Conditional Dyck-CFL Reachability Analysis for Complete and Efficient Library Summarization

Hao Tang[1], Di Wang[1], Yingfei Xiong[1](✉), Lingming Zhang[2], Xiaoyin Wang[3], and Lu Zhang[1]

[1] Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University, Beijing, China
{tanghaoth90,wayne.wangdi,xiongyf,zhanglucs}@pku.edu.cn
[2] Department of Computer Science, University of Texas at Dallas, TX, USA
lingming.zhang@utdallas.edu
[3] Department of Computer Science, University of Texas at San Antonio, TX, USA
xiaoyin.wang@utsa.edu

Abstract. Library summarization is an effective way to accelerate the analysis of client code. However, information about the client is unknown at the library summarization, preventing complete summarization of the library. An existing approach utilizes tree-adjoining languages (TALs) to provide conditional summaries, enabling the summarization of a library under certain premises. However, the use of TAL imposes several problems, preventing a complete summarization of a library and reducing the efficiency of the analysis.

In this paper we propose a new conditional summarization technique based on the context-free language (CFL) reachability analysis. Our technique overcomes the above two limitations of TAL, and is more accessible since CFL reachability is much more efficient and widely-used than TAL reachability. Furthermore, to overcome the high cost from premise combination, we also provide a technique to confine the number of premises while maintaining full summarization of the library.

We empirically compared our approach with the state-of-art TAL conditional summarization technique on 12 Java benchmark subjects from the SPECjvm2008 benchmark suite. The results demonstrate that our approach is able to significantly outperform TAL on both efficiency and precision.

1 Introduction

Building a summary for a library is a key technique for scaling static analysis of the library's client programs [7,3,17]. Such a summary can significantly boost client analysis, since client analysis can directly utilize the summary without further analyzing the library. However, in most analysis, it is not possible to treat library code as a complete program during the summarization, because many components required for the analysis are unknown without the presence of the client. For example, when a library calls a virtual method, the actual

callee may depend on the client code. Furthermore, if the callee is a call-back method, the body of the callee also depends on the client code.

Typical techniques (e.g., Rountev et al. [31,33], Lattner et al.[13], Madhavan et al. [17], and Arzt et al. [1]) for dealing with unknown in library summarization are based on distinguishing the known part from the unknown part, and building summaries only for the known part. These techniques are based on the principle of component level analysis (CLA) [31,33]. However, since the unknown components are often required in critical steps of the analysis, the summaries we can build for libraries are significantly limited. For example, in Java, any method that is not declared as final or private can be overridden by a sub class, and thus we cannot statically determine the target of most calls. As a result, in a major portion of a library, we can build summaries for only intra-procedural analysis, and postpone the more expensive inter-procedure analysis to the client analysis. We refer to these techniques as *unconditional summarization*, in contrast to the conditional summarization techniques discussed below.

To overcome the limitation in unconditional summarization, a recent approach by Tang et al. [39] provides *conditional summaries* for data dependency analysis, based on the tree-adjoining language (TAL) reachability analysis. We shall refer to this technique as TALCRA (TAL Conditional Reachabiltiy Analysis). The basic idea is to assume all possibilities of each unknown component, where each possibility is called a *premise*, and pre-compute a conditional summary for all possible clients. When the client code is available, TALCRA obtains the previously unknown component and instantiate the conditional summary into an unconditional summary. In this way, TALCRA can obtain a more complete summary than unconditional summarization techniques can.

However, TALCRA has several limitations. First, by nature, TAL reachability analysis may assume premises that would not exist, and thus is computationally more expensive than other analysis techniques such as context-free language (CFL) reachability [27]. Second, due to the expressiveness of TAL reachability, there can be only one premise for each conditional relation, and the premise cannot cross method boundaries. Thus, in the cases where there are more than one components, or the unknown component crosses multiple methods, TALCRA cannot build a complete summary for the library.

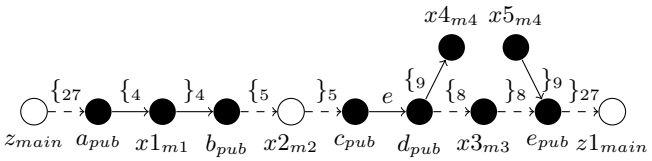
To understand these problems concretely, let us consider an example program in Figure 1a. A dependency graph for this program is shown in Figure 1b. In this figure, nodes are variables and edges are flows-to relations between the variables, i.e., the inverse of dependency relations. The solid nodes and edges can be deduced from the library alone, while the hollow nodes and dashed edges require client code. Using unconditional summarization techniques, we are able to infer only the following relations:

$$a_{pub} \text{ flows to } b_{pub}; \tag{1}$$

$$c_{pub} \text{ flows to } d_{pub}. \tag{2}$$

```
1 package library;
2 public abstract class
      LibraryClass {
3   public final int pub(int a){
4     int b = m1(a);
5     int c = m2(b);
6     int d = c;
7     int e;
8     if (...) e = m3(d);
9     else e = m4(d);
10    return e; }
11  private final int m1(int x1){
12    return x1; }
13  abstract protected int
        m2(int x2);
14  protected int m3(int x3){
15    return x3; }
16  private final int m4(int x4){
17    int x5 = 100; return x5; }
18 }
19
20 package client;
21 class ClientClass extends
       library.LibraryClass {
22   protected int m2(int x2){
23     return x2; }
24   public static void main(){
25     ClientClass client = new
         ClientClass();
26     int z = 255;
27     int z1 = client.pub(z); }
28 }
```

(a) Example Program



(b) Data-Dependency Graph

Fig. 1: A Data Dependency Example

Using TALCRA, we can further infer the following conditional relations:

$$a_{pub} \text{ flows to } c_{pub} \text{ if } b_{pub} \text{ flows to } c_{pub}; \tag{3}$$

$$c_{pub} \text{ flows to } e_{pub} \text{ if } d_{pub} \text{ flows to } e_{pub}. \tag{4}$$

However, TALCRA cannot infer the following two relations, both needed to fully summarize the library.

a_{pub} flows to e_{pub} if b_{pub} flows to c_{pub} and d_{pub} flows to e_{pub}, because TAL summarization does not support more than one premises. (5)

c_{pub} flows to e_{pub} if line 8 calls the method defined at line 14, because TAL summarization does not allow premises to cross method boundaries, and these premises are in fact two edges between nodes in different methods ($d_{pub} \xrightarrow{\{_8} x3_{m3}$ and $x3_{m3} \xrightarrow{\}_8} e_{pub}$); (6)

As a result, to build a complete summary of the library, TALCRA has to make the most conservative assumption, possibly reducing the precision of the analysis. For example, to build a summary for the library, we can use class hierarchy analysis (CHA) [5] to generate unconditional virtual call edges. The result from CHA is guaranteed to cover any client, but is not precise for the analysis of one client, where we can use more precise control flow analysis.

Also, TALCRA infers unnecessary conditional relations during library summarization, such as the following one. This relation is useless because its premise

can never be satisfied, but TAL cannot utilize this fact and may further deduce more useless relations based on this. This is because TAL reachabilty analysis views a conditional reachability relation from the known component by nature, and would treat any two separated paths as potentially connectable.

$$d_{pub} \text{ flows to } e_{pub} \text{ if } x4_{m4} \text{ flows to } x5_{m4}; \tag{7}$$

In this paper, we propose a novel approach to overcoming the limitations in TALCRA, called *ConCRA* (Conditional Dyck-CFL Reachability Analysis). Unlike TALCRA that relies on expensive TAL reachability analysis, our approach is built upon the well-known Dyck-context-free-language (Dyck-CFL) reachability analysis. CFL reachability analysis [27] is known to be applicable to a large class of program analysis problems, and the Dyck-CFL reachability problem is known to be able to express "almost all the applications of CFL reachability" [10]. Therefore, our approach is applicable to a large class of program analysis problems besides data dependency analysis.

The key idea of our approach is to attach premises to standard edges, and analyze using standard CFL rules by assuming the existence of premises. In this way, we can overcome the limitations in TALCRA. First, as we start from the unknown component, in contrast to TALCRA that starts from the known component, we can enumerate only the premises that may exist in some clients, avoiding the high computation cost of producing unnecessary conditional reachability relations. Second, as the premises are basically an attachment, there is no particular constraint over the premises and multiple premises per one relation is also supported. In the above example, our approach can produce the following conditional relations, which completely summarize the flows-to behavior in the library.

$$a_{pub} \text{ flows to } e_{pub} \text{ if } b_{pub} \text{ flows to } c_{pub} \text{ and } d_{pub} \text{ flows to } e_{pub}. \tag{8}$$

$$a_{pub} \text{ flows to } e_{pub} \text{ if } b_{pub} \text{ flows to } c_{pub} \text{ and line 8 calls the method defined at line 14 (i.e., edges } d_{pub} \xrightarrow{\{_8} x3_{m3} \text{ and } x3_{m3} \xrightarrow{\}_8} e_{pub} \text{ exist).} \tag{9}$$

However, allowing too many premises in one conditional relation may lead to too many conditional relations due to the combinatorial effect of the premises. We further propose to confine the number of premises in a reachability relation by introducing *bridging edges*. In this way, our approach can still achieve complete summaries with at most k premises in a relation. We denote the approach with at most k premises as ConCRA-k and the approach with any number of premises as ConCRA-f.

