2018

Collective program analysis

Ganesha Upadhyaya
Iowa State University

Collective program analysis

by

Ganesha Upadhyaya

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Iowa State University

Ames, Iowa

2018

DEDICATION

I would like to dedicate this dissertation to my lovely wife Priya without whose support I would not have been able to complete this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

"*You may see me struggle, but you will never see me quit*". This is what my advisor Hridesh Rajan has turned me into. Throughout my graduate study at Iowa State University, he has motivated me, guided me, and nurtured me with incredible patience. This dissertation would not have been possible without him. I would like to take this opportunity to express my thanks to everything that he has done for me. I can recollect those countless number of night outs that we did to meet conference paper deadlines. I have learnt a lot and one day I aspire to be as excellent researcher and mentor as him.

I would like to thank my committee members for their efforts and contributions to this work: Wei Le, Pavan Aduri, Robyn Lutz, and Arun Somani. I have received several suggestions on my work, which has helped greatly in both improving the technical content as well as presentation. I would like to specially thank Wei Le for mentoring me selflessly, especially during the past year. She has provided several insights and suggestions which have helped to shape the approach taken to solve some of problems addressed in this dissertation. I would also like to thank Tien Nguyen for inspiring me and providing many insightful suggestions to my work in this dissertation.

I would like to thank my senior colleagues in the Laboratory of Software Design: Robert Dyer, Hoan Nguyen, Mehdi Bagherzadeh, Tyler Sondag, Youssef Hanna, and Yuheng Long, with whom I had an opportunity to collaborate during early years of my graduate study. They all have inspired me immensely and provided timely and insightful suggestions. I would also like to thank some of my junior colleagues: Nitin Tiwari and Ramanathan Ramu with whom I had several opportunities to collaborate and publish. I would like to thank my other colleagues in the Laboratory of Software Design: Eric Lin, Samantha Syeda, Swarn Priya, David Johnston, Nathaniel Wernimont, and all others whom I may have missed, for insightful discussions throughout my stay in the lab. I would also like to thank Carla Harris, our student advisor for graduate program at Computer Science, for her continuous guidance and assistance with official matters of graduate program. She has been a go to person for me throughout these years and she always worked in favor of her students.

My family has been extremely supportive over all these years. I would not have made this far without their love and support.

My research described in this dissertation was supported in part by grants from the US National Science Foundation (NSF) under grants CCF-08-46059, CCF-14-23370, CCF-11-17937, CCF-13-49153, and CNS-15-13263. These grants provided me wonderful environment to work in.

The majority of content of this dissertation can be found in prior publications. Chapters 3 is based on our ICSE 2017 NIER paper that introduces the vision of collective program analysis [84]. Chapter 4 is based on our technical report and TSE journal submission, which is accepted and currently going through the final publication process [83]. Chapter 5 is based on our ICSE 2018 paper [85].

In this thesis I do not include some of the other research works that I did during my PhD career, for example, I was fortunate to work on the Panini project [53, 52, 9] where I implemented efficient compilation strategies for capsules [81, 82] to improve the performance of Panini programs. I also contributed to the design and implementation of Candoia, a platform and ecosystem for building mining software repository apps [77, 76].

ABSTRACT

Encouraged by the success of data-driven software engineering (SE) techniques that have found numerous applications e.g. in defect prediction, specification inference, etc, the demand for mining and analyzing source code repositories at scale has significantly increased. However, analyzing source code at scale remains expensive to the extent that data-driven solutions to certain SE problems are beyond our reach today. Extant techniques have focused on leveraging distributed computing to solve this problem, but with a concomitant increase in computational resource needs. In this thesis, we propose *collective program analysis (CPA)*, a technique to accelerate ultra-large-scale source code mining without demanding more computational resources and by utilizing the similarity between millions of source code artifacts.

First, we describe the general concept of collective program analysis. Given a mining task that is required to be run on thousands of artifacts, the artifacts with similar interactions are clustered together, such that the mining task is required to be run on only one candidate from each cluster to produce the mining result and the results for other candidates in the same cluster can be produced using extrapolation. The two technical innovations of collective program analysis are: mining task specific similarity and interaction pattern graph. Mining task specific similarity is about whether two or more artifacts can be considered similar for a given mining task. An interaction pattern graph represents the interaction between the mining task and the artifact when the mining task is run on the artifact. An interaction pattern graph is used to determine mining task specific similarity between artifacts.

Given a mining task and an artifact producing an interaction pattern graph soundly and efficiently can be very challenging. We propose a pre-analysis and program compaction technique to achieve this. Given a source code mining task and thousands of input programs on which the mining task needs to be run, our technique first extracts the information about what parts of an input program are relevant for the mining task and then removes the irrelevant parts from input programs, prior to running the mining task on them. Our key technical contributions are a static analysis to extract information about the parts of program that are relevant for a mining task and a sound program compaction technique that produces a reduced program on which the mining task has similar output as original program.

Upon producing interaction pattern graphs of thousands of artifacts, they have to be clustered and the mining task results have to be reused between similar artifacts to achieve acceleration. In the final part of this thesis, we fully describes collective program analysis and illustrate mining millions of control flow graphs (CFGs) by clustering similar CFGs.

CHAPTER 1. INTRODUCTION

Recently there has been significant interest and success in analyzing large corpora of source code repositories to solve a broad range of software engineering problems including but not limited to defect prediction [18], discovering programming patterns [86, 74], suggesting bug fixes [40, 39], specification inference [1, 48], etc. The approaches that perform source code mining and analysis at massive scale can be expensive. Fortunately, extant work has focused on leveraging distributed computing techniques to speedup ultra-large-scale source code mining [22, 10, 32], but with a concomitant increase in computational resource needs. As a result, much larger scale experiments have been attempted that perform mining over abstract syntax tree (AST), e.g. [24]. While mining over AST is promising, many SE use-cases seek richer input that further increases the necessary computational costs, e.g. the precondition mining analyzes the control flow graph (CFG). While it is certainly feasible to improve the capabilities of the underlying infrastructure by adding many more CPUs, nonprofit infrastructures e.g. Boa [22] are limited by their resources and commercial clouds can be exorbitantly costly for such tasks.

1.1 Contributions

In this thesis, we propose collective program analysis, a technique to accelerate ultra-large-scale source code mining. The key concepts of collective program analysis are: the mining task specific similarity, the interaction pattern graph, and the process of clustering and reusing mining results.

We describe our vision of ultra-large-scale mining and the concept of mining task specific similarity and interaction pattern graph. We demonstrate how ultra-large-scale mining can be accelerated using two case-studies.

We provide a pre-analysis and program compaction technique to generate the interaction pattern graph soundly and efficiently. The technique is able to analyze the mining task, automatically extract the information about what parts of the program are relevant for the mining task, and reduce the program. We demonstrate that for a variety source code mining tasks, our technique produces compact representation and yield significant speedup.

We describe how to cluster programs using interaction pattern graphs and our technique of collective program analysis for accelerating large scale source code analysis. We showcase the useful and effectiveness of collective program analysis in mining millions of control flow graphs.

1.2 Outline

In the next chapter, we review the literature and present the works related to accelerating large scale source code analysis. In chapter 3, we introduce the general concept of collective program analysis and the two key concepts: the mining task specific similarity and the interaction pattern graphs. In chapter 4, we describe a pre-analysis and program compaction technique to produce the interaction pattern graph soundly and efficiently. In chapter 5, we describe collective program analysis fully and demonstrate its application in mining millions of control flow graphs. Finally, in chapter 6, we provide a conclusion and several avenues for future work.

CHAPTER 2. RELATED WORK

2.1 Program Compaction

The concept of identifying and removing the irrelevant parts has been used in other approaches to improve the efficiency of the techniques [5, 64, 90, 20]. Allen *et al.* [5] proposed a staged points-to analysis framework for scaling points-to analysis to large code bases, in which the points-to analysis is performed in multiple stages. In each stage, they perform static program slicing and compaction to reduce the input program to a smaller program that is semantically equivalent for the points-to queries under consideration. Their slicing and compaction can eliminate variables and their assignments that can be expressed by other variables. Smaragdakis *et al.* [64] proposed an optimization technique for flow-insensitive points-to analysis, in which an input program is transformed by a set-based pre-analysis that eliminates statements that do not contribute new values to the sets of values the program variables may take. In both the techniques, reduction in the number of variables and allocation sites is the key to scaling points-to analysis. The pre-analysis stage of both the techniques tracks the flow of values to program variables. Any analysis requiring analyzing program variables may benefit from these techniques. In contrast, our technique is more generic and it goes beyond analyses that track program variables and their values, e.g., tracking method calls, tracking certain patterns. The main advantage in our technique is that the relevant information for an input analysis is extracted automatically by performing a static analysis, making our technique applicable to a larger set of analyses that analyze different informations. The concept of identifying and removing the irrelevant parts has been used in other approaches to improve the efficiency of the techniques [90, 20]. For example, Wu *et al.* [90] uses the idea to improve the efficiency of the call trace collection and Ding *et al.* [20] uses the idea to reduce the number of invocations of the symbolic execution in identifying the infeasible branches in the code. Both Wu *et al.*and Ding *et al.*identify the relevant parts of the input for the task at hand. The task in these approaches is fixed. In Wu *et al.*the task is call trace collection and in Ding *et al.*the task is infeasible branch detection using symbolic execution, while in our technique the task varies and our technique identifies the relevant parts of the input for the user task by analyzing the task. There have been efforts to scale path sensitive analysis of programs by detecting and eliminating infeasible paths (pruning the paths) before performing the analysis [72, 37]. Such a technique filters and retains only relevant paths that leads to unique execution behaviors. In their technique a path is relevant if it contains event nodes and events are specified by the path sensitive analysis. For example, safe synchronization path-sensitive analysis has lock and unlock as events and any path that contains lock or unlock will be considered relevant. Compared to this approach, our approach is not limited to just event-based path sensitive analysis, but can be beneficial to flow-sensitive and path-insensitive analysis. Unlike the prior work that requires a user specify the list of events in their event-based path sensitive analysis, our technique can automatically derive the information with respect to what is relevant to the analysis by performing a static analysis.

Program slicing is a technique for filtering a subset of the program statements (a slice) that may influence the values of variables at a given program point [88]. Program slicing has been shown to be useful in analyzing, debugging, and optimizing programs. For example, Lokuciejewski *et al.* [41] used program slicing to accelerate static loop analysis. Slicing cannot be used for our purpose, because the program points of interest (program statements) are not known. We require a technique like our static analysis that computes this information. Even if the statements of interest are known, slicing may include statements (affecting the values of variables at program points of interest) that may not contribute to the analysis output. Our technique only includes statements that contributes to the analysis output. Moreover, a program slice is a compilable and executable entity, while a reduced program that we produce is not. In our case, the original and the reduced programs produce the same result for the analysis of interest.

2.2 Reuse

Our work can also be compared to work in accelerating program analysis [38, 58, 92, 3]. Kulkarni *et al.* [38] proposed a technique for accelerating program analysis in Datalog. Their technique runs an offline analysis on a corpus of training programs and learns analysis facts over shared code. It reuses the learned facts to accelerate the analysis of other programs that share code with the training corpus. Other works also exists that performs pre-analysis of the library code to accelerate analysis of the programs that make use of the library code exists [58, 92, 3]. Our technique differs from these prior works in that, our technique does not precompute the analysis results of parts of the program, but rather identifies parts of the program that do not contributes to the analysis output and hence can be safely removed. While prior works can benefit programs that share some common code in the form of libraries, our technique can benefit all programs irrespective of the amount of shared code.

Reusing analysis results to accelerate interprocedural analysis by computing partial [31] or complete procedure summaries [57, 62] has also been studied. These techniques first run the analysis on procedures and then compute either partial or complete summaries of the analysis results to reuse them at the procedure call sites. The technique can greatly benefit programs in which procedures contain multiple similar call sites. In contrast, our technique can accelerate analysis of individual procedures. If the analysis requires inter-procedural context, our technique can be combined with the prior works, hence we consider our approach to be orthogonal to prior works with respect to accelerating inter-procedural analyses.

Caching the proofs for the constraints that occur repeatedly during the analysis of the same or similar programs is used to scale constraint-based program analysis [15, 7, 6, 46]. For instance, Aquino *et al.* [6] uses this technique for scaling of symbolic analysis, Mudduluru and Ramanathan [46] uses caching for efficient incremental analysis.

2.3 Source Code Similarity

Similar code or code clones includes "*look alike*" codes that are textually, syntactically, structurally similar and codes that are behaviorally or functionally similar. Existing approaches of identifying code clones can be categorized based on the types of intermediate representations they use [59]: token-based, AST-based, and graph-based. There are also other approaches that goes beyond structural similarity: code fingerprints[43], behavioral clones [26, 71], and run-time behavioral similarity [19]. McMillan *et al.* [43] went beyond structural similarity and showed a technique for identifying similar applications using code fingerprints. For instance, identifying clones by tracking API usages. Behavioral clones [70, 26] can be identified by checking the functional equivalence of the inputs and outputs of code fragments. Su *et al.* [71] argued that codes that does not show equivalence of the inputs and outputs may also behave similar, hence they propose using run-time behavior (i.e., instructions executed) for identifying code that is behaviorally similar. Demme and Sethumadhavan [19] proposed using run-time behaviors such as instruction mixes, static control/-data flow graphs, dynamic data flow graphs for grouping behaviorally similar functions to enable better program understanding. They also show that the behaviorally similar functions react similar to know optimizations. Nagappan *et al.* [47] have focused on using sampling to identify percentage of projects in a population that are similar to the given sample. Sampling is performed using different project dimensions such as project age, commits, churn, total lines of code, etc. Clone detection techniques are agnostic to the mining task that is performed on the artifacts.

We did not use syntactic clones (token-based or AST-based), because the benefits will be limited to copy-and-paste code. Semantic clones (code fragments with similar control and data flow) could not be used, because of lack of guarantee that analysis output will be similar. Moreover, semantically different code fragments may produce similar output for a given analysis and we would miss out on those. We cannot use functional clones (code fragments with similar input/output), because they may not produce similar analysis output. We also could not use behavioral clones (code fragments that perform similar computation captured using dynamic dependence graphs), because they cannot guarantee similar analysis output. An analysis may produce similar output for code fragments that are not behavioral clones. Further, in our setting, while analyzing thousands of projects, it is not feasible to instrument the code, run them, collect traces, and build dynamic dependence graphs to detect behavioral clones.

Code search engines [14, 45, 56, 73] adopt clustering-and-retrieval methods to accelerate the search beyond parallelization. Clustering is a natural and popular approach, where similar programs are grouped together (based on some similarity criteria [59, 43, 26, 71, 19]) and then given a code search query, nearest clusters are searched for retrieving the code examples. The clustering techniques used by the search engines are query agnostic in nature.

2.4 Source Code Analysis Infrastructures

Several infrastructures exist today for performing ultra-large-scale analysis [22, 10, 32]. Boa [22] is a language and infrastructure for analyzing open source projects. Sourcerer [10] is an infrastructure for large-scale collection and analysis of open source code. GHTorrent [32] is a dataset and tool suite for analyzing GitHub projects. These frameworks currently support structural or abstract syntax tree (AST) level analysis and a parallel framework such as map-reduce is used to improve the performance of ultra-large-scale analysis.

2.5 Implementation Techniques for Source Code Analysis

Atkinson and Griswold [8] discuss several implementation techniques for improving the efficiency of data-flow analysis, namely: factoring data-flow sets, visitation order of the statements, selective reclamation of the data-flow sets. They discuss two commonly used traversal strategies: iterative search and worklist, and propose a new worklist algorithm that results in 20% fewer node visits. In their algorithm, a node is processed only if the data-flow information of any of its successors (or predecessors) has changed. Tok *et al.* [79] proposed a new worklist algorithm for accelerating inter-procedural flow-sensitive data-flow analysis. They generate inter-procedural def-use chains on-the-fly to be used in their worklist algorithm to re-analyze only parts that are affected by the changes in the flow values. Hind and Pioli [33] proposed an optimized priority-based worklist algorithm for pointer alias analysis, in which the nodes awaiting processing are placed on a worklist prioritized by the topological order of the CFG, such that nodes higher in the CFG are processed before nodes lower in the CFG. Bourdoncle [12] proposed the notion of weak topological ordering (WTO) of directed graphs and two iterative strategies based on WTO for computing the analysis solutions in dataflow and abstraction interpretation domains. Bourdoncle' technique is more suitable for cyclic graphs, however for acyclic graphs Bourdoncle proposes any topological ordering. Kildall [36] proposes combining several optimizing functions with flow analysis algorithms for solving global code optimization problems. For some classes of data-flow analysis problems, there exist techniques for efficient analysis. For example, demand inter-procedural data-flow analysis [34] can produce precise results in polynomial time for inter-procedural, finite, distributive, subset problems (IFDS), constant propagation [87], etc. Cobleigh *et al.* [17] study the effect of worklist algorithms in model checking. They identified four dimensions along which a worklist algorithm can be varied. Based on four dimensions, they evaluate 9 variations of worklist algorithm.

2.6 BigCode Exploration

Predicting program properties by learning Big Code is also studied [55, 51]. Raychev *et al.* [55] present a technique that learns a probabilistic model from existing programs and then uses this model to predict properties of new programs. Peleg *et al.* [51] presents a framework for representing Big Code for drawing insights about programming practice and enabling code reuse. They first extract partial temporal specifications from code, represent partial temporal specifications as symbolic automata, and use symbolic automata to construct an abstract domain for static analysis of big code.

CHAPTER 3. INTERACTION PATTERN GRAPH

Mining open source software repositories such as GitHub is valuable and can be leveraged to help software engineering tasks, e.g. for defect prediction [18], bug fix suggestions [40], specification inference [48], etc. The approaches that leverage these open source repositories perform mining and analysis that cuts across projects. The current infrastructure support for ultra-large-scale mining leverages parallelization techniques such as map-reduce [22, 10, 32]. We propose a new direction for accelerating ultra-large-scale mining based on the *interaction pattern* between the mining task and the artifacts, and *mining task specific similarities*.

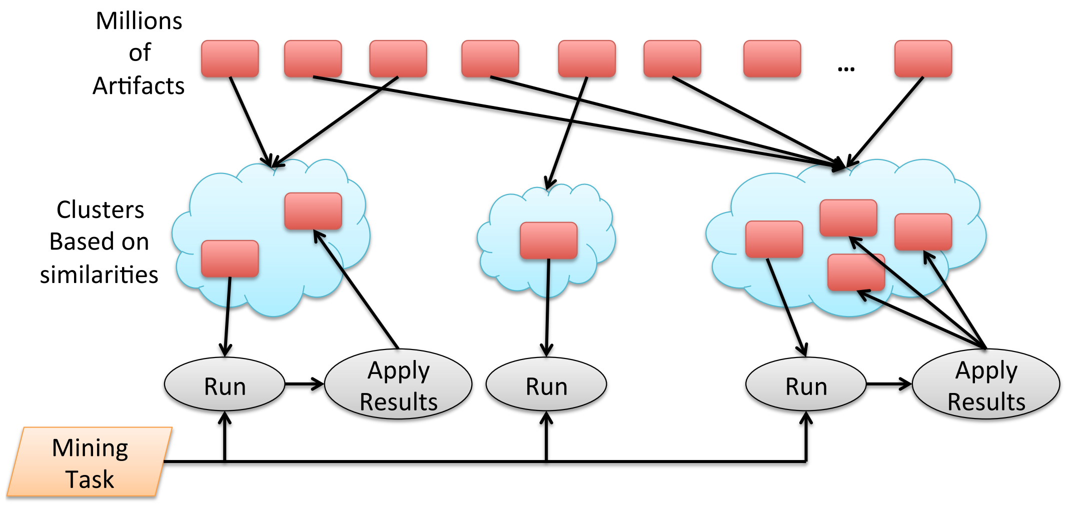


Figure 3.1: Accelerating ultra-large-scale mining vision

As illustrated in Figure 3.1, our idea is to analyze the interaction between the mining task and the artifacts and use the similarity between the interactions to cluster artifacts such that it is sufficient to perform mining on one candidate in each cluster and extrapolate the mining results to others. In this way, ultra-large-scale mining can be accelerated.

This work makes the following contributions:

- We propose a new research direction for accelerating ultra-large-scale mining by *task-specific clustering.*
- We introduce the notion of *interaction pattern graph* that represents the interaction between the mining task and the artifact, however we leave the details of soundly extracting the graph a subject of the future work.
- We present two case-studies that demonstrates the usage of our technique.

3.1 Mining Task Specific Similarity

Software source code is an important artifact that is mined often. There exist many techniques to identify code clones [59]. Syntactic clones identify codes that look alike. Semantic clones identify semantically similar codes (control or data flow similarities). Functional clones [26] detect codes that are functionally similar by comparing inputs and outputs of methods. Although syntactic and semantic clones may result in the same output for a mining task and hence they can be used to cluster and accelerate the ultra-large-scale mining, they may be less beneficial. This is because in case of syntactic clones, the amount of acceleration is limited by the amount of copy-and-paste code and in case of semantic clones, only a subset of analysis can accelerate, for instance control flow only analyses. Functional clones may not result in the same output for a mining task, hence they cannot be used to cluster. This has led us to go beyond syntactic, semantic, and functional clones to develop a notion of *mining-task-specific similarity.*

Given a mining task, a set of artifacts can be considered similar, if the mining task has similar interactions with the artifacts. When a mining task is run on an artifact, one can observe the interaction between the mining task and the artifact. The interaction is captured by the execution trace, where the trace describes, what parts of the artifact are of interest to the mining task, and whether such parts exist in the artifact. The interaction (or the execution trace) can be captured by running the mining task on the artifact, however the challenge is to determine the interaction without running the mining task on the artifacts.

```
1 String extractDomain(String email) {
2   int separator = email.lastIndexOf(':');
3   if (separator != -1 && ++separator < email.length)
4   {
5     return email.substring(separator);
6   }
7   return null;
8 }
```

```
1 boolean createFile(String pp, String op, String sc) {
2   boolean ok = true;
3   log.verbose("parsing: " + pp);
4   // 36 LOC containing try-catch, if, for
5   if (dotIndex == -1) {
6     className = name;
7   } else {
8     className = name.substring(0, dotIndex);
9   }
10  // 30 LOC containing try-catch, if
11 }
```

Figure 3.2: Two methods for illustrating the notion of mining task specific similarity

To illustrate the notion of mining task specific similarity, consider the two methods `extractDomain` and `createFile` shown in Figure 3.2. These two methods are not similar in terms of syntax, semantics, or functionality. However, these two methods can be considered similar if the mining task that we want to perform extracts the conditions that must be checked before calling `substring` Java API method. For instance, the `extractDomain` method checks a condition at line 3 before calling `substring` at line 5. Similarly, the `createFile` method checks a condition at line 5 before calling `substring` at line 8. Hence for the purpose of a mining task that wants to extract conditions that are checked before calling `substring` Java API method, the two methods `extractDomain` and `createFile` have similar interactions with the mining task and can be considered similar.

Further, our observation is that, by analyzing the mining task, it is possible to extract the parts of the artifacts that are of interest to the mining task. By utilizing this knowledge, it is possible to capture the interaction by performing a light-weight traversal of the artifact. This traversal identifies the parts of the artifact that are of interest to the mining task. Upon identifying these parts, the parts that are not of interest to the mining task can be removed, resulting in a reduced artifact that only contains parts relevant for the mining task. This reduced artifact is used to represent the interaction between the mining task and the artifact.

3.2 Interaction Pattern Graph

An interaction pattern graph is a reduced artifact graph that retains only those nodes of the artifact graph that are relevant for the given mining task. To illustrate, consider a task of mining API preconditions. API preconditions are the conditions that must be satisfied before calling an API method. API preconditions can be inferred by looking at the guard conditions at the API method call sites [48]. Input to the mining task is a set of API methods whose preconditions are to be mined and a large number of client projects that calls the API methods. The technique builds the control flow graphs (CFG) of the client methods and performs a dominator analysis to determine all control dependent nodes of the API method call node in the CFG, that contains predicate expressions. The output of the mining task is a set of predicate expressions on which the API method call is control dependent (in a way these predicate expressions guard the API method call).

```java
public void body(String namespace, String name, String text)
  throws Exception {
  String namespaceuri = null;
  String localpart = text;
  int colon = text.indexOf(':');
  if (colon >= 0) {
    String prefix = text.substring(0,colon);
    namespaceuri = digester.findNamespaceURI(prefix);
    localpart = text.substring(colon+1);
  }
  ContextHandler contextHandler = (ContextHandler)digester.peek();
  contextHandler.addSoapHeaders(localpart,namespaceuri);
}
```

Figure 3.3: Code snippet from Apache Tomcat GitHub project.

