On Accelerating Source Code Analysis At Massive Scale

Article *in* IEEE Transactions on Software Engineering · April 2018

DOI: 10.1109/TSE.2018.2828848

2 authors, including:

Ganesha Upadhyaya
Iowa State University
14 PUBLICATIONS **150** CITATIONS

On Accelerating Source Code Analysis At Massive Scale

Ganesha Upadhyaya, *Member, IEEE,* and Hridesh Rajan, *Member, IEEE*

Abstract—Encouraged by the success of data-driven software engineering (SE) techniques that have found numerous applications e.g. in defect prediction, specification inference, 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. This work proposes a technique that reduces the amount of computation performed by the ultra-large-scale source code mining task, especially those that make use of control and data flow analyses. Our key idea is to analyze the mining task to identify and remove the irrelevant portions of the source code, prior to running the mining task. We show a realization of our insight for mining and analyzing massive collections of control flow graphs of source codes. Our evaluation using 16 classical control-/data-flow analyses that are typical components of mining tasks and 7 Million CFGs shows that our technique can achieve on average a 40% reduction in the task computation time. Our case studies demonstrates the applicability of our technique to massive scale source code mining tasks.

Index Terms—Source code analysis, Mining Software Repositories, Ultra-large-scale Mining, Data-driven Software Engineering.

1 Introduction

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 [1], discovering programming patterns [2], [3], suggesting bug fixes [4], [5], specification inference [6], [7], [8], [9]. Approaches that perform source code mining and analysis at massive scale can be expensive. For example, a single exploratory run to mine API preconditions [8] 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 [10], [11], [12], 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., [13]. 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) [8]. While it is certainly feasible to improve the capabilities of the underlying infrastructure by adding many more CPUs, nonprofit infrastructures e.g. Boa [10] 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 [14], 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

• *The authors are with the Department of Computer Science, Iowa State University, Ames, IA 50011.*
E-mail: {ganeshau, hridesh}@iastate.edu

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 [10] (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, our paper 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 [10].
- 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.

Organization. The remainder of this paper is organized as follows. §2 provides a motivation for our technique and §3 describes the technique in detail. §4 and §5 presents our empirical evaluation and case studies respectively, and we discuss the related work in §6 before concluding in §7.

2 Motivation

Many source code mining tasks require performing control and data flow analysis over CFGs of a large collection of programs: API precondition mining [8], API usage pattern mining [7], [9], source code search [15], discovering vulnerabilities [16], to name a few. These source code mining tasks can be expensive. For example, consider the API precondition mining [8] 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 }
```

Fig. 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 1 shows an example client method that invokes `substring(int,int)` API method at line 7. This API

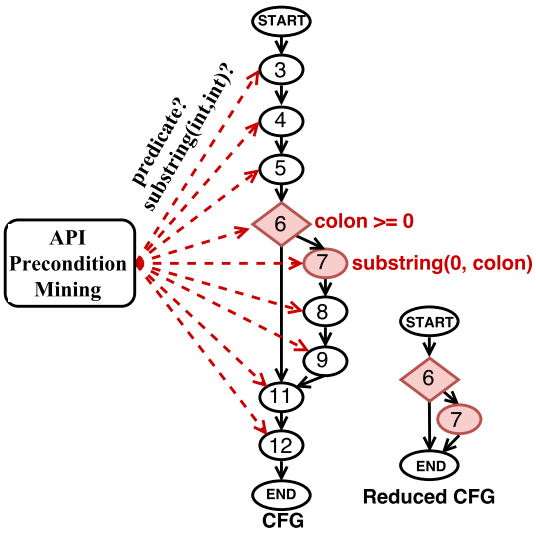

Fig. 2: When API precondition mining analysis is run on the CFG of the code shown in Figure 1, it visits every node in the CFG and queries whether the nodes have predicate expression or `substring(int,int)` API method call. If so, such nodes are relevant for API precondition mining.

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 2 (CFG node numbers corresponds to the line numbers in Figure 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 1, only line 6 & 7 are relevant (node 6 & 7 in the corresponding CFG shown in Figure 2). All other nodes are irrelevant and any computation performed on these nodes is not going to affect the mining results.

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 2 that contains only relevant nodes, the mining task can be accelerated substantially. As we show in our case study (§5.1), for the API precondition mining task, we were able to reduce the overall mining task time by 50%.

3 Approach

Figure 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.

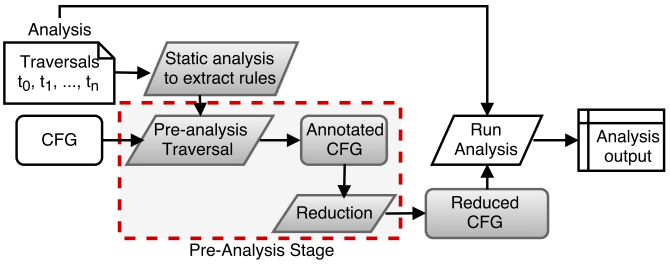

Fig. 3: An overview of our approach.

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 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 shows an example source code mining task written in Boa [10] (a domain-specific language for mining source code repositories) for mining API preconditions [8]. This source code mining task mainly contains three traversals: `mineT` (Figure 4a), `domT` (Figure 4b), and `analysisT` (Figure 4c), and a main program (Figure 4d) 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

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.

(a) The *mineT* traversal extracts all `substring` API call nodes and conditional nodes