To evaluate the effectiveness and efficiency of ConCRA, we implemented a context-sensitive, SSA-based, and field-insensitive data dependency analysis tool based on ConCRA. In particular, our approach was empirically compared with TALCRA technique on 12 Java benchmark subjects from the SPECjvm2008 benchmark suite. Because of the configurable nature of our approach, we also compare ConCRA-f and ConCRA-k with $k \in \{1, 5\}$.

The evaluation has several findings. (1) ConCRA is able to significantly outperform TALCRA, with up to 1.93X speedup for the library analysis and 5.04X

speedup for the client analysis (46.45X if compared to standard CFL technique). (2) When computing complete summaries, ConCRA is up to 24.7% (100% vs. 80.2%) more precise than TALCRA, where the latter relies on CHA to compute control flow analysis on library when the client information is not available. (3) By balancing the library summarization time and the client analysis time, ConCRA-1 seems to be the overall best configuration for the dependency analysis used in our evaluation.

To sum up, the paper makes the following main contributions:

- A general extension to the well-known Dyck-CFL reachability analysis for conditional reachability analysis, providing better efficiency and more complete library summarization than the state-of-the-art TALCRA approach.
- An efficient technique for confining the number of premises to avoid combinatorial explosion of premises.
- An empirical evaluation demonstrating the superiority of our techniques over the state-of-the-art TALCRA approach.

We organize the rest of this paper as follows. Section 2 presents the technical background. Section 3 presents our approach to conditional reachability analysis. Section 4 presents an experimental evaluation of the proposed approach. Section 5 discusses deeper issues of our research. Section 6 discusses existing research related to ours. Section 7 concludes this paper.

2 Background

Our approach is designed for the Dyck-CFL reachability problem. CFL reachability is a generalization of a large class of program analysis problems. CFL reachability concerns about the reachability between nodes on a graph. The nodes of the graph are usually the values under analysis at different program points, and the edges usually show possible reachability or dependencies between values. To increase the precision of the analysis, a CFL is used to confine the reachability. The edges are labelled with members of an alphabet Σ, and a node is considered to be CFL reachable to another node if and only if the labels on a path between the two nodes form a word in this language.

A frequently used CFL is the Dyck language, which is defined by the following grammars.

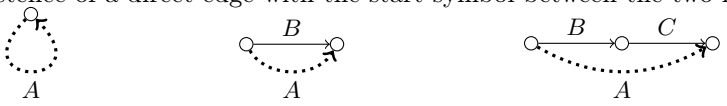
$$
\begin{aligned}
S &\rightarrow \{_i \ S \ \}_i \mid S \ S \mid e \mid \epsilon \\
L &\rightarrow L \ L \mid S \mid \ \{_i \\
R &\rightarrow R \ R \mid S \mid \ \}_i \\
D &\rightarrow R \ D \mid D \ L \mid S
\end{aligned}
$$

Basically, the Dyck language consists of a family of parentheses, which must be matched when paired. In the above definition, $\{_i$ and $\}_i$ are a family of parenthesis, and e is a terminal to be put on edges that are not parentheses. According to the nature of the problem, the start symbol could be one of S, L, R

or D, which allows no unpaired parentheses, only unpaired left parentheses, only unpaired right parentheses, or both.

The Dyck language captures the call-return relationship between procedures. When a procedure is called, the edges from the caller to the callee are labelled with $\{_i$, where i is the identifier of this call site. When the callee returns, the edges from the callee to the caller are labelled with $\}_i$. An example of Dyck-CFL reachability analysis is the data dependency analysis that we have seen in Section 1.

The CFL reachability problem can be solved by a dynamic programming algorithm that adds edges of nonterminals to the graph. The context free grammar is first normalized so that the right hand side of each production has at most two symbols. For example, $S \rightarrow \{_i\ S\ \}_i$ is normalized into $S \rightarrow \{_i\ P_i$ and $P_i \rightarrow S\ \}_i$. Then the three rules in Figure 2 are applied on the graph to add new edges on the graph. The solid lines are existing edges and the dotted lines are the added edges. The three rules can be exhaustively applied with a proper worklist algorithm to achieve a complexity of $O(l^3n^3)$ where l is the number of nonterminal symbols and n is the number of nodes in the graph. When all rules have been thoroughly applied, the reachability between two nodes is equal to the existence of a direct edge with the start symbol between the two nodes.



(a) $A \rightarrow \epsilon$ (b) $A \rightarrow B$ (c) $A \rightarrow B\ C$

Fig. 2: Rules for solving a CFL reachability problem

3 Approach

As mentioned in the introduction, our approach is based on conditional reachability. The core concept to implement conditional reachability is the *conditional edge*, which is a special edge whose existence depends on the existence of a set of other edges (known as *premises*). As a result, our summary for a library is a set of conditional edges, where the premises capture the potential reachability built by client code, and the conditional edges themselves capture the reachability between the boundary nodes of the library. When the client code is available, we can instantiate the conditional edges by analyzing the client code, which directly gives us the reachability between the boundary nodes of the library without analyzing the library code.

In the following sub-sections, we first present the basic definitions, then show how we analyze the client with the conditional edges, and finally show our two algorithms, i.e., ConCRA-f and ConCRA-k. The data-dependency graph in Figure 1b will be used to make our description more clearly.

3.1 Definitions

Suppose an alphabet Σ containing all symbols in a Dyck language. We start by defining general program graphs.

Definition 1 (Program Graph). *A program graph G is a pair (V, E) where V is a set of nodes, E is a set of directed edges between the nodes in V, labelled with the symbols in Σ. We use $G.V$ and $G.E$ to denote the nodes and edges of G, respectively. We use $s \xrightarrow{A} t$ to denote an edge e from s to t with label A, and use $e.src$, $e.tgt$ and $e.tag$ to denote s, t, A.*

Since our approach summarizes library code for client analysis, we need to define library graphs. The difference between a library graph and a program graph is that a library graph has a set of boundary nodes for interacting with the clients, and there are a set of premises that may be instantiated by a client.

Definition 2 (Library Graph). *A **library graph** G is a program graph (V, E) with the following additional components.*

- $\boldsymbol{V_{input}}$ *(Input to library code): The nodes in $G.V$ that can be connected from a client node via a $\{_i$-edge (e.g., a_{pub}).*
- $\boldsymbol{V_{output}}$ *(Output from library code): The nodes in $G.V$ that can connect to a client node via a $\}_i$-edge (e.g., e_{pub}).*
- $\boldsymbol{V_{call}}$ *(Input to call-backs). The nodes in $G.V$ that can be connected to a client node via a $\{_i$-edge (e.g., b_{pub}, d_{pub}).*
- $\boldsymbol{V_{return}}$ *(Output from call-backs). The nodes in $G.V$ that can be connected from a client node via a $\}_i$-edge (e.g., c_{pub}, e_{pub}).*
- $\boldsymbol{P}$ *(Premises). Premises are edges that can potentially be created using the information from a client, $P \cap G.E = \emptyset$.*

We call the set of nodes $V_{input} \cup V_{output} \cup V_{call} \cup V_{return}$ boundary nodes. Correspondingly, the rest of the nodes are called *inner nodes*. We further use $V_{entries} = V_{input} \cup V_{return}$ to denote all incoming boundary nodes and use $V_{exits} = V_{output} \cup V_{call}$ to denote all outgoing boundary nodes.

When the client code is available, the part of the graph representing the client code is added to the library graph, forming an *application graph*.

