection (baseline) and mutual refinement. "-" means time/space limits are exceeded.

5.4 RQ3: Combination with the LZR Algorithm

The LZR graph simplification algorithm [18] works for interleaved-Dyck-reachability, and is thus applicable to our taint analysis benchmarks as a pre-processing step. One important detail is that the LZR algorithm does graph edge contractions before calculating the contributing edges, while mutual refinement doesn't. This can lead to edges counted as contributing edges in mutual refinement but not



(a) Overhead on taint analysis.

(b) Overhead on value-flow analysis.

Fig. 9: Mutual refinement's performance overhead scatter plots (ratios). Time ratios are mutual refinement's time consumption numbers divided by the baseline's time consumption numbers. Memory ratios are similar.

counted as contributing edges in the LZR algorithm, because contracting edges can make LZR ignore the contracted edges. As a result, the LZR algorithm can potentially remove certain edges that mutual refinement cannot.

Table 4 compares (1) executing mutual refinement on the original graphs and (2) executing mutual refinement on the graphs simplified by the LZR algorithm. In the (2) case, the time/space consumption includes both algorithms (LZR and MR). On average, LZR can improve the precision of mutual refinement by 3.27%, and this relatively small number shows that mutual refinement itself can already achieve very high precision. Indeed, LZR+MR can reduce 81.38% more edges on average compared with LZR alone. LZR can also boost mutual refinement in terms of time/space consumption, such as the phospy benchmark in Table 4. Notably, the space consumption of LZR+MR is always lower-bounded by 269 MB, and that is because LZR has a minimal memory consumption of roughly 269 MB. This might be due to implementation details.

Summary: LZR can improve mutual refinement's precision, but only to a small extent (3.27% on average). Since LZR is fast, it can boost mutual refinement's performance in certain cases.

6 Discussion

6.1 Generality of Mutual Refinement

Mutual refinement can approximate any language-reachability problem as long as there exist CFL-reachability-based over-approximations. In particular, it is not restricted to the interleaved-Dyck reachability or any particular problem. We have shown two examples L_T and L_V in our motivating example and experiments, and here we show (1) L_T is not a CFL, and (2) $D_1 \odot D_k$-reachability, whose decidability is currently open [13], is reducible to L_V-reachability.

Benchmark	Precision (Pairs)		Time (Seconds)		Space (MB)		Edge Reduction	
	MR	LZR+MR	MR	LZR+MR	MR	LZR+MR	LZR	LZR+MR
backflash	2870	2870	0.67	0.60	36.70	269.18	677	1434
batterydoc	5484	5438	6.06	1.54	133.76	269.33	1826	3327
droidkongfu	5442	5422	4.55	2.66	90.06	269.40	589	1070
fakebanker	1172	928	0.12	0.17	11.96	269.17	414	745
fakedaum	3243	3243	1.07	0.87	37.40	269.18	1103	1747
faketaobao	328	325	0.02	0.10	7.23	269.13	191	309
jollyserv	303	303	0.14	0.25	15.62	269.01	361	634
loozfon	424	424	0.04	0.08	9.62	269.08	105	189
phospy	81925	80200	9436.26	8849.91	96882.00	69532.63	3838	6659
roidsec	16425	16425	5.94	5.86	168.93	269.12	605	1077
scipiex	10210	10173	266.33	260.05	3471.78	3426.72	1219	2683
simhosy	41992	35419	110.78	78.66	1669.46	1217.93	4801	8866
skullkey	-	-	-	-	-	-	19408	-
uranai	148	148	0.08	0.21	10.56	269.10	566	1063
zertsecurity	1110	1110	0.16	0.22	13.93	269.11	253	438

Table 4: A comparison between the original mutual refinement and the one combined with the LZR algorithm, including precision, time, space, and edge reduction. "-" means time/space limits for mutual refinement are exceeded.

L_T is not a CFL. Suppose L_T is a CFL. We construct a formal language M, each string m of which is an arbitrary interleaving of $p \in P$ and $d \in D$ in Section 2, interspersed with an arbitrary number of special symbols $a_1, \ldots, a_l$. Obviously, the language homomorphism

$$f(x) = \begin{cases} [_i \ , x = a_i \\ x \ , \text{otherwise} \end{cases}$$

maps M to L_T. By the assumption L_T is context-free, so M is also context-free since CFL is closed under inverse homomorphisms. Consider the regular language $R = \{s \mid s \text{ does not contain } a_1, \ldots, a_l\}$. Since the intersection of a context-free language and a regular language is also context-free, we have $M \cap R$ is context-free, but this is the interleaved-Dyck language, which is known to be non-context-free [24]. This is a contradiction.