```
1  // stores node ids of nodes with substring API call
2  apiCallNodes: set of int;
3
4  // stores conditions at nodes
5  conditionsAtNodes: map[int] of string;
6
7  hasSubstring := function(expr: Expression): bool {
8    // checks if there exists a substring API method invocation
9  }
10
11 isConditional := function(node: CFGNode): bool {
12   // checks if the CFG node has a conditional expression
13 }
14
15 mineT := traversal(node: CFGNode) {
16   if (def(node.expr) && hasSubstring(node.expr)) {
17     add(apiCallNodes, node.id);
18   }
19   if (isConditional(node)) {
20     conditionsAtNodes[node.id] = string(node.expr);
21   }
22 }
```

(b) The *domT* traversal computes a set of dominators for every CFG node

```
1  domT := traversal(node: CFGNode): set of int {
2    doms: set of int;
3
4    doms = getvalue(node);
5
6    if (!def(doms)) {
7      if (node.id == 0) {
8        s: set of int;
9        doms = s;
10     } else {
11       doms = allNodeIds;
12     }
13   }
14
15   foreach (i: int; def(node.predecessors[i])) {
16     pred := node.predecessors[i];
17     doms = intersection(doms, getvalue(pred));
18   }
19
20   add(doms, node.id);
21   return doms;
22 }
```

(c) The *analysisT* traversal computes the preconditions for every `substring` API call node

```
1  analysisT := traversal(node: CFGNode) {
2    preconditions: set of string;
3
4    if (contains(apiCallNodes, node.id)) {
5      doms := getvalue(node, domT);
6
7      foreach (dom: int = doms) {
8        if (haskey(conditionsAtNodes, dom)) {
9          add(preconditions, conditionsAtNodes[dom]);
10       }
11     }
12   }
13
14   location := ...
15   output[location] << preconditions;
16 }
```

(d) A visitor that invokes traversals on the CFG of every method in the project

