University of California

Santa Barbara

Program Similarity

Techniques and Applications

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy

in

Computer Science

by

Lawton Higgins Nichols

Committee in charge:

Professor Ben Hardekopf, Chair

Professor Tevfik Bultan

Professor Tim Sherwood

June 2020

The Dissertation of Lawton Higgins Nichols is approved.

Professor Tevfik Bultan

Professor Tim Sherwood

Professor Ben Hardekopf, Committee Chair

May 2020

Program Similarity Techniques and Applications

To Grandpa—
thanks for letting me play on your computer

Acknowledgements

This thesis was made possible in part by:

My family Thank you all for putting up with me while I was constantly stressed for six years—especially you, mom. Your support means a lot to me.

Ben Thank you so much for teaching me how to do research, and for being patient while I made million mistakes. I would not be in this position without your guidance.

Kyle Dewey You are the best mentor I could ask for—you have helped me navigate both research issues as well as life issues, and a more genuine human being has never walked the earth. Thank you for your help and for your kindness.

Tim and Tevfik You provided me with invaluable advice over the past six years. Thank you both for helping me figure out exactly what I was doing, since I am no good at motivating my work.

Diba Mirza and Kate Kharitonova When I got to UCSB I did not know whether I wanted to go into teaching, but now there is no doubt in my mind. Diba, thank you for your mentorship when I was just starting out with teaching my very first class, as well as throughout my graduate teaching career—my lectures are only good because I started with yours. Kate, thank you so much for your guidance and support while I was beginning to look for jobs, as well as for your perspective on everything. I had no idea what I was doing until I talked with you.

The PL Lab Thank you all for allowing me to coexist in your presence. A few shout-outs:

- **Mehmet Emre**, you are next-level. I swear that all our co-authored papers should have you as the first author instead. It has been an honor having my desk next to yours. I wish you the best in your professional and dancing careers, good sir.
- **Michael Christensen**, you work so much harder than me and wear the coolest puffer jackets. You actually have your life in order and have been a great role model.
- **Mika Gavrilov**, you made me cornbread and gave me your Magic: The Gathering cards. You are the personification of everything that is good in this world—thank you, my friend.
- **Joseph McMahan**, thank you for the tennis and the singing and the friendship! You helped make my life here suck less.
- **Vineeth Kashyap**, thank you for taking me under your wing as an intern while I was an early grad student—your suggestions during my time at GrammaTech helped shape my thesis.

The Smittcamp Family Honors College Without the support from the SFHC at Fresno State, as well as from **Dr. Honora Chapman**, I can guarantee that I would not be here. Thank you for providing such a wonderful environment in which to grow. And thank you to all of the friends I had there for making my time as an undergraduate magical.

UCSB Cotillion Dance To all my Cotillion friends, thank you for giving me a place to belong—life here was unbearably lonely until I found you. I want to thank my partners in particular for their friendship, and for putting up with me: **Lucia**, **Sam**, **Emily**, and **Xochitl**—you all rock.

and readers like you :)

Curriculum Vitæ
Lawton Higgins Nichols

Education

- 2014 – Present: University of California, Santa Barbara
  Ph.D. in Computer Science, expected June 2020.
  - M.S. awarded June 2019.
  - Certificate in College and University Teaching.
- 2010 – 2014: California State University, Fresno
  B.S. in Computer Science, summa cum laude. Minor in Mathematics.
  - Member of the Smittcamp Family Honors College.
  - Recipient of the President's Scholarship.

Experience

Research Assistant, UCSB Programming Languages Lab	Summer 2014 – Present
Instructor, UCSB CS 24 (Problem Solving with Computers II)	Fall 2019
Instructor, UCSB CS 16 (Problem Solving with Computers I)	Summer 2017, F18, S19, Summer 2019
Instructor, UCSB CS 8 (Introduction to Computer Science)	Summer 2018
Research Intern, GrammaTech, Ithaca, NY	Summer 2016
Teaching Assistant, UCSB CS 16 (Problem Solving with Computers I)	Winter 2019
Teaching Assistant, UCSB CS 162 (Programming Lang.)	Winter 2015, Spring 2015
Teaching Assistant, UCSB CS 160 (Compilers)	Fall 2014
Research Assistant, CSU Fresno	Fall 2013 – Spring 2014
Software Engineering Intern, General Atomics, San Diego, CA	Summer 2013
Student Assistant, CSU Fresno CS 40 (Intro. to CS)	Spring 2013

Publications

Kyle Dewey, Lawton Nichols, Ben Hardekopf. *Automated Data Structure Generation: Refuting Common Wisdom*. In *International Conference on Software Engineering (ICSE)*, 2015.

Joseph McMahan, Michael Christensen, Lawton Nichols, Jared Roesch, Sung-Yee Guo, Ben Hardekopf, Tim Sherwood. *An Architecture Supporting Formal and Compositional Binary Analysis*. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

Joseph McMahan, Michael Christensen, Lawton Nichols, Jared Roesch, Sung-Yee Guo, Ben Hardekopf, Tim Sherwood. *An Architecture for Analysis*. In *IEEE Micro*, May 2018.

Lawton Nichols, Mehmet Emre, Ben Hardekopf. *Structural and Nominal Cross-Language Clone Detection*. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2019. `https://doi.org/10.1007/978-3-030-16722-6_14`

Lawton Nichols, Kyle Dewey, Mehmet Emre, Sitao Chen, Ben Hardekopf. *Syntax-based Improvements to Plagiarism Detectors and their Evaluations*. In *Innovation and Technology in Computer Science Education (ITiCSE)*, 2019. `https://doi.org/10.1145/3304221.3319789`

Lawton Nichols, Mehmet Emre, Ben Hardekopf. *Fixpoint Reuse for Incremental JavaScript Analysis*. In *State Of the Art in Program Analysis (SOAP)*, 2019. `https://doi.org/10.1145/3315568.3329964`

Honors & Awards

- Was a member of the Smittcamp Family Honors College at CSU Fresno. Each student selected for the Honors College receives a full-ride President's Scholarship.
- UCSB Outstanding Teaching Assistant award, 2019.

Abstract

Program Similarity Techniques and Applications

by

Lawton Higgins Nichols

The world is full of programs. More are written every day, and so the corpus of written code is ever-increasing. The entropy of all this code, however, is not increasing as fast as one might expect. Many programs are identical or very similar to others, and this is due to many possible reasons: for example, software engineers reusing code or students solving the same programming assignment. Similar code also occurs naturally when programs are updated—there the old and new versions are close relatives of each other, and this can be exploited. Discovering and exploiting similarity is useful in areas as disparate as program analysis and automated student feedback.

Program similarity is not a solved problem, and my work advances the field of program similarity research along two axes: methods and applications. This work has involved the development of new similarity methods as well as the application of those methods to solve problems in a new or more effective way.

Contents

Curriculum Vitae **viii**

Abstract **x**

1 Introduction **1**
1.1 Thesis Statement 2
1.2 Source Code 2
1.3 Grant Support 2

2 Cross-Language Clone Detection **3**
2.1 Introduction 3
2.2 Background and Related Work 6
2.3 Overview 10
2.4 Structural Clone Detection 11
2.5 Hybrid Algorithm 16
2.6 Evaluation 17
2.7 The Full Algorithm 25
2.8 End-to-End Example 26

3 Plagiarism Detection **31**
3.1 Introduction 31
3.2 Background and Related Work 33
3.3 Our Method 36
3.4 Complete Example 42
3.5 Evaluation 47

4 Fixpoint Reuse **52**
4.1 Introduction 52
4.2 Related Work 54
4.3 Fixpoint Reuse 56
4.4 Fixpoint Reuse for SAFE 60
4.5 Evaluation 66
4.6 Background 79

5 Semantic Clone Detection **85**
5.1 Introduction 85
5.2 Overview and Example 89
5.3 Our Method in Detail 97
5.4 Evaluation 103
5.5 Related Work 113

6 Student Feedback Using Invariant Inference **118**
6.1 Introduction 118
6.2 Method Overview 120
6.3 Method in Detail 128
6.4 Evaluation 134
6.5 Related Work 143

7 Conclusions and Future Work **149**

I am just a poor boy, though my story's seldom told
I have squandered my resistance for a ballroom full of mumbles,
Such are conferences

Chapter 1

Introduction

The world is full of programs. More are written every day, and so the corpus of written code is ever-increasing. The entropy of the code, however, is not increasing as fast as one might expect. Many pieces of code are identical or very similar to others, and this is due to reuse such as copy-and-paste or code plagiarism. In most cases similar code should be brought to someone's attention, and research on program similarity seeks to help with these issues.

Similar code also occurs naturally when programs are updated—there the old and new versions are close relatives of each other, and similarities like this can be exploited; for example, incremental analysis allows for vast performance improvements on the assumption that program versions are not vastly different.

Program similarity is not a solved problem, and my work advances the field of program similarity research along two axes: methods and applications. My work has involved making new similarity methods as well as applying those methods to solve problems in a new or more effective way. The end goal of my work culminates in the

following thesis statement:

1.1 Thesis Statement

Using insights from programming languages and program analysis research, syntactic and semantic similarity methods can be enhanced and exploited in new ways. These techniques can be applied to areas as disparate as abstract interpretation and computer science education.

In the remainder of this dissertation I argue this thesis with concrete work. The remaining chapters are either techniques (Cross-Language Clone Detection, Semantic Clone Detection) or applications (Plagiarism Detection, Fixpoint Reuse, Student Feedback Using Invariant Inference) of program similarity.

1.2 Source Code

After the corresponding papers are published, the implementations of all tools and methods mentioned in this thesis (with the exception of the Plagiarism Detection work) will be located at `http://www.cs.ucsb.edu/~pllab` under the "Downloads" link. For the Plagiarism Detection work, we have placed the implementation on GitHub at `https://github.com/lawtonnichols/plagiarism-detector`.

1.3 Grant Support

The projects described in Chapter 2, Chapter 3, and Chapter 4 were supported by NSF CCF-1319060.

Chapter 2

Cross-Language Clone Detection

2.1 Introduction

The clone detection problem has long been recognized by the community, with many existing papers exploring different techniques for finding clones amongst code written in a single language [35, 121, 123, 75, 69]. However, in recent years an interesting twist has arisen due to the rising popularity of cross-language libraries and applications: *cross-language clones*. Consider the parser generator ANTLR [2], which has runtimes that are written in C#, C++, Go, Java, JavaScript, Python (2 and 3), and Swift. Also consider multi-platform mobile applications, which are often ported between Java and Objective-C or Swift, the languages used by Android and iPhone applications. In these kinds of settings, clones can actually cross language boundaries: a fragment of code in one language can be copied and massaged to conform to the syntax and semantics of another language. Existing single-language clone detection techniques are unable to effectively detect these sorts of cross-language clones. In this chapter we propose a method to detect cross-language clones and demonstrate that it (1) finds cross-language clones that no existing method can detect; and (2) performs comparably to existing single-language clone detectors for finding clones within a corpus of single-language

```
Trees._findAllNodes = function(t, index, findTokens, nodes) {
        // check this node (the root) first
        if(findTokens && (t instanceof TerminalNode)) {
                if(t.symbol.type===index) {
                        nodes.push(t);
                }
        } else if(!findTokens && (t instanceof ParserRuleContext)) {
                if(t.ruleIndex===index) {
                        nodes.push(t);
                }
        }
        // check children
        for(var i=0;i<t.getChildCount();i++) {
                Trees._findAllNodes(t.getChild(i), index, findTokens, nodes);
        }
};
```

```
template<typename T>
static void _findAllNodes(ParseTree *t, size_t index, bool findTokens, std::vector<T> &nodes) {
  // check this node (the root) first
  if (findTokens && is<TerminalNode *>(t)) {
    TerminalNode *tnode = dynamic_cast<TerminalNode *>(t);
    if (tnode->getSymbol()->getType() == index) {
      nodes.push_back(t);
    }
  } else if (!findTokens && is<ParserRuleContext *>(t)) {
    ParserRuleContext *ctx = dynamic_cast<ParserRuleContext *>(t);
    if (ctx->getRuleIndex() == index) {
      nodes.push_back(t);
    }
  }
  // check children
  for (size_t i = 0; i < t->children.size(); i++) {
    _findAllNodes(t->children[i], index, findTokens, nodes);
  }
}
```

Figure 2.1: A JavaScript (top) and C++ (bottom) clone pair doing a pre-order search.

```
VerletParticle2D.prototype.setWeight = function(w){
  this.weight = w;
  this.invWeight =
    (w !== 0) ? 1 / w : 0; //avoid divide by zero
};
```

```
public void setWeight(float w) {
  weight = w;
  invWeight = 1f / w;
}
```

Figure 2.2: A JavaScript (left) and Java (right) clone pair setting the weight and inverse weight of a particle in a graphics application. A bug-fix has been applied to the JavaScript clone but not the Java clone.

code sources. Therefore, our technique generalizes the current state of the art in clone detection by extending it to allow for both single-language and cross-language clone detection using a single technique.

To make this problem more concrete, consider Figure 2.1, which shows a real-life case (found during our evaluation described in Section 2.6) of code clones involving C++ and JavaScript source code from the ANTLR parser generator [2]. To demonstrate the importance of finding cross-language clones, consider Figure 2.2, which shows another real-life case (also found during our evaluation) of code clones involving JavaScript and Java in which a bug-fix has been applied to one of the clones but not the other. In addition, a quick search of the CVE (Common Vulnerabilities and Exposures) database yields a vulnerability due to incorrect message authentication checking that exists in multiple different language implementations of the relevant code [6].

There are only four existing papers that we are aware of that introduce new techniques for cross-language clone detection (discussed in more detail in Section 2.2). That initial work has either focused on clones across languages that share a common intermediate representation such as .NET [86, 22] or has deviated from classical clone detection and taken a more restricted, natural language-based approach, sometimes relying on assumptions that may not be met in real code [42, 41]. None of that existing work would detect the clone examples given in Figures 2.1 and 2.2 without extensive modification.

The main reason for these restrictions in previous work is that the *syntactic structure* (i.e., parse trees) of different languages can be extremely different even for code that, at the source level, seems similar. We demonstrate this phenomenon later in this chapter. In order to overcome this problem, previous work has either restricted itself to languages with a common intermediate representation (thus enforcing that the syntactic structure is similar for similar code) or abandoned structural matching entirely and looked only at the names of variables and other user-defined abstractions (what we call *nominal* clone detection). We observe that using purely structural or purely nominal matching is sub-optimal in a cross-language setting, in that each can yield both false positives and false negatives.

Our technique consists of (1) a method for enabling structural matching for cross-language clones even in those cases where syntactic structure is different (Section 2.4); and (2) a method for composing both structural and nominal matching into a singular matcher, maintaining the strengths of each while mitigating their individual weaknesses (Section 2.5). We have implemented our technique in a tool called FETT. that works at the granularity of function pairs; we use FETT to empirically compare our proposed technique against existing techniques (Section 2.6). We begin by describing related work and background information in Section 2.2 and giving a high-level

overview of our technique in Section 2.3.

2.2 Background and Related Work

The concept of clone detection is not new, and the different techniques involved have been surveyed extensively [35, 121]. Most existing non-semantics-based techniques can be categorized into the classes of "structural," "nominal," or "hybrid," which we define below.

Before we begin, there is a bit of misleading terminology in the literature: there exist many clone detection tools that are considered language-generic or language-agnostic (e.g., [123]), but can only be configured to work for programs written in a single language at a time. CCFinder [75], for example, can detect clones for six different programming languages; however, the user cannot (outside of naive text-only modes) truly cross language boundaries during a "language-generic" clone detection phase.

2.2.1 What Exactly Is a Cross-Language Clone?

Intuitively, we consider a cross-language clone to be the same as any same-language clone—two pieces of code that implement similar functionality—the only difference is the setting. We highlight here what kinds of clones our tool is able to find, and what kinds of clones we include in our evaluation based on their classification (i.e., Type I, II, III or IV [125]).

The usual code clone hierarchy does not translate well to a cross-language setting: type I and type II clones [125] may not exist across languages because of syntactic differences between languages (e.g., switch statements exist in C but not in Python). In this chapter, we present methods that discover syntactic clones modulo the differences

in language syntax, and we do this by creating a correspondence between related but different constructs. We do not consider semantic (type IV) clones that implement the same functionality in a different way (e.g., quicksort vs. selection sort). Readers familiar with the standard clone hierarchy can think of the clones that we find as type III clones generalized across languages.

2.2.2 Structural Program Similarity

Intuitively, two programs (or subprograms) can be considered similar if they look the same, disregarding identifier names—i.e., if their syntax trees have roughly the same shape. We refer to structural clone detection as the process of taking advantage of this similarity.

Same-language clone detection tools usually also consider identifier data, and we are not aware of any purely structural cross-language clone detector. A notable same-language tool that operates via structural similarity is Deckard, which converts syntax trees into vectors for fast comparison [69].

Structural similarity is useful in all settings, but it is a hard problem in a multi-language setting—all the hybrid structural/nominal methods we describe below make some restriction on the languages involved. A major part of the novelty of our technique is a method for purely structural matching across languages (though the final algorithm then combines structural with nominal (i.e., identifier-based) techniques for greater accuracy).

2.2.3 Nominal Program Similarity

Whereas structural similarity disregards identifiers and instead looks at code shape, nominal similarity does the exact opposite. Nominal similarity relies on the insight

that similar code, especially copied and pasted snippets, will have the same identifier names throughout, regardless of code structure.

Notable same-language clone detection tools that operate via nominal similarity are CCFinder and SourcererCC, which compare program tokens [75, 129].

Across Languages. Cheng et al. describe CLCMiner [42], the first cross-language clone detection tool that does not require the languages involved to translate to the same intermediate form. It compares revision histories (diffs) in repository logs for cross-platform C# and Java programs; the tokens inside commits are used to compute similarity scores. CLCMiner is the basis for the Nominal algorithm defined in Section 2.5.1.

Cheng et al. study a different notion of nominal similarity in [41], where they measure the effectiveness of token distributions in finding clones among cross-platform mobile applications; they obtain a negative result for identifier names alone. Flores et al. [59] use natural language processing techniques to discover cross language clones at the function level.

2.2.4 Hybrid Program Similarity

It is logical to combine structural and nominal similarity methods, as the results they provide are complementary. A notable same-language, hybrid clone detection tool is NiCad, which performs its comparisons at the parse tree level [126]. Syntax tree-based comparison is quite common [34, 149].

Tree similarity is computationally expensive [36], and it is more efficient to linearize programs in some way; sequence similarity algorithms can then do the comparison. Existing same-language work compares the tokens in the order in which they appear in the parse tree [65], and we also take advantage of linearization of full parse trees in this work.

Across Languages. Kraft et al. present C2D2[86], the first cross-language clone detection tool, for C# and Visual Basic programs. This work requires that the languages involved be compiled to the same intermediate representation (IR)—.NET IR in this case. From a graph derived from that IR, they create sequences of tokens for subgraphs and use a Levenshtein distance-based token similarity algorithm to compare them.

Al-Omari et al. build on Kraft et al.'s work and find clones by comparing CIL intermediate code text [22]. Again, they are restricted to .NET languages.

This work. Our method is a hybrid method, works on any language with a grammar definition, and relies on just the source code (in contrast to, e.g., CLCMiner which requires the existence of revision history). We linearize preprocessed parse trees at the function level and compare the linearized sequences in a novel way that generalizes Kraft et al.'s work and incorporates features of Cheng et al.'s work.

2.2.5 CLCMiner

Our main comparison is with the only tool designed for cross-language clone detection and capable of handling arbitrary languages: CLCMiner [42]. We provide further background on it here. CLCMiner is based on having the source code in a version control system, and requires a revision history by design. Section 2.5.1 gives a detailed explanation of our adaptation of CLCMiner. The original CLCMiner algorithm works on diffs and lexes them, whereas our version works on function parse trees.

We were not able to obtain access to the original CLCMiner source code from the authors. In order to compare against this method, we implement our own version which adapts CLCMiner to work with the entire text of a function and have it calculate the distance metric above when given a function pair. Our new implementation may perform better or worse than the original (which uses revision history rather than

function pairs) in certain cases.

We incorporate CLCMiner's distance metric in a novel way in FETT, and show that our combination of structural and nominal information produces better results. As we have adapted CLCMiner's algorithm to work on functions instead of diffs, it relies on having a parser to extract the functions and does not rely on a version control system. We refer to our nominal-only adaptation of CLCMiner's algorithm as "Nominal" for the rest of the chapter.

2.3 Overview

In this section we provide a high-level overview of FETT and provide justification for some of our steps. We give an end-to-end example of our clone detection process in Section 2.8. FETT's pipeline is:

1. Take as input a corpus of source code (which may exist in multiple languages);
2. Using existing ANTLR grammars, parse and create a separate parse tree for each function (we currently handle C++, Java, and JavaScript);
3. Simplify parse trees that have an unnecessarily large depth;
4. Abstract the multilingual parse trees into a common representation to facilitate comparison;
5. Linearize the resulting trees using a preorder traversal;
6. Compare all linearized function pairs using a Smith-Waterman local sequence alignment algorithm; and finally
7. Present the pairwise similarity scores to the user.

The following sections fill in the details of the structural and nominal aspects of FETT's cross-language clone detection process.

2.4 Structural Clone Detection

One key insight of our structural algorithm is that *abstract* syntax trees (ASTs), which eliminate details in the concrete parse trees about how exactly the input was parsed or what language it came from, tend to look more similar for similar code even across languages. Unfortunately, ASTs are not part of a language's specification, and AST grammars and formats are implementation dependent. We are not aware of any single compiler that has frontends for the variety of languages that we compare. Our structural clone detection algorithm processes *reduced parse trees* (Section 2.4.1) to eliminate nonessential details about parsing and obtain a structure similar to ASTs.

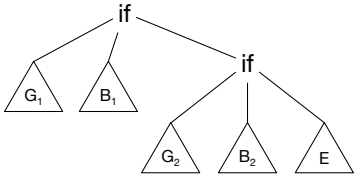
Another source of disparity between trees generated by two grammars is that the nonterminals are different. The other key insight of our structural algorithm is that abstracting reduced parse trees by putting nonterminals in *equivalence classes* (Section 2.4.2) strikes a balance between preserving necessary information and smoothing out differences across languages.

Our structural algorithm proceeds by extracting functions from an abstracted parse tree and then computes similarity scores between functions using the Smith-Waterman local sequence alignment algorithm.

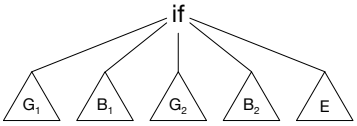
Flattening a tree using a preorder traversal helps smooth out most remaining inconsistencies between inter-language reduced parse trees. To demonstrate the dissimilarities due to grammatical differences that preorder traversal removes, see Figure 2.3: a grammar that uses nested `if` statements will have a parse tree like Figure 2.3b, while a grammar that uses unnested `if` statements will look more like Figure 2.3c. As the

`if` `(` *exp* `)` *block* [`else` *block*] (G1)
`if` *exp* `:` *block* [`elif` *exp* `:` *block*]* [`else` *block*] (G2)

(a) Two different kinds of grammars for if statements.



(b) An example parse tree using the nested `if` grammar (G1).



(c) An example parse tree using the unnested `if` grammar (G2).

Figure 2.3: Grammars and parse trees for nested vs. unnested `if` statements.

`else if` cases become more numerous in the first grammar the nesting becomes more severe, emphasizing the differences in the resulting parse trees.

2.4.1 Precedence woes

Some grammar definitions encode operator precedence into the grammar[1], whereas others use facilities provided by the parser generators to encode the precedence. Direct encoding of precedence causes spurious chains of nonterminals in the resulting parse tree, which would be removed when the parse tree is converted to an AST. We collapse the chains of nonterminals encountered in a parse tree for the direct encoding case to remove the chains and mitigate this disparity between different styles of grammars. Figure 2.4 demonstrates the kinds of issues that are apparent when a grammar hard-codes precedence—because precedence in this case appears in the form of nested productions, we always see "AdditiveExpression" even when there is only a multiplication expression present; this will throw off any clone detector that is working directly on plain parse trees.

If precedence is handled indirectly through the parser generator, then the resulting

[1]We encountered this only in the C++ grammar during our evaluation.

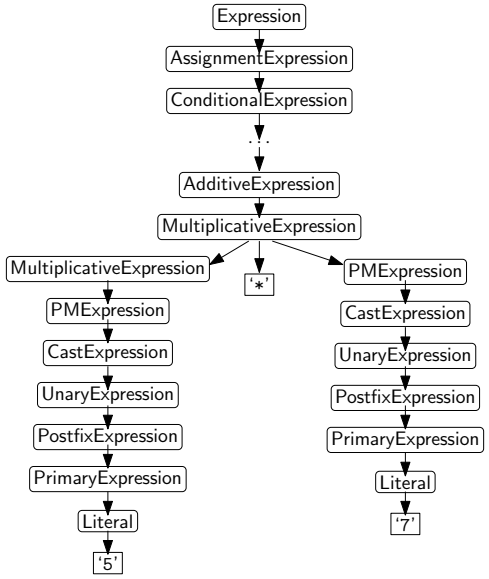


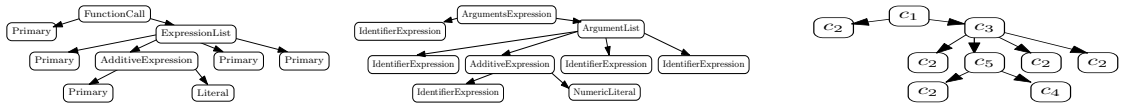
Figure 2.4: A subtree of the original C++ parse tree for the text "`5*7`".

parse tree is much closer to an AST. This is an example of an issue that only arises in a cross-language setting, and which makes cross-language clone detection strictly more difficult than same-language clone detection. We condense any chains of nonterminals, and we refer to the parse trees after this stage as *reduced parse trees*.

2.4.2 Abstracting Parse Tree Nonterminals

Consider the two reduced parse trees for the expression `binarySearch(array, mid+1, high, x)` in Figures 2.5a and 2.5b. Although they look similar to the naked eye, because the node names are different, even a tree edit distance algorithm would say that the trees are not similar at all. We thus need to abstract the nonterminal names while preserving essential information about the tree structure. After performing this abstraction, we call the resulting parse trees *abstracted parse trees*.

Our method instead groups node types with similar meanings across languages, so that node types that "mean" similar things are in the same group. To do this, we



(a) Reduced parse tree from a Java parser. (b) Reduced parse tree from a JavaScript parser. (c) Abstraction of the trees in Figures 2.5a and 2.5b.

Figure 2.5: Reduced parse trees for expression `binarySearch(array, mid+1, high, x)` in Java and JavaScript, and their abstraction. The terminals are omitted for simplicity.

manually categorize node types into equivalence classes *once per pair of languages*. For example, consider the equivalence classes c_1 = {FunctionCall, ArgumentsExpression}, c_2 = {Primary, IdentifierExpression}, c_3 = {ArgumentList, ExpressionList}, c_4 = {NumericLiteral, Literal}, c_5 = {AdditiveExpression} and the set $C = \{c_1, c_2, c_3, c_4, c_5\}$. After replacing each node in Figures 2.5a and 2.5b with its equivalence class in C, we end up with trees that are exactly the same (Figure 2.5c). In this specific example the abstracted trees are the same, though this is not always the case in practice.

We define the abstraction algorithm in two parts: $\mathsf{EqClassMapOf}(C)$ produces a map from each node to a symbol corresponding to its equivalence class. $\mathsf{Abstract}(tree, map)$ does the abstraction by traversing the given tree bottom up and applying the map. It removes the *nonterminals* which do not belong to any equivalence class. When the abstraction algorithm removes a node, it connects any children of the removed node to the removed node's parent.