Definition 3 (Application Graph). *Let G be a library graph. Program graph G' is an **application graph** using G if and only if G' satisfies the following constraints:*

*1. $G.V \subset G'.V \wedge G.E \subseteq G'.E$. We call $G.V$ **library nodes**, denoted as $G'.V_{lib}$, and the nodes in $G'.V - G.V$ as **client nodes**, denoted as $G'.V_{clt}$.*

2. Any edge $s \xrightarrow{A} t$ in $G'.E - G.E$ satisfies one of the following conditions.

*(1) **Edges between client nodes**. $s \in G'.V_{clt} \wedge t \in G'.V_{clt}$.*

*(2) **Edges between library nodes**. $s \in G.V$, $t \in G.V$, and $s \xrightarrow{A} t \in G.P$.*

*(3) **Input edges to the library**. $s \in G'.V_{clt}$, $t \in G.V_{input}$, A is $\{_i$ for some i, and any edge labelled with $\}_i$ is an output edge from the library (e.g., $z_{main} \xrightarrow{\{_{27}} a_{pub}$).*

(4) ***Output edges from the library***. *$s \in G.V_{output}$, $t \in G'.V_{clt}$, A is $\}_i$ for some i, and any edge labelled with $\{_i$ is an input edge to the library (e.g., $e_{pub} \xrightarrow{\}_{27}} z1_{main}$).*

(5) ***Input edges to call-backs***. *$s \in G.V_{call}$, $t \in G'.V_{clt}$, A is $\{_i$ for some i, and any edge labelled with $\}_i$ is an output edge from call-backs (e.g., $b_{pub} \xrightarrow{\{_5} x2_{m2}$).*

(6) ***Output edges from call-backs***. *$s \in G'.V_{clt}$, $t \in G.V_{return}$, A is $\}_i$ for some i, and any edge labelled with $\{_i$ is an input edge to call-backs (e.g., $x2_{m2} \xrightarrow{\}_5} c_{pub}$).*

3. For an input edge $s_1 \xrightarrow{\{_i} t_1$ to call-backs and an output edge $s_2 \xrightarrow{\}_i} t_2$ from call-backs, there is a premise $s_1 \xrightarrow{S} t_2 \in G.P$.

The premises P should satisfy constraints 2(2) and 3. Let us consider data-dependency library graphs. We notice that only values on virtual call sites (e.g., b_{pub}, c_{pub}, d_{pub}, e_{pub}) are related to these constraints. For each parameter and the return value of a virtual call site, we create a premise with label S between them (e.g., $b_{pub} \xrightarrow{S} c_{pub}$, $d_{pub} \xrightarrow{S} e_{pub}$) to satisfy constraint 3. For each virtual call site and any of its potential targets, we create premises between their parameters and return values (e.g., $d_{pub} \xrightarrow{\{_8} x3_{m3}$, $x3_{m3} \xrightarrow{\}_8} e_{pub}$) to satisfy constraint 2(2). We refer to these premises as *call-back premises* and *virtual-call premises*.

Since we are only concerned with the reachability between client nodes, we only need to summarize the reachability between the boundary nodes. As a result, the conditional reachability is defined as the reachability between two boundary nodes under the condition that some pairs of boundary nodes are reachable. The conditional edge is the key concept to implement conditional reachability.

Definition 4 (Conditional Edge). *A conditional edge e is a four-tuple, (A, X, s, t), denoted as $s \xrightarrow{A|_X} t$, where A is a nonterminal, X is a subset of the premises P.*

Given a conditional edge $s \xrightarrow{A|_X} t$, when all edges in X are present on the graph, we consider the premises of this conditional edge as satisfied, and instantiate this conditional edge by putting a normal edge $s \xrightarrow{A} t$ on the graph (e.g., $a_{pub} \xrightarrow{S|_X} e_{pub}$ (X=$\{b_{pub} \xrightarrow{S} c_{pub}$,$d_{pub} \xrightarrow{\{_8} x3_{m3}$,$x3_{m3} \xrightarrow{\}_8} e_{pub}\}$)). Note that the premises of a conditional edge can also be an empty set, e.g., $s \xrightarrow{A|_\emptyset} t$. In such cases a conditional edge is the same as a normal unconditional edge. In this paper, we do not distinguish a normal edge and a conditional edge with zero premise, and refer to them both as *unconditional edges* (e.g., $a_{pub} \xrightarrow{S|_\emptyset} b_{pub}$ and $a_{pub} \xrightarrow{S} b_{pub}$ are the same unconditional edge).

With conditional edges, we can summarize a library as a set of conditional edges, where the premises of these edges represent the potential reachability relationships between the boundary nodes.

In some cases, two conditional edges may have the subsumption relationship.

Definition 5 (Edge Subsumption). $(s_1 \xrightarrow{A|_X} t_1) \sqsupseteq (s_2 \xrightarrow{B|_Y} t_2)$ *if and only if* $A = B$, $X \subseteq Y$, $s_1 = s_2$ *and* $t_1 = t_2$.

This represents that the latter edge requires additional premises compared with the former edge, and thus is completely subsumed by the former edge and can be removed from the graph.

3.2 Analyzing Libraries with ConCRA-f

We summarize the library by adding conditional edges to the graph in a way similar to the standard CFL reachability analysis. Figure 3 depicts the rules for generating conditional edges. Initially, the algorithm properly handles $G.E$ and $G.P$ as initial conditional edges. The algorithm regards each original edge $s \xrightarrow{A} t$ in $G.E$ as $s \xrightarrow{A|_\emptyset} t$. The additional rule (d) handles each edge $s \xrightarrow{A} t$ in $G.P$ by adding $s \xrightarrow{A|_{\{s \xrightarrow{A} t\}}} t$ (a conditional edge that depends on exactly the same edge) to the graph. These initial conditional edges with zero or one premise are served as the starting points to generate conditional edges with more premises. Rules (a), (b), and (c) correspond to the three standard rules in Figure 2 to generate a conditional edge $s \xrightarrow{A|_X} t$. The algorithm handles A, s, and t by the standard rules. The only difference is that the algorithm sets X as the union of the premises of the concatenated conditional edges. Rules (a), (b), and (c) are the cases for zero, one, and two concatenated conditional edges, respectively. Note that the algorithm does not limit the number of premises in a conditional edge. Therefore, we denote the algorithm as ConCRA-f (The "f" means "full").

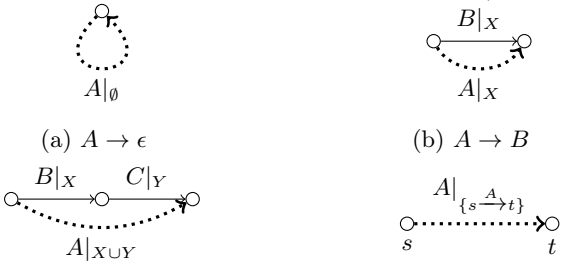


Fig. 3: Rules for building conditional edges in ConCRA-f

We exhaustively apply these rules until no more edges can be added. Then we locate each (un)conditional edge connecting two boundary nodes, i.e., $e.src$ and $e.tgt$ are both in $V_{entries} \cup V_{exits}$. The set of such edges is the summary of the library. In this way we capture all reachability relationships between boundary nodes under all possible premises.

While this basic approach works, we can further optimize it in the following ways:

– In complex situations, two conditional edges e_1 and e_2, where e_1 subsumes e_2, may both be added to the graph. In such cases, we can safely remove e_2 from the graph.
– We can sort the edges in the worklist by the number of premises, so that the edges with fewer premises are created first. In this way we can ensure the subsumed edges are removed once created.

These optimizations lead to Algorithm 1. Algorithm 1 is a worklist algorithm that implements the four rules in Figure 3 with the optimizations mentioned above. This algorithm maintains a worklist of edges W, iteratively adds each edge in W to the graph (by storing to $G.E$), and checks whether any new edge can be generated based on the current edge. Finally, the algorithm returns a set of conditional edges between boundary nodes. The first optimization is applied in line 11, where we process an edge only when there is no subsuming edge on the graph. Furthermore, the worklist W is also a priority queue on the number of the premises, ensuring that the edges with smaller number of premises are added first, implementing the second optimization.

Complexity. Let us denote the total number of premises as m, the number of symbols as l, and the number of nodes as n. Since each edge may be labelled with one of the $O(l)$ symbols and one of the $O(2^m)$ possible premises, the number of edges in a graph is $O(2^m n^2 l)$. In each iteration, we need to check the applicability of rules (a), (b), (c) for one edge and each production rule. Checking applicability of rules (a) and (b) on one edge has $O(1)$ complexity, and for rule (c), we may need to look at $O(2^m nl)$ other edges. As a result, we have a complexity of $O(2^{2m} n^3 l^3)$, exponential to the number of premises m.

3.3 Analyzing Clients

Client analysis is performed on the basis of the set of conditional edges computed by library summarization. When the client code is present, we build an application graph for the client. In the building process, we do not have to include all nodes and edges for the library graph, but include only the boundary nodes, the conditional edges between boundary nodes, and needed bridging edges. The last one is needed for the ConCRA-k analysis as explained later.

When the application graph is built, a further filtering can be performed. We remove those boundary nodes that do not connect to any client node by a direct edge. All conditional edges that connect to these nodes are also removed.

Client analysis with conditional edges is the same as the standard CFL reachability analysis using rules in Figure 2, with one additional rule described in Figure 4. This rule captures the instantiation of a conditional edge. When all premises for a conditional edge are present in the graph, we instantiate this conditional edge by adding an unconditional edge to the graph. The worklist algorithm implementing these rules is depicted in Algorithm 2.