```
1  visit(input, visitor {
2    before method: Method -> {
3      cfg := getcfg(method);
4
5      traverse(cfg, ..., mineT);
6
7      if (!isEmpty(apiCallNodes)) {
8        traverse(cfg, ..., domT);
9        traverse(cfg, ..., analysisT);
10     }
11   }
12 });
```

Fig. 4: API Precondition mining Boa program.

the dataset. We will use this example to describe the details of our technique.

As described in Figure 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.

3.1 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]

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

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. 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

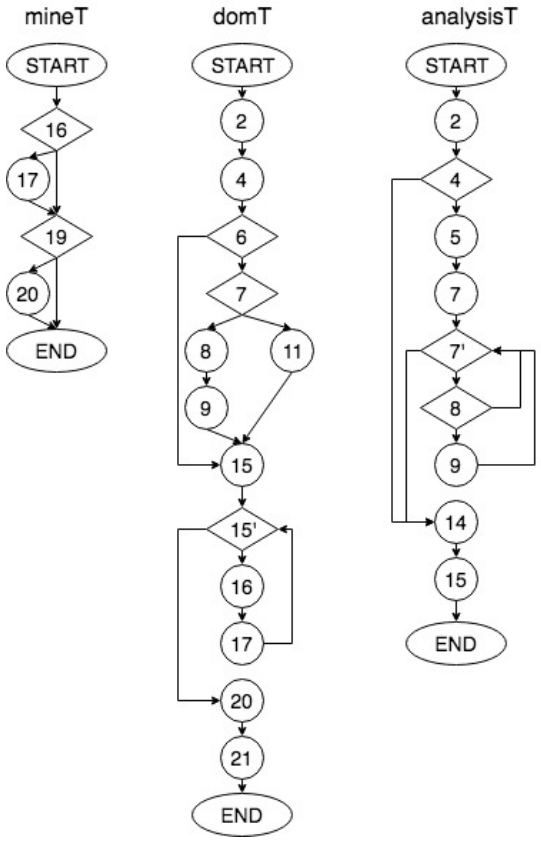

Fig. 5: Control flow graphs of the three traversals shown in Figure 4

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. 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 5. 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,

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 [17].

Algorithm 1: Extract rules from an analysis function

```
   Input: Set of paths Π, String iv, Set<String> ov
   Output: Rules set R
1  R ← {};
2  foreach π := (n_1, n_2, ..., n_k) ∈ Π do
3      pc ← true;
4      failToExtract ← false;
5      hasOutputExpr ← false;
6      foreach n_i ∈ π do
7          if n_i is a branch node then
8              α ← getAliasesAt(iv, n_i);
9              if getVariables(n_i) ∩ α ≠ φ then
10                 pe ← getPredicate(n_i);
11                 if pe is null then
12                     continue;
13                 if pe is true then
14                     failToExtract ← true;
15                     continue;
16                 if n_{i+1} is a true successor then
17                     pc ← pc ∧ pe;
18                 else
19                     pc ← pc ∧ ¬pe;
20         else
21             β ← getAliasesAt(ov, n_i);
22             if β ∩ getVariables(n_i) ≠ φ then
23                 hasOutputExpr ← true;
24                 if failToExtract is true then
25                     return {true};
26     if hasOutputExpr is true then
27         R ← R ∪ pc;
28 if R is empty then
29     R ← R ∪ true;
30 return R;
```

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 4c 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 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. 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

TABLE 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`

(iv) or its aliases, but not 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 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 (§3.2).

3.1.1 Example

We provide a running example of our static analysis using the API precondition mining task shown in Figure 4. 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 5 shows the CFG of `mineT` traversal, where the node numbers corresponds to the source line numbers in Figure 4. We can see that the `mineT` CFG contains 4 acyclic paths. For example, one such path is START $\rightarrow$  $\rightarrow$  $\rightarrow$  $\rightarrow$ 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 4a) 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 4a), 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)
}
```

Fig. 6: Rules set R produced by our static analysis for the API precondition mining task shown in Figure 4

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.

3.1.2 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.

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 [17], 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 [18]. 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) 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.

3.1.3 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 [17] 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 §4.5.

3.1.4 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

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.

TABLE 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

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 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 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.

3.2 Annotated Control Flow Graph

In §3.1 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 6. 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).

Algorithm 2: Pre-analysis traversal

```
Input: Control flow graph (CFG) G, Rules set R
Output: Annotated control flow graph (ACFG) G'
N_R := {};
foreach node n in G' do
    if n is ⊤ or ⊥ then
        N_R := N_R ∪ n;
    if n is a branch node then
        N_R := N_R ∪ n;
    foreach r ∈ R do
        if evalutes(r, n) then
            N_R := N_R ∪ n;
            break;
add N_R to G';
return G';
```

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.

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.

3.2.1 Example

Figure 7 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 6) 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).

