BCFA: Bespoke Control Flow Analysis for CFA at Scale

Ramanathan Ramu
Iowa State University
rramu@iastate.edu

Ganesha B Upadhyaya
Iowa State University
ganeshau@iastate.edu

Hoan A Nguyen
Iowa State University
hoan@iastate.edu

Hridesh Rajan
Iowa State University
hridesh@iastate.edu

ABSTRACT

Many data-driven software engineering tasks such as discovering programming patterns, mining API specifications, etc., perform source code analysis over control flow graphs (CFGs) at scale. Analyzing millions of CFGs can be expensive and performance of the analysis heavily depends on the underlying CFG traversal strategy. State-of-the-art analysis frameworks use a fixed traversal strategy. We argue that a single traversal strategy does not fit all kinds of analyses and CFGs and propose *bespoke control flow analysis* (BCFA). Given a control flow analysis (CFA) and a large number of CFGs, BCFA selects the most efficient traversal strategy for each CFG. BCFA extracts a set of properties of the CFA by analyzing the code of the CFA and combines it with properties of the CFG, such as branching factor and cyclicity, for selecting the optimal traversal strategy. We have implemented BCFA in *Boa*, and evaluated BCFA using a set of representative static analyses that mainly involve traversing CFGs and two large datasets containing 287 thousand and 162 million CFGs. Our results show that BCFA can speedup the large scale analyses by 1%-28%. Further, BCFA has low overheads; less than 0.2%, and low misprediction rate; less than 0.01%.

ACM Reference Format:
Ramanathan Ramu, Ganesha B Upadhyaya, Hoan A Nguyen, and Hridesh Rajan. 2020. BCFA: Bespoke Control Flow Analysis for CFA at Scale. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, May 23–29, 2020, Seoul, South Korea.* ACM, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Data-driven techniques have been increasigly adopted in many software engineering (SE) tasks: API precondition mining [21], API usage mining [1, 32], code search [20], discovering vulnerabilities [30], to name a few. These data-driven SE tasks perform source code analysis over different program representations like source code text, abstract syntax trees (ASTs), control-flow graphs (CFGs), etc, at scale. For example, API precondition mining analyzes millions of methods that contains API call sites to capture conditions that are often checked before invoking the API. The source code mining infrastructures [5, 9, 11] have started supporting CFG-level analysis to facilitate variety of data-driven SE tasks.

Performance of source code analysis over CFGs heavily depends on the order of the nodes visited during the traversals: *the traversal strategy*. Several graph traversal strategies exists from the graph traversal literatures, e.g., depth-first, post-order, reverse post-order, topological order, worklist-based strategies, etc, however the state-of-the-art analysis frameworks use fixed traversal strategy. For example, Soot analysis framework [17] uses topological ordering of the nodes to perform control flow analysis. Our observation is that for analyzing millions of programs with different characteristics, no single strategy performs best for all kinds of analyses and programs. Both properties of the analysis and the properties of the input program influence the traversal strategy selection. For example, for a control flow analysis that is data-flow sensitive, meaning the output for any node must be computed using the outputs of its neighbors, a traversal strategy that visits neighbors prior to visiting the node performs better than other kinds of traversal strategies. Similarly, if the CFG of the input program is sequential, a simple strategy that visits nodes in the random order performs better than a more sophisticated strategy.

We propose *bespoke control flow analysis* (BCFA), a novel source code analysis technique for performing large scale source code analysis over the control flow graphs. Given an analysis and a large collection of CFGs on which the analysis needs to be performed, BCFA selects an optimal traversal strategy for each CFG. In order to achieve that, BCFA deploys a novel decision tree that combines a set of analysis properties with a set of graph properties of the CFG. The analysis properties include data-flow sensitivity, loop sensitivity, and traversal direction, and the graph properties include cyclicity (whether the CFG contains branches and loops). There exists no technique that automatically selects a suitable strategy based on analysis and CFG properties. Since manually extracting the properties can be infeasible when analyzing millions of CFGs, we provide a technique to extract the analysis properties by analyzing the source code of the analysis.

We have implemented BCFA in Boa, a source code mining infrastructure [9] and evaluated BCFA using a set of 21 source code analyses that includes mainly control and data-flow analyses. The evaluation is performed on two datasets: a dataset containing well-maintained projects from DaCapo benchmark (with a total of 287K CFGs), and an ultra-large dataset containing more than 380K projects from GitHub (with a total of 162M CFGs). Our results showed that BCFA can speedup the large scale analyses by 1%-28% by selecting the most time-efficient traversal strategy. We also found that, BCFA

ICSE '20, May 23–29, 2020, Seoul, South Korea
© 2020 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

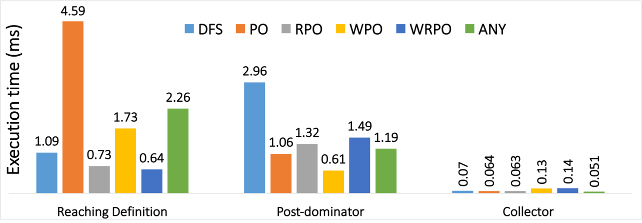

Figure 1: Running times (in ms) of the three analyses on graph A using different traversal strategies (lower better).

has low overheads for computing the analysis and graph properties; less than 0.2%, and low misprediction rate; less than 0.01%.

In summary, this paper makes the following contributions:

- It proposes a set of analysis properties and a set of graph properties that influences the selection of traversal strategy for CFGs.
- It describes a novel decision tree for selecting the most suitable traversal strategy using the analysis and the graph properties.
- It provides a technique to automatically extract the analysis properties by analyzing the source code of the analysis.
- It provides an implementation of BCFA in Boa [9] and a detailed evaluation using a wide-variety of source code analyses and two large datasets containing 287 thousand and 162 million CFGs.

2 MOTIVATION

Consider a software engineering task that infers the temporal specifications between pairs of API method calls, i.e., a call to *a* must be followed by a call to *b* [10, 23, 28, 31]. A data-driven approach for inference is to look for pairs of API calls that frequently go in pairs in the same order at API call sites in the client code. Such an approach contains (at least) three source code analyses on the control flow graph (CFG) of each client method: 1) identifying references of the API classes and call sites of the API methods which can be done using *reaching definition* analysis [22]; 2) identifying the pairs of API calls (*a*, *b*) where *b* follows *a* in the client code using *post-dominator* analysis [2]; and 3) collecting pairs of temporal API calls by traversing all nodes in the CFG—let us call this *collector* analysis. These analyses need to be run on a large number of client methods to produce temporal specifications with high confidence.

Implementing each of these analyses involves traversing the CFG of each client method. The traversal strategy could be chosen from a set of standard strategies e.g., depth-first search (DFS), post-order (PO), reverse post-order (RPO), worklist with post-ordering (WPO), worklist with reverse post-ordering (WRPO) and any order (ANY).[1]

Figure 1 shows the performance of these three analyses when using standard traversal strategies on the CFG, referred to as A, of a method in the DaCapo benchmark [6]. The CFG has 50 nodes and it has branches but no loops. Figure 1 shows that, for graph A, WRPO performs better than other strategies for the reaching definition analysis, while WPO outperforms the others for the post-dominator analysis and the ANY traversal works best for the collector analysis.

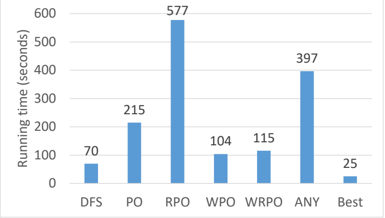

Figure 2: Running times (in s) of three analyses using different traversal strategies on a large codebase (lower better).

No Traversal Strategy Fits All and Analysis Properties Influence the Selection. The performance results show that there is no single traversal strategy that works best for all analyses. Reaching definition analysis is a forward data-flow analysis where the output at each node in the graph is dependent on the outputs of their predecessor nodes. So, DFS, RPO and WRPO by nature are the most suitable. However, worklist is the most efficient strategy here because it visits only the nodes that are yet to reach fixpoint[2] unlike other strategies that also visit nodes that have already reached fixpoint. Post-dominator analysis, on the other hand, is a backward analysis meaning that the output at each node in the graph is dependent on the outputs of their successor nodes. Therefore, the worklist with post-ordering is the most efficient traversal. For the collector analysis, any order traversal works better than other traversal strategies for graph A. This is because for this analysis the output at each node is not dependent on the output of any other nodes and hence it is independent of the order of nodes visited. The any order traversal strategy does not have the overhead of visiting the nodes in any particular order like DFS, PO, RPO nor the overhead to maintain the worklist.

