Efficient Subcubic Alias Analysis for C

Qirun Zhang[†] Xiao Xiao[†] Charles Zhang[†] Hao Yuan[‡] Zhendong Su[§]

[†]The Hong Kong University of Science and Technology {qrzhang, richardxx, charlesz}@cse.ust.hk

[‡]BOPU Technologies hao@bopufund.com

[§]University of California, Davis su@cs.ucdavis.edu

Abstract

Inclusion-based alias analysis for C can be formulated as a context-free language (CFL) reachability problem. It is well known that the traditional cubic CFL-reachability algorithm does not scale well in practice. We present a highly scalable and efficient CFL-reachability-based alias analysis for C. The key novelty of our algorithm is to propagate reachability information along only original graph edges and bypass a large portion of summary edges, while the traditional CFL-reachability algorithm propagates along all summary edges. We also utilize the Four Russians' Trick — a key enabling technique in the subcubic CFL-reachability algorithm — in our alias analysis. We have implemented our subcubic alias analysis and conducted extensive experiments on widely-used C programs from the pointer analysis literature. The results demonstrate that our alias analysis scales extremely well in practice. In particular, it can analyze the recent Linux kernel (which consists of 10M SLOC) in about 30 seconds.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers; F.3.2 [*Semantics of Programming Languages*]: Program Analysis

General Terms Algorithms, Languages, Experimentation

Keywords Alias analysis, CFL-reachability

1. Introduction

Programs written in C and Java make extensive use of pointers. Determining pointer aliasing is one of the fundamental static analysis problems [24, 29]. Given two pointers, the general approach to alias analysis is to check whether their points-to sets intersect [18]. In practice, many client applications query only alias information, and the general points-to-based approach is inefficient for answering alias queries. For example, generating the data dependencies for a data-race detector [26] with points-to information is found to be over 100 times slower than using alias information [48]. Although alias information can be computed by performing matrix multiplication over suitable representations of points-to and pointed-by information, the computation is typically both time- and memory-consuming [48]. Thus, it is desirable to have an efficient alias analysis that directly computes alias information.

Alias analysis can be carried out either exhaustively or in a demand-driven fashion. Almost all state-of-the-art alias analyses are demand-driven [37, 38, 40, 50, 54] and have been formulated as a context-free language (CFL) reachability problem [32, 51]. The CFL-reachability-based alias analysis achieves good precision and can be naturally extended to model field sensitivity. For instance, Zheng and Rugina [54] developed a demand-driven alias analysis for C, with precision equivalent to an inclusion-based (*i.e.*, Andersen-style) points-to analysis [3]. As for Java, Yan *et al.* [50] designed an efficient context- and field-sensitive alias analysis, followed by Shang *et al.*'s improved algorithm using dynamic method summaries [37]. Demand-driven alias analysis computes solutions *w.r.t.* given queries. However, for CFL-reachability-based approaches, computing reachability between *two* nodes may resort to a graph traversal among *all* nodes in the worst case [32, 52]. Consequently, in practice, the demand-driven approach can take a long time on specific queries. For instance, using 10ms time budget for each query, the alias analysis developed by Yan *et al.* takes 834 seconds to complete on even the smallest subject compress [50].

Exhaustive alias analysis, on the other hand, computes *all-pairs* alias information and processes each query in constant time [9, 17, 39, 49, 52]. In many cases, exhaustive alias analysis is complementary to demand-driven analysis. For example, it becomes quite useful when a client application needs to track the global alias information and find all aliased pointers [41, 44]. The *all-pairs* information is a prerequisite for persistent pointer information [48], and can also be used to discard the must-not-alias pointers [49]. Moreover, one of the key techniques in the state-of-the-art demand-driven alias analyses is using *all-pairs* alias summaries for each procedure [37, 50].

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-2585-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2660193.2660213

However, computing *all-pairs* CFL-reachability (*i.e.*, exhaustive analysis) is considerably more expensive than its *single-source-single-sink* counterpart (*i.e.*, demand-driven analysis). There has been much less development of CFL-reachability-based alias analysis targeting *all-pairs* reachability [49, 52]. The current practice of solving the *all-pairs* CFL-reachability problem, resorting to the popular dynamic programming style algorithm [34, 51], takes $O(n^3)$ time, where n denotes the number of nodes (*i.e.*, pointers). Consequently, straightforward implementations are ill-suited for handling real-world, large applications. Although Chaudhuri [8] proposed a subcubic *all-pairs* CFL-reachability algorithm, its potential benefits in practice have not been reported.

In this paper, we present an efficient, exhaustive alias analysis for C that solves *all-pairs* CFL-reachability on program expression graphs (PEGs) [54]. The main novelty of our algorithm is to compute CFL-reachability based on the original PEG edges and existing summary edges representing only memory aliases, while the traditional CFL-reachability algorithm computes all summary edges *w.r.t.* the given grammar [32, 51]. We also utilize the Four Russians' Trick [4] — a key enabling technique in Chaudhuri's subcubic CFL-reachability algorithm [8] — in our analysis. We have implemented our alias analysis and conducted extensive evaluations on a set of C programs with size ranging from 200K to 10M SLOC. In particular, we compare our algorithm with the traditional cubic CFL-reachability algorithm and Chaudhuri's subcubic algorithm, respectively. The results demonstrate that our alias analysis performs extremely well on *recent stable* releases of these C programs.

In summary, we make the following main contributions:

- We develop an efficient subcubic alias analysis for C. Our analysis algorithm solves *all-pairs* CFL-reachability on PEGs [54]. The experimental results show that our algorithm is three orders faster than the traditional CFL-reachability algorithm [32, 51], and five times faster than Chaudhuri's subcubic algorithm [8].
- We conduct large-scale experiments on *recent stable* releases of popular C programs used in the pointer analysis literature. To the best of our knowledge, this is the first study demonstrating the benefits of subcubic *all-pairs* alias analyses. Particularly, our alias analysis is able to analyze the 10M SLOC Linux kernel in about 30 seconds.

The rest of the paper is structured as follows. Section 2 reviews background materials on CFL-reachability-based alias analysis. Section 3 presents our subcubic alias analysis algorithm. Section 4 describes our evaluation comparing the traditional cubic CFL-reachability algorithm, Chaudhuri's subcubic algorithm and our new algorithm, using the client alias analysis. Section 5 surveys related work, and Section 6 concludes.

$$
\begin{array}{lcl}
a \in \mathit{Addresses} & ::= & a_{var} \mid a_{heap} \\
e \in \mathit{Expressions} & ::= & *e \mid a \\
s \in \mathit{Statements} & ::= & *e_1 := e_2
\end{array}
$$

Figure 1. Core syntax of C pointers.

2. Preliminaries

In this section, we review alias analysis and CFL-reachability.

2.1 CFL-Reachability

Context-free language (CFL) reachability [32, 51] is an extension to standard graph reachability. Let $CFG = (\Sigma, N, P, S)$ be a context-free grammar with an alphabet Σ, nonterminal symbols N, production rules P and start symbol S. Given a context-free grammar $CFG = (\Sigma, N, P, S)$ and a directed graph $G = (V, E)$ with each edge $(u, v) \in E$ labeled by a terminal $\mathcal{L}(u, v)$ from the alphabet Σ or ε, each path $p = v_0, v_1, v_2, \ldots, v_m$ in G *realizes* a string $R(p)$ over the alphabet by concatenating the edge labels in the path in order, *i.e.*, $R(p) = \mathcal{L}(v_0, v_1)\mathcal{L}(v_1, v_2)\ldots\mathcal{L}(v_{m-1}, v_m)$. Let X be a nonterminal, we define X-paths as follows:

Definition 1 (X-path). *A path $p = u, \ldots, v$ in G is an X-path if the* realized *string $R(p)$ can be derived from the nonterminal symbol $X \in N$, represented as a summary edge (u, X, v).*

Since an X-path also represents a summary edge, we use the terms X-path and X-edge interchangeably.

2.2 Pointer Expression Graphs

The input to our algorithm is a bidirected graph, known as a *Pointer Expression Graph* (PEG) [54]. A PEG represents the given C program in a canonical form that consists of pointer assignments. The pointer analysis based on PEGs is flow-insensitive, therefore, control flow between pointer assignments is irrelevant. PEGs model the core C-style pointer language shown in Figure 1. PEGs also handle additional C language features (*e.g.*, arrays, structures, and pointer arithmetics), as detailed by Zheng and Rugina [54, Section 6.1].

There are three basic ingredients in the core language in Figure 1: memory addresses $a \in \mathit{Addresses}$, pointer expressions $e \in \mathit{Expressions}$ and pointer statements $s \in \mathit{Statements}$. Memory addresses model the symbolic addresses of variables, and can be obtained via either the address-of operator (*e.g.*, `&x`) or memory allocation (*e.g.*, `malloc()`), denoted as a_{var} and a_{heap}, respectively. Pointer expressions model the behavior of the indirection operator (*e.g.*, `*x`) in C. Pointer variables are allowed by arbitrary pointer dereferences. Finally, pointer statements model program statements that manipulate pointers.

A PEG $G = (V, E)$ is a graph representation that depicts the canonical form of all pointer statements from the input C program. In a PEG, each node $v \in V$ represents a pointer expression e. A PEG also contains two kinds of edges:

$$M \quad ::= \quad \bar{d}\; V\; d \qquad \text{(ZR1)}$$
$$V \quad ::= \quad (M?\; \bar{a})^{*}\; M?\; (a\; M?)^{*} \qquad \text{(ZR2)}$$

Figure 2. The CFL-reachability formulation for C alias analysis.

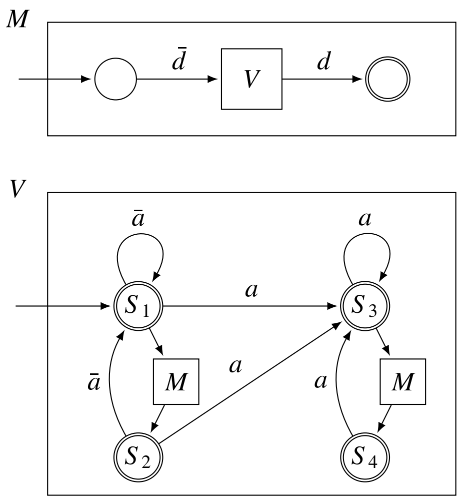


Figure 3. The recursive state machines.