$D_1 \odot D_k$-reachability is reducible to L_V-reachability. $D_1 \odot D_k$ is arbitrary interleaving of two Dyck languages D_1 and D_k, where D_1 is a Dyck language with one kind of parenthesis and D_k is a Dyck language with k kinds of parenthesis. Given any labeled graph consisting of only labels from D_1 and D_k, a pair of vertices (s, t) is $D_1 \odot D_k$-reachable if and only if (s_0, t_0) is L_T-reachable where we just add two vertices s_0 and t_0, and two edges $s_0 \xrightarrow{[} s$ and $t \xrightarrow{]} t_0$, where [and] are the open and close brackets in D_1.

6.2 Different Grammars for the Same CFL

A context-free language can be represented by many different context-free grammars. Ambiguous grammars introduce redundancies, so it can affect mutual refinement's performance because there are more derivations to traverse. However, the choice of grammars does not affect the precision, because we only track L-contributing edges and different grammars refer to the same formal language L. This is also reflected in Algorithm 2: $\mathsf{metaInfo}[e]$ is a set eliminating duplicate tracked edges, and con is also a set eliminating duplicate collected edges.

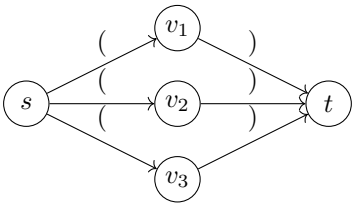
6.3 Order of Mutual Refinement

In mutual refinement, as Theorem 3 shows, any possible orders of executing the CFL-reachability-based over-approximations $C_1, C_2, \cdots, C_m$ result in the same global minimum. However, different orders might affect the convergence speed. In practice, we can run the available CFL-reachability-based over-approximations on sampled programs to find out a "good" order to use, and then execute the "good" order on all programs. There are other heuristics for order choosing, such as executing the one that can result in the best precision first.

6.4 Cost of Mutual Refinement

As shown in Section 5, mutual refinement, in general, needs more time and space than the straightforward intersection. This is because mutual refinement traces the contributing edges and might need more than one iteration to converge. However, after running one CFL-reachability over-approximation, the remaining ones only need to be executed on the simplified graph. Also, in practice, we do not have to wait for the convergence, but can run it for a fixed number of rounds (e.g. two rounds). Other possible optimizations include changing the order of mutual refinement, simplifying the graphs/grammars, etc. Mutual refinement reflects a trade-off between performance and precision.

Memory Overhead of Mutual Refinement. Our experiment shows that in some cases, mutual refinement can take about 80× memory compared with the straightforward intersection. We intuitively explain the reason. Consider the difference between the standard CFL-reachability algorithm (Algorithm 1) and our tracing version (Algorithm 2): in our tracing version, when a new edge is generated, the meta-information about edge dependencies is updated no matter whether the new edge is already in the graph or not. If there are multiple ways to generate the same edge, all of those ways need to be recorded. For example, in the following graph, where we perform the matched-parenthesis reachability with respect to the grammar $S \to S\ S \mid (\ S\ ) \mid \epsilon$, there are three ways to generate the summary edge $s \xrightarrow{S} t$, and all edges will be added to the meta information of $s \xrightarrow{S} t$, despite there is only one such summary edge in the final graph. So the memory cost of mutual refinement can be high. Whether this graph pattern occurs in reality depends on the specific analysis details.



Exploring whether we can reduce the memory cost is an interesting future direction. There exists work compressing information used during static analysis [33].

6.5 Generalization to the Single-Pair Case

In this paper, mutual refinement is formalized to retain the edges contributing to reachable pairs in the CFL-reachability-based over-approximations. Notice that in the single-source-single-target reachability case, we can retain only the edges contributing to the pair that we are interested in, and this can potentially remove even more edges from the graph and thus can also potentially increase the precision as well. We leave this for future work.