Properties of Input Graph Determine Strategy. For the illustrative example discussed above, DFS and RPO were worse than WRPO for the reaching definition analysis and PO was worse than WPO for post-dominator because they require one extra iteration of analysis before realizing that fixpoint has been reached. However, since graph A does not have any loops, if the graph A's nodes are visited in such a way that each node is visited after its predecessors for reaching definition analysis and after its successors for post-dominator analysis, then the additional iteration is actually redundant. Given that graph A has no loops, one could optimize RPO or PO to skip the extra iteration and fixpoint checking. Thus, the optimized RPO or PO would run the same number of iterations as the respective worklist-based ones and finish faster than them because the overhead of maintaining the worklist is eliminated.

The potential gains of selecting a suitable traversal strategy could be significant. To illustrate, consider Figure 2 that shows the performance of our entire illustrative example (inferring temporal specifications) on a large corpus of 287,000 CFGs extracted from the DaCapo benchmark dataset [6]. The bar chart shows the total running times of the three analyses. **Best** is an ideal strategy where we can always choose the most efficient with all necessary optimizations. The bar chart confirms that **Best** strategy could reduce

[1] In DFS, successors of a node are visited prior to visiting the node. PO is similar to DFS, except that there is no order between the multiple successors. In RPO, predecessors of a node are visited prior to visiting the node. WPO and WRPO are the worklist-based strategies, in which the worklist is a datastructure used to keep track of the nodes to be visited, worklist is initialized using either PO or RPO (PO for WPO and RPO for WRPO), and whenever a node from the worklist is removed and visited, all its successors (for forward direction analysis) or its predecessors (for backward direction analysis) are added to the worklist.

[2] Fixpoint is a state where the outputs of nodes no longer changes.

the running time on a large dataset from 64% (against DFS) to 96% (against RPO). Three tasks could be completed for the price of one.

On the other hand, the poor selection of the traversal strategy could make an analysis infeasible when performed at scale. Consider application [21] that uses post-dominator (PDOM) and reaching definitions (RD) analyses. For PDOM, a poor strategy (e.g., RPO, WRPO) would make the analysis never finish on our GitHub dataset. Similarly, for RD, a poor strategy (WPO on GitHub) would be 80% slower. Figure 6a shows several ‘-‘ for the GitHub column, indicating that picking a poor strategy makes the analysis infeasible.

3 BESPOKE CONTROL FLOW ANALYSIS

Figure 3 provides an overview of BCFA and its key components. The inputs are a source code analysis that contains one or more traversals (§3.7), and a graph. The output is an optimal traversal strategy for every traversal in the analysis. For selecting an optimal strategy for a traversal, BCFA computes a set of static properties of the analysis (§3.2), such as data-flow and loop sensitivity, and extracts a runtime property about the cyclicity in the graph (sequential/branch/loop). Using the computed analysis and graph properties, BCFA selects a traversal strategy from a set of candidates (§3.4) for each traversal in the analysis (§3.5) and optimizes it (§3.6). The analysis properties are computed only once for each analysis, whereas graph cyclicity is determined for every input graph.

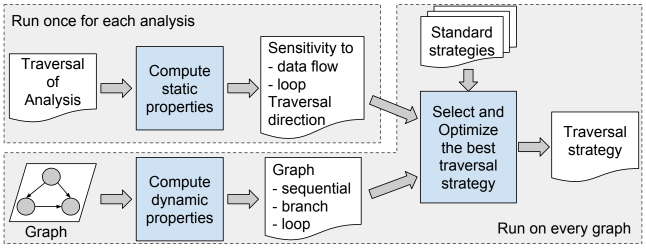

Figure 3: Overview of bespoke control flow analysis.

3.1 The Source Code Analysis Model

A source code analysis can be performed on either the source code text or the intermediate representations like control flow graph (CFG), program dependence graph (PDG), and call graph (CG). An *iterative data-flow analysis* over a CFG traverses the CFG by visiting the nodes and computing outputs for nodes (aka analysis facts) [3]. For computing an output for a node the analysis may use the outputs of node’s neighbors (successors or predecessors). For example, consider the *Reaching definitions (RD)* analysis [22] that computes a set of variable definitions that are available at each node in a CFG. Here, the output or the analysis fact at a node is a set of variables. The RD analysis performs two traversals of the CFG, where in the first traversal it visits every node in the CFG starting from the entry node and collects variables defined at nodes, and in the second traversal, the variables defined at the predecessors are propagated to compute the final set of variables at nodes. Note that, certain nodes may be visited several times (in case of loops in the CFG) until a fixpoint is reached.[2] A complex analysis may further contain one or more sub-analyses that traverses the CFG to compute different analysis facts.

3.2 Static Properties of the Input Analysis

These properties are the data-flow sensitivity, loop sensitivity, and traversal direction. We now describe these properties in detail and provide a technique to automatically compute them in §3.7.

3.2.1 *Data-Flow Sensitivity.*

The *data-flow sensitivity* property of a traversal models the dependence among the traversal outputs of nodes in the input graph. A traversal is data-flow sensitive if the output of a node is computed using the outputs of other nodes. For example, the *Reaching definitions (RD)* analysis described previously performs a traversal that aggregates the variable definitions from the predecessors of a node while computing the output for the node, hence it is considered data-flow sensitive.

A key observation that can be made from the description of the data-flow sensitivity property is that, a traversal strategy that visits the neighbors of a node prior to visiting the node will be more efficient than other kinds of traversal strategies, because the output of neighbors will be available prior to computing the output for the node and hence fixpoint can be reached faster.

3.2.2 *Loop Sensitivity.*

The *loop sensitivity* property models the effect of loops in the input graph on the analysis. If an input graph contains loops and if the analysis traversal is affected by them, the traversal may require multiple iterations (where certain nodes are visited multiple times) to compute the final output at nodes. In these iterations, the traversal outputs at nodes either shrinks or expands to reach a fixpoint.[3] Hence, a traversal is loop sensitive if the output of any node shrinks or expands in the subsequent iterations. As one could imagine the loop-sensitivity property can be difficult to compute, hence we have provided a technique in §3.7. The *Reaching definitions (RD)* analysis described previously has two traversal, in which the first traversal that simply extracts the variables defined at nodes is not loop-sensitive, whereas the second traversal that propagates the variables from the predecessors is loop-sensitive. *If an analysis traversal is not loop-sensitive, the loops in the input CFG will not have any influence on the selection of the traversal strategy.*

3.2.3 *Traversal Direction.*

A *traversal direction* is property that describes the direction of visiting the CFG. A CFG can be visited in either the FORWARD direction or the BACKWARD direction. In FORWARD, the entry node of the CFG is visited first, followed by the successors of the nodes, where in BACKWARD, the exit nodes of the CFG are visited first, followed by the predecessors of the nodes. The traversal direction is often specified by the analysis writers.

3.3 Graph Cyclicity of Input CFGs

So far we have described the three static properties of the analysis that influences the traversal strategy selection. A property, cyclicity, of the input graph also influences the selection. Based on the cyclicity, we classify graphs into four categories: sequential, branch only, loop w/o branch and loop w/ branch. In sequential graphs, all nodes have no more than one successor and predecessor. In graphs

[3] Shrink and expand are defined with repsect to the size of the datastructure used to store the analysis outputs at nodes. As the size is a dynamic factor, we determine the shrink or expand property by tracking the operations performed on the datastructure used to store the output. For example, ADD operation always expands and REMOVE always shrinks. §3.7 describes this in detail.

with branches, nodes may have more than one successors and predecessors. In graphs with loops, there exists cycles in the graphs. The graph cyclicity is determined during graph construction.

Graph cyclicity plays an important role in the selection of the appropriate traversal strategy along with the analysis property for an (analysis, CFG) pair. For CFGs with branches and loops, the outputs of all dependent nodes of a node (predecessors or successors) may not be available at the time of visiting the node, hence a traversal strategy must be selected that guarantees that the outputs of all dependent nodes of a node are available prior to computing the node's output, as it leads to fixpoint convergence in fewer iterations.

