Indexing Context-Sensitive Reachability

QINGKAI SHI, Ant Group, China

YONGCHAO WANG, The Hong Kong University of Science and Technology and Ant Group, China CHARLES ZHANG, The Hong Kong University of Science and Technology, China

Many context-sensitive data flow analyses can be formulated as a variant of the all-pairs Dyck-CFL reachability problem, which, in general, is of sub-cubic time complexity and quadratic space complexity. Such high complexity significantly limits the scalability of context-sensitive data flow analysis and is not affordable for analyzing large-scale software. This paper presents Flare, a reduction from the CFL reachability problem to the conventional graph reachability problem for context-sensitive data flow analysis. This reduction allows us to benefit from recent advances in reachability indexing schemes, which often consume almost linear space for answering reachability queries in almost constant time. We have applied our reduction to a context-sensitive alias analysis and a context-sensitive information-flow analysis for C/C++ programs. Experimental results on standard benchmarks and open-source software demonstrate that we can achieve orders of magnitude speedup at the cost of only moderate space to store the indexes. The implementation of our approach is publicly available.

 $CCS\ Concepts: \bullet\ Mathematics\ of\ computing \to Graph\ algorithms; \bullet\ Theory\ of\ computation \to Program\ analysis; \bullet\ Software\ and\ its\ engineering \to Automated\ static\ analysis.$

Additional Key Words and Phrases: Dyck-CFL reachability, context-sensitive data flow analysis, reachability indexing scheme, information-flow analysis, alias analysis.

ACM Reference Format:

Qingkai Shi, Yongchao Wang, and Charles Zhang. 2018. Indexing Context-Sensitive Reachability. *J. ACM* 37, 4, Article 111 (August 2018), 29 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The context-free language (CFL) reachability problem is a generalization of the conventional graph reachability problem [Yannakakis 1990]. A vertex v is CFL-reachable from a vertex u if and only if there is a path from the vertex u to the vertex v, and the string of the edge labels on the path follows a given context-free grammar. CFL reachability has been broadly used in program analysis for a wide range of applications, including context-sensitive data flow analysis [Reps et al. 1995], program slicing [Reps et al. 1994], shape analysis [Reps 1995], type-based flow analysis [Kodumal and Aiken 2004; Milanova 2020; Pratikakis et al. 2006; Rehof and Fähndrich 2001], pointer analysis [Pratikakis et al. 2006; Shang et al. 2012; Sridharan and Bodík 2006; Sridharan et al. 2005; Xu et al. 2009; Yan et al. 2011; Zhang et al. 2013, 2014; Zheng and Rugina 2008], and debugging [Cai et al. 2018], to name just a few.

This paper focuses on the problem of context-sensitive data flow analysis, where an extended Dyck-CFL is used to capture the paired call and return using matched parentheses. We use an

Authors' addresses: Qingkai Shi, Ant Group, China, qingkai.sqk@antgroup.com; Yongchao Wang, The Hong Kong University of Science and Technology and Ant Group, China, ywanghz@cse.ust.hk; Charles Zhang, The Hong Kong University of Science and Technology, Hong Kong, China, charlesz@cse.ust.hk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Association for Computing Machinery.

0004-5411/2018/8-ART111 \$15.00

https://doi.org/10.1145/1122445.1122456

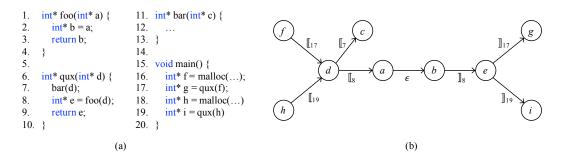


Fig. 1. (a) Code for illustration. (b) Inter-procedural data-dependence graph, where each directed edge represents a data-dependence relation in the code. The parentheses [i] and [i] stand for the call and return operations at Line i.

"extended" Dyck-CFL because the standard one fails to capture many valid data flows containing partially matched parentheses [Kodumal and Aiken 2004]. Intuitively, the extended Dyck-CFL includes all sub-strings of a standard Dyck word. For instance, in the inter-procedural data-dependence graph in Figure 1, the vertex i is context-sensitively reachable from the vertices b because the string of the edge labels, $[a]_{19}$, does not contain any mismatched parentheses and, thus, is a sub-string of a standard Dyck word like $[a]_{19}$ In contrast, the vertex i is not context-sensitively reachable from the vertex f because the string of the edge labels, $[a]_{19}$ contains mismatched parentheses. To distinguish from the standard Dyck-CFL reachability problem, we refer to the extended version as the context-sensitive reachability (CS-reachability) problem as it is specially used in the context-sensitive data flow analysis.

Recently, some fast algorithms have been proposed to address the standard Dyck-CFL reachability problems on a few special graphs, such as trees [Yuan and Eugster 2009; Zhang et al. 2013], bidirected graphs [Chatterjee et al. 2017; Zhang et al. 2013], and graphs of constant tree-width [Chatterjee et al. 2017]. Nevertheless, due to the differences in the underlying CFLs and graph structures, these approaches cannot be directly employed in context-sensitive data flow analysis. In practice, for context-sensitive alias analysis [Li et al. 2011, 2013], information-flow analysis [Arzt et al. 2014; Lerch et al. 2014], and all other IFDS-based data flow analyses, answering a CS-reachability query still relies on the typical tabulation algorithm [Reps et al. 1995, 1994], either in an exhaustive manner or in a demand-driven fashion. The exhaustive manner computes a transitive closure, which is of at least quadratic complexity and is unaffordable for large-scale software. The demand-driven manner traverses the graph for every reachability query and, thus, is not efficient at responding to a query.

This paper proposes indexing schemes for solving the all-pairs CS-reachability problem, so that we can efficiently tell the CS-reachability relation between any pair of vertices without computing an expensive transitive closure or performing a full graph traversal. Our key insight is that the CS-reachability problem can be reduced to a conventional reachability problem within linear time and space, by building a special graph structure we refer to as the indexing graph. Thanks to the recent advances in the field of graph database [Cheng et al. 2013; Jin et al. 2011, 2009; Wang et al. 2006; Yildirim et al. 2010], the reduction allows us to employ existing indexing schemes for conventional graph reachability to significantly speed up CS-reachability queries, at the cost of only a moderate space overhead.

¹By context-sensitive reachability, we do not mean the underlying language is a context-sensitive language but still a context-free language specially used for context-sensitive data flow analysis.

We have implemented a tool, namely Flare, to build the indexing graph for a given context-sensitive analysis, so that the CS-reachability problem can be reduced to the conventional graph reachability problem. Based on the reduction, we then apply two different existing indexing schemes for speeding up context-sensitive information-flow analysis and context-sensitive alias analysis, respectively. In the evaluation, we conducted experiments on twelve standard benchmark programs and four open-source systems to measure the time cost for building indexes, the space cost for storing the indexes, and the query time using the indexes. We also compared our method to a few baseline approaches, which showed that we can achieve orders of magnitude speedup for answering an alias or information-flow query with only a moderate overhead to build and store the indexes. In summary, the principal contributions of this paper are three-fold and listed as follows:

- We propose a reduction of linear time and space complexity from the CS-reachability problem
 to the conventional graph reachability problem. We prove its correctness and analyze its time
 and space complexity.
- We present two typical applications of our reduction, namely context-sensitive informationflow analysis and context-sensitive alias analysis. Through the two applications, we also summarize the criteria of selecting a proper indexing scheme in practice.
- We evaluate the time and the space overhead of building the indexes, and compare our method
 to existing techniques. The results showed orders of magnitude speedup for answering CSreachability queries with just a moderate space overhead.

2 BACKGROUND

In this section, we review the background of context-sensitive reachability (Section 2.1) as well as existing indexing schemes for conventional graph reachability (Section 2.2). We also discuss the connections and the gaps between context-sensitive reachability and existing reachability indexing schemes (Section 2.3).

2.1 Context-Sensitive Reachability

In this paper, we study the all-pairs context-sensitive reachability problem on various flow graphs of a program. These graphs include the program dependence graph [Ferrante et al. 1987], the value-flow graph [Cherem et al. 2007; Sui et al. 2014], the exploded super graph [Reps et al. 1995], and many others. Generally, these graphs can be uniformly defined as a program-valid graph, which captures the modular program structure [Chatterjee et al. 2017].

Definition 2.1 (Program-Valid Graph). Given an alphabet $\Sigma_k = \{\epsilon\} \cup \{[\![i,]\!]_i\}_{i=1}^k$, a program-valid graph G is a Σ_k -labeled directed graph that can be partitioned to sub-graphs such that every sub-graph has only ϵ -labeled edges, and there exists a constant $\alpha \geq 0$ such that every sub-graph has α or fewer vertices with $[\![i]$ -labeled incoming edges or $[\![i]$ -labeled outgoing edges.

Intuitively speaking, every sub-graph in a program-valid graph represents the local graph of a function. The constant α indicates that every function in a program has only a few function parameters and return values. For example, the inter-procedural data-dependence graph in Figure 1 is program-valid because it can be partitioned into four parts, $\{a,b\},\{c\},\{d,e\},$ and $\{f,g\}$. The four parts correspond to the four functions, foo, bar, qux, and main, respectively. Each part has at most two vertices with parenthesis-labeled incoming or outgoing edges, which stand for the function call and return operations in the program. From now on, given a program-valid graph, we use V to represent the vertex set and $E \subseteq V \times V$ the edge set. Given any edge $(u,v) \in E$, $L(u,v) \in \Sigma_k$ returns the edge label.

Fig. 2. The context-free grammar of an extended Dyck-CFL, which is defined on the alphabet $\Sigma_k = \{\epsilon\} \cup \{[\![i,]\!]i\}_{i=1}^k$ for achieving context-sensitivity [Kodumal and Aiken 2004]. The grammar for the standard Dyck-CFL only has the last production, M, that produces matched parentheses.

Definition 2.2 (Context-Sensitive Reachability). Given two vertices v_0 and v_m on a program-valid graph, we say the vertex v_m is context-sensitively reachable (or CS-reachable) from the vertex v_0 if and only if there is a path $(v_0, v_1, v_2, \ldots, v_m)$ on the graph such that the concatenation of the edge labels, $L(v_0, v_1)L(v_1, v_2)\ldots L(v_{m-1}, v_m)$, can be derived from the start symbol S of the context-free grammar in Figure 2.

By definition, the context-free grammar in Figure 2 allows three kinds of CS-reachable paths on the program-valid graph:

- (1) P-paths: paths whose edge-label strings can be derived from the symbol P of the grammar. By definition, a parenthesis on a P-path is either a right-parenthesis or correctly matched. In a program analysis, a P-path often represents the propagation of a data-flow fact from a callee function to a caller function. For instance, the path (b, e, q) in Figure 1 is a P-path.
- (2) N-paths: paths whose edge-label strings can be derived from the symbol N of the grammar. By definition, a parenthesis on an N-path is either a left-parenthesis or correctly matched. In a program analysis, an N-path often represents the propagation of a data-flow fact from a caller function to a callee function. For instance, the path (f, d, a, b) in Figure 1 is an N-path.
- (3) PN-paths: the concatenation of P-paths and N-paths, which implies that a data-flow fact returned from a callee function is passed again to a callee function.