3.3 Reduced Control Flow Graph

Using the annotated control flow graph (ACFG) that contains a set N_R of probable analysis relevant nodes, we

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.

```
1 _pre_analysis := traversal (node: CFGNode) {
2   // three rules from the rules set R
3   if (true && (def(node.expr) && hasSubstring(node.expr)) &&
4   ! isConditional(node)) {
5     add(cfg.rnodes, node);
6   }
7   if (true && !(def(node.expr) && hasSubstring(node.expr)) &&
8   isConditional(node)) {
9     add(cfg.rnodes, node);
10  }
11  if (true && (def(node.expr) && hasSubstring(node.expr)) &&
12  isConditional(node)) {
13    add(cfg.rnodes, node);
14  }
15
16  // rule to add default nodes
17  if (node.name == "START" || node.name == "END" || len(node.
        successors) > 1) {
18    add(cfg.rnodes, node);
19  }
20 }
```

Fig. 7: The generated pre-analysis traversal for evaluating rules in the rules set for API precondition mining task

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.

3.4 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 8 shows several examples of the reduction. For instance, in the first example, where node (j) is being pruned,

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'' ← G';
2  foreach node n in G'' do
3      if n ∉ 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);
```

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

3.5 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

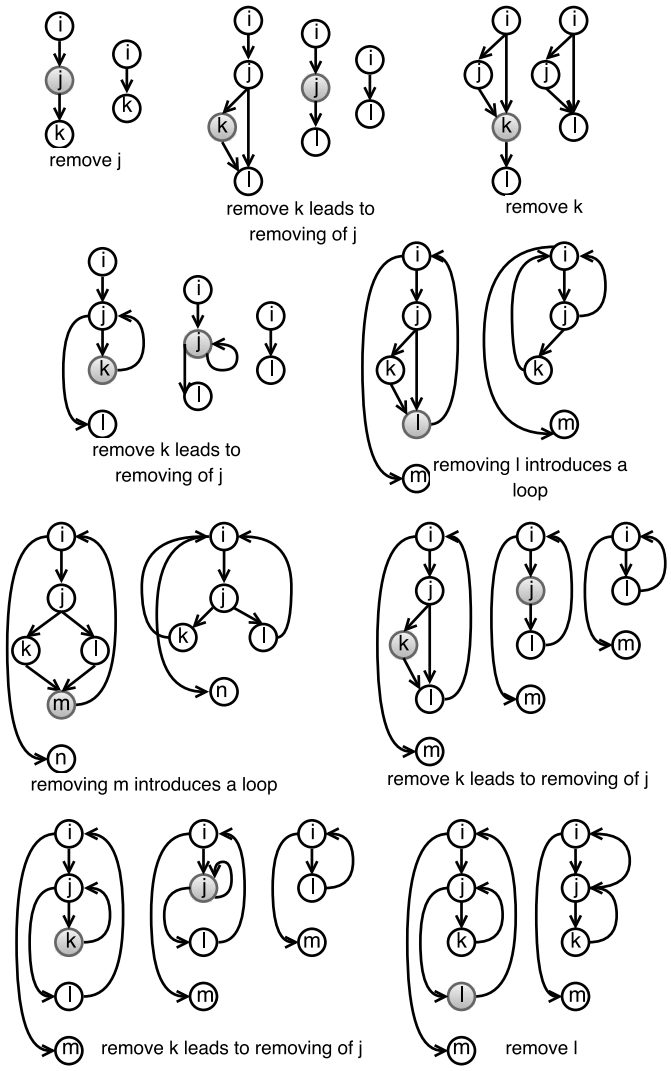

Fig. 8: ACFG to RCFG reduction examples.

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.

3.6 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 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? (§4.2)
2) How does our approach scale when the dataset size is increased uniformly? (§4.3)
3) What is the runtime overhead of our approach for performing a pre-analysis that identifies and removes the irrelevant nodes? (§4.4)
4) What is the compile-time overhead of the static analysis component of our approach that computes a rules set to identify relevant nodes? (§4.5)

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

TABLE 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								

4.1 Experimental Setup

4.1.1 Analyses

Table 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 mining API preconditions in [8]. 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 [10], 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.

4.1.2 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 [19], 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.

4.1.3 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 §4.4. We also discuss the compile-time overhead (for extracting rules) in §4.5. For measuring the execution times for Baseline and RCFG approaches, we use the methodology proposed by Georges *et al.* [20], 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.

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

4.2 Reduction In Analysis Time

We measured reduction in the analysis time of RCFG over Baseline. The results of our measurement is shown in Table 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 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 3), the reduction was either small or no reduction was seen. We first discuss the favorable cases and then provide insights into unfavorable cases.

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 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 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.

TABLE 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

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 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 §4.6.