3.4 Candidate Traversal Strategies

For every (analysis, CFG) pair, BCFA selects a traversal strategy from among the candidate traversal strategies listed next. *1) Any order (ANY):* nodes can be visited in any order (random). *2) Increasing order of node ids (INC):* nodes are visited in the increasing order of node ids. Node ids are assigned during the CFG construction based on the control flow order. *3) Decreasing order of node ids (DEC):* nodes are visited in the decreasing order of node ids. *4) Postorder (PO):* successors of a node is visited before visiting the node. *5) Reverse postorder (RPO):* predecessors of a node is visited before visiting the node. *6) Worklist with postorder (WPO):* nodes are visited in the order they appear in the worklist. Worklist is a datastructure used to keep track of the nodes to be visited. Worklist is initialized with postordering of the nodes. Whenever a node from the worklist is removed and visited, all its successors (for FORWARD) or predecessors (for BACKWARD) are added to the worklist as described in [3]. *7) Worklist with reverse postorder (WRPO):* similar to WPO, except that the worklist is initialized with reverse postordering.

The traversal strategies were selected by carefully reviewing the compiler textbooks, implementations, and source code analysis frameworks. The selected strategies are generally applicable to any analyses and graphs.[4]

3.5 Traversal Strategy Decision Tree

The core of BCFA is a decision tree for selecting an optimal traversal strategy shown in Figure 4. The leaf nodes of the tree are one of the seven traversal strategies and non-leaf nodes are the static and runtime checks. The decision tree has eleven paths marked from P_1 through P_{11}. Given a traversal and an input graph, one of the eleven paths will be taken to decide the best traversal strategy. The static checks are marked green and the runtime checks are marked red. The static properties includes data-flow sensitivity ($P_{DataFlow}$), loop sensitivity (P_{Loop}), and traversal direction. The runtime property that is checked is the graph cyclicity: sequential, branch, loop w/ branch, and loop w/o branch. Next, we describe the design rationale of the decision tree. To illustrate, we use the post-dominator analysis described in Listing 1, which has one data-flow insensitive traversal `initT` and one data-flow sensitive traversal `domT`, on a CFG shown in Figure 5 that has both branches and loops.

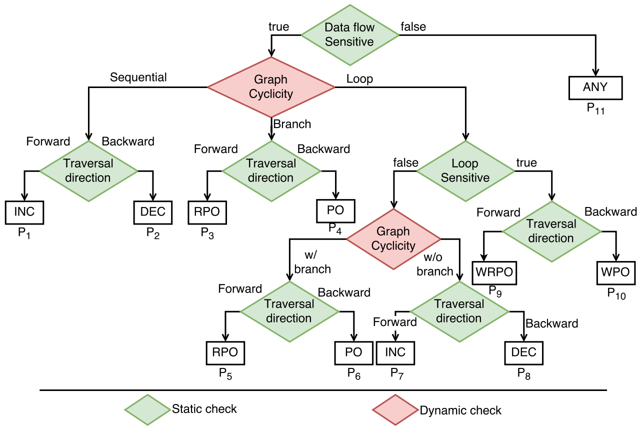

Figure 4: BCFA decision tree for selecting an optimal traversal strategy. P_0 to P_{11} are markings for paths.

The first property is the data-flow sensitivity (§3.2.1) which is computed by Algorithm 1. The rationale for checking this property first is that, if a traversal is not data-flow sensitive, the traversal can finish in a single iteration (no fixpoint computation is necessary), irrespective of the type of the input graph. In such cases, visiting nodes in any order will be efficient, hence we assign any order (*ANY*) traversal strategy (path P_{11}). This is the case for traversal `initT` in the post-dominator analysis (Listing 1).

For traversals that are data-flow sensitive, such as `domT` in the post-dominator analysis, we further check the cyclicity property of the input graphs. The loop sensitivity property is only applicable to graphs with loops, thus, is considered last.

Sequential Graphs (paths P_1 and P_2). In case of sequential graphs, each node has at most one predecessor or one successor, and the data flows to any node are from either its predecessor or its successor, not both. Thus, the most efficient strategy should traverse the graph in the direction of the data flow, so that the traversal can finish with only one iteration, and no fixpoint nor worklist is needed. We use traversal strategy *INC* (leaf node P_1) for FORWARD analysis and *DEC* (leaf node P_2) for BACKWARD analysis.

Graphs with branches (paths P_3 and P_4). These graphs contain branches but no loops, hence a node may have more than one successor or predecessor. If the traversal under consideration is data-flow sensitive, depending on the traversal direction, it requires the output of either the successors or predecessors to compute the output for any node. For FORWARD traversal direction, we need to visit all predecessors of any node prior to visiting the node and this traversal order is given by the post-order (*RPO*) and hence we select *RPO*. For BACKWARD traversal direction, we need to visit all successors of any node prior to visiting the node and this traversal order is given by the reverse post-order (*PO*), hence we select *PO*.

Graphs with loops (paths P_5 to P_{10}). These graphs contain both branches and loops, hence we need to check if the traversal is sensitive to the loop (the loop sensitive property). The decision for `domT` falls into these paths because it has loops.

Loop sensitive (paths P_9 and P_{10}). When the traversal is loop sensitive, for correctly propagating the output, the traversal visits nodes multiple times until a fix point condition is satisfied (user may provide a fix point function). We adopt a worklist-based traversal strategy because it visits only required nodes by maintaining a list of nodes left to be processed. For picking the best order of traversing

[4] We did not include strategies like chaotic iteration based on weak topological ordering because they are effective only for computing fixpoint of a continuous function over lattices of infinite height [7].

nodes in a graph to initialize the worklist, we further investigate the traversal direction. We know that, for FORWARD traversals, *RPO* strategy gives the best order of nodes and for BACKWARD traversals, *PO* strategy gives the best order of nodes, we pick worklist strategy with reverse post-order *WRPO* for FORWARD and worklist strategy with post-order *WPO* for BACKWARD traversal directions.

Loop insensitive (paths P_5 to P_8). When the traversal is loop insensitive, the selection problem reduces to the sequential and branch case that is discussed previously, because for loop insensitive traversal, the loops in the input graph is irrelevant and what remains relevant is the existence of the branches. For example, domT is loop insensitive, thus, is a case. As the graph contains branches, the next property to be checked is the traversal direction. Since the traversal direction for domT is BACKWARD, we pick *PO* traversal strategy for domT, as shown by the path P_6 in Figure 4.

3.6 Optimizing the Selected Traversal Strategy

The properties of the analysis and the input graph not only helps us select the best traversal strategy, it also helps perform several optimizations to the selected traversal strategies.

Running an analysis on a cyclic graph may require multiple iterations to compute the fixpoint solution. At least one last traversal is required to check that the results of the last two traversals have not changed. This additional traversal and the fixpoint checks at nodes can be eliminated based on the selected traversal strategy.

For data-flow insensitive traversals, the traversal outputs of nodes do not depend on the outputs of other nodes, hence both the additional traversal and the fixpoint checks can be eliminated irrespective of the graph cyclicity (decision path P_{11} in Figure 4).

In case of data-flow sensitive traversals, the traversal output of a node is computed using the traversal outputs of other nodes (predecessors or successors). For data-flow sensitive analysis on an acyclic graph, the additional traversal and fixpoint checks are not needed if selected traversal strategy guarantees that the nodes whose outputs are required to compute the output of a node are visited prior to visiting the node (paths P_1–P_4). In case of data-flow, loop sensitive traversals on graphs with loops, the optimization does not apply as no traversal strategy can guarantee that the nodes whose outputs are required to compute the output of a node are visited prior to visiting the node. Due to the presence of loops, BCFA selects the worklist-based traversal strategy (paths P_9 and P_{10}). However, in case of data-flow and loop in-sensitive traversals on graphs with loops, the loops can be ignored since the traversal is insensitive to it and the additional traversal and fixpoint checks can be eliminated if the selected traversal strategy guarantees that the nodes whose outputs are required to compute the output of a node, are visited prior to visiting the node (paths P_5–P_8).

3.7 Extracting Analysis Properties

In §3.2 we described a set of static properties of the analysis that influence the traversal strategy selection, we now provide a technique to automatically extract them by analyzing the analysis source code. Before that, we first describe the DSL that we chose (Boa) and the extensions that we added to support effective extraction of the static properties of the analysis programs written in this DSL.[5]

The Boa DSL provides APIs to access source code files, revisions, classes, methods, statements, expressions, etc. It also provides CFGs of methods over which the analysis can be performed. We extend Boa DSL to support writing iterative data-flow analysis.[6] A typical iterative data-flow analysis is described using: i) a traversal function (containing instructions to compute the analysis output for any CFG node), ii) a fixpoint function (that defines how to merge the outputs of predecessors or successors, if the analysis requires), and iii) the traversal direction.[7] We now describe our extension using a sample analysis shown in Listing 1.