ALGORITHM 1: Analyzing libraries with ConCRA-f

Input: Γ, a context free grammar
Input: G, an library graph
Output: R, a set of conditional edges
Data: W, a priority queue of edges, where edges with smaller number of premises has higher priority

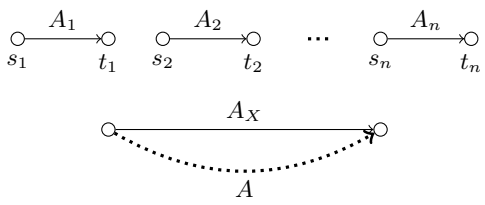
for *each* n *in* $G.V$ **do**
  for *each* $A \to \epsilon$ **do**
    $W \leftarrow W \cup \{n \xrightarrow{A|_\emptyset} n\}$; /* rule (a) */
for *each* $s \xrightarrow{A} t$ *in* $G.P$ **do**
  $x \leftarrow s \xrightarrow{A} t$
  $W \leftarrow W \cup \{s \xrightarrow{A|_{\{x\}}} t\}$; /* rule (d) */
$W \leftarrow W \cup G.E$
$G.E \leftarrow \emptyset$
while W *is not empty* **do**
  $(e : s \xrightarrow{A|_X} t) \leftarrow$ *the first edge in* W
  if $\forall e' \in G.E : \neg(e' \sqsupseteq e)$ **then**
    $G.E \leftarrow G.E \cup \{e\}$
    for *any* $B \to A \in \Gamma$ **do**
      $W \leftarrow W \cup \{s \xrightarrow{B|_X} t\}$; /* rule (b) */
    for *any* $B \to A\ C \in \Gamma \wedge t \xrightarrow{C|_Y} t' \in G.E$ **do**
      $W \leftarrow W \cup \{s \xrightarrow{B|_{X \cup Y}} t'\}$; /* rule (c) */
    for *any* $B \to C\ A \in \Gamma \wedge s' \xrightarrow{C|_Y} s \in G.E$ **do**
      $W \leftarrow W \cup \{s' \xrightarrow{B|_{X \cup Y}} t\}$; /* rule (c) */
  $W \leftarrow W - \{e\}$
for *each* e *in* $G.E$ **do**
  if *e.src and e.tgt are both boundary nodes* **then**
    $R \leftarrow R \cup \{e\}$

3.4 Soundness

Theorem 1 (Soundness of ConCRA-f). *Let G be an arbitrary application graph using library G', analyzing G via the ConCRA-f summary of G' produces exactly the same set of edges between client nodes as analyzing G directly.*

Proof. To prove this theorem, first, we need to show that any path between two client nodes recognized by our ConCRA-f summary together with client analysis will be recognized by the Dyck-CFL. This is easy to prove as our summarization rules in Figure 3 is a direct extension to CFL rules in Figure 2.

Second, we need to show that any path between two client nodes recognized by the Dyck-CFL will be recognized by our ConCRA-f summary together with client analysis. This part is more difficult. Due to space limit, we shall show only how to prove this theorem for edges labelled with S. The other labels L, R, and D can be similarly proved.



(d) if $X = \{s_1 \xrightarrow{A_1} t_1, s_2 \xrightarrow{A_2} t_2, \ldots, s_n \xrightarrow{A_n} t_n\}$

Fig. 4: Additional Rule for Client Analysis

The proof is an induction over the length n of the path to show the following two propositions hold, where the first one directly responds to our theorem: (1) any path with length equal or less than n between two client nodes recognized by Dyck-CFL will be recognized by client analysis; (2) any path with length equal or less than n between two library nodes recognized by Dyck-CFL will be recognized as a conditional edge in library summarization, and its premises will be produced by client analysis.

When $n = 0$, the path can only be produced by rule $S \to \epsilon$, and the two propositions trivially hold.

When $n = 1$, the path can only be produced by rule $S \to e$, or rule $S \to S\ S$ where one S on the righthand side is ϵ, and the two propositions trivially hold.

Suppose the two propositions hold for any length up to k. When length is $k + 1$, the edge can be recognized by either $S \to \{_i\ S_1\ \}_i$ or $S \to S_1\ S_2$. Let us consider $S \to \{_i\ S_1\ \}_i$ first. In the case where S is between two client nodes, S_1 can be a path between two client nodes or two library boundary nodes, but cannot be a path between one client node and a library node because of the pairing of parentheses in Definition 3. Thus, S_1 will be recognized because of induction hypothesis, and then S will be recognized.

In the case where S is between two library nodes, similarly S_1 can be a path between two client nodes or two library nodes. In the former case, S is a path between V_{call} and V_{return}, and thus is a premise itself. Also, S_1 will be recognized by client analysis because of the induction hypothesis, and thus S will be recognized. In the latter case, S_1 will be produced as a conditional edge with all its premises recognizable because of the induction hypothesis, and thus S will be recognized with all its premises recognized.

Then let us consider $S \to S_1\ S_2$. In the case where S is between two client nodes, S_1 and S_2 are also between two client nodes, otherwise the parentheses cannot be balanced. Thus, S_1 and S_2 will both be recognized by induction hypothesis, and then S will be recognized. The case where S is between two library nodes is similar.

Based on the above analysis, the two propositions hold for paths of any length, and thus the theorem holds.

ALGORITHM 2: Analyzing clients

Input: Γ, a context free grammar
Input: G, an application graph
Input: M, a set of conditional edges
Data: W, a worklist of edges

```
1  for each n in G.V do
2  |   for each A → ϵ do
3  |   |   W ← W ∪ {n -A-> n} ;                          /* rule (a) */
4  W ← W ∪ G.E
5  G.E ← ∅
6  while W is not empty do
7  |   (e : s -A-> t) ← an edge in W
8  |   if e ∉ G.E then
9  |   |   G.E ← G.E ∪ {e}
10 |   |   for any B → A ∈ Γ do
11 |   |   |   W ← W ∪ {s -B-> t} ;                      /* rule (b) */
12 |   |   for any B → A C ∈ Γ ∧ t -C-> t' ∈ G.E do
13 |   |   |   W ← W ∪ {s -B-> t'} ;                     /* rule (c) */
14 |   |   for any B → C A ∈ Γ ∧ s' -C-> s ∈ G.E do
15 |   |   |   W ← W ∪ {s' -B-> t} ;                     /* rule (c) */
16 |   |   for any s' -B|X-> t' ∈ M ∧ X ⊆ G.E do
17 |   |   |   W ← W ∪ {s' -B-> t'} ;                    /* rule (d) */
18 |   |   |   M ← M − {s' -B|X-> t'}
19 |   W ← W − {e}
```

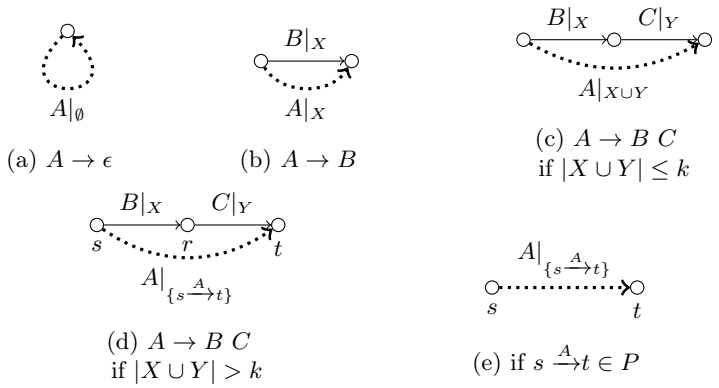
3.5 Analyzing Libraries with ConCRA-*k*

As analyzed before, the time complexity of ConCRA-f is exponential to the number of premises, making it difficult to scale up. This time complexity is caused by the massive number of edges with different premises between two nodes. To avoid this, we can set an upper bound k on the number of the premises of each conditional edge. We denote this technique as ConCRA-k.

Now the problem is how to represent all conditional edges with only k premises. Our idea is to introduce bridging edges to represent conditional edges with more premises using conditional edges with less premises.

This idea leads to the rules in Figure 5. The only difference from the rules in Figure 3 is that the original rule (c) is separated into two rules (c) and (d) in Figure 5. If the generated edge has less than or equal to k premises, the rule is the same as before (rule (c)). If the generated edge requires more than k premises, we reset its premises to one premise by introducing a new premise containing only the edge itself (rule (d)). The edges $s \xrightarrow{B|_X} r$ and $r \xrightarrow{C|_Y} t$ are the bridging edges of $A|_{\{s \xrightarrow{A} t\}}$.

The concrete algorithm is listed in Algorithm 3. This algorithm is mostly the same as Algorithm 1, and we only show the changed lines. The number at



(a) $A \to \epsilon$ (b) $A \to B$ (c) $A \to B\ C$ if $|X \cup Y| \leq k$

(d) $A \to B\ C$ if $|X \cup Y| > k$ (e) if $s \xrightarrow{A} t \in P$

Fig. 5: Rules for building conditional edges in ConCRA-k

the left of each line in Algorithm 3 indicates the corresponding line number in the original algorithm. First, when the premises of rules (c) or (d) are satisfied (line 15 and line 17), we also record any potential dependencies in M if the generated edge has more than k premises. When we add the edge with more than k premises to the graph (line 12), we reset its premise to one. Note that we only reset premise when the edge is about to be added to the graph. In this way we can ensure that line 12 could still filter some subsumed edges. Finally, when the conditional edges between boundary nodes are selected, we perform a backtracking to add back all needed intermediate conditional edges (line 22).