Hence, to answer a CS-reachability query, we in fact need to check if there is a *P*-path, *N*-path, or *PN*-path between two vertices. To this end, the state-of-the-art method is to employ Reps et al. [1995, 1994]'s tabulation algorithm, which has been used in a wide range of applications, including alias analysis [Li et al. 2011, 2013], information-flow analysis [Arzt et al. 2014; Lerch et al. 2014], as well as all other data flow analyses built on top of the IFDS framework [Reps et al. 1995].

Algorithm 1 illustrates the spirit of the tabulation algorithm. Basically, to answer a CS-reachability query, the algorithm performs a depth-first graph traversal over the input program-valid graph but, during the traversal, employs the summary edges to avoid repetitively visiting a function. As defined in Definition 2.3, a summary edge tabulates an input and an output of a function. Reps et al. [1994] showed that the number of summary edges is bounded by $O(\alpha^2|V|)$ and the time to build all summary edges is bounded by $O(\alpha|E| + \alpha^3|V|)$.

As an example of the summary edges, when traversing the inter-procedural data-dependence graph in Figure 1, we can add a summary edge from the vertex d to the vertex e. This summary edge allows us to skip the function qux whenever a graph traversal reaches the vertex d. Since we never visit a function more than once, answering a CS-reachability query using the tabulation algorithm is of linear complexity with respect to the graph size and the number of summary edges.

Definition 2.3 (Summary Edge and Summary Path). A summary edge (v_0, v_m) is an extra edge added to the program-valid graph G = (V, E) such that $v_0, v_m \in V$ and there is a path $(v_0, v_1, v_2, \ldots, v_m)$, which we refer to as a summary path, such that $L(v_0, v_1)L(v_{m-1}, v_m) = [i]_i$, and the label string $= L(v_1, v_2)L(v_2, v_3) \ldots L(v_{m-2}, v_{m-1})$ can be derived from the symbol M of the grammar in Figure 2.

Algorithm 1: Query if the target vertex t is CS-reachable from the source vertex s.

```
1 E^s = \{ a set of summary edges pre-computed using Reps et al. [1994]'s method \};
2 Procedure Query (G = (V, E), s \in V, t \in V)
       if s has been visited before by this procedure then return false;
       if s = t then return true;
       foreach (s, v) \in E \cup E^s do
           if L(s, v) = [i] then
 6
                if Tabulate(G, v, t) then return true;
 8
                if Query(G, v, t) then return true;
       return false;
10
  Procedure Tabulate(G = (V, E), s \in V, t \in V)
       if s has been visited before by this procedure then return false;
12
       if s = t then return true;
13
       foreach (s, v) \in E \cup E^s do
14
           if L(s,v) \neq \|_i then
15
                if Tabulate(G, v, t) then return true;
16
17
       return false;
```

While the tabulation algorithm avoids repetitively visiting a function when answering a CS-reachability query, it is not efficient for frequent CS-reachability queries because we need to traverse the graph for every query. To expedite CS-reachability queries, the usual manner is to build a transitive closure so that we can answer each query in constant time. However, building a transitive closure is notoriously expensive (at least quadratic complexity), which is unaffordable for large-scale graphs. To resolve the dilemma between traversing the graph for every query and computing an expensive transitive closure, this paper proposes a novel use of the summary edges, which allows us to answer each CS-reachability query within "almost" constant time via the indexing schemes for conventional graph reachability.

2.2 Indexing Schemes for Conventional Graph Reachability

Quickly answering conventional reachability queries has been the focus of research for over thirty years due to its wide spectrum of applications. In order to tell whether a vertex can reach another in a directed graph, in general, we can use two "extreme approaches". The first approach can answer any query in O(1) time. However, it comes at the cost of quadratic time and space for computing and storing the transitive closure. The other approach traverses the graph by depth-first or breadth-first search, attempting to find a path between two vertices, which takes linear time and space for each query. This is apparently very slow for frequent queries on a large graph.

Recent studies on indexing schemes aim to find a promising trade-off lying in-between the two extremes, reducing the pre-computation time and storage with "almost" constant answering time. We can put the large amount of indexing schemes into two groups: (1) compression of transitive closure (e.g., [Chen and Chen 2008; Cohen et al. 2003; Jin et al. 2011, 2009; Wang et al. 2006]) and (2) pruned search (e.g., [Chen et al. 2005; Seufert et al. 2013; Wei et al. 2014; Yildirim et al. 2010]).

2.2.1 Compression of Transitive Closure. Approaches in the first group aim to reduce the time and space cost of computing and storing the transitive closure. For instance, assuming k is a variable far less than |V|, the dual-labeling method takes $O(|V| + |E| + k^3)$ time to compress the size of transitive

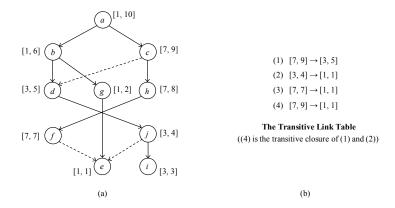


Fig. 3. A transitive-closure-compression-based indexing scheme [Wang et al. 2006]. (a) A graph where one of its spanning trees is represented by the solid edges and non-tree edges are represented by the dashed edges. Vertices are labeled by intervals for reachability queries. (b) The transitive link table for encoding reachability relations implied by the non-tree edges.

closure from $O(|V|^2)$ to $O(|V| + k^2)$, and preserve the capability of answering each reachability query in constant time [Wang et al. 2006].

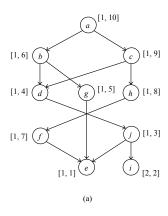
Example 2.4. Figure 3 illustrates the dual-labeling method, which firstly finds a spanning tree on a given directed acyclic graph² and labels each vertex v with an interval $[L_v, H_v]$ according to a traversal on the tree. For each interval, H_v is the rank of the vertex v in a post-order traversal of the tree, where the ranks are assumed to begin at 1; L_v denotes the lowest rank for any vertex in the sub-tree rooted at v. This approach guarantees that the vertex v is reachable from the vertex v on the tree if and only if $[L_v, H_v] \subseteq [L_u, H_u]$, because the post-order traversal enters a vertex before all its descendants and leaves after visiting all of its descendants.

For each non-tree edge, we record it in a transitive link table as illustrated in Figure 3(b), and compute a transitive closure. For instance, since $[7,9] \rightarrow [3,5]$ and $[3,4] \rightarrow [1,1]$ are non-tree edges, and $[3,4] \subseteq [3,5]$, we also include $[7,9] \rightarrow [1,1]$ in the table. We then can determine the reachability relation on the graph as follows: the vertex v is reachable from the vertex v on the graph if and only if $[L_v, H_v] \subseteq [L_u, H_u]$ or there exists an entry $[L_x, H_x] \rightarrow [L_y, H_y]$ in the link table such that $[L_x, H_x] \subseteq [L_u, H_u] \land [L_v, H_v] \subseteq [L_u, H_u]$.

Assuming the graph has k non-tree edges, we need $O(k^3)$ time to compute the transitive link table of size $O(k^2)$. Wang et al. [2006] showed that, when answering a reachability query, it is not necessary to take $O(k^2)$ time to find the entry $[L_x, H_x] \rightarrow [L_y, H_y]$ in the link table. It is actually a special range-temporal aggregation problem and can be solved in O(1) time. Thus, we can answer each reachability query in constant time.

2.2.2 Pruned Search. The pruned-search-based indexing schemes pre-compute information to speed up the depth-first or breadth-first graph traversal by pruning unnecessary searches. Grail is a typical indexing scheme in this group that can scale to very large graphs [Yildirim et al. 2010]. Basically, it labels each vertex with a constant number of intervals. We can tell if a vertex is NOT reachable from another by testing the interval containment. For reachable cases, it falls back to a graph traversal but is capable of using the intervals to prune unreachable paths.

²Since vertices in a strongly connected component (SCC) are reachable from each other, when indexing reachability, we can merge all vertices in every SCC into a single vertex to obtain a directed acyclic graph.



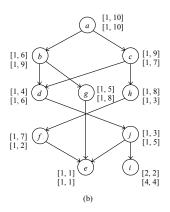


Fig. 4. A pruned-search-based indexing scheme [Yildirim et al. 2010]. (a) A graph where each vertex is labeled by an interval for reachability query. (b) Vertices are labeled by multiple intervals to reduce false positives.

Example 2.5. Figure 4 illustrates the Grail indexing scheme. Given a directed acyclic graph, it labels each vertex v with an interval $[L_v, H_v]$ as seen in Example 2.4 but based on a post-order traversal on the graph, as illustrated in Figure 4(a). The basic idea is that, on the directed acyclic graph, although $[L_v, H_v] \subseteq [L_u, H_u]$ cannot imply that the vertex v is reachable from the vertex u, $[L_v, H_v] \not\subseteq [L_u, H_u]$ is sufficient to imply that the vertex v is NOT reachable from the vertex v. For instance, in Figure 4(a), $[L_j, H_j] = [1, 3] \subseteq [1, 8] = [L_h, H_h]$, but the vertex v is not reachable from the vertex v.

To prune such false positives implied by the interval containment, Grail performs a randomized post-order traversal on the graph multiple times,³ leading to multiple interval labels as shown in Figure 4(b). With multiple interval labels, we can easily determine that the vertex j is not reachable from the vertex h, because the second interval of the vertex j, [1, 5], is not a subset of the second interval of the vertex h, [1, 3].

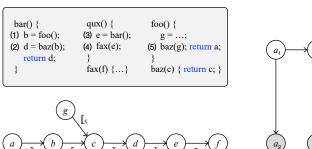
2.3 Gaps between the Indexing Schemes and CS-Reachability

The aforementioned indexing schemes can easily accelerate conventional reachability queries on a common directed graph. However, owing to the edge labels and the constraint brought by the context-free grammar, we cannot use them to speed up CS-reachability queries on a program-valid graph, unless we can address the following problem, i.e., reduce the CS-reachability problem to a conventional reachability problem:

Problem statement: Given a program-valid graph G = (V, E), find a common directed graph G = (V, E) and two functions $src : V \mapsto V$ and $dst : V \mapsto V$, such that, for any pair of vertices, $u, v \in V$ on the program valid graph G, the vertex v is CS-reachable from the vertex u if and only if the vertex dst(v) is reachable from the vertex src(u) on the common directed graph G.

The following sections illustrate an efficient reduction of linear complexity from CS-reachability to conventional graph reachability, which allows us to directly profit from the reachability indexing schemes discussed before.

³We can randomly order the children of each vertex during the post-order graph traversal.



summary edge

(a)

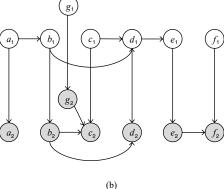


Fig. 5. (a) A program-valid graph with one summary edge. The program-valid graph can be regarded as a data-dependence graph of the code shown above — each directed edge represents a data-dependence relation and the parentheses $[i]_i$ and $[i]_i$ respectively stand for the call and the return at the ith call site. (b) The indexing graph we build to reduce CS-reachability to conventional graph reachability. Each CS-reachability query on the program-valid graph is equivalent to a conventional reachability query on the indexing graph.