Listing 1: Post-dominator analysis: an example source code analysis expressed using our DSL extension.

```
1  allNodes: Set<int>;
2  initT := traversal(n: Node) {
3    add(allNodes, n.id);
4  }
5  domT := traversal(n: Node): Set<int> {
6    Set<int> dom;
7    if (output(n, domT) != null) dom = output(n, domT);
8    else if (node.id == exitNodeId) dom = {};
9    else  dom = allNodes;
10   foreach (s : n.succs)
11     dom = intersection(dom, output(s, domT))
12   add(dom, n.id);
13   return dom;
14 }
15 fp := fixp(Set<int> curr, Set<int> prev): bool {
16   if (equals(curr, prev)) return true;
17   return false;
18 }
19 traverse(g, initT, ITERATIVE);
20 traverse(g, domT, BACKWARD, fp);
```

Post-dominator analysis. The post-dominator analysis is a backward control flow analysis that collects node ids of all nodes that post-dominates every node in the CFG [2]. This analysis can be expressed using our DSL extension as shown in Listing 1 that defines two traversals initT (lines 2-4) and domT (lines 5-14). A traversal is defined using a special traversal block:

```
t := traversal(n : Node) : T { tbody }
```

In this traversal block definition, t is the name of the traversal that takes a single parameter n representing the graph node that is being visited. A traversal may define a return type T representing the output type. The output type can be a primitive or a collection data type. In our example, the traversal initT does not define any output for CFG nodes, whereas, the traversal domT defines an output of type Set<int> for every node in the CFG. A block of code that generates the traversal output at a graph node is given by tbody. The tbody may contain common statements and expressions, such as variable declarations, assignments, conditional statements, loop statements, and method calls, along with some special expressions

[5] Our reason for selecting the Boa DSL is that it already provides an infrastructure to facilitate ultra-large scale analysis (several hundred millions of methods).

[6] The DSL extensions that we added are nothing specific to Boa, instead are common constructs that can be found in any framework that implements iterative data-flow style analysis, for example, Soot analysis framework.

[7] The common data structures to store the analysis outputs at CFG nodes is already available in Boa. The Boa DSL and our extension was sufficient to express all the analyses in our evaluation dataset.

discussed in this section. These traversals are invoked by using a special traverse expression (lines 19 and 20).

```
traverse(g, t, d, fp)
```

A traverse expression takes four parameters: g is the graph to be traversed, t is the traversal to be invoked, d is the traversal direction and fp is an optional variable name of the user-defined fixpoint function. A traversal direction is a value from the set {FORWARD, BACKWARD, ITERATIVE}. ITERATIVE represents a sequential analysis that visits nodes as they appear in the nodes collection.

Lines 15-18 defines a fixpoint function using fixp block, used in the traverse expression in line 20. fixp is a keyword for defining a fixpoint function. A fixpoint function can take 0 or more parameters, and must always return a boolean. A fixpoint function can be assigned a name, which can be passed in the traverse expression.

Line 1 defines a variable allNodes of type Set<int> which defines a collection type Set with elements of type int. The Boa DSL provides primitive and collection data types. Collection types include: Set and Seq, where Set is a collection with unique and unordered elements, whereas, Seq is a collection with non unique but ordered elements. Important set of operations that can be performed on collection types are add(C1, e) that adds an element e to collection C1, addAll(C1, C2) that adds all elements from collection C2 to collection C1, remove(C1, e) that removes an element e from Collection C1, removeAll(C1, C2) that removes all elements present in collection C2 from collection C1, union(C1, C2) that returns union of the elements in C1 and C2 and intersection(C1, C2) that returns intersection of the elements in C1 and C2. Line 3 in our example uses an operation add on collection allNodes.

A usage of special expression output(n, domT) can be seen in line 7. This is used for querying the traversal output associated with a graph node n, in the traversal domT. The traversal domT defines an output of type Set<int> for every node in the CFG. For managing the analysis output of nodes, domT traversal maintains an internal map that contains analysis output for every node, which can be queried using output(n, domT). A pre-defined variable g that represents the CFG is used in the traverse expressions in lines 19–20.

Figure 5 takes an example graph, and shows the results of initT and domT traversals. Our example graph is a CFG containing seven nodes with a branch and a loop. The initT traversal visits nodes sequentially and adds node id to the collection allNodes. The domT traversal visits nodes in the post-order[8] and computes a set of nodes that post-dominate every visited node (as indicated by the set of node ids). For instance, node 7 is post-dominated by itself, hence the output at node 7 is {7}. In Figure 5, under domT traversal, for each node visited, we show the key intermediate steps indicated by @ line number. These line numbers correspond to the line numbers shown in Listing 1. We will explain the intermediate results while visiting node 2. In the domT traversal, at line 9, the output set dom is initialized using allNodes, hence dom = {0, 1, 2, 3, 4, 5, 6, 7}. At line 10, node 2 has two successors: {3, 4}. At line 11, dom is updated by performing an intersection operation using the outputs of successors 3 and 4. Since the outputs of 3 and 4 are {1, 3, 5, 6, 7} and {1, 4, 5, 6, 7}, respectively, dom set becomes {1, 5, 6, 7}. At line 12, the id of the visited node is added to the dom set and it becomes {1, 2, 5, 6, 7}. Hence, the post-dominator set for node 2 is {1, 2, 5, 6, 7}. The post-dominator sets for other nodes are calculated similarly.

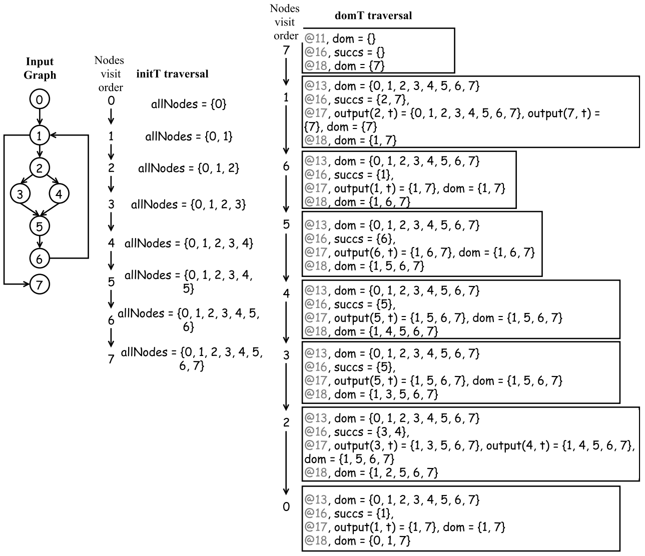
Input Graph
0
1
2
3
4
5
6
7
Nodes visit order
initT traversal
0 allNodes = {0}
1 allNodes = {0, 1}
2 allNodes = {0, 1, 2}
3 allNodes = {0, 1, 2, 3}
4 allNodes = {0, 1, 2, 3, 4}
5 allNodes = {0, 1, 2, 3, 4, 5}
6 allNodes = {0, 1, 2, 3, 4, 5, 6}
7 allNodes = {0, 1, 2, 3, 4, 5, 6, 7}
Nodes visit order
domT traversal
7
@11, dom = {}
@16, succs = {}
@18, dom = {7}
1
@13, dom = {0, 1, 2, 3, 4, 5, 6, 7}
@16, succs = {2, 7},
@17, output(2, t) = {0, 1, 2, 3, 4, 5, 6, 7}, output(7, t) = {7}, dom = {7}
@18, dom = {1, 7}
6
@13, dom = {0, 1, 2, 3, 4, 5, 6, 7}
@16, succs = {1},
@17, output(1, t) = {1, 7}, dom = {1, 7}
@18, dom = {1, 6, 7}
5
@13, dom = {0, 1, 2, 3, 4, 5, 6, 7}
@16, succs = {6},
@17, output(6, t) = {1, 6, 7}, dom = {1, 6, 7}
@18, dom = {1, 5, 6, 7}
4
@13, dom = {0, 1, 2, 3, 4, 5, 6, 7}
@16, succs = {5},
@17, output(5, t) = {1, 5, 6, 7}, dom = {1, 5, 6, 7}
@18, dom = {1, 4, 5, 6, 7}
3
@13, dom = {0, 1, 2, 3, 4, 5, 6, 7}
@16, succs = {5},
@17, output(5, t) = {1, 5, 6, 7}, dom = {1, 5, 6, 7}
@18, dom = {1, 3, 5, 6, 7}
2
@13, dom = {0, 1, 2, 3, 4, 5, 6, 7}
@16, succs = {3, 4},
@17, output(3, t) = {1, 3, 5, 6, 7}, output(4, t) = {1, 4, 5, 6, 7}, dom = {1, 5, 6, 7}
@18, dom = {1, 2, 5, 6, 7}
0
@13, dom = {0, 1, 2, 3, 4, 5, 6, 7}
@16, succs = {1},
@17, output(1, t) = {1, 7}, dom = {1, 7}
@18, dom = {0, 1, 7}

Figure 5: Running example of applying the post-dominator analysis on an input graph containing branch and loop.