Here, the artifacts are the client methods and the mining task queries method statements that calls API methods and conditional statements that guard the API method calls. To understand the interaction between the mining task and the artifact, consider the example code shown in

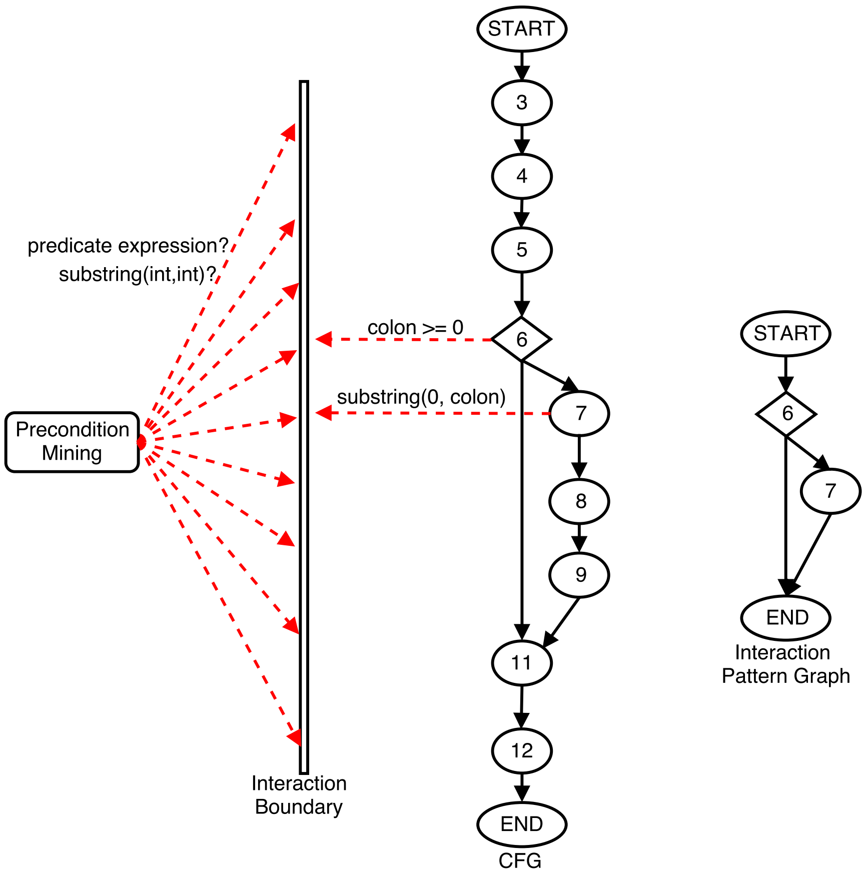


Figure 3.4: Interaction of the precondition mining task with the CFG of the code shown in Listing 3.3.

Figure 3.3. This example calls *substring*(*int*, *int*) API method from *java.lang.String* at line 7. This API method call is guarded by the precondition at line 6, whose predicate expression is *colon* >= 0. Now, let us consider Figure 3.4 that shows the interaction between the *Precondition Mining* task and the CFG of the method shown in Figure 3.3. When the *Precondition Mining* task is run on the CFG, it queries for predicate expressions and *substring*(*int*, *int*) API method calls. Each node in the CFG responds to the queries if they have predicate expressions or *substring*(*int*, *int*) API method calls. For the CFG shown in the figure, only node 6 and node 7 responds successfully as node 6 contains predicate expression *colon* >= 0 and node 7 makes the *substring*(*int*, *int*) API method call. The interaction can be summarized using the interaction pattern graph shown in

Figure 3.4. The interaction pattern graph contains only those nodes that successfully responded to the mining task queries (START and END are two special nodes).

3.3 Generating Interaction Pattern Graph

Mining a software artifact requires traversing the artifact graph and querying the nodes to extract some information. For instance, in our *API Precondition Mining* task, the artifact graph is the CFG of the method. The listing below shows the pseudo code of the traversal in the mining task.

```
API_precondition_mining (ArtifactGraph G) {
  output: {predicate expressions , API method calls}
  For each node in G
    if (node is a predicate expression )
      add predicate expression to output
    else if (node is an API method call)
      add API method call to output
  return output
}
```

Our claim is that it is possible to extract a set of rules from the traversals. For instance, the pseudo code above has two rules: *Rule 0: node is a predicate expression* and *Rule 1: node is an API method call*. We deduce these rules as follows. We analyze the body of the traversal and extract program paths. These program paths have path conditions that must be satisfied for that path to be taken. Our idea is to use the conjunction of path conditions as rules to identify the relevant nodes. The above pseudocode has two paths: one that takes the *if* branch and other path that takes the *else* branch. The path condition for the first path is *(node is predicate)* $\bigwedge \neg$*(node is an API method call)*. The path condition for the second path is $\neg$*(node is predicate)* $\bigwedge$ *(node is an API method call)*. In this way, by enumerating the paths in the traversal body and collecting the path conditions, one can construct a set of rules that helps to infer the interaction pattern.

Upon extracting the rules from the traversals, we perform a light-weight traversal of the CFG to generate the interaction pattern graph. In this traversal we keep only those nodes for which the rules evaluates to *true*. For instance, consider the CFG shown in Figure 3.4. Only *node 6* and *node 7* are kept in the interaction pattern graph, because for only these two nodes the rules evaluates to *true*.

3.4 Accelerating Mining Using Interaction Pattern Graph

An overview of our approach is shown in Figure 3.5. Given a mining task and a large collection of artifact graphs,[1] a light-weight traversal is performed on each artifact graph that identifies the parts relevant for the mining task and removes the irrelevant parts to produce an *interaction pattern graph*.[2] Upon generating the interaction pattern graph, we check if the pattern graph is already seen before while mining other artifact graphs, if not then we run the mining task on the original artifact graph to generate the output. An output function that provides expressions to generate the output is constructed. A simple output function is the mining task itself (as we see later in the realization of this model, we use the mining task as the output function). We persist the pattern along with its output function, such that next time when a match happens, we extract and apply the output function to generate the result, instead of running the mining task.

In our realization of the model shown in Figure 3.5, we have used the mining task itself as the output generating function, where upon generating the pattern graph, we run the mining task on the pattern graph to produce output. Note that, running the mining task on the interaction pattern graph has lower complexity than running it on the original artifact graph, because the interaction pattern graph contains fewer nodes that are of interest to the mining task.

3.5 Preliminary Results

In this section, we present two case studies that illustrate the usage of our technique.

[1] An artifact graph represents the relation between parts in the artifact. For instance, in case of source code artifact, a control flow graph represents the control flow relation between program statements.

[2] An interaction pattern graph represents the interaction between the mining task and the artifact graph.

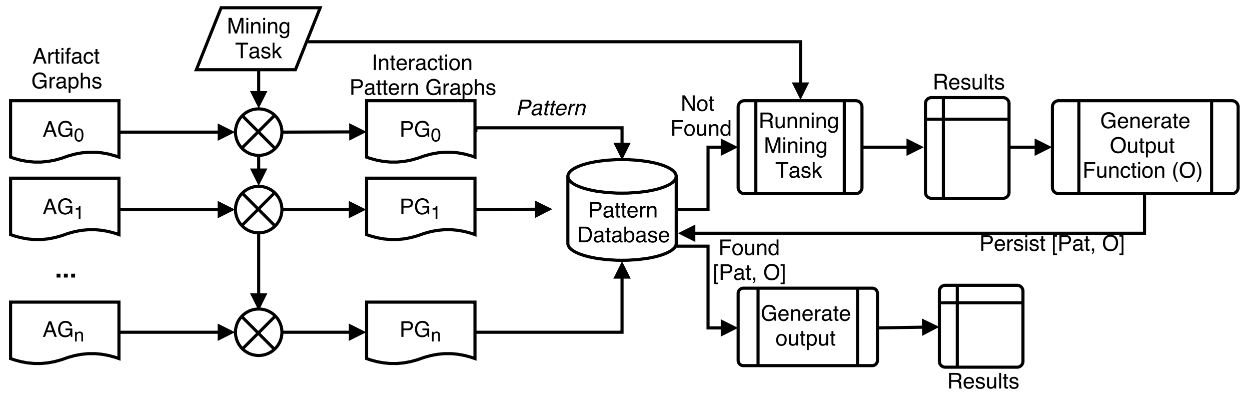


Figure 3.5: Overview of the approach: accelerating ultra-large scale mining using the interaction pattern between the mining task and the artifacts.

Table 3.1: Case study results summarized. IPA is our approach (Interaction Pattern Analysis), G is gain, R is reduction.

	Analysis Time (s)			Graph Size		
	Baseline	IPA	%G	Baseline	IPA	%R
CS1	1555.579	309.731	80	52,475,484	17,945,817	66
CS2	235.524	131.160	45	52,475,484	21,631,192	59

3.5.1 Case Study 1. Mining API Preconditions

In this case study we use *Mining API Preconditions*, that mines API preconditions of a given API method using the client methods that call the API method. Here, the API preconditions are the predicate expressions that guard the API method calls in the client methods. This mining task traverses the CFG of each client method, identifies nodes that have API method calls and collect the predicate expressions on which the API method call node is control dependent using a dominator analysis. The mining task outputs the normalized predicate expressions as preconditions for a given API method call.

We have used Boa [22] for writing the mining query and we run the mining query on the Boa [22] SourceForge dataset that contains over 7 million client methods. We compared our approach with a baseline that runs the precondition mining task on the client methods sequentially. The baseline took *1555.579 seconds* to collect *5965* preconditions at *2240* client methods of *substring*(int, int) API method. On the same dataset, our approach also collected the same *5965* preconditions at *2240* client methods as baseline and the mining task took *309.731 seconds*. In this case, our technique is able to reduce the mining time by 80%. On further investigation we found that the total number of CFG nodes in the CFGs were *52,475,484* and the total number of nodes in the interaction pattern graphs of those CFGs were *17,945,817*, that is, over 66% reduction in terms of graph size. To summarize, our approach accelerates the precondition mining task by running the mining task on the interaction pattern graph that contained only API method call nodes and predicate expression nodes.

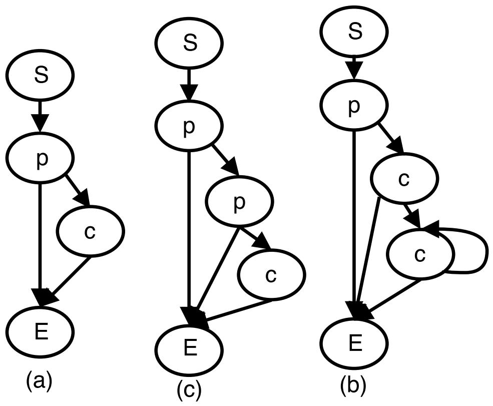


Figure 3.6: Top three interaction pattern graphs for the API Precondition Mining Task. Here, S and E are start and end nodes, p is the node with predicate expression, and c is the node that calls an API method.

Figure 3.6 shows top three interaction pattern graphs that appeared in the client methods that calls the *substring*(int, int) API method. We argue that candidates in each cluster shows similar behaviors with respect to the mining task. Studying the candidates in each cluster may itself be a new research direction for exploring and answering mining task related questions. For instance, we found that the interaction pattern graph shown in Figure 3.6(a), almost all the time provides predicate expressions that are generic to the API method and not specific to the client method that calls the API method.

3.5.2 Case Study 2. Vulnerability Detection Using Taint Analysis

In this case study our mining task detects possible vulnerabilities using a light-weight taint analysis. If the source of the value of a varible x is untrustworthy, we say that x is tainted. For instance, line $x = System.out.readLine()$ tells that x is tainted. Taint Analysis attempts to identify the variables that have been "*tainted*" with user controllable inputs and traces them in the program. If a tainted variable gets passed to a public sink without first being sanitized, it could cause a vulnerability. A public sink could be an output buffer or a console. For instance, in line $System.out.println(x)$, tainted x is passed to the console. Given a method, this mining task outputs the number of possible vulnerabilities in that method by performing an intra-procedural taint analysis.

We wrote the mining task in Boa and we ran it on the Boa's SourceForge dataset, which contains over 7 million methods. The baseline that runs the mining task on methods sequentially took *235.524 sec.* and our approach took *131.160 sec.* We were able to match all the *14309* possible vulnerabilities identified by both the approaches. The speed up that our technique achieved is close to 45%. On further investigation, we found that baseline CFGs contained *52,475,484* nodes and the corresponding interaction pattern graphs contained *21,631,192* nodes, that is, 59% reduction in terms of graph size. We verified several of these *14309* possible vulnerabilities, however it is impossible to verify them all.

3.6 Summary

In summary, our two case studies suggest that our technique of clustering based on the interaction between the mining task and the artifact can significantly accelerate the ultra-large scale mining. However, the acceleration that can be obtained depends both on the reduction in the graph size from artifact graph to interaction pattern graph, and complexity of the analysis.

3.7 Related Work

3.7.1 Accelerating Software Analysis

Kulkarni *et al.* [38] accelerates program analysis in Datalog by running the analysis offline on a corpus of training programs to learn analysis facts over shared code and then reuses the learnt facts to accelerate the analysis of other programs that share code with the training corpus. When compared to their approach, our approach does not require programs or artifacts to share code or other artifacts. Reusing analysis results to accelerate interprocedural analysis by computing partial or complete procedure summaries [57] is also studied. However, to best to our knowledge there is no technique that can benefit analysis across programs and cluster programs specific to analysis.

3.7.2 Clustering or Detecting Similar Code

Similar code or code clones includes "*look alike*" codes that are textually, syntactically, structurally similar and codes that are behaviorally or functionally similar. Existing approaches of identifying code clones can be categorized based on the types of intermediate representations they use [59]: token-based, AST-based, and graph-based. There are also other approaches that goes beyond structural similarity: code fingerprints[43], behavioral clones [26, 71], and run-time behavioral similarity [19].

CHAPTER 4. PRE-ANALYSIS AND PROGRAM COMPACTION

In chapter 3, we described the two key notions *interaction pattern graphs* and *mining task specific similarity* and their role in accelerating ultra-large-scale mining. In this chapter, we describe a technique to automatically construct the interaction pattern graphs given the mining task and the program and then use the reduced graph to accelerate analysis.

There has recently been significant interest and success in analyzing large corpora of source code repositories to solve a broad range of software engineering problems including but not limited to defect prediction [18], discovering programming patterns [86, 74], suggesting bug fixes [40, 39], specification inference [54, 95, 48, 1]. Approaches that perform source code mining and analysis at massive scale can be expensive. For example, a single exploratory run to mine API preconditions [48] of 3168 Java API methods over a dataset that contains 88,066,029 methods takes 8 hours and 40 minutes on a server-class CPU, excluding the time taken for normalizing, clustering and ranking the preconditions. Often multiple exploratory runs are needed before settling on a final result, and the time taken by experimental runs can become a significant hurdle to trying out new ideas.

Fortunately, extant work has focused on leveraging distributed computing techniques to speedup ultra-large-scale source code mining [22, 10, 32], but with a concomitant increase in computational resource requirements. As a result, many larger scale experiments have been attempted that perform mining over abstract syntax trees (ASTs), e.g., [24]. While mining ASTs is promising, many software engineering use-cases seek richer input that further increases the necessary computational costs, e.g. the precondition mining analyzes the control flow graph (CFG) [48]. While it is certainly feasible to improve the capabilities of the underlying infrastructure by adding many more CPUs, nonprofit infrastructures e.g. Boa [22] are limited by their resources and commercial clouds can be exorbitantly costly for use in such tasks.

In this work, we propose a complementary technique that accelerates ultra-large-scale mining tasks without demanding additional computational resources. Given a mining task and an ultra-large dataset of programs on which the mining task needs to be run, our technique first analyzes the mining task to extract information about parts of the input programs that will be relevant for the mining task, and then it uses this information to perform a pre-analysis that removes the irrelevant parts from the input programs prior to running the mining task on them. For example, if a mining task is about extracting the conditions that are checked before calling a certain API method, the relevant statements are those that contains API method invocations and the conditional statements surrounding the API method invocations. Our technique automatically extracts this information about the relevant statements by analyzing the mining task source code and removes all irrelevant statements from the input programs prior to running the mining task.

Prior work, most notably program slicing [88], has used the idea of reducing the input programs prior to analyzing them for debugging, analyzing, and optimizing the program. Program slicing removes statements that do not contribute to the variables of interest at various program points to produce a smaller program. Our problem requires us to go beyond variables in the programs to other program elements, such as method calls and loop constructs. Moreover, we also require a technique that can automatically extract the information about parts of the input program that are relevant to the mining task run by the user.

Source code mining can be performed either on the source code text or on the intermediate representations like abstract syntax trees (ASTs) and control flow graphs (CFGs). In this work, we target source code mining tasks that perform control and data flow analysis on millions of CFGs. Given the source code of the mining task, we first perform a static analysis to extract a set of rules that can help to identify the relevant nodes in the CFGs. Using these rules, we perform a lightweight pre-analysis that identifies and annotates the relevant nodes in the CFGs. We then perform a reduction of the CFGs to remove irrelevant nodes. Finally, we run the mining task on the compacted CFGs. Running the mining task on compacted CFGs is guaranteed to produce the same result as running the mining task on the original CFGs, but with significantly less computational cost. Our intuition behind acceleration is that, for source code mining tasks that iterates through the source code parts several times can save the unnecessary iterations and computations on the irrelevant statements, if the target source code is optimized to contain only the relevant statements for the given mining task.

We evaluated our technique using a collection of 16 representative control and data flow analyses that are often used in the source code mining tasks and compilers. We also present four case studies using the source code mining tasks drawn from prior works to show the applicability of our technique. Both the analyses used in the evaluation and the mining tasks used in the case studies are expressed using Boa [22] (a domain specific language for ultra-large-scale source code mining) and we have used several Boa datasets that contains hundreds to millions of CFGs for running the mining tasks. For certain mining tasks, our technique was able to reduce the task computation time by as much as 90%, while on average a 40% reduction is seen across our collection of 16 analyses. The reduction depends both on the complexity of the mining task and the percentage of the relevant/irrelevant parts in the input source code for the mining task. Our results indicate that our technique is most suitable for mining tasks that have small percentage of relevant statements in the mined programs.

Contributions. In summary, this chapter makes the following contributions:

- We present an acceleration technique for scaling source code mining tasks that analyze millions of control flow graphs.

- We present a static analysis that analyzes the mining task to automatically extract the information about parts of the source code that are relevant for the mining task.

- We show that by performing a lightweight pre-analysis of the source code that identifies and removes irrelevant parts prior to running the mining task, the overall mining process can be greatly accelerated without sacrificing the accuracy of the mining results.

- We have implemented our technique in the Boa domain-specific language and infrastructure for ultra-large-scale mining [22].

- Our evaluation shows that, for few mining tasks, our technique can reduce the task computation time by as much as 90%, while on average a 40% reduction is seen across our collection of 16 analyses.

4.1 Motivation

Many source code mining tasks require performing control and data flow analysis over CFGs of a large collection of programs: API precondition mining [48], API usage pattern mining [1, 95], source code search [44], discovering vulnerabilities [91], to name a few. These source code mining tasks can be expensive. For example, consider the API precondition mining [48] that analyzes the control flow graphs of millions of methods to extract conditions that are checked before invoking the API methods and uses these conditions to construct specifications for API methods.

To measure how expensive this mining task can get, we mined all 3168 API methods in the Java Development Kit (JDK) 1.6 using three datasets: D_1 (contains 6,741,465 CFGs), D_2 (88,066,029 CFGs), D_3 (161,577,735 CFGs). Larger the dataset, the API precondition mining task can produce more accurate preconditions. Running the task on D_1 took 36 minutes and D_2 took 8 hours and 40 minutes. We expect the task to take several days on D_3, hence we choose to run the task on D_3 using a distributed computing infrastructure, where the task took 23 minutes. From this experiment we can observe that, source code mining at massive scale without a parallel infrastructure can be very expensive. While, the parallel infrastructures can reduce the mining time significantly, the mining task may occupy the cluster for a significant amount of time leading to the unavailability of the resources for other tasks. So, does there exists other avenues to reduce the computational needs of the massive scale source code mining tasks?

Our work explores one such avenue, where the source code mining task is analyzed to identify the parts of the input programs that are relevant for the mining task, such that the mining task can be run on only the relevant parts to reduce the computational needs without sacrificing the accuracy of the mining results.

```
 1 public void body(String namespace, String name, String text)
 2   throws Exception {
 3   String namespaceuri = null;
 4   String localpart = text;
 5   int colon = text.indexOf(':');
 6   if (colon >= 0) {
 7     String prefix = text.substring(0,colon);
 8     namespaceuri = digester.findNamespaceURI(prefix);
 9     localpart = text.substring(colon+1);
10   }
11   ContextHandler contextHandler = (ContextHandler)digester.peek();
12   contextHandler.addSoapHeaders(localpart,namespaceuri);
13 }
```

Figure 4.1: Code snippet from Apache Tomcat GitHub project.

To elaborate, let us revisit the API precondition mining example. Let us consider that we want to mine the API preconditions of `substring(int,int)` API method. Figure 4.1 shows an example client method that invokes `substring(int,int)` API method at line 7. This API method invocation is guarded by the condition `colon >= 0` at line 6, hence the condition is considered as precondition. To extract this precondition, the mining task first builds the CFG of the method as shown in Figure 4.2 (CFG node numbers corresponds to the line numbers in Figure 4.1). The task performs several traversals of the CFG. In the first traversal, it identifies the API method call nodes as well as the conditional nodes that provide conditions. In the next traversal, the mining task performs a dominator analysis to compute a set of dominator statements for every statement in the program (A statement `x` dominates another statement `y`, if every execution of statement `y` includes the execution of statement `x`). In the final traversal, the mining task uses the results from the first two traversals to extract the conditions of all the dominating nodes of the API method invocation nodes. For this analysis the relevant nodes are: i) the nodes that contain `substring(int,int)` API method invocations and ii) the conditional nodes. In the client method shown in Figure 4.1, only line 6 & 7 are relevant (node 6 & 7 in the corresponding CFG shown in Figure 4.2). All other nodes are irrelevant and any computation performed on these nodes is not going to affect the mining results.

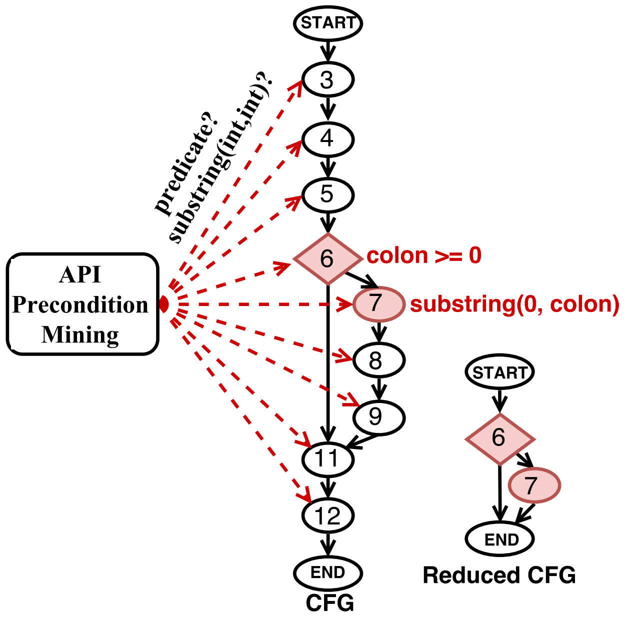


Figure 4.2: When API precondition mining analysis is run on the CFG of the code shown in Figure 4.1, it visits every node in the CFG and queries whether the nodes have predicate expression or `substring(int,int)` API method call. If so, such nodes are relevant for API precondition mining.

In the absence of our technique, the API precondition mining task would traverse all the nodes in the CFG in all the three traversals, where the traversal and the computation of the irrelevant nodes (computing dominators and extracting conditions of the dominators of the API invocation nodes) can be avoided to save the unnecessary computations. For instance, if the mining task is run on a reduced CFG as shown in Figure 4.2 that contains only relevant nodes, the mining task can be accelerated substantially. As we show in our case study (§4.4.1), for the API precondition mining task, we were able to reduce the overall mining task time by 50%.

4.2 Pre-Analysis and Program Compaction

Figure 4.3 shows an overview of our approach. The main three components of our approach are: i) a static analysis to extract rules, ii) a pre-analysis traversal to identify the analysis relevant nodes, and iii) a reduction to remove the analysis irrelevant nodes.

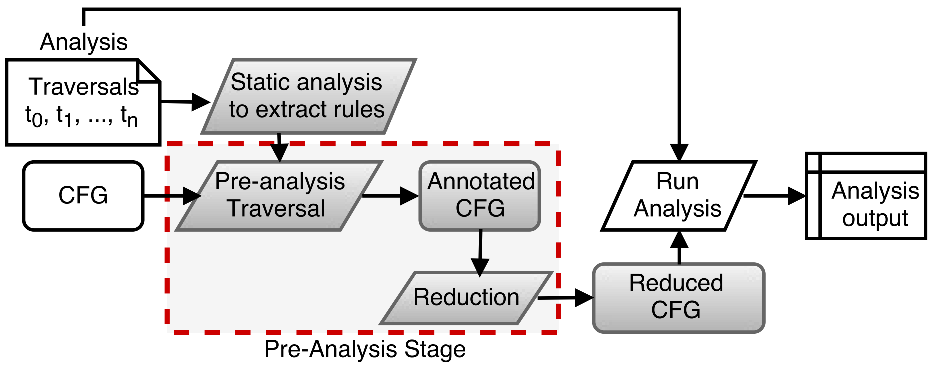


Figure 4.3: An overview of our approach.

Given a mining task and an input CFG, instead of running the mining task directly on the input CFG, we perform a lightweight pre-analysis that identifies the relevant nodes for the mining task and prunes the irrelevant nodes to obtain a reduced CFG. The pre-analysis stage is helped by a static analysis that provides a set of rules to identify relevant nodes. The rules set contains predicate expressions on the CFG node, which when evaluated on the nodes of the input CFG helps to identify relevant nodes. For example, consider the following rule: `[(node.expression == METHODCALL) && (node.expression. method == ‘‘substring’’)]`. This rule evaluates to *true* for all CFG nodes that contain `substring` method invocation expression. Inputs to the pre-analysis stage are: a CFG and a set of rules computed by our static analysis. The pre-analysis stage contains a traversal of the input CFG that runs the rules and annotates the analysis relevant nodes, and a reduction phase that prunes the analysis irrelevant nodes. Output of the pre-analysis stage is a reduced CFG. We run the analysis on the reduced CFG to produce the output.