3 OVERVIEW

Figure 5(a) shows a program-valid graph with a summary edge from the vertex b to the vertex d. The program-valid graph can be regarded as a data-dependence graph of the code in the figure — each directed edge represents a data-dependence relation and the parentheses [a] and [a] respectively stand for the call and the return at the ath call site. The summary edge is added over the call site ath at

3.1 Reduction in a Nutshell

Our approach, namely Flare, reduces the CS-reachability problem on the program-valid graph to the conventional reachability problem on a common directed graph we refer to as the indexing graph. This reduction allows us to transform a CS-reachability query to an equivalent conventional reachability query on the indexing graph. Thus, we then can directly use the indexing schemes introduced in Section 2.2 for optimization. We focus on addressing two problems: (1) how to build the indexing graph, and (2) how to transform a CS-reachability query to an equivalent query of conventional reachability.

Building the Indexing Graph. As shown in Figure 5(b), the indexing graph consists of two copies of the original program-valid graph. The copies of each vertex v are distinguished by the subscripts v_1 and v_2 . In the first copy, we remove all edges labeled by the left-parentheses, which are known as the call edges. In the second copy, we remove all edges labeled by the right-parentheses, which are known as the return edges. For each vertex v, we add an edge from the first copy v_1 to the second copy v_2 . All edge labels are removed from the indexing graph.

Querying Context-Sensitive Reachability. To answer a CS-reachability query, Q(u, v), which returns true if and only if the vertex v is CS-reachable from the other vertex u on the program-valid graph, we only need to tell if the vertex u_1 and the vertex v_2 have a conventional reachability relation on the indexing graph. Let us use the following queries to illustrate the idea.

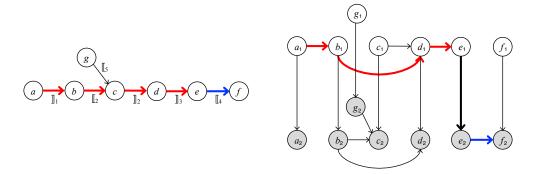


Fig. 6. Illustration of the correctness of our approach. Mapping the **P-path** (a, b, c, d, e), i.e., the red path, from the program-valid graph to the indexing graph. The sub-path (b, c, d) is replaced with the summary edge (b_1, d_1) on the indexing graph. Mapping the **N-path** (e, f), i.e., the blue path, from the program-valid graph to the indexing graph. Mapping the **PN-path**, the concatenation of the *P*-path and the *N*-path, between the program-valid graph and the indexing graph. The *P*-path and the *N*-path are connected via the vertex e on the program-valid graph and via the edge (e_1, e_2) on the indexing graph.

- (1) Q(a, f) = true. The vertex f is CS-reachable from the vertex a because there exists a path from the vertex a_1 to the vertex f_2 on the indexing graph. Checking the CS-reachability on the program-valid graph, we can find a PN-path (a, b, c, d, e, f) labeled by the string, $[a, f]_1[a, f]_2[a, f]_3[a, f]_4$, that can be derived from the context-free grammar.
- (2) Q(g,c) = true. The vertex c is CS-reachable from the vertex g because there exists a path from the vertex g_1 to the vertex c_2 on the indexing graph. Checking the CS-reachability on the program-valid graph, we can find an N-path (g,c) labeled by the parenthesis, $[s_1, t_2, t_3]$ that can be derived from the context-free grammar.
- (3) $\forall v \in V \setminus \{c\} : Q(g, v) = false$. The vertex g cannot context-sensitively reach any vertex except the vertex c on the program-valid graph. For example, Q(g, f) = false because, on the program-valid graph, the path from the vertex g to the vertex f has mismatched parentheses, i.e., $[s]_2$. On the indexing graph, there is no path from the vertex f to the vertex f.

3.2 Intuition of the Correctness

Using the previous example, this section discusses the intuition of why the reduction is always correct. The discussion here serves as a warm-up construction for our formalization in the next section. Intuitively, the rationale behind our approach is that, each CS-reachable path (i.e., *P*-path, *N*-path, or *PN*-path) on the program-valid graph corresponds to a path on the indexing graph, and vice versa. Thus, we can safely transform any CS-reachability query to an equivalent query of conventional graph reachability on the indexing graph.

Specifically, in the indexing graph, the first copy of the program-valid graph removes all edges labeled by the left-parentheses so that it contains and only contains P-paths. In other words, each P-path on the program-valid graph corresponds to a path on its first copy. As illustrated in Figure 6, the P-path, (a, b, c, d, e), which is labeled by $\|1\|\|2\|\|2\|$ on the program-valid graph, corresponds to the path (a_1, b_1, d_1, e_1) on the indexing graph. Note that the sub-path (b, c, d) is replaced with the summary edge (b_1, d_1) on the indexing graph.

Similarly, in the indexing graph, the second copy of the program-valid graph removes all edges labeled by the right-parentheses so that it contains and only contains N-paths. In other words, each N-path on the program-valid graph corresponds to a path on its second copy. As illustrated in

Figure 6, the *N*-path, (e, f), which is labeled by [4] on the program-valid graph, corresponds to the path (e_2, f_2) on the indexing graph.

Other edges from the vertex v_1 to the vertex v_2 on the indexing graph connect the P-paths and the N-paths, producing the PN-paths. For instance, in Figure 6, the PN-path (a, b, c, d, e, f) on the program-valid graph can be split into two sub-paths, the P-path (a, b, c, d, e) and the N-path (e, f), which respectively correspond to the path (a_1, b_1, d_1, e_1) and the path (e_2, f_2) on the indexing graph.

4 FORMALIZATION

In this section, we formally present the idea of building the indexing graph (Section 4.1), as well as how the indexing graph enables the reduction from the CS-reachability problem to the conventional graph reachability problem (Section 4.2). In the end, we discuss an optimization of the reduction (Section 4.3).

4.1 Indexing Graph

As discussed before, to reduce CS-reachability to conventional graph reachability, we need to build the indexing graph based on the program-valid graph. As a precondition, we need to compute all summary edges for a given program-valid graph. Reps et al. [1994] have presented an algorithm to compute the summary edges and proved the following lemma about the cost of the algorithm.

LEMMA 4.1 (COMPLEXITY OF THE SUMMARY EDGES [REPS ET Al. 1994]). The number of summary edges is bounded by $O(\alpha^2|V|)$ and the time to build all summary edges is bounded by $O(\alpha|E| + \alpha^3|V|)$.

Since α , which stands for the number of function parameters and return values, is a constant in practice, we can build summary edges efficiently using almost linear time and space with respect to the graph size. We can then define the indexing graph by construction.

Definition 4.2 (Indexing Graph). Given a program-valid graph G = (V, E) where $E = E^{\epsilon} \cup E^{l} \cup E^{r}$ is the union set of edges labeled by ϵ , left-parentheses, and right-parentheses, and the set E^{s} of summary edges, an indexing graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ can be built in the following two steps:

- (1) Build two copies of the program-valid graph as well as the summary edges:
 - $G_1 = (V_1, E_1 = E_1^{\epsilon} \cup E_1^{r} \cup E_1^{s}),$
 - $G_2 = (V_2, E_2 = E_2^{\epsilon} \cup E_2^l \cup E_2^s),$

where V_i is a copy of the vertex set V, and E_i^{ϵ} , E_i^{r} , E_i^{l} , and E_i^{s} are copies of E^{ϵ} , E^{r} , E^{l} , and E^{s} over the vertex set V_i , respectively.

- (2) Build the indexing graph $G = (V, \mathcal{E})$:
 - $\mathcal{V} = V_1 \cup V_2$,
 - $\mathcal{E} = E_1 \cup E_2 \cup \{(v_1, v_2) : v_1 \in V_1, v_2 \in V_2\},$

where we use v_i to represent the copy of the vertex $v \in V$ in V_i .

The following lemma establishes the fact that the size of the indexing graph is linear to the size of the original program-valid graph in practice.

Lemma 4.3 (Complexity of the Indexing Graph). Assuming α is a constant, we have the space complexity of the indexing graph, O(|V| + |E|), and the time complexity of building the indexing graph, O(|V| + |E|).

PROOF. We analyze the complexity of the indexing graph from two aspects, the space complexity and the time complexity.

Space Complexity. For vertices, the indexing graph contains and only contains two copies of the vertices in the original program-valid graph. Thus, we have $O(|\mathcal{V}|) = O(|V|)$. For edges, we have the following equations:

$$O(|\mathcal{E}|) = O(|E_1| + |E_2| + |\{(v_1, v_2) : v_1 \in V_1, v_2 \in V_2\}|)$$
 > Definition 4.2
$$= O((|E| - |E^l| + |E^s|) + (|E| - |E^r| + |E^s|) + |V|)$$
 > Definition 4.2
$$= O(2|E| + 2|E^s| + |V|)$$
 > $E^r, E^l \subseteq E$

$$= O(2|E| + 2\alpha^2|V| + |V|)$$
 > Lemma 4.1
$$= O(|E| + |V|)$$
 > Constant α

Putting $O(|\mathcal{V}|)$ and $O(|\mathcal{E}|)$ together, we have $O(|\mathcal{V}| + |\mathcal{E}|) = O(|V| + |E|)$.

Time Complexity. Before building the vertices in \mathcal{V} and the edges in \mathcal{E} , we need to compute the summary edges, of which the time complexity is $O(\alpha|E| + \alpha^3|V|)$ according to Lemma 4.1. Thus the time complexity of building the indexing graph is the sum of $O(\alpha|E| + \alpha^3|V|)$ and $O(|\mathcal{V}| + |\mathcal{E}|)$, which is O(|V| + |E|) if α is a constant.

4.2 Query of CS-Reachability

The indexing graph allows us to answer CS-reachability queries according to the conventional reachability relations on the indexing graph. That is, given a program-valid graph G = (V, E) and its indexing graph $G = (V = V_1 \cup V_2, E)$, to determine if a vertex $v \in V$ is CS-reachable from a vertex $u \in V$ on the program-valid graph, we only need to check if the vertex $v_2 \in V_2$ is reachable from the vertex $u_1 \in V_1$ on the indexing graph.

To prove the correctness of this claim, we first define two mappings, J_1 and J_2 , as well as their inverse mappings, J_1^{-1} and J_2^{-1} . They provide mappings between the paths (or the edges) on the input program-valid graph and the edges on the indexing graph. According to Definition 4.2, we can establish the mappings as follows.

$$J_{1}(u, \dots, v) = (u_{1}, v_{1}) \qquad \text{if } (u, v) \in E^{\epsilon} \cup E^{r} \cup E^{s}$$

$$J_{1}^{-1}(u_{1}, v_{1}) = \begin{cases} (u, v) & \text{if } (u_{1}, v_{1}) \in E_{1}^{\epsilon} \cup E_{1}^{r} \\ (u, \dots, v) & \text{if } (u_{1}, v_{1}) \in E_{1}^{s} \end{cases}$$

$$J_{2}(u, \dots, v) = (u_{2}, v_{2}) \qquad \text{if } (u, v) \in E^{\epsilon} \cup E^{l} \cup E^{s}$$

$$J_{2}^{-1}(u_{2}, v_{2}) = \begin{cases} (u, v) & \text{if } (u_{2}, v_{2}) \in E_{2}^{\epsilon} \cup E_{2}^{l} \\ (u, \dots, v) & \text{if } (u_{2}, v_{2}) \in E_{2}^{s} \end{cases}$$

The mapping j_1 defined above states that, if the path (u, \ldots, v) is a single edge (u, v) labeled by ϵ or a right-parenthesis, it is copied to G_1 when building the indexing graph. If (u, \ldots, v) is a summary path, we create a corresponding summary edge (u_1, v_1) on the copy G_1 . The mapping j_1^{-1} states that, for each edge on the copy G_1 , if it is a summary edge, it can be mapped back to at least one summary path; otherwise, it is mapped back to the original edge on the program-valid graph. Similarly, we establish the other two mappings, j_2 and j_2^{-1} , for the copy G_2 .