In the next two subsections, we describe how data-flow and loop sensitivity properties can be extracted from the analysis expressed using our extension. The traversal direction property need not be extracted as it is directly provided by the analysis writer.

3.7.1 *Computing Data-Flow Sensitivity.*

To determine the data-flow sensitivity property of a traversal, the operations performed in the traversal needs to be analyzed to check if the output of a node is computed using the outputs of other nodes. In our DSL extension, the only way to access the output of a node in a traversal is via output() expression, hence given a traversal t := traversal (n : Node) : T { tbody } as input, Algorithm 1 parses the statements in the traversal body tbody to identify method calls of the form output(n', t') that fetches the output of a node n' in the traversal t'. If such method calls exist, they are further investigated to determine if n' does not point to n and t' points to t. If this also holds, it means that the traversal output for the current node n is computed using the traversal outputs of other nodes (n') and hence the traversal is data-flow sensitive. For performing the points-to check, Algorithm 1 assumes that an alias environment is computed by using *must alias* analysis [15] which computes all names in the tbody that must alias each other at any program point. The must alias information ensures that Algorithm 1 never classifies a data-flow sensitive traversal as data-flow insensitive. The control and loop statements in the tbody do not have any impact on Algorithm 1 for computing the data-flow sensitivity property.

3.7.2 *Computing Loop Sensitivity.*

In general, computing the loop sensitivity property statically is challenging in the absence of an input graph, however the constructs and operations provided in our DSL extension enables static inference of this property.

A traversal is loop sensitive if the output of a node in any two successive iterations either shrinks or expands. We analyze the

[8] The traversal strategies chosen for initT and domT traversals are explained in §3.5.

Algorithm 1: Algorithm to detect data-flow sensitivity

```
Input: t := traversal (n : Node) : T { tbody }
Output: true/false
A ← getAliases(tbody, n);
foreach stmt ∈ tbody do
    if stmt = output(n ', t ') then
        if t ' == t and n ' ∉ A then
            return true;
return false;
```

operations performed in the traversal to determine if the traversal output expands or shrinks in the subsequent iterations. The operations add, addAll, and union always expand the output while remove, removeAll, and intersection always shrink the output.

Algorithm 2: Algorithm to detect loop sensitivity

```
   Input: t := traversal (n: Node)
       : T { tbody }
   Output: true/false
1  V ← {} // a set of output variables related to n;
2  V′ ← {} // a set of output variables not related to n;
3  expand ← false;
4  shrink ← false;
5  gen ← false;
6  kill ← false;
7  A ← getAliases(n);
8  foreach s ∈ tbody do
9      if s is v = output(n′, t′) then
10         if t′ == t then
11             if n′ ∈ A then
12                 V ← V ∪ v;
13             else
14                 V′ ← V′ ∪ v;
15 foreach s ∈ tbody do
16     if s = union(c1, c2) then
17         if (c1 ∈ V and c2 ∈ V′) || (c1 ∈ V′ and c2 ∈ V) then expand ← true ;
18     if s = intersection (c1, c2) then
19         if (c1 ∈ V and c2 ∈ V′) || (c1 ∈ V′ and c2 ∈ V) then shrink ← true ;
20     if s = add(c1, e) || s = addAll(c1, c2) then
21         if c1 ∈ V then
22             gen ← true;
23     if s = remove(c1, e) || s = removeAll(c1, c2) then
24         if c1 ∈ V then
25             kill ← true;
26 if (expand and gen) || (shrink and kill ) then return true;
27 else return false ;
```

Given a traversal t := traversal (n : Node) : T { tbody }, Algorithm 2 determines the loop sensitivity of t. Algorithm 2 investigates the statements in the tbody to determine if the traversal outputs of nodes in multiple iterations either expands or shrinks. For doing that, first it parses the statements to collect all output variables related and not related to input node n using the must alias information as in Algorithm 1. This is determined in lines 8-14, where all output variables are collected (output variables are variables that gets assigned by the output operation) and added to two sets V (a set of output variables related to n) and V' (a set of output variables not related to n). Upon collecting all output variables, Algorithm 2 makes another pass over all statements in the tbody to identify six kinds of operations: union, intersection , add, addAll, remove, and removeAll. In lines 16–17, the algorithm looks for union operation, where one of the variables involved is an output variables related to n and the other variable involved is not related to n. These conditions are simply the true conditions for the data-flow sensitivity, where the output of the current node is computed using the outputs of other nodes (neighbors). Similarly, in lines 18–19, the algorithm looks for intersection operation. The lines 20–25, identifies add and remove operations that adds or removes elements from the output related to node n. Finally, if there exists union and add operations, the output of a node always expands, and if there exists intersection and remove operations, the output of a node always shrinks. For a data-flow traversal to be loop sensitive, the output of nodes must either expand or shrink, not both (lines 26–27).

Table 1: List of source code analyses and properties of their involved traversals. Ts: total number of traversals. t_i: properties of the i-th traversal. *Flw*: data-flow sensitive. *Lp*: loop sensitive. *Dir*: traversal direction where —, → and ← mean iterative, forward and backward, respectively. ✓and ✗ for *Flw* and *Lp* indicates whether the property is *true* or *false.*

	Analysis	Ts	t_1			t_2			t_3		
			Flw	Lp	Dir	Flw	Lp	Dir	Flw	Lp	Dir
1	Copy propagation (CP)	3	✗	✗	—	✓	✓	→	✗	✗	—
2	Common sub-expression detection (CSD)	3	✗	✗	—	✓	✓	→	✗	✗	—
3	Dead code (DC)	3	✗	✗	—	✓	✓	←	✗	✗	—
4	Loop invariant code (LIC)	3	✗	✗	—	✓	✓	→	✗	✗	—
5	Upsafety analysis (USA)	3	✗	✗	—	✓	✓	→	✗	✗	—
6	Valid FileReader (VFR)	3	✗	✗	—	✓	✓	→	✗	✗	—
7	Mismatched wait/notify (MWN)	3	✗	✗	—	✓	✓	→	✗	✗	—
8	Available expression (AE)	2	✗	✗	—	✓	✓	→			
9	Dominator (DOM)	2	✗	✗	—	✓	✗	→			
10	Local may alias (LMA)	2	✗	✗	—	✓	✓	→			
11	Local must not alias (LMNA)	2	✗	✗	—	✓	✓	→			
12	Live variable (LV)	2	✗	✗	—	✓	✓	←			
13	Nullness analysis (NA)	2	✗	✗	—	✓	✓	→			
14	Post-dominator (PDOM)	2	✗	✗	—	✓	✗	←			
15	Reaching definition (RD)	2	✗	✗	—	✓	✓	→			
16	Resource status (RS)	2	✗	✗	—	✓	✓	→			
17	Very busy expression (VBE)	2	✗	✗	—	✓	✓	←			
18	Safe Synchronization (SS)	2	✗	✗	—	✓	✓	→			
19	Used and defined variable (UDV)	1	✗	✗	—						
20	Useless increment in return (UIR)	1	✗	✗	—						
21	Wait not in loop (WNIL)	1	✗	✗	—						

4 EMPIRICAL EVALUATION

We conducted an empirical evaluation on a set of 21 basic source code analyses on two public massive code datasets to evaluate several factors of BCFA. First, we show the benefit of using BCFA over standard strategies by evaluating the *reduction in running times* of BCFA over the standards ones (§4.2). Then, we evaluate the correctness of the analysis results using BCFA to show that the decision analyses and optimizations in BCFA do not affect the correctness of the source code analyses (§4.3). We also evaluate the precision of our selection algorithm by measuring how often BCFA selects the most time-efficient traversal (§4.4). We evaluate how the different components of BCFA and different kinds of static and runtime properties impact the overall performance in §4.5. Finally, we show practical uses of BCFA in three applications in §4.6.