2.4.3 Sequence Alignment for Clone Detection

Linearizing the trees via a preorder traversal of the nodes will remove most traces of the structural differences demonstrated in Figure 2.3. Moreover, the state of the art tree edit distance algorithms are not as scalable as sequence alignment algorithms[2].

[2]APTED, the state of the art tree edit distance algorithm has a time complexity of $O(n^3)$ [113] whereas the variant of Smith-Waterman algorithm we use is $O(n^2)$ [24].

These observations led us to explore sequence alignment algorithms as an alternative to tree-edit distance. Levenshtein distance is a popular choice in this category. Smith-Waterman is strictly more general than Levenshtein distance, and it supports assigning weights to different elements in the sequence. Hence, we use the Smith-Waterman algorithm on preordered trees to compute similarity scores. We evaluate the precision and recall of both Smith-Waterman and tree edit distance in Section 2.6 and observe that sequence alignment performs better in terms of precision and scalability.

We convert function subtrees to sequences by computing the preorder traversal. Finally, we execute Smith-Waterman using custom weights on each sequence pair and normalize the resulting score using the normalization factor Z described below. We chose the weights based on the hypothesis that certain nodes like conditionals indicate important program structure, and should generally appear in the same order in a cloned pair of functions; therefore, we assign higher weights to penalize the function pairs in which this alignment does not occur. In the algorithm, the function $\mathsf{SmithWaterman}(a, b, M, g)$ computes a similarity score between two sequences a and b using the Smith-Waterman algorithm with substitution matrix M and linear gap penalty coefficient g; a detailed explanation of these parameters can be found in [24].

Normalizing Smith-Waterman results. The result of the Smith-Waterman algorithm depends on the size of the input, and longer sequence pairs have higher scores. In order to find both short and long clones, we normalize the resulting similarity score from the Smith-Waterman algorithm to neutralize the bias towards longer clones.

We define the *self-similarity score* of a sequence a as the score assigned to the pair (a, a) by the *unnormalized* Smith-Waterman algorithm; denote this score $\mathsf{S}(a)$. We normalize score assigned to a pair (a, b) by $\frac{1}{Z}$ where $Z = \max\{\mathsf{S}(a), \mathsf{S}(b)\}$. Note that Z is an upper bound for the score obtained by Smith-Waterman, and the score is equal to

Z if and only if $a = b$. Thus, using the normalization factor $\frac{1}{Z}$ is useful if one is looking for similar whole functions rather than looking for a small snippet in a larger piece of code.

2.5 Hybrid Algorithm

Combining nominal and structural clone detection in a cross-language setting provides the best of both worlds, and mitigates any issues that running just one detection method might have.

Identifier names carry some meaning about the programmer intent and give a code snippet context. On the other hand, structure of code (conditionals, loops, function calls etc.) also carry information about programmer intent. Without this structural information, we might misidentify two pieces of code as clones. Our hybrid algorithm is guided by structural information while consulting the Nominal algorithm to use local context within structurally similar pieces of code.

2.5.1 Our Nominal Algorithm

We have adapted CLCMiner's algorithm to work on functions as our purely Nominal algorithm. For a given pair of functions (f_1, f_2), our nominal matching algorithm consists of two parts.

The first part takes a function f, removes the comments and splits the tokens on each non-letter character (such as underscores or dashes). It then splits the camel case tokens into words and converts them to lowercase—each function becomes a bag of words that is represented by a characteristic vector, which holds the number of occurrences of each word. We denote the resulting characteristic vector as $\mathsf{v}(f)$.

The second part of the algorithm computes a normalized distance between the two characteristic vectors v_1, v_2 according to the formula $\mathrm{d}(v_1, v_2) = \frac{\|v_1 - v_2\|_1}{\|v_1\|_1 + \|v_2\|_1}$ where $\|\cdot\|_1$ is the ℓ_1 norm (i.e., the sum of the absolute values of every entry in the vector). This algorithm computes a distance between two given functions; to make it comparable to the other algorithms, we use $1 - \mathrm{d}(v_1, v_2)$ as a similarity score.

2.5.2 Full Algorithm

Our full algorithm is shown in Section 2.7. It is a combination of the structural and nominal algorithms: we linearize the parse trees, and consecutive terminal nodes become bags of words. Nonterminals are compared using our structural method, and bags of words are compared using our nominal method.

2.6 Evaluation

In this section we compare our work against existing work on both cross-language and same-language clone detection.

2.6.1 Implementation and Environment

We have implemented our tool FETT in Scala and used the ANTLR parser framework as its front end, so that any language with an ANTLR grammar can be easily connected.

To test whether FETT can handle same-language clone detection with similar accuracy as specialized, language-specific tools, we configured NiCad 4.0 [126] to work at the function-level granularity and experimented with configurations until we found

the best-performing one for our tests[3].

Because we are comparing parse *trees*, we also want to determine how well we compete against the state-of-the-art tree edit distance algorithms, thus we compare one data set with APTED [112, 113]. We normalize the similarities using the method described in [93], and, as this normalization method requires a metric distance, we could not introduce weights for matches. We can still weight mismatches, though. We found that the parameters *mismatch* = 1, *deletion* = *insertion* = 5, *match* = 0 gave us the best results overall.

We chose the threshold for ignored functions (defined in Section 2.4.3) to be $\theta = 35$ for every experiment, and the exact tolerance parameters are given below for each case. We used the same set of equivalence classes with the same weights for all cases: conditional, loop, return, and function call were all weighted 5; assignments were weighted 2; and all other considered nodes were weighted 1.

Our experiments were run on a computer with an Intel i7 4790 3.6 GHz processor. FETT, Structural, Tree Edit Distance, and Nominal were given 8 GB maximum heap size and were set to use 4 threads.

2.6.2 Methodology

We used the standard statistical metrics of precision, recall, and *F*-measure to quantitatively assess the effectiveness of our different techniques.

Due to the sheer amount of possible clone candidates in large projects, it is difficult to manually obtain complete ground truth for clones in real-world programs. Hence, we created two separate data sets for evaluation:

Manual programs set (handwritten set). We implemented a set of small programs

[3]NiCad: threshold=0.5, minsize=4, maxsize=2500, rename=blind, filter=none, abstract=none, normalize=none

in different languages to create a setting in which we have complete knowledge of whether a pair of functions are clones. Statistics about the code are in Table 2.1.

Table 2.1: Statistics of handwritten clones.

Language Pair	LoC	#Functions	#Pairs	#Clones
Java	201	12	132	11
JavaScript	177	11		
Java	201	12	144	12
C++	195	12		
JavaScript	177	11	132	11
C++	195	12		

Randomly sampled program set (large set). We chose four libraries that have implementations in different languages and set the tolerance parameters[4] defined in Algorithm 1 to give the best results on a per-language pair basis. We randomly sampled functions from the files with the same names (ignoring extensions) and manually checked the pairs to create a sample with ground truth—this is essentially the sampling strategy used by Cheng et al. [42] applied to functions instead of diffs. We chose to reuse this sampling strategy due to the manual nature of our evaluation, and because we only possess finite human resources; it does not reflect the true distribution of clones, as function clone pairs are unlikely to be chosen in a standard uniform random sample—had we gone that route, our precision and recall scores would not have been meaningful. We are not aware of a better solution to this problem.

The first three libraries considered for this set are: the ANTLR parser framework, version 4 [2]; the toxiclibs computational design library [15]; and the ZXing barcode

[4]For FETT: $\mu = 6$ (match coefficient) and $g = -4$ (gap penalty) for the case of comparing Java and JavaScript, and $(\mu, g) = (9, -1)$ for Java/C++ and JavaScript/C++, and $(8, -3)$ for Java/Java. The nominal multiplier was set to 2 for all but the Java/C++ and JavaScript/C++ cases, where it was set to 3. For the Structural algorithm: $(7, -1)$ for JavaScript/Java, $(8, -4)$ for Java/C++, $(0.5, -2)$ for Java/Java, and $(9, -4)$ for JavaScript/C++.

image processing library [18]. We also considered two ports of the LAME MP3 encoding library in different languages that were ported by different developers to assess the efficacy of clone detection tools in such a scenario: lamejs, a JavaScript port [12]; and java-lame, a Java port [10]. Statistics about the libraries are in Table 2.2.

Table 2.2: Statistics of libraries considered for evaluation. LoC: non-blank non-comment lines of code, Fun's: # of functions found in each project, Nont'l (Nontrivial) Fun's: # of functions whose reduced parse trees are $> \theta$ (the chosen threshold), Pairs: the # of possible fun. pairs, Same-File Pairs: # of pairs of functions coming from files with the same name (ignoring extensions), Sel'd: # of selected pairs, Runtime: total time (H:M:S) to run our method.

Data set	Library	Lang. Pair	LoC	Fun's	Nont'l Fun's	Pairs	Same-File Pairs	Sel'd	Runtime	Clones
antlrj	ANTLR	Java Java	13,770 13,770	1,393 1,393	694 694	240,471	4,942	505	0:56:18	14
antlrjsj	ANTLR	Java JavaScript	13,770 7,323	1,393 728	694 405	281,070	6,240	663	0:25:01	45
antlrcppjs	ANTLR	C++ JavaScript	15,766 7,323	1,222 728	480 405	194,400	3,762	752	0:17:11	17
toxic	toxiclibs	Java JavaScript	36,178 36,976	3,734 4,108	2,156 2,321	5,004,076	11,637	1,060	3:01:12	63
zxing	ZXing	Java C++	38,968 22,784	2,659 866	1,689 405	684,045	1,388	254	2:10:51	45
lame	java-lame lamejs	Java JavaScript	20,950 11,112	575 285	436 232	101,152	4,645	873	0:27:37	34

2.6.3 Results

For our main set of tests, we compare FETT against (1) our purely Structural algorithm (i.e., no token similarity), and (2) our Nominal algorithm. We also apply the APTED tree edit distance algorithm combined with our abstraction method on our handwritten data set; tree edit distance takes at least an order of magnitude longer than the other tools, and we did not evaluate the large data set using tree edit distance because of this and due to its poor performance on the handwritten tests. We use NiCad on the Java-Java same-language case of our large data set.

Cumulative clone ratios. We look at the graphs of cumulative clone distributions to choose a good cut-off point for each of the three techniques. These graphs were

originally used in [42], and they are meant to give an intuition about where a clone detector separates clones from non-clones.

Similarity vs. cumulative clone ratio graphs track the ratio of clones to non-clones as the similarity score varies from 1.0 to 0. For example, at point 0.4 on the similarity axis, we plot the ratio of clones to non-clones of all samples with similarity scores > 0.4. A successful clone detector would have a similarity value at which there is a significant drop in this ratio, and that would create the optimal cutoff point. A clone detector may not assign very high scores to any pairs based on its similarity metric; in such cases, we start the plot from the first nonempty bin. Figure 2.7 shows the cumulative clone ratios for antlrj and toxic; graphs of other test cases are omitted because of space constraints, but they are of similar overall shape. We chose a cutoff point for each clone detector based on the drops from these graphs (e.g. we chose the cutoff point of 0.4 for FETT's Java/Java case). The relative shape of the graph is more important than absolute scores—squishing or stretching the similarity scores only affects the choice of the optimal cutoff point.

Handwritten test set. When evaluating the manually created (handwritten) data set, we used the same parameters $\mu = 7, g = -2$ overall for all pairs of functions in the data set and considered the combined results for both FETT and the Structural algorithm. FETT had its nominal multiplier set to 2. Figure 2.6 shows the clone distributions of different clone detection methods for the handwritten program set; and precision, recall, and F-measure (harmonic mean of precision and recall) for this set are given in Table 2.3. FETT and the Structural algorithm had a cutoff of 0.5, and the Nominal algorithm's cutoff was 0.6.

Handwritten test set discussion. The table and the figures paint a similar picture. Both FETT and the Structural algorithm seem to perform the best on this data set—the graphs

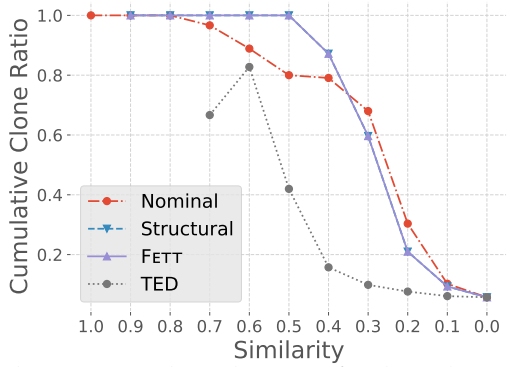


Figure 2.6: Cumulative clone ratio distribution for handwritten programs. Results of FETT and structural coincide.

for the higher similarity scores have a high clone ratio, and there is a sharp decline visible in both graphs as the similarity score is allowed to lower. The Nominal algorithm has a less sharp drop, and this indicates that it is assigning mid-range similarity scores with low precision. It is also notable that tree edit distance does so poorly; we believe that this is because we are not allowed to give weights to matches, as described above.

Table 2.3: Precision, recall, and *F*-measure for handwritten program set.

Data set	Method	Precision	Recall	*F*-measure
Handwritten	FETT	1.000	0.970	0.985
	Structural	1.000	0.970	0.985
	Nominal	0.886	0.939	0.912
	Tree Edit Dist.	0.821	0.697	0.754

Large test set. We now present and discuss all the cross-language results for our large test set. The same-language case is different from the cross-language cases, so the reader is asked to consult Figure 2.7b, which is indicative of all the cross-language cases, and not Figure 2.7a.

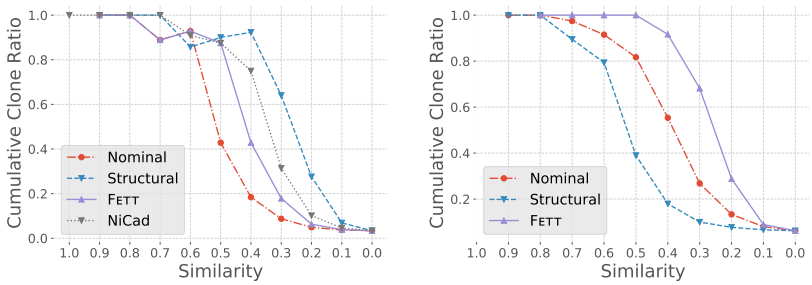
Cutoffs were chosen on a per-language pair basis that maximized a given tool's score. For FETT, for the three JavaScript/Java test cases and the Java/C++ test case, we used a cutoff of 0.4, and the rest used a cutoff of 0.5. For the Structural algorithm, we used a cutoff of 0.6 for JavaScript/Java, 0.5 for Java/C++ and JavaScript/C++, and 0.4

for Java/Java. For the Nominal algorithm, we used a cutoff of 0.5 for JavaScript/C++, and 0.6 for the rest.

Figure 2.8 shows precision, recall and F-measure of all the tools we compared for each data set and provides a visual and quantitative assessment of efficacy of all the techniques.

Large test set discussion. Clone ratios relate most closely to the precision scores for each data set, and from the results it appears that the Structural algorithm generally has the upper hand in this area—applying the intuition described above, we see that the Structural algorithm seems to cut off at the sharpest angle in most cases. It makes sense why this is the case, as pieces of code that look similar across languages are generally prime candidates for clones.

Precision is of course not the whole story. It is clear that FETT is able to take the best of both the nominal and structural worlds, and the F-measure is always the highest. When it comes to Structural's results, the toxiclibs case is an outlier, where we found that there were more cases of the structural differences; FETT's hybrid structural/nominal algorithm was able to make up for this, though.



(a) antlrj (b) toxic

Figure 2.7: Similarity vs. cumulative clone ratio for the samples from the large open-source program set.

Same-language test case. To assess performance on same-language clones, we compared our tool with NiCad on the Java version of ANTLR. Returning to the same figures,

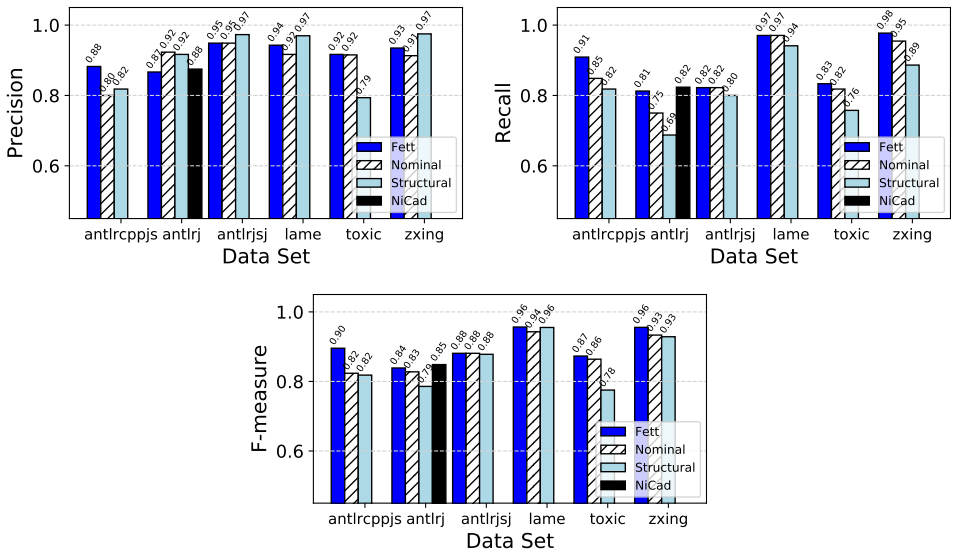


Figure 2.8: Precision, recall and *F*-measure of clone detection tools on the large program set.

the antlrj case is quite similar to the other language pairs in terms of precision, recall, and *F*-measure, which demonstrates that our tool is capable of holding its ground in a same-language setting.

FETT performs slightly worse (by one percentage point in terms of *F*-measure) than NiCad. This result is not surprising because NiCad uses more information about the code whereas we deliberately discard some information by abstracting parse trees to work in a cross-language setting. Even with our filtering of parse trees, FETT's *F*-measure score is very close, and this shows that our tool is capable of producing similar results to a dedicated same-language tool.

Overall results. We observe that the FETT's hybrid algorithm, in terms of *F*-measure, outperforms both the Nominal algorithm and the Structural algorithm consistently in our large test set experiments.

Limitations. FETT may have difficulty scaling to repositories with large numbers of large functions—a run of FETT on the entire toxiclibs library (comparing every function

pair, not just same file pairs) takes 5.13 hours—and so further improvements will be required to enable such a target. One possible future direction for improvements could be to develop semi-automated solutions where we have the user use her domain knowledge and pick out the files or functions to compare beforehand, or the user can prune the search space by telling the tool which modules are unrelated.

2.7 The Full Algorithm

Algorithm 1 Algorithm to find cross-language function clones.

1: **procedure** SIMILARITY($s, t, C, W, \mu, g, \theta$)
input:
Two sets of parse trees s and t, a set of equivalence classes between nodes C, an equivalence class weight function W, a match coefficient μ to control overall tolerance, a gap penalty coefficient g, a threshold θ on function subtree size
output:
A similarity score map S from pairs of functions to scores between 0 and 1.
2: $m \leftarrow \mathsf{NodeToEqClassMap}(C)$
3: $\hat{s} \leftarrow \{\mathsf{Abstract}(\mathsf{Preprocess}(\tau), m) \mid \tau \in s\}$
4: $\hat{t} \leftarrow \{\mathsf{Abstract}(\mathsf{Preprocess}(\tau), m) \mid \tau \in t\}$
5: $S \leftarrow \mathsf{EmptyMap}$
6: **for all** $f \in \{\mathsf{Functions}(\tau) \mid \tau \in \hat{s} \wedge |f| \geq \theta\}$ **do**
7: **for all** $g \in \left\{\mathsf{Functions}(\tau) \,\middle|\, \tau \in \hat{t} \wedge |g| \geq \theta\right\}$ **do**
8: $a \leftarrow \mathsf{Preorder}(f)$
9: $b \leftarrow \mathsf{Preorder}(g)$
10: $M \leftarrow \lambda i, j. \begin{cases} \nu\,(1 - \mathrm{d}(\mathsf{v}(i), \mathsf{v}(j))) & i, j \text{ are bags of words} \\ \mu W(i), & i = j \\ -\max(W(i), W(j)), & i \neq j \end{cases}$
11: Compute Z according to Section 2.4.3
12: $S \leftarrow S[(f, g) \mapsto \frac{\mathsf{SmithWaterman}_{(a,b,M,g)}}{Z}]$
13: **end for**
14: **end for**
15: **return** S
16: **end procedure**

The hybrid algorithm shown in Algorithm 1 works mostly the same as the structural algorithm as described in Section 2.4.3, but on a tree with terminals with two differences: (1) in the hybrid algorithm, the Preprocess function also merges consecutive terminals and converts terminals into bags of words, and (2) whenever the hybrid algorithm is

comparing two bags of words, it runs the nominal algorithm on the two terminals to compute the similarity between them; then, it multiplies this score with the nominal multiplier ν.

This method for introducing nominal information works at a finer granularity than our Nominal algorithm, and the token information is added in order (so the results are still highly structural); thus it should be less prone to false positives if the same tokens appear out of order. The nominal multiplier exists for tuning purposes, just like the other parameters, and intuitively it should be lower when an implementation has more name mismatches (e.g., for language pairs where the standard libraries look vastly different, or in an educational setting to check student code duplication, a smaller multiplier would be more suitable).

2.8 End-to-End Example

2.8.1 Parsing Is Such Sweet Sorrow

For our running example, consider the two functions in Figure 2.9. Both are implementations of binary search (one in C++, the other in Java), and they have some structural, nominal, and semantic differences.

We would like to systematically figure out that these two functions are indeed similar—to do so we must get them into a common form. We begin by parsing them.

2.8.2 A Tale of Two Parse Trees

We use the ANTLR parser generator and open-source grammar definitions to parse the two files and generate concrete parse trees. The original parse trees generated by ANTLR are shown in Figures 2.10a and 2.10b.

```
template <class T>
int binsearch(const T array[], int left, int right, T what) {
  if (right < left) return -1;
  int mid = (right + left) / 2;
  if (array[mid] > what)
    return binsearch(array, left, mid-1, what);
  if (array[mid] < what) {
    return binsearch(array, mid+1, right, what);
  }
  return mid;
}
```

(a) A generic C++ binary search implementation.

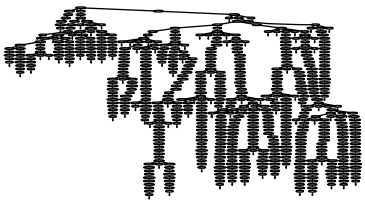
```
public static int binarySearch(int[] nums, int check,
                               int low, int high) {
  if (high < low) return -1;
  int center = (high + low) / 2;
  if (array[center] > check)
    return binarySearch(nums, check, low, center-1);
  else if (array[center] < check)
    return binarySearch(array, check, center+1, high);
  else
    return center;
}
```

(b) A non-generic Java binary search implementation.

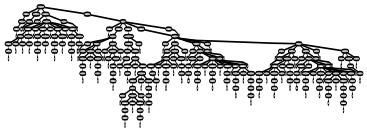
Figure 2.9: Two cross-language clones for binary search.

At a glance, it is obvious that the parse trees are vastly different; even though the two functions look similar textually, the parse trees do not reflect this. Our goal is to make these parse trees look more similar.

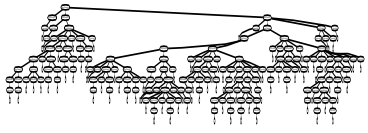
Some ANTLR grammars use hard-coded precedence—i.e., they have explicit precedence levels defined using nonterminals. A common example of this is the standard arithmetic expression grammar:



(a) Original C++ parse tree for the code in Figure 2.9a.



(b) Original Java parse tree for the code in Figure 2.9b.



(c) Reduced C++ parse tree obtained from the parse tree in Figure 2.10a.

Figure 2.10: Different forms of parse trees for the code snippets in Figure 2.9.

$$
\begin{aligned}
AddE &::= AddE + Term \\
&\mid AddE - Term \\
&\mid MulE
\end{aligned}
\qquad
\begin{aligned}
MulE &::= MulE * Number \\
&\mid MulE \ / \ Number \\
&\mid Number
\end{aligned}
$$

Parse trees for multiplication expressions would contain addition expression nonterminals, and this would confuse a cross-language clone detector when the other language's grammar does not encode precedence. We propose a simple technique that converts ANTLR parse trees that have hard-coded precedence into what we call *reduced parse trees*, resembling an abstract syntax tree. We then abstract upon all trees to arrive at a good common approximation. We refer to the process of creating reduced parse trees as *parse tree reduction*, and it is described in Section 2.5. After reduction, we are left with a new, smaller parse tree represented in Figure 2.10c. It is now both drastically smaller and more similar to the Java parse tree in terms of structure and number of nodes.

2.8.3 Score and Peace

We now have trees that superficially look similar, but they are still arranged differently and come from grammars with different nonterminals and levels of granularity. We address this issue by categorizing nonterminals into *equivalence classes* and comparing those classes. We call this process *parse tree abstraction*, and the resulting trees *abstracted parse trees*.

We must also find a way to compare the terminals. Before abstraction, we combine any consecutive terminal nodes into bags of words, correcting for camel-case, underscores, and capitalization. We can then compare the bags of words using characteristic vector similarity.

Finally, we linearize the abstracted trees by performing a *preorder traversal* of the two abstracted trees to minimize the remaining dissimilarities between the two languages.

For our example function pair, we place each node in the parse tree into its equivalence class, remove any "uninteresting" nonterminal nodes that do not belong to any equivalence class, and split the terminals on word boundaries. After linearization, this process yields the two final sequences to compare:

- C++: `FunDef, {"int"}, Id, {"binsearch"}, {"const"}, {"t"}, Id, {"array"}, {"int"}, Id, {"left"}, {"int"}, Id, {"right"}, {"t"}, Id, {"what"}, If, {"if"}, Relational, Id, {"right"}, Id, {"left"}, Return, {"return"}, Unary, Literal, Decl, {"int"}, Decl, Id, {"mid"}, Multiply, Add, ...`
- Java: `FunDef, {"int"}, Id, {"binary", "search"}, {"int"}, Id, {"nums"}, {"int"}, Id, {"check"}, {"int"}, Id, {"low"}, {"int"}, Id, {"high"}, If, {"if"}, Relational, Id, {"high"}, Id, {"low"}, Return, {"return"}, Unary, Id, Literal, Decl, {"int"}, Decl, Id, {"center"}, Multiply, Id, Add, ...`

We can now use a sequence alignment algorithm to compute the similarity of the two sequences. A pair of bags of words is compared by converting to a characteristic vector and calculating characteristic vector similarity, and these terminal sets never match against nonterminals. A pair of nonterminals is compared by checking the equivalence classes for equality. To incentivize aligning similar kinds of nonterminal statements with each other, we assign higher weights to some equivalence classes (such as those representing `if` statements). We also allow for a weight to be applied to the bags of words.

We feed the combined terminal/nonterminal sequences and weights to the Smith-Waterman local sequence alignment algorithm; this contrasts with Levenshtein distance, a global alignment algorithm that was used in Kraft et al.'s and Al-Omari et al.'s work. Local alignment algorithms are better suited for finding small pockets of similarity in sequences, whereas global alignment must consider the entire length of each sequence pair [23]. Thus, we believe that local alignment is a better choice for cross-language clone detection.

We normalize the result to get a similarity score between 0 and 1. These similarity scores do not have an absolute meaning but instead have a meaning relative to each other. In order to turn a score into a binary decision one can apply a threshold score

such that only function pairs with scores over the threshold are considered possible clones.

For this particular example, when we run our scoring algorithm with the parameters we chose for C++ and Java in Section 2.6, we get a score of 0.607; this is greater than the cutoff value of 0.5 we chose in our evaluation section, so we come to the conclusion that these two snippets are clones.