6.6 Generalization to Other Algorithms

The idea of tracing in mutual refinement can be potentially generalized to all algorithms using similar "dynamic programming style" approaches. Specifically, as long as the algorithm traverses all edges contributing to the ground truth solution, we can use tracing to extract those edges and use this as a refinement between different such algorithms. It is an interesting future direction to explore the generalization of mutual refinement to broader classes of algorithms.

7 Related Work

CFL-reachability is widely-used in program analysis [15, 19, 22, 23, 26, 30]. It has a (sub)cubic-time dynamic programming style algorithm [2, 20, 25, 35], and faster algorithms exist in special cases [1, 17, 36]. CFL-reachability can model function calls/returns [24, 26], field reads/writes [29, 34], locks/unlocks [11], etc.

In static analysis, many techniques have been proposed to reduce the size of graphs involved in the analysis [5, 18, 21]. Our mutual refinement process simplifies the graphs, but our main focus is leveraging the information of each CFL-reachability-based over-approximation to refine the results. In particular, the LZR fast graph simplification work [18] also defines similar concepts such as contributing edges, but their algorithm is specific for the interleaved-Dyck-reachability problem, while our mutual refinement works for any L-reachability problems preserving CFL-reachability-based over-approximations. Also, LZR is a pre-processor while mutual refinement is a complete solver.

Interleaved-Dyck-reachability is widely used in program analysis [18, 27, 37], but it is undecidable [24], so there exist many approximation algorithms. We can

use one Dyck language to approximate it and employ the standard cubic-time context-free language reachability algorithm [20, 25, 35]. The refinement-based context-sensitive points-to analysis work [29] used the method of modeling one Dyck language precisely while approximating the other Dyck language using a regular language [29]. The linear conjunctive language reachability [37] is another formulation of interleaved-Dyck-reachability which is precise, but the corresponding algorithm is approximate. Synchronized pushdown systems [27] model the idea of considering two context-free languages at the same time. Mutual refinement is not restricted to interleaved-Dyck-reachability.

In static analysis and verification, similar strategies of running different approaches in a staged way such that later stages benefit from earlier stages have been studied, such as the $\mathsf{Unity} - \mathsf{Relay}$ approach [32] to accumulate the precision of different selective context-sensitivity approaches, and the staged verification [6] where faster verifiers run first to reduce the load of later verifiers. Mutual refinement concerns graph-reachability-based program analysis, and we have theorems showing the existence and uniqueness of fix-points.

8 Conclusion

This paper proposed mutual refinement to combine different CFL-reachability over-approximations for computationally hard graph reachability problems. We proved theorems showing the existence and uniqueness of the optimal refinement result, the correctness of mutual refinement, and the precision guarantees. To realize mutual refinement, the modifications to the standard CFL-reachability algorithm are minimal, and the modified version's time/space complexities were carefully analyzed. We also conducted experiments showing that mutual refinement achieved better precision than the straightforward intersection of the sets of reachable vertex pairs, with reasonable extra time and space cost.

Acknowledgements. We thank the anonymous reviewers for their feedback on earlier drafts of this paper. This work was supported, in part, by the United States National Science Foundation (NSF) under grants No. 1917924, No. 2114627, and No. 2237440; and by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-21-C-4024. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

References

1. Chatterjee, K., Choudhary, B., Pavlogiannis, A.: Optimal dyck reachability for data-dependence and alias analysis. Proc. ACM Program. Lang. **2**(POPL), 30:1–30:30 (2018)
2. Chaudhuri, S.: Subcubic algorithms for recursive state machines. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 159–169. ACM (2008)