4.1 Analyses, Datasets and Experiment Setting

4.1.1 Analyses. We collected source code analyses that traverse CFGs from textbooks and tools. We also ensured that the analyses list covers all the static properties discussed in §3.2, i.e., data-flow sensitivity, loop sensitivity and traversal direction (forward, backward and iterative). We ended up with 21 source code analyses as shown in Table 1. They include 10 basic ones (analyses 1, 2, 8, 9, 10,

Table 2: Statistics of the generated control flow graphs.

Dataset	All graphs	Sequential	Branches	Loops		
				All graphs	Branches	No branches
DaCapo	287K	186K (65%)	73K (25%)	28K (10%)	21K (7%)	7K (2%)
GitHub	161,523K	111,583K (69%)	33,324K (21%)	16,617K (10%)	11,674K (7%)	4,943K (3%)

Table 3: Time contribution of each phase (in miliseconds).

Analysis	Avg. Time		Static	Runtime			
	DaCapo	GitHub		DaCapo		GitHub	
				Avg.	Total	Avg.	Total
CP	0.21	0.008	53	0.21	62,469	0.008	1359K
CSD	0.19	0.012	60	0.19	56,840	0.012	1991K
DC	0.19	0.010	45	0.19	54,822	0.010	1663K
LIC	0.21	0.006	69	0.20	60,223	0.006	992K
USA	0.19	0.006	90	0.19	54,268	0.009	1444K
VFR	0.18	0.007	42	0.18	52,483	0.007	1142K
MWN	0.18	0.006	36	0.18	52,165	0.006	1109K
AE	0.18	0.007	43	0.18	53,290	0.007	1169K
DOM	0.21	0.008	35	0.21	62,416	0.008	1307K
LMA	0.18	0.008	76	0.18	52,483	0.008	1346K
LMNA	0.18	0.008	80	0.18	53,182	0.008	1407K
LV	0.17	0.007	32	0.17	49,231	0.007	1273K
NA	0.16	0.008	64	0.16	46,589	0.008	1398K
PDOM	0.20	0.012	34	0.20	57,203	0.012	2040K
RD	0.20	0.007	48	0.20	57,359	0.007	1155K
RS	0.16	0.006	28	0.16	46,367	0.006	996K
VBE	0.17	0.006	44	0.17	49,138	0.006	1062K
SS	0.17	0.006	32	0.17	48,990	0.006	1009K
UDV	0.14	0.005	10	0.14	41,617	0.005	928K
UIR	0.14	0.006	14	0.14	41,146	0.006	1020K
WNIL	0.14	0.007	15	0.14	41,808	0.007	1210K

11, 12, 14, 15 and 19) from textbooks [2, 22] and 11 others for detecting source code bugs, and code smells from the Soot framework [26] (analyses 3, 4, 5, 13, 17 and 18), and FindBugs tool [4] (analyses 6, 7, 16, 20 and 21). Table 1 also shows the number of traversals each analysis contains and their static properties as described in §3.2. All analyses are intra-procedural. We implemented all twenty one of these analysis in Boa using the constructs described in §3.7.[9]

4.1.2 Datasets. We ran the analyses on two datasets: DaCapo 9.12 benchmark [6], and a large-scale dataset containing projects from GitHub. DaCapo dataset contains the source code of 10 open source Java projects: Apache Batik, Apache FOP, Apache Aurora, Apache Tomcat, Jython, Xalan-Java, PMD, H2 database, Sunflow and Daytrader. GitHub dataset contains the source code of more than 380K Java projects collected from GitHub.com. Each method in the datasets was used to generate a control flow graph (CFG) on which the analyses would be run. The statistics of the two datasets are shown in Table 2. Both have similar distributions of CFGs over graph cyclicity (i.e., sequential, branch, and loop).

4.1.3 Setting. We compared BCFA against six standard traversal strategies in §3.4: DFS, PO, RPO, WPO, WRPO and ANY. The running time for each analysis is measured from the start to the end of the analysis. The running time for BCFA also includes the time for computing the static and runtime properties, making the traversal strategy decision, optimizing it and then using the optimized traversal strategy to traverse the CFG, and run the analysis. The analyses on DaCapo dataset were run on a single machine with 24 GB of RAM and 24 cores running Linux 3.5.6-1.fc17 kernel. Running analyses on GitHub dataset on a single machine would take weeks to finish, so we ran them on a cluster that runs a standard Hadoop 1.2.1 with 1 name and job tracker node, 10 compute nodes with totally 148 cores, and 1 GB of RAM for each map/reduce task.

4.2 Running Time and Time Reduction

We first report the running times and then study the reductions (or speedup) against standard traversal strategies.

4.2.1 Running Time. Table 3 shows the running times for 21 analyses on the two datasets. On average (column **Avg. Time**), each analysis took 0.14–0.21 ms and 0.005–0.012 ms to analyze a graph in Dacapo and GitHub datasets, respectively. The variation in the average analysis time is mainly due to the difference in the machines used to run the analysis for DaCapo and GitHub datasets. Also, the graphs in DaCapo are on average much larger compared to GitHub. Columns **Static** and **Runtime** show the time contributions for different components of BCFA: the time for determining the static properties of each analysis which is done once for each analysis, and the time for constructing the CFG of each method and traversing the CFG which is done once for every constructed CFG. We can see that the time for collecting static information is negligible, less than 0.2% for DaCapo dataset and less than 0.01% for GitHub dataset, when compared to the total runtime information collection time, as it is performed only once per traversal. When compared to the average runtime information collection time, the static time is quite significant. However, the overhead introduced by static information collection phase diminishes as the number of CFGs increases and becomes insignificant when running on those two large datasets. This result shows the benefit of BCFA when applying on large-scale analysis.

4.2.2 Time Reduction. To evaluate the efficiency in running time of BCFA over other strategies, we ran 21 analyses on DaCapo and GitHub datasets using BCFA and other strategies. When comparing the BCFA to a standard strategy S, we computed the reduction rate $R = (T_S - T_H)/T_S$ where T_S and T_H are the running times using the standard strategy and BCFA, respectively. Some analyses have worst case traversal strategies which might not be feasible to run on GitHub dataset with 162 million graphs. For example, using post-order for forward data-flow analysis will visit the CFGs in the direction which is opposite to the natural direction of the analysis and hence takes a long time to complete. For such combinations of analyses and traversal strategies, the map and the reduce tasks time out in the cluster setting and, thus, did not have the running times. The corresponding cells in Figure 6a are denoted with symbol –.

The result in Figure 6a shows that BCFA helps reduce the running times in almost all cases. The values indicates the reduction in running time by adopting BCFA compared against the standard strategies. Most of positive reductions are from 10% or even from 50% . More importantly, the most time-efficient and the worst traversal strategies vary across the analyses which supports the need of BCFA. Over all analyses, the reduction was highest against any order and post-order (PO and WPO) strategies. The reduction was lowest against the strategy using depth-first search (DFS) and worklist with reverse post-ordering (WRPO). When compared with the next best performing traversal strategy for each analysis, BCFA

[9] Our implementation infrastructure Boa currently supports only method-level analysis, however our technique should be applicable to inter-procedural CFGs.