Chapter 3

Plagiarism Detection

3.1 Introduction

Plagiarism cheats plagiarizers out of their own education, and can lead to unfair grading of students who do not plagiarize. As such, the detection of software plagiarism is an important problem. While there is a large existing body of work on plagiarism detection (e.g., [103, 131, 117, 107, 134, 105, 120], we observe that plagiarism detection remains an unsolved problem. Specifically, existing plagiarism detection approaches tend to be inaccurrate, language-specific, or closed source [103], limiting their practicality.

Towards solving these problems, we observe the following:

- The syntax of most languages allows programs to differ in operationally indistinguishable ways, as by varying whitespace or variable names. This sort of syntactic noise is frequently exploited to obfuscate plagiarism [103].
- Most languages have features which are related to each other (e.g., both `if` and `switch` perform conditional code execution). A common plagiarism obfuscation is to substitute these features with each other [103].
- Many kinds of plagiarism obfuscations can be phrased as code additions, deletions,

or modifications.

In this work, we directly exploit these observations to inform the design of a novel plagiarism detection approach. Towards removing superfluous syntactic information, we adopt a syntax-aware approach with *filtering* refinements, which strip away anything the user considers uninteresting. As a countermeasure to language feature substitution, we define *abstraction* refinements, which allow the user to specify how similar different language features are to each other. Finally, we observe that the Smith-Waterman algorithm [133], classically from bioinformatics, is well-suited to plagiarism detection; the algorithm was was specifically designed to compare sequences in the presence of additions, deletions, and modifications. The use of these refinements, in conjunction with the Smith-Waterman algorithm, leads to a plagiarism detection solution which is accurate *by design*.

While our approach is syntax-aware, it is not tied to the syntax of any particular language. We use ANTLR [110] grammars for defining program syntax, which are commonly used when defining language parsers for compilers. Most languages can be defined with ANTLR grammars, and many already have ANTLR grammars available, making our approach applicable to most languages. While our filtering and abstraction refinements require an additional time investment to specify, this investment needs to be performed only once per language, making the specification burden overall minute.

To evaluate the accurracy of our approach, we compare it against multiple existing plagiarism detection approaches (namely, MOSS [131], JPLAG [120], and Zhang and Liu [117]) on Java programs. While designing our evaluation, we discovered that many evaluations are based on simulated plagiarized programs written by the very authors of the corresponding plagiarism detection technique (e.g., [117, 105, 65, 103, 138]). We observe that subtle biases can be introduced with this evaluation approach, as

it is possible for an author to subconsciously write programs which are more liable to be caught by their own technique. To reduce such bias, we base our evaluation on randomly-generated Java programs which have been automatically obfuscated using plagiarism obfuscation approaches observed in the wild (e.g., those in Martins et al. [103]). Our evaluation shows that our approach is superior to that of all competing approaches, in terms of true/false positives/negatives discovered.

Overall, our contributions are as follows:

- We introduce a novel syntax-aware plagiarism detection approach. We explain our technique in Section 3.3, and provide an example in Section 3.4.
- We introduce a new evaluation approach based on random program generation and obfuscation, and use this evaluation approach to evaluate our plagiarism detection approach. We find that our plagiarism detection approach offers superior accurracy to that of all competitors considered. Section 3.5 provides further details.
- We make our plagiarism detection and evaluation approaches, along with their corresponding source code, freely available.

3.2 Background and Related Work

Plagiarism is associated with malicious intent, and students who plagiarize code will often obfuscate it to avoid detection [103]. Plagiarism detection tools (hereafter referred to as "detectors") must see through these obfuscations.

Similarity-based detectors give scores to all possible pairs of programs, where higher-scoring pairs are more similar to each other (and more indicative of plagiarism) than lower-scoring pairs. Exactly what constitutes a "high score" is relative to the listing of scores; in practice, instructors would look at the code corresponding to some of the

highest-scoring pairs, in order to make a judgement call on whether or not plagiarism occurred. While this still requires instructors to perform manual code inspection to find plagiarism, it dramatically reduces the number of pairs to consider, going from hundreds to thousands of pairs to perhaps several. With this in mind, the purpose of detectors in practice is to eliminate unlikely cases of plagiarism, leaving only likely cases.

3.2.1 Evaluating Plagiarism Detectors

Effective detectors will consistently give high scores to plagiarism and low scores to non-plagiarism. This suggests an evaluation strategy: see how a given detector scores known cases of plagiarism and non-plagiarism, and measure how close these scores are to expectations. Ideally, this evaluation would involve real student assignment submissions, reflecting how detectors are intended to be used. However using real submissions has a problem: students must *honestly tell us* whether or not they plagiarized. Given the negative consequences of plagiarism, along with the fact that plagiarism is intentionally obfuscated to avoid detection, students cannot be relied upon to provide this information. As such, alternative evaluation strategies are frequently used.

A common alternative strategy is to manually create benchmarks which intentionally plagiarize code [117, 105, 65, 103, 138]. We argue that this is prone to bias, as authors may subconsciously write benchmarks which behave differently on their own detector. In contrast, we generate random programs and randomly perturb them in a manner consistent with plagiarism.

3.2.2 Related Work on Plagiarism Detection

Only two of the papers mentioned in a recent plagiarism detection survey [103] are open source, and each lacks widespread language support. We believe our approach fills this gap.

The most successful detector is MOSS [131], and we compare our tool against it in our evaluation. The algorithm behind MOSS (namely, Winnowing [131]) is freely available, though the MOSS tool itself is closed source. While Winnowing is language-agnostic, MOSS has language-specific modes, and setting these modes properly has dramatic impact on the results (see Section 3.5.3). As such, MOSS's approach is neither widely applicable nor freely available, unlike our approach. MOSS, Nayayanan et al. [107], and Chilowicz et al. [43] are all based on fingerprinting at various granularity levels, with MOSS using files, Nayayanan et al. using token sequences, and Chilowicz et al. using syntax trees.

JPLAG is a Java-specific detector which applies a string tiling algorithm at the token level [120]. Son et al.'s [134] approach, like ours, is based on comparing parse trees, though their comparison is based on convolution kernels instead of sequence alignment.

Tahaei and Noelle [141] detect plagiarism by observing multiple submissions of code and comparing the differences between those submsisions via logistic regression. In contrast, our work does not require multiple submissions of student programs. Their evaluation was done on student code labeled by the instructor.

Fu et al. [60] use a version of the TF-IDF statistic to find the most "surprising" differences on abstract syntax trees. Their evaluation was on short programs, and cases of plagiarism were generated from starter code using generators made by multiple people. This evaluation is similar to our own, but the types of program transformations

performed by the generators are unclear.

Prado et al. [119] introduce a detector which uses static and dynamic analyses to perform plagiarism detection, making it reliant on existing tools and analyses for the languages that it supports. In contrast, our method only requires an ANTLR grammar and minimal work to add a new language.

Sulistiani and Karnalim [138] compare token sequences, filtering them using cosine similarity for efficiency. We observe that token sequences remove some of the underlying structure of a program, and so our detector instead works with parse trees. Their evaluation consists of handmade cases of plagiarism.

Zhang and Liu's [117] approach is arguably the most similar to our own, as they also make use of the Smith-Waterman algorithm [133]. However, Zhang and Liu's approach has several key differences from that of our own: to the best of our knowledge, they convert trees to sequences via a preorder traversal instead of a postorder traversal (Section 3.3.1), their scoring function is less general than that of our own (Section 3.3.5), they do not perform sorting (Section 3.3.2), and most importantly, they lack our filtering and abstraction refinements (see Section 3.3.3). For these reasons, we have found that Zhang and Liu's approach cannot detect plagiarism as accurately as our proposed method, as our evaluation shows (Section 3.5.3).

3.3 Our Method

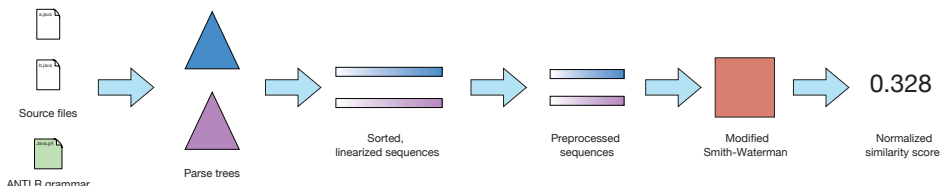


Figure 3.1: A graphical overview of our method.

Figure 3.1 provides a graphical view of our process. In the rest of this section, we

cover each part in more detail.

3.3.1 Parsing: ANTLR Grammars to Sequences

Our method requires as input the parse tree of a program—because parse trees can be made for any programming language, every further step works exactly the same regardless of the initial language. We assume that input programs are syntactically well-formed, which is reasonably ensured via a course policy which states that only compiling programs may receive credit. To get parse trees, we use the ANTLR parser generator [110], which has existing grammars for several popular languages.

Adding support for a new language

With this in mind, to apply our approach to a new language, a user needs only to: (1) find (or create) an ANTLR grammar describing the language; (2) use the ANTLR tool to create a corresponding parser for the grammar; (3) write fairly straightforward boilerplate code to attach the parser to our tool; (4) provide minimal information for filtering (see Section 3.3.3). To keep things concrete, we focus on Java for the remainder of this chapter, but any other language could be substituted in its place.

The parser from ANTLR produces parse trees corresponding to input programs. Once we obtain a parse tree for two given programs, we *linearize* the two trees by performing a postorder traversal and outputting the labels of each node traversed. Once we have the two programs that we wish to compare in this linearized form, we then preprocess them before performing sequence alignment on them. The first preprocessing step is sorting the functions.

3.3.2 Sorting

Because we are comparing entire files at a time, we must be wary of a common plagiarism operation: function rearrangement. The order of the functions should not matter when evaluating cases of plagiarism—it makes no difference if `foo()` comes before or after `bar()` if their contents are plagiarized.

The heuristic we use is to sort functions in each file by size (i.e., the number of nodes in the parse tree) before we linearize each parse tree. We found that this gives good results while avoiding the exponential blowup that would occur with trying every possible combination of functions.

After the linearized sequences are in this sorted order, we can further prune and enhance the information contained within them.

3.3.3 Filtering, Abstraction, and Weight

In this step, we determine which parse tree node labels to keep, what equivalence class they belong to (if any), and provide a relative score that indicates how important a given label is. Users must provide information specifying how labels are grouped, once per language. We explain each case along with its justification.

Filtering

Another common plagiarism operation is renaming variable names and other identifiers, so we cannot trust any such node in a parse tree. We have a filtering phase to get rid of these and other parse tree labels that we deem unsuitable for comparison. In fact, we found the ratio of useful to interesting nodes to be so small that we only require a list of nodes to *keep*.

As an example, consider the Java grammar rule for expressions:

```
expression
    : primary
    ...
    | methodCall
    | NEW creator
    | '(' typeType ')' expression
    | expression postfix=('++' | '--')
    | prefix=('+'|'-'|'++'|'--') expression
    ...
```

This is essentially a catch-all rule, and almost every line of Java code will contain an `expression` node in its sub-parse tree. As such, there no useful information in `expression`, and so we omit it from our output. There are many similar rules in the Java grammar.

Abstraction

Several nodes act in a similar way, and should be considered as such. For example, several `if/else` statements may be transformed into one `switch` statement, and vice versa. We therefore consider parse tree node labels for `if/else` and `switch` to be in the same equivalence class, and we consider them to "match" in our Smith-Waterman algorithm. Different kinds of loops also belong together in the same equivalence class.

This concludes our preprocessing steps. We are now ready to compare two program sequences with a modified Smith-Waterman algorithm. Before introducing our algorithm, we first discuss sequence comparison in general.

3.3.4 Comparing Sequences: False Start

Smith-Waterman is a relatively unfamiliar algorithm to most computer scientists, so we will briefly compare it to the more familiar and similar concept of string edit distance. Levenshtein distance is the most common string edit distance measurement

algorithm.

Levenshtein distance has notions of insertion, deletion, and mismatch "costs", and the final answer is the minimized cost of performing these different string-altering operations. Overall, Levenshtein distance determines the most cost-effective way to turn one string into another. Smith-Waterman has similar computational costs, but matching is more customizable; for example, Smith-Waterman allows a *matrix* of scores, which allows a programmer to give different match/mismatch scores to different pairs of elements of a sequence. In this work, we assign higher scores to programs that match at certain key nodes, such as loops and if statements.

Levenshtein distance also computes a cost over the entire string, in what is known as a global alignment. For example, comparing the strings `A = "cat"` and `B = "␣␣␣cat␣␣"` results in a score of 5, because the entire string A must be transformed into the entire string B. In contrast, Smith-Waterman performs a *local* alignment and finds the largest substrings of each string that match. Local alignment is important for our purposes because we want to give a high similarity score to a pair of programs when a *portion* of one is found in the other; our method would be less accurate if we only gave high similarity scores to program pairs when they have almost everything in common.

3.3.5 Comparing Sequences: Smith-Waterman

The Smith-Waterman algorithm [133] is a dynamic programming-based sequence alignment algorithm. It computes a numerical value representing how similar or dissimilar two sequences are. Originally implemented with bioinformatics in mind, it is used to compare biological sequences (e.g., RNA and proteins) and to determine how closely two such sequences overlap each other. The algorithm works for sequences

of any kind, not just sequences of characters; in this work, we align sequences of abstracted parse tree nodes.

Smith-Waterman is parameterized by a gap score (for insertions and deletions) and a scoring matrix (for matching and mismatching). This matrix is consulted each time nodes are compared to determine how similar they are, and so tuning this matrix has a major impact on the results. We simplify the process of constructing a scoring matrix by: (1) providing all nodes that match a default match score, and all nodes that do not match a default mismatch score; and (2) assigning relative weights to certain matches.

Weights

Experimentally, we found that some label classes are more important than others. For example, it is quite likely that several `if` statements will match in a plagiarized pair of programs, whereas assignment statements do not closely correspond. For this reason, we assign relative weights to certain equivalence classes of labels. Just like the rest of our method, these weights are completely customizable to fit any instructor's needs. The relevant portion of our weight/equivalence class file follows:

```
{"conditional": 5, "loop": 5, "return": 5,
"functioncall": 5, "assign": 2}
```

This syntax specifies that conditional statements (e.g., `if/else`, `switch`) that match are to be given $5\times$ the match score. If such a node does not match, we perform the same multiplcation against the mismatch score.

Final steps

After obtaining a similarity score from the Smith-Waterman algorithm, we normalize the scores to be between 0.0 and 1.0. Without such a normalization, larger plagiarized program pairs (which naturally have more similar portions than shorter

plagiarized program pairs) would have larger scores, rendering sorting by score useless.

3.4 Complete Example

This section goes through an in-depth example of our method.

Description of the programs

Figure 3.2 contains three example programs: A, B, and C. Programs A and B have been plagiarized using several common operations (e.g., identifier renaming, manipulation of spacing, etc.) [103], while program C is independent of A and B. We use this example to demonstrate our method, and we further elaborate on plagiarism operations in our evaluation (Section 3.5).

Parsing

We first parse the three programs, creating the parse trees shown in Figure 3.3. None of these parse trees look similar to each other, with differences in both the number of nodes and node structure. Our goal is to get Programs A and B into a format where they both "look" similar; it is for this reason that we preprocess and linearize the parse tree nodes.

Sorting

We then extract functions and sort them by their size, in terms of the number of parse tree nodes. In this example, sorting will not change the function order in Program A. However, in Program B, `func2` will be moved to come *before* `func1`. This sorting heuristic thus puts the plagiarized functions in the same order.

```java
public class A {
    public void foo() {
        for (int i = 1; i < 10; i++) {
            System.out.println(i);
        }
    }

    public int bar() {
        int sum = 0;
        for (int i = 0; i < 10; i++) {
            for (int j = 0; j < 10; j++) {
                sum += i + j;
            }
        }
        return sum;
    }
}
```

(a) Program A

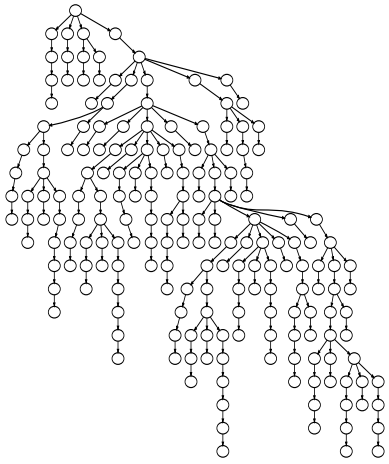
```java
public class B
{public int func1(int dummy1, int dummy2)
    {int extraStmt1 = 5; int extraStmt2 = 42;
     int s = 0; int i = 1;
        while (11 > i)
        { int j = 1; while (11 > j) {
                s += (i - 1) + (j - 1);
                j = j + 1;
            }
            ++i;
        }
        return s;
    }
    public void func2()
    {int z = 1; while (10 > z)
    {System.out.println(z); z++;}}
}
```

(b) Program B

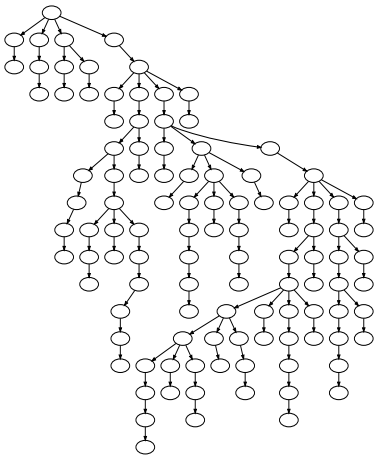
```java
public class C {
    public void hello() {
        System.out.println("Hello, world!");
    }

    public int sum2(int x) {
        int a = 0;
        for (;;) {
            if (x <= 0) break;
            a += x;
            x--;
        }
        return a;
    }
}
```

(c) Program C

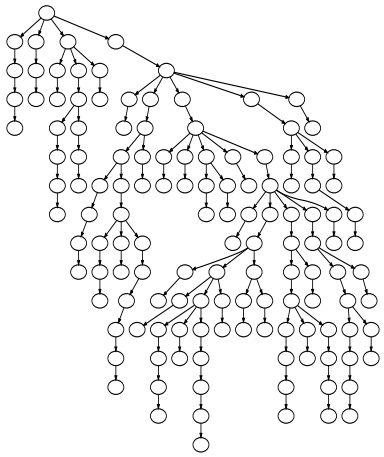
Figure 3.2: Three contrived example programs: A and B for a plagiarized pair, while C has no plagiarized information taken from A or B.



(a) Parse tree for Program A



(b) Parse tree for Program B



(c) Parse tree for Program C

Figure 3.3: A bird's eye view of the parse tree structure for Programs A, B, and C.

Linearization

We then perform a postorder traversal of the parse trees before we prune further. The contents, however, are the important part, and we want to get Programs A and B into a format where they have a lot in common. The first few nodes of Program A's `bar` function are (bold font indicating common elements):

TerminalNodeImpl, IdentifierNonterminal, (,
TerminalNodeImpl,), **TerminalNodeImpl,**
FormalParameters, {, **TerminalNodeImpl, int,**
TerminalNodeImpl, PrimitiveType, TypeType, sum,
TerminalNodeImpl, VariableDeclaratorId, =,
TerminalNodeImpl, 0, **TerminalNodeImpl,** Literal,
Primary, PrimaryExpression, VariableInitializer,
VariableDeclarator, VariableDeclarators,
LocalVariableDeclaration, ...

The first few nodes of Program B's `func1` function are:

TerminalNodeImpl, IdentifierNonterminal, (,
TerminalNodeImpl, int, **TerminalNodeImpl,**
PrimitiveType, TypeType, dummy1, **TerminalNodeImpl,**
VariableDeclaratorId, FormalParameter,
TerminalNodeImpl, **int, TerminalNodeImpl,**
PrimitiveType, TypeType, dummy2, **TerminalNodeImpl,**
VariableDeclaratorId, FormalParameter,
FormalParameterList,), **TerminalNodeImpl,**
FormalParameters, {, **TerminalNodeImpl,** int, ...

While there are similarities at the beginning, they do not last long. `func2`'s extra

parameters and names, as well as the extra statements at the beginning, are taking up a substantial amount of space, so we want to remove them. Similarly, any identifier names are naturally untrustworthy, and are thus ripe for removal. Not shown are the nodes for the `while` and `for` loops, which are currently considered different; we want to treat these as being similar to each other.

Pruning and Abstraction

Beforehand, we have compiled a set of "useful" parse tree nodes which we want to keep; it is only necessary to do this once per language. In this step, we prune out any node that is not in our useful set. After performing this pruning, we are left with sequences that are typically ~33% as large as the originals. Pruning reduces both the amount of input to process downstream (improving performance), as well as the amount of irrelevant "noise" in the input (improving accurracy/precision). The final nodes of Program A's `bar` function are (bold font indicating common elements):

`..., Primary,` **`Primary, AdditiveExpression,`**
`AssignExpression, ForLoop, ForLoop, Primary,`
`ReturnStatement`

The last nodes of Program B's `func1` function are:

`..., Literal,` **`Primary, AdditiveExpression,`**
`AssignExpression, WhileLoop,` `Primary,`
`PrefixIncDecNegPlus,` **`WhileLoop, Primary,`**
`ReturnStatement`

These pruned sequences share several similar elements, and sequence alignment will give a larger score on these than the originals. One final step that is not shown is abstraction, where we group similar nodes. In this example, we will end up grouping

`WhileLoop` and `ForLoop` into the same equivalence class, and so we will consider them to represent the same node; this allows sequence alignment to return a larger score for these plagiarized programs.

Smith-Waterman

Running Smith-Waterman on the abstracted sequences is the last step. We run the algorithm for each pair of input programs with the scoring matrix generated by the supplied node weights. The scores are relative to each other, and are not meaningful on an absolute scale. Higher-scoring pairs are more indicative of plagiarism than lower-scoring pairs, and in practice users need only look at the several highest-scoring pairs to find plagiarism.

The final scores

After all this work has been done, we are left with a similarity score between 0 and 1 for each pair of programs:

- Program A and Program B: 0.299
- Program A and Program C: 0.193
- Program B and Program C: 0.120

In this case, our method has correctly scored the plagiarized program pair higher than the non-plagiarized program pairs.

3.5 Evaluation

In this section we compare our method to existing methods.

Table 3.1: Results for each method. ↓ indicates that a lower value is better, and ↑ indicates that a higher value is better

Method	Cutoff	Time (s) (↓)	True Pos. (↑)	False Pos. (↓)	True Neg. (↑)	False Neg. (↓)	Precision (↑)	Recall (↑)	F-Measure (↑)
Our method	0.15	115.9	33	16	4,884	17	0.673	0.660	0.667
Zhang and Liu	0.58	970.3	16	37	4,863	34	0.302	0.320	0.311
MOSS, Java Mode	0.44	30.8	11	33	4,867	39	0.250	0.220	0.234
MOSS, Text Mode	N/A	4.1	0	0	4900	50	∞	0.000	0.000
JPLAG	0.17	4.7	18	46	4,854	32	0.281	0.360	0.316

3.5.1 What We Compare Against

We evaluate our method against MOSS [131] and JPLAG [120], both widely used plagiarism detection tools. We tested MOSS in Java mode and in text mode; we did this to evaluate what would happen if MOSS was run on a language that it does not support. Both MOSS and JPLAG were run with default parameters.

We also evaluate against our implementation of Zhang and Liu's method [117]. Both our method and Zhang and Liu's method require multiple parameters, namely a tree traversal order, a match score, a mismatch score, and a gap score. We experimentally determined optimal values for each of these parameters. For our method, we use postorder traversal, a match score of 1, a mismatch score of -1, and a gap score of -2. For Zhang and Liu, we use preorder traversal, a match score of 1, a mismatch score of -1, and a gap score of -1—we also tried a gap score of -2, but -1 gave better results.

3.5.2 Benchmarks and Methodology

To address the bias problem discussed in Section 3.2.1, we employ random program generation and random program transformation to create our benchmark suite. We have created 50 pairs of original and plagiarized Java programs for a total of 100 Java programs, and we evaluate by comparing all pairs of those 100 programs. We implemented our generator to perform several popular plagiarism obfuscations observed in the wild [103]:

- Addition of random amounts of whitespace.
- Changing every identifier name (e.g., variable names, argument names, function names, etc.).
- Changing type names to equivalent ones (we overapproximate this by transforming type names into random strings).
- Changing operations to equivalent ones (e.g., replacing $A + B < C$ with $C > B + A$).
- Replacing control structures with equivalent ones (e.g., swapping while with for and vice versa).
- Changing the order of statements and functions.

Our metrics are the standard ones of precision (true positives / (true positives + false positives)) and recall (true positives / (true positives + false negatives)). These numbers are combined via F-measure [130], which provides a single number to compare the relative performance of each method.

3.5.3 Results

Table 3.1 shows the results for every method that we tried. There were 4,950 possible pairs of programs to compare, and 50 "true" cases of plagiarism; all other pairs are considered to be false positives. Each method produces a score between 0 and 1 indicating the likelihood that a pair was plagiarized. To convert this score to a binary yes/no for whether or not plagiarism was detected, we selected a cutoff value for each technique, where scores $\geq$ to the cutoff were considered indicative of plagiarism. Cutoff scores were picked to maximize the F-measure for each method.

Discussion

Our method has the best F-measure, and the running time is in the middle of the group.

Both ours and Zhang and Liu's methods start with the same parse trees and subsequently run an $\mathcal{O}(n^2)$ algorithm. However, our method is nearly 8.4x faster than Zhang and Liu's method. This is because our filtering stage dramatically reduces the input size, minimizing n in the aforementioned time complexity. It is possible that pruning obviously non-matching pairs (e.g., programs of vastly different lengths) could help to further reduce our runtime, but we leave this investigation for future work.

MOSS' text mode returned no matching results, which was to be expected; all identifiers were changed, and so there would be no meaningful matches in any sliding window of results. As such, while Winnowing [131] (the technique MOSS is based on) is language-agnostic, MOSS itself is not. While some effort is needed to apply our technique to a new language, this is fundamentally impossible with MOSS, due to MOSS' closed-source nature.

MOSS' Java mode and JPLAG both had similar results. MOSS' time includes sending the files over the network and waiting for a response from the server, so it is possible that the running time of the main plagiarism detection algorithm is closer to that of JPLAG.

Threats to Validity

Our method cannot detect every kind of plagiarism. For example, it cannot currently handle obfuscation where functions are broken down into many small functions, though it would be possible to mitigate this issue (at the expense of increased runtime) by comparing all function pairs. Semantic similarity is also a research-worthy chal-

lenge: it is always possible to trick a syntax-based method by swapping the code with completely different code that does the same thing (e.g., insertion sort with quicksort).

Chapter 4

Fixpoint Reuse

4.1 Introduction

JavaScript programs are an integral part of the internet ecosystem, from the server to the client, and present a tempting target for malicious actors. For example, JavaScript-based browser addons have complete access to the browser's state and can do anything they want with that information, including collecting and disseminating users' sensitive data; examples of such behavior have been found in the wild [9, 13]. Thus, JavaScript is an important target for static analyses that attempt to ensure safety and security. Numerous such analyses have been published, e.g., to ensure that browser addons do not leak sensitive information [142, 77, 143].

However, a single-time static analysis is not sufficient when programs are continually updated with new versions. There are known instances where malicious code has been snuck into existing JavaScript programs during such updates [5]. To ensure safety and security, static analyses must be run on every version of a program, not just the first one. However, JavaScript is a highly dynamic and difficult language to analyze with precision, and the resource cost can be high. If there is a central entity serving as the main gateway for these programs (e.g., browser addon repositories) that is responsible

for running all of these analyses, they must shoulder the bulk of this cost. Being forced to re-run the analyses for every update and new program version only exacerbates these problems. **The contribution of this chapter is a technique called *fixpoint reuse* to mitigate the performance problems attendant on repeatedly statically analyzing the same JavaScript program over multiple updates and versions.**

Our technique falls under the general rubric of *incremental static analysis*, a topic that has been extensively studied over the years. However, no existing work deals with a dynamic language such as JavaScript. In particular, the existing work generally relies on two major assumptions: (1) an *a priori* known flow-graph model of the program; and (2) a known or (given the flow-graph model) trivially computable syntax mapping between the old and new program versions. Unfortunately, JavaScript programs do not have a simple flow-graph model, and in fact require extensive and expensive static analysis to compute precise control-flow and data-flow information. Thus, the existing works' assumptions do not hold and they are not immediately applicable to languages such as JavaScript.

We rely on two key insights to reposition incremental static analysis for JavaScript: (1) the problem of matching between two program versions is similar to the problem of clone-detection, and thus we can leverage existing clone-detection techniques [35, 76, 122]; and (2) whereas modern incremental analyses are precise (i.e., yield the same answer as a non-incremental analysis), we can relax the requirement for precision while still getting useful results. That is, our incremental analysis can yield additional false positives beyond what a from-scratch analysis would yield, but we show empirically that this does not happen very often. Together, these insights enable our technique to achieve speedups within $2\times$ of an *optimal incremental analysis* (which we define as an incremental analysis on a program version that is identical to the earlier version, thus allowing maximum reuse).

In the context of a central gateway such as a browser addon repository that is analyzing third-party programs, another benefit of our technique is that it does not rely on the gateway having to store past analysis results for every program that it analyzes. Previous analysis summaries can safely be left to the third-party developers to store and transmit with any program updates; our technique guarantees that the results of the analysis will still be sound. The most that a malicious developer could do is to degrade the performance and precision of the incremental analysis up to some limit, after which we would fall back to a normal from-scratch analysis. Our technique is flexible enough to handle a variety of scenarios that distribute the analysis work between the central authority and the app developer in different ways, while still allowing the central authority to guarantee the soundness of the results.

4.2 Related Work

In this section we review the work on incremental static analysis to put our technique in context.

4.2.1 Incremental Analysis via Restarting Iteration

Perhaps the most closely related work to our technique is from the early '80s. There are three works that present a technique called *restarting iteration* [45, 46, 64]. Unlike our fixpoint-reuse work, restarting iteration assumes a known control-flow graph and a provided mapping from old to new program version. Similarly to our fixpoint-reuse technique, the technique does not guarantee a precise incremental analysis, i.e., it could introduce additional false positives. The main contributions of our work in relation to this old work are (1) removing the assumption of a known, simple flow-graph, thus

making the technique applicable to dynamic languages such as JavaScript; and (2) providing a method to compute a mapping between program versions rather than assuming one will be provided, thus making the technique more practical.

4.2.2 Precise Incremental Analysis

Starting in the late '80s the work on restarting iteration was abandoned in favor of techniques that guarantee precise results—i.e., analyses that return the same results as a non-incremental analysis. This flavor of incremental analysis has dominated the field since that point [118, 40, 102, 74, 98, 135, 26, 89, 97, 140, 104, 44, 106, 87, 68]. Modern incremental analyses focus on pruning old results that might negatively impact precision. There have been a number of advancements, but all are for non-dynamic languages with simple flow-graph program models and assume that either the version mapping is provided or can be simply computed from the respective flow-graphs. None of the precise incrementalization methods is immediately applicable to languages such as JavaScript.

"Incremental" Analysis of JavaScript

Livshits and Guarnieri [95] present Gulfstream for streaming JavaScript programs. The word "incremental" is used in a different context in that paper: the analysis is incremental in the sense that it statically analyzes all JavaScript code that it can, and then when dynamic processes load *new JavaScript files*, those files are analyzed in an incremental fashion. The paper presents a points-to analysis of JavaScript that is unsound and makes use of analysis result invalidation; whereas our work maintains soundness, is a general abstract interpretation, and does not invalidate any previous information.

4.3 Fixpoint Reuse

In this section we describe the problem that we are solving and the basic ideas of our approach, called *fixpoint reuse*. We stay at a relatively high level in this section in order to convey the central concepts; Section 4.4 will go into more details within the context of our method's instantiation to a specific JavaScript analysis framework. We invite any reader without a background in program analysis to consult Section 4.6 for more information on abstract interpretation and taint analysis.

4.3.1 High-Level Summary

The three inputs are P_{prior} (the prior version of the program), FP_{prior} (the fixpoint analysis solution for P_{prior}), and P_{upd} (the new, updated version of the program). We assume FP_{prior} is in the form of a map from program points to abstract states. The goal is to compute $FP_{\widehat{upd}}$, an over-approximation of FP_{upd} (the precise fixpoint analysis solution for P_{upd}).

Our approach is to (1) compute a partial mapping $P_{prior} \rightarrow P_{upd}$ from program points in P_{prior} to program points in P_{upd} that correspond with high confidence, then (2) use $P_{prior} \rightarrow P_{upd}$ to seed the initial analysis state for $FP_{\widehat{upd}}$ with the abstract states for corresponding program points as given in FP_{prior}. We then (3) analyze P_{upd} starting from the seeded initial analysis state and ensuring that we visit every program point in P_{upd} at least once in order to guarantee a sound analysis.

Algorithm 2 shows a high-level view of the entire matching and reuse process. Section 4.4 describes exactly how we instantiate this generic algorithm in the case of JavaScript and the SAFE analysis framework.

Algorithm 2 Generic Fixpoint Reuse

1: **procedure** REUSE(P_{prior}, P_{upd}, FP_{prior})
2: **Input:** The two versions of the program, in some traversable form, and version P_{prior}'s fixpoint data structure
3: **Output:** $FP_{\widehat{upd}}$, a prepopulated fixpoint for version P_{upd}
4: Match program points, call sites, abstract addresses, and variable names of P_{prior} with those from P_{upd} using program similarity techniques
5: Populate $P_{prior} \rightarrow P_{upd}$
6: Populate $FP_{\widehat{upd}}$ using FP_{prior} and $P_{prior} \rightarrow P_{upd}$.
7: Analyze P_{upd} starting from $FP_{\widehat{upd}}$
8: **end procedure**

4.3.2 Example

To make this process more concrete, we provide a specific example based on taint-tracking program analysis. Consider the two program excerpts in Figure 4.1. Version P_{prior} contains a function `f`, which is called with the argument `secret`, and we do not want the value of this secret variable to leak to the outside world. Assume that these snippets are part of a larger program.

After running the analysis on version P_{prior}, we have a fixpoint solution FP_{prior} that maps every *(calling context, program point)* pair in P_{prior} to some abstract state. An abstract state will, for this analysis, map program variables and abstract heap locations to either *definitely tainted*, *definitely not tainted*, or *possibly tainted*. We can inspect the abstract state at the `output` statement to see if the value being output is tainted in any calling context. Suppose that the result is that there is no taint in this case.

Figure 4.1b shows an updated program version P_{upd} (which we can see by inspection does leak tainted data). Our incremental analysis will first use a program matching algorithm to find corresponding program points between the two versions. In this case, it finds that the definition of `f` in P_{prior} matches the definition of `f` in P_{upd}, that a number of the basic blocks inside `f` match between P_{prior} and P_{upd}, and finally that the

```
function f(x) {
  for (...) {
    for (...) {
      // expensive computation
    }
  }

  return result;
}

var secret = 42;
var next = f(secret);
```

(a) Version P_{prior}

```
function f(x) {
  for (...) {
    for (...) {
      // same expensive computation
    }
  }
  output(x); // leak!
  return result;
}

var secret = 42;
var next = f(secret);
```

(b) Version P_{upd}

Figure 4.1: Two programs, unalike in dignity

last two lines in each version also match.

The incremental analysis will then use this matching to transfer abstract states from FP_{prior} to $FP_{\widehat{upd}}$. As part of this matching and transfer, calling contexts and abstract heap locations from FP_{prior} will be renamed to the corresponding calling contexts and abstract heap locations appropriate to P_{upd}. The final analysis on P_{upd} will start from this seeded $FP_{\widehat{upd}}$ to compute the final fixpoint solution for P_{upd}. The analysis must be certain to visit every program point in P_{upd} at least once to guarantee a sound solution (e.g., by initializing the worklist with every program point rather than just the entry point). This requirement is due to the fact that otherwise the seeded $FP_{\widehat{upd}}$ may cause the analysis to prematurely converge at a pre-fixpoint solution.

The incremental analysis correctly concludes that there is a leak in the updated program. With reuse, the taint analysis on version P_{upd} could reuse the analysis results for a vast majority of the program and thus converge much faster than a from-scratch analysis. While this example is trivial, we have achieved good results and significant speedups using this method on real-world JavaScript code that runs in browsers and/or servers.

4.3.3 On Program Matching

Computing the map $P_{prior} \rightarrow P_{upd}$ is an important part of the process that can have extreme effects on the efficacy of the incremental analysis. When matching there are three possibilities:

1. We correctly match,
2. We incorrectly match, or
3. We cannot match.

The first case is the best case; the more correct matches we compute the more effective the incremental analysis will be in improving performance. The third case, while not ideal, isn't too harmful; the incremental analysis won't benefit from the prior analysis in this case, but it can simply compute the information in the same way as a from-scratch analysis.

The second case, however, is by far the worst case and demonstrates the non-triviality of the matching problem. An incorrect match means that the incremental analysis will be seeded with incorrect information from the prior analysis. While this incorrect information doesn't affect the soundness of the results, it does mean that the incremental analysis must propagate this incorrect information to all reachable program points, reducing performance and polluting precision. Thus, it is far better to fail to match a program point than it is to incorrectly match a program point. This means that our matching algorithm must carefully balance between matching often and matching well. Failing to match often enough means that we get no performance improvement; failing to match well means that we get both performance and precision *reduction*.

4.4 Fixpoint Reuse for SAFE

Our prototype implementation is built on top of SAFE version 2.0. The SAFE JavaScript analysis framework [91] does not perform its analysis at the level of the original JavaScript source code. Instead, the source is translated to a simpler intermediate representation (IR) that is more amenable to analysis—it breaks complicated expressions into simpler ones, and makes explicit the implicit operations of the JavaScript language (e.g., type coercion, argument array construction before a function call, etc.).

In order to reuse analysis results, we must therefore create a correspondence

mapping between programs at the level of SAFE's IR. In this section we describe its constituent pieces, along with the different matching methods we implemented and evaluated against.

4.4.1 Functions, Blocks, and Instructions, Oh My!

SAFE programs are divided into three main categories: functions, blocks, and instructions. We explain them via an example. Consider the JavaScript program in Figure 4.2a, made up of two functions and a free-standing statement that calls one of those functions. This program is translated into SAFE's intermediate program as Figure 4.2b.

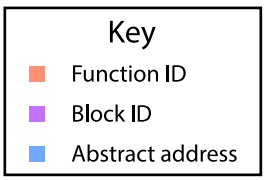
At both the textual- and the data structure-level, the program is represented as a hierarchy of three entities: functions, blocks, and instructions. Any statements that occur outside of a function are gathered together in the `top-level` function, which serves as the entry point to the translated IR version of the program and its subsequent analysis. For example, there are eight separate blocks in the `top-level` function—Entry[-1], Block[0], Call[1], AfterCall[2], AfterCatch[3], Block[4], Exit[-2], and ExitExc[-3]. There are nine instructions inside Block[0], which prepares for the call to `isEven`, and there is just one instruction in the Call[1] block, which performs the actual call to the function.

When we match JavaScript programs at the SAFE IR level, we match functions, blocks, and instructions, in that order. Once we are confident that two functions correspond, we then match their blocks, and once we believe we have chosen the best block correspondence we match individual instructions. Matching instructions is necessary for two main reasons: (1) correctly translating calling contexts from FP_{prior} to $FP_{\widehat{upd}}$ requires accurate mapping of call instructions, and (2) correctly translating

```
function isEven(x) {
  if (x == 0) {
    return true;
  } else {
    return isOdd(x-1);
  }
}

function isOdd(x) {
  if (x == 0) {
    return false;
  } else {
    return isEven(x-1);
  }
}

var b = isEven(42);
```

(a) A JavaScript program

```
function[0] top-level {
   Entry[-1] -> [0]

   Block[0] -> [1], ExitExc
  [0] isEven := function (1) @ #4, #5
  [1] isOdd := function (2) @ #9, #10
  [2] noop(StartOfFile)
  [3] <>obj<>15 := @ToObject(isEven) @ #11
  [4] <>temp<>16 := 42
  [5] <>arguments<>17 := allocArg(1) @ #12
  [6] <>arguments<>17["0"] := <>temp<>16
  [7] <>fun<>18 := @GetBase(isEven)
  [8] <>this<> := enterCode(<>fun<>18)

   Call[1] -> ExitExc
  [0] call(<>obj<>15, <>this<>, <>arguments<>17)

   AfterCall[2] -> [4]

   AfterCatch[3] -> ExitExc

   Block[4] -> Exit, ExitExc
  [0] b := <>Global<>ignore1
  [1] noop(EndOfFile)

   Exit[-2]

   ExitExc[-3]

}

function[1] isEven {
  ...
}

function[2] isOdd {
  ...
}
```



(b) The same program converted to SAFE IR

Figure 4.2: SAFE's intermediate representation for JavaScript programs

abstract heap addresses (in the IR, anything that begins with a "#") from FP_{prior} to $FP_{\widehat{upd}}$ requires accurate mapping of allocation instructions. In both cases, failing to accurately match corresponding program entities does not cause unsoundness but results in imprecision and will cause the analysis to visit unnecessary program locations and heap addresses. Thus it is important that we match with high confidence if we want any hope of making our analysis reuse method efficient and accurate.

4.4.2 Function Matching

Our function matching algorithm is based on an edit-distance calculation, as shown in Algorithm 3. The algorithm is parameterized by a function CRITERIA that determines the distance between pairs of functions as a numerical score—we consider two functions to "match" when the criteria is below a certain threshold. We instantiated CRITERIA with different choices as shown in Table 4.1 in order to evaluate which combination of distance criteria worked best. Given the distances, the algorithm matches those functions with the best distance score that is under our empirically-calculated threshold.

Algorithm 3 works under the assumption that functions between program versions may be nested differently but generally still appear in the same order. We took inspiration from Revolver, a work which found success using a longest common subsequence algorithm to find similar pieces of malware among JavaScript programs [76]. Longest common subsequence is a specific instantiation of the more general problem of edit distance, and so we chose to design our matching algorithms around edit distance calculations—it plays a part in our block and instruction matching methods as well. Matches can be extracted from the algorithm's resulting table.

Function Similarity Scoring Criteria. Table 4.1 shows the different kinds of function criteria that we evaluate against. These are different combinations of differences based

Algorithm 3 Edit Distance Function Matching

procedure EDIT-DISTANCE-MATCH-FUNCTIONS($\overrightarrow{funcs_{curr}}, \overrightarrow{funcs_{upd}}$, CRITERIA)
Input: Functions from version P_{prior}, functions from version P_{upd}, and a function CRITERIA that scores functions based on similarity
Output: A list of pairs of functions that have been deemed similar
  matchingFunctions $\leftarrow \emptyset$
  $M \leftarrow$ PARAMETERIZED-EDIT-DISTANCE($\overrightarrow{funcs_{curr}}, \overrightarrow{funcs_{upd}}$, CRITERIA)
  Inspect the score matrix M, and populate *matchingFunctions* with the function pairs that were successfully matched
  return *matchingFunctions*
end procedure

Table 4.1: Different function matching criteria

Name	Criteria
Position-only	Distance between function IDs, distance between function line numbers
Instruction-only	Difference between number of instructions, (Size of the larger multiset of identifiers that occur in each function) – (Number of common identifiers occurring in both the functions)
Combined	Combination of Position-only and Instruction-only
Staged	Instruction-only, with ties broken by Position-only

on function position and based on instructions contained within the functions. We chose the two axes of position-based and instruction-based matching after manually inspecting the version differences among our benchmarks: quite often we discovered that the functions appeared in the same order, and that the instructions for the most part were identical. We also noticed that there were cases of the same function body appearing in multiple places, so that led us to combine the two areas in different ways to determine which combination was best. The criteria are in the form of distance functions, so a higher number indicates a larger difference. We combine features using the geomean.

4.4.3 Block Matching

Our chosen block matching algorithm also uses an edit distance calculation. Edit distance-based block matching is similar to Algorithm 3, and the blocks of two functions are matched using edit distance in the order in which they appear in the SAFE IR. This choice is based on the assumption that that changes to functions will not drastically affect the analysis results, and so matching as many blocks as we can will be helpful when we transfer old analysis computations. The best scoring blocks that match under a threshold are returned.

Block Similarity Scoring Criteria. The block criteria we chose is a combination of the following:

- Block type (Normal, Call, etc.),
- Instruction count difference, and
- (Size of the larger multiset of identifiers that occur in each block) - (Number of common identifiers occurring in both the blocks).

We save the corresponding blocks, because Call blocks are used in abstract state calling contexts.

4.4.4 Instruction Matching

For each matched pair of blocks, we once again perform a distance calculation. Instructions are matched based on the type of the instruction, the number of allocation sites appearing in the instruction, and the names of the variables involved (modulo the generated numerical suffixes). We save the corresponding allocation sites and variables that appear as the left-hand side of assignment instructions, as they will need to be remapped between abstract heaps.

4.4.5 Fixpoint Reuse

After a successful program matching effort we must use this information to remap results from FP_{prior} into the new $FP_{\widehat{upd}}$ before the incremental analysis begins its calculation. We keep this section high-level; the details are tedious but straightforward.

Figure 4.3 shows a high-level version of the data structures we work with. SAFE contains a map that keeps track of the saved abstract state for every ControlPoint, which is a (Block, Context) pair. Contexts keep track of which functions were called immediately preceding the current one, and can contain zero or more call sites, which are themselves Blocks (specifically, the Call blocks where the function call to the current function or one of its predecessors took place). For example, if we start out from the top level of our program, and call a function `foo`, the context will change from $\emptyset$ to the list $[\texttt{foo}]$, assuming our context sensitivity is greater than zero. So, we must remap each block using our saved correspondence mapping generated during the matching phase; it is for this purpose that we keep track of corresponding blocks.

For the abstract states, we must traverse and find any addresses and local variable names that we know how to map over. Local variables are held in a specific part of the abstract state, while abstract addresses are spread everywhere (though mainly exist in the heap and abstract JavaScript objects). It is for this purpose that we keep track of corresponding instructions.

4.5 Evaluation

In this section we evaluate the efficacy of fixpoint reuse in terms of performance and precision. Because we're guaranteeing the soundness of the incremental analysis, we must at a minimum visit every program point in the updated version at least once.

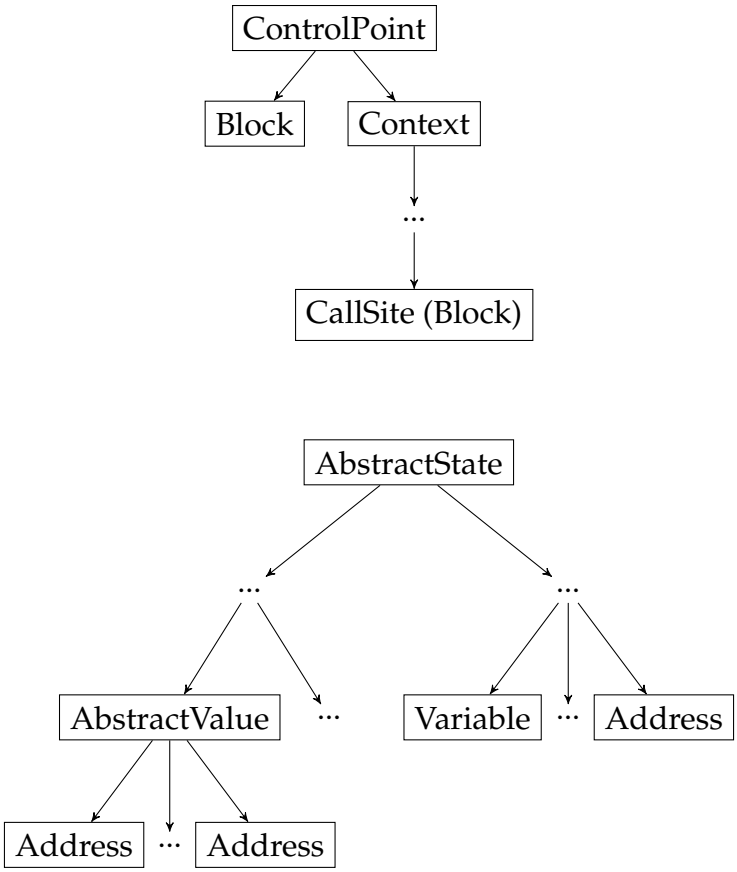


Figure 4.3: Reuse occurs inside of the ControlPoint and AbstractState data structures. This figure shows the general layout of these structures, and the leaves are things that we must map over between versions. For example, the abstract Addresses will often change with program versions.

Thus, the potential for speedup lies in reducing the number of times the analysis has to revisit a program point before convergence. The quality of the program matching between versions will play a large role.

We want to study the efficacy of fixpoint reuse on actual programs from the wild. We take four JavaScript-based browser addons and four Node.js programs along with between 1–4 updates for each program taken from available public repositories. These benchmarks are described in Table 4.2. These benchmarks were chosen from a set of similar programs because SAFE can completely model their code and analyze them using a reasonable amount of resources (we are limited in our benchmark selection by

Table 4.2: Open-Source Benchmarks. For every sequence of benchmark versions (e.g., [A, B, C]), we compare the closest pairs (i.e., (A, B) and (B, C)).

Benchmark Name	Version A	Lines	Version B	Diff	Distance
chess1 [4]	0.1.0.1	283	0.1.1.2	127+/116–	44
chess2	0.1.1.2	295	0.1.1.3	40+/10–	69
emoji-helper1 [8]	1.1.0	579	1.1.1	17+/3–	24
emoji-helper2	1.1.1	594	1.2.0	15+/1–	10
simple-translate [14]	2017.09.25	301	2017.10.14	2+/2–	0
k-cup-deals [11]	1.2	499	1.3	12+/0–	63
dateformat1 [7]	2011.03.13	166	2012.11.08	49+/7–	22
dateformat2	2012.11.08	208	2013.03.11	15+/8–	6
dateformat3	2013.03.11	216	2014.11.28	201+/55–	44
dateformat4	2014.11.28	261	2017.09.18	11+/6–	10
yallist1 [17]	2015.12.19	585	2017.03.11	24+/16–	8
yallist2	2017.03.11	594	2017.03.13	9+/0–	8
yallist3	2017.03.13	602	2017.04.25	2+/0–	0
balanced-match [3]	0.4.2	193	1.0.0	93+/102–	161
url-join1 [16]	2.0.0	149	2.0.1	1+/1–	0
url-join2	2.0.1	149	2.0.2	1+/1–	0

SAFE's capabilities). Following previous work on analyzing browser add-ons [77], we edit the original code to provide stubs for built-in browser functions, and we include some amount of driver code to ensure that the analysis visits all interesting locations in the source file. We manually selected sources and sinks for each file.

The actual analysis that we perform on these benchmarks is a taint analysis implemented using the SAFE JavaScript analysis infrastructure, suitably modified to implement fixpoint reuse. The implementation is available online. We use the taint results to measure the precision of the incremental analysis versus a from-scratch analysis.

To help calibrate expectations, we start with a limits study to determine the maximum speedup the incremental analysis could possibly get. We accomplish this by running the incremental analysis on "updated" benchmark versions that are exactly

the same as the original, thus ensuring a perfect program match and minimal revisiting of program points. The results are in Section 4.5.1.

Another factor that comes into play is how different the original and updated programs are. In the extreme, the updated program could be completely different from the original and not benefit from incremental analysis at all. To help understand the effect of program "distance", we have created a set of handmade benchmarks and a series of successively more "distant" updates for each benchmark, allowing us to study the effects of program distance in a controlled manner. The results are in Section 4.5.2.

Finally, we compare the speedups that we achieve on the actual updated program versions to determine how close to the optimal results we are. These results are in Section 4.5.3, and we study the sizes of the reused fixpoints in Section 4.5.4.

4.5.1 Limits Study

For our limits study we take each program version of each benchmark and run an incremental analysis on itself—in other words, we take the from-scratch analysis and apply fixpoint reuse to exactly the same program. This is the ideal case for reuse and provides the maximum benefit. Because we have a perfect program match, the only cost in the incremental case is for visiting each program point exactly once. We run three different experiments varying context-sensitivity from 0-CFA to 2-CFA; a "program point" for a context-sensitive analysis includes the context. The results are shown in Table 4.3.

4.5.2 Controlled Distance Study

Table 4.4 contains information on our handmade benchmarks: they are versions of the v8 Navier-Stokes (Table 4.4a) and the Richards (Table 4.4b) benchmarks with

Table 4.3: Best possible speedups. Exact times can be calculated using Table 4.6.

Benchmark	0CFA (×)	1CFA (×)	2CFA (×)
chess1	6.30	3.23	2.84
chess2	9.59	5.66	3.61
emoji-helper1	7.98	8.17	5.64
emoji-helper2	9.15	9.47	5.67
simple-translate	3.17	4.24	2.60
k-cup-deals	11.58	3.12	1.12
dateformat1	4.30	3.93	3.00
dateformat2	4.84	3.92	3.12
dateformat3	4.05	2.95	1.92
dateformat4	4.04	2.77	1.96
yallist1	9.85	13.05	11.93
yallist2	8.37	13.44	10.84
yallist3	9.44	13.36	11.55
balanced-match	7.92	12.58	14.46
url-join1	4.41	2.85	3.93
url-join2	4.68	2.93	4.30
Average	6.85	6.58	5.53

statements deleted. We made random (but attempted to avoid program-breaking) deletions—these files are then "played backwards" to appear as a sequence of code additions. Thus, successive versions contain greater and greater differences to the original version.

Our distance metric is derived from our program matching algorithm. Given two programs A and B, we compute the set of matching function pairs and, for each pair, we compute the block edit distance. The sum of the block edit distances over all matching function pairs is our measure of distance between A and B. We investigated several other possible distance metrics and found that they all behaved similarly.

We chose this methodology because additions seem to be the most common updates to code: in our real-world, open-source benchmarks, each commit contains over 4× the number of additions to deletions on average. Of the four outliers, only one was a

Table 4.4: Handmade Benchmarks

(a) Navier-Stokes (v8 lines of code: 398)

Versions (A–B)	Diff	Distance
v0–v8	35+/4–	68
v1–v8	32+/4–	62
v2–v8	27+/3–	53
v3–v8	20+/3–	43
v4–v8	17+/2–	39
v5–v8	11+/2–	22
v6–v8	8+/2–	11
v7–v8	2+/0–	1

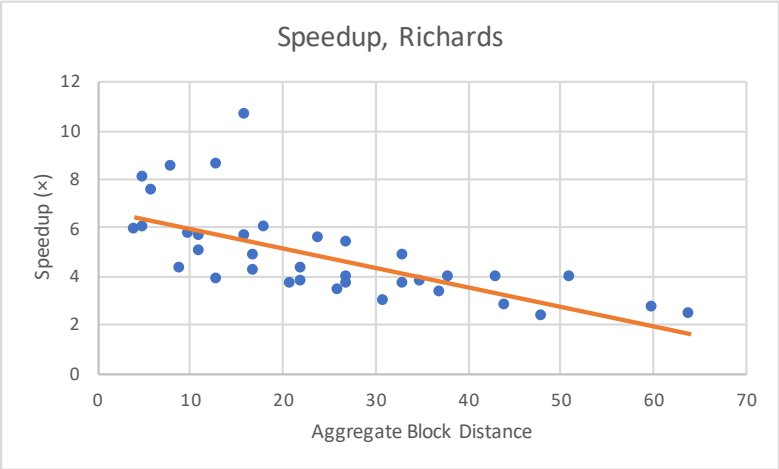
(b) Richards (v8 lines of code: 546)

Versions (A–B)	Diff	Distance
v0–v8	30+/2–	64
v1–v8	28+/2–	60
v2–v8	23+/2–	51
v3–v8	21+/2–	43
v4–v8	19+/1–	38
v5–v8	13+/0–	33
v6–v8	11+/0–	27
v7–v8	8+/0–	16

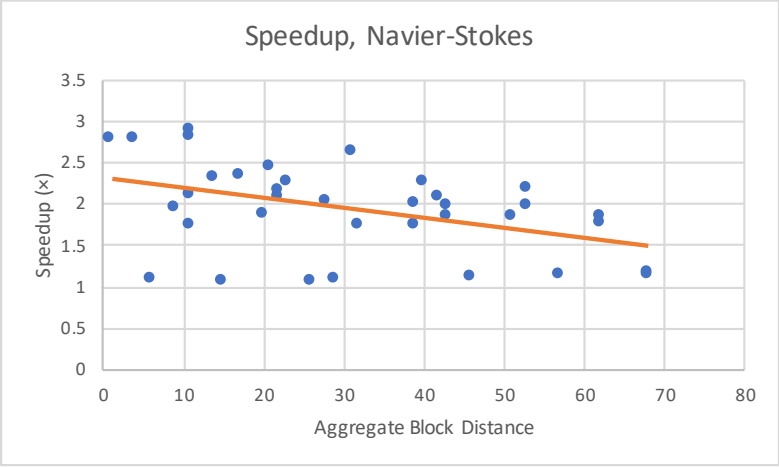
legitimate case of refactoring; others were superficial changes regarding whitespace or test suite configuration, and so these diffs were exaggerating the truth.

Figure 4.4 shows the results of our handmade benchmarks. We ran every combination of version pairs that respected the order, e.g., v1∼v2, v1∼v3, v1∼v4, v2∼v3, v2∼v4, v3∼v4, etc. We grouped each pair of programs based on their distance score. These 1CFA analysis results paint a picture of how program additions impact reuse.

Figure 4.4a shows the results for the Richards benchmarks. This benchmark consists of several small functions. For the original benchmark, the fixpoint took 9,081 iterations to converge, there were 494 unique program points visited, and there were 3 loops.



(a) Richards



(b) Navier-Stokes

Figure 4.4: Handmade Results

Figure 4.4b shows the results for the Navier-Stokes benchmarks. This benchmark consists of a small number of large functions. The original benchmark's fixpoint took 7,060 iterations to converge, there were 562 unique program points visited, and there were 26 loops. For both benchmarks, the chosen taints were calculated precisely for all version pairs.

Both sets of benchmarks tend to degrade in performance as the difference between version pairs increases, but the Richards benchmark appears to be more amenable to reuse and therefore has more to lose as the distance increases. Given the statistics in the previous paragraph, compared with the Navier-Stokes benchmark set, the Richards benchmark set has more iterations to save by reusing information, fewer program points to visit at least once, and significantly fewer loops through which any updated information must be propagated. This experiment helps to give insight into which kinds of programs see better fixpoint reuse performance, while also highlighting that some amount of improvement is usually possible as long as the changes are not too drastic. Even in the face of unrecoverable program differences (see the line of dots hovering above the $1\times$ speedup in Figure 4.4b), for these benchmarks our method does not do worse than a from-scratch analysis.

Unmatched instructions

Figure 4.5 provides insight into the abilities of our matching algorithm by showing the total number of instructions that could not be matched across all the different versions of the Richards handmade benchmarks. The Navier-Stokes results are similar.

As the textual difference between programs increases, it becomes more difficult to match functions and blocks—this difficulty culminates in the algorithm's inability to match individual instructions inside of blocks. The larger the gap between handmade benchmark versions, the more changes exist in the code, and the number of unmatched

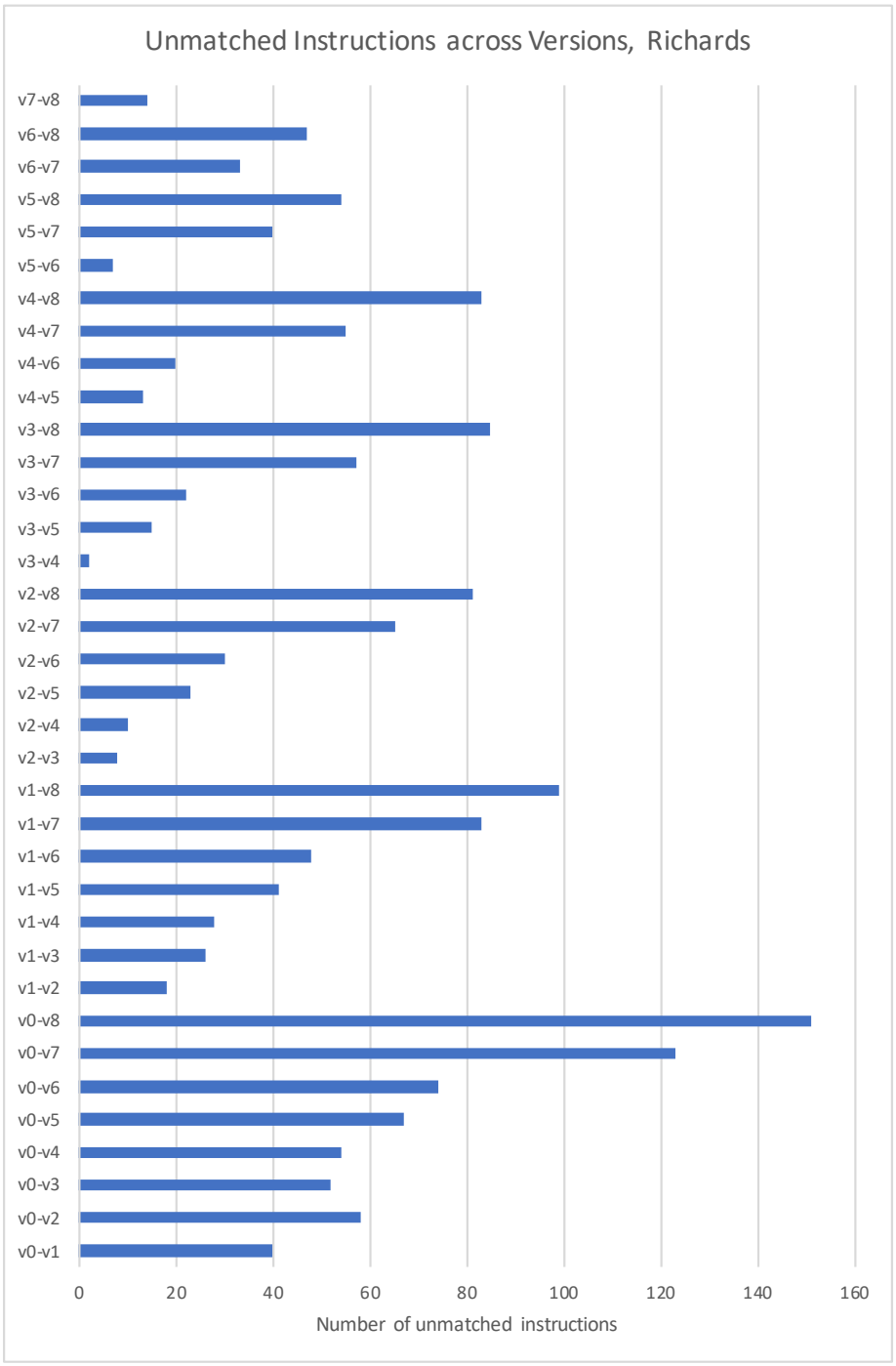


Figure 4.5: Number of unmatched instructions across the Richards handmade benchmarks.

Table 4.5: Correspondence map creation times.

Benchmark	Time (s)
chess1	1.03
chess2	1.07
emoji-helper1	4.70
emoji-helper2	4.82
simple-translate	1.18
k-cup-deals	2.24
dateformat1	0.82
dateformat2	0.75
dateformat3	0.77
dateformat4	0.84
yallist1	3.92
yallist2	4.27
yallist3	4.33
balanced-match	0.42
url-join1	0.55
url-join2	0.55

instructions reflects this. Matching is not exact, and so there is one outlier in the v0–v2 version pair, but the results show that our matching method keeps unmatched instructions to a minimum and performs well in the vast majority of cases.

For the instructions that could be matched, there is still a possibility that they were matched incorrectly—discovering errors in instruction matching would require manual effort, but we believe the lack of extra taints demonstrates that our edit distance-based matching is precise.

4.5.3 Real-World Evaluation

We run the real-world version updates at three context-sensitivity levels. The time it takes to perform the program matching process on a given version pair is the same for every context sensitivity level. Table 4.5 shows the times, and they are all quite small. For the longer-running analyses this number is completely negligible.

Table 4.6: Baseline results.

Benchmark	0CFA		1CFA		2CFA	
	Time (s)	Taints	Time (s)	Taints	Time (s)	Taints
chess1	39.46	3	22.20	3	24.28	5
chess2	71.77	3	39.05	3	39.01	4
emoji-helper1	213.90	1	135.78	1	84.55	1
emoji-helper2	251.39	1	150.80	1	95.04	1
simple-translate	13.60	1	17.34	1	14.29	2
k-cup-deals	86.89	1	30.58	1	20.45	1
dateformat1	52.28	4	56.71	8	50.92	8
dateformat2	67.06	4	60.39	8	55.52	8
dateformat3	66.60	4	42.74	7	34.34	7
dateformat4	68.92	4	44.25	7	35.76	7
yallist1	843.39	7	1443.23	60	2084.17	60
yallist2	857.51	7	1432.67	60	2033.51	60
yallist3	823.98	7	1429.20	60	2107.08	60
balanced-match	249.31	20	523.17	420	2397.87	420
url-join1	41.11	2	29.83	17	112.95	17
url-join2	41.60	2	30.42	17	114.35	17

Baseline results

Table 4.6 shows the results for running the static analysis on the updated version of each benchmark from scratch (i.e., with fixpoint reuse turned off). The number of taints output is the sum of all tainted sources for a given tainted sink state—note that states can become duplicated when context sensitivity increases, and that is why the number increases.

Incremental Results

Table 4.7 shows the results of reuse for each different context sensitivity level. We find that all taints are carried over with very little imprecision. The dateformat benchmark is the only case with imprecise taints, and this is due to the modeling of a JavaScript built-in object that causes the analysis to return the $\top_{\text{addr}}$ address (i.e.,

Table 4.7: Results, relative to from-scratch analysis. Exact times can be calculated using Table 4.6.

Benchmark	0CFA		1CFA		2CFA	
	Speedup	Taints	Speedup	Taints	Speedup	Taints
chess1	1.28	3	1.01	3	0.97	5
chess2	1.61	3	1.08	3	0.97	4
emoji-helper1	4.70	1	4.35	1	3.40	1
emoji-helper2	7.07	1	7.37	1	4.87	1
simple-translate	3.14	1	4.15	1	2.63	2
k-cup-deals	4.32	1	1.46	1	0.93	1
dateformat1	1.47	4	1.02	8	0.90	8
dateformat2	2.59	4	1.60	8	1.44	8
dateformat3	2.19	4	0.90	8	0.91	8
dateformat4	4.13	4	2.26	8	1.48	8
yallist1	1.18	7	1.03	60	1.07	60
yallist2	1.23	7	1.42	60	1.48	60
yallist3	9.36	7	14.41	60	13.16	60
balanced-match	0.88	20	0.97	420	1.02	420
url-join1	4.23	2	2.56	17	3.93	17
url-join2	4.27	2	2.60	17	3.99	17
Average:	3.35		3.01		2.70	

the abstract address corresponding to all concrete addresses). Because the heap is prepopulated with extra information, there are more locations to point to than in the from-scratch case.

All in all, while maintaining soundness and high precision in a proof-of-concept taint analysis, our fixpoint reuse method allows us to more than double the speed of an analysis on average for real-world programs.

For another perspective, Table 4.8 shows our speedup relative to our best possible incremental analysis results (i.e., the observed speedup divided by the optimal speedup). These results provide another means of observing program difference: the version pairs with the fewest differences have either an optimal or close-to-optimal speedup. Due to natural variation in analysis times, some results were slightly above

Table 4.8: Speedup results, relative to a perfect incremental analysis. Exact times can be calculated using Table 4.6.

Benchmark	0CFA	1CFA	2CFA
chess1	0.20	0.31	0.34
chess2	0.17	0.19	0.27
emoji-helper1	0.59	0.53	0.60
emoji-helper2	0.77	0.78	0.86
simple-translate	0.99	0.98	1.00
k-cup-deals	0.37	0.47	0.83
dateformat1	0.34	0.26	0.30
dateformat2	0.54	0.41	0.46
dateformat3	0.54	0.31	0.48
dateformat4	1.00	0.82	0.76
yallist1	0.12	0.08	0.09
yallist2	0.15	0.11	0.14
yallist3	0.99	1.00	1.00
balanced-match	0.11	0.08	0.07
url-join1	0.96	0.90	1.00
url-join2	0.91	0.89	0.93
Average:	0.55	0.51	0.57

1.0—we capped those results at 1.0 to paint a more accurate picture. On average, our reuse method is within a factor of two of the optimal speedup for these benchmarks, and we believe this is representative of the general case.

4.5.4 Size of Saved Fixpoints

Figure 4.6 shows the sizes of each fixpoint after compressing with the `gzip` compression utility. The longer-running benchmarks compute more information during the analysis, and are therefore larger. An increase in context sensitivity also causes more information to be saved, and this increases the size of each fixpoint as well. As these files can get large, an interesting area of future investigation could be investigating how to take advantage of the semantic struture of fixpoints to aid further compression.

4.6 Background

In this section we provide some background on abstract interpretation and taint analysis.

4.6.1 Abstract Interpretation

Abstract interpretation [48] discovers invariants about a program by running it in an abstract way. For example, assume that the computation of $2 + 3$ was too difficult to compute—one way of making this easier computationally is to get rid of the numbers, and instead calculate *positive* + *positive*—a table lookup would provide the final answer of *positive*. We can capture this process with a function $\alpha : \mathbb{Z} \to \mathit{PosNegZero}$, where *PosNegZero* is the powerset of the set $\{\mathit{positive}, \mathit{negative}, 0\}$; it is an example of an *abstract domain*.

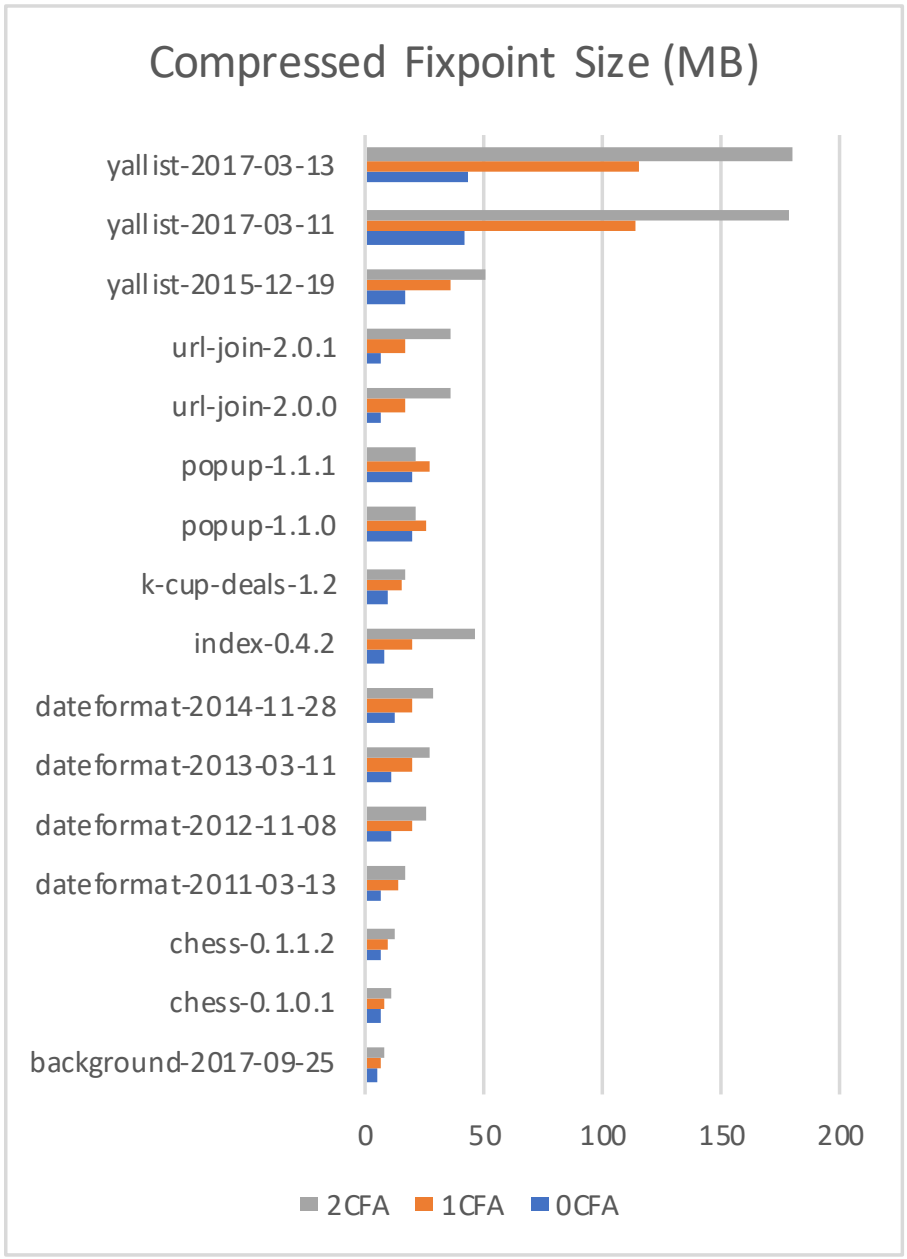


Figure 4.6: Size of gzip-compressed fixpoints in megabytes.

positive, however, is not the same as 5—it is an *abstraction*. Information is clearly yet deliberately lost when we move to an abstraction, and there is much work on the fine line between giving up too much information and just enough to solve the original problem. Information is ordered in the logical way: $\{positive\}$ is clearly more precise than $\{positive, 0\}$, so we write this fact as $\{positive\} \sqsubseteq \{positive, 0\}$ in this new abstract sense, so less precise things are abstractly *bigger* than more precise things—intuitively, such imprecise values "believe" that they are more things.

Programming languages are more complicated than simple arithmetic expressions: we must figure out how to "abstractly" run the computations involved during function calls and complicated loops and numerical operations. And, we must do all this while preserving the key property that all program analyses must have: the analysis must halt, even if the program does not. We want some information about a program, and we accept that we must settle for some information loss, but we don't want to wait a lifetime to get the final answer. A more complicated analysis that is essential for JavaScript analysis is called control-flow analysis: instead of generating values of variables, it creates a control flow graph for the analyzed program.

Fortunately, we are not forced to reinvent the wheel to implement a program analysis. One piece of machinery that gets us most of the way there is an interpreter for a programming language—it has most of the features that we desire, but it is too precise. We can implement an abstract interpreter on top of a regular (the technical term is *concrete*) interpreter fairly easily (see, e.g., [144]) after one has chosen the abstract domains to use.

Interpreters (and therefore abstract interpreters) often have a lot of moving parts—especially for a language like JavaScript, where values can be of many different types all at once, functions are first-class objects, inheritance is modeled with prototypes, etc. Abstract heaps are often used to keep track of data that has been allocated, and there is

also a stack of functions that have been called and pointers to where they should return to. None of this information goes away during the transition to an abstract interpreter.

Nondeterminism often plays a large role in abstract interpretation, due to lost precision—this can cause loops to arise where they would not appear otherwise. As an example of lost precision and nondeterminism, consider the following program statement:

```
if (1 <= 2)
  X();
else
  Y();
```

A concrete interpreter would immediately conclude that only the true branch will ever be executed, but consider the abstract interpreter with the *PosNegZero* numerical abstract domain from above: it would instead ask itself about the truth or falsity of the expression *positive* $\leq$ *positive*, and the correct answer is "I don't know", since `4 <= 3` looks exactly the same as `1 <= 2` from its viewpoint. Thus the abstract interpreter returns `true` and `false` for this expression, runs *both branches*, and must find a way to *combine* the information that it calculates for each branch to use as the result of the entire `if/else` expression.

This combination of imprecise values is key to our analysis reuse: maybe the program changes, and `X()` does something completely different while `Y()` remains the same—could we use our old information about the abstract result of `Y()` again? Our method revolves around populating abstract states with previously-computed values.

A term that appears often in this chapter is *context sensitivity*, which describes how an analysis keeps track of where functions that are called should return. It is infeasible (and in programs with call stacks of indeterminate depth, impossible) to keep track of the entire call stack in an abstract interpreter, and so many analyses settle for saving only the very top of the stack. Saving one function on the top of the call stack is abbreviated as 1CFA, two is 2CFA, and saving nothing is called either 0CFA or *context*

insensitive. These different context sensitivities make a difference in the precision and performance of an analysis, and this difference translates into the realm of reuse that we explore.

Abstract interpreters often run what is called a worklist algorithm, which keeps track of which locations in the program to visit and calculate abstract results for—initially, the first line of the program is inserted as the initial state, because the analysis has not yet saved any information about its execution. The analysis halts when the information it knows about each state does not change. When the analysis stops and the information converges, we say that it has reached a *fixpoint*. We refer to this conglomerate of saved information about each abstract state as the "fixpoint" of the entire analysis.

If we give an analysis some starting information from a previous program version, it has the potential to terminate more quickly, though there is the risk of having too much imprecision. That is the challenge we overcome in this chapter.

One important property that a analysis should have is soundness: a sound analysis overapproximates the results of any concrete run of a program and does not leave out a possible behavior—we ensure that our analysis reuse method retains the soundness of the underlying analysis.

4.6.2 Taint Analysis

As the name indicates, a program taint is something that is undesirable. Taints have to do with data dependence, i.e., which values get where. For example, consider the following code:

```
var x = 5;
var y = x + 7;
var z = y * 8 + 42;
```

We say that `x` *flows to* `z` (and also `y`). If the number `5` was a confidential value that we

did not wish to pass around to too many places, we would say that it is a tainted *source* and try to track where this information *flows*. Instead of propagating actual numbers or positivity information, this analysis tracks information about the taintedness of a given expression.

Taint tracking is a common program analysis, and the goal is to find whether any tainted sources reach any undesirable program locations, or *sinks*. A more high-level example of a source and a sink is that of a system password and a network request.

To evaluate our analysis reuse method, we implement taint analysis on top of an existing JavaScript abstract interpreter, and we use this analysis to evaluate the precision of our worklist reuse algorithm. We want the reused analysis to not return too many extra sinks that an analysis that started from scratch would not return, for some definition of "too many".

This may sound too good to be true, so we briefly provide some insight on why this goal of precise reuse without any pruning is attainable. Consider once more the example of the imprecise `if` statement from the previous subsection, and imagine again that `X()` is changed drastically while `Y()` remains the same. We will carry over the taints from `Y()` in the reused analysis, since it has an exact correspondence in the new program, but we will also remember any taints that occurred inside `X()`, and at first glance this may be interpreted as a bad thing. But, the key point is that we assume that the old program was "accepted" or already considered safe, so any "incorrect" taints will in fact not pollute the output. Our evaluation confirms this.

Chapter 5

Semantic Clone Detection

5.1 Introduction

Clone detection, the process of discovering duplicated and/or similar code across a code base, is an important software engineering problem to solve in order to help prevent code bloat and code redundancy, and the attendant issues that these raise—for example, failing to propagate a bug fix in one part of the code to other parts of the code that are (near-)duplicates of the first. The term "similar code" is vague; to make the concept more precise the community has categorized *code clones*, that is, sets of code fragments that are considered similar, into different types based on how "similarity" is defined [125]:

- **Types I, II, and III.** Similarity of code fragments is defined in a purely syntactic way, either in terms of program text or of syntax trees. As the numbered Type increases, so does the generality of the Type definitions.

- **Type IV.** Code fragments are Type IV clones if they perform similar functionality, compute a similar solution, or operate in a similar manner. Type IV clones are also known as *semantic clones*. They ignore syntax, and so they are not necessarily a superset of Type III clones, or have any particular relation at all to the sets of Type

I, Type II, or Type III clones. We focus on Type IV clones in this chapter.

This chapter proposes a new method for semantic clone detection, the least studied and most difficult to find class of code clones [125]. A simple example of the type of clones we detect are two distinct implementations of different sorting algorithms, e.g., insertion sort and bubble sort. The implementations of these two sorting routines look completely different syntactically, but semantically they accomplish the same task and hence are Type IV clones.

Prior work on semantic clone detection usually uses program dependence graphs (PDGs) to expose non-syntactic similarities between code fragments [84, 88, 94, 61]. The intuition behind those techniques is that similar data- and control-dependencies imply similar semantic behavior. Sometimes this heuristic is borne out, but other times it fails to identify clones that are clearly semantically identical (and sometimes fails to distinguish non-clones), as shown in our evaluation in Section 5.4. Our key insight is a complete departure from that previous approach; our intuition is that **semantically similar programs ask similar questions with similar frequencies**. This intuition gives rise to a novel heuristic for identifying semantic clones, which we show in our evaluation to work well in practice.

Our heuristic for detecting semantic clones is to compute the probability distribution over individual paths in a given code fragment,[1] and then judge whether two code fragments are semantic clones by comparing their path distributions to see if they are "close enough". We work in the style of Probabilistic Symbolic Execution [63]. Paths are defined by the conditions that are tested at various branching points in the code, i.e., the "questions" that the code asks. The distribution over those paths indicates the frequency that the code asks these questions. If the distributions are similar for the two

[1] Under certain assumptions as detailed later in this chapter, such as restricting infinite domains to a finite interval.

code fragments, then the fragments are asking the same kinds of questions with the same frequencies, and hence are likely (according to our intuition, and borne out by our evaluation) to be semantic clones.

To compute this heuristic we employ a symbolic execution engine to derive the set of paths that a code fragment can execute; we then employ a *model counter* for each derived path to obtain the paths' relative frequencies. A model counter takes a logical formula (such as a path condition from the symbolic execution engine) and returns the number of satisfying solutions for that formula. As an example, consider the two code fragments in Figure 5.1. Figure 5.1a and Figure 5.1b contain programs that are syntactically different yet semantically identical. For each code fragment there are two execution paths based on the value of variable `x` that are equally likely, as shown by the distributions in Figure 5.1c and Figure 5.1d. Hence our heuristic categorizes these code fragments as semantic clones.

On the other end of the spectrum, consider the two code fragments in Figure 5.2. In Figure 5.2a and Figure 5.2b we have programs that are semantically different yet syntactically similar. For Figure 5.1a we count the number of values of `i` that satisfy the constraint `i == 0` (obtaining 1), and then we do the same for the else branch's constraint `i != 0` (obtaining $2^{32} - 2$, assuming 32-bit integers). Assuming a uniform distribution over the input space, there is a sharp contrast between the branch distributions of these two programs, as shown in Figure 5.1c and Figure 5.1d. Hence our heuristic categorizes these code fragments as non-clones.

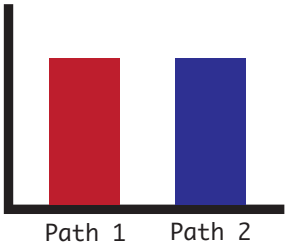
The implementation of our proposed technique operates at the granularity of functions: it forms discrete probability distributions over function execution paths and then compares these distributions using statistical techniques. If the resulting metric exceeds a given threshold then the two functions are considered clones, otherwise they are considered non-clones. The specific contributions of this chapter are the following:

```
if (x % 2 == 1)
  odd++;
else
  even++;
```

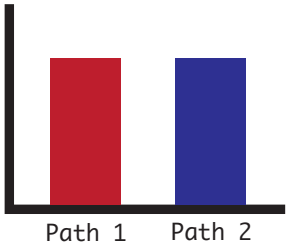
(a) Testing the parity of `x`. Assuming every value of `x` occurs equally often, both branches are taken with equal probability.

```
while (x & 1) {
  odd++;
  even--;
  x = 0;
}
even++;
```

(b) A semantically identical, syntactically different version of Figure 5.1a.



(c) The distribution of path frequencies for Figure 5.1a.



(d) The distribution of path frequencies for Figure 5.1b.

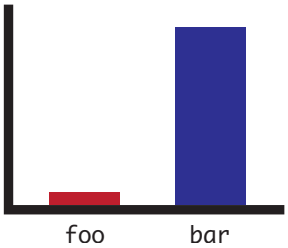
Figure 5.1: An example of the usefulness of model counting: the distributions of the branches in each code snippet are the same.

```
if (i == 0)
  foo();
else
  bar();
```

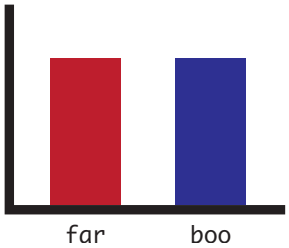
(a) Choosing between two function calls. Assuming every value of `i` occurs equally often, `bar()` gets executed more often than `foo()`.

```
if (i < INT_MAX / 2)
  far();
else
  boo();
```

(b) A syntactically similar, semantically different version of Figure 5.2a. Under similar assumptions, `far()` is executed equally as often as `boo()`.



(c) The distribution of path frequencies for Figure 5.2a.



(d) The distribution of path frequencies for Figure 5.2b.

Figure 5.2: An example of the usefulness of model counting: the distributions of the branches in each code snippet are vastly different.

- We present a novel clone detection method that is more sensitive to semantic differences than existing work (Section 5.3). Both this proposed method and existing work are imperfect heuristics; we believe that in certain cases (discussed in our evaluation) this method is complementary to existing methods—that is, they can be useful together.

- We provide an evaluation of our method and show that we detect semantic similarities and differences that existing methods cannot (Section 5.4).

- We make our source code and benchmarks freely available to the research community for modification and improvement.

Before describing the details of our technique, we provide a high-level overview and example to facilitate understanding and intuition (Section 5.2). We put our work into context by examining the related work in Section 5.5.

5.2 Overview and Example

In this section we provide a high-level walkthrough of our method by means of an example. Any technical details not fully explained here have been saved for the following section. To begin, we discuss our definition of semantic clones.

5.2.1 Our Definition of Semantic Clones

Semantic clones (also known as Type IV clones) in the literature are commonly defined in an abstract and informal way [88, 127, 128, 83, 114], and usually a sentence or two is used to describe them. For the most part, prior work has settled on the idea that a semantic clone is a pair of program snippets that perform the same functionality

while possibly appearing textually different. More advanced definitions are given in the form of Simions [71] and Code Relatives [137]—with Code Relatives being the more technical of the two—but these definitions are not applicable to all methods of semantic clone detection.

Instead, we interpret semantic clones using program execution traces as a basis (which contain, e.g., the values of variables, the current program point that is being executed, the contents of the stack, etc.) and imagine semantic similarity as matching certain feature sets extracted from those traces. For a specific instance of semantic clone detection, a researcher can supply a scoring function to a static or dynamic program trace, and this interpretation is general enough to cover our method as well as competing methods; we believe that it provides enough generality to represent most semantic clone detection methods.

5.2.2 The Programs

Our running example will use the programs displayed in Figure 5.3. The first two are both sorting functions: Figure 5.3a is an implementation of insertion sort and Figure 5.3b is an implementation of bubble sort. The end goal of each algorithm is the same, but the method by which the two programs achieve this goal is quite different; therefore, these two programs are exact semantic clones of one another, and we would like to algorithmically discern this fact.

Figure 5.3c contains code for the longest increasing subsequence algorithm, which tries to find a sorted subsequence inside an array. We claim that, while it is not an exact semantic match to the above sorting algorithms, this code fragment is semantically similar in the sense that it is concerned with the property of *sortedness*—this should be visible through an appropriate comparison of program execution traces, and thus

this fits our definition of semantic clones. We would like our semantic clone detection algorithm to discover this similarity.

Figure 5.3d contains an iterative version of binary search on integer arrays. This program is quite different from the others because it searches through an already sorted array and does not compare array elements to each other, thus we claim that it is *not* semantically similar to the previous code fragments. We would like our similarity detection algorithm to distinguish between this code fragment and all of the other fragments.

The first step of our method involves symbolic execution, which is necessary to understand the different ways that the programs interact with their data.

5.2.3 Exploring Execution Paths

We first run Symbolic Pathfinder (SPF) [111], a state-of-the-art symbolic execution engine for Java, on each program. Symbolic execution allows us to execute each program in all possible ways, up to a given bound. We must bound the data as well as the control paths, for example, we arbitrarily bound array sizes to three in this example.

Symbolic execution attempts to discover every feasible control path in the program (i.e., every possible path that a program can possibly take in a concrete execution); therefore the symbolic execution engine collects *path conditions* along the way—these are, in essence, a justification for why a particular path was feasible. A path condition is a conjunction of logical formulae that explains the decisions that a particular program's execution took with respect to branches, and they form the basis for our similarity detection algorithm. Those path conditions are sent to a solver whenever they are updated, and the engine only continues exploring a given path if its path condition is satisfiable. We configure SPF to use the Z3 [52] SMT solver.

```java
public static void insertSort(int[] A){
  for(int i = 1; i < A.length; i++){
    int value = A[i];
    int j = i - 1;
    while(j >= 0 && A[j] > value){
      A[j + 1] = A[j];
      j = j - 1;
    }
    A[j + 1] = value;
  }
}
```

(a) **Insertion sort**

```java
public static void bubbleSort(int[] comparable) {
    boolean changed = false;
    do {
        changed = false;
        for (int a = 0; a < comparable.length - 1; a++) {
            if (comparable[a] > comparable[a + 1]) {
                int tmp = comparable[a];
                comparable[a] = comparable[a + 1];
                comparable[a + 1] = tmp;
                changed = true;
            }
        }
    } while (changed);
}
```

(b) **Bubble sort**

```java
static int lis(int arr[], int n) {
  int lis[] = new int[n];
  int i,j,max = 0;
  /* Initialize LIS values for all indexes */
  for ( i = 0; i < n; i++ )
    lis[i] = 1;
  /* Compute optimized LIS values in bottom up manner */
  for ( i = 1; i < n; i++ )
    for ( j = 0; j < i; j++ )
            if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
        lis[i] = lis[j] + 1;
  /* Pick maximum of all LIS values */
  for ( i = 0; i < n; i++ )
    if ( max < lis[i] )
        max = lis[i];
  return max;
}
```

(c) **Longest increasing subsequence**

```java
public static int binarySearch(int[] nums, int check) {
    int hi = nums.length - 1;
    int lo = 0;
    while (hi >= lo) {
        int guess = (lo + hi) / 2;
        if (nums[guess] > check) {
            hi = guess - 1;
        } else if (nums[guess] < check) {
            lo = guess + 1;
        } else {
            return guess;
        }
    }
    return -1;
}
```

(d) **Iterative binary search**

Figure 5.3: Three semantically similar programs (a, b, c) and one control program that is semantically different from the others (d).

We run the programs being compared through SPF and ask for the resulting path conditions; a representative path condition from the insertion sort function is:

```
Path Condition: constraint
   a_1[1] > a_2[0] &&
   a_0[2] > a_2[0] &&
   a_0[2] > a_1[1] &&
   [I@15b_length[0] >= CONST_0
```

Even at this level, the path conditions for insertion sort and bubble sort turn out to be exactly the same. Sorting algorithms take a different path depending on the arrangement of the data in the provided array; the path conditions show that at the end of their execution, both insertion sort and bubble sort will have *asked the same questions* about their data. The methods by which the two programs obtained their information and effected their branching are quite different from one another—insertion sort is much more efficient, after all—but their discoveries were the same in the end.

As for the longest increasing subsequence path conditions, the similarities are not yet obvious, and so we must generalize our method if we hope to compare any two given functions. We choose to employ *model counting* in order to obtain a representation for program paths that may be readily compared with one another.

The iterative binary search path conditions are quite different from the rest, with many having completely different numbers of paths, and we expect that the final model counts will demonstrate this fact as well.

5.2.4 Model Counting Path Conditions

In order to detect semantically similar (but not necessarily identical) code fragments, we must abstract the path conditions for each fragment into a form that can be readily

{17341665826650, 17867170851700,
1751511700, 17867170851700,
17867170851700, 1804057051}
(a) **Insertion sort**

{17341665826650, 17867170851700,
1751511700, 17867170851700,
17867170851700, 1804057051}
(b) **Bubble sort**

{17341665826650, 17867170851700,
1751511700, 1751511700,
17867170851700, 18403185977251}
(c) **Longest increasing subsequence**

{338350, 166650, 5050, 166650, 338350,
5050, 101}
(d) **Iterative binary search**

Figure 5.4: Model counts for each path condition.

compared. We choose to abstract each set of path conditions as a discrete probability distribution. This abstraction obviously loses information about the individual path conditions; however, we show in our evaluation that this choice for abstraction works well in practice.

We use the Barvinok model counter [145] in order to obtain the number of satisfying solutions to constraints over integers. We focus on integer constraints in this chapter, but there exist other model counters that handle other kinds of constraints (e.g., strings [28]). Our method can be applied to any type of countable constraint without requiring any changes.

For this example we bound the integers to range from 0 to 100 because unbounded integers would produce a count of ∞, which is not useful. We explain and justify our bounding assumptions in Section 5.3. After placing our constraints in the format that Barvinok accepts, we obtain the counts shown in Figure 5.4.

Each integer is the number of satisfying solutions to the path condition that it represents. Again, we can see that insertion sort and bubble sort are equivalent, while

there is a slight difference between the two sorting functions and the longest increasing subsequence function. There is a large difference between iterative binary search and the others in terms of magnitude, but perhaps the relative probabilities tell a different story. In order to compare these counts and return a final similarity score, we convert these counts to distributions.

5.2.5 Path Distributions

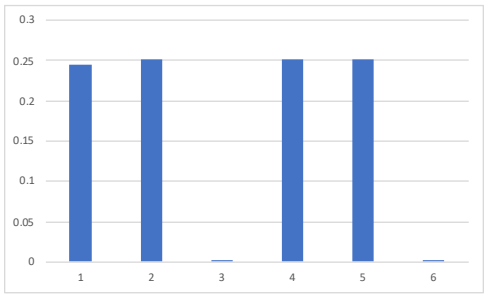
If we assume, as is customary [115, 32], that all paths and values are equally likely, it is simple to convert the path counts to path probabilities [63]. We will justify these assumptions and others in Section 5.3; for now it suffices to say that these same assumptions are made across all programs, so each program is on a level playing field. After performing this conversion we are left with the discrete probability distributions shown graphically in Figure 5.5.

Again, insertion sort and bubble sort remain identical. Longest increasing subsequence is quite similar to the other two, but not identical, and we would like the ability to quantify this difference. Iterative binary search also has some clear differences in this graphical form. As we now have obtained probability distributions, one of our insights is that we may now compare them using standard statistical techniques.

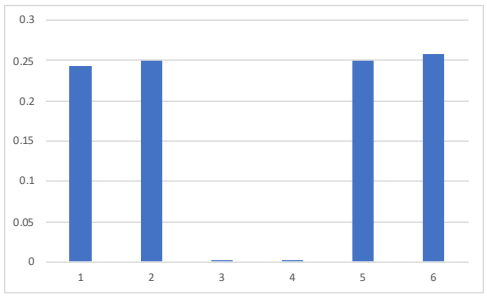
5.2.6 Comparing Distributions

We save the specifics of our score calculation for the next section; it suffices to say here that we use a statistical divergence algorithm to obtain a number quantifying the difference between two discrete distributions. Our final results are shown in Table 5.1, along with how long it took for the scores to be calculated.

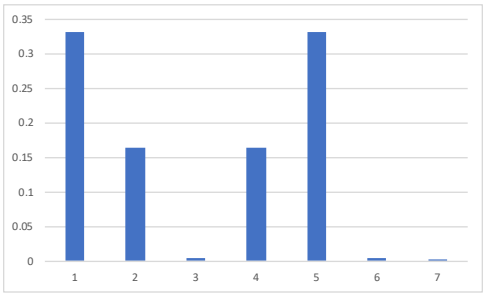
The score is an unbounded divergence value that represents how different the two



(a) **Insertion and Bubble sort**



(b) **Longest increasing sequence**



(c) **Iterative binary search**

Figure 5.5: Path distributions obtained from the path counts. Insertion and bubble sort are identical and have been conflated into the same graph.

Table 5.1: Final scores and timing data for the example programs.

Program Pair	Score	Running Time
(Insertion sort, Bubble sort)	0.000000	8.14 s
(Insertion sort, Longest increasing subsequence)	0.000119	8.03 s
(Insertion sort, Iterative binary search)	1.335384	5.92 s

programs are. While the scores have no absolute cutoff between similar and different, they are meaningful relative to one another. It is clear that insertion sort and bubble sort are deemed to be exactly the same, and insertion sort and longest increasing subsequence are calculated to be quite similar. For our control example we achieve a much larger score, i.e., insertion sort and binary search are judged to be different from each other. All scores are calculated in a matter of seconds.

Having peeked into the inner workings of our method at a high level, we fill in the gaps and missing technical details in the following section.

5.3 Our Method in Detail

5.3.1 SPF Specifics

We use Symbolic PathFinder (SPF) [111] to symbolically execute programs in order to obtain path conditions. SPF has certain requirements that we must fulfill:

- We must create a configuration file to manage each symbolic execution run. Inside this file we configure the solver to be Z3 [52] and we set the symbolic method (i.e., what part is to be symbolically executed) to be the function that we are comparing for semantic similarity.

- We must create driver code to exercise each function that we are comparing for semantic similarity. For example, the driver code for our insertion sort example from Section 5.2 contains the `main` function shown in Figure 5.6. This code creates an array of concrete values and fills it with symbolic integers. The last thing the driver does is print the current path condition. Because every execution path in the program is visited, this one statement is visited many times (once for each path). Thus, every path condition is shown in SPF's final output.

```
public static void main(String [] args) {
    int[] a = {8, 6, 7};
    for (int i = 0; i < a.length; ++i) {
        a[i] = Debug.makeSymbolicInteger("a_" + i);
    }
    insertSort(a);
    Debug.printPC("\n Path Condition: ");
}
```

Figure 5.6: An example of the kind of driver code required by SPF.

```
P := { [x1,x2] :
    0 <= x1 <= 100 and
    0 <= x2 <= 100 and
    x1 >= 2 and
    x1 <= 2
};
card P;
```

Figure 5.7: A sample Barvinok query file.

Assumptions. We set the min and max symbolic integers to be 0 and 100, respectively. Small ranges such as this are commonly used in SPF [19] to make the symbolic execution tractable, and as long as all programs are treated the same the specific range chosen makes little difference to our method.

5.3.2 Barvinok Specifics

We parse the output from SPF and perform a syntactic transformation to turn it into a form that the Barvinok model counter [145] can understand. A sample Barvinok query file is shown in Figure 5.7. This file defines a two-dimensional polytope P constrained by four inequalities. The `card` keyword in the `card P;` line stands for cardinality, and this expression asks Barvinok to count the number of integer solutions to the given constraints. We perform this process for every path condition output by SPF.

After obtaining the counts for each path condition for a pair of programs, we have the beginnings of two discrete probability distributions that we can compare using statistical methods.

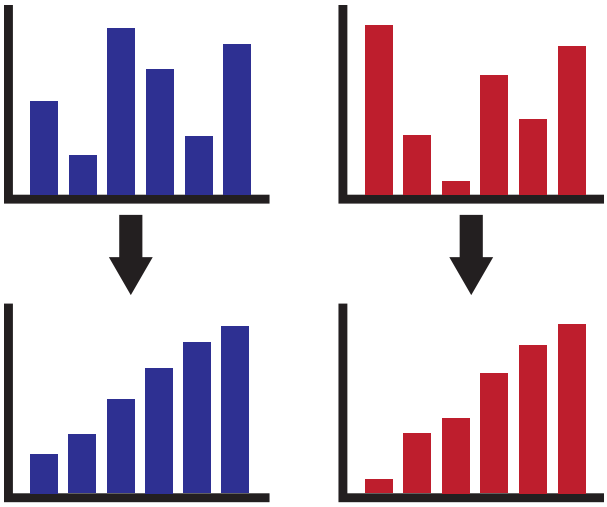


Figure 5.8: An example of sorting the discrete distributions to aid in alignment.

5.3.3 Distribution Comparison

Assumptions. In order to create proper probability distributions from a set of path counts, we must create percentages from each count. We make a commonly-used simplifying assumption here: that every input to the original function (within the chosen bounds) is equally likely. We make this choice because we cannot assume that we have access to the exact probabilities for program inputs. If a user were able to provide those probabilities for every program our method would likely see increased precision. Our assumption allows us to divide each individual count by the total count in order to generate a distribution that sums to 1.0.

To have any hope of comparing two different discrete probability distributions, we must know which elements correspond. This is a tricky problem to solve exactly; our solution is to abstract the semantics further by sorting the probabilities in ascending order and aligning the probabilities between the two distributions based on their position. An example of this process is shown in Figure 5.8. This abstraction does lose precision, but as shown in our evaluation it works well in practice.

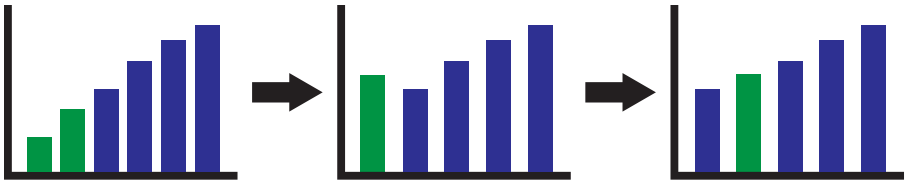


Figure 5.9: An example of shrinking a discrete distribution to enable comparison.

It is also impossible to compare two distributions if the number of elements differ. Our solution to this issue is to shrink the larger distribution one element at a time by combining the two smallest probabilities into one and re-sorting. This process alters the original distribution, which helps to penalize comparisons of two distributions with different numbers of elements. Our hypothesis is that programs which have similar numbers of paths are more likely to be semantically similar. An example of this shrinking process is shown in Figure 5.9. If the number of paths differs significantly, then we mark the two distributions as different without further comparison. Our algorithm (shown in Algorithm 4) yields and compares two discrete probability distributions, and the final piece to explain is how we compare the distributions to yield a similarity metric.

KL Divergence. The natural way to translate distributions into scores is via a divergence metric, and arguably the most popular such metric is Kullback-Leibler Divergence (KL Divergence) [90]—we use this metric in our method. It is an information-theoretic metric that operates via entropy calculations, measuring how "surprising" the difference between two probability distributions is. The result is a number between 0 and ∞, with 0 representing exact similarity and larger numbers representing relative dissimilarity. The numbers returned by calculating the KL divergence are relative to one another and are not absolute; only through experimentation can an appropriate cutoff between similar and dissimilar programs be chosen.

We are now armed with a complete method for comparing two functions via

Algorithm 4 Distribution comparison

function COMPARE-DISTRIBUTIONS($counts_P$, $counts_Q$)
Input: Two lists, $counts_P$ and $counts_Q$, containing the number of satisfying solutions output by the Barvinok model counter.
Output: A numerical representation of how "far apart" the two sets of counts were.
    if $|counts_P| > |counts_Q|$ **then**
        Swap $counts_P$ and $counts_Q$ ▷ Make $counts_Q$ the larger list
    end if
    if $|counts_Q| \geq 3 \cdot |counts_Q|$ **then**
        return ∞ ▷ Return a large distance when the number of paths differs significantly
    end if
    $P \leftarrow map(\lambda x.x/sum(counts_P), counts_P)$ ▷ Convert counts to probabilities
    $Q \leftarrow map(\lambda x.x/sum(counts_Q), counts_Q)$
    Sort P and Q
    while $|Q| > |P|$ **do**
        $Q[1] \leftarrow Q[1] + Q[0]$
        $Q \leftarrow Q[1..]$ ▷ Make the distributions the same length
        Sort Q
    end while
    return KL-DIVERGENCE(P, Q)
end function

$$\{17341665826650, 17867170851700, 1751511700, 17867170851700, 17867170851700, 1804057051\},$$
$$\{17341665826650, 17867170851700, 1751511700, 1751511700, 17867170851700, 18403185977251\}$$
$$\Downarrow$$
$$[2.468770 \times 10^{-5}, 2.542833 \times 10^{-5}, 0.244432, 0.251839, 0.251839, 0.251839],$$
$$[2.450260 \times 10^{-5}, 2.450260 \times 10^{-5}, 0.242599, 0.249951, 0.249951, 0.257450]$$
$$\Downarrow$$

$$\begin{aligned}
&2.468770 \times 10^{-5} \cdot \log_2(\frac{2.468770 \times 10^{-5}}{2.450260 \times 10^{-5}}) + \\
&2.542833 \times 10^{-5} \cdot \log_2(\frac{2.542833 \times 10^{-5}}{2.450260 \times 10^{-5}}) + \\
&0.244432 \cdot \log_2(\frac{0.244432}{0.242599}) + \\
&0.251839 \cdot \log_2(\frac{0.251839}{0.249951}) + \\
&0.251839 \cdot \log_2(\frac{0.251839}{0.249951}) + \\
&0.251839 \cdot \log_2(\frac{0.251839}{0.257450}) \\
&\approx 0.00012
\end{aligned}$$

Figure 5.10: The final steps of comparing the insertion sort and longest increasing subsequence functions from Section 5.2.

path conditions. As an example, we reproduce the final steps in the comparison of the insertion sort and longest increasing subsequence functions from Section 5.2 in Figure 5.10. In the following section, we evaluate the performance of this method against related work.

5.4 Evaluation

5.4.1 Methodology

For our evaluation we compare our Type IV semantic clone detection technique against a state of the art PDG-based Type IV semantic clone detection technique, using two separate benchmark suites: (1) a handcrafted suite that contains both semantically-similar but syntactically-dissimilar programs (SEMTRUESYNFALSE) and syntactically-similar but semantically-dissimilar programs (SEMFALSESYNTRUE); and (2) a subset of BigCloneBench [139] semantic clone/non-clone pairs. We make our implementations of all of the techniques and our benchmark suites freely available (see Section 5.1).

PDG Comparison Specifics. The source code for existing PDG-based clone detection techniques is not available, and so we implement our own based on the literature. We generate program dependence graphs for a pair of functions using the `sourcedg` PDG generator described in Marin and Rivero [101]. The usual method for comparing two program dependence graphs is via subgraph isomorphism, but the standard implementations are not sufficient. For example, the popular VF2 algorithm [47] performs *graph*-subgraph isomorphism testing, whereas we want to find *subgraph*-subgraph isomorphism (i.e., we want to match a sub-program to another sub-program). Therefore, we compare ourselves against two separate techniques for graph comparison. The first

is an eigenvector-based approach that works by comparing Laplacian spectrums [85, 1], and the second uses a graph edit distance technique [21]. The graph edit distance algorithm that we use is incremental in the sense that it returns better approximations the longer it is run; we let this algorithm run for 30 seconds in our evaluation.

We first describe and evaluate our handcrafted benchmark suite, and then we do the same for our benchmarks taken from BigCloneBench. We provide timing data for all methods in Section 5.4.4.

5.4.2 Handcrafted Benchmark Suite

Limitations. In this work we are limited by the tools that we have at our disposal. Specifically, Symbolic PathFinder cannot handle many types of program constructs (e.g., doubly-nested symbolic arrays), and benchmarks used in works that employ SPF are typically small [see, e.g., 56, 146, 147, 38, 29, 115, 33]. In addition, Barvinok can only count integer constraints. Therefore, our choice of benchmarks is limited to relatively small programs operating over integers. These restrictions are purely a matter of the infrastructure that we base our implementation on, and not of our technique itself. Our technique can take advantage of any symbolic execution engine and model counter that are available, and it does not impose any additional restrictions itself.

Description of Handcrafted Benchmarks. As mentioned previously, we have chosen to evaluate along two axes: (1) against semantically similar, syntactically different programs (SEMTRUESYNFALSE); and (2) against syntactically similar, semantically different programs (SEMFALSESYNTRUE). Figure 5.11 lists our chosen benchmarks along the first axis; each set consists of semantically-similar but syntactically-dissimilar programs. Any two programs that belong to different sets are semantically dissimilar.

- Binary search (iterative), Binary search (recursive)
- Insertion sort, Bubble sort, Selection sort, Heapsort, Cocktail sort, Circle sort, Shell sort, Longest increasing subsequence
- Palindrome, Reverse array
- Factorial (recursive), Factorial (iterative)
- Linear search forward (recursive), Linear search backward (iterative)

Figure 5.11: Sets of semantically similar, syntactically different benchmark programs, i.e., SEMTRUESYNFALSE.

All of the algorithms the programs are based on are common algorithms used in the wild. We evaluate our method's ability to cluster the programs into the correct sets in Section 5.4.2.

To evaluate along the second axis, we have created syntactically similar, semantically different programs for a selection of the above benchmarks. These benchmarks all *syntactically look like* the original benchmarks—we invite the reader to view the source code to see exactly how these benchmark programs operate. We evaluate our method's ability to differentiate between these syntactic "decoy" programs in Section 5.4.2.

Results for SEMTRUESYNFALSE

We provide a graphical interpretation of our results and then present them using the standard metrics of precision and recall.

Our Method. Results for our method on the benchmarks from Figure 5.11 are shown in Figure 5.12. The pairwise scores for each benchmark are shown in the form of a heatmap, and lower scores indicate greater similarity was detected. At a glance it is apparent that our method does well for most of the benchmarks—the sorting functions,

for example, are all marked as similar to each other; the same is true of most other groups.

Our method is not perfect, and contains some inaccuracies. The factorial benchmarks are marked as similar to the sorting benchmarks because of the distributions of the paths: for both sets of benchmarks, the probability of a given path is mostly equal to another, and that is why we see this output. The PDG methods have a better ability to differentiate this particular pair of benchmark sets, and so our method would benefit from such a complementary technique.

The palindrome and array reversal functions are not marked as similar, and this is another error of our method. The array reversal program does not contain any branches, and so our method fails to generate any path conditions—we return a default high distance value (10.0) in this case. This is one limitation of our method: we operate on path conditions, and if there are no path conditions we have nothing to compare. The PDG method also has trouble differentiating this pair of benchmarks.

PDG-Based Method. Results for the program dependence graph-based method are shown in Figure 5.13. Lower scores again indicate more similar programs. The figure only shows the results of the Eigenvector Distance method; the Graph Edit Distance method returned similar results, just differently scaled. The PDG method seems to have trouble on several benchmarks, notably iterative and recursive algorithm pairs; this makes sense because the programs look quite different, hence their program dependence graphs will also differ. The method does give low scores to many semantically similar functions, such as the sorting algorithms, though the range of scores among similar functions is larger than for our method.

Precision and Recall. For another view into the data, we also compute precision and recall numbers. To do so we must define a "cutoff" threshold value for each method,

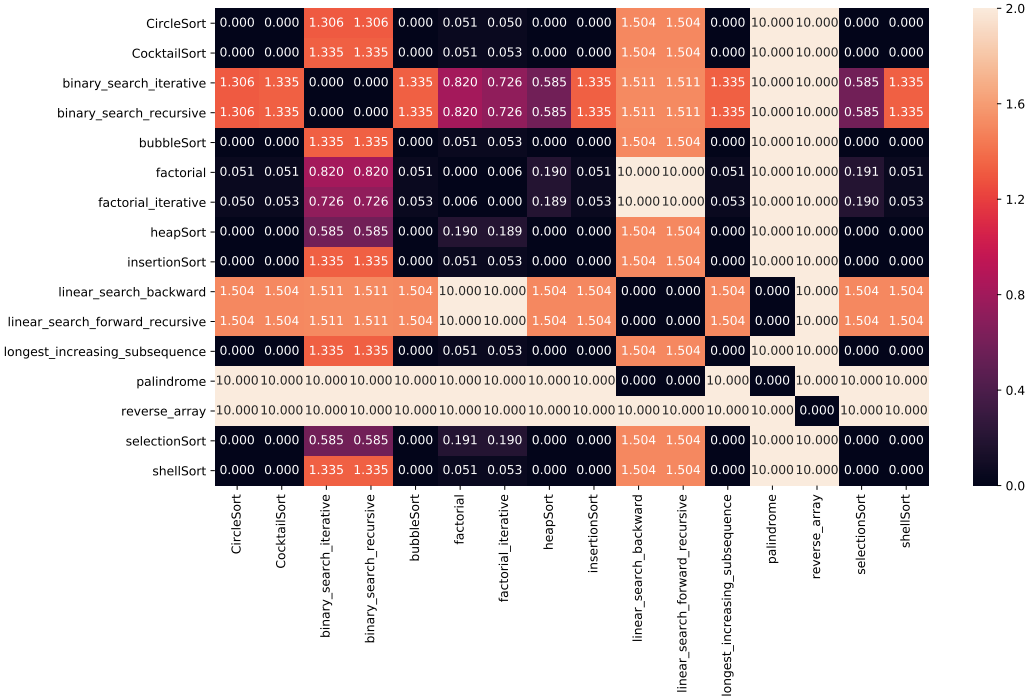


Figure 5.12: Results of our method on SEMTRUESYNFALSE. Lower scores indicate higher similarity.

such that each pair on one side of the threshold is called a clone and on the other side is called a non-clone. To present each method at its best, we computed a separate cutoff for each method that gave that method the best results in terms of its overall F-measure [130]. The results are in Table 5.2. Precision (number of true positives divided by the sum of true and false positives) measures the percentage of pairs that a method claims are clones that really are clones. Recall (number of true positives divided by the sum of true positives and false negatives) measures the percentage of pairs that really are clones that a method correctly identifies as clones. F-measure combines precision and recall into a single overall score, and our method is the clear winner.

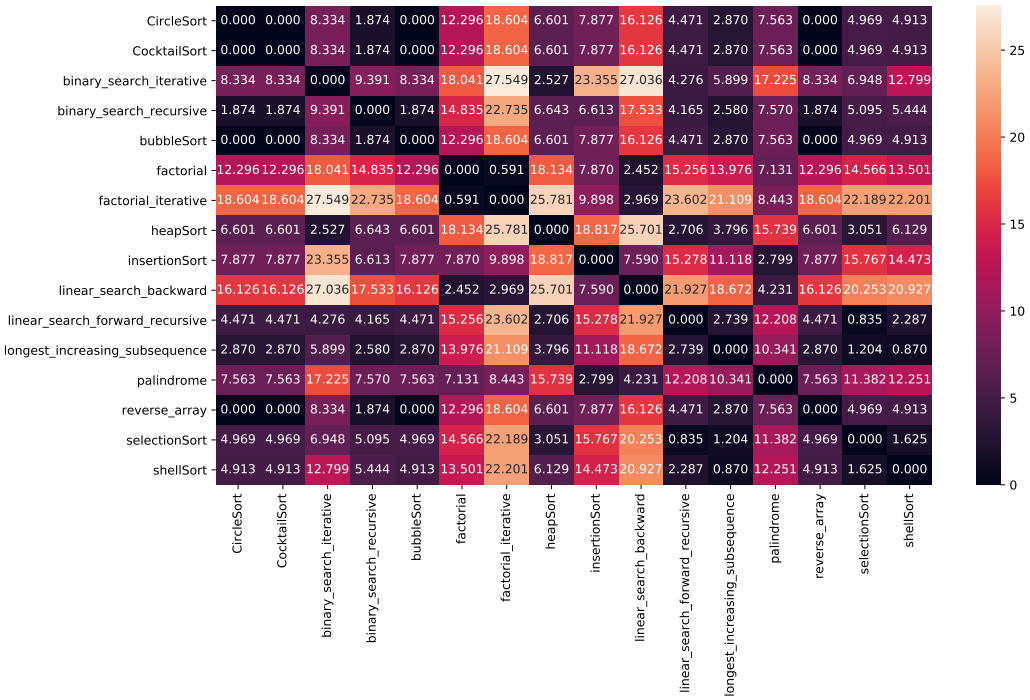


Figure 5.13: Results of the Eigenvector PDG-based method on SEMTRUESYNFALSE. The Graph Edit Distance heatmap is similar. Lower scores indicate higher similarity.

Table 5.2: F-measure, Precision, and Recall for SEMTRUESYNFALSE: a numeric perspective on the results. TP stands for "true positives", FN for "false negatives", etc. Higher is better for the Precision, Recall, and F-measure columns.

Method	TP	FP	TN	FN	Cutoff	Precision	Recall	F-measure
Our method	31	2	86	1	0.0065	0.939	0.969	0.954
PDG (Eigenvector Dist.)	26	41	47	6	7.8772	0.388	0.813	0.525
PDG (Graph Edit Dist.)	25	42	46	7	0.5135	0.373	0.781	0.505

Table 5.3: For each benchmark in the left column, we choose a semantically similar, syntactically different benchmark to compare against from the SEMTRUESYNFALSE benchmarks.

Benchmark	SEMTRUESYNFALSE
Binary search, iterative	Binary search, recursive
Insertion sort	Bubble sort
Palindrome	Reverse array
Selection sort	Heapsort
Cocktail sort	Circle sort
Factorial, recursive	Factorial, iterative
Longest increasing subsequence	Shell sort
Linear search forward, recursive	Linear search backward, iterative

Results for SEMFALSESYNTRUE

We compare the different clone detection techniques' abilities to distinguish semantically dissimilar but syntactically similar programs. Because the reported scores are relative, we also provide the results of comparing the programs in the left-hand column of Table 5.3 against semantically similar, syntactically dissimilar programs as described above. The idea is that the results for the SEMFALSESYNTRUE pairs should be much higher (i.e., indicate greater distance) than the results for the SEMTRUESYNFALSE pairs.

Our Method. Results for our method are shown in Figure 5.14. The graph is arranged with each benchmark grouped next to its SEMTRUESYNFALSE (blue) and SEMFALSESYNTRUE (orange) counterparts. Small slivers indicate 0 or very small distances.

The goal is for the SEMTRUESYNFALSE benchmarks to be given low distances and for the SEMFALSESYNTRUE benchmarks to be given high distances; thus, one can find a clear "cutoff" point that separates the semantically similar programs from

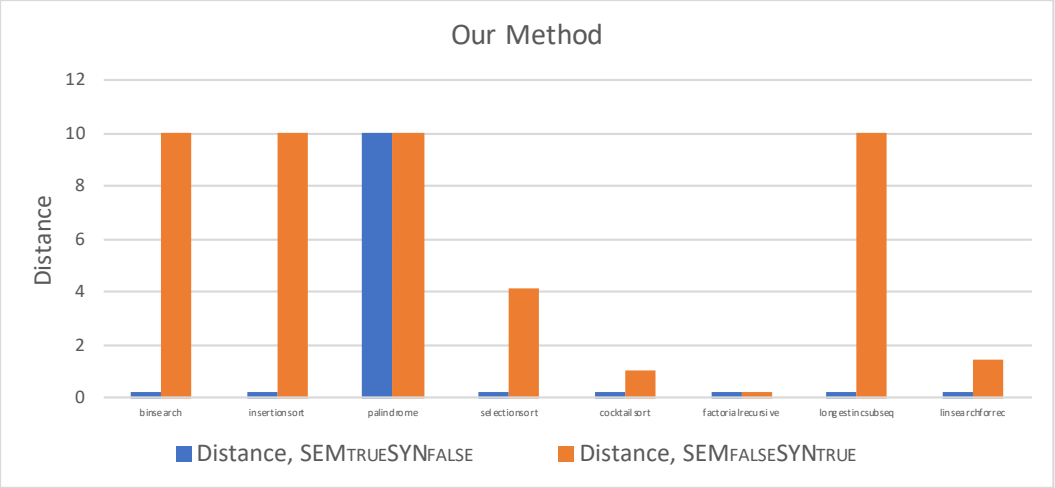


Figure 5.14: Results of our method. Lower is better for SEMTRUESYNFALSE, and higher is better for SEMFALSESYNTRUE.

the semantically different programs. In all cases but two our method has this ideal property.

The first outlier is the palindrome case, which was explained in the previous subsection.

The second is the recursive factorial case. The comparison function in the SEMFALSESYNTRUE case is a recursive sum function, and the one comparison taking place in both of those benchmarks is a test for the base case. One could argue that this benchmark is not semantically different enough, but we prefer to highlight it as a possible limitation.

We note that, as is apparent from the following figures, the PDG-based methods cannot properly distinguish these two outlier benchmarks either.

PDG-Based Methods. Results for the program dependence graph-based methods are shown in Figure 5.15. Again, the goal is for the blue bars to be low and the red bars to be high. The Eigenvector Distance method and the Graph Edit Distance method return relatively similar results, and the difficulties with iterative and recursive function pairs

Table 5.4: F-measure, Precision, and Recall for the SEMFALSESYNTRUE benchmarks.

Method	TP	FP	TN	FN	Cutoff	Precision	Recall	F-measure
Our method	7	0	8	1	0.0064	1.0	0.875	0.933
PDG (Eigenvector Dist.)	7	6	2	1	7.8769	0.538	0.875	0.667
PDG (Graph Edit Dist.)	8	7	1	0	0.6875	0.533	1.0	0.696

are carried over from the previous subsection.

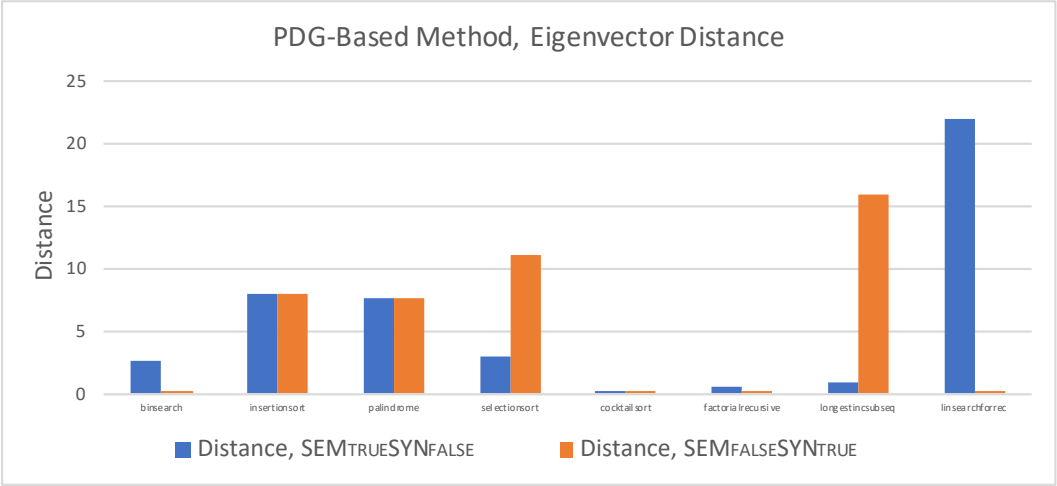
Precision and Recall. Again we find an optimal "cutoff" value for each method that best splits the clones from the non-clones, and the results are in Table 5.4. The results are similar to those in the previous subsection.

5.4.3 BigCloneBench Benchmark Suite

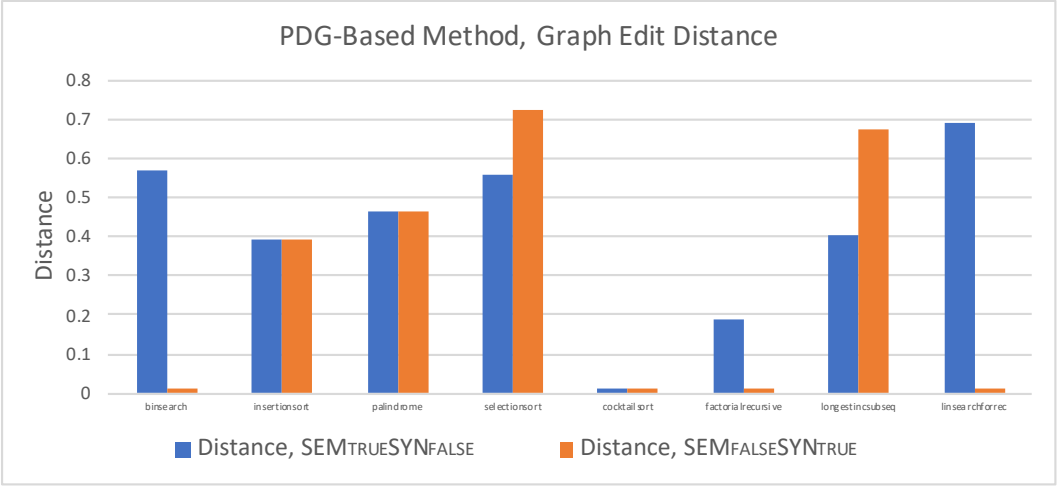
BigCloneBench benchmark description. BigCloneBench [139] contains Type IV clone pairs (placed into a category called "Weakly Type III/Type IV"), and they are specified as clone pairs having a token similarity of less than 50% or a line similarity of less than 50%. The benchmark suite also has false positive Type IV clone pairs. We chose 50 pairs of true Type IV clone pairs and 50 pairs of false positive Type IV clone pairs as our benchmark suite—we manually made minimal edits to these programs so that they would work with our clone detection method.

Results. The results of running each of the different methods through our BigCloneBench benchmark suite is shown in Table 5.5, and our method is on top with the highest F-measure. For the graph edit distance-based PDG method, the best cutoff was chosen so that almost everything was marked as a clone; the eigenvector distance-based method also had poor precision, with the best cutoff maximizing recall.

These results indicate that our method can also detect semantic clones with high precision and recall in a well-regarded benchmark suite.



(a) Eigenvector Distance



(b) Graph Edit Distance

Figure 5.15: Results of the PDG-based methods. Lower is better for SEMTRUESYN-FALSE, and higher is better for SEMFALSESYNTRUE.

Table 5.5: F-measure, Precision, and Recall for the BigCloneBench benchmarks.

Method	TP	FP	TN	FN	Cutoff	Precision	Recall	F-measure
Our method	42	13	37	8	5.923	0.764	0.840	0.800
PDG (Eigenvector Dist.)	43	29	21	7	10.105	0.597	0.86	0.705
PDG (Graph Edit Dist.)	50	48	2	0	0.871	0.510	1.000	0.676

Table 5.6: Aggregate timing data for all methods and benchmarks. The column labeled "BCB" signifies the BigCloneBench benchmarks.

Statistic	Our Method		Eigen. Dist.		Graph Edit Dist.	
	Handcrafted	BCB	Handcrafted	BCB	Handcrafted	BCB
Min	2.227 s	2.119 s	1.191 s	1.168 s	1.330 s	1.169 s
Max	8.269 s	21.177 s	1.411 s	1.565 s	31.408 s	31.560 s
Mean	3.598 s	2.665 s	1.319 s	1.341 s	25.952 s	20.345 s
Median	2.554 s	2.393 s	1.329 s	1.330 s	31.306 s	31.314 s
Std. Dev.	1.561 s	1.971 s	0.046 s	0.076 s	10.980 s	14.458 s

5.4.4 Average Running Times for Both Benchmark Suites

In Table 5.6 we list the aggregate timing information for all of our benchmark suites and methods. We have compiled the data separately for the Handcrafted benchmarks and for the BigCloneBench benchmarks, and we have done so for our own method and the two we compare against.

It is clear that our method is in between the Graph Edit Distance method, which often takes the full time we allot to it, and the Eigenvector Distance method. We observe a roughly 2–3$\times$ slowdown on average when compared to the faster of the two PDG similarity methods, and this makes our method a reasonable alternative or complement in many cases.

5.5 Related Work

We discuss related work first in non-semantic and then semantic clone detection.

5.5.1 Non-Semantic Clone Detection

Non-semantic clone detection has been studied extensively [125, 121]; it also preceded semantic clone detection as a research area. In this subsection we provide a few

examples to give context before we introduce work related to semantic clone detection.

All clone detectors accept programs in a certain representation, and the representation affects what kinds of matching techniques can be used. The representation yields performance and precision tradeoffs as well. The three main levels of program representations used in non-semantic clone detection are strings, tokens, and syntax trees; we give a notable example of each here from the literature:

- **NICAD** [126] compares textual lines of code (i.e., a string representation of each line) after a normalization process.
- **CCFinder** [75] employs a tokenizer, and then transforms the tokens according to certain transformation rules. This method avoids issues regarding whitespace that affect string-based methods.
- **Deckard** [69] converts syntax trees into feature vectors for fast comparison. The method they describe allows access to a tree distance-like result at a much lower computational cost. This method is able to compare program structures in a way that the string-based and token-based methods cannot.

5.5.2 Semantic Clone Detection

Defining Semantic Similarity. As explained in Section 5.1 there is no existing formal, precise definition of *semantic similarity* that we can use to obtain a ground truth and classify code fragments as true clones or non-clones. Existing work relies on more intuitive notions of semantic similarity, i.e., the two code fragments "do similar things" [88, 127, 128, 83, 114]. Two notable works take this vague definition further: Simions and Code Relatives. Simions [71] are defined in terms of similar input/output behavior, while we believe that inputs and outputs do not paint the full picture. Code Relatives

[137] make the case for examining the behavior of program executions via a comparison function, which is like our own interpretation, but the program executions are defined only as traces of program locations—again, this cannot account for all types of semantic clone techniques, including our own.

Works most similar to ours. The most similar related work to our proposed technique is from Filieri et al. [57]. That work proposes the idea of using Symbolic PathFinder (SPF) [111] to quantify the impact of changes between two programs, but it does not describe any algorithms to perform such differencing. The running example they use depicts the comparison of symbolic execution trees, however that method would not work for programs that are not incremental updates from each other, as it is not clear how to compare such trees when the control flow graphs look vastly different. In contrast, our method sidesteps these issues by observing the probabilities of each branch and performing comparisons in a more syntax-agnostic way. We also describe our algorithm in depth and evaluate it against existing work.

Another more recent work that shares similarities with our own is SemCluster [114]. This work employs model counting to help cluster student code submissions. Key differences to our work are (1) that SemCluster uses dynamic analysis to compute path conditions (and therefore is limited by the number of test cases available), and (2) that SemCluster applies dynamic data-flow similarity techniques to explicitly *exclude* semantically similar programs that do not share data-flow patterns (e.g., insertion sort and bubble sort).

Program Dependence Graphs. Most existing works on semantic clone detection involve the use of program dependence graphs (PDGs) [84, 88, 94, 61]. Each paper describes different methods used to match code fragments based on their dependence graphs. Komondoor and Horwitz [84] find isomorphic PDG subgraphs via program

slicing. Krinke [88] finds similar subgraphs within a given pair of PDGs. Liu et al. [94] apply subgraph isomorphism checking to PDGs in the context of plagiairism detection for student code. Gabel et al. [61] converts program dependence graphs to characteristic vectors and applies Deckard-like methods to compare them.

All of the above-mentioned methods are based on the underlying assumption that similar programs have similar control and data dependencies, as that is what a program dependence graph provides. This assumption, however, is not always a valid one—different algorithms can solve the same problem in vastly different ways, and the structure of data and control in such semantically similar program pairs often does not correspond. Likewise, recursive and iterative versions of the same function naturally differ in terms of control flow, and often contain extra variables that affect the data flow dependencies.

Our method works at the level of branch conditions made in each path that a program takes, and this underlying structure tends to be similar across semantically similar code pairs. This does mean that our technique does not tend to work well in cases where there are few branches, and so PDG-based methods are complementary to our work for that case.

Symbolic Execution. Luo et al. [99] find subsequences of semantically equivalent basic blocks in order to detect source code plagiarism. This is related to an earlier work which employed subgraph isomorphism checking on control flow graphs, matching the basic blocks using theorem proving and symbolic execution [62]. In contrast, our work operates in the style of Probabilistic Symbolic Execution [63]—i.e., on path conditions rather than basic blocks—and does not focus on exact equivalence and theorem proving, which can be time-intensive processes; instead we define a notion of distance between distributions of sets of execution path conditions. Stolee et al.

[136] use symbolic execution to extract an execution tree and then form input/output constraints (instead of path constrains) to encode the semantics of a program.

Other Approaches. Alternative methods for semantic clone detection take advantage of comment text [100], make use of fingerprinting [81], compare abstract memory states [82], employ random testing [70], and use machine learning techniques [127, 151]. Quantitative solutions such as the one proposed in this chapter have not been employed in those works.

Chapter 6

Student Feedback Using Invariant Inference

6.1 Introduction

One of the primary ways to gain confidence in the correctness of a program is software testing, i.e., executing a program on a set of inputs and checking the resulting outputs against some oracle. When tests fail, the programmer must determine why failure occurred and how to fix it. These same issues arise in the context of computer science education: instructors often grade and provide feedback on programming assignments by checking student submissions against a predetermined set of test cases and informing the student how many test cases failed. If trial submissions are allowed then a student can use this feedback to improve their solution and submit again; if not then a student might still wish to know what was wrong with their solution to improve for the next assignment.

Simply knowing how many tests failed does not do much to help a student understand what went wrong, and even showing the failing tests does not explain much to a neophyte programmer. Ideally an instructor can take the time to review each student's work with them and help them understand their respective issues, but this model is

unrealistic in large classroom settings. **The focus of this chapter is how to provide automated feedback about programming assignments that have been graded via a test oracle**, in order to help students understand and fix incorrect submissions. To do so, we take a radically different approach to the problem than existing work which leverages program repair or ad-hoc analyses. **Our key insight is to explain to the student the characteristics that distinguish passing inputs and failing inputs**. In other words, we run the student's submission on a set of test inputs and use the oracle to classify the inputs as *passing* or *failing*; we then infer invariants about these two sets that provide the student some insight into what their solution is doing right and what it is doing wrong.

The passing tests represent cases where the student's understanding of the problem matches that of the reference solution, while the failing tests represent cases where the student's understanding is inconsistent with the reference solution. The computed invariants help characterize these similarities and differences and explain them to the student. We believe that the invariants are useful to the student to help them understand what logical aspects of the problem they are misunderstanding. Our focus in this chapter is on providing feedback for logical program errors, though we could extend our method to provide feedback on other kinds of errors by adding additional classification categories for inputs that cause various kinds of crashes or that exhibit undefined behavior (e.g., according to UBSan [20]).

While our approach could in theory be used for any kind of assignment for which a student would submit an executable program that can be tested, in this chapter we look specifically at assignments suitable for a sequence of introductory C++ programming classes. Note that our general approach could be useful for any setting (even outside CS education) where we can take a set of inputs and use a classifier to partition those inputs into interesting sets.

In this chapter we make the following contributions:

- We describe a general method to generate automated student feedback for programming assignments, based on classifying test inputs into two sets (*failing* and *passing* inputs) and using invariant inference to characterize the differences between those sets (Section 6.2).
- We describe a specific instantiation of our proposed method targeting feedback for introductory C++ programming assignments using the Daikon [55] invariant inferencer (Section 6.3).
- We evaluate our proposed method on actual student submissions for a set of assignments taken from a sequence of courses introducing C++ programming (Section 6.4).
- We make our implementation publically available as open source for the research and educational communities.

We describe related work in Section 6.5.

6.2 Method Overview

We discuss our method in the context of a course that allows students to submit their potential solutions online multiple times before they submit their final version. Each time they submit a potential solution it is automatically checked against a predefined set of tests (created by the instructor) and the students are informed of whether their solution fails any tests. This context is taken directly from the way that the introductory programming courses are taught in the authors' department.

Our goal in this work is to improve the automated feedback that students get on their potential solutions to help them understand what they are doing wrong. We focus specifically on logic errors that represent a misunderstanding of either the problem itself or of how the programming language being used works (though our method could potentially be useful for helping with other kinds of errors such as crashes, we do not evaluate it on such in this chapter).

Existing methods for automated feedback as discussed in Section 6.5 examine the text of the student's program in order to compare it against other student solutions or a reference solution, and/or attempt to repair the program to make it pass the given tests. The feedback given to the student is in the form of suggested program changes. In contrast to the existing methods, we take a completely different (and complementary) approach that treats the student's program as a black box. Instead, we wish to generate feedback that helps the student focus on potential misunderstandings they have about the problem being solved or the language being used.

Given a reference solution and a set of test inputs, we can classify each test input by whether the student solution's output agrees with the reference output (the input is *passing*) or the student's output does not agree with the reference output (the input is *failing*). Failing inputs represent a mismatch between the way that the student understood the problem and the way the instructor understood the problem (or, alternatively, the student misunderstood how some particular language feature actually works).

In order to help the student narrow in on their misunderstandings, the feedback we generate characterizes the passing and failing sets by generating invariants which are then translated into a form that is usable by the student.

6.2.1 Generating Invariants

We will call a tuple of $\langle \text{INPUT}, \text{OUTPUT}_S, \text{OUTPUT}_R \rangle$ a *sample*, where INPUT is the test input, OUTPUT_S is the output of the student submission, and OUTPUT_R is the output of the reference solution. Samples are computed by running a program (either the reference solution or a student solution) on the inputs and recording the respective outputs. Let **Refs** be the set of samples obtained by running the reference solution on each of the inputs (necessarily $\text{OUTPUT}_S = \text{OUTPUT}_R$). Let **Pass** and **Fail** be the sets of samples obtained by running the student solution on each of the inputs and categorizing the results according to whether they agree with the reference solution or not (i.e., for **Pass** $\text{OUTPUT}_S = \text{OUTPUT}_R$ and for **Fail** $\text{OUTPUT}_S \neq \text{OUTPUT}_R$). Then **Pass** $\subseteq$ **Refs** and **Fail** corresponds to **Refs** $\setminus$ **Pass** except with the student and reference outputs being different. Including the student and reference solutions in the same sample is useful for the **Fail** case, where we can derive invariants relating the student's solution to the reference solution.

Invariant inference takes a set of samples and derives a formula that is true for all elements of that set. The exact inference mechanism can vary based on the content of the samples and the types of invariants desired. In Section 6.3 we describe a specific inference mechanism based on Daikon [55] which uses invariant templates, but our method is mostly agnostic to this choice and other inference mechanisms would be interesting to explore (e.g., formal language learning [109] for samples involving strings).

We denote the invariants inferred from a particular set of samples using $[\![\cdot]\!]$, so that $[\![\textbf{Refs}]\!]$, $[\![\textbf{Pass}]\!]$, and $[\![\textbf{Fail}]\!]$ are the invariants inferred from each respective set of samples. $[\![\textbf{Refs}]\!]$ describes characteristics that are true of all correct samples; this is what the student should be trying to achieve. $[\![\textbf{Pass}]\!]$ describes characteristics that are

true of all samples the student's solution got correct; this is what the student is doing right. $[\![\textbf{Fail}]\!]$ is more complicated because it actually consists of two type of statements: (1) things that are true of **Fail** that are not true of **Pass** or **Refs**, indicating potential misunderstandings that led to error; and (2) things that are also true of **Pass** or **Refs**, indicating things that the student's solution is getting correct even though the final answer was wrong. The former statements in (1) help the student pinpoint where they need to fix something, while the latter statements in (2) help the student narrow down potential sources of error by eliminating cases that they are actually getting correct.