Source code mining task. We assume the following formulation for a source code mining task: a source code mining task may contain a set of analyses. A source code analysis such as control flow analysis, data flow analysis, can be expressed as one or more traversals over the control flow graphs (CFGs). A traversal visits every node in the CFG and executes a block of code, often

Figure 4.5 The *mineT* traversal extracts all `substring` API call nodes and conditional nodes

```
// stores node ids of nodes with substring API call
apiCallNodes: set of int;

// stores conditions at nodes
conditionsAtNodes: map[int] of string;

hasSubstring := function(expr: Expression): bool {
  // checks if there exists a substring API method invocation
}

isConditional := function(node: CFGNode): bool {
  // checks if the CFG node has a conditional expression
}

mineT := traversal(node: CFGNode) {
  if (def(node.expr) && hasSubstring(node.expr)) {
    add(apiCallNodes, node.id);
  }
  if (isConditional(node)) {
    conditionsAtNodes[node.id] = string(node.expr);
  }
}
```

Figure 4.6 A visitor that invokes traversals on the CFG of every method in the project

```
visit(input, visitor {
  before method: Method -> {
    cfg := getcfg(method);

    traverse(cfg, ..., mineT);

    if (!isEmpty(apiCallNodes)) {
      traverse(cfg, ..., domT);
      traverse(cfg, ..., analysisT);
    }
  }
});
```

Figure 4.7: API Precondition mining Boa program showing *mineT* traversal and *main*

known as an analysis function. An analysis function takes a CFG node as input and produces an output for that node (aka analysis fact).[1]

Figure 4.7 and Figure 4.11 shows an example source code mining task written in Boa [22] (a domain-specific language for mining source code repositories) for mining API preconditions [48]. This source code mining task mainly contains three traversals: `mineT` (Figure 4.5), `domT` (Figure 4.9), and `analysisT` (Figure 4.10), and a main program (Figure 4.6) that invokes the three traversals on every method visited in the project. Note that, this source code mining task is run on all the projects in the dataset. We will use this example to describe the details of our technique.

As described in Figure 4.3, the first step in our approach is a static analysis that analyzes the mining task to extract a set of rules that describes the types of relevant nodes.

[1]This is a standard formulation of control and data flow analysis which can be seen in other analysis frameworks such as SOOT and WALA.

Figure 4.9 The *domT* traversal computes a set of dominators for every CFG node

```
domT := traversal(node: CFGNode): set of int {
  doms: set of int;

  doms = getvalue(node);

  if (!def(doms)) {
    if (node.id == 0) {
      s: set of int;
      doms = s;
    } else {
      doms = allNodeIds;
    }
  }

  foreach (i: int; def(node.predecessors[i])) {
    pred := node.predecessors[i];
    doms = intersection(doms, getvalue(pred));
  }

  add(doms, node.id);
  return doms;
}
```

Figure 4.10 The *analysisT* traversal computes the preconditions for every `substring` API call node

```
analysisT := traversal(node: CFGNode) {
   preconditions: set of string;

   if (contains(apiCallNodes, node.id)) {
     doms := getvalue(node, domT);

     foreach (dom: int = doms) {
       if (haskey(conditionsAtNodes, dom)) {
         add(preconditions, conditionsAtNodes[dom
                ]);
       }
     }
   }

   location := ...
   output[location] << preconditions;
}
```

Figure 4.11: API Precondition mining Boa program showing *domT* and *analysisT* traversals

4.2.1 Extracting Rules to Infer Relevant Nodes

Given a mining task that contains one or more traversals, our static analysis analyzes each traversal by constructing the control flow graph representation of the traversal and enumerating all acyclic paths in it.

Definition 1. *A* ***Control Flow Graph (CFG)*** *of a program is defined as* $G = (N, E, \top, \bot)$, *where* G *is a directed graph with a set of nodes* N *representing program statements and a set of edges* E *representing the control flow between statements.* $\top$ *and* $\bot$ *denote the entry and exit nodes of the CFG.*[2]

We use the notation $G_A = (N_A, E_A, \top_A, \bot_A)$ to represent the CFG of the analysis traversal, and $G = (N, E, \top, \bot)$ to represent the code corpora CFG that is input to an analysis.

[2] *CFGs with multiple exit nodes are converted to structured CFGs by adding a dummy exit node to which all exit nodes are connected.*

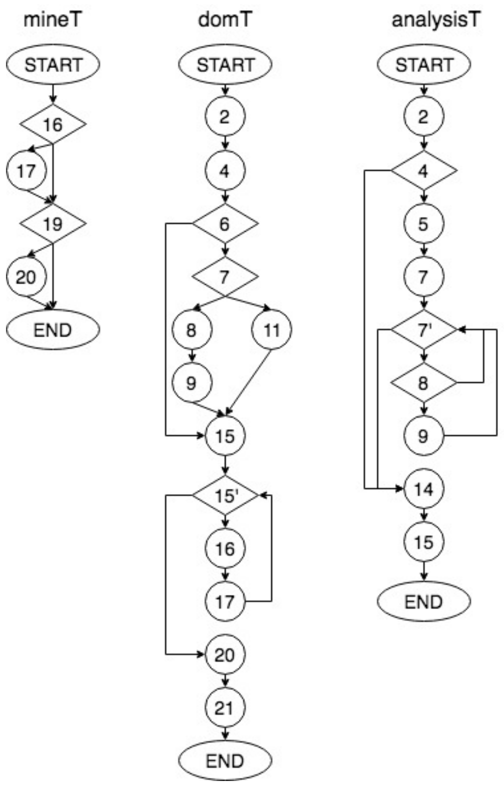


Figure 4.12: Control flow graphs of the three traversals shown in Figure 4.7 and Figure 4.11

A (control flow) **path** π of G_A is a finite sequence $\langle n_1, n_2, \ldots, n_k \rangle$ of nodes, such that $n_1, n_2, \ldots, n_k \in N_A$ and for any $1 \leq i < k$, $(n_i, n_{i+1}) \in E_A$, where $k \geq 1$, $n_1 = \top_A$ and $n_k = \bot_A$.
A set of paths, $\Pi = \{\pi_0, \pi_1, \ldots\}$ is a set of acyclic paths in the control flow graph G_A of the traversal (or an analysis function). An acyclic path contains nodes that appear exactly once except the loop header node that may appear twice.

The key idea of our static analysis is to select a subset of all acyclic paths based on the two conditions: 1) the path contains statements that provide predicates on the input variable (a CFG node), and 2) the path contains statements that contributes to the analysis output.

An input variable of a traversal (or an analysis function) is always a CFG node as described in the traversal definition. For example, in the `mineT` traversal definition, `node` is the input variable. Output variables of an analysis function can be one of the three kinds: 1) the variables returned by the traversals as outputs, for instance, `doms` in case of `domT` traversal, 2) the global variables, for instance, `apiCallNodes` and `conditionsAtNodes` in case of `mineT` traversal, or 3) the variables that are written to console as outputs, for instance, `preconditions` in case of `analysisT` traversal.

Every selected path produces a rule that is a path condition.[3] For example, consider the API precondition mining task shown in Figure 4.7 and Figure 4.11. This mining task contains three traversals (`mineT`, `domT`, and `analysisT`), where `mineT`, `domT`, and `analysisT` contains 4, 6, and 4 acyclic paths respectively as shown in Figure 4.12. Our static analysis analyzes each of these paths to select a subset of paths satisfying the two conditions described above.

Given a set of acyclic paths Π of a traversal (or an analysis function), and input/output variables of the traversal, Algorithm 1 computes a rules set R that contains path conditions extracted from the acyclic paths. For each path, Algorithm 1 visits the nodes in the path and checks if the node is a branch node. Algorithm 1 then fetches a list of aliases of the input variable (iv) to check if the branch node contains the input variable or its aliases (lines 8-9),[4] if so, it gets the predicate expression contained in the node using an auxiliary function *getPredicate* (line 7). The *getPredicate* auxiliary function can return either *null*, or *true*, or the predicate expression.

The *getPredicate* function returns *null* when the predicate expression of the branch node does not contain any expression that accesses the statement/expression using the input variable (iv) or its aliases. For example, the predicate expression "`contains(apiCallNodes, node.id)`" in line 4 in Figure 4.10 does not access the statement/expression, whereas, the predicate expression "`def(node.expr) && hasSubstring(node.expr)`" accesses the expression using the input variable *node*. Table 4.1 provides several examples of predicate expressions at branch nodes and the returned values by the *getPredicate* function for our example mining task described in Figure 4.7 and Figure 4.11. When *getPredicate* returns *null*, Algorithm 1 simply ignores the branch node and continues processing other nodes.

The *getPredicate* function returns *true* when there exists a predicate expression that contains an expression that accesses the statement/expression using the input variable (iv) or its aliases, but not

[3] A path condition is a conjunction of all node conditions and it represents a condition that must be true for the path to be taken at runtime.

[4] Before starting the rules extraction process, we first perform an alias analysis to determine aliases of input and output variables using a conservative linear time type-based alias analysis [21].

Algorithm 1: Extract rules from an analysis function

```
Input: Set of paths $\Pi$, String $iv$, Set<String> $ov$
Output: Rules set $R$
$R \leftarrow \{\}$;
foreach $\pi := (n_1, n_2, \ldots, n_k) \in \Pi$ do
    $pc \leftarrow true$;
    $failToExtract \leftarrow false$;
    $hasOutputExpr \leftarrow false$;
    foreach $n_i \in \pi$ do
        if $n_i$ is a branch node then
            $\alpha \leftarrow getAliasesAt(iv,\ n_i)$;
            if $getVariables(n_i) \cap \alpha \neq \phi$ then
                $pe \leftarrow getPredicate(n_i)$;
                if $pe$ is null then
                    continue;
                if $pe$ is true then
                    $failToExtract \leftarrow true$;
                    continue;
                if $n_{i+1}$ is a true successor then
                    $pc \leftarrow pc \wedge pe$;
                else
                    $pc \leftarrow pc \wedge \neg pe$;
        else
            $\beta \leftarrow getAliasesAt(ov,\ n_i)$;
            if $\beta \cap getVariables(n_i) \neq \phi$ then
                $hasOutputExpr \leftarrow true$;
                if $failToExtract$ is true then
                    return $\{true\}$;
    if $hasOutputExpr$ is true then
        $R \leftarrow R \cup pc$;
if $R$ is empty then
    $R \leftarrow R \cup true$;
return $R$;
```

Table 4.1: Predicate extraction scenarios for *getPredicate* auxiliary function

Branch Expression	Context Description	*getPredicate*
`node.id == 0`	Accessing the node id	`null`
`def(node.expr)`	Accessing the node expression	`node.expr`
`node.id == 0 && def(node.expr)`	Accessing node id and expression	`node.id == 0 && def(node.expr)`
`myfunc(node)`	`myfunc` accesses node statement/expression but does not contain any non-local access	`myfunc(node)`
`myfunc(node)`	`myfunc` accesses node statement/expression but contains some non-local accesses	`true`
`def(expr)`	`expr := node.expr`	`def(node.expr)`
`def(pred.expr)`	`pred := node.predecessors[i]`	`true`
`gVar > 10`	Accessing a global variable	`null`
`def(node.expr) && gVar > 10`	Accessing the node expression and a global variable	`true`
`def(node.expr) && getvalue(node)`	Accessing the node expression and querying the node value	`true`
`def(expr)`	`if () { expr := node.expr }`	`true`

all symbols in the predicate expression could be resolved. This could happen in several scenarios, for example, the predicate expression contains a global variable, or a local variable whose value could not be resolved, or a function call that in turn is not local (the invoked function has access to global variables). We have considered all possible scenarios that could happen in our analysis language and we list some of the examples in Table 4.1. The *getPredicate* function returning *true* is a failure case and it indicates that there exists a predicate but could not be extracted. We note this failure using `failToExtract` boolean (in Algorithm 1), which is used later in line 24 to terminate the rules extraction with a sound behavior that assumes that all nodes in the input CFG are relevant for the mining task.

The final case is when the *getPredicate* function is able to successfully extract the predicate expression from a branch node. Note that, the predicate expression returned in this case is fully resolved in terms of symbols and contains only the input variable-based statements/expressions. In this case, we determine whether to add the predicate expression or its negation based on the branch (true or false branch) that the successor node in the path belongs to. The path condition is the "*logical and*" of all predicate expressions in the path (lines 16-19). If the current visited node is not a branch node, then we get the aliases of the output variables *ov* and check if the node contains the output variable or its aliases (lines 21-22). The idea here is to keep the path conditions of only those paths that contributes to the output. At the end of visiting all nodes in the path, the computed path condition is added to the rule set R, if the current path contributes to the output (lines 26-27). We use the rule set R computed by Algorithm 1 to produce an annotated control flow graph (ACFG) in the pre-analysis traversal (§4.2.2).

4.2.1.1 Example

We provide a running example of our static analysis using the API precondition mining task shown in Figure 4.7 and Figure 4.11. This mining task contains three traversals: `mineT`, `domT`, `analysisT` and Algorithm 1 is applied to each traversal individually and the final rules set combines the rules extracted for each traversal.

Let us first consider the application of Algorithm 1 to `mineT` traversal. Figure 4.12 shows the CFG of `mineT` traversal, where the node numbers corresponds to the source line numbers in Figure 4.7 andFigure 4.11 that the `mineT` CFG contains 4 acyclic paths. For example, one such path is START $\rightarrow$ ◇16 $\rightarrow$ ○17 $\rightarrow$ ◇19 $\rightarrow$ END. Similarly, other three paths can be enumerated. The input variable to `mineT` traversal is *node* and the output variable set contains `apiCallNodes` and `conditionsAtNodes`. Algorithm 1 visits paths one-by-one and every node in the path is visited one-by-one. Consider the path described above (START $\rightarrow$ 16 $\rightarrow$ 17 $\rightarrow$ 19 $\rightarrow$ END), node 16 is a branch node and the expression in the branch node (line 16 in Figure 4.5) contains the input variable, hence *getPredicate* is invoked. As the expression in the branch node can be fully resolved, *getPredicate* successfully returns the predicate expression "`def(node.expr) && hasSubstring(node.expr)`". Also, the next node in the path is 17, a true successor, the predicate expression is added to *pc*. The next node visited is 17 and it contains an output variable `apiCallNodes` (line 17 in Figure 4.5), hence `hasOutputExpr` is set to `true`. The next node visited is 19, a branch node and it also contains the input variable, hence *getPredicate* is invoked, which returns the predicate expression "`isConditional(node)`". As the next node in the path is END, a false successor, the negation of the predicate expression is added to *pc*, which now becomes `true` $\wedge$ `def(node.expr) && hasSubstring(node.expr)` $\wedge\neg$`isConditional(node)`. As the path contains no more nodes and `hasOutputExpr` is `true`, *pc* is added to R. Similarly, all other three paths are processed and the final rules set after processing all four paths in the `mineT` traversal is:

```
{
true ∧
def(node.expr) && hasSubstring(node.expr) ∧
¬isConditional(node),
true ∧
¬def(node.expr) && hasSubstring(node.expr) ∧
isConditional(node),
true ∧
def(node.expr) && hasSubstring(node.expr) ∧
isConditional(node)
}
```

Figure 4.13: Rules set R produced by our static analysis for the API precondition mining task shown in Figure 4.7 and Figure 4.11

Running Algorithm 1 on paths of `domT` and `analysisT` does not add more rules to set R because all the branch conditions contain no expression that accesses statement/expression using the input variable *node* and *getPredicate* returns `null` for such cases. So, at the end of static analysis on the API precondition mining task we obtain a rules set R as provided above. Intuitively, the rules set R contains three rules that describes that an input CFG node is relevant if it contains: a substring method call, or a conditional expression, or both.

4.2.1.2 Soundness

The soundness of our static analysis (Algorithm 1) concerns the ability of the analysis to capture all analysis relevant nodes. Missing any analysis relevant node may lead to the removal of such node, which in turn leads to invalid analysis output. Hence, it is important that our static analysis extracts rules that can capture all analysis relevant nodes. Using the soundness arguments presented below, we argue that the rules collected by Algorithm 1 are sufficient to identify all analysis relevant nodes. We first define the analysis relevant nodes.

Definition 2. *A node is relevant for an analysis, if it takes a path in the analysis that produces some output. If* Π *is the set of all analysis paths,* $\Pi_o \subseteq \Pi$ *is the set of all paths such that* $\forall \pi := (n_1, n_2, \ldots, n_k) \in \Pi_o, \exists n_i$ *that produces some output.*

Table 4.2: Pruning-effective predicates for various control and data flow analyses

Analysis	**Pruning-Effective Predicates**
Available Expression (AE)	1) Node has expression, expression is not a method call, and expression contains variable access - Return the expression and accessed variables 2) Node has variable definition - Return the defined variable
Common Sub.Expr Elim (CSE)	Same as AE
Constant Propagation (CP)	1) Node has variable definition - Return the defined variable 2) Node has variable access - Return the accessed variables
Copy Propagation (CP')	Same as CP
Dead Code (DC)	Same as CP
Loop Invariant (LI)	1) Node has variable definition and node is part of a loop - Return the defined variable 2) Node has variable access and node is part of a loop - Return the accessed variables 3) Node has a loop statement such as FOR, WHILE, etc. - Return the node id
Local May Alias (LMA)	1) Node has variable definition - Return the defined variable
Local Must Not Alias (LMNA)	Same as LMA
Live Variables (LV)	Same as CP
Precondition Mining (PM)	1) Node has a call to the input API - Return the node id 2) Node has a condition - Return the conditional expression (or predicate)
Reaching Definitions (RD)	Same as CP
Resource Leak (RL)	1) Node has a call to Java resource-related API - Return the variable holding the resource
Safe Synchronization (SS)	1) Node has a call to Java Locking API - Return the variable holding the lock 2) Node has a call to Java Unlocking API - Return the variable holding the lock
Taint Analysis (TA)	1) Node has a call to Java input related API that reads external input to program variables - Return the affected variables 2) Node has a call to Java output related API that writes program variables to external output - Return the variables written to external output 3) Node has a variable copy statement - Return the copied variables
Up Safety (UP)	Same as AE
Very Busy Expression (VBE)	Same as AE

For Algorithm 1 to ensure that all analysis relevant nodes will be collected later in the pre-analysis stage, it must ensure that all paths that produces some output are considered.

Lemma 1. *All output producing paths (Π_o) are considered in Algorithm 1.*

Proof sketch. *Algorithm 1 iterates through every path and checks if there exists a statement/expression that writes to the output variable ov to determine if the path should be considered. Provided that getAliasesAt is a sound algorithm [21], we can see that any path that contains statements/expressions that writes to the output variable or its aliases are considered.*

We know that, a path is taken if the path conditions along the path are satisfied [65]. So, to ensure that all paths that produces some output are considered, Algorithm 1 must ensure that all path conditions along these paths are captured.

Lemma 2. *All conditions of the output producing paths (Π_o) are included in R.*

Proof sketch. *Given a set of paths that produce output (Π_o), the input variable iv, a sound algorithm getAliasesAt, the Algorithm 1 extracts all path conditions that involve the input variable iv or its aliases using getPredicate and adds to the rules set R. We argue that no path condition is missed. There are three cases to consider:*

- *Case 1. When no path contains path conditions (sequential traversal function code), then true is added as a rule to ensure that all paths are considered (lines 28-29).*[5]
- *Case 2. When there exists no path that contains conditions on the input variable iv or its aliases, or in other words, no path condition could be added to set R (the case where line 9 evaluates to false for all branch nodes in the path), true is added to ensure that all paths are considered (lines 26-27).*
- *Case 3. The cases where there exists some path conditions that involve the input variable iv or its aliases, but could not be extracted (when `failToExtract` is true), Algorithm 1 returns true (lines 24-25) to ensure that everything is considered relevant.*

In all other cases, the path conditions that involves the input variable iv or its aliases are added to the rules set R. We can see that, we do not miss any path that generates output and we collect either the path conditions on the input variable iv or *true*. Since we do not miss any path that generates output, the relevant nodes which takes the paths that generates output will also be not missed. This is presented as our soundness theorem next.

Theorem 3. *(Soundness). If $N_R \subseteq N$ is a set of all relevant nodes, $\forall n \in N_R, \exists r \in R$, such that evaluates$(r, n)$ is true, where the auxiliary function evaluates given a rule r (which is a predicate)*

[5] *It is not possible that some paths contain path conditions and other don't. It is either all paths contain path conditions or none because if a path condition along one path exists then the negation of that path condition also exists along other paths.*

and a CFG node, checks the satisfiability to return true or false. As it can be seen in evaluates is invoked in Algorithm 2, which is a dynamic evaluation step that simply runs each rule on the input CFG node to determine if the node is relevant or not. Since all the rules in the set R contain only the input variable evaluates function can never fail.

Proof sketch. *By Lemma 1 and Lemma 2, the theorem holds.*

4.2.1.3 Time Complexity

The time complexity of Algorithm 1 is $O(p * n)$, where p is the number of acyclic paths in the CFG of the analysis function (can grow exponentially, but finite) and n is the number of nodes in the CFG of the analysis function (the number of nodes can be considered nearly equal to the number of program statements). Prior to Algorithm 1 there are two key steps: computing the acyclic paths and computing the alias information. Our acyclic path enumeration step performs a DFS traversal of the CFGs of the analysis functions that has $O(n+e)$ time complexity in terms of number of nodes n and number of edges e. The alias analysis used in our approach is a type-based alias analysis [21] that has a linear time complexity in terms of the number of program statements. Overall, our static analysis overhead includes: 1) time for building the CFG of the analysis functions, 2) time for enumerating all acyclic paths, 3) time for alias analysis, and 4) time for extracting the rules (Algorithm 1). We present the static analysis overheads for all our 16 mining tasks used in our evaluation in §4.3.5.

4.2.1.4 Effectiveness

While the soundness of our static analysis (Algorithm 1) is guaranteed by both the sound alias analysis and the *getPredicate* auxiliary function, *getPredicate* requires that input-related predicates are expressed in the free form in analyses to be effective. In other words, *getPredicate* can effectively extract the input-related predicates, if they are (or can be) expressed in terms of the input variable (a CFG node). While this is a limitation of our static analysis, however we argue that it does not impose severe restrictions on the expressibility of code analyses. Often, the control and data flow analyses follow a well-defined lattice-based analysis framework in which a function that generates output for the given node is defined that depends on the statement or expression contained in the CFG node and outputs of neighbors (predecessors or successors). The part that extracts some information from the node is most likely check the type or kind of statement/expression contained in the node. For instance, to extract variable defined in a CFG node, the analysis will first check if the node contains a definition statement. If so, it extracts the defined variable. Table 4.2 list the input-related predicates for various analyses used in our evaluation along with the information extracted at nodes. As it can be seen in Table 4.2, for all the analyses the pruning-effective predicates are expressed on the statement and expression contained in the CFG node. It is quite possible to write an arbitrary control/data flow analysis where are approach can be ineffective in that Algorithm-1 considers every node relevant, but we have ensured soundness.

4.2.2 Annotated Control Flow Graph

In §4.2.1 we described our static analysis to extract rules. The output of our static analysis is a set of rules that contains predicate expressions on the CFG nodes. For instance, the rules set for API Precondition mining contains three rules as shown in Figure 4.13. Using the rules set, we perform a pre-analysis traversal that visits every node in the CFG, checks if there exists a rule $r \in R$ such that $evaluates(r, n)$ is *true*, and creates a set $N_R \subseteq N$ of nodes that contains nodes for which at least one rule in the rules set R evaluates to *true*. The auxiliary function *evaluates* given a rule r (which is a predicate) and a CFG node, checks the satisfiability to return *true* or *false*. The set N_R represents the probable set of analysis relevant nodes. Note that some special nodes are also added to the set N_R as shown in Algorithm 2. These are the entry and exit nodes, and the branch nodes.[6] Finally, the newly created set N_R is added to the CFG to create a modified CFG which we call and annotated control flow graph (ACFG).

Definition 4. *An* ***Annotated Control Flow Graph (ACFG)*** *of a CFG* $G = (N, E, \top, \bot)$ *is a CFG* $G' = (N, E, N_R, \top, \bot)$ *with a set of nodes* $N_R \subseteq N$ *computed using Algorithm 2.*

[6] At the time of pre-analysis, we consider all branch nodes as relevant for the analysis and later refine them to include only those branch nodes that have relevant nodes in at least one of the branches.

Algorithm 2: Pre-analysis traversal

```
Input: Control flow graph (CFG) $G$, Rules set $R$
Output: Annotated control flow graph (ACFG) $G'$
1  $N_R := \{\}$;
2  foreach node n in $G'$ do
3      if n is $\top$ or $\bot$ then
4          $N_R := N_R \cup n$;
5      if n is a branch node then
6          $N_R := N_R \cup n$;
7      foreach $r \in R$ do
8          if evalutes(r, n) then
9              $N_R := N_R \cup n$;
10             break;
11 add $N_R$ to $G'$;
12 return $G'$;
```

Definition 5. *Given an ACFG $G' = (N, E, N_R, \top, \bot)$, a node $n \in N$ is an **analysis relevant node** if:*

- *n is a $\top$ or a $\bot$ node,*
- *n is also in N_R, but not a branch node,*
- *n is a branch node with at least one branch that has an analysis relevant node.*

4.2.2.1 Example

Figure 4.14 shows the generated pre-analysis traversal for API precondition mining task. The pre-analysis mainly contains all the rules in the rules set R (shown in Figure 4.13) and a default rule to include the special nodes (START, END, and branch nodes). If any of the rules in the rules set R or the default rule is *true*, the current node is added to *rnodes* (a list of relevant nodes in the CFG).

```
_pre_analysis  := traversal (node: CFGNode) {
  // three rules from the rules set R
  if (true && (def(node.expr) && hasSubstring(node.expr)) &&
  ! isConditional (node)) {
    add(cfg.rnodes, node);
  }
  if (true && !(def(node.expr) && hasSubstring(node.expr)) &&
  isConditional (node)) {
    add(cfg.rnodes, node);
  }
  if (true && (def(node.expr) && hasSubstring(node.expr)) &&
  isConditional (node)) {
    add(cfg.rnodes, node);
  }

  // rule to add default nodes
  if (node.name == "START" || node.name == "END" || len(node.successors) > 1) {
    add(cfg.rnodes, node);
  }
}
```

Figure 4.14: The generated pre-analysis traversal for evaluating rules in the rules set for API precondition mining task

4.2.3 Reduced Control Flow Graph

Using the annotated control flow graph (ACFG) that contains a set N_R of probable analysis relevant nodes, we perform a sound reduction that refines the set N_R[7] and also removes the analysis irrelevant nodes[8] to create a reduced or compacted CFG called a reduced control flow graph (RCFG). An RCFG is a pruned CFG that contains only the analysis relevant nodes. An RCFG is constructed by performing a reduction on the ACFG.