3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2022)
4. Cousot, P.: Asychronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. (1977)
5. Fähndrich, M., Foster, J.S., Su, Z., Aiken, A.: Partial online cycle elimination in inclusion constraint graphs. In: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998. pp. 85–96. ACM (1998)
6. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. ACM Trans. Softw. Eng. Methodol. **17**(2), 9:1–9:34 (2008)
7. Fleming, P.J., Wallace, J.J.: How not to lie with statistics: The correct way to summarize benchmark results. Commun. ACM **29**(3), 218–221 (1986)
8. Harrison, M.A.: Introduction to formal language theory. Addison-Wesley Longman Publishing Co., Inc. (1978)
9. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation. Acm Sigact News **32**(1), 60–65 (2001)
10. Huang, W., Dong, Y., Milanova, A., Dolby, J.: Scalable and precise taint analysis for android. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015. pp. 106–117. ACM (2015)
11. Kahlon, V.: Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise cfl-reachability for threads communicating via locks. In: Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA. pp. 27–36. IEEE Computer Society (2009)
12. Kildall, G.A.: A unified approach to global program optimization. In: Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973. pp. 194–206. ACM Press (1973)
13. Kjelstrøm, A.H., Pavlogiannis, A.: The decidability and complexity of interleaved bidirected dyck reachability. Proc. ACM Program. Lang. **6**(POPL), 1–26 (2022)
14. Kleene, S.C.: Introduction to metamathematics (1952)
15. Kodumal, J., Aiken, A.: The set constraint/cfl reachability connection in practice. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004. pp. 207–218. ACM (2004)
16. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. pp. 75–88. IEEE Computer Society (2004)
17. Lei, Y., Sui, Y., Ding, S., Zhang, Q.: Taming transitive redundancy for context-free language reachability. Proc. ACM Program. Lang. **6**(OOPSLA2), 1556–1582 (2022)
18. Li, Y., Zhang, Q., Reps, T.W.: Fast graph simplification for interleaved dyck-reachability. In: Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020. pp. 780–793. ACM (2020)
19. Lu, Y., Shang, L., Xie, X., Xue, J.: An incremental points-to analysis with cfl-reachability. In: Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7791, pp. 61–81. Springer (2013)

20. Melski, D., Reps, T.W.: Interconvertibility of a class of set constraints and context-free-language reachability. Theor. Comput. Sci. **248**(1-2), 29–98 (2000)
21. Milanova, A.: Flowcfl: generalized type-based reachability analysis: graph reduction and equivalence of cfl-based and type-based reachability. Proc. ACM Program. Lang. **4**(OOPSLA), 178:1–178:29 (2020)
22. Pratikakis, P., Foster, J.S., Hicks, M.: Existential label flow inference via CFL reachability. In: Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4134, pp. 88–106. Springer (2006)
23. Rehof, J., Fähndrich, M.: Type-base flow analysis: from polymorphic subtyping to cfl-reachability. In: Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001. pp. 54–66. ACM (2001)
24. Reps, T.: Undecidability of context-sensitive data-dependence analysis. ACM Transactions on Programming Languages and Systems (TOPLAS) **22**(1), 162–186 (2000)
25. Reps, T.W.: Program analysis via graph reachability. Inf. Softw. Technol. **40**(11-12), 701–726 (1998)
26. Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995. pp. 49–61. ACM Press (1995)
27. Späth, J., Ali, K., Bodden, E.: Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. Proc. ACM Program. Lang. **3**(POPL), 48:1–48:29 (2019)
28. SPEC: Spec cpu 2017. https://www.spec.org/cpu2017/ (2017), accessed: Nov 6, 2022
29. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for java. In: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006. pp. 387–400. ACM (2006)
30. Su, Y., Ye, D., Xue, J.: Parallel pointer analysis with cfl-reachability. In: 43rd International Conference on Parallel Processing, ICPP 2014, Minneapolis, MN, USA, September 9-12, 2014. pp. 451–460. IEEE Computer Society (2014)
31. Sui, Y., Xue, J.: Svf: interprocedural static value-flow analysis in llvm. In: Proceedings of the 25th international conference on compiler construction. pp. 265–266. ACM (2016)
32. Tan, T., Li, Y., Ma, X., Xu, C., Smaragdakis, Y.: Making pointer analysis more precise by unleashing the power of selective context sensitivity. Proc. ACM Program. Lang. **5**(OOPSLA), 1–27 (2021)
33. Xiao, X., Zhang, Q., Zhou, J., Zhang, C.: Persistent pointer information. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 463–474. ACM (2014)
34. Yan, D., Xu, G., Rountev, A.: Demand-driven context-sensitive alias analysis for java. In: Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011. pp. 155–165. ACM (2011)
35. Yannakakis, M.: Graph-theoretic methods in database theory. In: Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1990. pp. 230–242. ACM Press (1990)

36. Zhang, Q., Lyu, M.R., Yuan, H., Su, Z.: Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. pp. 435–446. ACM (2013)
37. Zhang, Q., Su, Z.: Context-sensitive data-dependence analysis via linear conjunctive language reachability. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017. pp. 344–358. ACM (2017)