Next, we prove the correctness of our CS-reachability query in two steps, i.e., the necessity and the sufficiency.

LEMMA 4.4 (NECESSITY). Given a program-valid graph G = (V, E) and its indexing graph $G = (V = V_1 \cup V_2, E)$, if there is a CS-reachable path from $u \in V$ to $v \in V$ on the program-valid graph, then there must exist a path from $u_1 \in V_1$ to $v_2 \in V_2$ on the indexing graph.

PROOF. According to our discussion before, a CS-reachable path on the program-valid graph may be a *P*-path, *N*-path, or *PN*-path, which are discussed below, respectively.

Case 1. The production of " $P \to MP \mid]_i P \mid \epsilon$ " states that, any P-path (u, \ldots, v) can be partitioned into multiple segments, say

$$(u,\ldots,x)(x,\ldots,y)(y,\ldots,z)\ldots(\ldots,v)$$

where each segment is either a summary path or an edge in $E^{\epsilon} \cup E^{r}$. Thus, applying the mapping j_1 to each segment will result in a path on G_1 :

$$j_1(u,\ldots,x)j_1(x,\ldots,y)j_1(y,\ldots,z)\ldots j_1(\ldots,v)=(u_1,\ldots,v_1).$$

Since $(v_1, v_2) \in \mathcal{E}$, we can find a path (u_1, \ldots, v_1, v_2) on the indexing graph.

Case 2. The production of " $N \to MN \mid [iN \mid \epsilon]$ " states that, any N-path (u, \ldots, v) can be partitioned into multiple segments, say

$$(u,\ldots,x)(x,\ldots,y)(y,\ldots,z)\ldots(\ldots,v)$$

where each segment is either a summary path or an edge in $E^{\epsilon} \cup E^{l}$. Thus, applying the mapping J_2 to each segment will result in a path on G_2 :

$$j_2(u,\ldots,x)j_2(x,\ldots,y)j_2(y,\ldots,z)\ldots j_2(\ldots,v)=(u_2,\ldots,v_2).$$

Since $(u_1, u_2) \in \mathcal{E}$, we can find a path (u_1, u_2, \dots, v_2) on the indexing graph.

Case 3. Given a PN-path, (u, \ldots, v) , according to the context-free grammar, we can split it into two segments, say $(u, \ldots, x)(x, \ldots, v)$, where the segment (u, \ldots, x) is a P-path, and the segment (x, \ldots, v) is an N-path. According to the above discussions, we can find a path (u_1, \ldots, x_1) on G_1 and a path (x_2, \ldots, x_2) on G_2 . Since $(x_1, x_2) \in \mathcal{E}$, we can find a path $(u_1, \ldots, x_1, x_2, \ldots, v_2)$ on the indexing graph.

LEMMA 4.5 (SUFFICIENCY). Given a program-valid graph G = (V, E) and its indexing graph $G = (V = V_1 \cup V_2, E)$, if there is a path from $u_1 \in V_1$ to $v_2 \in V_2$ on the indexing graph, there must exist a CS-reachable path from $u \in V$ to $v \in V$ on the program-valid graph.

PROOF. Given a path (u_1, \ldots, v_2) on the indexing graph, by definition, it must be in one of the following three forms: (1) (u_1, \ldots, v_1, v_2) , (2) (u_1, u_2, \ldots, v_2) , or (3) $(u_1, \ldots, u_1)(x_1, x_2)(x_2, \ldots, v_2)$.

Case 1. The path is in the form of (u_1, \ldots, v_1, v_2) . By definition, all vertices from u_1 to v_1 are in V_1 , and each edge (x_1, y_1) on the path is in $E_1^{\epsilon} \cup E_1^r \cup E_1^s$. Since $j_1^{-1}(x_1, y_1)$ is either a summary path (x, \ldots, y) or an edge $(x, y) \in E^{\epsilon} \cup E^r$ on the original program-valid graph, applying j_1^{-1} to each edge on the path (u_1, \ldots, v_1) results in a P-path (u, \ldots, v) on the program-valid graph.

Case 2. The path is in the form of (u_1, u_2, \ldots, v_2) . By definition, all vertices from u_2 to v_2 are in V_2 , and each edge (x_2, y_2) on the path is in $E_2^{\epsilon} \cup E_2^l \cup E_2^s$. Since $J_2^{-1}(x_2, y_2)$ is either a summary path (x, \ldots, y) or an edge $(x, y) \in E^{\epsilon} \cup E^l$ on the original program-valid graph, applying J_2^{-1} to each edge on the path (u_2, \ldots, v_2) results in a N-path (u, \ldots, v) on the program-valid graph.

Case 3. The path is in the form of $(u_1, \ldots, x_1)(x_1, x_2)(x_2, \ldots, v_2)$. Based on the discussion of Case 1 and Case 2, the prefix (u_1, \ldots, x_1) corresponds to a P-path (u, \ldots, x) on the program-valid graph; the suffix (x_2, \ldots, v_2) corresponds to an N-path (x, \ldots, v) on the program-valid graph. Thus, the concatenation of the two paths, i.e., $(u, \ldots, x, \ldots, v)$, is a PN-path on the program-valid graph. \square

Putting Lemma 4.3, Lemma 4.4, and Lemma 4.5 together, we have the following theorem that summarizes our result.

Theorem 4.6. The CS-reachability problem on a program-valid graph can be reduced to a conventional graph reachability problem on the indexing graph in linear time and space with respect to the size of the input program-valid graph.

Algorithm 2: Iterating the vertices of an indexing graph.

```
1 Procedure \operatorname{vertices}(G = (V, E^{\epsilon} \cup E^{r} \cup E^{l} \cup E^{s}))
2 | foreach v \in V do
3 | foreach i \in \{1, 2\} do
4 | do some operation on (v, i); /* iterate all v_i \in \mathcal{V}^*/
```

Algorithm 3: Iterating the successors of a given vertex on the indexing graph.

```
1 Procedure successors (G = (V, E^{\epsilon} \cup E^{r} \cup E^{l} \cup E^{s}), (v \in V, i \in \{1, 2\}))
2 | if i = 1 then
3 | do some operation on (v, 2); /^{*} \forall v \in V, (v_{1}, v_{2}) \in \mathcal{E}^{*}/
4 | foreach (v, u) \in E^{\epsilon} \cup E^{r} \cup E^{s} do
5 | do some operation on (u, 1) /^{*} (v_{1}, u_{1}) \in \mathcal{E}^{*}/
6 | else
7 | foreach (v, u) \in E^{\epsilon} \cup E^{l} \cup E^{s} do
8 | do some operation on (u, 2) /^{*} (v_{2}, u_{2}) \in \mathcal{E}^{*}/
```

4.3 Saving the Copies of the Program-Valid Graph

Instead of proposing a sophisticated CFL-reachability algorithm like many previous works, we have presented an approach that simply copies the input program-valid graph twice to build the indexing graph for addressing the CS-reachability problem. In practice, we can take a further step to make our approach simpler — we do not need to physically copy the program-valid graph for building the indexing graph.

Our key insight is that the indexing graph shares the vertices and the edges with the summary-edge-augmented program-valid graph $G = (V, E^\epsilon \cup E^r \cup E^l \cup E^s)$. Thus, we do not need to physically generate the copies, G_1 and G_2 , but reuse the data structure of G and logically distinguish the copies using an extra integer in $\{1, 2\}$. Algorithm 2 and Algorithm 3 demonstrate our idea of implementing the basic operations over the indexing graph, i.e., iterating the vertices and iterating the successors of a given vertex. The algorithms do not use the physically copied vertices $v_i \in V$ but represent the vertex as a pair $(v \in V, i \in \{1, 2\})$, which reuses the vertex v in the program-valid graph. Similarly, we can replace a physically copied edge $(v_i, u_j) \in \mathcal{E}$ with a pair ((v, i), (u, j)). Note that Algorithm 2 and Algorithm 3 are sufficient for implementing all graph algorithms over the indexing graph including the indexing algorithms discussed in Section 2.2. This is because they have essentially represented the indexing graph as an adjacent list, one primary data structure for graphs.

To conclude, in practice, the only overhead of building the indexing graph and reducing CS-reachability to conventional graph reachability is to compute the summary edges, which is a well-studied problem and can be addressed efficiently as stated in Lemma 4.1. Despite its simplicity, we show how wide its applicability is in the next section.

5 APPLICATIONS

Our results can speed up a wide range of context-sensitive data flow analyses working on different program-valid graphs. To show the practicality, we apply our results to two program analyses, i.e., Lerch et al. [2014]'s information-flow analysis and Li et al. [2013]'s alias analysis, as summarized in Table 1. The two analyses work on two different program-valid graphs, which are known as

Application	Information-Flow Analysis ¹ [Lerch et al. 2014]	Alias Analysis ² [Li et al. 2013]		
Program-Valid Graph	Exploded Super-graph	Value-Flow Graph		
Basic Approach	Reps et al. [1995, 1994]'s	Tabulation Algorithm		
Indexing Scheme	Grail [Yildirim et al. 2010]	PathTree [Jin et al. 2011]		
Index Size	O(k V)	O(k V)		
Indexing Time	O(k(V + E))	O(k E)		
Query Time	O(k) or $O(k(V + E))$	$O(1)$ or $O(\log^2 k)$		

Table 1. The basics of the two applications and the indexing schemes for them.

the *exploded super-graph* and the *value-flow graph*, respectively.⁴ Despite many differences, both applications formulate their problems as CS-reachability queries and their core engines follow the same spirit of Reps et al. [1995, 1994]'s tabulation algorithm, which, as shown in Algorithm 1, traverses the underlying program-valid graph for answering CS-reachability queries. Since each query needs a graph traversal, it is inefficient when CS-reachability is frequently queried.

To improve the query performance, we build the indexing graphs based on their input program-valid graphs and employ existing indexing schemes for acceleration. Given that there are a large number of indexing schemes we can choose as discussed in Section 2.2, we summarize the criteria of selecting indexing schemes in the rules below. Table 1 summarizes the indexing schemes chosen for the two applications.

Rule 1. If a program analysis needs to query both the CS-reachability relation and the paths between two vertices, we prefer to use the pruned-search-based indexing schemes, which can return paths when responding to CS-reachability queries.

Rule 2. If a program analysis does not need to return any paths when responding to a CS-reachability query, we prefer to use the indexing schemes that compress the transitive closure, which often exhibits better query performance than the pruned-search-based indexing schemes.

Indexing Context-Sensitive Information-Flow Analysis. Information flow is the transfer of information from a variable x to a variable y in a given program, which, in this application, is formulated as a CS-reachability problem over the exploded super-graph. When answering a CS-reachability query, this application needs to find one or multiple CS-reachable paths that lead to the information flow. This is critical when the information-flow analysis is used to detect security violations as we need to check the violation-triggering paths so as to fix the violations. Therefore, we follow Rule 1 to use the pruned-search-based indexing schemes as they can provide paths as the evidence of reachability. In the implementation, we use the Grail indexing scheme [Yildirim et al. 2010], which, as illustrated in Example 2.5, builds the reachability index by randomly traversing an input graph k times ($k \le 5$ in practice and we use k = 5 in the implementation). The Grail indexing scheme allows us to answer most unreachable queries in O(k) time and, for reachable queries, due to the pruned search, we can return a path within linear time and space with respect to the path length. Apparently, the reachability indexing scheme significantly improves the query efficiency over a normal tabulation algorithm.