Definition 6. *A **Reduced Control Flow Graph (RCFG)** of an ACFG $G' = (N, E, N_R, \top, \bot)$ is a pruned ACFG with analysis irrelevant nodes pruned. A RCFG is defined as $G'' = (N', E', \top', \bot')$, where G'' is a directed graph with a set of nodes $N' \subseteq N$ representing program statement and a set of edges E' representing the control flow between statements. $\top$ and $\bot$ are the entry and exit nodes. The edges $E - E'$ are the removed edges and $E' - E$ are the newly created edges.*

[7]Refining the set N_R involves removing those branch nodes that were added in Algorithm 2 but do not contain analysis relevant nodes in at least one branch. This is one of the conditions for analysis relevant nodes as defined in Definition 5.

[8]Analysis irrelevant nodes are those nodes in N that don't satisfy Definition 5.

Algorithm 3: Build RCFG

```
Input: Annotated control flow graph (ACFG) $G' := (N, E, N_R, \top, \bot)$
Output: Reduced control flow graph (RCFG) $G''$
1  $G'' \leftarrow G'$;
2  foreach node $n$ in $G''$ do
3      if $n \notin N_R$ then
4          Adjust($G''$, $n$);
5          remove $n$ from $G''$;
6  foreach node $n$ in $G''$ do
7      if $n$ is a branch node then
8          if SuccOf($n$) contains $n$ then
9              remove $n$ from SuccOf($n$);
10         if $n$ has only one successor then
11             Adjust($G''$, $n$);
12             remove $n$ from $G''$;
13 return $G''$;
```

Algorithm 4: A procedure to adjust predecessors and successors of a node being removed

```
1  Procedure Adjust(CFG $G$, CFGNode $n$)
2      foreach predecessor $p$ of $n$ in $G$ do
3          remove $n$ from SuccsOf($p$);
4          foreach successor $s$ of $n$ in $G$ do
5              remove $n$ from PredsOf($s$);
6              add $p$ to PredsOf($s$);
7              add $s$ to SuccsOf($p$);
```

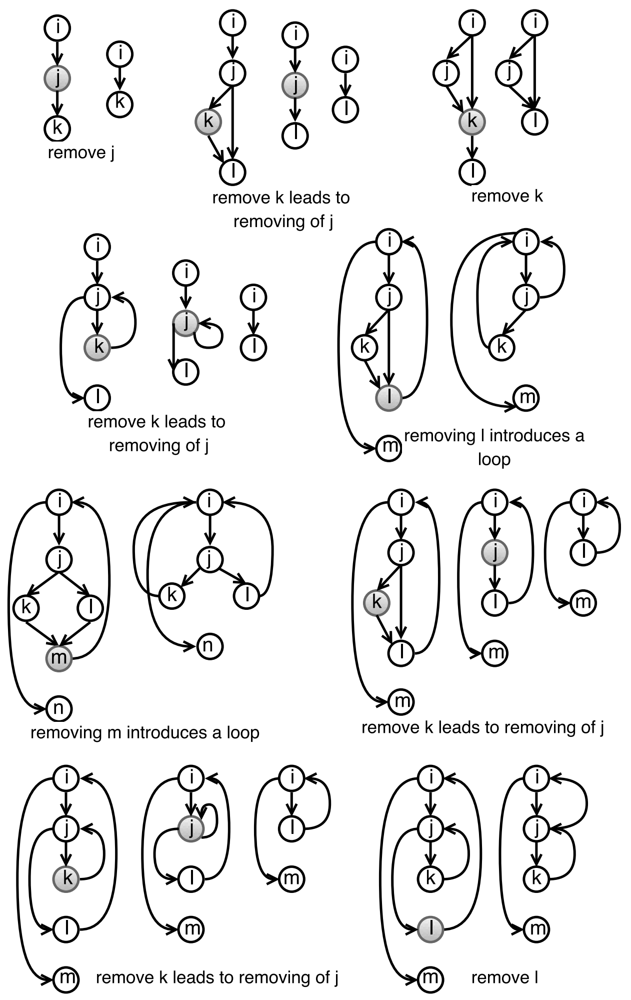

remove j
remove k leads to removing of j
remove k
remove k leads to removing of j
removing l introduces a loop
removing m introduces a loop
remove k leads to removing of j
remove k leads to removing of j
remove l

Figure 4.15: ACFG to RCFG reduction examples.

4.2.4 ACFG To RCFG Reduction

Algorithm 3 describes the reduction from ACFG to RCFG. The algorithm visits the nodes in the ACFG and checks if the node exists in N_R. The nodes that does not exists in N_R are pruned (lines 2-5). Before removing an analysis irrelevant node, new edges are created between the predecessors and the successors of the node that is being removed (line 4 and Algorithm 4). After removing all analysis irrelevant nodes, we pass through the RCFG and remove irrelevant branch nodes (lines 6-11). Irrelevant branch nodes are those branch nodes that have only one successor in the RCFG (this node is no longer a valid branch node). Note that, our definition of analysis relevant nodes (Definition 5) includes branch nodes with at least one branch that has an analysis relevant node.

Figure 4.15 shows several examples of the reduction. For instance, in the first example, where node $\textcircled{j}$ is being pruned, a new edge between $\textcircled{i}$ and $\textcircled{k}$ are created. Consider our second example, in which a node $\textcircled{k}$ that is part of a branch is being removed. Removing $\textcircled{k}$ leads to removing $\textcircled{j}$, which is a branch node with only one successor (an invalid branch). Next, consider an interesting case of removing node $\textcircled{l}$ in our fifth example. The node $\textcircled{l}$ has a backedge to loop condition node $\textcircled{i}$ and there are two paths from $\textcircled{j}$ to $\textcircled{l}$ ($\textcircled{j} \rightarrow \textcircled{l}$ and $\textcircled{j} \rightarrow \textcircled{k} \rightarrow \textcircled{l}$). Removing node $\textcircled{l}$ leads to an additional loop. This is because there existed two paths in the CFG from $\textcircled{j}$ to $\textcircled{i}$. Similarly, other examples show reductions performed by Algorithm 3.

4.2.5 Soundness of Reduction

We claim that our reduction from CFG to RCFG is sound, where soundness means that the analysis results for the RCFG nodes are same as the analysis results for the corresponding CFG nodes.

For flow-insensitive analysis, the analysis results depends only on the results produced at nodes. For flow-sensitive analysis, the analysis results depends on the results produced at nodes, and the flow of results between nodes.

It is easy to see that, for flow-insensitive analysis, the analysis results of RCFG and CFG should match, because all the nodes that produce results are retained in the RCFG.

For flow-sensitive analysis, the result producing nodes in RCFG and CFG are same. For the flow of results between nodes in RCFG and CFG to be same, the flow between nodes in the CFG should be retained for the corresponding nodes in the RCFG and no new flows should be created.

Definition 7. *Given any two nodes n_1 and n_2 of a CFG G, the analysis results can flow from n_1 to n_2, iff there exists a path $n_1 \rightarrow n_2$. This flow is represented as $n_1 \rightarrow^* n_2$.*

Lemma 3. *The flow between analysis relevant nodes in the CFG should be retained for the corresponding nodes in the RCFG. That is, for any two analysis relevant nodes n_1 and n_2 in the CFG G, if $n_1 \rightarrow^* n_2$ exists in G, then $n_1 \rightarrow^* n_2$ should also exists in RCFG G''.*

Proof sketch. For ensuring flows in the CFG is retained in the RCFG, every path between any two analysis relevant nodes in the CFG should have a corresponding path between those nodes in the RCFG. This is ensured in our reduction algorithm (Algorithm 3), where for removing a node, an edge from each predecessors to each successors is established (lines 4 and 10). If there existed a flow $n_1 \rightarrow^* n_2$ for a path $n_1 \rightarrow n_k \rightarrow n_2$ via an intermediate node n_k, the Algorithm 3, while removing n_k, establishes a path $n_1 \rightarrow n_2$ by creating a new edge (n_1, n_2), and hence the flow $n_1 \rightarrow^* n_2$ is also retained.

Lemma 4. *No new flows should be created between nodes in the RCFG that does not exists between the corresponding nodes in the CFG. For any two analysis relevant nodes n_1 and n_2 in the CFG G, if $n_1 \rightarrow^* n_2$ does not exists in G, $n_1 \rightarrow^* n_2$ should not exists in G''.*

Proof sketch. For ensuring no new flows are created, every path between any two nodes in the RCFG should have a corresponding path between those nodes in the CFG. This is ensured in our reduction algorithm (Algorithm 3), where for removing a node, an edge from each predecessors to each successors is established, *iff* there exists a path from the predecessor to the successor in the CFG. For any two analysis relevant nodes n_1 and n_2, a flow $n_1 \rightarrow^* n_2$ does not exists, if there is no path $n_1 \rightarrow n_2$. Algorithm 3 ensures that, while removing a node n_k in a path $n_1 \rightarrow n_k \rightarrow n_2$, a new edge between $n_1 \rightarrow n_2$ is created in G'', if there exists a path $n_1 \rightarrow n_2$ in G. This way Algorithm 3 guarantees that no new paths are created, and hence no new flows are created.

Theorem 8. *For flow-sensitive analysis, the analysis results for RCFG and CFG are same.*

Proof sketch. For flow-sensitive analysis, the fact that the result producing nodes in RCFG and CFG are same, and by Lemma 3 and Lemma 4, it follows that analysis results for RCFG and CFG are same.

4.2.6 Efficiency of Reduction

Our reduction algorithm has linear time complexity in terms of the CFG size. The reduction has two pass over the CFG nodes, where in the first pass the analysis irrelevant nodes are pruned (lines 2-5 in Algorithm 3) and in the second pass the irrelevant branch nodes are removed (lines 6-11).

4.3 Empirical Evaluation

We evaluate effectiveness, correctness, and scalability of our approach, specifically our evaluation addresses the following research questions:

1. How much reduction in the analysis time can be achieved by our technique that performs a pre-analysis to remove irrelevant code parts prior to running the analysis, when compared to a baseline that runs the analysis on the original source code? (§4.3.2)
2. How does our approach scale when the dataset size is increased uniformly? (§4.3.3)
3. What is the runtime overhead of our approach for performing a pre-analysis that identifies and removes the irrelevant nodes? (§4.3.4)
4. What is the compile-time overhead of the static analysis component of our approach that computes a rules set to identify relevant nodes? (§4.3.5)

Henceforth, we refer to our technique as `RCFG` and baseline as `Baseline`.

4.3.1 Experimental Setup

4.3.1.1 Analyses

Table 4.3 shows a collection of 16 flow analyses used to evaluate our approach. This collection mainly contains analyses that are used either in the ultra-large scale source code mining tasks or in the source code optimizations in compilers. For example, *Precondition Mining* (`PM`) is used for

Table 4.3: Reduction in analysis time and reduction in graph size for `DaCapo` and `SourceForge` datasets over 16 analysis. The column CFG provides the analysis time in `Baseline` approach and the column RCFG provides the analysis time in our approach. RCFG analysis time includes the annotation and reduction overheads. Column R provides the reduction in the analysis time and % R provides the percentage reduction in the analysis time. Under Graph Size (% Reduction), the columns N, E, B, L represents nodes, edges, branches, and loops. The table also provides the minimum, maximum, average, and median for both reduction (R) and percentage reduction (%R) in the analysis time.

		Analysis Time (seconds)								Graph Size (%Reduction)							
		DaCapo				SourceForge				DaCapo				SourceForge			
	Analysis	CFG	RCFG	R	%R	CFG	RCFG	R	%R	N	E	B	L	N	E	B	L
1	Available Expressions (AE)	13.31	6.37	6.93	52.09	137.93	68.83	69.10	50.10	41.45	46.94	39.01	36.15	41.83	47.01	37.22	35.85
2	Common Sub. Elimination (CSE)	13.96	6.25	7.72	55.26	157.90	75.66	82.24	52.08	41.45	46.94	39.01	36.15	41.83	47.01	37.22	35.85
3	Constant Propogation (CP)	9.84	10.31	-0.47	-4.75	315.59	316.09	-0.50	-0.16	18.03	20.27	13.40	4.74	16.04	17.42	8.01	2.62
4	Copy Propogation (CP')	13.29	13.29	0.00	0.00	380.33	380.04	0.29	0.08	18.03	20.27	13.40	4.74	16.04	17.42	8.01	2.62
5	Dead Code (DC)	13.66	15.52	-1.86	-13.64	501.96	494.94	7.02	1.40	18.03	20.27	13.40	4.74	16.04	17.42	8.01	2.62
6	Live Variables (LV)	4.83	5.20	-0.37	-7.70	173.42	172.35	1.07	0.62	18.03	20.27	13.40	4.74	16.04	17.42	8.01	2.62
7	Local May Alias (LMA)	5.83	2.41	3.42	58.72	177.73	72.11	105.62	59.42	42.28	47.93	40.31	38.92	42.47	47.80	38.61	38.29
8	Local Must Not Alias (LMNA)	5.80	2.08	3.72	64.18	158.49	54.37	104.12	65.69	42.28	47.93	40.31	38.92	42.47	47.80	38.61	38.29
9	Loop Invariant (LI)	11.57	4.23	7.34	63.42	318.17	122.66	195.51	61.45	42.58	48.28	40.74	39.27	42.66	48.04	38.91	38.68
10	Precondition Mining (PM)	41.49	2.25	39.24	94.58	912.26	34.12	878.14	96.26	62.42	70.27	57.90	56.27	63.98	71.90	59.25	56.91
11	Reaching Definitions (RD)	9.80	10.47	-0.67	-6.87	289.84	283.13	6.71	2.32	18.03	20.27	13.40	4.74	16.04	17.42	8.01	2.62
12	Resource Leak (RL)	0.03	0.00	0.03	93.33	0.36	0.29	0.07	19.10	63.34	74.72	70.02	73.82	67.66	78.09	74.30	70.27
13	Safe Synchronization (SS)	0.02	0.01	0.01	61.11	0.02	0.00	0.02	95.45	52.40	61.23	52.98	68.75	59.95	68.71	62.50	73.80
14	Taint (TA)	0.97	0.55	0.42	43.49	20.54	5.31	15.23	74.17	50.56	57.10	47.19	44.87	51.87	58.22	46.50	44.14
15	Upsafety (UP)	13.24	5.99	7.25	54.76	139.07	69.89	69.18	49.75	41.45	46.94	39.01	36.15	41.83	47.01	37.22	35.85
16	Very Busy Expressions (VBE)	14.15	7.79	6.36	44.95	266.88	141.26	125.62	47.07	38.62	43.60	35.08	22.02	39.38	43.99	32.94	22.08
			Min	-1.86	-13.64		Min	-0.50	-0.16								
			Max	39.24	94.58		Max	878.14	96.26								
			Avg	4.94	40.81		Avg	103.72	42.18								
			Med	1.92	53.43		Med	42.17	49.93								

mining API preconditions in [48]. Similarly, `RD`, `RL`, `SS`, `TA` are used in source code mining tasks, whereas analyses `AE`, `CSE`, `CP`, `CP'`, `DC`, `LV`, `LMA`, `LMNA`, `UP`, and `VBE` are mainly used in the compilers for optimizing source code, however `LMA` and `LMNA` are also used in source code mining tasks. We have used two main criterias to select analyses: complexity of analysis and amount of relevant code parts for the analysis. For example, `RD` is the simplest in terms of complexity that performs a single traversal, whereas, `PM` is the most complex analysis in our collection that performs three traversals (to collect API and conditional nodes, to perform dominator analysis, and to compute preconditions). For `RD`, program statements that define or use variables are relevant (which can be a major portion of the program) and for `PM`, statements that invoke APIs or conditional statements are relevant. We have also made sure to include analyses to obtain maximum coverage over the properties of flow analysis such as flow direction (*forward* and *backward*), merge operation (*union* and *intersection*), and the complexity of the data structures that store analysis facts at nodes (single value, set of values, multisets, expressions, set of expressions). Analyses are written using Boa [22], a domain specific language (DSL) for ultra-large-scale mining. While adopting the analyses, we have used optimal versions to the best to our knowledge.

4.3.1.2 Dataset

We have used two Boa datasets: `DaCapo` and `SourceForge`. The `DaCapo` dataset contains 304,468 control flow graphs extracted from the 10 GitHub Java projects [11], and `SourceForge` dataset contains over 7 million control flow graphs extracted from the 4,938 SourceForge Java projects.[9] The rationale for using two datasets is that, the `DaCapo` dataset contains well-crafted benchmark programs, whereas `SourceForge` contains arbitrary Java open-source programs. These two diverse datasets helps us validate the consistency of our results better.

[9] Both `DaCapo` and `SourceForge` datasets contain all kinds of arbitrary CFGs with varying graph sizes, branching factor, loops, etc.

4.3.1.3 Methodology

We compare the execution time of our approach (RCFG) against the execution time of the Baseline. The execution time of Baseline for an analysis is the total time for running the analysis on all the control flow graphs (CFGs) in the dataset, where the execution time of our approach for an analysis is the total time for running the analysis on all the reduced control flow graphs (RCFGs) in the dataset along with all the runtime overheads. The various runtime overheads in our approach includes the time for identifying and annotating relevant nodes and the time for performing the reduction of the control flow graph (remove irrelevant nodes) to obtain reduced control flow graphs (RCFGs). Note that the individual runtime overhead component times are reported separately in §4.3.4. We also discuss the compile-time overhead (for extracting rules) in §4.3.5. For measuring the execution times for Baseline and RCFG approaches, we use the methodology proposed by Georges *et al.* [30], where the execution times are averaged over three runs, when the variability across these measurements is minimal (under 2%). Note that, the compared execution times are for the entire dataset for each analysis and not for individual CFGs. Our experiments were run on a machine with 24 GB of memory and 24-cores, running on Linux 3.5.6-1.fc17 kernel.

4.3.2 Reduction In Analysis Time

We measured reduction in the analysis time of RCFG over Baseline. The results of our measurement is shown in Table 4.3. For example, running the Available Expressions (AE) analysis on DaCapo dataset that contains 304,468 CFGs took 13.31s in the Baseline approach and 6.37s in the RCFG approach. The reduction in the analysis time for AE is 6.93s and the percentage reduction in the analysis time is 52.09%.

Table 4.3 also shows the minimum, maximum, average, and median values of both reduction and % reduction in the analysis time. From these values it can be seen that, on average (across 16 analyses) our approach was able to save 5s on the DaCapo dataset and 104s on the SourceForge dataset. In terms of percentage reduction, on average, a 41% reduction is seen for the DaCapo dataset and a 42% reduction is seen for the SourceForge dataset. Our approach was able to obtain a maximum reduction for the Precondition Mining (PM) analysis, were on the `DaCapo` dataset, 39s (a 95%) was saved and on the `SourceForge` dataset, 878s (a 96%) was saved. Across `DaCapo` and `SourceForge` datasets, for 11 out of 16 analysis, our approach was able to obtain a substantial reduction in the analysis time. For 5 analysis (highlighted in gray in Table 4.3), the reduction was either small or no reduction was seen. We first discuss the favorable cases and then provide insights into unfavorable cases.

Table 4.4: Relevant source code parts for various analyses.

AE	1) Assignment statements with RHS contains variables, but not method call or new expression 2) Variable definitions
CSE	Same as AE
CP	1) Variable definitions 2) Variable accesses
CP'	Same as CP
DC	Same as CP
LV	Same as CP
LMA	Variable definitions
LMNA	Same as LMA
LI	1) Variable definitions of loop variables 2) Variable accesses of loop variables 3) Loop statements (FOR, WHILE, DO)
PM	1) String.substring API method calls 2) Predicate expressions
RD	Same as CP
RL	Contains read, write or close method calls from InputStream
SS	Lock.lock or Lock.unlock method calls
TA	1) Variable definitions with RHS contains Console.readLine or FileInputStream.read method calls 2) Variable definitions with RHS contains variable access 3) System.out.println(), FileOutputStream.close() calls
UP	Same as AE
VBE	1) Binop expression that have variable access, but are not method calls 2) Variable definitions

The reduction in the analysis time stems from the reduction in the graph size of `RCFG` over `CFG`, hence we measured the graph size reduction in terms of `Nodes`, `Edges`, `Branches`, and `Loops` for understanding the analysis time reductions. We accumulated these metrics over all the graphs in the datasets. The results of the measurement is shown in Table 4.3 under Graph Size (% Reduction) column. The reduction in graph size is highly correlated to the reduction in analysis time. Higher the reduction in graph size, higher will be the reduction in analysis time. For instance, consider the Precondition Mining (PM) analysis that had 95% and 96% reduction in the analysis time for `DaCapo` and `SourceForge` datasets respectively. For PM, the reduction in the graph size in terms of `Nodes`, `Edges`, `Branches`, `Loops`, were 62.42%, 70.27%, 57.9%, 56.27% for `DaCapo`, and 63.98%, 71.9%, 59.25%, 56.91% for `SourceForge` dataset. As it can be seen, for the Precondition Mining (PM) analysis, our approach was able to reduce the graphs substantially and as a result the analysis time was reduced substantially. To summarize the favorable results, it can be seen that for 11 of 16 analysis, our technique was able to reduce the analysis time substantially (on average over 60% reduction for 11 analyses, over 40% reduction over all analyses). This reduction can be explained using the reduction in graph size in terms of `Nodes`, `Edges`, `Branches`, and `Loops`. Further, Table 4.4 lists the relevant parts of the code for various analysis to give more insights into the reduction. The analysis that contains common statements as relevant parts sees less reductions. For instance, CP has variable definitions and variable accesses as relevant statements, which are very common in majority of the source code, hence sees very less reductions. Whereas, PM has `String.substring` API method calls and conditional expressions as relevant statements, which are not very common in majority of the source code, hence sees very high reductions.

For 5 analysis, the reduction in analysis time was small or no reduction is seen. The reduction in graph sizes for these 5 analysis are also low. These analysis are: constant propagation (CP), copy propagation (CP'), dead code (DC), live variables (LV), and reaching definitions (RD). One thing to notice is that, for all these five analyses, the set of relevant statements are same: variable definitions and variable accesses, which are frequent in any source code. Hence, for these analysis the graph size of the `RCFG` is similar to the `CFG`, and our technique could not reduce the graph size much. Since the graph size for `RCFG` and `CFG` are similar, their analysis times will also be similar (not much reduction in the analysis time). For some analysis, the `RCFG` approach time exceeds the `Baseline` approach time due to the additional overheads that `RCFG` approach has for annotating

```
public void rtrim() {
  int index = text.length();
  while ((index \> 0) && (text.charAt(
      index - 1) <='')) {
    index--;
  }
  text = text.substring(0, index);
}
```

```
private static Color parseAsSystemColor(String value)
            throws PropertyException {
  int poss = value.indexOf("(");
  int pose = value.indexOf(")");
  if (poss != -1 && pose != -1) {
    value = value.substring(poss + 1, pose);
  } else {
    throw new PropertyException("Unknown color format: " +
        value
    + ". Must be system-color(x)");
  }
  return colorMap.get(value);
}
```

```
private int[] parseIntArray(Parser p) throws IOException {
  IntArray array = new IntArray();
  boolean done = false;
  do {
    String s = p.getNextToken();
    if (s.startsWith("["))
      s = s.substring(1);
    if (s.endsWith("]")) {
      s = s.substring(0, s.length() - 1);
      done = true;
    }
    array.add(Integer.parseInt(s));
  } while (!done);
  return array.trim();
}
```

Figure 4.17: Three methods from the DaCapo dataset. Highlighted parts are the relevant statements for PM analysis.

and reducing the CFG to produce RCFG. From these unfavorable results we can conclude that for analyses for which the reduction in graph size is not substantial (or the relevant statements are common statements), `RCFG` may incur overheads leading to larger analysis time than `Baseline`. However, the overhead is not substantial. For instance, for `DaCapo`, CP: -4.75%, CP': 0.005%, DC: -13.64%, LV: -7.7%, and RD: -6.87%. For a larger dataset, such as `SourceForge`, the overheads are further small: CP: -0.16%, CP': 0.08%, DC: 1.4%, LV: 0.62%, RD: 2.32%. This indicates that, the analysis that are unfavorable to the `RCFG` approach, do not incur substantial overheads. Further, it is possible to detect whether an analysis will benefit from our pre-analysis that reduces the CFG size prior to running the analysis by computing the amount of relevant nodes during the pre-analysis traversal that evaluates the rules (extracted from our static analysis) to mark relevant nodes. We compute a ratio of number of relevant nodes to all nodes in our pre-analysis traversal and decide whether to prune prior to running the analysis or skip the pruning and simply run the analysis. Details of this experiment is reported in §4.3.6.