```
1 public void rtrim() {
2   int index = text.length();
3   while ((index \> 0) && (text.charAt(
          index - 1) <=")) {
4     index--;
5   }
6   text = text.substring(0, index);
7 }
```

```
1 private static Color parseAsSystemColor(
        String value)
2             throws PropertyException {
3   int poss = value.indexOf("(");
4   int pose = value.indexOf(")");
5   if (poss != -1 && pose != -1) {
6     value = value.substring(poss + 1, pose);
7   } else {
8     throw new PropertyException("Unknown
            color format: " + value
9     + ". Must be system-color(x)");
10  }
11  return colorMap.get(value);
12 }
```

```
1 private int [] parseIntArray(Parser p)
        throws IOException {
2   IntArray array = new IntArray();
3   boolean done = false;
4   do {
5     String s = p.getNextToken();
6     if (s.startsWith("["))
7       s = s.substring(1);
8     if (s.endsWith("]")) {
9       s = s.substring(0, s.length() - 1);
10      done = true;
11    }
12    array.add(Integer.parseInt(s));
13  } while (!done);
14  return array.trim();
15 }
```

Fig. 9: Three methods from the DaCapo dataset. Highlighted parts are the relevant statements for PM analysis.

Same set of methods can demonstrate different amounts of relevant statements for various analyses. For example, consider the three methods shown in Figure 9. For PM analysis, the statements that invoke substring API method and conditional statements are the relevant statements. As highlighted in Figure 9, 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 9 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.

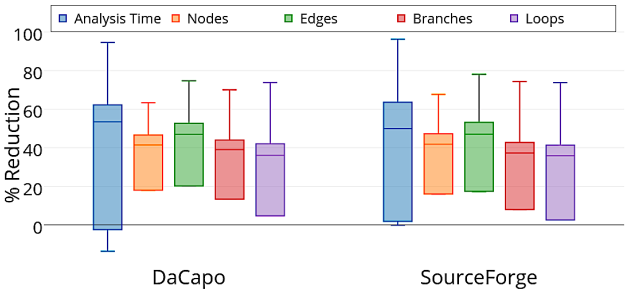

Fig. 10: Box plots of % reduction in analysis time, and graph size metrics across 16 analysis for DaCapo and SourceForge datasets.

Figure 10 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 [10] 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

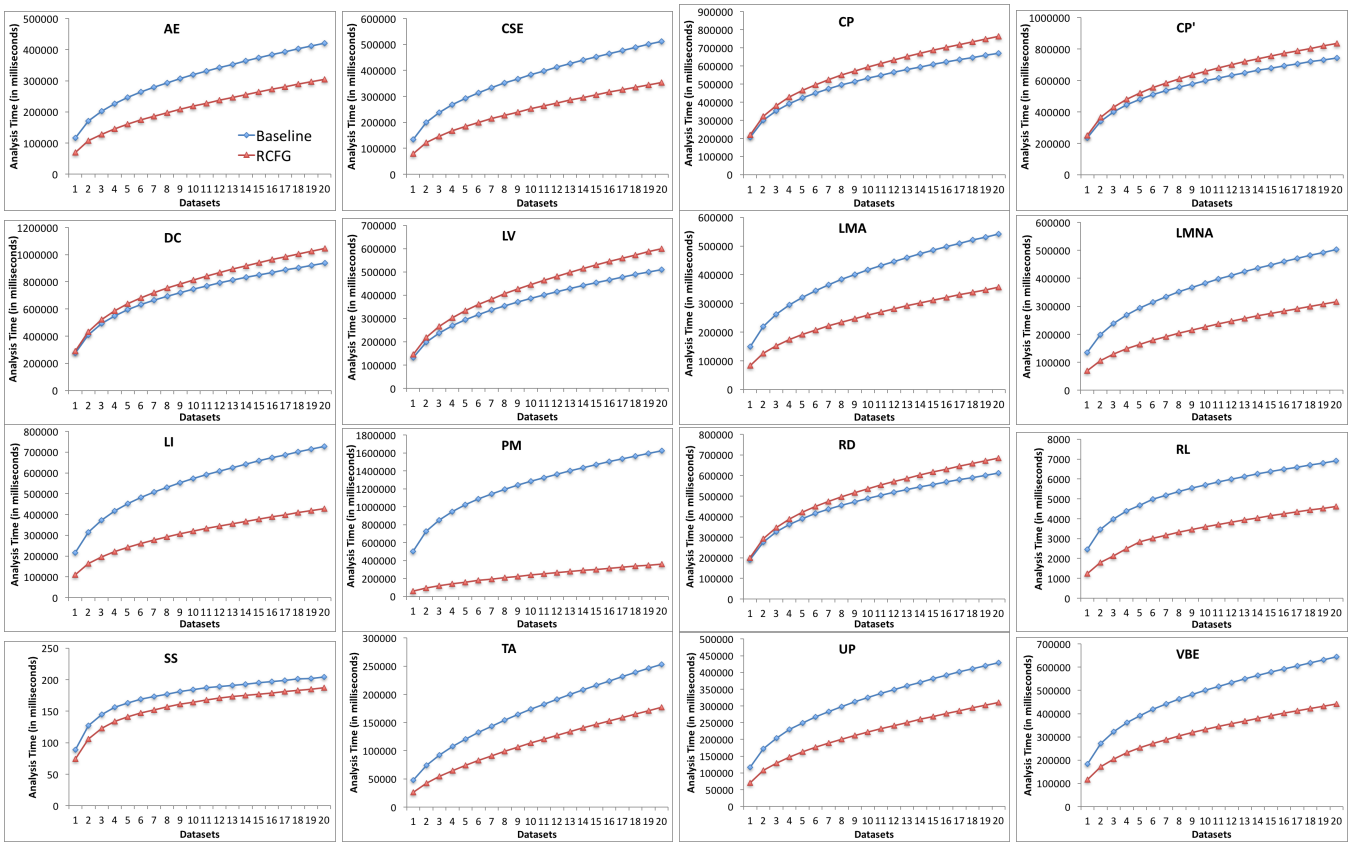

Fig. 11: Scalability of 16 analyses for `Baseline` and `RCFG` approaches. Each chart contains two lines, `Baseline` (blue) and `RCFG` (red). Each line contains 20 data points representing 20 datasets.

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 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 $bucket_i$.

We measure the analysis time of the `Baseline` and the `RCFG` approaches for all 16 analyses and plot the result in Figure 11. 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).

4.4 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.

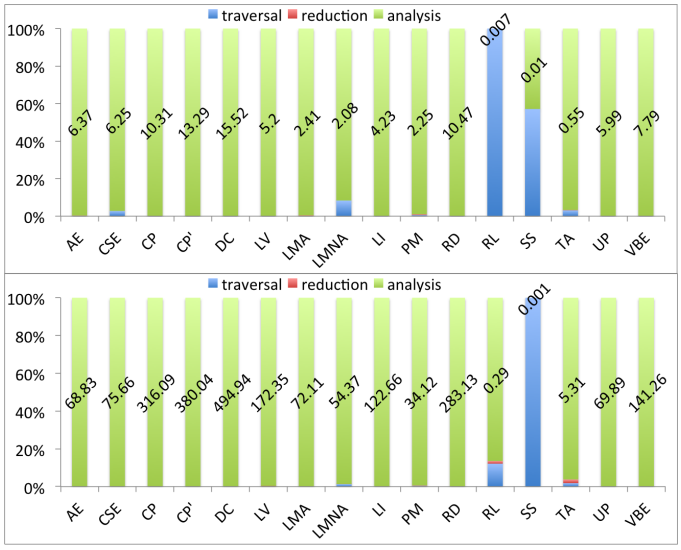

Fig. 12: Distribution of `RCFG` analysis time into i) *traversal*, ii) *reduction*, and iii) *analysis* for DaCapo and SourceForge datasets. X-axis: 16 analyses, Y-axis: % Reduction, DataLabels: `RCFG` time in seconds.

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 12 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 12, 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 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.

4.5 Overhead of Static Analysis

We presented our static analysis in section §3.1 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 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 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).

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.

4.6 Configurable Pruning

Our performance results described in §4.2 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%).

TABLE 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

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 12), 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 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 (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,

TABLE 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

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).