 $k \le 5$ denotes how many times we randomly traverse the graph to build the index.

 $^{^{2}}$ k is the number of paths that can cover the input graph.

 $^{^4}$ We provide examples to illustrate their program-valid graphs in Appendix A and Appendix B, respectively.

Indexing Context-Sensitive Alias Analysis. Alias analysis statically determines if two pointer variables can point to the same memory location during program execution. In this application, the aliasing problem is formulated as a CS-reachability problem over the value-flow graph. Since we often only need to answer "yes" or "no" when querying if a pointer is the alias of the other, it is not necessary to find any CS-reachable path between two pointer variables. Thus, we follow Rule 2 to use the indexing schemes that compress the transitive closure. Specifically, we use the PathTree indexing scheme, which is based on a path-decomposition that partitions the input directed graph into k paths [Jin et al. 2011]. Basically, the Path-Tree method takes O(k|E|) time to compress the size of transitive closure from $O(|V|^2)$ to O(k|V|). The compressed transitive closure, a.k.a., the index, allows us to answer reachability queries in O(1) time for most cases and in $O(\log^2 k)$ time for the others. In the implementation, we use SCARAB [Jin et al. 2012], a unified reachability computation framework, to improve the performance of PathTree. Basically, SCARAB improves the performance by reducing the graph size, or more specifically, by extracting a "reachability backbone" that carries the major reachability information. Armed with the indexing scheme, the alias analysis can answer aliasing queries far more efficiently than Li et al. [2013]'s original approach, with just a moderate time and space overhead for building and storing the index.

6 EVALUATION

We implemented the context-sensitive information-flow analysis [Lerch et al. 2014] and context-sensitive alias analysis [Li et al. 2013], as well as their indexed counterparts, namely FlareIFA and FlareAA, on top of the LLVM compiler infrastructure [Lattner and Adve 2004]. Given an input program, we compile it to the LLVM bitcode and follow their original method to build the program-valid graphs, i.e., the exploded super-graph and the value-flow graph, respectively. We then follow Reps et al. [1994]'s approach to compute all summary edges and build the indexing graph according to Definition 4.2. As discussed before, we use Grail [Yildirim et al. 2010] and PathTree [Jin et al. 2011] as the indexing schemes for information-flow analysis and alias analysis, respectively. The code of Grail and PathTree is open source. We use their source code in our implementation to avoid unnecessary biases introduced by engineering issues. The whole artifact for evaluation is publicly available online.

6.1 Experiment Setup

To demonstrate how our approach improves the performance of the context-sensitive informationflow analysis and the context-sensitive alias analysis, we conducted a series of experiments over standard benchmark programs and open-source software.

Benchmark Programs. Our experiments are performed over the programs from SPEC CINT2000, a standard benchmark suite widely used in literature [Henning 2000] as well as four real-world and much larger programs: *git*, *vim*, *icu*, and *ffmpeg*. The basic information of the sixteen benchmark programs is listed in Table 2, including the lines of code (LoC) of each program as well as the number of vertices and edges on their program-valid graphs. As shown in the table, the sizes of the programs range from a few thousand lines of code to nearly one million and a program-valid graph may contain tens of millions of vertices and edges, which makes it challenging to answer reachability queries quickly, let alone CS-reachability queries.

Baseline Approaches. For each of the information-flow analysis and the alias analysis, we organize the experiments in three parts. In the first two parts, we show that it is not possible to

⁵Grail: https://github.com/zakimjz/grail; PathTree: http://www.cs.kent.edu/~nruan/soft.html.

⁶Artifact for evaluation: https://github.com/qingkaishi/context-sensitive-reachability.

Git: https://git-scm.com/; Vim: https://www.vim.org/; ICU: http://site.icu-project.org/; FFmpeg: http://ffmpeg.org/.

ID	Dungstungs	1.0	Information-F	low Analysis	Alias Analysis	
ן עו	Program	LoC	# Vertices	# Edges	# Vertices	# Edges
1	mcf	2K	33.3K	35.0K	22.2K	29.4K
2	bzip2	3K	257.3K	277.8K	59.5K	76.2K
3	gzip	6K	314.5K	332.2K	135.6K	182.8K
4	parser	8K	540.5K	566.7K	574.2K	749.1K
5	vpr	11K	3.0M	3.4M	347.7K	421.7K
6	crafty	13K	651.0K	678.5K	280.2K	362.1K
7	twolf	18K	635.4K	696.6K	468.0K	624.0K
8	eon	22K	798.8K	969.9K	766.0K	852.0K
9	gap	36K	689.8K	788.3K	3.2M	3.8M
10	vortex	49K	659.1K	714.2K	4.4M	5.6M
11	perlbmk	73K	2.4M	2.6M	9.2M	11.7M
12	gcc	135K	9.7M	10.7M	17.0M	22.2M
13	git-2.32.0	248K	5.4M	5.7M	20.6M	25.4M
14	vim-8.2.3047	386K	14.7M	19.0M	40.0M	50.0M
15	icu-69.1	594K	5.9M	6.5M	14.4M	18.2M
16	ffmpeg-3.0	940K	5.3M	6.2M	33.8M	44.6M

Table 2. Benchmark programs from SPEC CINT 2000.

compute a full transitive closure for answering the CS-reachability queries while we can compute the reachability indexes within a reasonable time and space overhead. This is because computing a transitive closure requires sub-cubic complexity for general CFL-reachability while we can often build reachability indexes within almost linear time complexity. The experiments of computing the transitive closure are run with a limit of six hours. In the third part, we show that the indexed analyses, i.e., FlareIFA and FlareAA, are much faster than their original counter-parts [Lerch et al. 2014; Li et al. 2013]. As discussed before, the original analyses follow the same spirit of Reps et al. [1995, 1994]'s tabulation algorithm (see Algorithm 1), which, essentially, traverses the input program-valid graph to answer each CS-reachability query.

We notice that there have been a few optimized CFL-reachability algorithms proposed in recent years, particularly for the standard Dyck-CFL-reachability problem [Chatterjee et al. 2017; Yuan and Eugster 2009; Zhang et al. 2013]. However, due to the differences in the underlying CFL and graph structures, their approaches cannot be directly employed in context-sensitive data flow analysis. Thus, we cannot compare our approach to them. Instead, we discuss them in Section 7.

We also notice that there are many query caching mechanisms [Zhou et al. 2018] and graph simplification algorithms, such as eliminating reachability-irrelevant vertices and edges [Li et al. 2020], which can also improve the query performance. It is noteworthy that all these optimizations are orthogonal to our approach. Thus, it does not make any sense for comparison. In practice, our approach can be used together with them for better performance. For instance, our reduction can be performed on a simplified program-valid graph, which will lead to a smaller indexing graph and, thus, faster query speed.

Environment. All experiments were run on a server with eighty "Intel Xeon CPU E5-2698 v4 @ 2.20GHz" processors and 256GB of memory running Ubuntu-16.04.

6.2 Information-Flow Analysis

The evaluation of the information-flow analysis is in three parts, which aim to show that (1) computing the transitive closure is not practical (Section 6.2.1), (2) compared to computing a transitive

ID	Tir	ne		Memory			
110	Indexing Graph	Grail	Total	Indexing Graph	Grail	Total	
1	0.36	0.03	0.39	0.25	1.52	1.78	
2	2.34	0.33	2.67	1.96	11.78	13.74	
3	3.36	0.24	3.60	2.40	14.39	16.79	
4	4.83	0.42	5.25	4.12	24.74	28.87	
5	30.39	5.25	35.64	23.20	139.18	162.38	
6	6.06	1.26	7.32	4.97	29.80	34.77	
7	6.01	2.49	8.50	4.85	29.08	33.93	
8	8.84	1.56	10.40	6.09	36.57	42.66	
9	6.63	0.78	7.41	5.26	31.57	36.84	
10	5.58	2.16	7.74	5.03	30.17	35.20	
11	21.32	6.3	27.62	18.38	110.29	128.67	
12	95.92	22.71	118.63	74.36	446.14	520.50	
13	43.95	3.9	47.85	41.46	248.76	290.22	
14	117.18	80.7	197.88	129.51	657.08	786.60	
15	45.81	2.76	48.57	45.35	272.12	317.47	
16	41.45	3.99	45.44	40.63	243.80	284.44	

Table 3. The time cost (seconds) and the memory cost (MB) of indexing the information-flow analysis.

closure, the overhead of computing the reachability index is reasonable (Section 6.2.2), and (3) the reachability index significantly speeds up CS-reachability queries (Section 6.2.3).

6.2.1 Transitive Closure is not Practical. For information-flow analysis, computing a transitive closure allows us to answer an unreachable CS-reachability query (or, in this application, an information-flow query) in constant time. For reachable cases, the transitive closure allows us to find paths from a source vertex to a target vertex within linear time and space with respect to the path size. This is because when searching a path from the source vertex, we can always prune unreachable paths based on the transitive closure. However, due to the high complexity of computing the transitive closure [Chaudhuri 2008], we failed to compute the transitive closure for ten of our sixteen benchmark programs. Thus, while computing a transitive closure is an ideal approach to the information-flow analysis, it is not practical for analyzing large-scale software.

6.2.2 Overhead of Indexing is Reasonable. The indexing procedure for the information-flow analysis includes two parts. The first part is to compute the indexing graph and the second part is to build the Grail indexes [Yildirim et al. 2010] on the indexing graph. As discussed in Section 4.3, the main cost of building an indexing graph is to compute the summary edges. Table 3 lists the time and the memory cost of computing the indexing graphs as well as the cost of computing the Grail indexes. Figure 7 shows that the cost of building an indexing graph is of linear complexity with respect to the input graph size. For the largest graph that contains tens of millions of vertices, we only need less than 2 minutes and about 80 MB of space to build the indexing graph.

Figure 8 shows the total space and time we need to build the Grail index. The space overhead is moderate, as the index size is only about 650MB for our largest program. Meanwhile, as illustrated in Figure 8, the time cost of building the index is of linear complexity with respect to the graph size. Therefore, the indexing scheme for the information-flow analysis scales quite gracefully in practice. Figure 9 shows that, in comparison to computing a transitive closure, the overhead of computing the reachability index is negligible in practice. Even for the first six small programs for which we succeed in computing the transitive closure, computing the index is 540× faster and saves 99.7% of the space compared to computing the transitive closure.

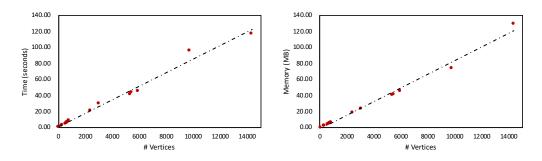


Fig. 7. Building the indexing graph exhibits linear complexity in the information-flow analysis.

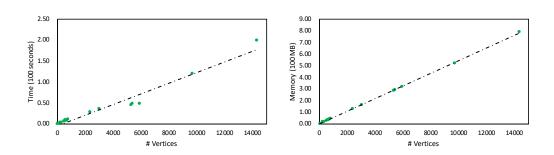


Fig. 8. Indexing the information-flow analysis exhibits linear complexity in practice.