- *Pointer dereference edges ($\mathcal{D}$-edges)*: For each pointer deference $*e$, there is a directed edge from e to $*e$ labeled by d. Let the nodes representing e and $*e$ be u and v, respectively. We denote such labeled edges as $(u, d, v) \in E$.
- *Pointer assignment edges ($\mathcal{A}$-edges)*: For each assignment statement $*e_1 := e_2$, there is a directed edge from e_2 (as node u) to $*e_1$ (as node v) labeled by a. We denote it as $(u, a, v) \in E$.

For example, the *top-level* pointer variables are represented as the nodes without outgoing $\mathcal{D}$-edges. Moreover, the *address-taken* variables are represented as the nodes without incoming $\mathcal{D}$- or $\mathcal{A}$-edges. For each $\mathcal{D}$-edge and $\mathcal{A}$-edge in the PEG, there always exist corresponding reverse $\bar{\mathcal{D}}$- and $\bar{\mathcal{A}}$-edges in the opposite direction, *i.e.*, $\forall (u, d, v), (u, a, v) \in E$, we have $(v, \bar{d}, u), (v, \bar{a}, u) \in E$. Therefore, PEGs are bidirected. Note that the bidirectedness is a prerequisite for CFL-reachability-based formulations of pointer analysis [32].

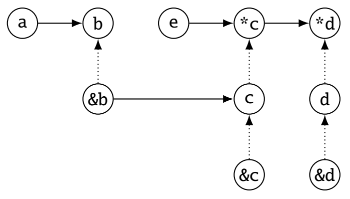
2.3 The Zheng-Rugina Alias Analysis Formulation

In a PEG, the alias analysis problem is formulated by the *CFG* shown in Figure 2, using EBNF notation. The *CFG* can also be represented using recursive state machines [2]. The equivalent recursive state machines of Zheng-Rugina formulation are adopted in Figure 3.

The formulation distinguishes two kinds of aliases:

```
int *a,*b;
int *c,*d,*e;
  b = a;
  c = &b;
  *d = *c;
  *c = e;
```

(a) A code snippet.



(b) Its PEG.

Figure 4. An example of alias analysis with the PEG. In the PEG, dotted edges represent $\mathcal{D}$-edges and solid edges $\mathcal{A}$-edges. The reverse edges (*i.e.*, $\bar{\mathcal{A}}$- and $\bar{\mathcal{D}}$-edges) are omitted for brevity.

- Memory aliases (M): two pointer variables are memory aliases if they denote the same memory location.
- Value aliases (V): two pointer variables are value aliases if they are evaluated to the same pointer value.

According to the grammar, nodes u and v in the PEG are aliases if there exist an M-path or V-path between them. Moreover, the memory aliases and value aliases, represented as summary edges (u, M, v) and (u, V, v), can be considered as binary relations on all node pairs. Following the discussion by Zheng and Rugina [54], we summarize the properties on the M and V relations as follows:

- V is *nullable*, M is not *nullable*;
- Both V and M are *symmetric*;
- V is *reflexive*, M is *reflexive* for non-address-taken variables;
- Neither V nor M is *transitive*.

Finally, we give an analysis example that summarizes the above discussions.

Example 1. *Figure 4 gives an example of alias analysis using a PEG. The C code snippet (left) and its PEG (right) are shown. In the PEG, nodes* `b` *and* `*c` *are memory aliases because the realized string $R(p)$ of path p =* `b`, `&b`, `c`, `*c` *is "$\bar{d}ad$", which can be generated from M in Figure 2. Similarly, nodes* `a` *and* `*d` *are value aliases since the realized string "$a\bar{d}ada$" can be generated from V. Note that V is not transitive. In the PEG, nodes* `a` *and* `*d`*, nodes* `*d` *and* `e` *are both value aliases. However, nodes* `a` *and* `e` *are not value aliases since the realized string "$a\bar{d}ad\bar{a}$" cannot be generated from V.*

2.4 Advantages of PEG

Alias analysis for C via CFL-reachability on PEGs has several advantages over traditional pointer analysis formulated as a dynamic transitive closure problem. We discuss some of the advantages below.

The most attractive feature is that PEGs depict the complex pointer assignments (e.g., `*d = *c`) directly without introducing any temporaries. As discussed in Appendix A, traditional inclusion-based pointer analyses need to transfer pointer statements into normal forms. These transformations may cause precision loss, since they introduce additional points-to or alias pairs to the original program [5, 6, 20]. For example, they may introduce a temporary variable `x` to break the assignment `*d = *c` in Figure 4(b) into two statements `*d = x` and `x = *c`, respectively. However, in the PEG representation, the pointer assignment is directly represented as an $\mathcal{A}$-edge (`*c`, a, `*d`).

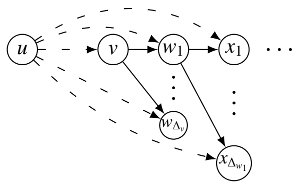
In PEGs, it is impossible for a variable in one connected component to be aliased with variables from other connected components. Therefore, PEGs can be decomposed before alias computation. Furthermore, we can reduce the memory consumption of alias analysis by separately processing each disconnected PEG. In contrast, for a traditional inclusion-based pointer analysis formulated as a dynamic transitive closure problem, it is difficult to distinguish the connected components because all edges in the graph represent points-to relationship and new edges can be inserted during the transitive closure computation [14]. The full reachability information of the constraint graph is known only after the points-to analysis, which cannot be exploited to optimize memory usage. For example, in Figure 4(b), variables `e` and `b` are connected only if variable `c` is resolved to point to `b`. In the literature, there has been work that heuristically determines the components and performs analysis on each component independently [21, 53]. However, the idea of connected component decomposition on PEGs is quite natural and can be performed using a simple depth-first search (DFS). Consequently, the CFL-reachability algorithm can work on each connected component of potentially much smaller size to achieve better performance.

3. Alias Analysis Algorithm

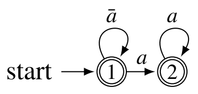
In this section, we present our alias analysis algorithm for C. Our algorithm takes PEGs as input and computes *all-pairs* CFL-reachability based on the Zheng-Rugina formulation. We begin by illustrating the basic idea of our algorithm (Section 3.1). We then describe our technique for reachability information propagation (Section 3.2), and give the alias analysis algorithm (Section 3.3). Finally, we describe how to apply the Four Russians' Trick to our algorithm (Section 3.4).

3.1 Basic Idea

Let us first discuss value aliases (V) concerned with only $\mathcal{A}$- and $\bar{\mathcal{A}}$-edges (*i.e.*, without $\mathcal{D}$- or $\bar{\mathcal{D}}$-edges). Then, we can safely substitute every occurrence of M with ε in rule (ZR2) from Figure 2. Therefore, V now generates the regular language $\bar{a}^*a^*$. We depict the equivalent finite state machine along with the reachability propagation illustration in Figure 5. Specifically, Figure 5(a) shows the reachability propagation of an arbitrary summary edge (u, X, v), where X corresponds to either state 1 or 2 in Figure 5(b). To generate new summary edges in Figure 5(a), it suffices to traverse the outgoing edges of node v. All these traversed edges are $\mathcal{A}$- and $\bar{\mathcal{A}}$-edges, which are the original edges of the input PEG. We give a concrete example below to demonstrate reachability propagation.



(a) Reachability propagation.



(b) The finite state machine.

Figure 5. Propagating reachability information along outgoing edges of node v according to a regular language $\bar{a}^*a^*$. In the left figure, dashed arrows denote summary edges and solid arrows $\mathcal{A}$- or $\bar{\mathcal{A}}$-edges. The notation Δ_u denotes the out-degree of node u.

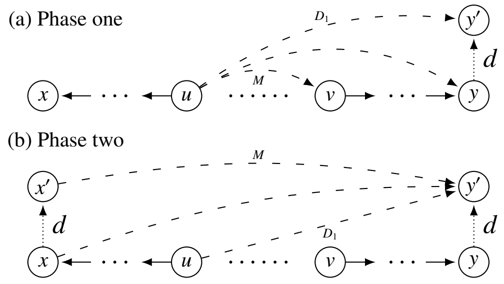


Figure 6. Two-phase propagation for an M-edge (u, M, v).

Example 2. *Consider a summary edge (u, X_1, v) in Figure 5(a), where X_1 corresponds to state 1 in Figure 5(b). We traverse outgoing $\bar{\mathcal{A}}$-edges of v. Using an $\bar{\mathcal{A}}$-edge $(v, \bar{a}, w_1)$, we generate the new summary edge (u, X_1, w_1). Then we continue traversing outgoing edges of w_1. Similarly, with an $\mathcal{A}$-edge (w_1, a, x_1), we generate (u, X_2, x_1). The new X_2-edge corresponds to state 2 representing the fact that reachability propagation is consistent with the state transitions in Figure 5(b).*

Then we take memory aliases (M) into consideration. Therefore, value aliases now concern all PEG edges. It is clear from rule (ZR1) in Figure 2 that each M-path is generated from a V-path. For value aliases described by rule (ZR2), we can infer that each V-path is generated by a path p whose $R(p) = \bar{a}^*a^*$, injected with zero or more M-paths.

The basic idea of our algorithm is to initialize a worklist that contains all reflexive M-paths (u, M, u), and use these M-paths to initialize the generation of new V-paths. The reachability propagation procedure is similar to the example in Figure 5. Specifically, for each M-path (u, M, v) popped from the worklist, we first propagate reachability information along the outgoing edges of v until a $\mathcal{D}$-edge (y, d, y') is encountered, depicted in Figure 6(a). We define the resulting summary edge (u, D_1, y') as a *pivot edge*. With this pivot

$$
\begin{array}{rcl}
M & ::= & DV \;\; d \\
DV & ::= & \bar{d} \;\; V \\
V & ::= & MAM \;\; AM_s \\
MAM & ::= & MA_s \;\; M_q \\
M_q & ::= & \varepsilon \\
M_q & ::= & M \\
MA_s & ::= & \varepsilon \\
MA_s & ::= & MA_s \;\; MA \\
MA & ::= & M_q \;\; \bar{a} \\
AM_s & ::= & \varepsilon \\
AM_s & ::= & AM_s \;\; AM \\
AM & ::= & a \;\; M_q
\end{array}
$$

Figure 7. The normal form of the *CFG* in Figure 2. It is used by the CFL-reachability algorithms in our paper. The subscripts q and s denote the question (?) and star (*) in standard EBNF notation, respectively.