4.7 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.

4.8 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.

5 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].

5.1 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 1. This explains the reduction in the task time.

5.2 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 determine the data-dependence between the API method invocations [9], [15]. 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.

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).

5.3 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.

5.4 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.

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.

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`

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.

6 Related Work

There has been works that accelerates points-to analysis by performing pre-analysis and program compaction [21], [22]. Allen *et al.* [21] 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.* [22] 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 [23], [24]. For example, Wu *et al.* [23] uses the idea to improve the efficiency of the call trace collection and Ding *et al.* [24] 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 [25], [26]. 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 [27], [28], [29], [30]. Kulkarni *et al.* [27] 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 [28], [29], [30]. 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 [32], [33] 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 [14]. Program slicing has been shown to be useful in analyzing, debugging,

and optimizing programs. For example, Lokuciejewski *et al.* [34] 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.

7 Conclusion

Data-driven software engineering demands mining and analyzing source code repositories at massive scale, and this activity can be expensive. Extant techniques have focused on leveraging distributed computing techniques to solve this problem, but with a concomitant increase in the computational resource needs. This work proposes a complementary technique that reduces the amount of computation performed by the ultra-large-scale source code mining tasks without compromising the accuracy of the results. The key idea is to analyze the mining task to identify and remove parts of the source code that are irrelevant to the mining task prior to running the mining task. We have described a realization of our insights for mining tasks that performs control and data flow analysis at massive scale. Our evaluation using 16 classical control and data flow analyses has demonstrated substantial reduction in the mining task time. Our case studies demonstrated the applicability of our technique to massive-scale source code mining tasks.

While the proposed technique has shown to scale large-scale source code mining tasks that perform method-level control and data flow analysis, an immediate avenue of future work is to extend our technique to scale mining tasks that perform cross-method control and data flow analysis (or project-level analysis). Source code mining tasks based on other source code intermediate representations like abstract syntax trees (ASTs), callgraphs (CGs), and program dependence graphs (PDGs) may also benefit from the core ideas of our technique. In our evaluation we observed that a large number of reduced control flow graph (RCFG) representations of programs in our datasets looked similar in terms of nodes and edges. This provokes an interesting investigation whether this similarity can be used to run the mining task on only unique reduced control flow graphs and reuse the results. We plan to investigate such future directions to further our overarching goal of scaling massive-scale source code mining tasks.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under grants CCF-15-18789, CCF-15-12947, and CCF-14-23370. The views expressed in this work are those of the authors and do not necessarily reflect the official policy or position of the NSF. The authors would like to thank the anonymous reviewers of TSE, ESEC/FSE 2017, and Dr. Wei Le for constructive comments and suggestions for improving the manuscript.