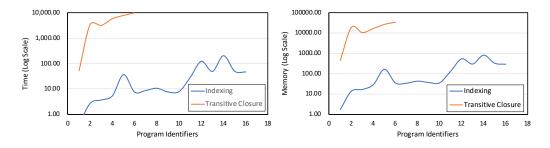


Fig. 9. Compared to computing the transitive closure, the overhead of indexing is reasonable for the information-flow analysis.

6.2.3 Indexing Enables Much Faster Queries. As listed in Table 4, the reachability index allows the indexed information-flow analysis, namely FlareIFA, to answer 10,000 reachable queries in 6,500 milliseconds and 10,000 unreachable queries in 350 milliseconds, $110\times$ to $7,642\times$ and $565\times$ to $36,541\times$ faster than the baseline approach [Lerch et al. 2014]. Readers may notice that answering a reachable query takes much longer than answering an unreachable query. This is because, for a reachable query in the information-flow analysis, we need to additionally compute a path from the source vertex to the target vertex.

Table 4. Time cost (milliseconds) of answering 10,000 reachable queries (R), 10,000 unreachable queries (R), and all queries (Total) in the information-flow analysis. We also compute the minimum (min), median (med), and maximum (max) speedup compared to the baseline approach.

ID	FLAREIFA			[Lerch et al. 2014]		
	R	$\neg R$	Total	R	$\neg R$	Total
1	22	2	23	35,505	2,103	37,608
2	150	62	212	34,000	35,159	69,159
3	214	240	454	329,274	138,917	468,191
4	352	132	485	714,638	150,179	864,818
5	2,317	320	2,637	2,755,797	983,474	3,739,271
6	838	23	861	786,158	41,560	827,718
7	290	9	299	31,943	29,668	13,611
8	2,123	181	2,304	558,344	255,466	813,810
9	1,321	114	1,435	1,499,938	269,286	1,769,224
10	1,194	101	1,296	1,729,948	351,161	2,081,109
11	3,654	205	3,859	5,712,390	193,101	5,905,491
12	6,416	315	6,731	21,600,091	11,495,662	33,095,753
13	1,012	166	1,179	1,504,814	1,439,299	2,944,113
14	2,787	50	2,837	1,824,255	1,717,196	3,541,450
15	1,898	242	2,140	14,504,613	555,593	15,060,206
16	1,532	213	1,745	1,138,623	1,200,232	2,338,855
min	110×	565×	206×			
med	1,319×	2,323×	1,379×		N/A	
max	7,642×	36,541×	7,036×			

6.3 Alias Analysis

Same as the information-flow analysis, the evaluation of the alias analysis also consists of three parts, aiming to show that (1) computing a transitive closure is not practical (Section 6.3.1), (2) the overhead of computing the reachability index is reasonable (Section 6.3.2), and (3) the reachability index significantly speeds up the CS-reachability queries (Section 6.3.3).

- 6.3.1 Transitive Closure is not Practical. Computing a transitive closure allows us to answer both reachable and unreachable CS-reachability queries (or, in this application, aliasing queries) in constant time. However, it is of sub-cubic time complexity and quadratic space complexity to compute a transitive closure [Chaudhuri 2008], which is unaffordable in practice. In the experiment, we finished the computation only for the programs with less than 40 KLoC and failed for all other larger programs. This fact shows that computing the transitive closure is not practical for alias analysis, either.
- 6.3.2 Overhead of Indexing is Reasonable. Same as the information-flow analysis, the indexing procedure for the alias analysis also includes two parts. The first part is to compute the indexing graph and the second part is to employ the PathTree indexing scheme [Jin et al. 2011] over the indexing graph. As discussed in Section 4.3, the main cost of building the indexing graph is to compute the summary edges. Table 5 lists the time and the memory cost of computing the indexing graphs as well as the cost of computing the PathTree indexes. As illustrated in Figure 10, for alias analysis, the cost of building the indexing graph is also of linear complexity with respect to the input graph size. For the largest graph that contains about forty million vertices, we only need about three minutes and 320MB of space to build the indexing graph.

ID	Time	e (seconds)		Memory (MB)			
110	Indexing Graph	PathTree	Total	Indexing Graph	PathTree	Total	
1	0.09	1.80	1.89	0.21	1.08	1.29	
2	0.33	8.79	9.12	0.56	2.91	3.47	
3	1.53	34.60	36.13	1.28	7.95	9.23	
4	8.00	105.67	113.67	5.13	28.76	33.89	
5	3.25	41.19	44.44	3.16	16.80	19.96	
6	0.84	33.55	34.39	2.36	13.26	15.62	
7	3.26	69.85	73.11	4.07	23.97	28.04	
8	1.97	83.15	85.12	8.33	38.46	46.79	
9	11.63	404.20	415.83	25.50	142.98	168.48	
10	33.85	645.69	679.54	36.00	204.34	240.34	
11	37.39	1894.22	1931.61	72.07	427.44	499.51	
12	110.80	6507.63	6618.43	141.52	925.09	1066.61	
13	130.52	3144.47	3274.99	173.92	890.78	1064.70	
14	196.54	10038.21	10234.75	320.35	1763.15	2083.50	
15	124.52	3580.17	3704.69	127.60	684.07	811.67	
16	113.56	10737.98	10851.54	297.56	1558.47	1856.03	

Table 5. The time cost and the memory cost of indexing the alias analysis.

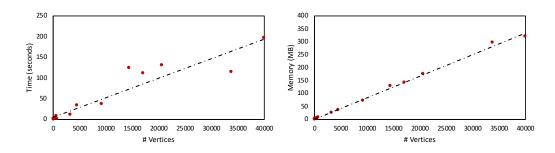


Fig. 10. Building the indexing graph exhibits linear complexity in the alias analysis.

Figure 11 shows the total time and space we need to build the index using PathTree (including the cost of building the indexing graph and the cost of computing the PathTree index). Compared to the Grail indexing scheme in the information-flow analysis, both the time cost and the memory cost are higher but still tend to be linear as shown in Figure 11. For the largest program, it takes less than 3 hours to build the index. It is noteworthy that the scalability of PathTree has been shown in previous works [Jin et al. 2011] and the PathTree indexing scheme is not our technical contribution. In practice, if an application cannot afford the overhead of PathTree, we can choose other indexing schemes as discussed in Section 2.2.

In spite of the high time cost, we show in Figure 12 that, in comparison to computing a transitive closure, the overhead of computing the reachability index is much lower and reasonable in practice. As demonstrated in Figure 12, even in a log-scale coordinate system, the curves of computing the transitive closure are much higher than those of our approach. Particularly, we cannot finish computing the transitive closure for programs with more than four million vertices while we can finish computing the PathTree index for all benchmark programs. Meanwhile, compared to the transitive closures we succeed computing, the size of our reachability index is much smaller, saving 99.1% of the space on average.

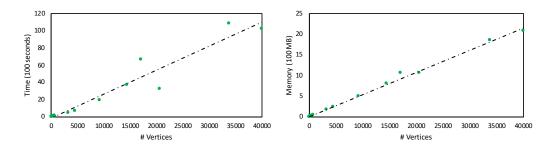


Fig. 11. Indexing the alias analysis exhibits linear complexity in practice.

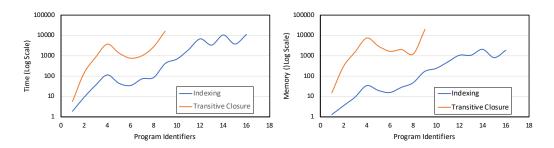


Fig. 12. Comparing to computing the transitive closure, the overhead of indexing is reasonable for the context-sensitive alias analysis.

6.3.3 Indexing Enables Much Faster Queries. As shown in Table 6, the reachability index allows the indexed alias analysis (FlareAA) to answer 10,000 reachable queries in 200 milliseconds and 10,000 unreachable queries in 30 milliseconds, 81× to 270,885× and 134× to 47,199× faster than the baseline approach [Li et al. 2013]. In total, compared to the baseline approach, the indexing scheme for the alias analysis can speed up the context-sensitive aliasing queries with a speedup from 91× to 248, 812×, with 5, 227× as the median.

In addition to the promising speedup over the conventional approach, we can observe that the query performance is very stable and works like constant time complexity —- the time cost of answering 10,000 reachable and unreachable queries is always around or less than 200 milliseconds and 30 milliseconds, respectively.⁸ This result confirms the impact of our approach, which allows us to answer CS-reachability queries in almost constant time with a moderate space overhead.

6.4 Summary

According to the evaluation results above, we can come up with two conclusions. First, the proposed reduction from CS-reachability to conventional graph reachability allows us to benefit from existing indexing schemes to achieve orders of magnitude speedup, at the cost of a reasonable overhead. As demonstrated in the evaluation, our approach can scale gracefully for large-scale programs and for a graph with tens of millions of vertices.

Second, at the same time of respecting Rule 1 and Rule 2, when choosing an indexing scheme for a program analysis, we need to consider the overhead induced by the capability of returning paths

⁸The differences in the query performance between the reachable and the unreachable cases are caused by the internal design of the PathTree indexing scheme, which is not the contribution of this paper and, thus, is omitted.

Table 6. Time cost (milliseconds) of answering 10,000 reachable queries (R), 10,000 unreachable queries (R), and all queries (Total) in the alias analysis. We also compute the minimum (min), median (med), and maximum (max) speedup compared to the baseline approach.

ID	FLAREAA			[Li et al. 2013]		
ш	R	$\neg R$	Total	R	$\neg R$	Total
1	80	18	99	6,532	2,454	8,985
2	102	6	108	83,012	21,783	104,795
3	114	12	127	319,752	57,450	377,202
4	124	8	133	690,446	52,262	742,708
5	108	8	116	243,917	38,684	282,601
6	123	8	131	160,285	27,328	187,613
7	157	7	164	244,014	20,068	264,082
8	149	8	157	289,450	34,759	324,209
9	172	11	183	834,999	50,195	885,194
10	155	10	165	3,555,830	55,351	3,611,181
11	191	8	198	4,809,880	52,284	4,862,164
12	222	12	234	14,419,500	122,908	14,542,408
13	196	11	207	7,789,460	225,227	8,014,687
14	200	10	210	3,194,340	402,685	3,597,025
15	195	15	210	16,580,400	402,119	16,982,519
16	216	24	239	58,455,400	1,115,120	59,570,520
min	81×	134×	91×			
med	5,213×	5,328×	5,227×		N/A	
max	270,885×	47,199×	248,812×			

for each reachable query. As demonstrated in the evaluation, such an overhead could make the query slower but still considerably faster than the baseline approach.

7 RELATED WORK

In this section, we discuss two strands of related work. Section 7.1 discusses the language-reachability problems in program analysis, particularly for context-sensitive analysis and pointer analysis. Section 7.2 introduces some advanced reachability indexing schemes with the potential of optimizing program analyses.