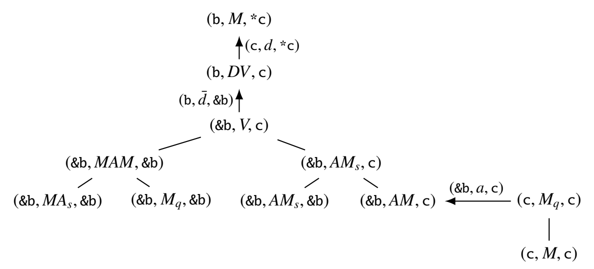


Figure 8. Computing M-edge (b, M, *c) according to an existing M-edge (c, M, c) in Figure 4(b). The solid edge denotes propagating *w.r.t.* a grammar rule from Figure 7. The arrow edge denotes propagating using an existing PEG edge.

edge, we continue propagating reachability information along the outgoing edges of u until a matched $\bar{\mathcal{D}}$-edge $(x', \bar{d}, x)$ is encountered, depicted in Figure 6(b). The newly generated M-path (x', M, y') is then inserted to the worklist, which is used to generate other V-paths. We name this procedure *two-phase propagation.*

The key benefit of our approach is that for each summary edge (u, X, v) popped from the worklist, our algorithm propagates reachability information along only the original PEG edges (*i.e.*, $\mathcal{A}$-, $\bar{\mathcal{A}}$-, $\mathcal{D}$- and $\bar{\mathcal{D}}$-edges) and *existing* M-edges in the current graph. In contrast, the traditional CFL-reachability algorithm considers all summary edges in the current graph *w.r.t.* the given *CFG*. Consequently, it computes more intermediate summary edges and takes more steps to obtain the V- and M-edges. Next, we give a concrete example to demonstrate this important benefit of our algorithm.

$$\begin{tikzcd}
\boxed{1} \arrow[r, "\bar{a}"] & \cdots \arrow[r, "\bar{a}"] & \boxed{2} \arrow[r, "\bar{a}"] & \cdots \arrow[r, "\bar{a}"] & \boxed{3} \arrow[r, "a"] & \cdots \arrow[r, "a"] & \boxed{4} \arrow[r, "a"] & \cdots \arrow[r, "a"] & \boxed{5}
\end{tikzcd}$$

Figure 9. The representative positions of M in V.

Positions	**Outgoing Paths** p	
	Left $R(p)$	**Right** $R(p)$
1	d	$\bar{a}^*a^*d$
2	a^*d	$\bar{a}^*a^*d$
3	a^*d	a^*d
4	$\bar{a}^*a^*d$	a^*d
5	$\bar{a}^*a^*d$	d

Table 1. Realized strings of outgoing paths from an M-edge (u, M, v) according to positions in Figure 9. The outgoing paths of node u realize left strings. Similarly, the outgoing paths of node v realize right strings. Note that we have omitted M-edges in this table.

Example 3. *Consider an M-edge (c, M, c) in Figure 4(b). We describe the main steps to compute a new M-edge (b, M, *c) using both the traditional CFL-reachability algorithm and ours.*

The traditional CFL-reachability algorithm takes 11 steps to compute the new summary edge, as shown in Figure 8. In each step, it refers to the given CFG in Figure 7 to compute intermediate summary edges. Specifically, it computes reflexive edges such as (&b, MA_s, &b) w.r.t. *the ε-rules in Figure 7. Moreover, it computes new intermediate summary edges such as (&b, MAM, &b) based on existing edges (&b, MA_s, &b) and (&b, M_q, &b)* w.r.t. *the rule "MAM ::= MA_s M_q". The procedure is similar to CFL parsing,* i.e.*, Figure 8 describes a partial parsing tree of the string "$\bar{d}a\bar{d}dd$", which corresponds to the path p = b, &b, c, &c, c, *c in the PEG.*

*Our two-phase propagation approach only takes three steps. Specifically, it takes one step (*i.e.*, (c, M, c) $\xrightarrow{(c,d,*c)}$ (c, D_1', *c)) in phase one propagation. In phase two propagation, it takes two additional steps, namely, (c, D_1', *c) $\xrightarrow{(c,\bar{a},\&b)}$ (&b, D_1, *c) $\xrightarrow{(\&b,d,b)}$ (b, M, *c). The details of our two-phase propagation are given in the subsequent sections. We can see from this example that our algorithm takes fewer steps and computes fewer intermediate summary edges.*

3.2 Propagating CFL-Reachability Information

There are two challenges in our two-phase propagation scheme: (1) to determine a proper propagation direction to find the pivot D_1'-edge, and (2) to handle M-edges correctly during propagation. We address both challenges in Sections 3.2.1 and 3.2.2, respectively. Section 3.2.3 gives the detailed description of our two-phase propagation.

3.2.1 Propagation Directions

Our algorithm generates new V-paths based on existing M-paths. Consider an M-path popped from the worklist, we need to identify its representative position in a V-path in order to initialize the two-phase propagation. When discussing propagation directions, we omit M-edges encountered during the two-phase propagation. We say a V-path is *nontrivial* if its derivation rule (ZR2) contains at least one a or $\bar{a}$. Figure 9 shows a nontrivial V-path along with five representative positions for M-paths. Specifically, positions 1 and 5 describe two endpoints of the V-path. According to rule (ZR1), reachability information can be propagated through a $\bar{\mathcal{D}}$-edge and a $\mathcal{D}$-edge at the endpoints. Positions 2 and 4 indicate that the M-paths can exist anywhere on the $\bar{a}^*$- and a^*-paths, respectively. Finally, position 3 represents the unique position that is between the $\bar{a}^*$- and a^*-paths.

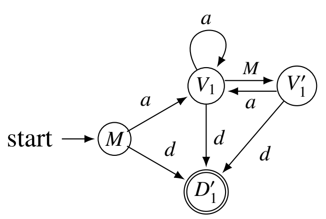
Table 1 summarizes all possible outgoing paths of an M-edge with their realized strings. Take position 4 in Figure 9 for example. The outgoing paths along the reversed edges from position 4 to position 1 realize "$\bar{a}^* a^*$". Together with the last outgoing $\mathcal{D}$-edge, the left realized string of position 4 in Table 1 is "$\bar{a}^* a^* d$". From the table, we can see that positions 1 and 5 are symmetric according to their left and right realized strings. Similarly, positions 2 and 4 are symmetric, too. Moreover, M-paths are also symmetric as discussed in Section 2.3. Therefore, for all M-edges (u, M, v) in positions 1 and 2, we can always use their symmetric edge (v, M, u) to propagate reachability information as if they were in positions 4 and 5. When the M-edges (u, M, v) are in positions 3, 4 and 5, the reachability information can always be propagated via outgoing $\mathcal{A}$- and $\mathcal{D}$-edges of node v, *i.e.*, through the right outgoing paths in Table 1. The propagation continues until it reaches the endpoint y with a $\mathcal{D}$-edge (y, d, y'), depicted in Figure 6. We then obtain the pivot D_1'-path (u, D_1, y') for the phase two propagation. To sum up, for each M-edge (u, M, v), we can use outgoing $\mathcal{A}$- and $\mathcal{D}$-edges of nodes u and v to initialize our two-phase propagation, without considering outgoing M-edges.

3.2.2 Properties of M-paths

We then discuss how to handle M-edges encountered during the two-phase propagation. Our alias analysis algorithm fully utilizes the properties of M-paths. For instance, in previous sections, we make use of their reflexivity to initialize the worklist and their symmetry to start the two-phase propagation. This section further discusses its *nonconsecutive* property. We say a path $p' = u', \ldots, v'$ is a *subpath* of path $p = u, \ldots, v$ iff both u' and v' are in p such that $p = u, \ldots, u', \ldots, v', \ldots, v$. Moreover, if the subpath p' is an X-path, we say it is an X-subpath. Formally, we have the following lemma:

Lemma 1. *For any V-path in the PEG, the M-subpaths are always nonconsecutive.*

Proof. All V-paths are generated *w.r.t.* the *CFG* in Figure 2. According to rule (ZR1), M-subpaths that belong to different V-paths are always separated by at least one $\bar{\mathcal{D}}$- or $\mathcal{D}$-edge. To prove the nonconsecutive property in rule (ZR2), we use



Input (α) **State** (Y_α)	***a***	***d***	***M***
M	V_1	D_1'	×
$\boldsymbol{V_1}$	V_1	D_1'	V_1'
$\boldsymbol{V_1'}$	V_1	D_1'	×

Figure 10. Phase one propagation.

case analysis. Since there are three unique Ms in this rule, we denote the whole rule as "$(M^1?\ \bar{a})^*\ M^2?\ (a\ M^3?)^*$". This leads to three cases:

- M^1 *and* $M^{\{1,2,3\}}$*:* There is an "$\bar{a}$" that immediately follows M^1. Therefore, each M^k that follows M^1 is separated by at least one "$\bar{a}$", where $k \in \{1, 2, 3\}$.
- M^2 *and* $M^{\{2,3\}}$*:* The question mark represents zero or one occurrence. As a result, M^2 cannot follow itself. Every M^3 is preceded by an "a", therefore, there exists at least one "a" between M^2 and M^3.
- M^3 *and* M^3*:* Due to the preceded "a", this case is similar to the case above. □

According to Section 3.2.1, for each M-path (u, M, v), we use the outgoing edges of nodes u and v to initialize phase one propagation. Due to the nonconsecutive property (Lemma 1), we cannot use outgoing M-edges. Thus, we have:

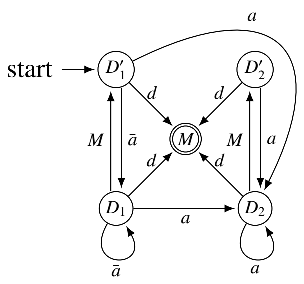
Lemma 2. *For each edge (u, M, v), it suffices to consider the outgoing $\mathcal{A}$- and $\mathcal{D}$-edges of u or v to initiate the first phase propagation.*

During the two-phase propagation, our algorithm handles the nonconsecutive property of M-paths by marking any intermediate X-path as an X'-path when an M-path is encountered. An X'-path cannot further make use of any additional M-paths to propagate reachability information. Specifically, in the phase one propagation shown in Figure 6(a), we propagate reachability information along outgoing edges of node v. Therefore, an X'-path (u, X', w) can only use outgoing $\mathcal{A}$- and $\mathcal{D}$-edges of w. Similarly, in the phase two propagation shown in Figure 6(b), an X'-path (w, X', y') cannot use outgoing M-paths of w. Moreover, an X'-path transits back to an X-path when encountering an $\mathcal{A}$- or $\bar{\mathcal{A}}$-edge.