Analysis	DaCapo						GitHub					
	DFS	PO	RPO	WPO	WRPO	ANY	DFS	PO	RPO	WPO	WRPO	ANY
CP	17%	83%	9%	66%	11%	72%	17%	88%	12%	80%	5%	82%
CSD	41%	93%	39%	74%	4%	89%	31%	–	24%	–	12%	–
DC	41%	30%	89%	7%	64%	81%	25%	22%	–	7%	–	–
LIC	17%	84%	8%	67%	7%	73%	19%	89%	15%	81%	19%	88%
USA	36%	92%	34%	72%	9%	87%	22%	–	17%	–	9%	–
VFR	20%	41%	18%	51%	15%	62%	15%	40%	10%	44%	9%	53%
MWN	21%	35%	16%	35%	22%	49%	17%	31%	12%	33%	11%	46%
AE	40%	14%	39%	73%	14%	87%	16%	–	16%	–	11%	–
DOM	54%	97%	48%	70%	6%	95%	27%	–	32%	–	6%	–
LMA	35%	46%	28%	74%	6%	46%	22%	–	13%	–	6%	–
LMNA	29%	39%	22%	68%	9%	41%	21%	–	15%	–	7%	–
LV	38%	30%	84%	11%	56%	75%	25%	21%	68%	11%	69%	80%
NA	26%	88%	30%	50%	10%	80%	13%	87%	12%	71%	10%	85%
PDOM	51%	41%	95%	10%	72%	95%	24%	20%	–	24%	–	–
RD	15%	80%	7%	62%	9%	68%	19%	91%	10%	79%	5%	86%
RS	31%	31%	30%	31%	28%	30%	16%	40%	9%	31%	7%	49%
VBE	40%	36%	88%	13%	76%	81%	28%	24%	–	10%	–	–
SS	26%	39%	22%	37%	25%	57%	20%	35%	13%	34%	10%	50%
UDV	6%	5%	6%	10%	9%	3%	3%	4%	2%	7%	6%	0%
UIR	2%	2%	1%	3%	3%	0%	2%	5%	4%	7%	7%	0%
WNIL	3%	4%	5%	6%	8%	2%	3%	6%	5%	5%	6%	0%
Overall	31%	83%	70%	55%	35%	81%	–	–	–	–	–	–

(a) Time reduction for each analysis.

Property	DaCapo					
	DFS	PO	RPO	WPO	WRPO	ANY
Data-flow	32%	84%	72%	57%	36%	83%
¬Data-flow	4%	4%	4%	6%	6%	2%

(b) Reduction over analysis properties.

Property	DaCapo					
	DFS	PO	RPO	WPO	WRPO	ANY
Sequential	20%	74%	63%	55%	28%	72%
Branch	31%	81%	66%	58%	40%	92%
Loop	53%	88%	75%	62%	37%	95%

(c) Reduction over graph properties.

Figure 6: Reduction in running times. Background colors indicate ranges of values: no reduction , (0%, 10%) , [10%, 50%) and [50%, 100%] .

reduces the overall execution time by about 13 minutes to 72 minutes on GitHub dataset. We do not report the overall numbers for GitHub dataset due to the presence of failed runs.

Figure 6b shows time reductions for different types of analyses. For *data-flow sensitive* ones, the reduction rates were high ranging from 32% to 84%. The running time was not improved much for *non data-flow sensitive* traversals, which correspond to the last three rows in Figure 6a with mostly one digit reductions). We actually perform almost as same as any-order traversal strategy for analyses in this category. This is because any-order traversal strategy is the best strategy for all the CFGs in these analyses. BCFA also chooses any-order traversal strategy and, thus, the performance is the same.

Figure 6c shows time reduction for different cyclicity types of input graphs. We can see that reductions over graphs with loops is highest and those over any graphs is lowest.

4.3 Correctness of Analysis Results

To evaluate the correctness of analysis results, we first chose worklist as standard strategy to run analyses on DaCapo dataset to create the groundtruth of the results. We then ran analyses using our hybrid approach and compared the results with the groundtruth. In all analyses on all input graphs from the dataset, the results from BCFA always exactly matched the corresponding ones in the groundtruth.

4.4 Traversal Strategy Selection Precision

In this experiment, we evaluated how well BCFA picks the most time-efficient strategy. We ran the 21 analyses on the DaCapo dataset using all the candidate traversals and the one selected by BCFA. One selection is counted for each pair of a traversal and an input graph where the BCFA selects a traversal strategy based on the properties of the analysis and input graph. A selection is considered correct if its running time is at least as good as the running time of the fastest among all candidates. The precision is computed as the ratio between the number of correct selections over the total number of all selections. The precision was 100% and 99.9% for *loop insensitive* and *loop sensitive* traversals, respectively.

Table 4: Traversal strategy prediction precision.

Analysis	Precision
DOM, PDOM, WNIL, UDV, UIR	100.00%
CP, CSD, DC, LIC, USA, VFR, MWN, AE, LMA, LMNA, LV, NA, RD, RS, VBE, SS	99.99%

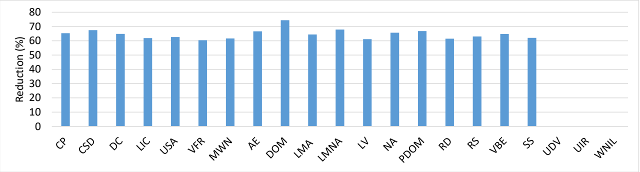

Figure 7: Time reduction due to traversal optimization.

As shown in Table 4, the selection precision is 100% for all analyses that are not *loop sensitive*. For analyses that involve *loop sensitive* traversals, the prediction precision is 99.99%. Further analysis revealed that the selection precision is 100% for sequential CFGs & CFGs with branches and no loop—BCFA always picks the most time-efficient traversal strategy. For CFGs with loops, the selection precision is 100% for *loop insensitive* traversals. The mispredictions occur with *loop sensitive* traversals on CFGs with loops. This is because for *loop sensitive* traversals, BCFA picks worklist as the best strategy. The worklist approach was picked because it visits only as many nodes as needed when compared to other traversal strategies which visit redundant nodes. However using worklist imposes an overhead of creating and maintaining a worklist containing all nodes in the CFG. This overhead is negligible for small CFGs. However, when running analyses on large CFGs, this overhead could become higher than the cost for visiting redundant nodes. Therefore, selecting worklist for *loop sensitive* traversals on large CFGs might not always result in the best running times.

4.5 Analysis on Traversal Optimization

We evaluated the importance of optimizing the chosen traversal strategy by comparing BCFA with the non-optimized version. Figure 7 shows the reduction rate on the running times for the 21 analyses. For analyses that involve at least one *data-flow sensitive* traversal, the optimization helps to reduce at least 60% of running time. This is because optimizations in such traversals reduce the number of iterations of traversals over the graphs by eliminating the redundant result re-computation traversals and the unnecessary fixpoint condition checking traversals. For analyses involving only *data-flow insensitive* traversal, there is no reduction in execution time, as BCFA does not attempt to optimize.

4.6 Case Studies

This section presents three applications adopted from prior works that showed significant benefit from BCFA approach. These applications includes one or more analyses listed in Table 1. We computed the reduction in the overall analysis time when compared to WRPO traversal strategy (the second best performing traversal after BCFA) and the results are shown in Figure 8.

Case	BCFA	WRPO	Reduce
APM	1527 min.	1702 min.	10%
AUM	883 min.	963 min.	8%
SVT	1417 min.	1501 min.	6%

Figure 8: Running time of the case studies on GitHub data.

API Precondition Mining (APM). This case study mines a large corpus of API usages to derive potential preconditions for API methods [21]. The key idea is that API preconditions would be checked frequently in a corpus with a large number of API usages, while project-specific conditions would be less frequent. This case study mined the preconditions for all methods of `java.lang.String`.

API Usage Mining (AUM). This case study analyzes API usage code and mines API usage patterns [29]. The mined patterns help developers understand and write API usages more effectively with less errors. Our analysis mined usage patterns for `java.util` APIs.

Finding Security Vulnerabilities with Tainted Object Propagation (SVT). This case study formulated a variety of widespread SQL injections, as tainted object propagation problems [18]. Our analysis looked for all SQL injection vulnerabilities matching the specifications in the statically analyzed code.

Figure 8 shows that BCFA helps reduce running times significantly by 80–175 minutes, which is from 6%–10% relatively. For understanding whether 10% reduction is really significant, considering the context is important. A save of 3 hours (10%) on a parallel infrastructure is significant. If the underlying parallel infrastructure is open/free/shared ([5, 9]), a 3 hour save enables supporting more concurrent users and analyses. If the infrastructure is a paid cluster (e.g., AWS), a 3 hour less computing time could translate to save of substantial dollar amount.

4.7 Threats to Validity

Our datasets do not contain a balanced distribution of different graph cyclicity.The majority of graphs in both DaCapo and GitHub datasets are sequential (65% and 69%, respectively) and only 10% have loops. The impact of this threat is that paths and decisions along sequential graphs are taken more often. This threat is not easy to mitigate, as it is not pratical to find a code dataset with a balanced distribution of graphs of various types. Nonetheless, our evaluation shows that the selection and optimization of the best traversal strategy for these 35% of the graphs (graphs with branches and loops) plays an important role in improving the overall performance of the analysis over a large dataset of graphs.