7.1 Language Reachability for Program Analysis

Many program analysis problems can be formulated as CFL-reachability problems. Particularly, Dyck-CFL lays the basis for "almost all of the applications of CFL reachability in program analysis" [Kodumal and Aiken 2004]. Many studies on solving all-pairs CFL or Dyck-CFL reachability have been conducted in various contexts including recursive state machines [Alur et al. 2005], visibly push-down languages [Alur and Madhusudan 2004], and streaming XML [Alur 2007]. They usually work with a dynamic programming algorithm, which can be regarded as a generalization of the CYK algorithm for CFL-recognition [Younger 1967] and is of cubic time complexity [Kodumal and Aiken 2004; Reps et al. 1995; Yannakakis 1990]. Melski and Reps [2000] and Kodumal and Aiken [2004] studied the relationship between CFL reachability and set constraint, but their algorithms did not break through the cubic bottleneck. Chaudhuri [2008] and Zhang et al. [2014] showed that the well-known Four Russians' Trick could be employed to achieve sub-cubic algorithms. Different from the above studies that were conducted in a general setting, we focus on an extended Dyck-CFL reachability problem for context-sensitive program analysis. In this setting, we can easily reduce

the CS-reachability problem on a program-valid graph to a conventional reachability problem on the indexing graph, which allows us to benefit from various indexing schemes from the field of graph databases. In what follows, we discuss two main applications of language reachability.

7.1.1 Context-Sensitive Analysis. Tang et al. [2015] employed the tree-adjoining-language (TAL) reachability problem to formulate the context-sensitive data-dependence analysis in the presence of callbacks. However, their algorithms are of $O(|V|^6)$ time complexity and, thus, are not scalable in practice. Chatterjee et al. [2017] solved the problem by utilizing the "constant tree-width" feature of a local data-dependence graph. However, their algorithm can only answer a reachability query between vertices in the same calling context. Our work is different from theirs as we reduce the CS-reachability problem to the conventional reachability problem.

Zhang and Su [2017] formulated the context-sensitive and field-sensitive data-dependence analysis as a linear-conjunctive-language (LCL) reachability problem. Compared to the CFL-reachability formulation used in this paper, LCL-reachability provides a more precise model due to the field sensitivity. However, the exact LCL-reachability problem is known to be undecidable. Thus, only approximation algorithms can be provided [Reps 2000]. Li et al. [2021] further refined the results of context-sensitive and field-sensitive data-dependence analysis by formulating it as an interleaved Dyck-CFL reachability problem over a bidirected graph and proved that the interleaved Dyck-CFL reachability problem with more than two parenthesis pairs is NP-hard. To improve the algorithm efficiency in practice, Li et al. [2020] proposed an approach to reducing the graph size before conducting the interleaved Dyck-CFL reachability analysis.

Pointer Analysis. The other common use of CFL-reachability in program analysis is to 7.1.2 resolve pointer relations on a bidirected graph, where each edge (u, v) labeled by a left-parenthesis corresponds to an inverse edge (v, u) labeled by a right-parenthesis. When the underlying language is the Dyck language, Zhang et al. [2013] and Chatterjee et al. [2017] have demonstrated that a transitive closure can be computed in almost linear time. However, it is much harder in a general setting. Thus, many techniques have been proposed to optimize the CFL-based pointer analysis. Zheng and Rugina [2008] proposed a demand-driven method to resolve pointer relations without computing the transitive closure. Xu et al. [2009] optimized the CFL-based pointer analysis by computing must-not-alias information for all pairs of variables, which then can be used to quickly filter out infeasible paths during the more precise pointer analysis. Zhang et al. [2014] optimized the CFL-based pointer analysis by selectively propagating reachability information, which allows us to bypass a large portion of edges. Dietrich et al. [2015] proposed a novel transitive-closure data structure with a pre-computed set of potentially matching load/store pairs to accelerate the fix-point calculation. Wang et al. [2017] proposed a graph system that allows CFL-based pointer analysis to work in a single machine by utilizing the disk space. Our work is different from theirs because we do not focus on bidirected graphs and the underlying context-free language is different.

7.2 Indexing Schemes and Their Potential Use in Program Analysis

Our reduction from the CS-reachability problem to the conventional reachability problem can be utilized together with other advanced graph database techniques, thereby enabling more program analysis applications. We briefly discuss some of them below.

7.2.1 Indexing for Dynamic Graphs. Beyond the indexing schemes for conventional graph reachability discussed in Section 2, recent studies also consider additional constraints when evaluating reachability queries. Some of these indexing schemes can be used directly to optimize program analyses. Roditty and Zwick [2008], Bouros et al. [2009], and Zhu et al. [2014] proposed improved

reachability algorithms to handle dynamic graphs, where edges and vertices may be dynamically created, updated, or deleted.

These indexing schemes, which we refer to as incremental indexing schemes, can be applied to incremental program analysis where program variables or dependence relations are changed during software evolution. When software evolves, we continuously revise the indexing graph proposed in the paper. An incremental indexing scheme can quickly capture the graph changes and rebuild the indexes for answering CS-reachability queries.

7.2.2 Indexing for Label-Constraint Reachability. Jin et al. [2010] proposed the problem of label-constraint reachability (LCR), in which a vertex v is reachable from a vertex u if and only if there exists a path from the vertex u to the vertex v and the set of the edge labels on the path is a subset of a given label set. This problem has been extensively studied [Valstar et al. 2017; Zou et al. 2014]. Recently, Peng et al. [2020] proposed an indexing scheme to answer billion-scale LCR queries. Hassan et al. [2016] and Rice and Tsotras [2010] proposed approaches to finding the shortest path for LCR problems.

Many program analyses can be modeled as an LCR problem. For instance, in a taint analysis, by modeling the sanitizing operations as edge labels, we can employ the LCR indexing schemes to check if the tainted data are propagated to a destination with proper sanitizations. Using the indexing graph proposed in the paper, we can label its edges with sanitization labels and use an LCR indexing scheme to enable an efficient context-sensitive taint analysis.

7.2.3 Benefiting from Other Graph Database Techniques. Since the indexing graph is a common directed graph, we can profit from a lot of existing graph database techniques, such as query caching techniques and graph simplification algorithms, to accelerate reachability queries on the indexing graph. The query caching techniques, such as C-Graph [Zhou et al. 2018], explore the data locality to speed up graph processing tasks. All modern graph databases, such as Neo4j and HyperGraphDB, implement such caching mechanisms for acceleration. These approaches are orthogonal to our idea and can be used together with our approach for better performance.

The graph simplification techniques aim to reduce the graph size so as to accelerate reachability queries. First, we can follow existing approaches to discover and merge vertices with equivalent reachability relations, e.g., vertices in a strongly connected component or vertices with the same successors and predecessors, thereby reducing the graph size [Fan et al. 2012; Zhou et al. 2017]. Second, we can employ existing advances to perform transitive reduction, which aims to remove unnecessary edges with respect to reachability queries [Aho et al. 1972; Habib et al. 1993; Simon 1988; Valdes et al. 1982; Williams 2012; Zhou et al. 2017].

8 CONCLUSION

We have presented a reduction from the problem of context-sensitive reachability to the problem of conventional graph reachability. This reduction allows us to efficiently answer context-sensitive reachability queries using the indexing schemes for conventional graph reachability. We apply our approach to speeding up two context-sensitive data flow analyses and compare them with the state-of-the-art approaches. The evaluation results demonstrate that we can achieve orders of magnitude speedup for answering a query, at the cost of only a moderate overhead to build and store the indexes. Since reducing the complexity of CFL-reachability is theoretically very hard in a general setting, providing proper indexing schemes could be a promising solution for reducing the cost of CFL-reachability queries.

⁹Neo4j: https://neo4j.com; HyperGraphDB: http://www.hypergraphdb.org.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and Peisen Yao for their valuable feedback on our earlier paper draft. This work was supported in part by the Ant Research Program from Ant Group and the RGC16206517 and ITS/440/18FP grants from the Hong Kong Research Grant Council.

REFERENCES

- Alfred Aho, Michael Garey, and Jeffrey Ullman. 1972. The transitive reduction of a directed graph. SIAM J. Comput. 1, 2 (1972), 131–137. https://doi.org/10.1137/0201008
- Rajeev Alur. 2007. Marrying words and trees. In Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '07). ACM, 233–242. https://doi.org/10.1145/1265530.1265564
- Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. 2005. Analysis of recursive state machines. ACM Transactions on Programming Languages and Systems (TOPLAS) 27, 4 (2005), 786–818. https://doi.org/10.1145/1075382.1075387
- Rajeev Alur and Parthasarathy Madhusudan. 2004. Visibly pushdown languages. In *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC '04)*. ACM, 202–211. https://doi.org/10.1145/1007352.1007390
- 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. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 259–269. https://doi.org/10.1145/2594291.2594299
- Panagiotis Bouros, Spiros Skiadopoulos, Theodore Dalamagas, Dimitris Sacharidis, and Timos Sellis. 2009. Evaluating reachability queries over path collections. In *Proceedings of the 21st International Conference on Scientific and Statistical Database Management (SSDBM '09)*. Springer, 398–416. https://doi.org/10.1007/978-3-642-02279-1_29
- Cheng Cai, Qirun Zhang, Zhiqiang Zuo, Khanh Nguyen, Guoqing Xu, and Zhendong Su. 2018. Calling-to-reference context translation via constraint-guided CFL-reachability. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, 196–210. https://doi.org/10.1145/3192366.3192378
- 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), 30:1–30:30. https://doi.org/10. 1145/3158118
- Swarat Chaudhuri. 2008. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, 159–169. https://doi.org/10.1145/1328438.1328460
- Li Chen, Amarnath Gupta, and M. Erdem Kurul. 2005. Stack-based algorithms for pattern matching on dags. In *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB Endowment, 493–504. https://doi.org/10.14778/2180912.2180919
- Yangjun Chen and Yibin Chen. 2008. An efficient algorithm for answering graph reachability queries. In *Proceedings of the 24nd International Conference on Data Engineering (ICDE '08)*. IEEE, 893–902. https://doi.org/10.1109/ICDE.2008.4497498
- James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. 2013. TF-Label: A topological-folding labeling scheme for reachability querying in a large graph. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13). ACM, 193–204. https://doi.org/10.1145/2463676.2465286
- Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, 480–491. https://doi.org/10.1145/1250734.1250789
- Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. SIAM J. Comput. 32, 5 (2003), 1338–1355. https://doi.org/10.1137/S0097539702403098
- Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. 2015. Giga-scale exhaustive points-to analysis for java in under a minute. In Proceedings of the 2015 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '15). ACM, 535–551. https://doi.org/10.1145/2814270.2814307
- Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. 2012. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, 157–168. https://doi.org/10.1145/2213836.2213855
- Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS) 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041
- Michel Habib, Michel Morvan, and J-X Rampon. 1993. On the calculation of transitive reduction-closure of orders. *Discrete Mathematics* 111, 1-3 (1993), 289–303. https://doi.org/10.1016/0012-365X(93)90164-O
- Mohamed Hassan, Walid Aref, and Ahmed Aly. 2016. Graph indexing for shortest-path finding over dynamic sub-graphs. In *Proceedings of the 2016 ACM International Conference on Management of Data (SIGMOD '16)*. ACM, 1183–1197. https://doi.org/10.1145/2882903.2882933