3.2.3 Two-Phase Propagation

We now give the details of the two-phase propagation used in our alias analysis algorithm.

Phase One Propagation. In this phase, we propagate reachability information as depicted in Figure 6(a). The right outgoing $R(p)$ of positions 3, 4 and 5 in Table 1 is "a^*d". Therefore, we use the finite state machine in Figure 10 to handle all possible encountered outgoing edges. Specifically, in the three positions, node v of an M-edge (u, M, v) may encounter arbitrary outgoing $\mathcal{A}$-edges during propagation. We represent it as state V_1. Moreover, if state V_1 encounters an outgoing



Input (α) State (Y_α)	***a***	***ā***	***d***	***M***
$\boldsymbol{D_1'}$	D_2	D_1	M	×
$\boldsymbol{D_1}$	D_2	D_1	M	D_1'
$\boldsymbol{D_2'}$	D_2	×	M	×
$\boldsymbol{D_2}$	D_2	×	M	D_2'

Figure 11. Phase two propagation.

M-edge, it handles the nonconsecutive property and transits to state V_1'. Finally, when the rightmost $\mathcal{D}$-edge is encountered, states M, V_1 and V_1' transit to D_1' and the phase two propagation begins.

Phase Two Propagation. In this phase, we propagate reachability information *w.r.t.* the pivot D_1'-edge (u, D_1', y') depicted in Figure 6(b). We describe the corresponding finite state machine in Figure 11. Specifically, according to positions 3, 4 and 5 in Table 1, the possible outgoing paths of node u realize the string "$\bar{a}^*a^*d$". Therefore, we need two states D_1 and D_2, which is similar to the example in Figure 5. As before, two additional states D_1' and D_2' are required to handle the nonconsecutive property. Finally, when the leftmost $\mathcal{D}$-edge is encountered, a new M-edge is generated and the two-phase propagation terminates.

Each M-path in PEG is generated by prepending a $\bar{\mathcal{D}}$-edge and appending a $\mathcal{D}$-edge to an existing V-path. We use an M-subpath of a V-path to trigger the two-phase propagation and compute reachability summaries according to the finite state machines in Figures 10 and 11, respectively. Therefore, we have the following correctness lemma:

Lemma 3. *The two-phase propagation correctly computes new M-edges based on existing M-edges.*

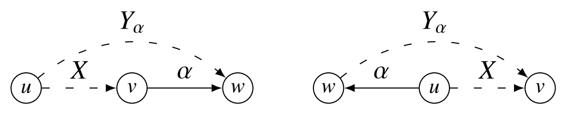
3.3 Alias Analysis Algorithms

This section introduces the two core algorithms for our *all-pairs* alias analysis: memory alias (Section 3.3.1) and value alias (Section 3.3.2), respectively.

3.3.1 Memory Alias Algorithm

The main algorithm for computing *all-pairs* memory aliases is given in Algorithm 1. It is a worklist-based algorithm that follows the traditional dynamic programming scheme for solving *all-pairs* CFL-reachability. The algorithm takes a PEG as input, and proceeds in two major steps:

- *Initialization.* The worklist W is initialized on lines 1-3. All non-address-taken nodes are considered. Specifically, every non-address-taken node u has an incoming $\mathcal{D}$-edge (u, d, v). Therefore, the realized path $R(p)$ for $p = u, v, u$ is "$\bar{d}d$", which describes a reflexive memory alias relation (u, M, u). The resulting M-edge is inserted into the graph. Note that the reflexivity of M plays an important role in our correctness analysis.



(a). Phase one. (b) Phase two.

Figure 12. Adding summary edges in two phases.

- *Reachability information propagation.* When a summary edge (u, X, v) is popped from the worklist, the reachability information is propagated using the two-phase propagation. Specifically, we use the FIND-TRANSITION procedure to look for the relevant transitions in the corresponding phase. For example, in the phase one propagation on lines 7-14, for each outgoing neighbor w of v connected via edge (v, α, w), the FIND-TRANSITION procedure returns state Y_α according to the transition table in Figure 10. The summary edge (u, Y_α, w) is then inserted to the PEG depicted in Figure 12. The phase two propagation on lines 16-23 is handled similarly.

The algorithm terminates when the worklist W becomes empty. All summary edges describing memory aliases are presented in the final PEG.

Theorem 1. *Algorithm 1 correctly computes all M-paths in the given PEG.*

Proof. Algorithm 1 handles all reflexive M-paths on lines 2-3. Therefore, we consider only non-reflexive M-paths. We use proof by contradiction. Suppose the claim is false; that is, there exists at least one non-reflexive M-path that Algorithm 1 does not compute. Let the path be $p = u, u_1, \ldots, v_1, v$, where $\mathcal{L}(u, u_1) = \bar{d}$, $\mathcal{L}(v_1, v) = d$ and path $p_1 = u_1, \ldots, v_1$ is a V-subpath. Since the concerned M-path is non-reflexive, the length of the V-path must be greater than 1. According to Lemma 3, the corresponding V-path should have no M-subpaths, *i.e.*, the V-path can only be "$\bar{a}^*a^*$". Note that M-paths are reflexive for all non-address-taken variables due to Section 2.3. Therefore, all PEG nodes w_1 in the path p_1 represent address-taken variables, which contradicts the fact that one cannot assign an address-taken variable to another address-taken variable in C language. And this completes the proof. □

3.3.2 Value Alias Algorithm

The memory alias algorithm does not compute V-paths for top-level variables, because these variables do not have outgoing $\mathcal{D}$-edges to obtain the pivot D_1'-edge. The basic idea of our value alias algorithm is to compute V-reachability based on existing phase one summary edges, without resorting to the pivot edge. A naïve approach to compute all V-paths is to reuse the existing $\mathcal{A}$-, $\bar{\mathcal{A}}$- and M-edges and perform an additional two-phase propagation. According to rule (ZR2),

Algorithm 1: Computing memory aliases.

```
Input  : PEG $G = (V, E)$;
Output : the set of summary edges
foreach $v \in V$ do
    if $v$ has incoming $\mathcal{D}$-edges then
        insert $(v, M, v)$ to $G$ and to $W$
while $W \neq \emptyset$ do
    $(u, X, v) \leftarrow$ Select-From($W$)
    /* Phase 1 propagation.                    */
    if $X = M$ or $X = V_1$ or $X = V_1'$ then
        foreach $\alpha \in \{a, d, M\}$ do
            $Y_\alpha \leftarrow$ Find-Transition($X, \alpha$)
            if $Y_\alpha == \times$ then continue
            foreach $w \in$ Out($v, \alpha$) do
                if $(u, Y_\alpha, w) \notin G$ then
                    insert $(u, Y_\alpha, w)$ to $G$ and to $W$
                    if $Y_\alpha == M$ then insert $(w, Y_\alpha, u)$ to
                    $G$ and to $W$
    /* Phase 2 propagation.                    */
    if $X = D_1$ or $X = D_1'$ or $X = D_2$ or $X = D_2'$ then
        foreach $\alpha \in \{a, \bar{a}, d, M\}$ do
            $Y_\alpha \leftarrow$ Find-Transition($X, \alpha$)
            if $Y_\alpha == \times$ then continue
            foreach $w \in$ Out($u, \alpha$) do
                if $(w, Y_\alpha, v) \notin G$ then
                    insert $(w, Y_\alpha, v)$ to $G$ and to $W$
                    if $Y_\alpha == M$ then insert $(v, Y_\alpha, w)$ to
                    $G$ and to $W$
```

Algorithm 2: Computing value aliases.

```
Input  : PEG $G = (V, E)$ with pre-computed $M$-edges;
Output : the set of summary edges
Initialize worklist $W_v$ to be empty
Compute the transitive closure of existing $V_1$-paths
Compute all $V_1{}'$-paths using $M$- and $V_1$-paths
foreach $\mathcal{A}$-edge $(u, a, v)$ do
    foreach $\alpha \in \{V_1, V_1', M\}$ do
        foreach $w \in$ Out($v, \alpha$) do
            if $(u, V_2, w) \notin G$ then
                insert $(u, V_2, w)$ to $G$ and to $W_v$
foreach $(u, V_2, v) \in W_v$ do
    foreach $\alpha \in \{V_1, V_1', M, V_2\}$ do
        foreach $w \in$ Out($u, \alpha$) do
            if $(w, V_2, v) \notin G$ then
                insert $(w, V_2, v)$ to $G$
```

the correctness is immediate. Unlike the naïve approach, we reuse the existing V_1 and V_1'-edges instead, and reformulate it as an Υ-reachability problem. Algorithm 2 gives our value alias algorithm. We consider it the main new algorithm of this paper and discuss the correctness as follows.

Any trivial V-path is an M-path which has been correctly computed by Algorithm 1. Therefore, we focus on nontrivial V-paths. We structure the proof into two parts. In the first part, we show that the *all-pairs* nontrivial V-reachability on PEGs can be expressed as *all-pairs* Υ-reachability. Then, we show that our value alias algorithm correctly solves the *all-pairs* Υ-reachability problem.

Let us consider rule (ZR2) from Figure 2 that describes nontrivial V-paths. Due to the bidirectedness of PEG, it suffices to consider a nontrivial V-path with at least an $\mathcal{A}$-edge. Rule (ZR2) can be rewritten as follows:

$$V \quad ::= \quad (M?\,\bar{a})^*\ M?\ a\ M?\ (a\ M?)^*$$

We introduce two additional rules: $\Gamma \ ::= \ M?\,(a\,M?)^*$ and $\bar{\Gamma} \ ::= \ (M?\,\bar{a})^*\ M?$. Thus, the above rule becomes:

$$V \quad ::= \quad \bar{\Gamma}\ a\ \Gamma \tag{A1}$$

According to the bidirectedness of PEGs and the symmetry of M-edges, it is immediate that $\bar{\Gamma}$ is the inverse of Γ. Therefore, it suffices to consider only Γ-paths in the PEGs. Namely, given a PEG $G = (V, E)$, we have $(v, \Gamma, u) \in E$ for all $(u, \bar{\Gamma}, v) \in E$. We also give a new definition of the reachability considered in the proof.