Same set of methods can demonstrate different amounts of relevant statements for various analyses. For example, consider the three methods shown in Figure 4.17. For PM analysis, the statements that invoke `substring` API method and conditional statements are the relevant statements. As highlighted in Figure 4.17, for PM, there are less relevant statements, hence more opportunity for reduction. Whereas, if we consider LV analysis, for which all statements that contain variable definitions and variable accesses are relevant, all statements in the three methods shown in Figure 4.17 are relevant, hence no reduction can be performed. The amount of reduction varies from analysis to analysis and it is mainly dependent on the kinds of relevant statements for the input analysis and the percentage of such statements in the input corpus.

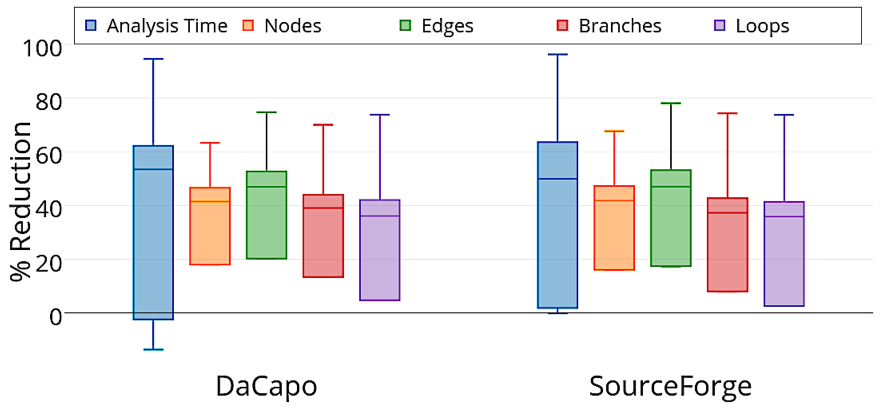


Figure 4.18: Box plots of % reduction in analysis time, and graph size metrics across 16 analysis for `DaCapo` and `SourceForge` datasets.

Figure 4.18 shows a boxplot of percentage reduction and graph size reduction for all our 16 analyses. As it can be seen, for `DaCapo` dataset, the % reduction in analysis time is in between

-2.37 to 62.26 (first and third quartiles) with median 53.42. For `SourceForge` dataset, the reduction in analysis time is in between 1.86 to 63.57 with median 49.92. We can draw following conclusions from the boxplots: i) for most analysis the reduction in the analysis time is substantial (indicated by the median and third quartile), and ii) for the analyses that does not benefit from the `RCFG` approach, do not incur too much overhead (indicated by the first quartile).

To summarize, our set of 16 analyses showed great variance in terms of the amount of reduction in the analysis time that our approach was able to achieve. The actual reduction in the analysis times were between several seconds to several minutes (the maximum was 15 minutes), however the percentage reduction in the analysis times were substantial (the maximum was 96% and average was 40%). Though, the reduction seems small, as we show in our case study of several actual ultra-large-scale mining tasks that uses several of these foundational analyses, when run on an ultra-large dataset (containing 162 million CFGs, 23 times larger than our `SourceForge` dataset) can achieve substantial reduction (as much as an hour). Further, the ultra-large-scale mining tasks that we target are often the exploratory analysis which requires running the mining tasks several times, where the results of the mining task are analyzed to revise the mining task and rerunning it, before settling on a final result. The time taken by the experimental runs becomes a real hurdle to trying out new ideas. Moreover, the mining tasks are run as part of the shared infrastructure like Boa [22] with many concurrent users (Boa has more than 800 users), where any time/computation that is saved has considerable impacts, in that, many concurrent users can be supported and the response time (or the user experience of users) can be significantly improved.

In terms of which analyses can benefit from our approach, we can see that the reduction in the analysis time depends both on the complexity of the analysis and the percentage of the program that is relevant for the analysis. For those analysis for which most of the program parts are relevant (or in other words, an input program cannot be reduced to a smaller program), our technique may not be very beneficial. For those analysis for which the relevant program parts are small, our technique can greatly reduce the analysis time. Also, for analysis that have simple complexity, for instance, analysis that perform single traversal (or parses the program once) may not benefit from our approach. Ideal scenarios for our technique to greatly help is when the analysis requires multiple traversals over the program (or program graphs) and the analysis relevant parts are small in the input programs.

4.3.3 Scalability

In this section, we measure the scalability of `Baseline` and `RCFG` approaches over increasing dataset sizes for our 16 analyses. For simulating the increasing dataset sizes, we have divided our 7 million CFGs of `SourceForge` dataset into 20 buckets, such that each bucket contains equal number of graphs with similar characteristics in terms of graph size, branches, and loops. Using the 20 buckets, we created 20 datasets of increasing sizes (D_0 to D_{19}), where D_i contains graphs in $bucket_0$ to $bucket_i$.

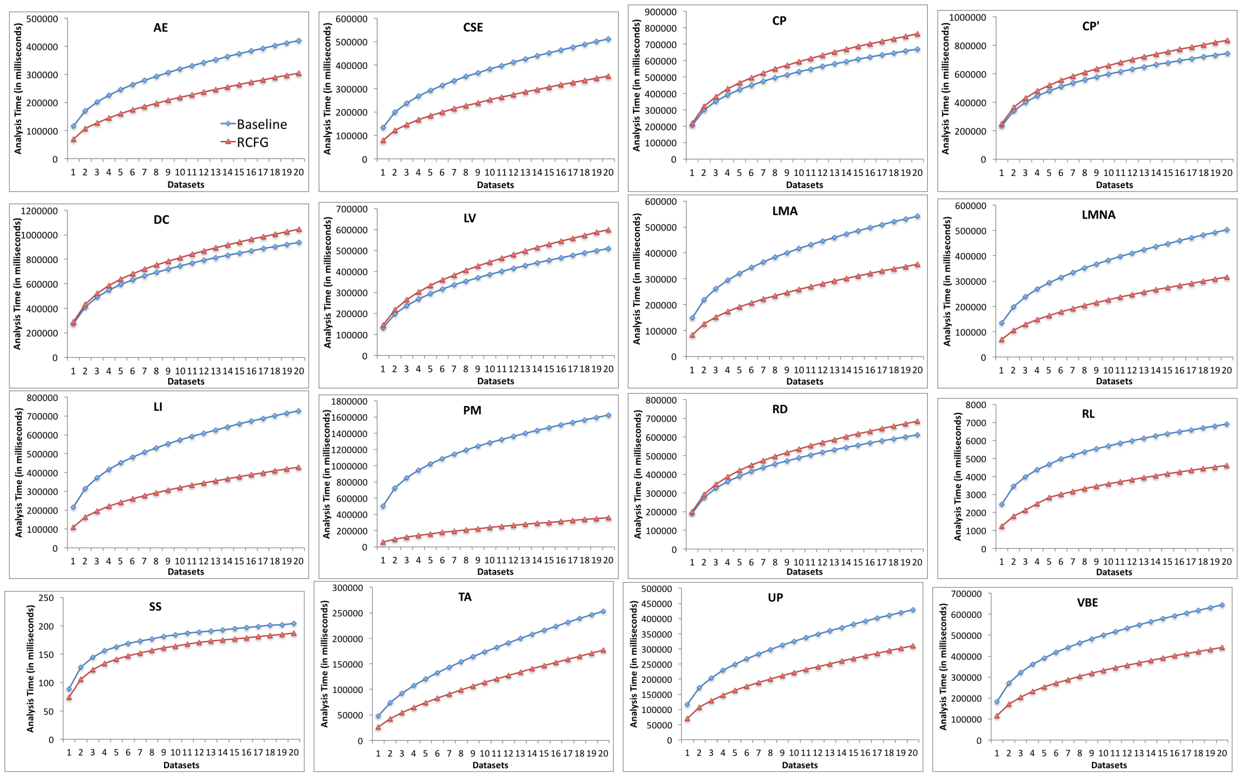


Figure 4.19: Scalability of 16 analyses for `Baseline` and `RCFG` approaches. Each chart contains two lines, `Baseline` (blue) and `RCFG` (red). Each line contains 20 data points representing 20 datasets.

We measure the analysis time of the Baseline and the RCFG approaches for all 16 analyses and plot the result in Figure 4.19. Our results shows that, as the dataset size increases, for both Baseline and RCFG, the analysis time increases sub-linearly. Our results also shows that, for increasing dataset sizes, RCFG performs better than Baseline for 11 of 16 analyses (where RCFG line is below Baseline line in the charts). For 5 analyses Baseline is better than RCFG. These analyses are the unfavorable candidates that we discussed previously (CP, CP', DC, LV, and RD).

4.3.4 Accuracy & Efficiency of Reduction

We evaluate the accuracy of the reduction by comparing the results of the RCFG and the Baseline approaches. We used DaCapo dataset for running the analyses and comparing the results. We found that, for all the analysis, the two results match 100%. This was expected, as RCFG contains all the nodes that produce output, RCFG retains all flows between any two nodes, and RCFG does not introduce new flows.

For evaluating the efficiency of reduction, we measured time for different components of the RCFG approach. The RCFG approach has three components: 1) *traversal* that annotates analysis relevant nodes, 2) *reduction* that prunes analysis irrelevant nodes, and 3) the actual *analysis*. Figure 4.20 shows the distribution of the RCFG time over these three components for DaCapo and SourceForge datasets over 16 analysis. The data labels provide the numbers for the RCFG time in seconds. From the results shown in Figure 4.20, it can be seen that, majority of the RCFG time is contributed by the actual analysis and not the overheads. We see, for some analysis, the *traversal* (that annotates the relevant nodes) contributes more than the actual analysis (RL in DaCapo, SS in SourceForge), however, for all analysis the reduction time is negligible, when compared to the actual analysis time. Further, we measured the time for each of the three components and aggregated it for all 16 analysis for all the graphs in the DaCapo and SourceForge datasets. For DaCapo, the traversal, reduction, and analysis times were 0.398, 0.047, and 92.268 seconds respectively, and for SourceForge, the traversal, reduction, and analysis times where 1.702, 1.055, 2288.295 seconds respectively. As we can see, the reduction time for both DaCapo and SourceForge datasets is very negligible when compared

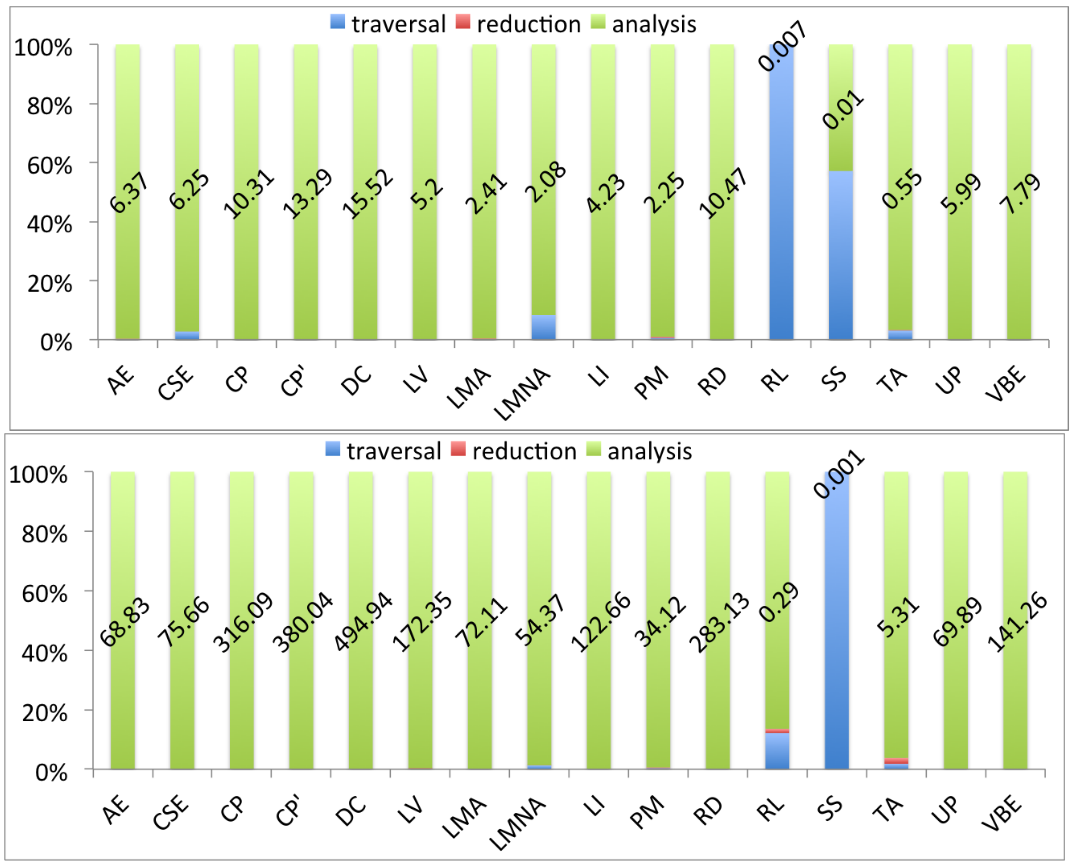


Figure 4.20: Distribution of `RCFG` analysis time into i) *traversal*, ii) *reduction*, and iii) *analysis* for DaCapo and SourceForge datasets. X-axis: 16 analyses, Y-axis: % Reduction, DataLabels: `RCFG` time in seconds.

to the analysis time for all 16 analysis. In summary, analysis results of `RCFG` and `Baseline` match 100%, indicating the soundness of the reduction. The negligible time for reduction when compared to actual analysis time, indicates that our reduction is efficient.

4.3.5 Overhead of Static Analysis

We presented our static analysis in section §4.2.1 and discussed its time complexity. In this section, we present our measurements of the overhead of the static analysis for all the 16 analyses. Table 4.5 presents these overheads along with some characteristics of the analyses, such as number of lines of code (Boa program LOC), number of analysis functions (or the CFGs), and number of paths (total number of paths in all the CFGs that are analyzed). Table 4.5 also presents the total overhead (T_{total}) of our static analysis along with the overheads of each of its components: CFG building time (T_1), path generation time (T_2), alias analysis time (T_3), and rules extraction time (T_4).

Table 4.5: Static analysis overheads. #LOC: Number of lines of code of the analysis, #F: Number of analysis functions (or CFGs), #P: Number of paths analyzed, T_1: CFG building time, T_2: Path enumeration time, T_3: Alias analysis time, T_4: Rules extraction time, $T_{total} = T_1+T_2+T_3+T_4$. The unit of T_1 - T_4 and T_{total} is milliseconds

Analysis	#LOC	#F	#P	T_1	T_2	T_3	T_4	T_{total}
AE	168	3	58	27	169	34	73	303
CSE	189	3	154	52	1032	49	117	1250
CP	188	5	102	24	547	29	62	662
CP'	110	3	98	6	650	59	96	811
DC	104	2	40	3	201	22	56	282
LV	101	2	16	15	28	19	16	78
LMA	118	2	1890	17	143476	32	541	144066
LMNA	121	2	53	31	193	29	39	292
LI	155	4	174	6	817	72	89	984
PM	102	3	28	24	100	15	66	205
RD	108	3	98	10	442	39	64	555
RL	133	2	27	5	48	19	15	87
SS	132	2	27	16	72	48	21	157
TA	106	1	936	28	29607	38	1077	30750
UP	185	3	58	7	194	27	52	280
VBE	159	3	58	7	188	33	62	290
			Median	15.5	197.5	32.5	63	297.5

Based on the median value over 16 analyses, the overhead is around 300ms. What this means is that, the compilation time of the analysis program is increased by 300 milliseconds. A majority of this overhead is contributed by the path enumeration phase. We can see the worst case overheads for two analyses: LMA and TA. In both the cases, the overheads are large due to the large amount of time required for path enumeration. These analyses have deeply nested branches and loops as part of their analyses functions which increases the number of paths and the path enumeration time. In summary, as our static analysis explores paths in the analysis, analysis with many paths may incur a non-negligible overhead, however this overhead is an one-time compilation overhead.

4.3.6 Configurable Pruning

Our performance results described in §4.3.2 showed that for five analyses (CP, CP', DC, LV, RD) our approach underperformed with respect to the `Baseline` and our rationale for this behavior was that, for these five analyses, the relevant statements were common statements (variable definitions and variable uses) and input CFGs could not be reduced significantly. We performed an experiment to skip reduction when the amount of irrelevant nodes is less than a configurable threshold (10% and 20%).

The methodology of this experiment was as follows. We augmented our pre-analysis traversal that marks the relevant nodes to also compute the percentage of irrelevant nodes. When the percentage of irrelevant nodes is less than a predefined threshold, we skip the reduction phase and directly run the analysis. We performed this experiment using two threshold values 10% and 20%, and we used `DaCapo` dataset for this experiment. The goal of this experiment was to evaluate whether skipping the reduction phase when there is not enough reduction opportunity can improve the overall analysis time of our approach. We experimented with only two values of the threshold because we suspect that higher threshold values will be not beneficial, as the reduction phase is very lightweight and has low overheads (as shown in Figure 4.20), skipping the reduction when there exists a decent amount of nodes could lead to a missed opportunity.

The result of this experiment is shown in Table 4.6. The table shows original `Baseline` and `RCFG` analysis times (in seconds) and analysis times for two variants of `RCFG` that skips reduction when the amount of irrelevant nodes is less than 10% and 20%. The table also shows the total number of CFGs on which the reduction was skipped in the two `RCFG` variants. The results clearly indicates that this enhancement to our approach is beneficial when we skip the reduction if the amount of irrelevant nodes is less than 10% (all the values under `RCFG` (0.1) column are less than `RCFG` column). While there were many graphs on which the reduction was skipped (the column `G`

Table 4.6: Reduction in the analysis time at reduction thresholds of 0.1 and 0.2.

	CFG	RCFG	RCFG (0.1)	RCFG (0.2)	#G (0.1)	#G (0.2)
AE	13.31	6.37	4.94 ↓	6.44 ↑	33941	45959
CSE	13.96	6.25	5.91 ↓	5.89 ↓	33941	45959
CP	9.84	10.31	9.92 ↓	10.36 ↑	107945	133573
CP’	13.29	13.29	12.99 ↓	13.16 ↑	107945	133573
DC	13.66	15.52	15.49 ↓	16.10 ↑	107945	133573
LI	4.83	5.20	4.52 ↓	4.71 ↑	107945	133573
LMA	5.83	2.41	2.39 ↓	2.33 ↓	33602	42661
LMNA	5.80	2.08	2.05 ↓	2.27 ↑	34129	44602
LV	11.57	4.23	4.20 ↓	5.11 ↑	107945	133573
PM	41.49	2.25	2.12 ↓	2.14 ↑	10271	10271
RD	9.80	10.47	10.40 ↓	10.54 ↑	107945	133573
RL	0.03	0.00	0.00 ↓	0.00 ↓	75	93
SS	0.02	0.01	0.00 ↓	0.02 ↑	2	2
TA	0.97	0.55	0.54 ↓	0.56 ↑	659	782
UP	13.24	5.99	5.72 ↓	5.96 ↑	33941	45959
VBE	14.15	7.79	7.71 ↓	5.40 ↓	33941	45959

(0.1)), the benefit obtained was less, mainly because the reduction phase itself is lightweight and less time consuming. The column RCFG (0.2) is variant of RCFG that skips reduction if the amount of irrelevant nodes were less than 20%. As we suspected, for this threshold, the RCFG analysis times were more than RCFG (0.1) for most analyses. This indicates that, skipping the reduction when the amount of irrelevant nodes were less than 20% was not beneficial. In summary, our approach can be augmented with a lightweight check-and-act step that skips reduction when the amount of irrelevant nodes is not substantial. This step can help to improve the performance of RCFG, however the threshold that decides whether to skip the reduction or not should not be too high (upto 10% was beneficial).

4.3.7 Summary and Discussions

In summary, our evaluation showed that across 16 representative analyses, 11 analyses could benefit significantly from our approach and 5 analyses could not benefit from our approach. On average our approach was able to reduce the analysis time by 40%. The two factors that decide whether an analysis benefits from our approach are the kinds of statements that are relevant for the analysis and the complexity of the analysis. The analyses for which the relevant statements are frequently occurring statements like variable definitions and variable accesses, our approach is not suitable. Our results also shows that a lightweight check-and-act step can help to detect and skip the reduction to mitigate the reduction overheads in case of unfavorable analyses. Our evaluation also studied the compile-time and runtime overheads of our technique, where the runtime overhead of the pre-analysis stage that marks relevant nodes and reduces the input graph is minimal, whereas the compile-time overhead can be non-trivial for analyses that have large number of nested conditions, however most analyses in our evaluation set did not suffer from the compile-time overhead.

There exists several software engineering (SE) tasks that require deeper analysis of the source code such as specification inference, API usage mining, discovering vulnerabilities, discovering programming patterns, defect prediction, bug fix suggestion, code search. The proposed technique can scale these SE tasks to large code base, which was found to be difficult previously. Moreover, exploratory analyses involving SE tasks requires multiple iterations of running SE tasks on large code bases and revising them before acceptable results are obtained. Such exploratory analyses were very time consuming in the past. The proposed technique can help to reduce the overall turnaround time of the run-revise-analyze cycle.

The proposed technique can help SE practitioners reduce the resource requirements of their infrastructure. In case of paid infrastructures, more tasks can be performed for the same infrastructure cost. For public free infrastructure providers, such as Boa, that supports running simultaneous tasks from several users, where slow running queries of certain user can impact the experience of all other users, the proposed technique can help to increase the number of simultaneous users. Also, many software engineering firms, e.g. Microsoft (CodeMine), ABB (Software Engineering Team Analytics Portal), deploy data-driven software analytics techniques to track and improve the various stages of the software development lifecycle within their organization. Running these analytics on thousands of developer data and activities over several days/months/years can be both time and resource consuming. The key concepts of our work can help to reduce the size of the input for various software analytics techniques by understanding and analyzing the task performed in them. As a result, more analytics can be deployed within the organizations for lesser cost.

The general idea of the proposed technique can also be used in other domains such as data-mining, where optimizing the data-mining queries is one way to accelerate data-mining, analyzing the data-mining queries and preprocessing the data to reduce the input space before running the data-mining queries could help to accelerate data-mining. Similarly, other domains could use the core ideas proposed in our work.

4.3.8 Threats to Validity

A threat to validity is for the applicability of our results. We have studied several source code mining tasks that perform control- and data-flow analysis and showed that significant acceleration can be achieved (on average 40%). However, these results may not be true for mining tasks that have completely different characteristics than the studied subjects. To mitigate this threat, we have included the tasks that have varying complexities in terms of the number of CFG traversals they require and the operations performed by them. We did not had to worry about the representativeness of our dataset that contains CFGs, because the dataset is prepared using the open source code repositories with thousands of projects and millions of methods, which often includes all kinds of complex methods. Further, the amount of reduction that our technique is able to achieve shows significant variations validating our selection of mining tasks. We haven't considered the mining tasks that requires global analysis such as callgraph analysis or inter-procedural control flow analysis. We plan to investigate them as part of our future work.

4.4 Case Studies

In this section, we show the applicability of our technique using several concrete source code mining tasks. We run these tasks on a humongous dataset containing 162 million CFGs drawn from the Boa GitHub large dataset. We use the distributed computing infrastructure of Boa to run the mining tasks. We profile and measure the task time and compare the two approaches, `Baseline` and `RCFG`, to measure the acceleration[10].

4.4.1 Mining Java API Preconditions

In this case study, we mined the API preconditions of all the 3168 Java Development Kit (JDK) API methods. The mining task contained three traversals. The first traversal collects the nodes with API method invocations and predicate expressions. The second traversal performs a dominator analysis on the CFG. The third traversal combines the results of the first two traversals to output the predicate expressions of all dominating nodes of the API method invocation nodes. The `Baseline` approach took 2 hours, 7 minutes and 40 seconds and the `RCFG` approach took 1 hour, 6 minutes and 51 seconds to mine 11,934,796 client methods that had the JDK API method invocations. Overall, a 47.63% reduction in the task computation time was seen. For analyzing the results, we also measured the percentage graph size reductions in terms of `Nodes`, `Edges`, `Branches`, and `Loops`. The values were, 41.21, 46.10, 37.91, and 37.57 respectively. These numbers indicate a substantial reduction in the graph size of `RCFG` when compared to `CFG`. The nodes that are relevant for the task are the nodes that had API method invocations and the conditional nodes that provide predicate expressions. All other nodes are irrelevant and they do not exist in RCFGs. One can expect that, in the client methods that invoke the JDK API methods, there are significant amount of statements not related API method invocations or predicate expressions, as we show an example client method in Figure 4.1. This explains the reduction in the task time.

4.4.2 Mining Java API Usage Sequences

In this case study, we mined the API usage sequences of all the 3168 Java API methods. An example API usage sequence is `Iterator.hasNext()` and `Iterator.next()`. For mining the usage sequences, the mining task traverses the CFGs to identify API method invocations. If a CFG contains two or more API method invocations, the mining task performs a data-flow analysis to

[10]The task time excludes the distributed job configuration time and the CFG building times, because these are same for both `Baseline` and `RCFG` approaches. However, the `RCFG` time includes all runtime overheads (annotation and reduction overheads).

determine the data-dependence between the API method invocations [1, 44]. Finally, the task outputs the API method call sequences that are data-dependent for offline clustering and determining the frequent API call sequences (the API methods that are used together). For this mining task, the `Baseline` approach took 1 hour, 33 minutes and 5 seconds and the `RCFG` approach took 1 hour, 16 minutes and 20 seconds to mine 24,479,901 API usage sequences. The API sequences generated as output by the mining task can be used for clustering and computing the frequently occurring API sequences. Overall, a 18% reduction in the task computation time is observed. The nodes that are relevant for this mining task are: the API method call nodes and the nodes that define the variables used by the API method calls. All other nodes are irrelevant. Here, the opportunity for reduction is less, as all the statements that contains variable definitions are relevant along with the API method call statements and the variable definitions are quite common in source codes. The percentage graph size reduction metrics supports our reasoning, where the values were: (`Nodes`, `Edges`, `Branches`, `Loops`) = (17.99, 18.32, 11.12, 5.21), which were on the lower side.