Theorem 2 (Soundness of ConCRA-k). *Let G be an arbitrary application graph using library G', analyzing G via the ConCRA-k summary of G' produces exactly the same set of edges between client nodes as analyzing G directly.*

Proof. The proof of this theorem is similar to Theorem 1. The only difference is that this time we need to reason that the chain of conditional edges produced by rule 5d is complete, and this can be easily seen by examining all rules that introduce new premises.

Complexity. Let us denote the number of premises as m, the number of production rules as l, and the number of nodes as n. There are two differences from ConCRA-f. First, since any newly added edges may have its premises reset, we need to consider $m + n^2 l$ rather than just m. Second, there is only $C^k_{m+n^2 l}$ possible sets of premises. As a result, we have a complexity of $O((C^k_{m+n^2 l})^2 n^3 l^3)$. This complexity is much smaller than ConCRA-f with a small k. For example, we have a polynomial complexity when $k = 1$.

4 Empirical Evaluation

This section empirically evaluates the proposed techniques.

ALGORITHM 3: Analyzing libraries with ConCRA-k

```
...
Output: M, a dictionary from intermediate conditional edges to their
        dependencies
...
11 if ∀e' ∈ G.E : ¬(e' ⊒ e) then
12     if |X| > k then
           x ← (s --A--> t)
           e ← (s --A|{x}--> t)
       G.E ← G.E ∪ {e}
       ...
15     for any B → A C ∈ Γ ∧ t --C|Y--> t' ∈ G.E do
16         W ← W ∪ {s --B|X∪Y--> t'} ;                    /* rule (c), (d) */
           if |X ∪ Y| > k then
               /* append(k, v) adds v to key k                              */
               M.append(s --B--> t', {e, t --C|Y--> t'})
17     for any B → C A ∈ Γ ∧ s' --C|Y--> s ∈ G.E do
18         W ← W ∪ {s' --B|X∪Y--> t} ;                    /* rule (c), (d) */
           if |X ∪ Y| > k then
               M.append(s' --B--> t, {s' --C|Y--> s, e})
       ...
22 R ← AddDependencies(R)
```

4.1 Evaluated Analyses

Our evaluation is based on the same context-sensitive, SSA-based, and field-insensitive dependency analysis used in evaluating TALCRA [39] so that we can compare with TALCRA. More concretely, we track the flows-to relations (the inverse of dependency relations) between stack variables; we treat each variable as a node and an assignment `x=y` as an edge from `y` to `x`; a method call `z=x.f(y)` corresponds to a left-parenthesis edge from `y` to the argument of `x.f()` and a right-parenthesis edge from the return value to `z` (The call site determines the index of the parenthesis). We refer to a parenthesis edge as a call edge. A virtual-call edge is a call edge belonging to a virtual call site. Our example in Figure 1 demonstrates the graph used in this analysis.

To summarize a library for this analysis, we need to provide the set of premises $G.P$. $G.P$ obtains *call-back premises* and *virtual-call premises* as we mentioned in Section 3. Note that TALCRA does not support the latter type of premises, so we also evaluate our techniques on the configuration where each virtual-call edge in the libray is treated as a normal edge. Therefore, the evaluation contains two configurations: (1) *VC-config*: virtual-call edges are reserved as premises in the library graph; (2) *CHA-config*: virtual-call edges are treated as normal edges in the library graph. *CHA-config* library summaries are imprecise because the virtual-call edges generated by imprecise library call graph construction approaches (e.g., class hierachy analysis) are treated as normal edges.

Procedure AddDependencies(E)

```
1  for each s --A|X--> t ∈ E do
2  |  W ← W ∪ X
3  end
4  while W is not empty do
5  |  e ← an edge in W
6  |  if M contains e then
   |  |  /* lookup(e) returns the dependencies of e */
7  |  |  for each e' = s' --A'|X'--> t' ∈ M.lookup(e) do
8  |  |  |  E ← E ∪ {e'}
9  |  |  |  W ← W ∪ X'
10 |  |  end
11 |  end
12 |  W ← W − {e}
13 end
14 return E
```

4.2 Implementation

Our implementation has two parts. The first part, implemented in Java using the SOOT framework[4], generates the dependency graphs and exports the graphs to files. The virtual call sites are resolved by CHA [5] in the library analysis, and are resolved by Spark [14] in the client analysis. The virtual-call premises are those produced by CHA. The second part, implemented in C++, reads the files and performs the library summarization and client analysis of ConCRA, TALCRA, as well as CLA [31]. We obtained the newest TALCRA implementation from TALCRA web site[5]. We reimplemented CLA to use CFL reachability as summary representation, rather than the original functional approach. Bascially, there are two differences with CLA and ConCRA: (1) CLA does not use premises and only summarizes unconditional edges between boundary nodes; (2) Boundary nodes used by CLA include not only boundary nodes used by ConCRA, but also the call-site nodes to procedures that may (transitively) call a virtual call.

We adopt several efficient data structures in library summarization and client analysis. The worklist W for ConCRA-k and ConCRA-f is segregated into several sets $W_1, W_2, \cdots$. All newly-generated conditional edges with i premises are put into W_i. The first edge returned by W is thus always an edge in the non-empty W_i with the smallest i. Each conditional edge $s \xrightarrow{A|_X} t$ added into $G.E$ is indexed by $\langle A, s\rangle$, $\langle A, t\rangle$ using 2-dimensional arrays, and $\langle A, s, t\rangle$ using a 2-dimensional array of hash tables. The two arrays are used for efficiently acquiring the edges with specific labels and common source or target nodes. The array of hash tables is used to check whether an edge e is subsumed (Definition 5) by other edge e'.

We also implemented a standard whole-program CFL reachability analysis as a control technique.

[4] http://sable.github.io/soot/

[5] http://www.utdallas.edu/\%7elxz144130/tal.html, accessed 2016-01-29.

4.3 Setup

Benchmark. Our benchmark includes all 12 subjects in the SPECjvm2008[6] benchmark. The SPECjvm benchmark was widely used in evaluating the state-of-art library-summarization work [39] as well as many related approaches in the program analysis area [37,36,41,42].

In our evaluation, we treat JDK as library and build summaries for a major portion of JDK. More specifically, we build summaries for two JAR files, `rt.jar` and `jce.jar`, which include most commonly used Java packages, such as `java.util`, `java.io`, `java.lang`, etc. Summarizing a popular portion of the library instead of the whole library is a common practice used in existing evaluation on summarization techniques [1,17,32] to reduce the summarization time.

Table 1 shows the statistics. Columns 1 lists all the benchmark subjects. Columns 2-4 show the numbers of client nodes, the library nodes accessed by the client, and all nodes in each subject's data dependency graph[7]. Similarly, Columns 5-7 present the Jimple code lines of the client, the library methods called by the client, and the whole application for each subject. Here the library nodes and library code refer to only the part of JDK in our summary.

Table 1: Benchmark statistics

	# Nodes			# Jimple Code Lines		
Bench.	Clt	Lib	Total	Clt	Lib	Total
check	1701	10838	12539	7752	160078	167830
compiler	917	10699	11616	4184	160042	164226
compress	1428	10576	12004	5025	160042	165067
crypto	2515	19002	21517	11547	229158	240705
derby	1380	16722	18102	10189	210054	220243
hello	598	10881	11479	847	160042	160889
mpeg	17588	37980	55568	243007	402569	645576
scimark	1557	10709	12266	7034	160042	167076
serial	11509	38419	49928	187517	407990	595507
startup	1083	16512	17595	2472	200060	202532
sunflow	19021	26606	45627	139362	283016	422378
xml	11749	23444	35193	122631	268431	391062
total	71046	232388	303434	741567	2801524	3543091

Table 2 shows the information about summarized part of JDK including the Jimple code lines, the size of the dependency graph, and the two types of premises. Jimple is the fundamental intermediate representation of Java in Soot. We use the lines of Jimple code instead of Java source code because we do not have full Java source code for the benchmark. Based on our experience with

[6] `http://www.spec.org/jvm2008/`

[7] Please note the statistics are different from the TALCRA paper [39] because that evaluation was performed on an early version of the TALCRA tool that built the graph differently.

JDK, the Jimple code lines are about 4 times as many as the original source code, excluding comments and empty lines.

Table 2: Library statistics

Jimple Code Lines	526648
Dependency Graph Nodes	66736
Dependency Graph Edges	161239
Virtual Call Premises	64777
Call-back Premises	10236

Compared techniques. On both *CHA-config* and *VC-config*, we evaluated ConCRA-f and ConCRA-k (with k values between 1 and 5) techniques, as well as a standard CFL-reachability analysis and the CLA technique [31]. Moreover, on the *CHA-config*, we evaluated the state-of-art TALCRA technique [39].