Definition 2 (Υ-reachability). *A PEG node v is Υ-reachable from u iff there exists an $\mathcal{A}$-edge (x, a, y) such that u is Γ-reachable from x and v is Γ-reachable from y, respectively.*

According to rule (A1), the following lemma is immediate.

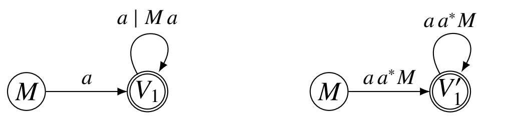
Lemma 4. *The* all-pairs *nontrivial V-reachability on PEGs is equivalent to the* all-pairs *Υ-reachability.*

We next discuss the proprieties of Υ-reachability. Nonterminal Γ represents a regular expression. Since regular expression is a Kleene algebra, we use two theorems of Kleene algebra [10] to simplify the regular expressions discussed in our proof. Kleene algebras are algebraic structures with operators $+$, $\cdot$, *, 0, and 1, with certain properties.[1] In particular, we use the *denesting rule* and the *shifting rule* [23, p. 57] described below:

$$(a^* b)^* a^* \quad = \quad (a \mid b)^* \tag{R1}$$

$$a(ba)^* \quad = \quad (ab)^* a \tag{R2}$$

[1] In regular expressions, operator "+" represents as "|", and "·" is usually omitted.



(*a*). State V'_1 eliminated. (*b*) State V_1 eliminated.

Figure 13. State elimination of Figure 10. Note that there is no need to consider state D'_1.

Therefore, we have the following:

$$\Gamma = M?\,(a\,M?)^* \overset{R2}{=} (M?\,a)^* M?$$

For an arbitrary PEG node, all nonempty outgoing Γ-paths can be classified into three categories:

- Γ_0-*paths:* trivial Γ-paths where $\Gamma_0 = M$;
- Γ_1-*paths:* nontrivial Γ-paths where $\Gamma_1 = M\,a\,(M?\,a)^* M?$;
- Γ_2-*paths:* nontrivial Γ-paths where $\Gamma_2 = a\,(M?\,a)^* M?$.

The following two lemmas summarize two properties of the Γ-paths.

Lemma 5. *All Γ_2-paths in PEGs can be obtained by prepending an $\mathcal{A}$-path to existing Γ_0- or Γ_1-paths.*

Proof. According to the definition, all Γ_2-paths can be classified into four categories:

(1) $\Gamma_2 = a$;
(2) $\Gamma_2 = a\,M$;
(3) $\Gamma_2 = a\,M\,a\,(M?\,a)^* M?$;
(4) $\Gamma_2 = a\,a\,(M?\,a)^* M?$.

It is clear that the "Γ_2"s in categories (2) and (3) are derived by prepending an "a" to Γ_0 and Γ_1, respectively. Moreover, for each $\mathcal{A}$-edge (u, a, v), node v always represents a non-address-taken variable with the reflexive M-edge (v, M, v). Therefore, the Γ_2-paths (u, Γ_2, v) in category (1) can always be generated using existing (u, a, v) and (v, Γ_0, v). Similarly, due to the reflexive M-edge (v, M, v), there always exists a Γ_1-edge (v, Γ_1, w) for each $(v, \Gamma_{2(4)}, w)$, where $\Gamma_{2(4)} = a\,(M?\,a)^* M?$. Therefore, it is similar to category (3). □

Lemma 6. *For any PEG node u that represents a non-address-taken variable, there always exists a Γ_1-path (u, Γ_1, v) for each Γ_2-path (u, Γ_2, v).*

Proof. There exists a reflexive M-path (u, M, u) for each PEG node u that represents a non-address-taken variable. As a result, for each Γ_2-path (u, Γ_2, v), we always have a Γ_1-path $u, u, \ldots, v$ with (u, M, u) and (u, Γ_2, v). □

Next, we discuss how Algorithm 2 solves *all-pairs* ϒ-reachability. On lines 2 and 3, Algorithm 2 handles V_1- and V'_1-paths. According to Figure 10, we describe the regular expressions that V_1 and V'_1 represent, respectively. Specifically, we apply the standard "state elimination" method [19] to the finite state machine and simplify the resulting expressions. The intermediate finite state machines with accepting states V_1 and V'_1 are given in Figure 13. We also give the full steps as follows:

$$\begin{aligned} V_1 &= M\,a\,(a \mid M\,a)^* \\ &= M\,a\,(M?\,a)^* \\ V'_1 &= M\,a\,a^* M\,(a\,a^* M)^* \\ &\overset{R2}{=} M\,a\,(a^* M\,a)^* a^* M \overset{R1}{=} M\,a\,(a \mid M\,a)^* M \\ &= M\,a\,(M?\,a)^* M \end{aligned}$$

From the simplified expressions, it is clear that $\Gamma_1 ::= V_1 \mid V'_1$ and $V'_1 ::= V_1\,M$. Note that relation V_1 is transitive. According to the state transitions in Figure 10, for any two V_1-paths $p_1 = (u_1, M, v_1)(v_1, a, w_1)\ldots(y_1, a, z_1)$ and $p_2 = (z_1, M, v_2)(v_2, a, w_2)\ldots(y_2, a, z_2)$, the path that concatenates p_1 and p_2 is always a V_1-path. Algorithm 2 obtains all V_1-paths by computing the transitive closure (line 2). However, relation V'_1 is not transitive due to Lemma 1. According to the above simplified expressions, we obtain all V'_1-paths by appending an M-edge to existing V_1-paths (line 3). Therefore, it computes all Γ_1-paths. On the other hand, the Γ_0-paths (*i.e.*, M-paths) have already been obtained by the memory alias algorithm. The following lemma summarizes the discussion.

Lemma 7. *Algorithm 2 correctly computes all Γ_0- and Γ_1-paths for each PEG node.*

Algorithm 2 considers all $\mathcal{A}$-paths on line 4. For any $\mathcal{A}$-path (x, a, y), Algorithm 2 traverses all outgoing Γ_0- and Γ_1-paths on lines 5-8. Therefore, it finds all Γ_2-paths in the PEG due to Lemma 5. According to the $\mathcal{A}$-edge, node y must represent a non-address-taken variable. Therefore, due to Lemma 6, it suffices to use only outgoing Γ_0- and Γ_1-paths of y as all outgoing Γ-paths. Combining the above discussion, we have the following lemma.

Lemma 8. *Given an $\mathcal{A}$-edge (x, a, y), Algorithm 2 correctly computes all Γ_2-paths for each PEG node. Moreover, it finds all nodes v that are Γ-reachable from y.*

Algorithm 2 stores all Γ-reachable nodes v from x using outgoing V_2-paths (x, V_2, v). Therefore, the M-, $\{V_1, V'_1\}$- and V_2-paths in the PEG necessarily and sufficiently represent all Γ_0-, Γ_1- and Γ_2-paths. On lines 10-13, Algorithm 2 traverses all nodes u that are Γ-reachable from x. Finally, Algorithm 2 pairs nodes u and v by inserting a summary edge (u, V_2, v), *i.e.*, it correctly solves the *all-pairs* ϒ-reachability given in Definition 2. Any ϒ-reachability query concerned with PEG nodes u and v can be answered by testing whether there exists a V_2-path between nodes u and v. Combining Lemma 4, we obtain the following theorem.

Theorem 2. *Algorithm 2 correctly computes all V-paths in the given PEG.*

Complexity Analysis. We conduct the complexity analysis for Algorithms 1 and 2. The worst-case complexity of Algorithm 1 is $O((m + M)n)$, where M denotes the number of M-edges, and n and m denote the numbers of nodes and edges in the original graph, respectively. For each summary edge (u, X, v), Algorithm 1 traverses its neighbors connected via $\mathcal{A}$-edges, $\mathcal{D}$-edges and M-edges. Let k and Δ_v denote the grammar size and the degree of node v concerning these three kinds of edges, respectively. From Figures 10 and 11, we have $k = 7$ since there are seven kinds of summary edges. The total number of steps required are $7 \cdot \Sigma_{(u,v)}\Delta_v = 7 \cdot \Sigma_u(\Sigma_v\Delta_v)$. For the value alias algorithm in Algorithm 2, the running time is dominated by lines 9-13. There can be $O(n^2)$ V_2-edges in the worst case. For each V_2-edge, the **foreach** loop on line 11 takes $O(n)$ time. Therefore, the worst-case time complexity for the whole alias analysis algorithm is $O(n^3)$.

Connected Component Decomposition. The worst-case time complexity depends on the number n of nodes in the PEG. Our connected component (CC) decomposition technique reduces n by decomposing the original PEG into connected components. Since the nodes in two connected components are unreachable, computing the reachability on those smaller components yields the same results as computing on the original PEG. The CC decomposition can be done using a simple linear-time depth-first search on the PEG. We further investigate this optimization's practical benefits in the evaluation section.

3.4 Saving a Logarithmic Factor

The Four Russians' trick [4] is known as a popular technique for speeding up set operations under the random access machine model with uniform cost criterion. The original paper proposed an $O(\log d)(n^3/\log n)$ algorithm for finding the transitive closure of a directed graph with n nodes and diameter d. The technique has been applied in various contexts. Examples include shortest path problems [7], Boolean matrix multiplication [1, Ch. 6], and k-clique problems [45], to name just a few.

In particular, this technique has also been adopted for fast recognition of context-free languages [36] as well as reachability problems in recursive state machines [8], resulting in a logarithmic speedup. We can apply this technique to Algorithm 1 directly. We first recall some preliminaries. We begin by assuming the RAM model has word length $\theta(\log n)$ and constant-time bitwise operations on words. Let U denote a universe of n elements. A subset of U can be represented as a bit vector (*a.k.a.* characteristic vector) of length n by representing each element as a single bit. The characteristic vector is then stored in $O(\lceil n/\log n\rceil)$ words each with $\theta(\log n)$ bits. Following the work of Chaudhuri [8], we refer to the resulting data structure as *fast set*, which permits the following two operations:

- *insert(X, i)*: insert an element i into *fast set X*.
- *diff(X, Y)*: compute the set difference between *fast sets X* and Y and return a list of all the resulting elements.