References

[1] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Engg.*, vol. 17, no. 4-5, pp. 531–577, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9173-9

[2] A. Wasylkowski, A. Zeller, and C. Lindig, "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*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 35–44. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287632

[3] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 283–294. [Online]. Available: http://dx.doi.org/10.1109/ASE.2009.72

[4] B. Livshits and T. Zimmermann, "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*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 296–305. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081754

[5] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 176–192, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1109/TSE.2006.28

[6] H. Rajan, T. N. Nguyen, G. T. Leavens, and R. Dyer, "Inferring behavioral specifications from large-scale repositories by leveraging collective intelligence," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 579–582. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819107

[7] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "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*, ser. ICSE '18. New York, NY, USA: ACM, 2018.

[8] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of apis in large-scale code corpus," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 166–177. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635924

[9] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: From usage scenarios to specifications," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 25–34. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287630

[10] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 422–431. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486844

[11] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Sci. Comput. Program.*, vol. 79, pp. 241–259, Jan. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2012.04.008

[12] G. Gousios, "The ghtorent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 233–236. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487132

[13] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "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*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 779–790. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568295

[14] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=800078.802557

[15] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1069–1087, Sep. 2012. [Online]. Available: http://dx.doi.org/10.1109/TSE.2011.84

[16] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 590–604. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.44

[17] A. Diwan, K. S. McKinley, and J. E. B. Moss, "Type-based alias analysis," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 106–117. [Online]. Available: http://doi.acm.org/10.1145/277650.277670

[18] G. Snelting, T. Robschink, and J. Krinke, "Efficient path conditions in dependence graphs for software safety analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 4, pp. 410–457, Oct. 2006. [Online]. Available: http://doi.acm.org/10.1145/1178625.1178628

[19] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "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*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 169–190. [Online]. Available: http://doi.acm.org/10.1145/1167473.1167488

[20] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 57–76. [Online]. Available: http://doi.acm.org/10.1145/1297027.1297033

[21] N. Allen, B. Scholz, and P. Krishnan, "Staged points-to analysis for large code bases," in *Compiler Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 131–150.

[22] Y. Smaragdakis, G. Balatsouras, and G. Kastrinis, "Set-based pre-processing for points-to analysis," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 253–270. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509524

[23] R. Wu, X. Xiao, S.-C. Cheung, H. Zhang, and C. Zhang, "Casper: An efficient approach to call trace collection," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 678–690. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837619

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

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

[26] S. Kothari, A. Tamrawi, J. Sauceda, and J. Mathews, "Let's verify linux: Accelerated learning of analytical reasoning through automation and collaboration," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 394–403. [Online]. Available: http://doi.acm.org/10.1145/2889160.2889192

[27] S. Kulkarni, R. Mangal, X. Zhang, and M. Naik, "Accelerating program analyses by cross-program training," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: ACM, 2016, pp. 359–377. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984023

[28] A. Rountev, M. Sharp, and G. Xu, "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*, ser. CC'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 53–68. [Online]. Available: http://dl.acm.org/citation.cfm?id=1788374.1788380

[29] D. Yan, G. Xu, and A. Rountev, "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*, ser. SOAP '12. New York, NY, USA: ACM, 2012, pp. 9–14. [Online]. Available: http://doi.acm.org/10.1145/2259051.2259053

[30] K. Ali and O. Lhoták, "Application-only call graph construction," in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ser. ECOOP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 688–712.

[31] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali, "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*, ser. POPL '10. New York, NY, USA: ACM, 2010, pp. 43–56. [Online]. Available: http://doi.acm.org/10.1145/1706299.1706307

[32] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61. [Online]. Available: http://doi.acm.org/10.1145/199448.199462

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

[34] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, "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*, ser. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 136–146. [Online]. Available: http://dx.doi.org/10.1109/CGO.2009.17

Ganesha Upadhyaya is a doctoral candidate at Iowa State University. His research interests include program analysis, mining software repositories, and concurrent programming. He has published and presented several works at OOPSLA, ICSE, MSR, Modularity, and AGERE. He is a member of the IEEE.

Hridesh Rajan is the Kingland Professor in the Computer Science Department at Iowa State University (ISU) where he has been since 2005. His research interests include programming languages, software engineering, and data science. He founded the Midwest Big Data Summer School to deliver broadly accessible data science curricula and serves as a Steering Committee member of the Midwest Big Data Hub (MBDH). He has been recognized by the US National Science Foundation (NSF) with a CAREER award in 2009 and by the college of LAS with an Early Achievement in Research Award in 2010, and a Big-12 Fellowship in 2012. He is a senior member of the ACM and of the IEEE.