To evaluate the effectiveness of each studied technique, we measured the time cost for each technique in both library summarization and client analysis. Furthermore, we evaluated the precision of client analysis by measuring the produced dependency edges.

Evaluation Platform. Our evaluation was performed on a Dell PowerEdge R730 Server with 8-core 16-thread Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz and 256 Gigabyte RAM running OpenJDK 1.7.0_79 on Ubuntu 14.10.

4.4 ConCRA-f/ConCRA-k vs. TALCRA

On the *CHA-config*, we compare our techniques against TALCRA.

The results of library summarization are shown in Table 3. Columns 1 to 5 list the library summarization statistics of CLA, TALCRA, ConCRA-1, ConCRA-2, and ConCRA-f, respectively. Row 2 shows the summarization time. Row 2 also lists the speedup information compared with the CLA technique in Columns 2 to 5. Row 3 shows the maximal memory usage during the summarization. We omit the results of ConCRA-k ($k = 3, 4, 5$) because they are almost identical to the result of ConCRA-f.

Table 3: Library Summarization (*CHA-config*)

CLA	TALCRA	ConCRA-1	ConCRA-2	ConCRA-f
72.50s	87.78s 0.83X	45.33s 1.60X	63.30s 1.15X	71.79s 1.01X
1.45G	3.66G	5.15G	6.81G	7.39G

The experimental results of client analysis are shown in Table 4. In the table, Column 1 lists all the benchmark subjects. Column 2 presents the time cost for the standard CFL reachability analysis on the entire application graph (including all the client and library nodes) as the control technique. Column 3 presents the client analysis time for the CLA approach. Column 4 presents the client analysis time for the TALCRA approach. Columns 5 to 7 present the corresponding client analysis time for our ConCRA-k ($k = 1, 2$) and ConCRA-f techniques. Columns 4 to 7 also contains the speedup information compared with the standard CFL

reachability analysis. The last row presents the arithmetic mean of speedups on all subjects achieved by these techniques. Again, we omit the results of ConCRA-k ($k = 3, 4, 5$) because the results are very close to ConCRA-f's (less than 8% variation).

Table 4: Client Analysis for *CHA-config* (Run Times in Milliseconds)

Benchmark	CFL	CLA		TALCRA		ConCRA-1		ConCRA-2		ConCRA-f	
check	433	397	1.09X	303	1.43X	82	5.29X	84	5.14X	85	5.09X
compiler	401	380	1.06X	274	1.46X	80	5.01X	80	5.00X	80	5.03X
compress	412	399	1.03X	290	1.42X	98	4.21X	94	4.37X	94	4.40X
crypto	4058	1820	2.23X	468	8.68X	101	40.18X	105	38.70X	102	39.63X
derby	4121	1486	2.77X	430	9.59X	96	42.97X	96	43.11X	94	43.94X
helloworld	400	390	1.02X	270	1.48X	75	5.32X	76	5.27X	76	5.26X
mpegaudio	623527	518671	1.20X	253623	2.46X	154986	4.02X	154384	4.04X	154580	4.03X
scimark	444	417	1.06X	306	1.45X	104	4.26X	106	4.18X	105	4.24X
serial	619523	518384	1.20X	247012	2.51X	151562	4.09X	151937	4.08X	151478	4.09X
startup	3745	1523	2.46X	406	9.21X	81	46.45X	84	44.62X	84	44.39X
sunflow	21675	33741	0.64X	12843	1.69X	5292	4.10X	5290	4.10X	5265	4.12X
xml	9361	7291	1.28X	3566	2.62X	2799	3.34X	2835	3.30X	2801	3.34X
			1.42X		3.67X		14.10X		13.83X		13.96X

Library summarization. To summarize the JDK library, ConCRA-f and ConCRA-k are both faster than TALCRA. Compared with the TALCRA approach, ConCRA-1, ConCRA-2, and ConCRA-f have 1.93X, 1.39X, and 1.22X speed-up, respectively. We can observe that the library summarization time grows when k increases, and the time is close to ConCRA-f when $k > 2$. The reason is that the conditional edges with more than 2 premises in the summary of ConCRA-f only account for 26.4% (9,070,850 out of 34,363,200) of the summary. This finding also explains why our techniques are practical despite the high theoretical complexity upper bound analyzed in Section 3. The overall amount of conditional edges (34,363,200) is reasonable in practice.

Client analysis. Compared to standard CFL technique, all the ConCRA implementations significantly reduce the client analysis time. Across all subjects, ConCRA-1 speeds up the client analysis time by 3.34X to 46.45X, with an arithmetic mean of 14.10X. Our ConCRA implementations also outperform TALCRA with speedups up to 5.04X.

Interestingly, ConCRA-f and ConCRA-k with a large k do not exhibit superiority over ConCRA-1. We suspect that there are two possible reasons: (1) larger k leads to larger summaries, and the memory management time increases significantly; (2) there is only a few number of instantiated conditional edges that has many premises, so the performance boost from larger k is not significant.

4.5 ConCRA-f/ConCRA-k vs. CLA

On the *VC-config*, we compare our techniques against CLA.

The results of library summarization are shown in Table 5, in the same format as Table 3. Columns 1 to 4 list the library summarization statistics of CLA, ConCRA-1, ConCRA-2, and ConCRA-f, respectively. All of ConCRA-k with

$k = 3, 4, 5$ and ConCRA-f fail to build summaries on the *VC-config* within the 2-hour time limit.

Table 5: Library Summarization (*VC-config*)

CLA	ConCRA-1	ConCRA-2	ConCRA-f
0.40s	34.74s 0.01X	1059.61s 0.00X	TimeOut
0.26G	5.34G	32.71G	-

The results of client analysis are shown in Table 6. The meaning of Column 1 to 2 are same as in Table 4. Column 3 presents the client analysis time for the CLA approach. Columns 4 to 5 present the corresponding client analysis time for our ConCRA-k ($k = 1, 2$) techniques.

Table 6: Client Analysis for *VC-config* (Run Times in Milliseconds)

Benchmark	CFL	CLA		ConCRA-1		ConCRA-2	
check	433	142	3.05X	137	3.16X	770	0.56X
compiler	401	139	2.88X	157	2.55X	728	0.55X
compress	412	149	2.77X	225	1.83X	954	0.43X
crypto	4058	1079	3.76X	151	26.94X	356	11.40X
derby	4121	1040	3.96X	120	34.23X	153	26.93X
helloworld	400	137	2.91X	77	5.21X	83	4.80X
mpegaudio	623527	340015	1.83X	167021	3.73X	169673	3.67X
scimark	444	167	2.66X	246	1.81X	986	0.45X
serial	619523	334689	1.85X	165408	3.75X	166616	3.72X
startup	3745	988	3.79X	239	15.70X	2978	1.26X
sunflow	21675	10844	2.00X	4755	4.56X	8183	2.65X
xml	9361	4724	1.98X	3164	2.96X	6496	1.44X
		2.79X		8.87X		4.82X	

Library summarization. ConCRA-1 and ConCRA-2 spends more time than CLA, since ConCRA calculates a more complete summary than CLA does. ConCRA-1 can finish summarization in less than 1 minute. ConCRA-2 spends much more time than ConCRA-1, since the introduction of virtual call premises significantly enlarges the set of premises, causing ConCRA-2 to consider much more potential combinations of premises than ConCRA-1.

Client analysis. Compared to standard CFL technique, ConCRA-1 speeds up the client analysis time significantly with an arithmetic mean of 8.87X. ConCRA-1 also achieves an average speedup about 3 times as high as CLA does. ConCRA-1 is slightly slower than CLA on a few subjects (compiler, compress, and scimark). We can observe these subjects are relatively small, which caused the memory management became bottleneck in the analysis.

ConCRA-2 does not perform as well as ConCRA-1, and on some subjects, it is even slower than the whole-program CFL technique. The summary ConCRA-2 calculates is so large that the overhead of memory management time becomes prominent in client analysis. This finding indicates that ConCRA-1 has the overall best performance on the data dependency analysis in our evaluation.

4.6 Precision

As we analyzed before, techniques on the *VC-config* should be as precise as the standard CFL reachability analysis while techniques on the *CHA-config* may be imprecise due to the missing client information. In our evaluation, we count the number of dependency edges produced by each algorithm, and we also compare the analysis results against each other algorithm.

Table 7 shows client results about precision. Column 2 presents the number of dependencies found by standard CFL reachability analysis on the entire application graph. Column 3 presents the number of dependencies found on the *CHA-config* and its precision compared to CFL results. Column 4 stands for the *VC-config*. The precision is calculated by dividing the number from CFL analysis with the number from the respective analysis.