Lemma 9. *Given* fast sets $X, Y \subseteq \{1, \ldots, n\}$*, and* $i \in \{1, \ldots, n\}$*,*

(i) insert *(X, i) takes* $O(1)$ *time;*
(ii) diff *(X, Y) takes* $O(\lceil n/\log n\rceil + v)$ *time, where v is the number of elements in the result set.*

Proof. Claim (i) is obvious by determining the position of i in relevant word of X and then performing the bitwise `or` operation. Claim (ii) follows in two steps. First, we perform the bitwise operations on the words comprising X and Y, resulting in $Z = X \setminus Y$. This takes $O(\lceil n/\log n\rceil)$ time under the assumed RAM model. Then, we list all elements in Z by repeatedly finding and turning off the most significant bit, which takes time $O(v)$, where v is the number of elements. If this operations are not directly supported, we can pre-compute the answers to all words (or pairs of words) with $O(n)$ pre-processing time and subsequently perform table lookups. □

In our algorithm, we can represent the In and Out sets using *fast sets*. Therefore, lines 11-12 in Algorithm 1

```
**foreach** $w \in \text{Out}(v, \alpha)$ **do**
  **if** $(u, Y_\alpha, w) \notin G$ **then**
```

can be changed using the *fast-sets* operations as

$$\textbf{foreach } w \in \textit{diff}(\text{Out}(v, \alpha), \text{Out}(u, Y_\alpha)) \textbf{ do}.$$

Similarly, lines 20-21 can be changed to

$$\textbf{foreach } w \in \textit{diff}(\text{Out}(u, \alpha), \text{In}(v, Y_\alpha)) \textbf{ do}.$$

The new algorithm takes $O(n/\log n)$ time to traverse the three kinds of edges for each node n. This technique can also be applied to the value alias algorithm in Algorithm 2. As a result, the total time complexity is $O(n(n \cdot n/\log n)) = O(n^3/\log n)$.

4. Evaluation

To evaluate the effectiveness, we apply our subcubic alias analysis on some *recent stable* releases of widely-used C programs in the pointer analysis literature. Moreover, we evaluate three algorithms solving *all-pairs* CFL-reachability on PEGs: The traditional cubic CFL-reachability algorithm [34, 51], Chaudhuri's subcubic algorithm [8] and our proposed algorithm. The results demonstrate that our alias analysis algorithm that solves *all-pairs* CFL-reachability performs extremely well in practice. For instance, it can analyze the Linux kernel in about 30 seconds.

4.1 Experimental Setup

Benchmark Selection. The set of C programs used in our evaluation is described in Table 2. For each program, we list its number of source lines of code (SLOC), its number of procedures, the size of its Program Expression Graphs (PEGs), and the number of temporaries introduced in

Program	SLOC	#Procs	PEG		#Temps
			# Nodes	# Edges	
Gdb-7.5.1	1,828K	10,536	649,564	1,219,788	6,472
Emacs-24.2	254K	3,626	687,691	1,290,200	2,424
Insight-6.8-1a	1,742K	10,507	787,289	1,494,168	7,898
Gimp-2.8.4	702K	17,842	872,681	1,675,546	13,876
Ghostscript-9.07	851K	12,211	1,198,753	2,368,086	6,806
Wine-1.5.25	2,306K	70,923	4,652,983	8,472,950	25,064
Linux-3.8.2	10,601K	138,095	12,807,645	23,398,670	69,840

Table 2. Benchmark applications. The SLOC is reported by `sloccount` counting only C code.

the traditional inclusion-based pointer analysis. The subject programs were obtained from the official websites. Gdb is the GNU debugger; Emacs is a text editor; Insight is a GUI for Gdb; Gimp is an image processing application; Ghostscript is a PostScript interpreter and PDF generator; Wine is a Windows emulator; and Linux is the kernel of the Linux operating system.

PEG Generation. All C programs used in our evaluation are processed by Gcc-4.8.1. We then apply the tool described in the wave propagation work [27] to generate constraint files for traditional inclusion-based pointer analysis. Specifically, the tool dumps intermediate constraints for each source file based on Gcc's whole program analysis result. The intermediate constraints contain call graph information (with resolved function pointers) for interprocedural analysis in our work. We finally develop a Perl script to generate a global PEG from the intermediate constraints. In particular, the temporaries introduced in transforming the source code to the four standard forms are eliminated, represented as the "#Temp" column in Table 2. We observe that the number of temporaries is relatively small compared to the number of pointer variables (*i.e.*, "#Nodes" in the PEGs).

Implementation. Both our analysis algorithm and the traditional CFL-reachability algorithms are implemented in C++ with extensive use of the Standard Template Library (STL). The Four Russians' Trick used in Chaudhuri's subcubic CFL-reachability algorithms is implemented with a combination of Gcc's own built-in functions operating on bitvectors. Those built-in functions are each translated to a single CPU instruction. For example, the bit scan reverse (`bsr`) instruction can locate the first set bit, clear the bit and return its position in one CPU instruction, which can be considered as constant time. For more bitwise tricks, we refer the reader to Warren's book [46] and Andersen's bit twiddling hacks page.[2]

All of the executables are compiled with Gcc-4.8.1 with "-O2" optimization. The algorithms take the same PEGs as input. Their outputs are verified to ensure consistency and correctness. Note that our algorithm computes the memory aliases (*i.e.*, the *M*-edges) and value aliases (*i.e.*, the *V*-edges) in two phases. All experiments were conducted on a machine with Intel Xeon E5520 CPU and 32GB RAM, running Ubuntu-13.10.

[2] `http://graphics.stanford.edu/~seander/bithacks.html`

4.2 Performance of Alias Analysis Algorithms

We present the first performance report of subcubic alias analysis. We use the normal forms in Figure 7 for both cubic and Chaudhuri's subcubic CFL-reachability algorithms. Note that our algorithm based on the two-phase propagation does not require such normal forms. We then compare the time and memory consumption among traditional cubic CFL-reachability algorithm, Chaudhuri's subcubic algorithm and our proposed algorithm. We also evaluate the practical benefits of applying CC decomposition in CFL-reachability-based alias analysis.

4.2.1 Time Consumption

Table 3 shows the time and memory consumption of the three evaluated algorithms computing *all-pairs* CFL-reachability. The time and memory consumption data is collected differently. Specifically, the running time columns in Table 3 report the *accumulated* running time on all PEGs. On the other hand, the memory consumption columns in Table 3 report the *maximum* memory consumed in the project, since the used memory can be freed before processing next PEG. Moreover, our algorithm computes CFL-reachability in two stages (discussed in Section 4.4.1). The *M* column of our approach indicates the running time of computing only memory aliases, while the "*M* + *V*" column represents the running time of computing both memory and value aliases.

From the running time columns, we can see that the traditional cubic *all-pairs* CFL-reachability algorithm does not scale well. For example, the cubic algorithm takes about 15 minutes to complete on Gimp, which is already the best running time result of all programs used in our study. This result explains why there has been no practical *all-pairs* CFL-reachability-based pointer analysis for C. We note that Chaudhuri's subcubic algorithm brings tremendous speedup in practice. Specifically, Chaudhuri's subcubic algorithm using the Four Russians' Trick is more than 262.5 times faster than the cubic algorithm on average. The Linux kernel project takes Chaudhuri's subcubic algorithm the longest time to analyze. However, it is still within three minutes,

Program	Time				Memory			
	Cubic	Subcubic	Our		Cubic	Subcubic	Our	
			M	***V* + *M***			***M***	***V* + *M***
Gdb-7.5.1	3721.12	12.00	1.76	3.34	150.40	103.43	44.35	51.48
Emacs-24.2	96270.20	140.50	29.38	56.10	1095.00	1911.17	540.47	619.15
Insight-6.8-1a	990.40	9.57	0.78	1.50	116.19	150.59	44.14	50.86
Gimp-2.8.4	885.09	8.98	0.70	1.30	56.85	51.51	18.29	20.47
Ghostscript-9.07	6053.17	31.88	4.55	8.47	256.25	272.54	112.15	128.75
Wine-1.5.25	12748.60	72.99	5.01	9.69	451.50	1450.96	60.36	69.97
Linux-3.8.2	49540.80	179.55	19.76	38.83	236.44	198.95	52.45	81.59

Table 3. The performance of three CFL-reachability-based alias analyses: time in seconds and memory in MB.

which is quite acceptable for such a large project. Note that it typically takes more than 30 minutes to compile the Linux kernel (without executing `make` in parallel) on the machine used for our experiments.

Among the three evaluated algorithms, our algorithm achieves the best performance. In particular, our memory alias algorithm is 9.6 times faster than Chaudhuri's subcubic algorithm. Moreover, our whole alias analysis algorithm is 5.0 times faster than Chaudhuri's subcubic algorithm and is three orders faster than the cubic CFL-reachability algorithm.

4.2.2 Memory Consumption

The actual memory consumption of the cubic algorithm is slightly different from the subcubic algorithms. Despite the memory taken by the iterative computation, most of the memory is taken by the underlying data structures used to represent the graphs. Specifically, the cubic algorithm typically uses an adjacency list to store all nodes in the graphs. It can be observed from Table 2 that the PEGs are quite sparse in practice, with $m = O(n)$ where n and m represent the number of nodes and edges, respectively. As a result, the space required to store the PEGs in the cubic algorithm is $O(m) = O(n)$.

On the other hand, the space required to store the PEGs in subcubic algorithms is $O(n^2)$, because each node needs several fast sets for representing all summary edges in the graph. In particular, all the terminals and nonterminals should be considered to initialize the corresponding fast sets. For instance, the normal forms used in the CFL-reachability algorithms contain four terminals and nine nonterminals, as shown in Figure 7. The minimal amount of memory required to represent the largest PEG in Emacs with 15,690 nodes are 400 MB. However, only 11.8 MB is required to store the largest PEG in Gimp with 2,693 nodes. From the memory columns in Table 3, we can observe that both the cubic and Chaudhuri's subcubic CFL-reachability algorithms demand similar amount of memory for the largest PEG. For Emacs and Wine, Chaudhuri's subcubic algorithm consumes 1.7 times and 3.2 times more memory, respectively, since the two programs have larger PEGs. Our algorithm demands the least amount of memory. In particular, our algorithm needs only ten terminals and nonterminals in total. On average, our subcubic alias algorithm consumes 5.1 times less memory than Chaudhuri's subcubic algorithm.