4.4.3 Mining Typically Synchronized Java APIs

In this case study, we mined the typically synchronized Java API method calls to help inform when the API methods are used without synchronization. In other words, the mining task determined which Java API method calls are protected using the lock primitives in practice by mining a large number of usage instances. The task first traverses the CFGs to determine if there exists safe synchronization using Java locking primitives (`java.util.concurrent.locks`). There exists a safe synchronization, if all the locks acquired are release along all program paths within the method. In the next traversal, the task identifies all API method calls that are surrounded with safe synchronization and output them to compute the most synchronized Java APIs. According to our mined results, the top 5 synchronized Java API method calls were:

1. `Condition.await()`
2. `Condition.signalAll()`
3. `Thread.start()`

4. `Iterator.next()`
5. `Iterator.hasNext()`

We were surprised to see `Thread.start()` in the top 5, however manually verifying many occurrences indicated that the source code related to the project' test cases often surround `Thread.start()` with locks.

For this mining task, nodes that are relevant are: `lock()` and `unlock()` API method call nodes, and the Java API method call nodes. For this task, the `Baseline` approach took 11.1 seconds and the `RCFG` approach took 8.45 seconds, i.e., a 23.72% reduction in the task computation time. The % graph size reduction metrics `Nodes`, `Edges`, `Branches`, and `Loops` were 32.12, 35.33, 25.21, and 18.46 respectively, which supports the reduction in the task time.

4.4.4 Mining Typically Leaked Java APIs

In this case study, we mined the typically leaked Java APIs. There exists 70 APIs for managing the system resources in JDK, such as `InputStream`, `OutputStream`, `BufferedReader`. A resource can be leaked if it is not closed after its use[11]. The mining task performs a resource leak analysis that captures if a resource used is not closed along all program paths. The task collects all the resources that are used in the program (as analysis facts), propagates them along the program using flow analysis, and checks if any resource is not closed at the program exit point. According to our mined results, the top 5 Java resources that often leaked were:

1. `java.io.InputStream`
2. `java.sql.Connection`
3. `java.util.logging.Handler`
4. `java.io.OutputStream`
5. `java.sql.ResultSet`

[11]Note that, both lock/unlock and resource leaks may go beyond a single method boundary. Such cases are not considered, as we do not perform an inter-procedural analysis.

The nodes that are relevant for this mining task are the resource related API method call nodes. All other nodes are irrelevant. For this task, the `Baseline` approach took 6 minutes and 30 seconds and the `RCFG` approach took 6 minutes and 18 seconds, i.e., only a 2.97% reduction in the task computation time. We expected significant reduction in the task time using RCFGs, however the results were contradictory. For further analysis, we measured the % graph size reduction metrics: `Nodes`, `Edges`, `Branches`, and `Loops`, whose values were, 42.31, 44.91. 38.81, 37.23 respectively, shows significant reduction only added to our surprise. Further investigation indicated that, although the RCFGs were much smaller than CFGs, the complexity of the mining task was small enough, such that the benefit obtained by running the task on the RCFGs were overshadowed by the overhead of the pre-analysis traversals and the reductions in our approach.

4.5 Related Work

There has been works that accelerates points-to analysis by performing pre-analysis and program compaction [5, 64]. Allen *et al.* [5] proposed a staged points-to analysis framework for scaling points-to analysis to large code bases, in which the points-to analysis is performed in multiple stages. In each stage, they perform static program slicing and compaction to reduce the input program to a smaller program that is semantically equivalent for the points-to queries under consideration. Their slicing and compaction can eliminate variables and their assignments that can be expressed by other variables. Smaragdakis *et al.* [64] proposed an optimization technique for flow-insensitive points-to analysis, in which an input program is transformed by a set-based pre-analysis that eliminates statements that do not contribute new values to the sets of values the program variables may take. In both the techniques, reduction in the number of variables and allocation sites is the key to scaling points-to analysis. The pre-analysis stage of both the techniques tracks the flow of values to program variables. Any analysis requiring analyzing program variables may benefit from these techniques. In contrast, our technique is more generic and it goes beyond analyses that track program variables and their values, e.g., tracking method calls, tracking certain patterns. The main advantage in our technique is that the relevant information for an input analysis is extracted automatically by performing a static analysis, making our technique applicable to a larger set of analyses that analyze different informations.

The concept of identifying and removing the irrelevant parts has been used in other approaches to improve the efficiency of the techniques [90, 20]. For example, Wu *et al.* [90] uses the idea to improve the efficiency of the call trace collection and Ding *et al.* [20] uses the idea to reduce the number of invocations of the symbolic execution in identifying the infeasible branches in the code. Both Wu *et al.*and Ding *et al.*identify the relevant parts of the input for the task at hand. The task in these approaches is fixed. In Wu *et al.*the task is call trace collection and in Ding *et al.*the task is infeasible branch detection using symbolic execution, while in our technique the task varies and our technique identifies the relevant parts of the input for the user task by analyzing the task.

There have been efforts to scale path sensitive analysis of programs by detecting and eliminating infeasible paths (pruning the paths) before performing the analysis [72, 37]. Such a technique filters and retains only relevant paths that leads to unique execution behaviors. In their technique a path is relevant if it contains event nodes and events are specified by the path sensitive analysis. For example, safe synchronization path-sensitive analysis has lock and unlock as events and any path that contains lock or unlock will be considered relevant. Compared to this approach, our approach is not limited to just event-based path sensitive analysis, but can be beneficial to flow-sensitive and path-insensitive analysis. Unlike the prior work that requires a user specify the list of events in their event-based path sensitive analysis, our technique can automatically derive the information with respect to what is relevant to the analysis by performing a static analysis.

Our work can also be compared to work in accelerating program analysis [38, 58, 92, 3]. Kulkarni *et al.* [38] proposed a technique for accelerating program analysis in Datalog. Their technique runs an offline analysis on a corpus of training programs and learns analysis facts over shared code. It reuses the learned facts to accelerate the analysis of other programs that share code with the training corpus. Other works also exists that performs pre-analysis of the library code to accelerate analysis of the programs that make use of the library code exists [58, 92, 3]. Our technique differs from these prior works in that, our technique does not precompute the analysis results of parts of the program, but rather identifies parts of the program that do not contributes to the analysis output and hence can be safely removed. While prior works can benefit programs that share some common code in the form of libraries, our technique can benefit all programs irrespective of the amount of shared code.

Reusing analysis results to accelerate interprocedural analysis by computing partial [31] or complete procedure summaries [57, 62] has also been studied. These techniques first run the analysis on procedures and then compute either partial or complete summaries of the analysis results to reuse them at the procedure call sites. The technique can greatly benefit programs in which procedures contain multiple similar call sites. In contrast, our technique can accelerate analysis of individual procedures. If the analysis requires inter-procedural context, our technique can be combined with the prior works, hence we consider our approach to be orthogonal to prior works with respect to accelerating inter-procedural analyses.

Program slicing is a technique for filtering a subset of the program statements (a slice) that may influence the values of variables at a given program point [88]. Program slicing has been shown to be useful in analyzing, debugging, and optimizing programs. For example, Lokuciejewski *et al.* [41] used program slicing to accelerate static loop analysis. Slicing cannot be used for our purpose, because the program points of interest (program statements) are not known. We require a technique like our static analysis that computes this information. Even if the statements of interest are known, slicing may include statements (affecting the values of variables at program points of interest) that may not contribute to the analysis output. Our technique only includes statements that contributes to the analysis output. Moreover, a program slice is a compilable and executable entity, while a reduced program that we produce is not. In our case, the original and the reduced programs produce the same result for the analysis of interest.

CHAPTER 5. COLLECTIVE PROGRAM ANALYSIS

In this chapter, we propose *collective program analysis* (`CPA`), a technique that leverages analysis specific similarity to scale source code analysis to large code bases.

Data-driven software engineering technique has gained popularity in solving variety of software engineering (SE) problems, such as defect prediction [18], bug fix suggestions [40, 39], programming pattern discovery [86, 74], and specification inference [54, 48, 95, 1]. The solutions to these SE problems generally require expensive source code analyses, such as data-flow analysis. Parallelization and task optimizations are the two widely adopted techniques to scale source code analyses to large code bases [22, 10, 32].

We propose *collective program analysis* (`CPA`), a complementary technique that leverages analysis specific similarity to scale source code analysis to large code bases. The key idea of `CPA` is to cluster programs based on analysis specific similarity, such that running the analysis on one candidate in each cluster is sufficient to produce the result for others. For instance, if a user wants to run an analysis to check for null dereference bugs in millions of programs, `CPA` would run the analysis on only the unique programs and reuse the results on others.

The three core concepts in `CPA` are the concept of *analysis specific similarity*, the technique of abstractly representing programs to reveal analysis specific similarity, and the technique of storing and reusing the analysis results between similar programs. Analysis specific similarity (or analysis equivalence) is about, whether two or more programs can be considered similar for a given analysis. Programs can be considered similar if they execute the same set of instructions in the analysis. For instance, if an analysis is about counting the number of assert statements, irrespective of how different the two programs are, if they have the same number of assert statements, they can be considered similar for the purpose of the assert counting analysis.

Code clones are the popular way of representing similar code [59]. Syntactic clones represent code fragments that are look alike (at token-level or AST-level), semantic clones represent code fragments that have similar control and data flow, functional clones represent code fragments that have similar input and output behaviors, and behavioral clones are the code fragments that perform similar computation. We did not use syntactic clones, because the benefits will be limited to copy-and-paste code. Semantic clones could not be used, because of lack of guarantee that analysis output will be similar. Moreover, semantically different code fragments may produce similar output for a given analysis and we would miss out on those. For the same reason, we also could not use the functional and behavioral clones. For these reasons, we go beyond the existing notion of similarity and define analysis specific similarity. We show that for analysis expressed in the lattice-based data-flow framework, we can use the transfer functions to identify analysis specific similarity.

Programs may have statements that are irrelevant for the given analysis. These are the statements that do not contributes to the analysis output. For identifying the analysis specific similarity it is necessary to remove the irrelevant statements and abstractly represent the reduced program. We use a sparse representation to remove the irrelevant statements without sacrificing the precision of the result [83]. Comparing sparse representations to determine analysis equivalence becomes a graph isomorphism problem for data-flow analysis that have sparse control flow graphs. We use a canonical labeling scheme to make this comparison efficient [93]. Using the labeling scheme we can produce unique patterns to facilitate the comparison. For reusing the results between the analysis equivalent programs, we store the results in an efficient key-value store based pattern database [50].

We evaluate our approach by measuring the reduction in the analysis time for 10 source code analysis tasks that involve data-flow analysis. We use two large datasets of programs: a `DaCapo` dataset containing DaCapo 9.12 benchmarks [11] and 287 thousand methods, a `SourceForge` dataset containing 4,938 open-source SourceForge projects and 6.8 million methods. When compared to a baseline that runs the analysis on every program in the dataset, `CPA` reduces the analysis time by 69% on average and when compared to another technique that removes irrelevant program statements prior to running the analysis, `CPA` reduces the analysis time by 36% on average. We also see a large amount of reuse opportunities in our datasets for almost all analyses.

5.1 Motivating Example

Consider a *Resource Leak* analysis that identifies possible resource leaks in the programs by tracking the resources that are acquired and released throughout the program by performing a flow analysis [80]. The analysis reports a problem when any acquired resource is not released on every path in the program.[1] If a user wants to run this analysis on a large code base that contains millions of methods, he would end up running the analysis on every method in the code base. An optimization can be performed to skip analyzing methods that do not contain resource related statements, however the methods that have resource related statements must be analyzed.

To illustrate, consider the three methods `writeObj`, `main`, and `loadPropertyFile` extracted from our `SourceForge` dataset shown in Figure 5.1. These three methods differ by syntax, semantics, functionality, and behaviorally, however for the resource leak analysis they all behave similar, because all of them acquire a resource and release along one of the execution paths, leading to a resource leak (In event of exception, the resource is not released). Although the three methods were similar for the resource leak analysis, all of them were analyzed to report leak. If there existed a technique that could capture this similarity, it could perform the analysis on any one of these three methods and simply return *true* for the other two methods, indicating a resource leak.

When analyzing a small number of methods or a handful of projects, there may not exist a lot of analysis specific similarity between the source code elements, such as methods, however in case of large code bases, a large amount of analysis equivalent methods exists. For instance, the resource usage pattern leading to a leak shown in Figure 5.1 exists in 5151 methods in our `SourceForge` dataset. This means that, we only need to run the resource leak analysis on one method out of 5151 and reuse the result (in this case whether a leak exists or not) for the remaining 5150 methods.

[1]There exists a finite number system resources, such as files, streams, sockets, database connections, and user programs that acquire an instance of a resource must release that instance by explicitly calling the release or close method. Failure to release the resource could lead to resource leak or unavailability.

```
public void writeObj(String filename) {
  try {
    FileWriter file = new FileWriter(filename);
    for (..)
      file.write(...);
    ...
    file.close();
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

```
public static void main(String[] args) {
  try {
    ...
    OutputStream out = new FileOutputStream("...");
    ...
    out.close();
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

```
public void loadPropertyFile(String file,...) {
  try {
    try {
      ...
    } catch (Exception e) {}

    BufferedInputStream bis = new Buffered...
    ...
    bis.close();
  } catch (Exception ex) {
    throw new WrappedRuntimeException(ex);
  }
}
```

Figure 5.1: The three methods extracted from our `SourceForge` dataset that have different resource usage patterns, however there exists a similarity that all of them may lead to a resource leak.

The `SourceForge` dataset contains a total of 82,900 methods that have resource related code out of 6,741,465 methods in the dataset. We were able to see 5689 unique patterns and the leak pattern discussed here appears in the top 3 patterns. Likewise, when analyzing large code bases, there exists a large amount of analysis equivalent codes and a large percentage of reuse opportunity to utilize for accelerating the overall analysis of large code bases.

5.2 `CPA`: Collective Program Analysis

Figure 5.2 provides a high-level overview of collective program analysis (`CPA`). Given a source code analysis that needs to be run on a large dataset of programs, we first run a light-weight pre-analysis on each program that identifies and removes irrelevant parts of the program, and labels the remaining statements (the analysis relevant statements). This labeled compact program is called a sparse representation. We then generate a pattern for the sparse representation and check the pattern against a pattern database. If the pattern is not found, the analysis is run on the sparse representation to produce the result, whereas if the pattern already exists, then the stored result is extracted and returned as the analysis output.

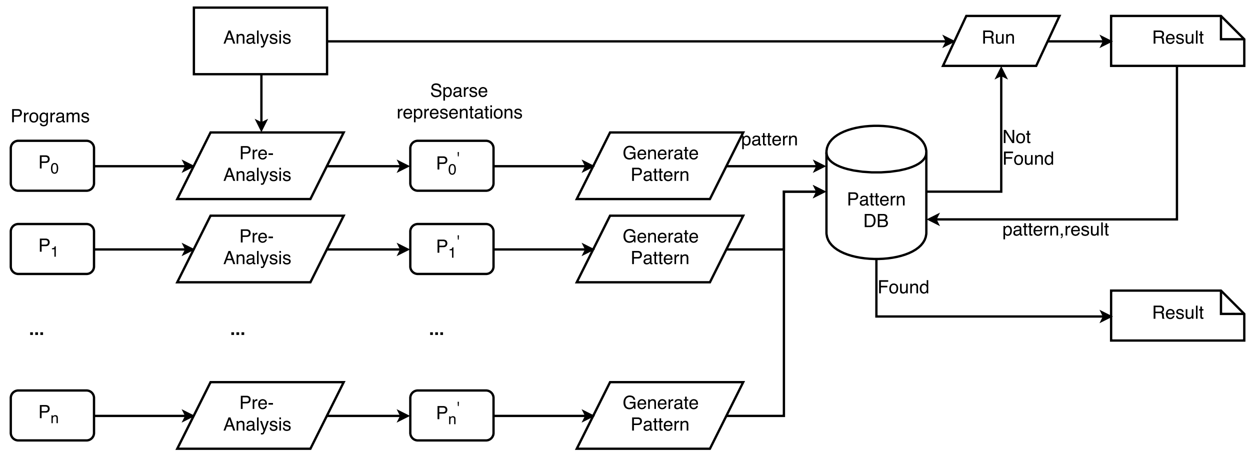


Figure 5.2: An overview of Collective Program Analysis (CPA)

While our solution looks intuitive, there exists several challenges in realizing `CPA`. For example, how to generate a sparse representation given an analysis and a program, how to generate a pattern for sparse representation such that analysis equivalent sparse representations can be identified, and how to utilize the sparse representation to reuse the analysis results. We will describe these challenges and our solutions in detail. But, first we describe the analysis model under assumption.

5.2.1 The Analysis Model

A source code analysis can be performed either on the source code text or on the intermediate representations like abstract syntax trees (ASTs), control flow graphs (CFGs), etc. A control and data flow analysis is performed on a CFG and is often expressed using the lattice-based data-flow framework [49]. In this framework, a data-flow analysis is described by defining a lattice, which describes the set of values to be associated with program statements, and a set of transfer functions that describes how each program statement transforms the input values to output values.[2] Two sets of data-flow values are maintained at each node: `IN` and `OUT` that describes the input and output values at each node. The data-flow analysis solves a set of flow equations involving the two sets `IN` and `OUT`, and transfer functions. Based on the data-flow values computed at the nodes, assertions can be made about the program behavior. For example, the *Resource Leak* analysis described in the motivation section maintains a set of variables representing the resources as data-flow values and it has mainly three kinds of transfer functions for handling resource `acquire`, resource `release`, and resource `copy/aliasing`.[3] From hereon, whenever we refer to analysis, we mean the data-flow analysis expressed in this framework.

Definition 9. *A* ***Control Flow Graph*** *of a program is a directed graph $CFG = (N, E, n_0, n_e)$, with a set of nodes N representing program statements and a set of edges E representing the control flow between statements. n_0 and n_e denote the entry and exit nodes of the CFG.*[4]

5.2.2 Sparse Representation

Given an analysis and a large set of programs, we perform a pre-analysis on each program to produce a sparse representation. A sparse representation is a reduced program that contains only the statements that are relevant for the analysis. Intuitively, a program statement is relevant for an analysis, if it contributes to the analysis output (or generates some information). With respect to the analysis model under consideration, the relevancy is defined as follows:

Definition 10. *A program statement is* ***relevant*** *for an analysis, if there exists a non-identity transfer function for that statement in the analysis. That is, if the analysis has defined a transfer function f_i^k for statement i, where k represents the transfer function kind and $f^k \neq \iota$, then i is relevant for the analysis. In the data-flow analysis model there always exists an identity transfer*

[2]A merge operator that describes how two data-flow values can be combined, a partial order that describes the relation between values, and top and bottom values are also provided. However, for describing `CPA`, transfer functions are sufficient.

[3]We ignore the method calls for simplifying the description, however in our implementation the method calls are over-approximated.

[4]*A CFG may contain multiple exit points, however we connect them to a auxiliary exit node.*

function ι along with the user defined transfer functions to represent those statement that have no effect on the analysis output.

Definition 11. *Given a program P with a set of statements S, a sparse representation is a tuple, $< P', M >$, where P' contains a subset of the statements $S' \subseteq S$, such that $\forall i \in S'$, i is a relevant statement. $M : S \rightarrow f^k$ is a map that provides the information about the kind of the transfer function that is applicable to each relevant statement i in set S'.*

As `CPA` takes data-flow analysis and the control flow graphs (CFGs) of programs, we have to generate the sparse representations of CFGs. For this purpose, we utilize a prior work that proposes reduced control flow graphs (RCFGs) [83]. In a nutshell, a RCFG is a reduced CFG that contains only those nodes for which there exits a non-identity transfer function in the analysis. RCFG is constructed using a pre-analysis that takes an analysis specification and a CFG as input, extracts all the conditions for the analysis transfer functions and checks the conditions against the CFG nodes to identify analysis relevant nodes. We extended RCFG to also store the kind of the transfer function that are applicable to CFG nodes as special properties of nodes. This information is required in a later stage of the `CPA` to generate patterns for CFGs.

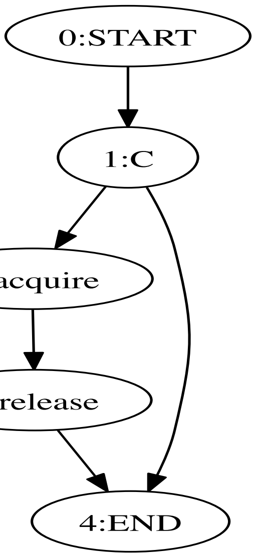


Figure 5.3: Sparse representation of `writeObj` method shown in Figure 5.1.

To provide a concrete example of a sparse representation, let us revisit the *Resource Leak* analysis described in our motivation and the `writeObj` method shown in Figure 5.1. The *Resource Leak* analysis contains three kinds of transfer functions: `acquire`, `release`, and `copy`. Using the transfer functions, we can identify the relevant statements in the `writeObj` method. The relevant statements for this method are at line 3 and line 7, because line 3 creates a `FileWriter` resource variable and it has an associated transfer function `acquire`, and line 7 releases the resource by invoking `close` method and it has an associate transfer function `release`. All other statements do not have an associated transfer function and hence are considered irrelevant and removed except some special nodes, such as `START` and `END`. RCFG also retains the branch nodes that have at least one successor with a relevant statement. The resulting sparse representation is as shown in Figure 5.3. This graph is a RCFG of the `writeObj` method. It contains two nodes 3 and 7 that have non-identity transfer functions `acquire` and `release` respectively and a special branch node marked `C`.

5.2.3 Analysis Equivalence

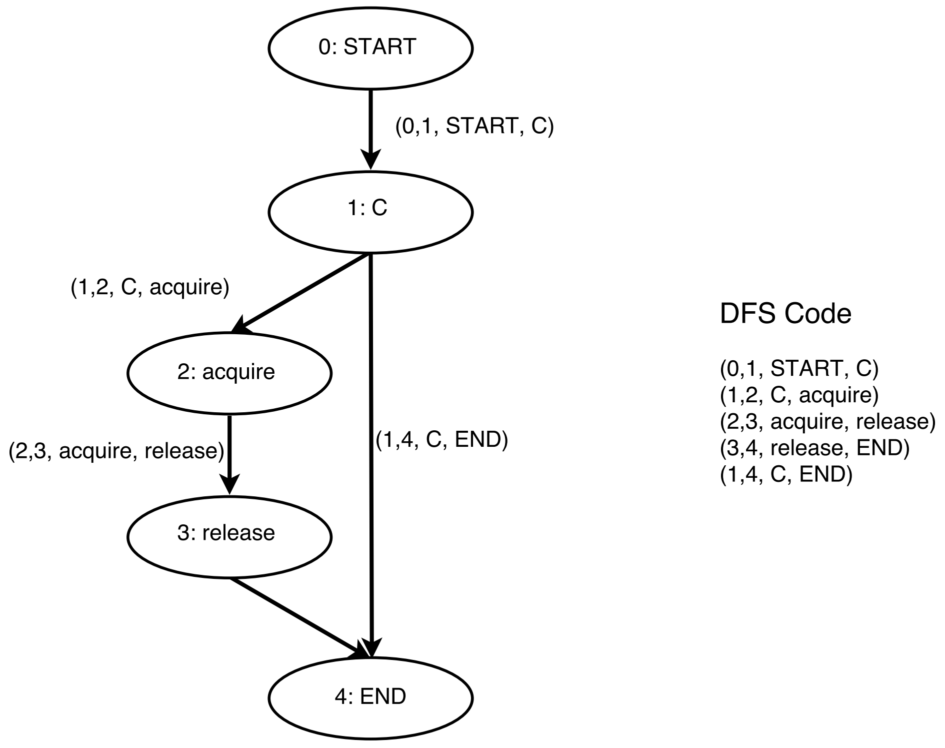
Given the sparse representations of programs, our next problem is to find similarities between them. In case of sparse representations of CFGs, finding similarities is a graph isomorphism problem with respect to certain labeling scheme. A prior work gspan [93] has proposed using a Depth-first search (DFS) code as the unique canonical label for graphs to find isomorphism. We utilize the DFS code technique for obtaining the canonical form of the sparse representation.

Given a graph (directed or undirected) with nodes and edges, a **DFS Code** is an edge sequence constructed based on a linear order, $\prec_T$ by following rules (assume $e_1 = (i_1, j_1), e_2 = (i_2, j_2)$, where e_1, e_2 are edges and i, j are node ids):

- if $i_1 = i_2$ and $j_1 < j_2$, $e_1 \prec_T e_2$,
- if $i_1 < j_1$ and $j_1 = i_2$, $e_1 \prec_T e_2$, and
- if $e_1 \prec_T e_2$ and $e_2 \prec_T e_3$, $e_1 \prec_T e_3$.

Each edge in the DFS code is represented as a 5-tuple: ¡i,j,l_i,$l_{(i,j)}$,l_j¿ where i, j are node ids, l_i, l_j are node labels, and $l_{(i,j)}$ represents the edge label of an edge (i, j).

In the DFS code that we generate, we use only 4-tuple and ignore the edge label $l_{(i,j)}$, because it is only required for multi-edge graphs and CFGs are not multi-edge graphs. For node labels, we use the transfer function kinds. For instance, for the *Resource Leak* analysis, we use `acquire`, `release` and `copy` for node labels. Note that, every node in the sparse representation of the CFG has an associated non-identity transfer function. Figure 5.4 shows the DFS code constructed for the sparse graph of `writeObj` method shown in Figure 5.1. As shown in the figure, each edge is represented as a 4-tuple. For instance, edge from node 2 to node 3 is represented as (2, 3, `acquire`, `release`). By following the the $\prec_T$ order, we obtained the DFS code shown in the figure.



DFS Code

(0,1, START, C)
(1,2, C, acquire)
(2,3, acquire, release)
(3,4, release, END)
(1,4, C, END)

Figure 5.4: DFS code for the sparse control flow graph of the `writeObj` method shown in Figure 5.1.

An undirected graph could have several DFS codes (based on the starting node) and the minimum DFS code provides the canonical label, such that if two graphs G and G' that have the same minimum DFS codes are isomorphic to each other [93].

Theorem 12 (Isomorphic graphs produce equal DFS code)**.** *Given two graphs G and G', G is isomorphic to G' iff, their minimum DFS codes are equal.*

Proof. *The proof is based on [93].*

Theorem 13 (A CFG has a unique, minimal DFS code)**.** *A CFG always has a single DFS code that is minimum, because there exists a single start node and the edges are directed.*

Proof Sketch. *The proof is by contradiction. Consider that a CFG has two DFS codes C_1 and C_2. Both C_1 and C_2 must have the same first edge because there exists only one start node for a CFG. From the destination node of the first edge, C_1 and C_2 might have two different edges. However, this is not possible because the next edge is picked by following the linear order $\prec_T$, which is deterministic and it always picks the same edge. If this process of picking the next edge is continued to form the edge sequences in C_1 and C_2, we can see that both C_1 and C_2 must have the same edges in the same order in the sequences.*

Given that we have a mechanism to encode the sparse graphs as DFS codes, we define analysis equivalence of sparse representations of CFGs as graphs with same DFS code.

Definition 14. ***(Analysis Equivalence)*** *Two CFGs G_1 and G_2 are equivalent for a given analysis or* analysis equivalent *if the DFS codes of the corresponding sparse graphs are same.*

To provide a concrete example, consider the *Resource Leak* analysis and the three methods `writeObj`, `main`, and `loadPropertyFile` shown in Figure 5.1. Although the CFGs of these three methods are different, after removing all the irrelevant nodes, the sparse graphs obtained are same, as shown in Figure 5.3. For this sparse graph, the DFS code constructed is shown in Figure 5.4. As these three methods have the same DFS code, their sparse representations are analysis equivalent.

An important property of the analysis equivalent sparse representations is that the analysis output for these graphs are similar. When we say similar, we mean that the analysis executes exactly same set of instructions to compute results for nodes in the two sparse representations. We formulate this property as a theorem and provide proof sketch.

Theorem 15 (Analysis equivalence implies result similarity)**.** *Two analysis equivalent sparse representations produces similar results.*

Proof Sketch. *Two analysis equivalent sparse representations will have same number of nodes and each node is associated with the same kind of transfer function, which means that the result produced at nodes are similar (by the application of the transfer functions). The flow of results between the nodes in two sparse representations is also similar because the edges between the nodes in the two sparse representations are also similar. This means that, if the two sparse representations starts off with an initial state (often top element of the data-flow analysis), they must produce similar results.*

5.2.4 Leveraging Analysis Specific Similarity

In the previous section, we described a technique for identifying the analysis specific similarity between programs, the final step of `CPA` is to cluster programs and reuse the analysis results. We use a pattern database [50] to store the DFS codes of the sparse representations as keys and analysis results as values. As described in our overview diagram shown in Figure 5.2, after producing the DFS codes for sparse representations, our approach first checks whether a result is already present in the database. We define the presence of the DFS code as a *hit* and the absence as *miss*. In case of a hit, we simply return the stored result. In case of a miss, we run the analysis on the sparse representations to produce the result and store the result along with the DFS code into the database for future use.

We require that analysis results of sparse representations cannot contain any concrete program data, for instance, variable names. While the analysis can compute any concrete result for each program statement, the analysis results for the sparse representation must be free from the concrete program data. For example, *Resource Leak* analysis collects and propagates the variable names of resource variables, however at the end it produces a boolean assertion indicating “whether a resource leak exists in the program?". This is not a severe restriction for CPA to be applicable, because the analyses can still compute program specific outputs, however the final output has to be an assertion or any result that is free from program specific data.

5.3 Evaluation

The main goal of our approach is to accelerate large scale source code analysis that involves control and data-flow analysis, hence we mainly evaluate the performance. However, we also present our correctness evaluation along with some interesting results of applying CPA. Below are the research questions answered in this section.

- **RQ1.** How much can our approach (CPA) speed up the source code analyses that involves analyzing thousands and millions of control flow graphs?
- **RQ2.** How much reuse opportunity exists when performing collective program analysis?
- **RQ3.** What is the impact of the abstraction (in the form of sparse representation) on the correctness and precision of the analysis results?

5.3.1 Performance

5.3.1.1 Methodology

We compare our approach against a baseline that runs the analysis on all programs in the dataset without any optimization or reuse. We also compare against a prior work [83, 84] that identifies and removes irrelevant statements prior to analyzing programs. We measure the analysis time for all three approaches (CFG, RCFG, and CPA) and compute the percentage reduction in the analysis time of CPA over CFG (denoted as R) and RCFG (denoted as R') respectively. The analysis times were averaged over the last three runs, when the variability across these measurements is minimal (under 2%) by following the methodology proposed by Georges *et al.* [30]. Note that, the cache (or pattern database) is cleared after every run to ensure same setting for each run.

Our experiments were run on a machine with 24 GB of memory and 24-cores, running on Linux 3.5.6-1.fc17 kernel.

Table 5.1: Analyses used in our evaluation.

#	Analysis	Description
1	`Avail`	Expression optimization opportunities
2	`Dom`	Control flow dominators
3	`Escape`	Escape analysis
4	`Leak`	Resource leaks
5	`Live`	Liveness of statements
6	`MayAlias`	Local alias relations
7	`Null`	Null check after dereference
8	`Pointer`	Points-to relations
9	`Safe`	Unsafe synchronizations
10	`Taint`	Vulnerability detections

5.3.1.2 Analyses

We have used 10 source code analyses to evaluate our approach as listed in Table 5.1. We used several criteria to select the candidate analyses. We have included analyses to obtain maximum coverage over the flow analysis properties, such as analysis direction (*forward*, *backward*), merge operation (*union*, *intersection*), complexity of the analysis, and complexity of the data-structures used to store the analysis results at nodes. The analyses are written using Boa [22, 25, 23], a domain specific language (DSL) for ultra-large-scale mining and we have used Boa compiler and runtime for executing the analyses. Next, we briefly describe each analysis used in our evaluation.

`Avail.` [49] Available expression analysis tries to find optimization opportunities in the source code, such as value of a binop expression computed once can be reused in the later program points, if the variables in the expression are not re-defined. This is a standard compiler optimization drawn from the textbook. We included this analysis to represent how an optimization problem can benefit from `CPA`. The analysis will report if there exists one or more optimization opportunities.

`Dom.` [4] Control flow dominators are useful in many analyses that requires control dependence, for instance in computing the program dependence graph (PDG), however computing the dominators is expensive, hence we included this in our list of analyses to demonstrate how our technique can accelerate computing dominators. This is also a special kind of analysis, where all nodes are relevant for the analysis and the sparse representation constitutes the whole CFG. The analysis will report a map containing a list of dominators for each CFG node.

`Escape.` [89] Escape analysis computes whether the objects allocated inside methods stay within the method (captured) or escapes to another methods. This information is useful to decide whether to allocate memory for such objects in the stack instead of heap. The analysis outputs *true*, if there exists any captured objects, otherwise *false*.

`Leak.` [80] This is a resource leak checker that captures the resource usage in programs to identify possible leaks. The analysis tracks all 106 JDK resource related API usages in programs. If any resource acquired is not released at the exit node, it outputs that leak may exist.

`Live.` [49] This analysis tracks the liveness of local variables used in the program. There exists many client applications of this analysis such as identifying and removing the dead code, register allocation, etc. This analysis simply reports all the variable definition and use sites along with their control flow dependencies.

`MayAlias.` [66] Precisely computing the alias information is expensive and sometimes may not be possible. In such situations, computing the may alias information by following the direct assignments can be handy. This may alias analysis computes the alias information and reports the alias sets.

`Null.` [13] This analysis checks if there exists a dereference that is post-dominated by a null check. Such a pattern indicates that the dereference may cause null pointer exception, because it can be `null`. The analysis reports if there exists such problems in the program.

`Pointer.` [68] Pointer or points-to analysis implemented here is a flow-sensitive and context-insensitive points-to analysis. It computes the points-to graph. A points-to graph provides the information whether the variables in the program may point to the same memory location. This analysis outputs the points-to graph with abstract variables and nodes (meaning concrete variable names are mapped to symbols).

`Safe.` [80] The safe synchronization checker looks for the lock acquire/release patterns to identify bugs. Acquiring locks and not releasing them may cause deadlock and starvation in the program. The analysis tracks all the variables on which the lock is acquired and checks if the locks on these variables are released on every program path. If not, it reports that the problem exists.

`Taint.` [28] Taint analysis detects and reports possible vulnerabilities by performing a taint analysis. The analysis identifies the variables that read data from external inputs like console, tracks their dataflow, and checks if the data from these variables are written to output.

5.3.1.3 Datasets

We have used two datasets for evaluating `CPA`. The first dataset consists of all projects included in the DaCapo benchmark [11], a well-established benchmark of Java programs. This dataset contains 45,054 classes and 286,888 non-empty methods. The DaCapo dataset is prepared using the GitHub project links of the 10 DaCapo projects. The second dataset consists of 4,938 open source SourceForge projects. This dataset consists of 191,945 classes, and 6,741,465 non-empty method. Note that, the order of methods in our datasets is random and it is determined by the dataset creators [22]. As such, the order does not influence `CPA`, while prior caching does.

Table 5.2: Reduction in the analysis time of `CPA`.

	Time (in ms)												
	`DaCapo`							`SourceForge`					
Analysis	`CFG`	`RCFG`	`CPA`	`CPA-UB`	`CPA-CR`	R	R’	`CFG`	`RCFG`	`CPA`	`CPA-UB`	R	R’
`Avail`	3274	1822	1368	872	1039	58%	25%	63971	35087	24688	16593	61%	30%
`Dom`	247855	247855	75559	1898	3778	70%	70%	6439232	6439232	3614844	32664	44%	44%
`Escape`	12624	8707	3588	902	2153	72%	59%	250697	160654	71086	21454	72%	56%
`Leak`	227	35	39	32	37	83%	-9%	5947	830	458	348	92%	45%
`Live`	5470	4329	2628	820	1866	52%	39%	138929	111027	65369	17953	53%	41%
`MayAlias`	7823	4137	2238	870	1544	71%	46%	168542	85204	43657	16292	74%	49%
`Null`	3841	2254	1365	257	683	64%	39%	104838	65551	36885	5108	65%	44%
`Pointer`	6246	3367	2019	888	1716	68%	40%	109031	62279	40446	17223	63%	35%
`Safe`	9	2	2	2	2	75%	0%	70	17	16	14	77%	4%
`Taint`	499	172	123	55	62	75%	28%	15981	3886	2266	800	86%	42%
					Average	69%	34%					69%	39%

5.3.1.4 Results and Analysis

Table 5.2 compares our approach (CPA) against the baseline (CFG) and the prior work (RCFG). The analysis time in case of CFG is the actual analysis time, whereas, in case of RCFG, it includes the two overheads (to identify and remove the irrelevant nodes) and in case of CPA, it includes several overheads (to produce the sparse graph, to generate pattern, to check the pattern database, and retrieve the result in case of hit, and to persist the results in case of miss). The analysis times are reported in milliseconds. For some analysis, the analysis times are low, for instance Safe. This is mainly because we have optimized all our analyses to skip through the irrelevant methods (methods that do not contain the information the analysis is looking for). Finding and skipping through the irrelevant methods is done at a low cost.

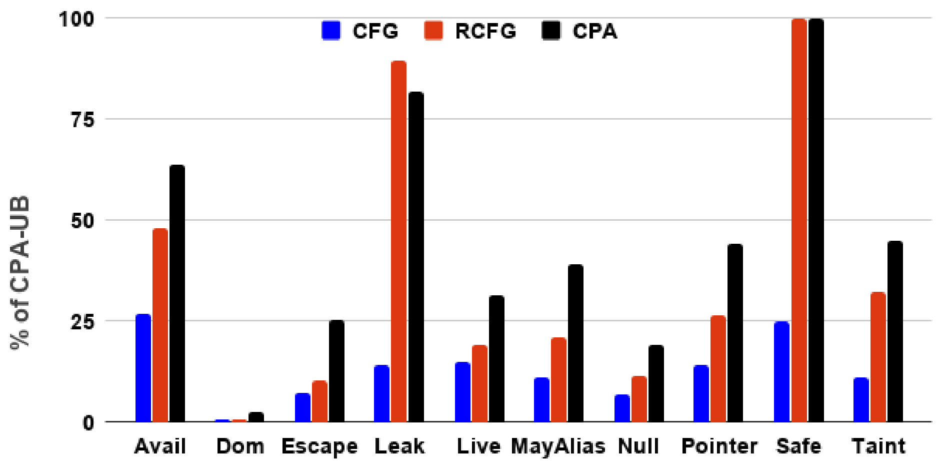


Figure 5.5: % benefit of the upper bound achieved by CFG, RCFG, CPA. Higher bars are better.

Table 5.2 shows our results. The two columns R and R' shows the percentage reduction in the analysis time of CPA over CFG and RCFG respectively. On average CPA was able to reduce the analysis time by 69% over CFG and 36% over RCFG (averaged over both DaCapo and SourceForge datasets, which are individually 34% and 39%). Note that, for Dom, the CFG and RCFG times are exactly same, because for this analysis all nodes are relevant, hence RCFG is simply CFG. For Leak analysis on DaCapo dataset, CPA shows negative gain when compared to RCFG. This happens mainly because of the low number of instances on which the analysis is run. As shown in Table 5.4, under

DaCapo, the Leak analysis is run on only 220 unique methods and the cost of CPA overheads may exceed the benefit, hence we do not expect CPA to improve the performance. Similar situation can be also be seen for Safe analysis.

CPA uses an online strategy of caching the analysis results at the same time as running the analysis on millions of programs. CPA can also be used with prior caching, hence we compute the ideal gain (or an upper-bound) by re-running the experiments on the same dataset after caching the results in the first run. The analysis times are reported in Table 5.2 under CPA-UB column. Figure 5.5 helps to understand how far CFG, RCFG, and CPA are from the ideal speedup (CPA-UB). As it can be seen in Figure 5.5, for most analysis, CPA is the closest to 100%, when compared to others (CFG and RCFG), except for Leak and Safe. The reason is as explained earlier, the number of methods on which the analysis is run is small, hence the overheads of CPA exceeds its benefits. Another interesting observation that can be made is that except for Leak and Safe, for all other analysis, there exists substantial opportunities to improve the performance of CPA to get it closer to CPA-UB. This can be performed by training CPA on some representative projects, caching the results, and using them on the test projects.

5.3.1.5 Cross Project

We also performed a cross-validation experiment, where we excluded one project at a time from the DaCapo dataset that contains a total of 10 projects, ran the analysis, cached the results, and measured the analysis time for the excluded project. We repeated this experiment and aggregated the analysis times for all 10 projects. We reported the analysis times under CPA-CR column in Table 5.2. As seen in the CPA-CR column, CPA-CR analysis time lies between CPA and CPA-UB. For some analyses CPA-CR is able to nearly meet the upper-bound. For example, Dom. For Leak and Taint, prior caching had less effect on the analysis time, mainly because the number of instances on which the analysis is run was small. For other analyses, a consistent speedup is seen over CPA. This suggests that, CPA with some prior caching can improve the performance over the online-strategy.

Table 5.3: CPA time distribution across four components. The absolute times are in milliseconds and the values inside "()" are the contribution of the component towards CPA time.

	DaCapo				SourceForge			
Analysis	CPA	pattern	analysis	persist	CPA	pattern	analysis	persist
Avail	1368	863 (63%)	496 (36%)	8 (1%)	24688	16454 (67%)	8095 (33%)	138 (%1)
Dom	75559	1875 (02%)	73661 (97%)	23 (0%)	3614844	32437 (01%)	3582180 (99%)	225 (0%)
Escape	3588	892 (25%)	2686 (75%)	10 (0%)	71086	21206 (30%)	49632 (70%)	247 (0%)
Leak	39	30 (77%)	7 (18%)	1 (3%)	458	342 (75%)	110 (24%)	5 (1%)
Live	2628	811 (31%)	1807 (69%)	8 (0%)	65369	17776 (27%)	47416 (73%)	176 (0%)
MayAlias	2238	862 (39%)	1368 (61%)	7 (0%)	43657	16167 (37%)	27364 (63%)	125 (0%)
Null	1365	252 (18%)	1108 (81%)	4 (0%)	36885	5027 (14%)	31777 (86%)	80 (0%)
Pointer	2019	879 (44%)	1130 (56%)	8 (0%)	40446	17056 (42%)	23223 (57%)	166 (0%)
Safe	2	2 (71%)	0 (00%)	0 (0%)	16	13 (81%)	2 (13%)	0 (0%)
Taint	123	52 (43%)	68 (55%)	2 (1%)	2266	784 (35%)	1466 (65%)	15 (1%)

5.3.1.6 CPA Components

For every CFG of the method to be analyzed, CPA produces a sparse representation of the CFG, generates a pattern that represents the sparse graph, and checks the pattern database for a result. The overhead for this stage is represented as pattern in Table 5.3. When there is a miss, i.e., the result does not exists for a pattern, then CPA runs the analysis to produce a result (analysis stage) and cache the result (persist stage). When there is a hit, i.e., a result is found, nothing else needs to be done. It is interesting to see how the overall CPA time is distributed across these components. The component results are shown in Table 5.3, where pattern, analysis, and persist are the three components. The absolute times are in milliseconds and we also show the contributions of each of the three components towards CPA time (numbers inside parentheses).

It can be seen that, persist time is almost always negligible. The pattern time, which is the time to construct the sparse graph, generate pattern, and check the database sometimes exceeds the actual analysis time. For example, Avail, Leak, and Safe. This is mainly because in these analyses, the amount of relevant nodes are very small. Thus, the time for removing the irrelevant nodes to construct the sparse graph becomes substantial.

Table 5.4: Amount of reuse opportunity available in various analysis.

	DaCapo			SourceForge		
Analysis	Total	Unique	Reuse	Total	Unique	Reuse
`Avail`	286888	15402	95%	6741465	266081	96%
`Dom`	286888	20737	93%	6741465	345715	95%
`Escape`	286888	23347	92%	6741465	430978	94%
`Leak`	3087	220	93%	71231	2741	96%
`Live`	286888	19417	93%	6741465	366315	95%
`MayAlias`	286888	12652	96%	6741465	211010	97%
`Null`	49036	7857	84%	746539	148671	80%
`Pointer`	286888	16150	94%	6741465	313337	95%
`Safe`	77	14	82%	1310	89	93%
`Taint`	6169	1208	80%	147446	22664	85%

5.3.1.7 Reuse Opportunity

Table 5.2 shows that our approach was able to substantially reduce the analysis time across 10 analyses and two datasets. The reduction mainly stems from the amount of reuse opportunity that exists in large datasets of programs. We measured the total number of unique graphs to compute the reuse percentage. The results are shown in Table 5.4. For all the analyses, `CPA` was able to reuse the analysis results over 80% of the time. A very high percentage of reuse clearly suggests why our approach was able to achieve substantial reduction in the analysis time. Further, it also supports the fact that source code is repetitive.

Table 5.5 lists the transfer functions for all our 10 analyses. The names of these transfer functions provides information about the kind of statements that are relevant for the analyses. For instance, `def(v)` transfer function applies to all statements that have variable definitions. As we use transfer function names to label the nodes and produce the pattern, these names are used in the top patterns discussed next.

In case of `Taint` analysis, we had a total of 6169 methods in the `DaCapo` dataset that were analyzed (other methods didn't had relevant code) and they formed 1208 unique sparse graphs. The analysis reported possibility of vulnerabilities for 101 sparse graphs. Figure 5.6 shows the top 3 patterns along with their frequencies ((a), (b), and (c)). Our analysis did not report any

Table 5.5: Transfer functions to identify relevant nodes.

Analysis	Relevant Nodes
`Avail`	`def(v)`, `binop(`v_1`, `v_2`)`
`Dom`	all
`Escape`	`copy(`v_1`, `v_2`)`, `load(`v_1`, `v_2`.f)`, `store(`$v_1.f$`, `v_2`)`, `gload(`v_1`, `$cl.f$`)`
	`gstore(`$cl.f$`, `v_2`)`, `call(v, m, `v_0`,...,`v_k`)`, `new(v, cl)`, `return(v)`
`Leak`	`open(v)`, `close(v)`, `copy(`v_1`, `v_2`)`
`Live`	`def(v)`, `use(v)`
`MayAlias`	`def(`v_1`, c)`, `def(`v_1`, `v_2`)`
`Null`	`deref(v)`, `copy(`v_1`, `v_2`)`, `nullcheck(v)`
`Pointer`	`copy(`v_1`, `v_2`)`, `new(v, cl)`, `load(`v_1`, `v_2`.f)`
	`store(`$v_1.f$`, `v_2`)`, `return(v)`, `call(v, m, `v_0`,...,`v_k`)`
`Safe`	`lock(v)`, `unlock(v)`
`Taint`	`input(v)`, `output(v)`, `copy(`v_1`, `v_2`)`

vulnerabilities for any methods that have the sparse graphs shown in (a), (b), (c), because all these three sparse graphs only have either `input` or `output` nodes. For vulnerability to occur, both must exists. Consider (d) which has both `input` and `output` was one of the frequent vulnerability pattern in the `DaCapo` dataset. We manually verified 20 out of 101 reported instances for the existence of possible vulnerabilities.

In case of `Safe` analysis that checks for the correct use of lock/unlock primitives using JDK concurrent libraries, we had 76 instances reported correct and 1 reported as having a problem. Out of the 76, 50 of them followed a single pattern that is shown in Figure 5.7. This is a correct usage pattern for lock/unlock. The one instance that was reported problematic was a false positive and it requires inter-procedural analysis to eliminate it.

`Leak` analysis results were most surprising for us. There existed 3087 usages of JDK resource related APIs (JDK has 106 resource related APIs) and our analysis reported 2277 possible leaks. Out of these 336 were definitely leaks and others were possible leaks and confirming them would require inter-procedural analysis. Out of the 336 definite leaks, the top pattern appeared in 32 methods. Figure 5.8 shows this pattern.

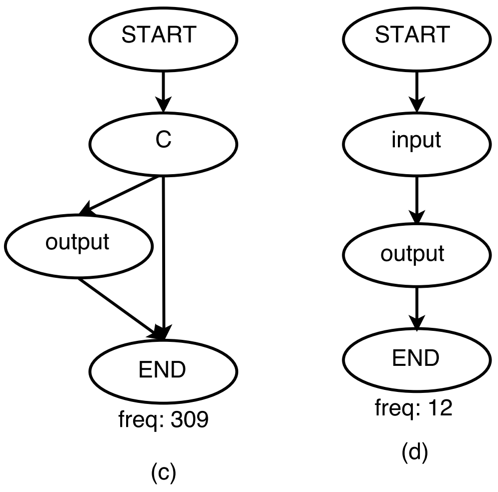


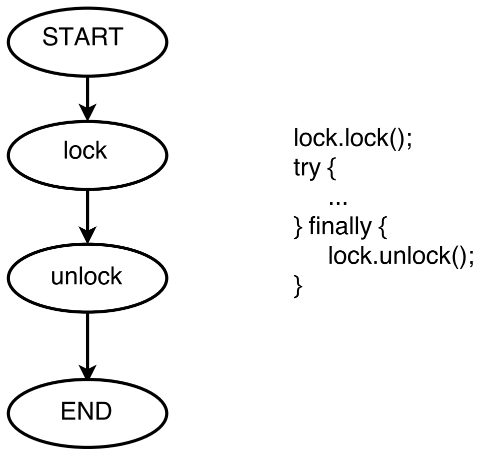
Figure 5.6: Top patterns seen in case of taint analysis that detects vulnerabilities

5.3.2 Correctness and Precision

In §5.2.3 we provided a proof sketch as to why the analysis results of `CPA` must match with that of `CFG`. To empirically evaluate the correctness of the results, we conducted two experiments using all 10 analysis and `DaCapo` dataset. In the first experiment, we compared the analysis result of `CFG` and `CPA` for every method that is analyzed. Table 5.6 provides information about the results computed for each analysis. We were able to match the two results perfectly.

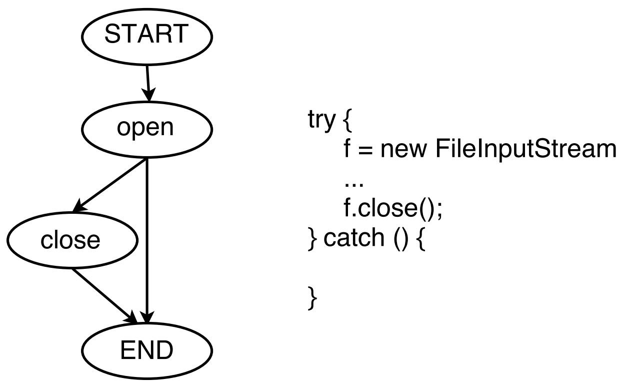
Table 5.6: Analysis results computed for various analysis.

Analysis	Computed Result
`Avail`	*true* or *false*
`Dom`	list of dominators for each node
`Escape`	points-to escape graph with abstract variables
`Leak`	*true* or *false*
`Live`	definitions and uses of abstract variables
`MayAlias`	alias sets of abstract variables
`Null`	*true* or *false*
`Pointer`	points-to graph with abstract variables
`Safe`	*true* or *false*
`Taint`	*true* or *false*



```
lock.lock();
try {
    ...
} finally {
    lock.unlock();
}
```

Figure 5.7: The most frequent lock/unlock pattern and the code example of the pattern.



```
try {
    f = new FileInputStream
    ...
    f.close();
} catch () {

}
```

Figure 5.8: Most frequent buggy resource leak pattern.

As for most of the analysis in Table 5.6 the computed results are just `boolean` values, we double-check the results by profiling the transfer functions executed for `CFG` and the sparse graph of `CPA`, and compare the sequence of transfer functions. We skip through the identity transfer functions in case of `CFG`, as the `CFG` may contain many irrelevant nodes. As the order of nodes visited in both `CFG` and sparse graph of `CPA` are same, we were able to see a 100% match.

5.3.3 Limitations

In this work we have applied `CPA` to accelerate analyses at method-level, where results for each method is computed independently without using the results at the method call sites. Instead of applying the results of methods at their call sites we have adopted an over-approximation strategy. As such, there are no theoretical limitations preventing the use of our technique in a compositional whole-program analysis setting, where the results of the called methods can be used at their call sites, if available. This design choice was mainly due to the analysis framework used in our evaluation, which does not support whole-program analysis as of this writing.

Another limitation that currently exists in `CPA` is that it can only store abstract analysis results. For instance, boolean value to indicate the existence of certain kinds of bug. `CPA` also allows using abstract variables and location names in the analysis as results. For instance, variable v_0 points to the first variable encountered while analyzing the method statements. Similarly, the location loc_0 points to the first relevant statement that exists in the sparse representation of the method. The abstract variables and location names helped us to model many important analyses, such as `Live`, `Escape`, `Pointer`, etc. In future, we plan to support better output types.

5.4 Summary

This work proposed *collective program analysis* (CPA), a technique for accelerating large scale source code analysis by leveraging analysis specific similarity. The key idea of `CPA` is clustering programs that are similar for the purpose of the analysis, such that it is sufficient to run the analysis on one program from each cluster to produce result for others. To find analysis specific similarity between programs, a sparse representation and a canonical labeling scheme was used. The technique is applied to source code analysis problems that requires data-flow analysis. When compared to the state-of-the-art, where the analysis is directly performed on the CFGs, `CPA` was able to reduce the analysis time by 69%. When compared to an optimization technique that removes the irrelevant parts of the program before running the analysis, `CPA` was able to reduce the analysis time by 36%. Both of these results were consistent across two datasets that contained several hundred thousand methods to over 7 million methods. The sparse representation used in the `CPA` was able to create a high percentage of reuse opportunity (more than 80%). In future, we plan to extend `CPA` to whole-program analysis and extend `CPA` to support more output types.

5.5 Related Work

Our work on CPA is related to the prior work on both improving the efficiency of software analysis and finding software clones.

5.5.1 Improving the efficiency of source code analysis

There exists a trade off between improving the efficiency of the analysis and improving the accuracy of the analysis results. Removing the unimportant parts of the code before analyzing it has been a popular choice [83, 16, 78, 5, 64]. For instance, Upadhyaya and Rajan [83] proposed RCFG a reduced control flow graph that contains only statements that are related to analysis. Our work on CPA has adopted the RCFG work to produce the sparse graph. CPA uses RCFG as its sparse representation to cluster similar graphs and reuse the analysis results to further accelerate analyses. As we have shown in our evaluation, CPA was able to achieve on average a 36% speedup over RCFG. There also exists other sparse representations such as sparse evaluation graph (SEG) [16] that are more suitable for def-use style data-flow analysis. There exists works that eliminates unnecessary computations in the traversal of the program statements to improve the efficiency of analysis [78, 8]. These techniques remove the unnecessary iterations to improve the efficiency, whereas our work removes the unnecessary statements to produce sparse graphs and also reuses the results by clustering sparse graphs.

Allen *et al.* [5] and Smaragdakis *et al.* [64] have proposed a pre-analysis stage prior to actual analysis to scale points-to analysis to large code bases. They perform static analysis and program compaction to remove statements that do not contribute to the points-to results. Their work is specialized for scaling points-to analysis, whereas CPA is more general, in that it can accelerate analysis that use data-flow analysis and expressed using the lattice framework.

Program slicing is a fundamental technique to produce a compilable and runnable program that contains statements of interest specified by a slicing criteria [88]. Many slicing techniques have been proposed [75]. Our pruning technique is similar to slicing, in that we also remove the irrelevant statements, however our pruning technique is a pre-processing step rather than a transformation and it does not produce a compilable and runnable code like slicing. Slicing cannot be used for our purpose, because the program statements of interest are not known. Even if the statements of interest are known, slicing may includes statements (affecting the values of variables at program points of interest) that may not contribute to the analysis output. Our technique only includes statements that contributes to the analysis output.

Reusing the analysis results is another way to improve the efficiency of program analysis [38, 31, 57]. Kulkarni *et al.* [38] proposed a technique to accelerate program analysis in Datalog. The idea of their technique is to run an offline analysis on a corpus of training programs to learn the analysis facts and then reuses the learnt facts to accelerate the analysis of other programs that share some code with the training corpus. Inter-procedural analysis are often accelerated by reusing the analysis results in the form of partial [31] and complete [57] procedure summaries, where the analysis results of procedures can be reused at their call sites. Our technique does not require that programs share code, it only requires that programs executed same set of analysis instructions to produce results.

5.5.2 Finding software clones

Our technique is also related to code clones [59], as `CPA` also clusters sparse representations of programs to reuse the analysis results. There exists different types of clones. Syntactic clones are look alike code fragments, semantic clones share common expressions and they have similar control flows, and functional clones are similar in terms of the inputs and outputs. There are also other approaches that goes beyond structural similarity, like code fingerprints[43], behavioral clones [70, 26, 71], and run-time behavioral similarity [19].

We did not use syntactic clones (token-based or AST-based), because the benefits will be limited to copy-and-paste code. Semantic clones (code fragments with similar control and data flow) could not be used, because of lack of guarantee that analysis output will be similar. Moreover, semantically different code fragments may produce similar output for a given analysis and we would miss out on those. We cannot use functional clones (code fragments with similar input/output), because they may not produce similar analysis output. We also could not use behavioral clones (code fragments that perform similar computation captured using dynamic dependence graphs), because they cannot guarantee similar analysis output. An analysis may produce similar output for code fragments that are not behavioral clones. Further, in our setting, while analyzing thousands of projects, it is not feasible to instrument the code, run them, collect traces, and build dynamic dependence graphs to detect behavioral clones.

CHAPTER 6. CONCLUSION AND FUTURE WORK

Popularity of data-driven software engineering has led to an increasing demand on the infrastructures to support efficient execution of tasks that require deeper source code analysis. Extant techniques have focused on leveraging distributed computing to meet the demand, but with a concomitant increase in computational resource needs. This thesis presents two acceleration techniques that do not increase the computational resource needs.

The first acceleration technique performs a pre-analysis and program compaction to reduce the amount of source code on which the analysis will be performed. The technique can automatically extract the parts of the source code that are relevant for the analysis and remove the irrelevant parts before performing the analysis. Our intuition behind acceleration is that often there exists a substantial portion of the program that are irrelevant for the user analyses and the unnecessary computations performed on the irrelevant parts can be avoided to accelerate the overall analysis. We show that the technique is sound in terms of analysis output, where the output before and after the reduction are same. We also show that the pre-analysis and compaction are light-weight and they can be performed without much overhead. At the same time, the technique is able to achieve substantial speedup.

The second acceleration technique collective program analysis is based on two novel ideas: the mining task specific similarity and the interaction pattern graph. The key idea is to cluster artifacts that have similar interaction pattern graphs, such that running the task on unique artifacts is sufficient to produce the results for others. Our intuition behind acceleration is that, since a large portion of source code that exists today is repetitive, repeating the analysis on repetitive code can be avoided to accelerate the overall analysis of the large code bases. Also, a right representation of similarity is desired to identity and reuse the analysis results. We applied collective program analysis on analyzing millions of control flow graphs and we were able to achieve substantial speedup. Our results also reveal that large code bases often have large amount of reuse opportunities in terms of analysis output.

6.1 Future Work

There are several avenues for future work that can be investigated. First, applying the acceleration techniques proposed in this thesis to project level or whole program mode. Second, the core techniques used to accelerate large scale source code analysis can also be utilized in other source code related activities. For instance, in program comprehension, in synthesizing programs, etc.

6.1.1 Project Level or Whole Program Analysis

An immediate avenue for future work could be to extend our acceleration techniques to whole program analysis or project level mining. The acceleration technique proposed in this thesis can accelerate method-level mining tasks that require to be run on millions of methods. More precise analyses or mining tasks may require going beyond method boundaries. For instance, computing memory access patterns. One main challenge for this future work is enabling or building whole program analysis capabilities into large scale source code mining frameworks. To best to our knowledge, there does not exists any infrastructure which can support running whole program analyses on thousands of projects. Our acceleration techniques are implemented in Boa large scale source code mining infrastructure, hence when Boa evolves to support whole program analysis, our acceleration techniques can be extended easily. As part of this future work, it may also be worthwhile to implement our acceleration techniques as part of an industrial strength static analyzer, such as Facebook's infer [27] to make our techniques available outside Boa infrastructure.

6.1.2 Program Comprehension

Program comprehension or understanding is a central activity during software mantenance, evolution, and re-engineering. A study on program comprehension reports that up to 50% of the maintenance effort is spent on understanding the code [2]. Program comprehension research is about building models of application artifacts like code at various abstraction levels. There exists several comprehension strategies [69] like top-down (application to code), bottom-up (code to application), knowledge-based (using domain knowledge), and mixed (combining top-down, bottom-up and domain knowledge). Also, there exists several representation of source code like text, token-based, abstract syntax trees (ASTs), control and data flow graphs, callgraphs, program dependence graphs, etc, to aid program comprehension. There also exists several tools to make comprehension effective [29, 94, 67, 63, 35, 42].

While there has been several decades of research on program comprehension, recent studies have suggested new opportunities to improve program comprehension further along conducting more human studies for understanding and evaluating comprehension strategies, models, and tools [61]. We believe following three factors could play a role in the future of comprehension strategies, models, and tools: 1) similarity between programs, 2) goal-directed comprehension, and 3) characteristics of programmers. We believe some of the key ideas proposed in this thesis could bring in the factors listed above on to program comprehension activities. For instance, our work on finding similarity between program for a given mining task can help to identify similarity between programs to aid comprehension. For instance, a difficult program can be understood using already known program, if the two programs share some similarity with respect to the comprehension task at hand. However, there exists several open questions to be answered. For instance, how to express comprehension task easily such that similarity between programs for the given comprehension task can be identified, how to build a knowledge-base of already known programs for the comprehender, how to recommend aid programs that helps with comprehending a difficult program, and so on.

6.1.3 Clone Datasets

As part of our acceleration technique in collective program analysis, we cluster programs that have mining task specific similarities. An avenue for future work could be to generate several clone datasets of the clusters generated in our approach and measure how it enhances the clone detection applications. There exist several clone datasets for syntactically and semantically similar code, however creating behavioral clones is still an open research topic. Clone detection has many applications in plagiarism detection, program repair, program synthesis, etc[59, 60]. We believe the clones generated in our approach are more closer to behavioral clones and hence it may provide an opportunity to study them to utilize them in applications like program repair.

BIBLIOGRAPHY

[1] Acharya, M., Xie, T., Pei, J., and Xu, J. (2007). Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 25–34, New York, NY, USA. ACM.

[2] Al-Saiyd, N. A. (2017). Source code comprehension analysis in software maintenance. In *2017 2nd International Conference on Computer and Communication Systems (ICCCS)*, pages 1–5.

[3] Ali, K. and Lhoták, O. (2012). Application-Only Call Graph Construction. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 688–712, Berlin, Heidelberg. Springer-Verlag.

[4] Allen, F. E. (1970). Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA. ACM.

[5] Allen, N., Scholz, B., and Krishnan, P. (2015). *Staged Points-to Analysis for Large Code Bases*, pages 131–150. Springer Berlin Heidelberg, Berlin, Heidelberg.

[6] Aquino, A. (2015). Scalable program analysis through proof caching (doctoral symposium). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 432–435, New York, NY, USA. ACM.

[7] Aquino, A., Bianchi, F. A., Chen, M., Denaro, G., and Pezzè, M. (2015). Reusing constraint proofs in program analysis. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 305–315, New York, NY, USA. ACM.

[8] Atkinson, D. C. and Griswold, W. G. (2001). Implementation techniques for efficient data-flow analysis of large programs. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 52–, Washington, DC, USA. IEEE Computer Society.

[9] Bagherzadeh, M. and Rajan, H. (2015). Panini: A concurrent programming model for solving pervasive and oblivious interference. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 93–108, New York, NY, USA. ACM.

[10] Bajracharya, S., Ossher, J., and Lopes, C. (2014). Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Sci. Comput. Program.*, 79:241–259.

[11] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA. ACM.

[12] Bourdoncle, F. (1993). Efficient chaotic iteration strategies with widenings. In *IN Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag.

[13] Brown, F., Nötzli, A., and Engler, D. (2016). How to Build Static Checking Systems Using Orders of Magnitude Less Code. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 143–157, New York, NY, USA. ACM.

[14] Bruch, M., Monperrus, M., and Mezini, M. (2009). Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA. ACM.

[15] Chen, M. (2014). Reusing constraint proofs for scalable program analysis. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 449–452, New York, NY, USA. ACM.

[16] Choi, J.-D., Cytron, R., and Ferrante, J. (1991). Automatic Construction of Sparse Data Flow Evaluation Graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 55–66, New York, NY, USA. ACM.

[17] Cobleigh, J. M., Clarke, L. A., and Osterweil, L. J. (2001). The Right Algorithm at the Right Time: Comparing Data Flow Analysis Algorithms for Finite State Verification. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 37–46, Washington, DC, USA. IEEE Computer Society.

[18] D'Ambros, M., Lanza, M., and Robbes, R. (2012). Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577.

[19] Demme, J. and Sethumadhavan, S. (2012). Approximate graph clustering for program characterization. *ACM Trans. Archit. Code Optim.*, 8(4):21:1–21:21.

[20] Ding, S., Zhang, H., and Tan, H. B. K. (2014). Detecting infeasible branches based on code patterns. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 74–83.

[21] Diwan, A., McKinley, K. S., and Moss, J. E. B. (1998). Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 106–117, New York, NY, USA. ACM.

[22] Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013a). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 422–431, Piscataway, NJ, USA. IEEE Press.

[23] Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2015). Boa: Ultra-large-scale software repository and source-code mining. *ACM Trans. Softw. Eng. Methodol.*, 25(1):7:1–7:34.

[24] Dyer, R., Rajan, H., Nguyen, H. A., and Nguyen, T. N. (2014). Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 779–790, New York, NY, USA. ACM.

[25] Dyer, R., Rajan, H., and Nguyen, T. N. (2013b). Declarative visitors to ease fine-grained source code mining with full history on billions of AST nodes. In *GPCE: the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE, pages 23–32.

[26] Elva, R. and Leavens, G. T. (2012). Semantic clone detection using method ioe-behavior. In *Proceedings of the 6th International Workshop on Software Clones*, IWSC '12, pages 80–81, Piscataway, NJ, USA. IEEE Press.

[27] Facebook, I. Infer static analyzer. http://fbinfer.com/.

[28] Fehnker, A., Huuck, R., and Rödiger, W. (2011). Model Checking Dataflow for Malicious Input. In *Proceedings of the Workshop on Embedded Systems Security*, WESS '11, pages 4:1–4:10, New York, NY, USA. ACM.

[29] Fowkes, J., Chanthirasegaran, P., Ranca, R., Allamanis, M., Lapata, M., and Sutton, C. (2016). Tassal: Autofolding for source code summarization. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 649–652, New York, NY, USA. ACM.

[30] Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA. ACM.

[31] Godefroid, P., Nori, A. V., Rajamani, S. K., and Tetali, S. D. (2010). Compositional may-must program analysis: Unleashing the power of alternation. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 43–56, New York, NY, USA. ACM.

[32] Gousios, G. (2013). The GHTorent Dataset and Tool Suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA. IEEE Press.

[33] Hind, M. and Pioli, A. (1998). Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In *Proceedings of the 5th International Symposium on Static Analysis*, SAS '98, pages 57–81, London, UK, UK. Springer-Verlag.

[34] Horwitz, S., Reps, T., and Sagiv, M. (1995). Demand Interprocedural Dataflow Analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '95, pages 104–115, New York, NY, USA. ACM.

[35] Khoo, Y. P., Foster, J. S., Hicks, M., and Sazawal, V. (2008). Path projection for user-centered static analysis tools. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '08, pages 57–63, New York, NY, USA. ACM.

[36] Kildall, G. A. (1973). A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA. ACM.

[37] Kothari, S., Tamrawi, A., Sauceda, J., and Mathews, J. (2016). Let's verify linux: Accelerated learning of analytical reasoning through automation and collaboration. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 394–403, New York, NY, USA. ACM.

[38] Kulkarni, S., Mangal, R., Zhang, X., and Naik, M. (2016). Accelerating program analyses by cross-program training. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 359–377, New York, NY, USA. ACM.

[39] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. (2006). Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192.

[40] Livshits, B. and Zimmermann, T. (2005). Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 296–305, New York, NY, USA. ACM.

[41] Lokuciejewski, P., Cordes, D., Falk, H., and Marwedel, P. (2009). A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 136–146, Washington, DC, USA. IEEE Computer Society.

[42] Lucia, A., Penta, M., Oliveto, R., Panichella, A., and Panichella, S. (2014). Labeling source code with information retrieval methods: An empirical study. *Empirical Softw. Engg.*, 19(5):1383–1420.

[43] McMillan, C., Grechanik, M., and Poshyvanyk, D. (2012a). Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 364–374, Piscataway, NJ, USA. IEEE Press.

[44] McMillan, C., Grechanik, M., Poshyvanyk, D., Fu, C., and Xie, Q. (2012b). Exemplar: A source code search engine for finding highly relevant applications. *IEEE Trans. Softw. Eng.*, 38(5):1069–1087.

[45] Mishne, A., Shoham, S., and Yahav, E. (2012). Typestate-based semantic code search over partial programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 997–1016, New York, NY, USA. ACM.

[46] Mudduluru, R. and Ramanathan, M. K. (2014). Efficient incremental static analysis using path abstraction. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411*, pages 125–139, New York, NY, USA. Springer-Verlag New York, Inc.

[47] Nagappan, M., Zimmermann, T., and Bird, C. (2013). Diversity in software engineering research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 466–476, New York, NY, USA. ACM.

[48] Nguyen, H. A., Dyer, R., Nguyen, T. N., and Rajan, H. (2014). Mining preconditions of apis in large-scale code corpus. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 166–177, New York, NY, USA. ACM.

[49] Nielson, F., Nielson, H. R., and Hankin, C. (2010). *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

[50] Olson, M. A., Bostic, K., and Seltzer, M. I. (1999). Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191.

[51] Peleg, H., Shoham, S., Yahav, E., and Yang, H. (2016). Symbolic automata for representing big code. *Acta Inf.*, 53(4):327–356.

[52] Rajan, H. (2015). Capsule-oriented programming. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 611–614, Piscataway, NJ, USA. IEEE Press.

[53] Rajan, H., Kautz, S. M., Lin, E., Mooney, S. L., Long, Y., and Upadhyaya, G. (2014). Capsule-oriented Programming in the Panini Language. Technical Report 14-08.

[54] Rajan, H., Nguyen, T. N., Leavens, G. T., and Dyer, R. (2015). Inferring behavioral specifications from large-scale repositories by leveraging collective intelligence. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 579–582, Piscataway, NJ, USA. IEEE Press.

[55] Raychev, V., Vechev, M., and Krause, A. (2015). Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA. ACM.

[56] Reiss, S. P. (2009). Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 243–253, Washington, DC, USA. IEEE Computer Society.

[57] Reps, T., Horwitz, S., and Sagiv, M. (1995). Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA. ACM.

[58] Rountev, A., Sharp, M., and Xu, G. (2008). Ide dataflow analysis in the presence of large object-oriented libraries. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 53–68, Berlin, Heidelberg. Springer-Verlag.

[59] Roy, C. K., Cordy, J. R., and Koschke, R. (2009). Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *SCIENCE OF COMPUTER PROGRAMMING*, pages 470–495.

[60] Roy, C. K., Zibran, M. F., and Koschke, R. (2014). The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 18–33.

[61] Schröter, I., Krüger, J., Siegmund, J., and Leich, T. (2017). Comprehending studies on program comprehension. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 308–311, Piscataway, NJ, USA. IEEE Press.

[62] Sharir, M. and Pnueli, A. (1978). *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences. ComputerScience Department.

[63] Silva, J. (2012). A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41.

[64] Smaragdakis, Y., Balatsouras, G., and Kastrinis, G. (2013). Set-based pre-processing for points-to analysis. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 253–270, New York, NY, USA. ACM.

[65] Snelting, G., Robschink, T., and Krinke, J. (2006). Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457.

[66] Soot (2015). *Local May Alias Analysis*. https://github.com/Sable/soot/.

[67] Sparck Jones, K. (2007). Automatic summarising: The state of the art. *Inf. Process. Manage.*, 43(6):1449–1481.

[68] Sridharan, M., Chandra, S., Dolby, J., Fink, S. J., and Yahav, E. (2013). Aliasing in object-oriented programming. chapter Alias Analysis for Object-oriented Programs, pages 196–232. Springer-Verlag, Berlin, Heidelberg.

[69] Storey, M.-A. (2005). Theories, methods and tools in program comprehension: Past, present and future. In *Proceedings of the 13th International Workshop on Program Comprehension*, IWPC '05, pages 181–191, Washington, DC, USA. IEEE Computer Society.

[70] Su, F.-H., Bell, J., Harvey, K., Sethumadhavan, S., Kaiser, G., and Jebara, T. (2016a). Code relatives: Detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 702–714, New York, NY, USA. ACM.

[71] Su, F. H., Bell, J., and Kaiser, G. (2016b). Challenges in behavioral code clone detection. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 3, pages 21–22.

[72] Tamrawi, A. and Kothari, S. (2016). Projected control graph for accurate and efficient analysis of safety and security vulnerabilities. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 113–120.

[73] Thummalapenta, S. and Xie, T. (2007). Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213, New York, NY, USA. ACM.

[74] Thummalapenta, S. and Xie, T. (2009). Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 283–294, Washington, DC, USA. IEEE Computer Society.

[75] Tip, F. (1995). A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189.

[76] Tiwari, N. M., Upadhyaya, G., Nguyen, H. A., and Rajan, H. (2017). Candoia: A platform for building and sharing mining software repositories tools as apps. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 53–63, Piscataway, NJ, USA. IEEE Press.

[77] Tiwari, N. M., Upadhyaya, G., and Rajan, H. (2016). Candoia: A platform and ecosystem for mining software repositories tools. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 759–764, New York, NY, USA. ACM.

[78] Tok, T. B. (2007). *Removing Unimportant Computations in Interprocedural Program Analysis*. PhD thesis, Austin, TX, USA. AAI3290942.

[79] Tok, T. B., Guyer, S. Z., and Lin, C. (2006). Efficient Flow-sensitive Interprocedural Data-flow Analysis in the Presence of Pointers. In *Proceedings of the 15th International Conference on Compiler Construction*, CC'06, pages 17–31, Berlin, Heidelberg. Springer-Verlag.

[80] Torlak, E. and Chandra, S. (2010). Effective Interprocedural Resource Leak Detection. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 535–544, New York, NY, USA. ACM.

[81] Upadhyaya, G. and Rajan, H. (2014). An automatic actors to threads mapping technique for jvm-based actor frameworks. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents &#38; Decentralized Control*, AGERE! '14, pages 29–41, New York, NY, USA. ACM.

[82] Upadhyaya, G. and Rajan, H. (2015). Effectively mapping linguistic abstractions for message-passing concurrency to threads on the java virtual machine. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 840–859, New York, NY, USA. ACM.

[83] Upadhyaya, G. and Rajan, H. (2017a). On Accelerating Source Code Analysis At Massive Scale. Technical Report TR17-02, Iowa State University.

[84] Upadhyaya, G. and Rajan, H. (2017b). On accelerating ultra-large-scale mining. In *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*, ICSE-NIER '17, pages 39–42, Piscataway, NJ, USA. IEEE Press.

[85] Upadhyaya, G. and Rajan, H. (2018). Collective program analysis. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, New York, NY, USA. ACM.

[86] Wasylkowski, A., Zeller, A., and Lindig, C. (2007). Detecting object usage anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44, New York, NY, USA. ACM.

[87] Wegman, M. N. and Zadeck, F. K. (1991). Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210.

[88] Weiser, M. (1981). Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA. IEEE Press.

[89] Whaley, J. and Rinard, M. (1999). Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 187–206, New York, NY, USA. ACM.

[90] Wu, R., Xiao, X., Cheung, S.-C., Zhang, H., and Zhang, C. (2016). Casper: An efficient approach to call trace collection. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 678–690, New York, NY, USA. ACM.

[91] Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 590–604, Washington, DC, USA. IEEE Computer Society.

[92] Yan, D., Xu, G., and Rountev, A. (2012). Rethinking soot for summary-based whole-program analysis. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 9–14, New York, NY, USA. ACM.

[93] Yan, X. and Han, J. (2002). gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, pages 721–, Washington, DC, USA. IEEE Computer Society.

[94] Yuan, B., Murali, V., and Jermaine, C. (2017). Abridging source code. *Proc. ACM Program. Lang.*, 1(OOPSLA):58:1–58:26.

[95] Zhang, T., Upadhyaya, G., Reinhardt, A., Rajan, H., and Kim, M. (2018). Are code examples on an online q&a forum reliable? a study of api misuse on stack overflow. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, New York, NY, USA. ACM.