5 RELATED WORKS

Atkinson and Griswold [3] discuss several implementation techniques for improving the efficiency of data-flow analysis, namely: factoring data-flow sets, visitation order of the statements, selective reclamation of the data-flow sets. They discuss two commonly used traversal strategies: iterative search and worklist, and propose a new worklist algorithm that results in 20% fewer node visits. In their algorithm, a node is processed only if the data-flow information of any of its successors (or predecessors) has changed. Tok *et al.* [24] proposed a new worklist algorithm for accelerating inter-procedural flow-sensitive data-flow analysis. They generate inter-procedural def-use chains on-the-fly to be used in their worklist algorithm to re-analyze only parts that are affected by the changes in the flow values. Hind and Pioli [12] proposed an optimized priority-based worklist algorithm for pointer alias analysis, in which the nodes awaiting processing are placed on a worklist prioritized by the topological order of the CFG, such that nodes higher in the CFG are processed before nodes lower in the CFG. Bourdoncle [7] proposed the notion of weak topological ordering (WTO) of directed graphs and two iterative strategies based on WTO for computing the analysis solutions in dataflow and abstraction interpretation domains. Bourdoncle' technique is more suitable for cyclic graphs, however for acyclic graphs Bourdoncle proposes any topological ordering. Kildall [16] proposes combining several optimizing functions with flow analysis algorithms for solving global code optimization problems. For some classes of data-flow analysis problems, there exist techniques for efficient analysis. For example, demand interprocedural data-flow analysis [14] can produce precise results in polynomial time for inter-procedural, finite, distributive, subset problems (IFDS), constant propagation [27], etc. These works propose new traversal strategies for improving the efficiency of certain class of source code analysis, whereas BCFA is a novel technique for selecting the best traversal strategy from a list of candidate traversal strategies, based on the static properties of the analysis and the runtime characteristics of the input graph. Upadhyaya and Rajan [25] proposed Collective Program Analysis (CPA) that leverages similarities between CFGs to speedup analyzing millions of CFGs by only analyzing unique CFGs. CPA utilizes pre-defined traversal strategy to traverse CFGs, however our technique selects optimal traversal strategy and could be utilized in CPA.

Cobleigh *et al.* [8] study the effect of worklist algorithms in model checking. They identified four dimensions along which a worklist algorithm can be varied. Based on four dimensions, they evaluate 9 variations of worklist algorithm. They do not solve traversal strategy selection problem. Moreover, they do not take analysis properties into account. We consider both static properties of the analysis, such as data-flow sensitivity and loop sensitivity, and the cyclicity of the graph. Further, we also consider non-worklist based algorithms, such as post-order, reverse post-order, control flow order, any order, etc., as candidate strategies.

Several infrastructures exist today for performing ultra-large-scale analysis [5, 9, 11]. Boa [9] is a language and infrastructure for analyzing open source projects. Sourcerer [5] is an infrastructure for large-scale collection and analysis of open source code. GHTorrent [11] is a dataset and tool suite for analyzing GitHub projects. These frameworks currently support structural or abstract syntax tree (AST) level analysis and a parallel framework such as map-reduce is used to improve the performance of ultra-large-scale analysis. By selecting the best traversal strategy, BCFA could help improve their performance beyond parallelization.

There have been much works that targeted graph traversal optimization. Green-Marl [13] is a domain specific language for expressing graph analysis. It uses the high-level algorithmic description of the graph analysis to exploit the exposed data level parallelism. Green-Marl's optimization is similar to ours in utilizing the properties of the analysis description, however BCFA also utilizes the properties of the graphs. Moreover, Green-Marl's optimization is through parallelism while ours is by selecting the suitable traversal strategy. Pregel [19] is a map-reduce like framework that aims to bring distributed processing to graph algorithms. While Pregel's performance gain is through parallelism, BCFA achieves it by traversing the graph efficiently.

6 CONCLUSION

We proposed bespoke control flow analysis, a technique for optimizing source code analysis over control flow graphs (CFGs). Given the code of the analysis and a large set of CFGs, BCFA extracts a set of static properties of the analysis and combines it with a property of the CFG to automatically select an optimal CFG traversal strategy for every CFG. BCFA has minimal overhead, less than 0.2%; and leads to speedup between 1%-28%. BCFA has been integrated into Boa [9] and has already been utilized in [32].

REFERENCES

[1] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 25–34. https://doi.org/10.1145/1287624.1287630

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] Darren C. Atkinson and William G. Griswold. 2001. Implementation Techniques for Efficient Data-Flow Analysis of Large Programs. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01) (ICSM '01)*. IEEE Computer Society, Washington, DC, USA, 52–. https://doi.org/10.1109/ICSM.2001.972711

[4] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. 2007. Evaluating Static Analysis Defect Warnings on Production Software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*. ACM, New York, NY, USA, 1–8. http://doi.acm.org/10.1145/1251535.1251536

[5] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. 2014. Sourcerer: An Infrastructure for Large-scale Collection and Analysis of Open-source Code. *Sci. Comput. Program.* 79 (Jan. 2014), 241–259. https://doi.org/10.1016/j.scico.2012.04.008

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[7] François Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications*, Dines Bjørner, Manfred Broy, and Igor V. Pottosin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 128–141.

[8] Jamieson M. Cobleigh, Lori A. Clarke, and Leon J. Osterweil. 2001. The Right Algorithm at the Right Time: Comparing Data Flow Analysis Algorithms for Finite State Verification. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*. IEEE Computer Society, Washington, DC, USA, 37–46. http://dl.acm.org/citation.cfm?id=381473.381477

[9] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 422–431. http://dl.acm.org/citation.cfm?id=2486788.2486844

[10] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM, New York, NY, USA, 57–72. https://doi.org/10.1145/502034.502041

[11] Georgios Gousios. 2013. The GHTorent Dataset and Tool Suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132

[12] Michael Hind and Anthony Pioli. 1998. Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In *SAS*. Springer-Verlag, 57–81.

[13] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 349–362. https://doi.org/10.1145/2150976.2151013

[14] Susan Horwitz, Thomas Reps, and Mooly Sagiv. 1995. Demand Interprocedural Dataflow Analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '95)*. ACM, New York, NY, USA, 104–115. https://doi.org/10.1145/222124.222146

[15] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. 1998. Single and Loving It: Must-alias Analysis for Higher-order Languages. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*. ACM, New York, NY, USA, 329–341. https://doi.org/10.1145/268946.268973

[16] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*. ACM, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

[17] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15. 35.

[18] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM'05)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1251398.1251416

[19] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[20] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. 2012. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. *IEEE Trans. Softw. Eng.* 38, 5 (Sept. 2012), 1069–1087. https://doi.org/10.1109/TSE.2011.84

[21] Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. 2014. Mining Preconditions of APIs in Large-scale Code Corpus. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 166–177. https://doi.org/10.1145/2635868.2635924

[22] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis.* Springer Publishing Company, Incorporated.

[23] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Path-Sensitive Inference of Function Precedence Protocols. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 240–250. https://doi.org/10.1109/ICSE.2007.63

[24] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. 2006. Efficient Flow-sensitive Interprocedural Data-flow Analysis in the Presence of Pointers. In *Proceedings of the 15th International Conference on Compiler Construction (CC'06)*. Springer-Verlag, Berlin, Heidelberg, 17–31. https://doi.org/10.1007/11688839_3

[25] Ganesha Upadhyaya and Hridesh Rajan. 2018. Collective Program Analysis. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 620–631. https://doi.org/10.1145/3180155.3180252

[26] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008

[27] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 181–210. https://doi.org/10.1145/103135.103136

[28] Westley Weimer and George C. Necula. 2005. Mining Temporal Specifications for Error Detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*. Springer-Verlag, Berlin, Heidelberg, 461–476. https://doi.org/10.1007/978-3-540-31980-1_30

[29] Tao Xie and Jian Pei. 2006. MAPO: Mining API Usages from Open Source Repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*. ACM, New York, NY, USA, 54–57. https://doi.org/10.1145/1137983.1137997

[30] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 590–604. https://doi.org/10.1109/SP.2014.44

[31] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 282–291. https://doi.org/10.1145/1134285.1134325

[32] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are Code Examples on an Online Q&A Forum Reliable? A Study of API Misuse on Stack Overflow. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA.