Collective Program Analysis

Ganesha Upadhyaya and Hridesh Rajan

ABSTRACT

Popularity of data-driven software engineering has led to an increasing demand on the infrastructures to support efficient execution of tasks that require deeper source code analysis. While task optimization and parallelization are the adopted solutions, other research directions are less explored. We present *collective program analysis* (CPA), a technique for scaling large scale source code analysis by leveraging analysis specific similarity. Analysis specific similarity is about, whether two or more programs can be considered similar for a given analysis. The key idea of collective program analysis is to cluster programs based on analysis specific similarity, such that running the analysis on one candidate in each cluster is sufficient to produce the result for others. For determining the analysis specific similarity and for clustering analysis-equivalent programs, we use a sparse representation and a canonical labeling scheme. A sparse representation contains only the parts that are relevant for the analysis and the canonical labeling helps with finding isomorphic sparse representations. In a nutshell, two or more programs with same sparse representation must behave similarly for the given analysis. Our evaluation shows that for a variety of source code analysis tasks when run on a large dataset of programs, our technique is able to achieve substantial reduction in the analysis times; on average 69% when compared to baseline and on average 36% when compared to a prior technique. We also show that there exists a large amount of analysis-equivalent programs in large datasets for variety of analysis.

ACM Reference Format:
Ganesha Upadhyaya and Hridesh Rajan. 2017. Collective Program Analysis. In *Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, May 2018 (ICSE'18),* 11 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Data-driven software engineering technique has gained popularity in solving variety of software engineering (SE) problems such as defect prediction [11], bug fix suggestions [20, 21], programming pattern discovery [31, 37], and specification inference [1, 23]. The solutions to these SE problems may require deeper and expensive source code analysis, such as data-flow analysis. Parallelization and task optimizations are the two widely adopted techniques to scale source code analysis to large code bases [6, 13, 18].

In this work, we have explored an orthogonal research direction and we propose *collective program analysis* (CPA), a technique that leverages analysis specific similarity to scale source code analysis to large code bases. The key idea of collective program analysis is to cluster programs based on analysis specific similarity, such that running the analysis on one candidate in each cluster is sufficient to produce the result for others. For instance, if a user wants to run an analysis to check for null dereference bugs in million of programs, CPA will run the analysis on only the unique programs and reuse the results of the analysis on others.

The core of CPA are the concept of analysis specific similarity, the technique to abstract and represent programs to reveal analysis specific similarity, and the technique of storing and reusing the results of similar programs. Code clones are the popular way to identifying similar code [26]. Syntactic clones identify codes that are look alike, semantic clones identify codes that have similar control flow, and functional clones detect codes that have similar input/output behaviors. Syntactic and semantic clones can be used in CPA, however they will be less beneficial, because the amount of acceleration will be limited to the amount of copy-and-paste code exists in the large code bases in case of syntactic clones, and in case of semantic clones only a subset of analysis can benefit, for instance control flow analysis. Functional clones may not result in the same output, hence cannot be used for our purpose. For these reasons, we go beyond the existing notion of similarity and define analysis specific similarity. The analysis specific similarity is about, whether two or more programs can be considered similar for a given analysis. For a given analysis, programs can be considered similar if they follow the same set of instructions in the analysis. For instance, if an analysis is about counting the number of assert statements, irrespective of how different the two programs are, if they have same number of assert statements, they can be considered similar for the purpose of assert counting analysis. We show that for analysis expressed in the lattice-based data-flow framework, we can use the transfer functions to identify analysis specific similarity (or analysis equivalence).

Analysis equivalent programs may have statements that are irrelevant for the analysis. We use a sparse representation to remove the irrelevant statements that does not sacrifice the precision of the result [36]. Comparing the sparse representations to determine analysis equivalence becomes a graph isomorphism problem for data-flow analysis that have sparse control flow graphs. We use a canonical labeling scheme to make this comparison efficient [40]. For reusing the results between the analysis equivalent programs, we store the results in an efficient pattern database.

We evaluate our approach by measuring the reduction in the analysis time for 10 source code analysis tasks that involve data-flow analysis. We use two large dataset of programs. When compared to baseline CPA reduces the analysis time by 69% on average across all analyses and when compared to another technique, CPA reduces the analysis time by 36% on average. Further, we see a large amount of reuse opportunities in our large datasets for almost all analyses.

2 MOTIVATING EXAMPLE

Consider a *Resource Leak* analysis that identifies possible resource leaks in programs by tracking the resources that are acquired and released throughout the program by performing a flow analysis [35]. The analysis reports a problem in case of any acquired resource

ICSE'18, May 2018, Gothenburg, Sweden
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

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

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

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

Figure 1: Three methods extracted from `sunflow`, `fop`, and `xalan-j` DaCapo projects that have different resource usage patterns, however all of them acquire or explicitly release the resource in the sequential code.

is not released on every path in the program[1]. If a user wants to run this analysis on a large code base that contains millions of methods, he would end up running the analysis on each method in the code base. An optimization can be performed to skip analyzing methods that do not contain resource related statements, however the methods that have resource related statements needs to be analyzed.

To illustrate, consider the three methods `writeObj`, `main`, and `loadPropertyFile` shown in Figure 1, which are extracted from the three DaCapo projects `sunflow`, `fop`, and `xalan-j` respectively. These three methods use different resources in different ways, however for the resource leak analysis they all behave similar, because all of them acquire a resource and release along a single path leading to a resource leak (In event of exception, the resource is not closed). Though, the three methods were similar for resource leak analysis, all three of them needed to be analyzed to determine leak. If there existed a technique that could capture this similarity, it can perform the analysis on one of these three methods and simply return *true* for other two methods, indicating that a resource leak exists.

When analyzing a small number of methods or a handful of projects, there may not exist a lot of analysis specific similarity between the sourcode elements, such as methods, however in case of large code bases a large amount of analysis equivalent methods exists. For instance, the resource usage pattern leading to a leak shown in Figure 1 exists in 5151 methods in our `SourceForge` dataset. This means that, we only need to run the resource leak analysis on one method out of 5151 and reuse the result (in this case whether a leak exists or not) for the remaining 5150 methods. The `SourceForge` dataset contains a total of 82,900 methods that have resource related code out of 6,741,465 methods in the dataset. We were able to see 5689 unique patterns and the leak pattern discussed here appears in the top 3 patterns. Likewise, when analyzing large code bases, there exists a large amount of analysis equivalent codes and a large percentage of reuse opportunity to utilize for accelerating the overall analysis of large code bases.

3 `CPA`: COLLECTIVE PROGRAM ANALYSIS

Figure 2 provides a high-level overview of collective program analysis (`CPA`). Given a source code analysis that is needed to be run on a large code base of programs, we first produce the sparse representations of programs by performing a light-weight pre-analysis that identifies and removes parts of the program that are irrelevant for the analysis. We then generate a pattern for the sparse representation of the program. The pattern is then checked against a pattern database. In case the pattern does not exists in the database, the analysis is run on the sparse representation to produce the analysis result, whereas if the pattern already exists, then the stored result is extracted and returned as analysis output.

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

3.1 The Analysis Model

We assume that a source code analysis is formulated as a data-flow analysis problem using the lattice-based data-flow framework. In this framework, a data-flow analysis problem is described by defining a lattice which describes the set of values to be associated with program statements, and a set of transfer functions that describes how each program statement transforms the input values to output values[2]. Two sets of data-flow values are maintained at each node: `IN` and `OUT` that describes the input and output values at each node. Finally, the data-flow analysis solves a set of flow equations involving the two sets `IN` and `OUT`, and transfer functions. Based on the data-flow values computed at the nodes, assertions can be made about the program behavior. For instance, the *Resource Leak* analysis described in the motivation section maintains a set of variables

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

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

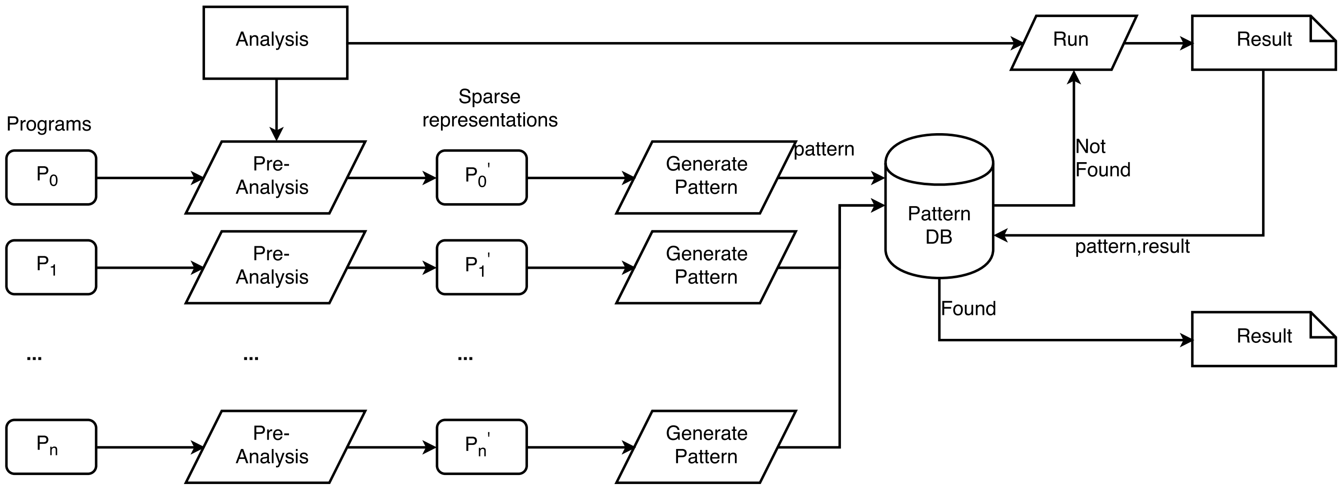


Figure 2: Collective Program Analysis overview

representing the resources as data-flow values and it has mainly three kinds of transfer functions for handling resource `acquire`, resource `release`, and resource `copy/aliasing`[3].

From hereon, whenever we refer to analysis, we mean the data-flow analysis expressed in this framework. The data-flow analysis is performed on the control flow graph representation of the program.

Definition 3.1. A **Control Flow Graph** of a program is a directed graph $CFG = (N, E, n_0, n_e)$, with a set of nodes N representing program statements and a set of edges E representing the control flow between statements. n_0 and n_e denote the entry and exit nodes of the CFG[4].

3.2 Sparse Representation

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

Definition 3.2. A program statement is relevant for an analysis, if there exists a non-identity transfer function for that statement in the analysis. That is, if the analysis has defined a transfer function f_i^k for statement i, where k represents the transfer function kind and $f^k \neq \iota$, then i is relevant for the analysis. In the data-flow analysis model there always exists an identity transfer function ι along with the user defined transfer functions to represent those statement that have no effect on the analysis.

Definition 3.3. Given a program P with a set of statements S, a sparse representation is a tuple, $< P', M >$, where P' contains a subset of the statements $S' \subseteq S$, such that $\forall i \in S'$, i is a relevant statement. $M : S \rightarrow f^k$ is a map that provides the information about the kind of the transfer function applicable to each relevant statement i in set S'.

As our analysis are data-flow analysis and programs are represented as control flow graphs, we have to generate sparse representations of CFGs. For this purpose we utilize a prior work that proposes reduced control flow graphs (RCFGs) [36]. In a nutshell, a RCFG is a reduced CFG that contains only those nodes for which there exits a non-identity transfer function in the analysis. RCFG is constructed using a pre-analysis that takes an analysis specification and a CFG as input, extracts all the conditions for the analysis transfer functions and checks the conditions against CFG nodes to identify analysis relevant nodes. We extended RCFG to also store the kind of the transfer function that are applicable to CFG nodes as special properties of nodes. This information is required in later stages of CPA.

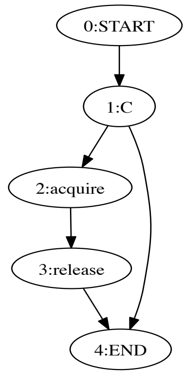


Figure 3: Sparse representation of `writeObj` method shown in Figure 1.

To provide a concrete example of a sparse representation, let us revisit the *Resource Leak* analysis described in the motivation and

[3]We ignore the method calls for simplifying the description, however in our implementation the method calls are over-approximated.
[4]A CFG may contain multiple exit points however we connect them to a auxiliary exit node.

the `writeObj` method. The *Resource Leak* analysis contains three kinds of transfer functions: `acquire`, `release`, and `copy`. Using the transfer functions, we can identify the relevant statements in the `writeObj` method. The relevant statements for this methods are at line 3 and line 7, because line 3 has an associated transfer function `acquire` and line 7 has `release`. All other statements do not have an associated transfer function and hence are considered irrelevant and removed except some special nodes such as `START` and `END`. RCFG also retains the branch nodes that have at least one successor with a relevant statement. The resulting sparse representation is as shown in Figure 3. The graph is a RCFG of the CFG of the method `writeObj`. It contains two nodes 3 and 7 that have associated transfer functions `acquire` and `release` respectively and a special branch node marked `C`.

3.3 Analysis Equivalence

Given the sparse representations of programs, our next problem is to find similarities between them. In case of sparse representations of CFGs, finding similarities is a graph isomorphism problem. A prior work gspan [40] has proposed using a Depth-first search (DFS) code as the unique canonical label for graphs to find isomorphism. We utilize the DFS code technique for obtaining the canonical form of the sparse representation.

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

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

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

In the DFS code that we generate, we use only 4-tuple and ignore the edge label $l_{(i,j)}$, because it is only required for multi-edge graphs and CFGs are not multi-edge graphs. For node labels we use the transfer function kinds. For instance, for the the *Resource Leak* analysis, we use `acquire`, `release` and `copy` for node labels. Note that, every node in the sparse representation of the CFG has an associated non-identity transfer function. Figure 4 shows the DFS code constructed for the sparse graph of `writeObj` method shown in Figure 1. As shown in the figure each edge is represented as a 4-tuple. For instance, edge from node 2 to node 3 is represented as (2,3,aquire,release). By following the the $\prec_T$ order, we obtained the DFS code shown in the figure.

An undirected graph could have several DFS codes (based on the starting node) and the minimum DFS code provides the canonical label, such that if two graphs G and G' that have the same minimum DFS codes are isomorphic to each other [40].

Theorem 3.4. *Given two graphs G and G', G is isomorphic to G' iff, their minimum DFS codes are equal.*

Proof. *The proof is based on [40].*

Theorem 3.5. *A CFG always has a single DFS code that is minimum because there exists a single start node and the edges are directed.*

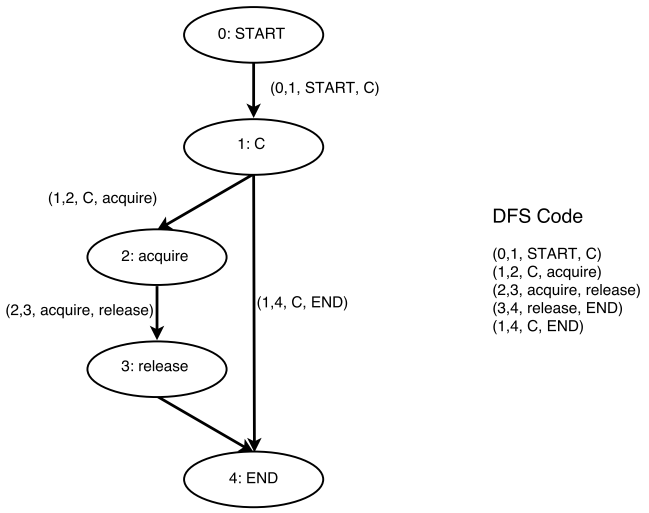


Figure 4: DFS code for the sparse control flow graph of the `writeObj` method shown in Figure 1.

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

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

Definition 3.6. **(Analysis Equivalence)** Two CFGs G_1 and G_2 are equivalent for a given analysis or *analysis equivalent* if the DFS codes of the corresponding sparse graphs are same.

To provide a concreate example, consider the *Resource Leak* analysis and the three methods `writeObj`, `main`, and `loadPropertyFile` shown in Figure 1. Although the CFGs of these three methods are different, after removing all the irrelevant nodes the sparse graph obtained is as shown in Figure 3. For this sparse graph the DFS code constructed is shown in Figure 4. As these three methods have the same DFS code, their sparse representations are analyais equivalent.

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

Theorem 3.7. *Two analysis equivalent sparse representations produces similar results.*

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

3.4 Leveraging Analysis Specific Similarity

Now that we have a technique to identify analysis specific similarity between programs, the final step of CPA is to cluster programs and reuse the analysis results between programs. We use a pattern database to store the DFS code of the sparse representations as key and analysis result as value. As described in our overview diagram shown in Figure 2, after producing the DFS code for the sparse representations, our approach first checks whether a result is already present in the database. We define the presence of the DFS code as a *hit* and the absense as *miss*. In case of a hit, we simply return the stored result. In case of a miss, we run the analysis on the sparse representations to produce the result and store the result along with the DFS code into the database for future reuse.

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

4 EVALUATION

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

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

4.1 Performance

4.1.1 Methodology. We compare our approach against a baseline that runs the analysis on all programs without any optimization or reuse. We also compare against a prior work [36] that identifies and removes irrelevant statements prior to analyzing programs. We measure the analysis time for all three approaches (CFG, RCFG, and CPA) and compare them to compute the percentage reduction in the analysis time of CPA over CFG (denoted as R) and the percentage reduction in the analysis time of CPA over RCFG (denoted as R'). The analysis times were averaged over the last three runs, when the variability across these measurements is minimal (under 2%) by following the methodology proposed by Georges *et al.* [16]. Our experiments were run on a machine with 24 GB of memory and 24-cores, running on Linux 3.5.6-1.fc17 kernel.

Table 1: Analyses used in our evaluation.

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

4.1.2 Analyses. We have used 10 source code analysis to evaluate our approach as listed in Table 1. Below we briefly describe them.

`Avail.` [24] Available expression analysis tries to find optimization opportunities in the source code, such as value of a binop expression computed once can be reused in the later program points, if the variables in the expression are not re-defined. This is a standard compiler optimization drawn from the textbook. We included this analysis to represent how optimization problems can benefit from CPA. The analysis will report if there exists one or more optimization opportunities.

`Dom.` [3] Control flow dominators are useful in many analysis that requires control dependence, for instance in computing the program dependence graph (PDG), however computing the dominators is expensive, hence we included this in our list of analysis to demonstrate how our technique can accelerate computing dominators. This is also a special kind of analysis where all nodes are relevant for the analysis and the sparse representation constitutes the whole CFG. The analysis will report a map containing list of dominators for each CFG node.

`Escape.` [39] Escape analysis computes whether the objects allocated inside methods stay within the method or escapes to outside world. This information is useful to stack allocate all the captured objects instead of heap. The analysis outputs *true* if there exists any captured objects, otherwise *false*.

`Leak.` [35] This is a resource leak checker that captures the resource usage in the programs to identify possible leaks. The analysis tracks all 106 JDK resource related API usages in programs. If any resource acquired is not released at the exit node, it outputs that leak may exist.

`Live.` [24] This analysis tracks the liveness of local variables used in the program. There exists many client applications of this analysis such as identifying and removing dead code, register allocation, etc. This analysis simply reports all the variable definition and use sites along with their control flow dependencies.

MayAlias. [28] Precisely computing the alias information is expensive and sometimes may not be possible. In such situations, computing the may alias information by following the direct assignments can be handy. This may alias analysis computes the alias information and reports the alias sets.

Null. [9] This analysis checks if there exists a dereference that is post-dominated by a null check. Such a pattern indicates that the dereference may cause null pointer exception, because it can be null. The analysis reports if there exists such problems in the program.

Pointer. [29] Pointer or points-to analysis implemented here is a flow-sensitive and context-insensitive points-to analysis. It computes the points-to graph. A points-to graph provides the information whether the variables in the program may point to the same object. This analysis outputs the points-to graph with abstract variables and nodes (meaning concrete variable names are mapped to symbols).

Safe. [35] The safe synchronization checker looks for the lock acquire/release patterns to identify bugs. Acquiring locks and not releasing them may cause deadlock and starvation in the program. The analysis tracks all the variables on which the lock is acquired and checks if the locks on these variables is released on every program path. If not, it reports that the problem exists.

Taint. [15] Taint analysis detects and reports possibly vulnerabilities by performing a taint analysis. The analysis identifies the variables that read data from external inputs like console, tracks their dataflow, and checks if the data from these variables are written to output.

4.1.3 Datasets.

We have used two datasets for evaluating this work. The first dataset consists of all projects included in the DaCapo benchmark [8], a well-established benchmark of Java programs. This dataset contains 45,054 classes and 286,888 non-empty methods. The DaCapo dataset is prepared using the GitHub project links of the 10 DaCapo projects. The second dataset consists of 4,938 open source SourceForge projects. This dataset consists of 191,945 classes, and 6,741,465 non-empty method.

4.1.4 Results and Analysis.

Table 2 compares our approach (CPA) against the baseline (CFG) that directly runs the analysis on CFGs and a prior work (RCFG) that removes irrelevant nodes of the CFGs prior to running the analysis. Note that, the reported analysis time in case of CFG is the actual analysis time, whereas, in case of RCFG, it includes the two overheads (to identify and to remove the irrelevant nodes) and in case of CPA, it includes several overheads (to produce the sparse graph, to generate pattern, to check the pattern database, and retrieve the result in case of hit, and to persist the results in case of miss). Analysis times are reported in milliseconds and for some analysis the times are low, for instance Safe. This is mainly because we have optimized all the analysis to skip through the irrelevant methods (methods that do not contain the information the analysis is looking for). Finding and skipping through the irrelevant methods is done at a low cost.

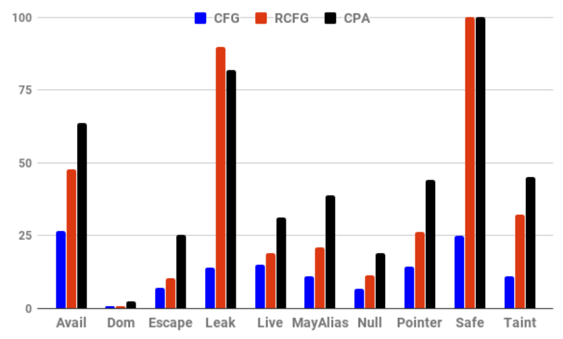


Figure 5: % benefit of the upper bound achieved by CFG, RCFG, CPA. Higher bars are better.

Table 2 shows our results. The two columns R and R' shows the percentage reduction in the analysis time of CPA when compared to CFG and RCFG respectively. On average CPA is able to reduce the analysis time by 69% against CFG and 36% against RCFG (averaged over both DaCapo and SourceForge datasets, which are individually 34% and 39%). Note that, for Dom the CFG and RCFG times are exactly same, because for this analysis all nodes are relevant, hence RCFG is simply CFG. For Leak analysis on DaCapo dataset, CPA shows negative gain when compared to RCFG. This happens mainly because of the low number of instances on which the analysis is run. As shown in Table 4, under DaCapo, the Leak analysis is run on only 220 unique methods and the cost of CPA overheads may exceed the benefit, hence we do not expect CPA to improve the performance. Similar situation can be also be seen for Safe analysis.

As CPA adopts an online strategy of persisting and retrieving the cached results during the process of analyzing thousands of methods, the order in which projects or methods in the dataset are visited may influence the gain, hence we compute the ideal gain (or an upper-bound) by re-running the experiments on the same dataset after caching the results in the first run. The analysis times are reported in Table 2 under CPA-UB column. Figure 5 helps to understand how far CFG, RCFG, and CPA are from ideal speedup that we can achieve (CPA-UB). As it can be seen in Figure 5, for most analysis, CPA is the closest to 100%, when compared to others (CFG and RCFG), except for Leak and Safe. The reason is as explained earlier, the numebr of methods on which the analysis is run is small, hence the overheads of CPA exceeds its benefits. Another interesting observation that can be made is that except for Leak and Safe, for all other analysis, there exists substantial opportunities to improve the performance of CPA to get it closer to CPA-UB. This can be performed by training CPA by running the analysis using training projects, caching the results and then runing the analysis on test projects.

4.1.5 CPA Components.

For every control flow graph of the method to be analyzed, our approach (CPA) first produces a sparse representation of the graph, generates a pattern that represents the sparse graph and checks the pattern database for a result. The overhead for this stage is represented as pattern in Table 3. When there is a miss, i.e., the result does not exists for the pattern, then

Table 2: Reduction in the analysis time of CPA.

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

Table 3: CPA time distribution across four components. The absolute times are in milliseconds and the values inside "()" are the contribution of the component towards CPA time.

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

CPA runs the analysis to produce a result (analysis stage) and cache the result for (persist stage). When there is a hit, i.e., a result is found, nothing else to be done. It is interesting to see how the overall CPA time is distributed across these components. The component results are shown in Table 3, where pattern, analysis, and persist are the three components. The absolute times are in milliseconds and we also show contributions of each of the three components towards the CPA time (numbers inside paranthesis).

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

4.1.6 Reuse Opportunity. Table 2 shows that our approach was able to substantially reduce the analysis time across 10 analysis and two datasets. The reduction mainly stems from the result reuse across similar graphs. We measured the total number of unique graphs to compute the reuse percentage. The results are shown in Table 4. The very high percentage of reuse clearly suggests why our approach was able to achieve substantial reduction in the analysis time. Further interesting results such as the top patterns in terms of reuse are discussed next.

Table 4: Amount of reuse opportunity available in various analysis.

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

Next, we list the transfer functions for all our 10 analysis in Table 5. The names of these transfer functions provides information

Table 5: Transfer functions to identify relevant nodes.

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

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

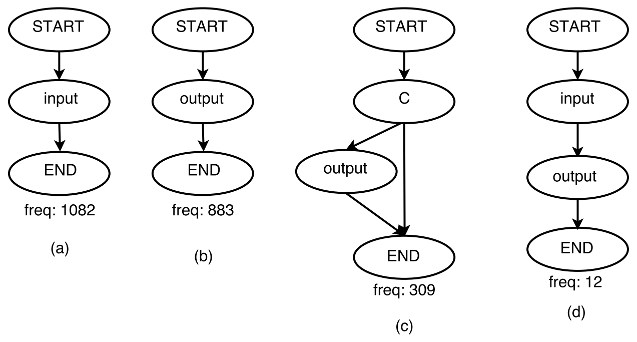
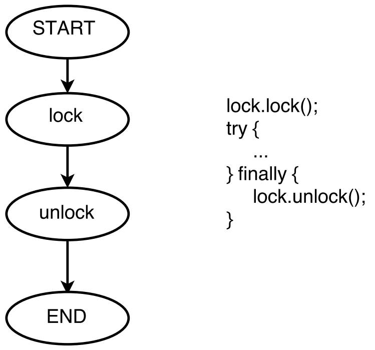


Figure 6: Top patterns seen in case of taint analysis that detects vulnerabilities

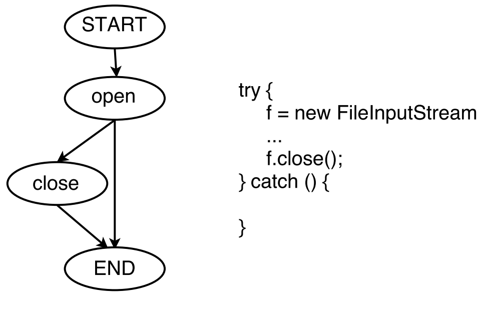
In case of `Taint` analysis, we had a total of 6169 methods in the `DaCapo` dataset that were analyzed (other methods didn't had relevant code) and they formed 1208 unique sparse graphs. The analysis reported possibility of vulnerabilities for 101 sparse graphs. Figure 6 shows the top 3 patterns along with their frequencies ((a), (b), and (c)). Our analysis did not report any vulnerabilities for any methods that have the sparse graphs shown in (a), (b), (c), because all these three sparse graphs only have either `input` or `output` nodes. For vulnerability to occur, both must exists. Consider (d) which has both `input` and `output` was one of the frequent vulnerability pattern in the `DaCapo` dataset. We manually verified 20 out of 101 reported instances for the existence of possible vulnerabilities.

In case of `Safe` analysis that checks for correct use of lock/unlock primitives using JDK concurrent libraries, we had 76 instances reported correct and 1 reported as having problem. Out of the 76, 50 of them followed a single pattern that is shown in Figure 7. This is a correct usage pattern for lock/unlock. The 1 instance that was reported problematic was a false positive and it requires inter-procedural analysis to eliminate it.



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

Figure 7: Most frequent lock/unlock pattern and the code example of the pattern.



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

}
```

Figure 8: Most frequent buggy resource leak pattern.

`Leak` analysis results were most surprising for us. There existed 3087 usages of JDK resource related APIs (JDK has 106 resource related APIs) and our analysis reported 2277 possible leaks. Out of these 336 were definitely leaks and others were possible leaks and confirming them would require inter-procedural analysis. Out of the 336 the top pattern appeared in 32 methods. Figure 8 shows this pattern.

4.2 Correctness and Precision

In §3.3 we provided a proof sketch as to why the analysis results of CPA must match with that of CFG. To empirically evaluate the correctness of the results, we conducted two experiments using all 10 analysis and DaCapo dataset. In the first experiment, we compared the analysis result of CFG and CPA for every method that is analyzed. Table 6 provides information about the results computed for each analysis. We were able to match the two results perfectly.

Table 6: Analysis results computed for various analysis.

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

As for most of the analysis in Table 6 the computed results are just boolean, we double-check the results by profiling the transfer functions executed for CFG and the sparse graph of CPA and compare the sequence of transfer functions. We skip through the identity transfer functions in case of CFG, as the CFG may contain many irrelevant nodes. As the order of nodes visited in both CFG and sparse graph of CPA are same, we were able to see a 100% match.

4.3 Limitations

In this work we have applied CPA to accelerate analysis at method-level, where results for each method is computed separately without using the results at the method call sites. Instead of applying the results of methods at their call sites we have adopted an over-approximation strategy. As such, there should not be any limitation for using our technique in a compositional whole-program analysis setting, where the results of the called methods are used if available. This design choice was mainly due to the framework used in our evaluation, which does not support whole-program analysis.

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

5 RELATED WORKS

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

5.1 Improving the efficiency of source code analysis

There exists a trade off between improving the efficiency of the analysis and improving the accuracy of the analysis results. Removing the unimportant parts of the code before analyzing it has been a popular choice [4, 10, 27, 34, 36]. For instance, Upadhyaya and Hridesh [36] proposed RCFG a reduced control flow graph that contains only statements that are related to analysis. Our work on CPA has adopted the RCFG work to produce the sparse graph. Our work on CPA uses RCFG as sparse representation to cluster similar graphs and reuse the results to further improve the efficiency of analysis. There also exists other sparse representations such as sparse evaluation graph (SEG) [10] that are more suitable for def-use style data-flow analysis. There exists works that eliminates unnecessary computations in the traversal of the program statements to improve the efficiency of analysis [5, 34]. These techniques remove the unnecessary iterations to improve the efficiency, whereas our work removes the unnecessary statements to produce sparse graphs and also reuses the results by clustering sparse graphs.

Allen *et al.* [4] and Smaragdakis *et al.* [27] have proposed a pre-analysis stage prior to actual analysis to scale points-to analysis to large code bases. They perform static analysis and program compaction to remove statements that do not contributes to the points-to results. Their work is specialized for scaling points-to analysis, whereas CPA is more general in that, it can accelerate analysis that use data-flow analysis and expressed using the lattice framework.

Program slicing is a fundamental technique to produce a compilable and runnable program that contains statements of interest specified by a slicing criteria [2, 7, 32, 38]. Many slicing techniques have been proposed [33]. Our pruning technique is similar to slicing in that we also remove the irrelevant statements. Our pruning technique is meant as a pre-processing step rather than a transformation. Our pruning does not produce a compilable and runnable code like slicing.

Reusing the analysis results is another way to improve the efficiency [17, 19, 25]. Kulkarni *et al.* [19] proposed a technique to accelerate program analysis in Datalog. The idea of the technique run an offline analysis on a corpus of training programs to learn analysis facts over shared code and then reuses the learnt facts to accelerate the analysis of other programs that share code with the training corpus. Inter-procedural analysis are often accelerated by reusing the analysis results in the form of partial [17] and complete [25] procedure summaries. Our technique does not require that programs share code, it only requires that programs executed same set of analysis instructions to produce results.

5.2 Finding software clones

Our technique is also related to code clones [26] because CPA also clusters sparse representations to reuse the analysis results. There are various types of clones. Syntactic clones are look alike code fragments, whereas semantic clones share common expressions and have similar control flows. Functional clones are more similar in terms of the input and output they operate on. There are also other approaches that goes beyond structural similarity: code fingerprints[22], behavioral clones [14, 30], and run-time behavioral similarity [12]. Code clones techniques are agnostic to the analysis, however CPA clusters program based on the a sparse representation that is derived from the analysis specification.

6 CONCLUSION AND FUTURE WORK

In this work we proposed *collective program analysis* (CPA), a technique to accelerate large scale source code analysis by leveraging analysis specific similarity between programs. The key idea of CPA is to cluster programs that are similar for the purpose of the analysis, such that it is sufficient to run the analysis on one program from each cluster to produce result for others. To find analysis specific similarity between programs, a sparse representation and a canonical labeling scheme was used. The technique is applied to source code analysis problems that requires data-flow analysis. When compared to the state-of-the-art, where the analysis is directly performed on the CFGs, CPA was able to reduce the analysis time by 69%. When compared to an optimization technique that removes irrelevant parts of the program before running the analysis on them, CPA was able to reduce the analysis time by 37%. Both of these results were consistent across two datasets that contained several hundred thousand methods to over 7 million methods. The sparse representation used in CPA was able to create a high percentage of reuse opportunity (more than 90%). An immediate future work is to extend CPA to whole-program analysis. There could be many challenges, especially to set up the environment in the large scale source code analysis frameworks, where callgraphs for thousands of projects needs to be build prior to applying the CPA. Another avenue for future work extend CPA to support variety of analysis results.

ACKNOWLEDGMENTS

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] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 246–256. https://doi.org/10.1145/93542.93576

[3] Frances E. Allen. 1970. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*. ACM, New York, NY, USA, 1–19. https://doi.org/10.1145/800028.808479

[4] Nicholas Allen, Bernhard Scholz, and Padmanabhan Krishnan. 2015. Staged Points-to Analysis for Large Code Bases. In *CC*. Springer Berlin Heidelberg, 131–150.

[5] 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

[6] 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 (2014), 241–259.

[7] David W. Binkley and Keith Brian Gallagher. 1996. Program Slicing. *Advances in Computers* 43 (1996), 1 – 50.

[8] 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 *OOPSLA*. ACM, 169–190.

[9] Fraser Brown, Andres Nötzli, and Dawson Engler. 2016. How to Build Static Checking Systems Using Orders of Magnitude Less Code. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 143–157. https://doi.org/10.1145/2872362.2872364

[10] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. 1991. Automatic Construction of Sparse Data Flow Evaluation Graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '91)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/99583.99594

[11] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2011. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Softw. Engg.* (2011).

[12] John Demme and Simha Sethumadhavan. 2012. Approximate Graph Clustering for Program Characterization. *ACM Trans. Archit. Code Optim.* (2012), 21:1–21:21.

[13] 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

[14] Rochelle Elva and Gary T. Leavens. 2012. Semantic Clone Detection Using Method IOE-behavior. In *IWSC'12*. IEEE Press, 80–81.

[15] Ansgar Fehnker, Ralf Huuck, and Wolf Rödiger. 2011. Model Checking Dataflow for Malicious Input. In *Proceedings of the Workshop on Embedded Systems Security (WESS '11)*. ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/2072274.2072278

[16] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

[17] Patrice Godefroid, Aditya V Nori, Sriram K Rajamani, and Sai Deep Tetali. 2010. Compositional may-must program analysis: unleashing the power of alternation. In *ACM Sigplan Notices*, Vol. 45. ACM, 43–56.

[18] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *MSR '13*. IEEE Press, 233–236.

[19] Sulekha Kulkarni, Ravi Mangal, Xin Zhang, and Mayur Naik. 2016. Accelerating Program Analyses by Cross-program Training. In *OOPSLA'16*. ACM, 359–377.

[20] Zhenmin Li, Shan Lu, and Suvda Myagmar. 2006. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Softw. Eng.* 32, 3 (2006), 176–192.

[21] Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes* 30, 5 (2005), 296–305.

[22] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. 2012. Detecting Similar Software Applications. In *ICSE'12*. IEEE Press, 364–374.

[23] Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. 2014. Mining Preconditions of APIs in Large-scale Code Corpus. In *FSE'14*. ACM, 166–177.

[24] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[25] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*. ACM, 49–61.

[26] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Sci. Comput. Program.* (2009), 470–495.

[27] Yannis Smaragdakis, George Balatsouras, and George Kastrinis. 2013. Set-based Pre-processing for Points-to Analysis. In *OOPSLA*. ACM, 253–270.

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

[29] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Aliasing in Object-Oriented Programming. Springer-Verlag, Berlin, Heidelberg, Chapter Alias Analysis for Object-oriented Programs, 196–232. http://dl.acm.org/citation.cfm?id=2554511.2554523

[30] F. H. Su, J. Bell, and G. Kaiser. 2016. Challenges in Behavioral Code Clone Detection. In *SANER'16*, Vol. 3. 21–22.

[31] Suresh Thummalapenta and Tao Xie. 2009. Alattin: Mining Alternative Patterns for Detecting Neglected Conditions. In *ASE'09*. IEEE CS, 283–294.

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

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

[34] Teck Bok Tok. 2007. *Removing Unimportant Computations in Interprocedural Program Analysis*. Ph.D. Dissertation. Austin, TX, USA. Advisor(s) Lin, Calvin. AAI3290942.

[35] Emina Torlak and Satish Chandra. 2010. Effective Interprocedural Resource Leak Detection. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 535–544. https://doi.org/10.1145/1806799.1806876

[36] Ganesha Upadhyaya and Hridesh Rajan. 2017. *On Accelerating Source Code Analysis At Massive Scale*. Technical Report. Iowa State University.

[37] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *ESEC/FSE'07*. ACM, 35–44.

[38] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449. http://dl.acm.org/citation.cfm?id=800078.802557

[39] John Whaley and Martin Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*. ACM, New York, NY, USA, 187–206. https://doi.org/10.1145/320384.320400

[40] Xifeng Yan and Jiawei Han. 2002. gSpan: Graph-Based Substructure Pattern Mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM '02)*. IEEE Computer Society, Washington, DC, USA, 721–. http://dl.acm.org/citation.cfm?id=844380.844811