4.3 Understanding the Speedup

4.3.1 Graph Density

In order to understand the performance gain better, we calculate the graph densities in Table 4. The columns in Table 4 represent the number of original edges, and the number of memory and value alias edges, as well as all summary edges in the final graphs, respectively, for each program. Note that Chaudhuri's subcubic algorithm computes the same results as the traditional cubic CFL-reachability algorithm. Together with Table 2, we first note that the number of edges $m \ll n^2$ for all programs, which indicates that the alias analysis graphs are unlikely to be dense in practice. We also observe that our algorithm computes fewer summary edges in the final graphs. Specifically, the number of final summary edges in our memory alias algorithm and whole alias analysis algorithm is 4.2 times and 1.4 times fewer than traditional CFL-reachability algorithms, respectively. This gives the evidence that our fast algorithm takes fewer steps to compute the *all-pairs* M- and V-reachability.

Moreover, our algorithm computes the memory aliases based only on original PEG edges and memory alias edges in the final graphs. The number of memory alias summary edges are 3.1 times fewer than the number of original edges. On the other hand, the number of other summary edges is far greater than the number of original edges. For example, the V-edges and final edges are, respectively, 14.5 times and 31.7 times more. In practice, our algorithm performs better than Chaudhuri's subcubic CFL-reachability algorithm based on the two facts that it propagates reachability along fewer edges and computes fewer summary edges in total.

4.3.2 Impact of CC Decomposition

The scalability of the subcubic algorithms depends on the size of the input graph, as observed in Zhang *et al.*'s recent work [52]. For instance, an 8GB RAM machine can only afford to store a PEG with at most 70,164 nodes. Therefore,

Program	#Orig. Edges	#V-Edges	#M-Edges	#Final Edges		
				CFL	Our	
					M	$M + V$
Gdb-7.5.1	1,219,788	12,904,372	356,075	29,961,321	8,277,405	21,275,891
Emacs-24.2	1,290,200	49,011,799	758,925	112,772,537	41,055,899	89,282,405
Insight-6.8-1a	1,494,168	12,560,471	432,657	27,573,822	6,283,299	18,826,885
Gimp-2.8.4	1,675,546	16,809,343	518,511	33,151,263	6,602,726	22,448,402
Ghostscript-9.07	2,368,086	35,910,829	705,121	76,022,402	20,435,212	58,528,624
Wine-1.5.25	8,472,950	79,613,731	2,769,135	161,447,212	33,447,747	108,817,695
Linux-3.8.2	23,398,670	234,930,383	6,272,658	482,622,025	102,138,615	324,789,282

Table 4. The graph density information for each algorithm.

Program	#CC	PEG sizes of CCs	
		Maximum	Average
Gdb-7.5.1	66,154	4,350	9.82
Emacs-24.2	67,608	15,690	10.17
Insight-6.8-1a	75,373	4,350	10.45
Gimp-2.8.4	79,572	2,693	10.97
Ghostscript-9.07	87,768	7,025	13.66
Wine-1.5.25	537,370	5,106	8.66
Linux-3.8.2	1,449,718	4,755	8.83

Table 5. Connected component information on the benchmark programs.

it is infeasible to feed the global PEGs described in Table 2 for alias computation.

The CC decomposition technique can significantly reduce the size of each PEG. The cost is negligible, since a simple linear-time DFS through the global PEG is sufficient. Table 5 shows, for each program, the number of its connected components, and the maximum and average sizes of its PEGs across the connected components. As expected, each CC typically has fewer than 20 PEG nodes on average, which can be effectively handled by the subcubic algorithms in practice. We find that the overall performance of each of the evaluated algorithms depends on the largest PEGs encountered during analysis. In our evaluation, Emacs has the largest PEG after CC decomposition. Therefore, it requires the most memory in all evaluated algorithms.

4.4 Discussions

Finally, we discuss the main findings in this work.

4.4.1 Staged CFL-Reachability

Perhaps the most interesting finding in our work is that it is possible to design a staged CFL-reachability algorithm which is faster than the traditional CFL-reachability algorithm. In particular, different stages focus on *all-pairs* reachability for different summary edges. We show that our memory alias algorithm is 9.6 times and 1.2 times faster than Chaudhuri's subcubic algorithm and our value alias algorithm, respectively. Moreover, our value alias algorithm depends on the results of memory aliases. In practice, client analyses may only be interested in some of the summary edges (*e.g.*, M-edges). For these clients, it is possible to design an efficient persistence scheme [48] to store the intermediate results to avoid duplicated computation among different stages.

The CFL-reachability algorithm can also use the summary edges from different stages to bootstrap each other. For example, in our value alias algorithm, we can reuse all V_1-, V_1'- and M-edges obtained from the memory alias algorithm. The value alias algorithm only needs to perform additional propagation over top-level variables. The key to enable a staged CFL-reachability algorithm is to exploit the dependencies between the relevant nonterminals in the *CFG*.

An Application to Points-to Analysis. The Zheng and Rugina points-to formulation [54] only concerns memory alias edges. Therefore, the staged CFL-reachability algorithm is more efficient than traditional CFL-reachability algorithm that computes both V- and M-paths.

4.4.2 CFL-Reachability via Partial Summary Edges

In practice, it is possible to design a more efficient CFL-reachability algorithm using only some of the nonterminal edges. For example, in our memory alias algorithm, each worklist item uses only the original PEG edges and M-edges to propagate reachability information. As shown in Table 4, the memory edges are quite sparse. Propagating reachability information through those sparse edges in our memory alias algorithm yields more than 9.6 times speedup over Chaudhuri's subcubic CFL-reachability algorithm.

On the other hand, our fast algorithm depends on the properties of M-edges on PEGs, which does not intend to improve the general CFL-reachability algorithm. Specifically, our algorithm does not compute the *all-pairs* M- and V-reachability for arbitrary graphs since an arbitrary graph does not preserve the reflexivity of M-edges on non-address-taken nodes. Our algorithm gives the evidence that it is possible to scale CFL-reachability computation on some specialized problem instances, without resorting to all summary edges.

4.4.3 Limitations of the Subcubic Algorithm

The Four Russians' Trick is the key technique to scale the *all-pairs* CFL-reachability algorithm. In particular, it improves the worst-case complexity of the traditional CFL-reachability algorithm and brings tremendous speedup in practice. However, the subcubic CFL-reachability algorithms have two sources of limitations.

The first source of limitation is the size of the input graph. Theoretically, the subcubic algorithms have quadratic space complexities. As aforementioned, in practice, an 8GB RAM machine can only afford to store a graph with at most 70,164 nodes. In order to make the subcubic algorithm scale, we need to reduce the size of the input graphs. As in our alias analysis, the input graphs are the PEGs of each CC, which makes the analysis scalable.

The second source of limitation is the size of the input grammar. The subcubic algorithm needs to allocate space for storing the summary edges of all nonterminals and terminals from the input grammar. Reducing the grammar size may have practical benefits in saving required memory. It is possible to exploit properties on the input grammar and represent the grammar using fewer nonterminals. For example, our analysis algorithm uses fewer nonterminals than Chaudhuri's subcubic algorithm, thus, consumes less memory (Table 3).

5. Related Work

This section surveys the most relevant related work to our study, namely CFL-reachability, points-to analysis, and alias analysis.

5.1 CFL-Reachability

The CFL-reachability framework was initially proposed by Yannakakis for Datalog chain query evaluation [51]. Later, it has been used to formulate interprocedural dataflow analysis [34] and many other program analysis problems [12, 22, 28, 30–33, 38, 40, 49]. Both points-to analysis and alias analysis can be formulated as CFL-reachability problems.

CFL-reachability algorithms have cubic time worst-case complexity, commonly known as the *cubic bottleneck* in flow analysis [16]. As a result, there has been no practical CFL-reachability-based alias analysis solving *all-pairs* CFL-reachability for C. When the underlying CFL is restricted to the Dyck languages that generate matched pairs of parentheses, there exists improved algorithms solving the *all-pairs* Dyck-CFL-reachability with applications to alias analysis for Java [52] and polymorphic flow analysis [22] are proposed. Moreover, Chaudhuri proposed a subcubic algorithm [8]. However, its practical benefits remain unreported. In this paper, we have presented the design, implementation and evaluation of the first subcubic CFL-reachability-based alias analysis.

5.2 Points-to Analysis

Precise pointer analysis is a computationally hard problem. Any practical pointer analysis should approximate the precise solution. Compared to equality-based (Steensgaard-style) analysis [42], inclusion-based (*i.e.*, Andersen-style) points-to analysis [3] is recognized as the most precise approximation. A recent paper [5] concludes that "while better algorithms for the precise flow-insensitive analysis are still of theoretical interest, their practical impact for C programs is likely to be negligible."

Traditional inclusion-based points-to analysis has been formulated as a dynamic transitive closure problem, with a cubic time complexity in the worst-case [15, 39]. Over the last decade, many enhancements have been proposed to scale the inclusion-based pointer analysis [11, 21, 35], for instance, cycle elimination [13], projection merging [43], improved dynamic transitive closure algorithms [15, 17], and using better data structures for points-to sets [47]. Points-to analysis is recognized as a natural approach to alias analysis because aliasing relationship can be decided by consulting the points-to sets of any pairs of variables [18].

5.3 Alias Analysis

The goal of alias analysis is to decide if two pointer variables may point to the same memory location during program execution. The problem is first formulated by Choi *et al.* [9] and Landi and Ryder [25]. Most state-of-the-art alias analyses have been formulated as a CFL-reachability problem on edge-labeled graphs [38, 40, 49, 50, 54]. Specially, the CFL-reachability-based analyses do not need a points-to analysis to obtain the points-to sets first.

Our alias analysis algorithm is based on an existing CFL-reachability formulation on PEGs, with precision equivalent to an inclusion-based pointer analysis [54]. The scalability of CFL-reachability-based analyses has also been an important, extensively studied problem. All of the aforementioned state-of-the-art alias analyses are *demand-driven*, solving the *single-source-single-sink* CFL-reachability problem. Our alias analysis algorithm solves the *all-pairs* CFL-reachability problem and scales to large, real-world applications.

6. Conclusion

In this paper, we have presented a scalable and efficient subcubic alias analysis for C. We have also presented the first study that reports the performance of subcubic CFL-reachability algorithm in practice. To evaluate its scalability,

Statement	Constraint	Name
`p = &q`	$loc(q) \in pt(p)$	[AddrOf]
`p = q`	$pt(q) \subseteq pt(p)$	[Copy]
`*p = q`	$\forall a \in pt(p).pt(q) \subseteq pt(a)$	[Store]
`p = *q`	$\forall a \in pt(q).pt(a) \subseteq pt(p)$	[Load]

Figure 14. Constraints for flow-insensitive inclusion-based points-to analysis.

we have conducted extensive experiments on recent stable releases of the most popular C programs from the pointer analysis literature. Our results show that our proposed CFL-reachability-based alias analysis scales extremely well.