There are two major findings. First, the dependency relations produced by techniques on the *VC-config* are the same as those produced by the standard CFL analysis, and both are a subset of the dependency relations produced by techniques on the *CHA-config*. This serves as a side evidence that our implementation is correct. Second, the results by techniques on the *CHA-config* are imprecise. On some subjects, the precision can be as low as 80.20%.

Table 7: Analysis Precision

Benchmark	CFL	*CHA-config* (CLA,TALCRA,ConCRA)	*VC-config* (CLA,ConCRA)
check	2012	2012(100.00%)	2012(100.00%)
compiler	764	836(91.39%)	764(100.00%)
compress	9729	9801(99.27%)	9729(100.00%)
crypto	4593	5197(88.38%)	4593(100.00%)
derby	7166	7180(99.81%)	7166(100.00%)
helloworld	329	337(97.63%)	329(100.00%)
mpegaudio	2713173	2793994(97.11%)	2713173(100.00%)
scimark	16028	16100(99.55%)	16028(100.00%)
serial	2479497	2551066(97.19%)	2479497(100.00%)
startup	840	841(99.88%)	840(100.00%)
sunflow	407547	508150(80.20%)	407547(100.00%)
xml	554780	555576(99.86%)	554780(100.00%)

In conclusion, both ConCRA-f and ConCRA-k are able to achieve speedups over the state-of-art TALCRA technique on the *CHA-config*. They significantly outperform traditional CFL technique for client analysis (e.g. ConCRA-1 and ConCRA-f can achieve speedups of up to 46.45X and 44.39X, respectively). They also achieve speedups up to 5.04X compared to TALCRA. In case of the *VC-config*, ConCRA-1 achieves significant speedups over the standard CFL technique for client analysis up to 34.23X, without precision loss. ConCRA-1 also generally outperforms the CLA technique.

5 Discussion

Multiple Library Summaries. For the simplicity of the evaluation, we treat the JDK implementation as the library code and all the other code and third-party libraries of each subject as the client code. However, our approach supports the client code analysis with multiple library summaries. For example, for client code c using two external libraries (l_1 and l_2). The client code analysis of c can be directly built on top of the two separate library summaries by simply collecting all conditional edges. We can even combine the two summaries into a large summary by assuming virtual-call premises between them and apply the library summarization rules.

Field Sensitivity. Following existing work for conditional reachability analysis [39], we also evaluate our approach based on the context-sensitive, flow-sensitive, and field-insensitive data-dependency analysis. Field-sensitivity can also be encoded as a CFL reachability problem. However, achieving both context-sensitive and field-sensitive analysis has been shown to be undecidable [26]. To maintain both sensitivity to some extent, researchers have proposed to use regular language to approximate one CFL and keep the other one complete [37]. Our conditional reachability analysis based on CFL provides a natural way to adapt existing technique to further obtain field sensitivity to some extent. For example, we can regularize the CFL for field sensitivity (i.e., RL_f) and keep the CFL for context sensitivity (i.e., CFL_c). Therefore, data-dependency analysis considering both sensitivity can be approximated as the CFL-reachability problem using $RL_f \cap CFL_c$. Then, we may still use our conditional reachability analysis based on CFL to obtain library summary information to speed up client analysis. Furthermore, in parallel with our work, Zhang and Su [45] recently proposed an efficient algorithm based on linear-conjunctive-language reachability for solving context-sensitive and field-sensitive data dependence analysis. The idea of conditional reachability may also be applied to their approach, which is a future work remaining to be explored.

Heap Objects and Global Variables. A standard way to handle heap objects in dependence graphs is to promote them to the input and output of respective methods by interprocedural mod-ref analysis. Existing tools such as WALA can directly generate such graphs (system depedenence graph with heap paramters). However, this kind of graphs cannot be directly summarized by our approach because heap objects and global variables defined in the clients may need to promote to the library side. A possible solution is to assume the promoted nodes and edges are part of the client graph. First we analyze library without the promoted nodes and edges from the clients. Then when a client is available, we analyze the precise boundary between the library and the client, and turn the library analysis result into a summary. The concrete algorithm is a future work to be explored.

6 Related Work

Our work is mainly related to existing research efforts on conditional analysis, CFL reachability, and software library summarization.

6.1 Conditional Analysis

Our analysis differs from normal reachability analysis because our analysis further takes into account the conditions under which a code element can reach another code element. In this sense, our research is related to existing research efforts on conditional data dependency and information flow. Snelting et al. [35] proposed a technique to extract conditions defined on program input variables that must hold for certain data dependency in the program under analysis. Komondoor and Ramalingam [11] proposed to identify conditional data dependency for recovery of data models in programs written with weakly-typed languages. Sukumaran et al. [38] proposed to extend the program dependency graph with conditions on the edges to specify the corresponding condition of a certain dependency edge. Tschantz and Wing [40] developed a technique that detects not only active but also passive conditional information flows to extract confidentiality policies from software programs. Lochbihler and Snelting [15] further considered data dependency controlled by temporal path conditions. Recently, Jaffar et al. [9] proposed path-sensitive backward slicing that considers path conditions when locating code elements that may affect certain program outputs. The conditions considered in the above research efforts are all path conditions in branch predicates. In contrast, the conditions considered in our approach are reachability relationships between code elements.

Conditional must-not-alias analysis [19] calculates whether a pair of variables must not refer to a same memory location when another pair of variables do not refer to a same memory location. This analysis is first used to detect race conditions [19], and later in accelerating CFL-reachability-based points-to analysis [42]. Similar to conditional must-not-alias analysis, our approach also considers the reachability relationship between variables as conditions. However, since the purpose of our approach is different (i.e., summarizing library code with unknown components), we use as conditions the reachability relationships at the library-client interface that are not available at the time of summarization, and we further consider conditional reachability with multiple premises, neither of which are covered in conditional must-not-alias analysis.

6.2 CFL Reachability

CFL reachability is a general framework developed in the area of database by Yannakakis [43], and Reps et al. [28] first applied the framework to interprocedural slicing. Later, the framework is applied to a series of program analysis tasks, including inter-procedural dataflow analysis [27,23,20], points-to analysis [37,42], alias analysis [47,44], shape analysis [24,30,8], constant propagation [34], label-flow analysis [20], information flow analysis [18,16], race detec-

tion [21], and specification inference [2]. In 1998, Reps [25] wrote a survey on the application of CFL reachability on various program analysis tasks.

Similar to us, the IDE framework [34] also attaches additional information to the graph. This framework is designed to problems such as constant propagation, where the "environment information", such as the values of variables, is attached to each node. Later, Reps et al. [29] proposed a novel data flow analysis framework based on reachability analysis of pushdown automata which allows adding weights to the edges of pushdown automata. Compared to these approaches, our approach attaches conditional information to the edges, and solves the library summarization problem with the conditional edges.

6.3 Library Summarization

The main purpose of our proposed technique is to summarize libraries with consideration of unknown components from client side, which is one of the emerging but not well-solved problem in static analysis. In literature, we notice several existing research efforts that try to address unknown components (e.g., call-backs) when summarizing library code.

Rountev et al.[31,33] proposed a technique to accelerate dataflow analysis by summarizing library code. Madhaven et al.[17] developed a general framework to deal with unknown components in library summarization. Arzt et al. [1] applies Rountev et al.'s technique [33] to taint analysis on Android applications. These approaches identify the part of the library code that is not affected by the unknown components, and build a partial summary for this part of library code. Compared to these approaches, ConCRA is able to generate summaries (i.e., in the form of conditional reachability) for the code affected by unknown components. Lattner et al.[13] proposed a heap-cloning-based approach to context-sensitive summarization of libraries with call-backs for pointer analysis. This approach is in principle similar to Rountev et al.[31,33] and Madhaven et al.[17], but is specifically tuned for pointer analysis. Furthermore, their approach is tightly coupled with the problem of pointer analysis, and cannot be easily migrated to other problems, whereas ConCRA is able to be applied on a large class of problem using Dyck-CFL reachability analysis.

Other research efforts eliminate unknown components by make the most conservative assumptions on them. Ravitch et al. [22] developed a technique to automatically generate bindings for inter-programming language function calls. Bastani et al. [2] deals with a related but different problem. Instead of building library summaries for client analysis, they deal with the case where the library code is missing, and try to infer a specification of the library code for manual revision. Das et al. [4] proposed angelic verification to handle unknown external function calls in program verification.

The work most closely related to our approach is by Tang et al.[39]. As mentioned before, they proposed the TAL (Tree Adjoining Language) reachability, and a summarization technique based on TAL reachability. TAL is a class of languages that can be generated by production rules over two strings, rather than CFL whose production rules are over one string, and the two strings can

be viewed as two separated paths that can be connected by a premise. Because the types of premises are confined by TAL, it is not easy to extend this approach to support multiple premises and more types of premises. Furthermore, in the TALCRA, a technique called chaining nodes is used, whose effect is similar to our ConCRA-1 analysis. However, they require a separate algorithm to identify chaining nodes, in contrast that our approach generates the bridging edges naturally within one pass. Because of this separation, TALCRA may generate more "bridging edges" than necessary and can remove them only after the identification of chaining nodes, on the other hand our approach would not generate extra bridging edges and is thus more efficient. Moreover, our approach also allows to be adapted to ConCRA-k with any k, which is not supported by TALCRA.

There have been some other recent advancements on library code summarization. Dillig et al. [6] proposed a flow-sensitive memory-safety analysis, in which they used library summarization, and considered strong updates in the summary building process. Kulkarni et al. [12] proposed to learn summaries from a training corpus to accelerate the analysis of other programs that share code with the corpus. In constract to our approach, they require the developers to write a check function for each analysis to determine the soundness of the summary with respect to the current analysis task. Zhu et al. [48] proposed to infer information-flow specifications of library code by analyzing the client code. However, these specifications need to be manually verified against library documents to ensure correctness. Zhang et al. [46] proposed a general framework to hybrid top-down and bottom-up analysis. In their analysis, bottom-up analysis and top-down analysis can complement each other to achieve better performance. None of the above four approaches support automatic summarization of library code with unknown components.

7 Conclusion

In this paper, we demonstrate that by directly extending CFL-reachability analysis rules with premises, we can turn a standard CFL-reachability analysis into a conditional summarization approach with client analysis, and this approach is more efficient and more general than existing summarization techniques based on the dedicated TAL-reachability analysis. We believe that this approach indicates the potential existence of a more generic method to extend existing analysis techniques into a library summarization technique. This is a future direction to be explored.

Acknowledgement. This work is supported by the National Key Research and Development Program under Grant No. 2016YFB1000105, and the National Natural Science Foundation of China under Grant No. 61421091, 61225007, 61672045.

References

1. Arzt, S., Bodden, E.: Stubdroid: Automatic inference of precise data-flow summaries for the android framework. In: Proc. ICSE. pp. 725–735 (2016)

2. Bastani, O., Anand, S., Aiken, A.: Specification inference using context-free language reachability. In: Proc. POPL. pp. 553–566 (2015)
3. Cousot, P., Cousot, R.: Modular static program analysis. In: Proc. CC. pp. 159–178 (2002)
4. Das, A., Lahiri, S.K., Lal, A., Li, Y.: Angelic verification: Precise verification modulo unknowns. In: Proc. CAV. pp. 324–342 (2015)
5. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Proc. ECOOP. pp. 77–101 (1995)
6. Dillig, I., Dillig, T., Aiken, A., Sagiv, M.: Precise and compact modular procedure summaries for heap manipulating programs. In: Proc. PLDI. pp. 567–577 (2011)
7. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: Proc. PASTE. pp. 54–61 (2001)
8. Itzhaky, S., Bjørner, N., Reps, T.W., Sagiv, M., Thakur, A.V.: Property-directed shape analysis. In: Proc. CAV. pp. 35–51 (2014)
9. Jaffar, J., Murali, V., Navas, J., Santosa, A.: Path-sensitive backward slicing. In: Proc. SAS. pp. 231–247 (2012)
10. Kodumal, J., Aiken, A.: The set constraint/cfl reachability connection in practice. In: Proc. PLDI. pp. 207–218 (2004)
11. Komondoor, R., Ramalingam, G.: Recovering data models via guarded dependences. In: Proc. WCRE. pp. 110–119 (2007)
12. Kulkarni, S., Mangal, R., Zhang, X., Naik, M.: Accelerating program analyses by cross-program training. In: Proc. OOPSLA. pp. 359–377 (2016)
13. Lattner, C., Lenharth, A., Adve, V.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Proc. PLDI. pp. 278–289 (2007)
14. Lhoták, O., Hendren, L.: Scaling java points-to analysis using spark. In: Proc. CC. pp. 153–169 (2003)
15. Lochbihler, A., Snelting, G.: On temporal path conditions in dependence graphs. ASE 16(2), 263–290 (2009)
16. Macedo, H., Touili, T.: Mining malware specifications through static reachability analysis. In: Proc. ESORICS. pp. 517–535 (2013)
17. Madhavan, R., Ramalingam, G., Vaswani, K.: Modular heap analysis for higher-order programs. In: Proc. SAS. pp. 370–387 (2012)
18. Milanova, A., Huang, W., Dong, Y.: Cfl-reachability and context-sensitive integrity types. In: Proc. PPPJ. pp. 99–109 (2014)
19. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: Proc. POPL. pp. 327–338 (2007)
20. Pratikakis, P., Foster, J.S., Hicks, M.: Existential label flow inference via cfl reachability. In: Proc. SAS. pp. 88–106 (2006)
21. Pratikakis, P., Foster, J.S., Hicks, M.W.: LOCKSMITH: Context-sensitive correlation analysis for race detection. In: Proc. PLDI. pp. 320–331 (2006)
22. Ravitch, T., Jackson, S., Aderhold, E., Liblit, B.: Automatic generation of library bindings using static analysis. In: Proc. PLDI. pp. 352–362 (2009)
23. Rehof, J., Fähndrich, M.: Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In: Proc. POPL. pp. 54–66 (2001)
24. Reps, T.: Shape analysis as a generalized path problem. In: Proc. PEPM. pp. 1–11 (1995)
25. Reps, T.: Program analysis via graph reachability. Information and Software Technology 40(11-12), 701–726 (1998)
26. Reps, T.: Undecidability of context-sensitive data-dependence analysis. TOPLAS 22(1), 162–186 (2000)

27. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proc. POPL. pp. 49–61 (1995)
28. Reps, T., Horwitz, S., Sagiv, M., Rosay, G.: Speeding up slicing. In: Proc. FSE. pp. 11–20 (1994)
29. Reps, T., Schwoon, S., Jha, S.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: Proc. SAS. pp. 189–213 (2003)
30. Rinetzky, N., Poetzsch-Heffter, A., Ramalingam, G., Sagiv, M., Yahav, E.: Modular shape analysis for dynamically encapsulated programs. In: Proc. ESOP. pp. 220–236 (2007)
31. Rountev, A., Kagan, S., Marlowe, T.: Interprocedural dataflow analysis in the presence of large libraries. In: Proc. CC. pp. 2–16 (2006)
32. Rountev, A., Ryder, B.G.: Points-to and side-effect analyses for programs built with precompiled libraries. In: Proc. CC. pp. 20–36 (2001)
33. Rountev, A., Sharp, M., Xu, G.: IDE dataflow analysis in the presence of large object-oriented libraries. In: Proc. CC. pp. 53–68 (2008)
34. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theor. Comput. Sci. 167(1-2), 131–170 (1996)
35. Snelting, G., Robschink, T., Krinke, J.: Efficient path conditions in dependence graphs for software safety analysis. TOSEM 15(4), 410–457 (2006)
36. Sridharan, M., Gopan, D., Shan, L., Bodík, R.: Demand-driven points-to analysis for Java. In: Proc. OOPSLA. pp. 57–76 (2005)
37. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: Proc. PLDI. pp. 387–400 (2006)
38. Sukumaran, S., Sreenivas, A., Metta, R.: The dependence condition graph: Precise conditions for dependence between program points. Computer Languages, Systems & Structures 36(1), 96 – 121 (2010)
39. Tang, H., Wang, X., Zhang, L., Xie, B., Zhang, L., Mei, H.: Summary-based context-sensitive data-dependence analysis in presence of callbacks. In: Proc. POPL. pp. 83–95 (2015)
40. Tschantz, M.C., Wing, J.M.: Extracting conditional confidentiality policies. In: Proc. SEFM. pp. 107–116 (2008)
41. Xu, G., Rountev, A.: Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In: Proc. ISSTA. pp. 225–235 (2008)
42. Xu, G., Rountev, A., Sridharan, M.: Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In: Proc. ECOOP. pp. 98–122 (2009)
43. Yannakakis, M.: Graph-theoretic methods in database theory. In: Proc. PODS. pp. 230–242 (1990)
44. Zhang, Q., Lyu, M.R., Yuan, H., Su, Z.: Fast algorithms for Dyck-CFL reachability with applications to alias analysis. In: Proc. PLDI. pp. 435–446 (2013)
45. Zhang, Q., Su, Z.: Context-sensitive data-dependence analysis via linear conjunctive language reachability. In: Proc. POPL. pp. 344–358 (2017)
46. Zhang, X., Mangal, R., Naik, M., Yang, H.: Hybrid top-down and bottom-up interprocedural analysis. In: Proc. PLDI. pp. 249–258 (2014)
47. Zheng, X., Rugina, R.: Demand-driven alias analysis for C. In: Proc. POPL. pp. 351–363 (2008)
48. Zhu, H., Dillig, T., Dillig, I.: Automated inference of library specifications for source-sink property verification. In: Proc. APLAS. pp. 290–306 (2013)