- John L. Henning. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer* 33, 7 (2000), 28–35. https://doi.org/10.1109/2.869367
- Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. 2010. Computing label-constraint reachability in graph databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, 123–134. https://doi.org/10.1145/1807167.1807183
- Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Yu Xu. 2012. SCARAB: Scaling reachability computation on large graphs. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12). ACM, 169–180. https://doi.org/10.1145/2213836.2213856
- Ruoming Jin, Ning Ruan, Yang Xiang, and Haixun Wang. 2011. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Transactions on Database Systems (TODS)* 36, 1 (2011), 7:1–7:44. https://doi.org/10.1145/1929934.1929941
- Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3-hop: A high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, 813–826. https://doi.org/10.1145/1559845.1559930
- John Kodumal and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. In *Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, 207–218. https://doi.org/10.1145/996841.996867
- Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO '04)*. IEEE, 75:1–75:12. https://doi.org/10.1109/CGO.2004.1281665
- Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. 2014. FlowTwist: Efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '14)*. ACM, 98–108. https://doi.org/10.1145/2635868.2635878
- Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 13th European Software Engineering Conference Held Jointly with the 19th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*. ACM, 343–353. https://doi.org/10.1145/2025113.2025160
- Lian Li, Cristina Cifuentes, and Nathan Keynes. 2013. Precise and scalable context-sensitive pointer analysis via value flow graph. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, 85–96. https://doi.org/10.1145/2491894.2466483
- 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 (PLDI '20)*. ACM, 780–793. https://doi.org/10.1145/3385412.3386021
- Yuanbo Li, Qirun Zhang, and Thomas Reps. 2021. On the complexity of bidirected interleaved Dyck-reachability. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434340
- David Melski and Thomas Reps. 2000. Interconvertibility of a class of set constraints and context-free-language reachability. Theoretical Computer Science 248, 1-2 (2000), 29–98. https://doi.org/10.1016/S0304-3975(00)00049-9
- Ana Milanova. 2020. FlowCFL: generalized type-based reachability analysis: graph reduction and equivalence of CFL-based and type-based reachability. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29. https://doi.org/10.1145/3428246
- You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2020. Answering billion-scale label-constrained reachability queries within microsecond. *Proceedings of the VLDB Endowment* 13, 6 (2020), 812–825. https://doi.org/10.14778/3380750. 3380753
- Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. Existential label flow inference via CFL reachability. In *Proceedings of the 13th International Static Analysis Symposium (SAS '06)*. Springer, 88–106. https://doi.org/10.1007/11823230_7
- Jakob Rehof and Manuel Fähndrich. 2001. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01). ACM, 54-66. https://doi.org/10.1145/360204.360208
- Thomas Reps. 1995. Shape analysis as a generalized path problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '95)*. ACM, 1–11. https://doi.org/10.1145/215465. 215466
- Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. ACM Transactions on Programming Languages and Systems (TOPLAS) 22, 1 (2000), 162–186. https://doi.org/10.1145/345099.345137
- Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, 49–61. https://doi.org/10.1145/199448.199462

- Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding up slicing. In *Proceedings of the 2nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '94)*. ACM, 11–20. https://doi.org/10.1145/193173.195287
- Michael Rice and Vassilis Tsotras. 2010. Graph indexing of road networks for shortest path queries with label restrictions. Proceedings of the VLDB Endowment 4, 2 (2010), 69–80. https://doi.org/10.14778/1921071.1921074
- Liam Roditty and Uri Zwick. 2008. Improved dynamic reachability algorithms for directed graphs. SIAM J. Comput. 37, 5 (2008), 1455–1471. https://doi.org/10.1137/060650271
- Stephan Seufert, Avishek Anand, Srikanta Bedathur, and Gerhard Weikum. 2013. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *Proceedings of the 29nd International Conference on Data Engineering (ICDE '13)*. IEEE, 1009–1020. https://doi.org/10.1109/ICDE.2013.6544893
- Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO '12)*. ACM, 264–274. https://doi.org/10. 1145/2259016.2259050
- Klaus Simon. 1988. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science* 58, 1-3 (1988), 325–346. https://doi.org/10.1007/3-540-16761-7_87
- Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06). ACM, 387–400. https://doi.org/10.1145/1133981.1134027
- Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, 59–76. https://doi.org/10.1145/1094811.1094817
- Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122. https://doi.org/10.1109/TSE.2014.2302311
- Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, 83–95. https://doi.org/10.1145/2676726.2676997
- Jacobo Valdes, Robert Tarjan, and Eugene Lawler. 1982. The recognition of series parallel digraphs. SIAM J. Comput. 11, 2 (1982), 298–313. https://doi.org/10.1137/0211023
- Lucien DJ Valstar, George HL Fletcher, and Yuichi Yoshida. 2017. Landmark indexing for evaluation of label-constrained reachability queries. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '11)*. ACM, 345–358. https://doi.org/10.1145/3035918.3035955
- Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. 2006. Dual labeling: Answering graph reachability queries in constant time. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. IEEE, 75–75. https://doi.org/10.1109/ICDE.2006.53
- 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. In Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17). ACM, 389–404. https://doi.org/10.1145/3037697.3037744
- Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2014. Reachability querying: An independent permutation labeling approach. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1191–1202. https://doi.org/10.14778/2732977.2732992
- Virginia Vassilevska Williams. 2012. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the 44th ACM Symposium on Theory of Computing (STOC '12)*. ACM, 887–898. https://doi.org/10.1145/2213977.2214056
- Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP '09)*. Springer, 98–122. https://doi.org/10.1007/978-3-642-03013-0_6
- Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings* of the 20th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '11). ACM, 155–165. https://doi.org/10.1145/2001420.2001440
- Mihalis Yannakakis. 1990. Graph-theoretic methods in database theory. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '90)*. ACM, 230–242. https://doi.org/10.1145/298514.298576
- Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. 2010. Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 276–284. https://doi.org/10.14778/1920841.1920879
- Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n3. *Information and Control* 10, 2 (1967), 189–208. https://doi.org/10.1016/S0019-9958(67)80007-X
- Hao Yuan and Patrick Eugster. 2009. An efficient algorithm for solving the dyck-cfl reachability problem on trees. In *Proceedings of the 18th European Symposium on Programming (ESOP '09)*. Springer, 175–189. https://doi.org/10.1007/978-3-642-00590-9_13

Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 435–446. https://doi.org/10.1145/2499370.2462159

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 (POPL '17)*. ACM, 344–358. https://doi.org/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 SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '14). ACM, 829–845. https://doi.org/10.1145/2660193.2660213

Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08). ACM, 197–208. https://doi.org/10.1145/1328438.1328464 Junfeng Zhou, Shijie Zhou, Jeffrey Xu Yu, Hao Wei, Ziyang Chen, and Xian Tang. 2017. DAG reduction: Fast answering reachability queries. In Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17). ACM, 375–390. https://doi.org/10.1145/3035918.3035927

Li Zhou, Ren Chen, Yinglong Xia, and Radu Teodorescu. 2018. C-Graph: A highly efficient concurrent graph reachability query framework. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP '18)*. ACM, 79:1–79:10. https://doi.org/10.1145/3225058.3225136

Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: A total order approach. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14). ACM, 1323–1334. https://doi.org/10.1145/2588555.2612181

Lei Zou, Kun Xu, Jeffrey Xu Yu, Lei Chen, Yanghua Xiao, and Dongyan Zhao. 2014. Efficient processing of label-constraint reachability queries in large graphs. *Information Systems* 40, MAR (2014), 47–66. https://doi.org/10.1016/j.is.2013.10.003

A EXAMPLE OF THE EXPLODED SUPER-GRAPH

The IFDS framework solves a wide range of data flow problems with distributive flow functions over finite domains [Reps et al. 1995]. These problems can be reduced to a graph reachability problem over the exploded super-graph. Figure 13 demonstrates an example of the exploded supergraph used in the information-flow analysis [Lerch et al. 2014]. A vertex on the exploded-super graph stands for a data flow fact holding at a program point. For instance, the vertex (a, 2) means that the data flow fact, i.e., the variable a includes some sensitive information, holds at Line 2. An edge on the exploded-super graph represents the propagation of data

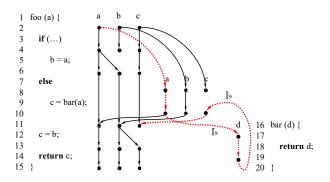


Fig. 13. The program-valid graph, a.k.a., the exploded-super graph, used in the information-flow analysis [Lerch et al. 2014]. The dotted and red path is a CS-reachable path and represents a valid information flow.

flow facts via a statement. As an example, the assignment b = a will generate a data flow fact that the variable b receives some information, assuming the variable a holds the information before. Thus, we have an edge from the vertex (a, 4) to the vertex (b, 6). At the same time, the data flow fact that the variable a holds the information should be kept. Thus, we have an edge from the vertex (a, 4) to the vertex (a, 6). At a call site, we can build the graph edges, namely call edges and return edges, as we reach a series of assignments that assign the actual parameters to the formal parameters and assign the return values to the return-value receivers. Additionally, we need to add the parentheses, [a] and [a], on the call edges and the return edges to label the call site at Line a. In the example, the red and dotted path from the vertex (a, 2) to the vertex (a, 1) is a CS-reachable

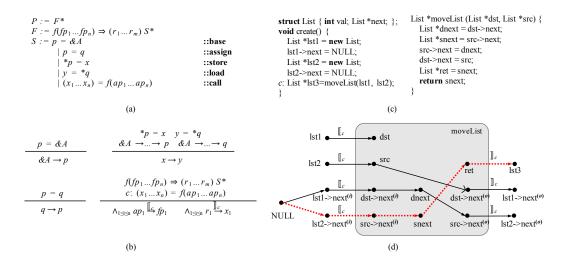


Fig. 14. The program-valid graph, a.k.a., the value-flow graph, used in the alias analysis [Li et al. 2013]. The dotted and red path is a CS-reachable path and all pointers on the path are aliases.

path as the string of the edge labels only includes a pair of matched parentheses, thus representing a valid information flow.

B EXAMPLE OF THE VALUE-FLOW GRAPH

Li et al. [2013]'s context-sensitive alias analysis is formulated over the small languages shown in Figure 14(a). Most statements in the language are standard except that every function is assumed to be a pure function (i.e., without any side-effects) and allows multiple formal parameters fp_i and multiple returns r_i . A function call is represented as $(x_1 ldots x_n) = f(ap_1 ldots ap_n)$, where x_i are return value-receivers and ap_i are actual parameters.

The value-flow graph is built based on the rules in Figure 14(b). For any "base" statement p=&A and "assign" statement p=q, we add an edge from &A and q to p, respectively. For a pair of store and load assignments, *p=x and y=*q, we add an edge from x to y if and only if p and q have the same ancestor, which implies that p and q are pointer aliases. For a call statement with a label c, we build edges by regarding it as a series of assignments that assign the formal parameters to the actual parameters and assign the return values to their receivers. Additionally, we respectively add the parentheses $[\![c]$ and $[\![c]$ on the call edges and the return edges to distinguish different call sites.

Figure 14(d) illustrates the value-flow graph for the C/C++ code snippet in Figure 14(c). To respect the language in Figure 14(a), we need to purify the function, moveList, by introducing two extra formal parameters to represent dst->next and src->next, which are referenced in the function. Also, we need to introduce two extra return values to represent dst->next and src->next as they are also modified in the function. These extra formal parameters and return values are represented as the vertices labeled by (i) and (o) in the value-flow graph. For instance, the vertex src->next is tands for the value stored in the memory location src->next before the function call. With the value-flow graph, we can formulate the context-sensitive alias analysis as a CS-reachability problem. For instance, the dotted and red path is a CS-reachable path and all pointers on the path are aliases.