Acknowledgments

We would like to thank the anonymous reviewers for helpful comments on earlier drafts of this paper. The work described in this paper was supported by RGC GRF grant RGC621912, the HKUST RFID Center, and United States NSF Grants 1117603, 1319187, and 1349528.

A. Alias Analysis and CFL-Reachability

A.1 Alias Analysis

Given a program, the goal of alias analysis is to determine which pairs of pointer variables may refer to the same memory location [20]. The general approach to alias analysis is to adopt a points-to analysis to compute the set of variables that a pointer may point to, and then determine whether their points-to sets intersect [18]. Unfortunately, precise pointer analysis is well-known as a computationally hard problem [6, 24, 29]. When discarding the impact of control-flow dependencies (*i.e.*, flow-sensitivity) and procedure calls (*i.e.*, context-sensitivity), the precise analysis problem is still **NP**-hard [20]. Any practical pointer analysis must approximate the precise solution.

Inclusion-based (*i.e.*, Andersen-style) points-to analysis [3] is commonly recognized as the most precise flow- and context-insensitive approximation [5]. It is usually formulated as a dynamic transitive closure problem on constraint graphs [13, 15, 17]. A simple pass through the input program generates four kinds of inclusion constraints shown in Figure 14. The first constraint concerns the address-taken variables while the other three manipulate only pointer variables. Specially, multiple uses of dereferences are replaced with a sequence of Store and Load statements by introducing new temporaries. For example, a statement like `**p = q` is normalized into `*p = temp` and `*temp = q`. For a variable v, $loc(v)$ represents the memory location denoted by v and $pt(v)$ represents its points-to set. The points-to analysis problem is solved by computing the dynamic transitive closure of a constraint graph, where nodes represent pointer variables and edges represent inclusions.

A.2 CFL-Reachability

The CFL-reachability problem is to determine whether there is an S-path from node u to v in G, where S is the start symbol of the given *CFG*. In particular, the CFL-reachability problem has four variants:

(1) *The all-pairs S-path problem:* For every pair of nodes u and v, is there an S-path in G from u to v?

(2) *The single-source S-path problem:* Given a source node u, for all nodes v, is there an S-path in G from u to v?

(3) *The single-target S-path problem:* Given a target node v, for all nodes u, is there an S-path in G from u to v?

(4) *The single-source-single-sink problem:* Given two nodes u and v, is there an S-path in G from u to v?

In the literature, there is a popular dynamic programming algorithm [34, 51] for solving the *all-pairs* CFL-reachability problem. It is described in Algorithm 3, where W denotes a worklist, (u, A, v) denotes the directed edge (u, v) with label $\mathcal{L}(u, v) = A$, and $\textsc{Out}(u, A)$ denotes the set of all outgoing A-edges of u, *i.e.*, $\textsc{Out}(u, A) = \{v \mid (u, A, v)\}$. The main algorithm has two steps: (1) *CFG Normalization*. The underlying *CFG* must be converted to a normal form which is similar to the Chomsky Normal Form. When the grammar is in the normal form, all production rules are of the form $A \rightarrow BC$, $A \rightarrow B$ or $A \rightarrow \varepsilon$, where A is nonterminal, B and C are terminals or nonterminals, and ε denotes the empty string. In this work, we used the normal forms in Figure 7 for traditional CFL-reachability algorithms; and (2) *"Filling in" New Edges*. In order to compute the S-paths, new edges are added to the graph. For example, lines 11-14 describe that for the production rule $A \rightarrow BC$ and edge (i, B, j), all outgoing edges of node j are considered. If there is an outgoing edge (j, C, k), a new summary edge (i, A, k) is added to G if it is not in the current graph. The algorithm terminates if there are no more new edges to be added.

Pointer analysis can be formulated as a CFL-reachability problem. In the seminal paper introducing CFL-reachability, Reps gave a CFL-reachability formulation for the inclusion-based (*i.e.*, Andersen-style) points-to analysis [32]. Many state-of-the-art pointer analyses [38, 40, 49, 50, 54] are formulated using CFL-reachability.

A.3 Complexity Analysis

Both the inclusion-based pointer analysis and CFL-reachability problems have cubic time complexity in the worst case [15, 32, 39]. The inclusion-based pointer analysis works on a constraint graph where each node represents a pointer variable and each edge represents set inclusion. In the worst case, there are $O(n^2)$ inclusions in the graph. In essence, the inclusion-based analysis algorithm computes dynamic transitive closure, which immediately yields its $O(n^3)$ time complexity.

Algorithm 3: CFL-Reachability Algorithm.

```
Input  : Edge-labeled directed graph $G = (V, E)$;
         normalized $CFG = (\Sigma, N, P, S)$;
Output : the set of summary edges;
1  add $E$ to $W$ ;
2  foreach production $A \to \varepsilon \in P$ do
3      foreach node $v \in V$ do
4          if $(v, A, v) \notin G$ then
5              insert $(v, A, v)$ to $G$ and to $W$ ;
6  while $W \neq \emptyset$ do
7      $(i, B, j) \leftarrow$ SELECT-FROM($W$) ;
8      foreach production $A \to B \in P$ do
9          if $(i, A, j) \notin G$ then
10             insert $(i, A, j)$ to $G$ and to $W$ ;
11     foreach production $A \to BC \in P$ do
12         foreach $k \in$ OUT($j, C$) do
13             if $(i, A, k) \notin G$ then
14                 insert $(i, A, k)$ to $G$ and to $W$ ;
15     foreach production $A \to CB \in P$ do
16         foreach $k \in$ IN($i, C$) do
17             if $(k, A, j) \notin G$ then
18                 insert $(k, A, j)$ to $G$ and to $W$ ;
```

The situations for CFL-reachability is similar. The running time of Algorithm 3 is dominated by lines 12 and 16. When each item is removed from the worklist, it takes time $O(n)$ to generate new items. In the worst case, there can be $O(n^2)$ items in the worklist. As as result, the overall algorithm takes time $O(n^3)$ in the worst case.

The worst case complexity of both problems is difficult to improve. Only recently, Chaudhuri shows that the well-known Four Russians' Trick [4] can be employed on lines 12-13 and lines 16-17 in the CFL-reachability algorithm to yield a subcubic algorithm with an $O(n^3/\log n)$ time complexity [8].

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[3] L. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[4] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economic construction of the transitive closure of a directed graph. *Soviet Mathematics Doklady*, 11:1209–1210, 1970.

[5] S. Blackshear, B.-Y. E. Chang, S. Sankaranarayanan, and M. Sridharan. The flow-insensitive precision of Andersen's analysis in practice. In *SAS*, pages 60–76, 2011.

[6] V. T. Chakaravarthy. New results on the computability and complexity of points-to analysis. In *POPL*, pages 115–125, 2003.

[7] T. M. Chan. All-pairs shortest paths for unweighted undirected graphs in *o(mn)* time. In *SODA*, pages 514–523, 2006.

[8] S. Chaudhuri. Subcubic algorithms for recursive state machines. In *POPL*, pages 159–169, 2008.

[9] J.-D. Choi, M. G. Burke, and P. R. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245, 1993.

[10] J. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[11] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI*, pages 35–46, 2000.

[12] C. Earl, I. Sergey, M. Might, and D. V. Horn. Introspective pushdown analysis of higher-order programs. In *ICFP*, pages 177–188, 2012.

[13] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI*, pages 85–96, 1998.

[14] B. Hardekopf. personal communication, 2012.

[15] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299, 2007.

[16] N. Heintze and D. A. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *LICS*, pages 342–351, 1997.

[17] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *PLDI*, pages 254–263, 2001.

[18] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61, 2001.

[19] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley-Longman, 2001.

[20] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, 1997.

[21] V. Kahlon. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI*, pages 249–259, 2008.

[22] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *PLDI*, pages 207–218, 2004.

[23] D. Kozen. *Automata and computability*. Undergraduate texts in computer science. Springer, 1997.

[24] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *POPL*, pages 93–103, 1991.

[25] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *PLDI*, pages 235–248, 1992.

[26] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI*, pages 308–319, 2006.

[27] F. M. Q. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO*, pages 126–135, 2009.

[28] P. Pratikakis, J. S. Foster, and M. Hicks. Existential label flow inference via cfl reachability. In *SAS*, pages 88–106, 2006.

[29] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[30] J. Rehof and M. Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *POPL*, pages 54–66, 2001.

[31] T. W. Reps. Shape analysis as a generalized path problem. In *PEPM*, pages 1–11, 1995.

[32] T. W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998.

[33] T. W. Reps, S. Horwitz, S. Sagiv, and G. Rosay. Speeding up slicing. In *SIGSOFT FSE*, pages 11–20, 1994.

[34] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[35] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *PLDI*, pages 47–56, 2000.

[36] W. Rytter. Fast recognition of pushdown automaton and context-free languages. *Information and Control*, 67(1-3): 12–22, 1985.

[37] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO*, pages 264–274, 2012.

[38] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[39] M. Sridharan and S. J. Fink. The complexity of andersen's analysis in practice. In *SAS*, pages 205–221, 2009.

[40] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *OOPSLA*, pages 59–76, 2005.

[41] M. Sridharan, S. J. Fink, and R. Bodík. Thin slicing. In *PLDI*, pages 112–122, 2007.

[42] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.

[43] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *POPL*, pages 81–95, 2000.

[44] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *PLDI*, pages 87–97, 2009.

[45] V. Vassilevska. Efficient algorithms for clique problems. *Information Processing Letters*, 109(4):254–257, 2009.

[46] H. S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[47] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

[48] X. Xiao, Q. Zhang, J. Zhou, and C. Zhang. Persistent pointer information. In *PLDI*, pages 463–474, 2014.

[49] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*, pages 98–122, 2009.

[50] D. Yan, G. H. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA*, pages 155–165, 2011.

[51] M. Yannakakis. Graph-theoretic methods in database theory. In *PODS*, pages 230–242, 1990.

[52] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *PLDI*, pages 435–446, 2013.

[53] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *SIGSOFT FSE*, pages 81–92, 1996.

[54] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *POPL*, pages 197–208, 2008.