In addition to these invariants, we also compute another useful set of samples: $\textbf{Pass} \cup \textbf{Fail}$. This set contains all the student solution's samples whether they were correct or not. Then $[\![\textbf{Pass} \cup \textbf{Fail}]\!]$ describes characteristics that are true about the student's solution itself, ignoring whether it was doing the right thing or not.

6.2.2 Method Workflow

The general workflow for generating student feedback is the following:

1. The instructor creates a reference solution for the assignment and generates a large number of diverse inputs, which they execute the reference solution on to get the set **Refs** of $\langle \text{INPUT}, \text{OUTPUT}_S, \text{OUTPUT}_R \rangle$ tuples.

2. When a student submits a solution, their program is run on the same inputs to get a set of $\langle \text{INPUT}, \text{OUTPUT}_S, \text{OUTPUT}_R \rangle$ samples which are sorted into **Pass** or **Fail** depending on whether the sample's student output agrees with the sample's reference output.

3. From the **Pass** and **Fail** sets we create the additional sample set $\textbf{Fail} \cup \textbf{Pass}$.

4. We infer invariants for each tuple set using an appropriate inferencer, yielding the invariants $[\![\textbf{Refs}]\!]$, $[\![\textbf{Pass}]\!]$, $[\![\textbf{Fail}]\!]$, and $[\![\textbf{Fail} \cup \textbf{Pass}]\!]$.

5. We translate each invariant above into a natural language description suitable for a neophyte programmer.[1] These descriptions are sent as feedback to the student.

6.2.3 Method Example

We explain and motivate our proposed feedback method via a concrete example taken from an actual programming assignment given in an introduction to programming course.[2]. The problem can be stated as follows (modified for succinctness from the actual assignment description given to students):

> **Anagram Detection:** Given two strings S_1 and S_2 return *true* if S_1 and S_2 are permutations of each other, otherwise return *false*.

There are a number of logical errors that a student could make in their solution; one possibility is that they enforce the following mistaken invariant: $|S_1| = |S_2| \wedge (c \in S_1 \iff c \in S_2)$. This incorrect solution does not check the actual quantity of each character and would, for example, misclassify the strings "aab" and "abb" as anagrams of each other. The student's enforced invariant is weaker than the reference solution's invariant, i.e., it will accept a proper superset of string pairs as anagrams.

The test inputs are pairs of strings and the output is a boolean, therefore samples are of the form $\langle$STRING, STRING, BOOL, BOOL$\rangle$. Table 6.1 contains a subset of the possible **Pass** and **Fail** sets given the mistaken solution described above (we don't list **Refs** explicitly for space, but it is trivially derivable from **Pass** and **Fail** as is **Pass** $\cup$ **Fail**).

[1] While the exact translation process would depend on the form of invariant being inferred, for the work described in this chapter we found that a simple, straightforward mapping from formulae to

Pass	**Fail**
⟨A, A, TRUE, TRUE⟩	⟨AAB, ABB, TRUE, FALSE⟩
⟨AB, BA, TRUE, TRUE⟩	⟨BAA, BBA, TRUE, FALSE⟩
⟨AABB, BABA, TRUE, TRUE⟩	⟨ABB, AAB, TRUE, FALSE⟩
⟨A, AA, FALSE, FALSE⟩	⟨AABB, BBBA, TRUE, FALSE⟩
⟨AA, AB, FALSE, FALSE⟩	⟨ABCC, AABC, TRUE, FALSE⟩
⋮	⋮

Table 6.1: Example **Pass** and **Fail** sets. The **Refs** set is the union of **Pass** and **Fail** except the **Fail** student outputs would all be changed to FALSE.

From these sets we can infer the following invariants (keeping in mind that we are showing only a subset of the actual sets), where $\#_c(S)$ is the number of occurrences of character c in string S. We don't show all possible invariants we can infer, only a sample of them. Note that it can be useful to derive invariants that are strictly weaker than (i.e., implied by) stronger invariants rather than only reporting the strongest possible invariant; these weaker invariants provide specificity that can help the student understand the problem.

⟦Refs⟧

$$\begin{aligned}
\text{out}_S = \text{out}_R = \text{TRUE} \Rightarrow & |S_1| = |S_2|, \\
& \forall c.\#_c(S_1) > 0 \Leftrightarrow \#_c(S_2) > 0, \\
& \forall c.\#_c(S_1) = \#_c(S_2) \\
\text{out}_S = \text{out}_R = \text{FALSE} \Rightarrow & \exists c.\#_c(S_1) \neq \#_c(S_2)
\end{aligned}$$

natural language sufficed.

[2]UCSB's CMPSC 16: Problem Solving with Computers I

⟦Pass⟧

Same as **⟦Refs⟧**

⟦Fail⟧

$$\begin{aligned}\neg \text{out}_R = \text{out}_S = \text{TRUE} \Rightarrow & |S_1| = |S_2|, \\ & \forall c.\#_c(S_1) > 0 \Leftrightarrow \#_c(S_2) > 0, \\ & \exists c_1, c_2.\#_{c_1}(S_1) > \#_{c_1}(S_2) \wedge \#_{c_2}(S_1) < \#_{c_2}(S_2), \\ & \exists c.\#_c(S_1) \neq \#_c(S_2)\end{aligned}$$

⟦Pass ∪ Fail⟧

$$\begin{aligned}\text{out}_S = \text{TRUE} \Rightarrow & |S_1| = |S_2|, \\ & \forall c.\#_c(S_1) \geq 0 \Leftrightarrow \#_c(S_2) \geq 0, \\ & \exists c_1, c_2.\#_{c_1}(S_1) \geq \#_{c_1}(S_2) \wedge \#_{c_2}(S_1) \leq \#_{c_2}(S_2) \\ \text{out}_S = \text{FALSE} \Rightarrow & \exists c.\#_c(S_1) \neq \#_c(S_2)\end{aligned}$$

Since we record both the student and reference solution in a sample, technically the last set **Pass** ∪ **Fail** could be used to derive all of the invariants (e.g., $\text{OUT}_S = \text{OUT}_R \Rightarrow ...; \text{OUT}_S \neq \text{OUT}_R \Rightarrow ...; \text{out}_S = \text{TRUE} \Rightarrow ...; \text{out}_S = \text{FALSE} \Rightarrow ...; \text{out}_R = \text{TRUE} \Rightarrow ...; \text{out}_R = \text{FALSE} \Rightarrow ...$; etc). However, doing so puts the onus on the inference engine to figure out all of the necessary implications. By explicitly separating out these

sets *a priori* we greatly reduce the burden on the inferencer and make inference more practical.

The ⟦**Refs**⟧ invariant describes a correct solution, i.e., what is true about all inputs that should yield TRUE and what is true about all inputs that should yield FALSE. Unsurprisingly, one is the negation of the other. We also include weaker invariants for the TRUE case such as $|S_1| = |S_2|$ and $\forall c.\#_c(S_1) > 0 \Leftrightarrow \#_c(S_2) > 0$ that are implied by the strongest invariant $\forall c.\#_c(S_1) = \#_c(S_2)$.

The ⟦**Pass**⟧ invariant is the same as ⟦**Refs**⟧ because the mistaken student solution will still recognize all correct anagrams; the problem with the student solution lies with failing to reject some non-anagrams.

The ⟦**Fail**⟧ invariant has something in common with ⟦**Refs**⟧ because the incorrectly accepted inputs are *almost* anagrams; that is, they share some characteristics in common with true anagrams. However, the existential invariants show exactly the difference between ⟦**Fail**⟧ and ⟦**Refs**⟧.

The ⟦**Pass** ∪ **Fail**⟧ invariant is a union of ⟦**Fail**⟧ without the last clause and the FALSE case of ⟦**Refs**⟧ because the student solution is enforcing a weaker invariant than the reference solution.

6.2.4 Method Limitations

While our evaluation shows that our proposed method can yield good results, it is not perfect and has some limitations that we discuss here:

- Our method depends on the presence and quality of an invariant inferencer suitable for the types of inputs and outputs required by the assignment. If a suitable inferencer is lacking then our method will not apply.

- The instructor may need to guide the invariant inferencer by giving it hints about

the invariants that are of particular interest for a given assignment. For example, given an inferencer based on template instantiation the instructor may need to provide suitable templates if the inferencer's standard set is not adequate.

- The more mistaken a student's solution, the weaker the inferred invariants can become. In an extreme case all of the test inputs end up in **Fail** and none in **Pass** (which means that $\textbf{Refs} \setminus \textbf{Pass} = \textbf{Refs}$ and $\textbf{Pass} \cup \textbf{Fail} = \textbf{Fail}$). The weakness of the invariants derived from these sets may lessen their use in pinpointing misunderstandings, though they are still true invariants and can still be useful.

The last limitation could be mitigated by partitioning **Fail** based on instructor-provided predicates derived from common, expected student mistakes. Each partition can have invariants inferred separately, recovering stronger, more useful invariants to provide as feedback. We leave investigating this possibility for future work.

6.3 Method in Detail

The method described in Section 6.2 is a general approach rather than a specific system. We describe how to instantiate that approach to a specific system in this section, motivated in our choices by a set of actual assignments used in a series of introductory C++ programming courses.

The remainder of this section discusses the types of samples we handle (i.e., what kinds of inputs and outputs the assignments require) and how to generate appropriate inputs; the exact mechanism for invariant inference that we employ; and how we translate the final invariants into a form that is useful for neophyte programmers.

$$Sample = String \uplus Integer \uplus Boolean \uplus Sample^{\star}$$

Figure 6.1: Definition of samples: a sample is a sequence of strings, integers, booleans, or sequences of samples. A sample will always have a minimum of two positions: at least one input and at least one output.

6.3.1 Types of Samples

The types of samples we handle are shown in Figure 6.1; they consist of tuples that contain strings, booleans, integers, and (recursively) more tuples. These types of samples are sufficient to deal with a wide variety of assignments, including those based on data structures such as binary search trees, linked lists, and hash tables. For example, a binary search tree input can be represented as a sequence of integers in the order they will be inserted into the tree; a hash table with integer keys and string values can be represented as a sequence of (integer, string) pairs, etc. Essentially, any inputs or outputs that can be flattened into a sequence are representable in this format. We describe the exact samples used in our evaluation in Section 6.4.

Our method depends on the quantity and diversity of inputs used to test the student solutions, as they directly influence the quality of the inferred invariants. Hand-crafted inputs, while they can be used as part of the tests, are not likely to be sufficient to generate useful invariants. Thus, we need to be able to automatically generate suitable inputs from the space we've defined above. There are sophisticated automatic input generation schemes available in the literature (e.g., Dewey et al. [54]), but we found that a simple scheme of random generation under certain constraints was sufficient for our experiments.

We restrict integers to a finite range, and treat strings as a sequence of characters. Now each basic type (integer, character, and boolean) is a finite domain. Given a specific sample type (e.g., $\langle$STRING, INTEGER*, INTEGER$\rangle$), for each position of the sample:

- If it is an integer, character, or boolean position we pick a random value of the appropriate type from the restricted finite domains defined earlier;

- If it is a sequence type (including string) we pick a random integer within the integer finite domain to be the length of the sequence, then for each element of the sequence we recursively generate values using this same procedure.

For some inputs we need to enforce certain properties to be certain of getting relevant inputs. For example, the anagram problem requires us to generate some inputs that are anagrams as well as some that are not, and it is unlikely that completely random generation would generate many anagrams. We employed a simple strategy for these cases (which only arise for two assignments, one involving anagrams and one involving palindromes):

- To ensure that we generate sufficient anagrams, we employ the above strategy to randomly generate the first string and then instead of generating a second string we randomly permute the first string to get the second string.

- To ensure that we generate sufficient palindromes, we employ the above strategy to randomly generate a string and then reverse that string and append it to the original string to get a palindrome (with some tweaks to ensure we get both even- and odd-length palindromes).

This simple generation strategy does not allow us to enforce complex properties on the inputs (e.g., sortedness of lists or balanced trees) but such properties were not necessary for the assignments we evaluated against. If properties like this are necessary then there are, as mentioned previously, more complex generation strategies that can handle such a requirement.

6.3.2 Invariant Inference

Rather than build our own invariant inferencer, we used the existing Daikon dynamic invariant inferencer [55]. Daikon is intended to infer invariants over a program's entire execution, but since we only want to derive invariants relating the inputs and outputs we adapted the Daikon workflow to our needs.

Daikon

In order to explain our adaptation, we first describe the standard Daikon usage and how it works. Daikon is a template-based inferencer and comes with a wide variety of existing templates. A set of invariant templates look, for example, like $\$1 = \$2$, $\$1 \leq \$2$, $\$1 \ \% \ \$2 = 0$, $\text{LENGTH}(\$1) \geq \$2$, etc. The invariants that Daikon infers will be instantiations of these templates with program variables and values from the program's execution. Daikon consists of a *front-end* that is responsible for instrumenting the code, running tests, and collecting execution traces (as described below); and a *back-end* that is responsible for taking the traces and inferring invariants from them.

To infer invariants, first the program is compiled using the Daikon front-end infrastructure in order to add instrumentation to the program's binary. This instrumentation is added throughout the program's structure, e.g., at all function entry and exit points. The instrumentation is responsible for outputting the values of the program variables in scope at that point in the execution (and values reachable from those variables, e.g., using pointer dereference or array indexing).

Then the front-end executes the instrumented binary on a set of program inputs. Each execution results in a trace consisting of a mapping from program points to values. The end result is a set of traces, one for each input. Once the traces are collected, the front-end collates the traces and sends them to the back-end inferencer.

The inferencer is responsible for inferring invariants for each program point using the provided templates and the trace values at that program point. A simple, naive strategy would be to take each template, instantiate the template variables with all possible combinations of program variables and values for a particular trace to see which ones hold for that trace, then for each surviving potential invariant see if it holds across all the traces. Any invariant instantiation which is true over all traces would be reported as a likely invariant.

This naive strategy is both prohibitively expensive and likely to report invariants that have little supporting evidence in the traces. The actual strategy Daikon employs is more sophisticated and uses various optimizations and statistical techniques to report likely invariants in reasonable time.

Observer Methods

The template-based methodology for invariant inference can be restrictive. A well-known trick for increasing the expressiveness of the invariants that Daikon can report are so-called *observer methods* [27]. An observer method is a pure (i.e., side-effect free) predicate over the program variables and values. Daikon by default operates on primitive values (integers, pointers, etc). To infer more complex invariants or invariants over complex data structures, we can encode the desired invariant as a predicate using an observer method. Daikon will apply the observer method to various combinations of program variables and values and, using the same techniques as for instantiating templates, discover if the observer method corresponds to an invariant predicate for a given set of variables and values.

An example observer method that we use in our experiments is `contains-capital-letter`, which takes a string and returns whether that string contains a capital letter or not. This information is not a standard Daikon invariant template, but

it is useful because for several of the assignments that we evaluate it turns out that the presence or absence of capital letters is relevant.

Putting It Together

We need to adapt Daikon for our purposes, because we are not interested in instrumenting and collecting traces over executions of the student code but instead in inferring invariants over samples. Thus, we ignore Daikon's front-end altogether and only use its back-end.

When a student's solution is submitted we execute it on all of the test inputs and create a set of samples as described in Section 6.2. Each sample corresponds to a Daikon execution trace, where we can think of the sample as a trace containing only a single "program point". The samples (in the format of fictitious "traces") are then fed to the Daikon back-end exactly as if they had come from the Daikon front-end.

6.3.3 Invariant Translation

Neophyte programmers are unlikely to be able to usefully interpret invariants given as logical formulae. Thus, we need to translate the final invariants into some form that they will be able to understand. Since we are using template-based invariant inference plus observer methods, we are able to predefine straightforward translations of the invariant templates and observer method results into natural English. The results are somewhat stilted, but still perfectly understandable.

For example, when defining the observer method `contains-capital-letter(x)` we can predefine a translation as "the string `x` contains a capital letter", where `x` would be replaced with the appropriate string variable. For a template such as \$1 = \$2 we can predefine a translation as "the value of \$1

is equal to the value of $2", where $1 and $2 would be replaced with the appropriate program variables or values.

6.4 Evaluation

We evaluate our proposed feedback method on actual student submissions for ten assignments taken from a series of introduction to C++ programming courses, taken from an archive of previous course offerings. For legal privacy reasons we cannot provide the set of student submissions in the artifact linked to earlier in this chapter; the artifact only provides our implementation of our proposed method.

6.4.1 Evaluation Methodology

For each of the ten assignments we randomly select 25 student submissions that compile, run without crashing, yet provide incorrect answers. For two of the assignments (Binary Search Tree Node Removal and List Copy Assignment Operator) we were not able to obtain 25 valid samples from the set of incorrect submissions due to compilation errors, runtime errors, or because the student did not fill in the code stub at all, and therefore we use 12 and 8 samples, respectively, for those assignments.

For each assignment, for each student submission, we generate feedback in the form of a list of (translated) invariants. We judge the quality of this list by (1) whether it contains actionable, useful feedback that describes a specific property of the student's submission that points to why or how the submission was incorrect; and (2) by how many of the (translated) invariants in the list are useful rather than extraneous.

This evaluation is necessarily subjective, and to try to mitigate bias the primary author and implementor of our feedback method was not involved in this judgement

process. Instead, the second author, a grad student who was not involved in the preliminary research and implementation of our method, independently made the judgements.

6.4.2 Description of Benchmarks

We describe each of our ten benchmarks below:

- **Copy Odd Elements across Arrays.** Students are to traverse a source array and copy only the odd elements into a destination array.
  - Inputs: INTEGER* $\times$ INTEGER* $\times$ INTEGER. The source and destination arrays and the minimum size of the arrays.
  - Outputs: INTEGER. The number of elements copied.
- **ASCII Art with Custom Width$\times$Height.** Students are to write a function that creates a letter 'Z' made of '*' characters of a given width and height. Certain widths and heights, (such as 2$\times$2) are invalid and should result in an empty string.
  - Inputs: INTEGER $\times$ INTEGER. The width and height.
  - Outputs: STRING. The resulting ASCII art.
- **Anagram Identification.** Students are to return whether the two input strings are anagrams of each other, disregarding whitespace, punctuation, and capitalization.
  - Inputs: STRING $\times$ STRING.
  - Outputs: BOOLEAN.
- **Palindrome Identification.** Students are to return whether the input string is a palindrome, disregarding capitalization.

  - Inputs: STRING
  - Outputs: BOOLEAN

- **Binary Search Tree Predecessor.** Students are to take a binary search tree and an integer present in the tree and provide the largest number in the tree smaller than the given integer.
  - Inputs: INTEGER* $\times$ INTEGER. The contents of the binary tree in the order they are inserted and the integer bound.
  - Outputs: INTEGER.
- **Binary Search Tree Node Removal.** Students are to delete an entry from a binary search tree.
  - Inputs: INTEGER* $\times$ INTEGER. The contents of the binary tree in the order they are inserted and the integer to delete.
  - Outputs: INTEGER*. The updated tree.
- **List Average.** Students are to compute and return the average over all the elements in a linked list containing integer values.
  - Inputs: INTEGER*. The contents of the linked list in the order they are inserted.
  - Outputs: DOUBLE.
- **List Copy Assignment Operator.** Students are to complete the C++ copy assignment operator for a custom linked list class.
  - Inputs: INTEGER* $\times$ INTEGER*. The left- and right-hand sides of the assignment.
  - Outputs: INTEGER* $\times$ INTEGER*. The resulting linked lists.

- **Hash Table Assignment Operator.** Students are to complete the C++ copy assignment operator for a custom hash table class that uses integers as keys and strings as values.

  - Inputs: $(\text{INTEGER} \times \text{STRING})^\star \times (\text{INTEGER} \times \text{STRING})^\star$. The left- and right-hand sides of the assignment.
  - Outputs: $(\text{INTEGER} \times \text{STRING})^\star \times (\text{INTEGER} \times \text{STRING})^\star$. The resulting hash tables.

- **Hash Table Insertion and Retrieval.** Students are to implement a custom hash table class, along with the associated functions for inserting and retrieving elements.

  - Inputs: $(\text{INTEGER} \times \text{STRING})^\star$. The hash table.
  - Outputs: $(\text{INTEGER} \times \text{STRING})^\star$ (the new hash table, for insertion) or STRING (for retrieval).

6.4.3 Results

Figure 6.2 shows the percentage of assignment submissions where feedback gave useful results for our random samples of incorrect student submissions. Any non-zero percentage of useful feedback means that the feedback would have helped at least one student understand what was wrong with their code, and our results exceed this humble goal: 80% of the assignment submissions provide some useful feedback 100% of the time, and in total 96% of our benchmarks provide some useful feedback.

The hash table assignment submissions performed the worst, indicating that we need to work on better methods for generating interesting invariants for complex data structures that improve on treating them like a flat list. Still, the results are promising,

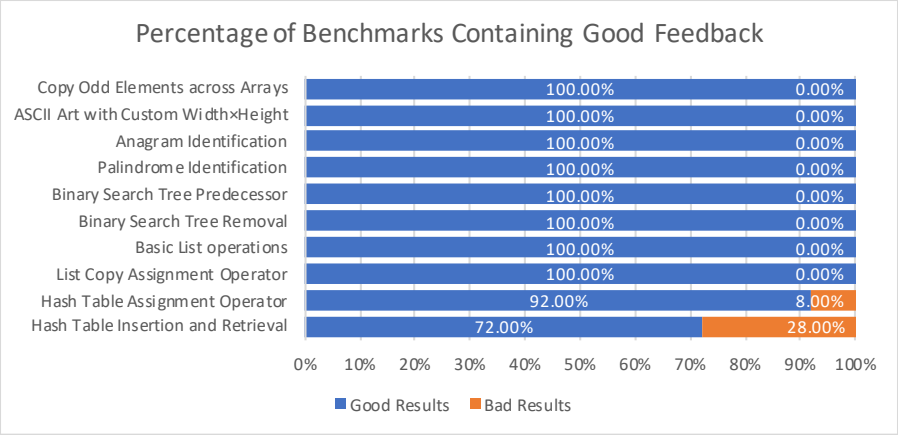


Figure 6.2: Ratio of assignment submissions for which at least one useful invariant was output as feedback.

and one could potentially add better, domain-specific observer functions to improve the performance of our method.

For each assignment, while we provide useful feedback the vast majority of the time, some of the lines of feedback that we return can be spurious—students must look through the results to find the useful feedback and determine what is helpful for them. Figure 6.3 presents this per-feedback view for our results. There, for each assignment, we calculate the percentage of *individual* invariants displayed that are of use to the student. It is promising that we tend to provide a majority of useful lines of feedback, but we believe that focusing on minimizing our outputs could be a good area of future exploration.

The outlier in the figure is the Hash Table Insertion and Retrieval benchmark. It is interesting to note that when two hash tables are available to compare (as is true in the Hash Table Assignment Operator benchmark), we provide better results significantly more often. We believe that this is because two hash tables allow us to derive invariants that relate the structures even though our method does not understand the structure itself, while only having a single hash table does not give our method any leverage for inferring interesting invariants.

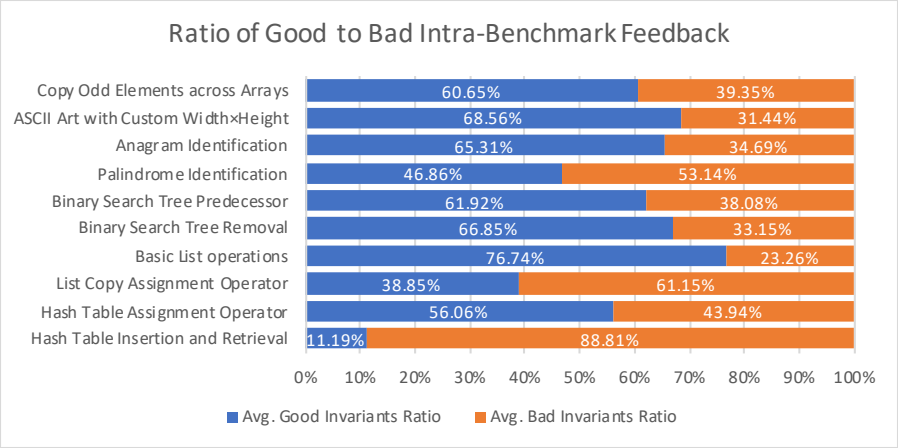


Figure 6.3: Average ratio of useful to non-usefull invariants per assignment submission. While we generate useful feedback for most submissions, some invariants that we output are spurious and do not provide information about the student's error.

Table 6.2: Average running times of our method, over 25 separate student submissions.

Benchmark	Avg. Compilation Time (s)	Avg. Execution Time (s)	Avg. Invariant Inference Time (s)
Copy Odd Elements across Arrays	1.910	0.994	25.251
ASCII Art with Custom Width×Height	2.953	0.095	20.487
Anagram Identification	2.985	1.311	27.413
Palindrome Identification	2.444	0.443	21.790
Binary Search Tree Predecessor	2.915	0.982	25.594
Binary Search Tree Removal	2.730	1.163	22.221
List Average	3.268	0.186	22.111
List Copy Assignment Operator	3.015	0.350	18.575
Hash Table Assignment Operator	3.699	1.541	23.235
Hash Table Insertion and Retrieval	3.266	1.170	20.234

Running Time

Table 6.2 contains average running times for our method. The largest invariant inference time is for the Anagram Identification benchmark, and this is likely due to the implications that we ask our invariant inference engine to infer. It is clear from the results that this output can be generated with only a small delay, and this will allow our method to be used as a part of today's autograders, which function in an "almost real-time" environment.

6.4.4 Sample Outputs

Figure 6.4 shows two representative output examples from two of our more complicated benchmarks; we omit the full output for space reasons.

Binary Search Feedback.

In Figure 6.4a we have a snippet of the output feedback from the Binary Search Tree Predecessor—we show the **Fail** set, which indicates the properties of the code that hold whenever the student's solution and the instructor's solution do not agree. A student, upon receiving this feedback, will go down the list to try to understand what went wrong. The first four invariants eliminate potential complications due to edge cases (the given value was the minimum value in the tree, or the maximal value, or the root of the tree, or not in the tree at all). The student can be assured, then, that none of these edge cases are the problem and can dismiss them from consideration.

The fifth invariant states that the return value in every failing case was the sentinel value 0, which represents the assertion that the predecessor for the given node does not exist, while the sixth invariant shows that this is not the case for the reference solution. Finally, the seventh, eighth, and tenth invariants pinpoint that the depth of the predecessor node is the culprit, and that in all failing cases the predecessor node is an ancestor of the given node—a case that the student apparently forgot to account for.

List Average Feedback.

For the List Average example in Figure 6.4b, a student may proceed as follows. Going through **Fail**, a student will discover that the instructor's average for each of the failing cases is never 0. That same invariant, however, does not appear about the student's average, indicating that the student is sometimes returning 0 when they

should not. The issue always lies with a non-empty list case, shown by the invariant involving the list length. Perhaps the most helpful single invariant is the one relating the absolute value of the student's average to the absolute value of the instructor's average, and the student's is always smaller—the issue with this student's code is that the average is being computed using an `int` variable rather than a `double` variable, chopping off any necessary decimal values. If the final invariant of **Fail** does not indicate this issue to the student, the *combination* of the third invariant of **Pass** and the second invariant of **Fail** (which both talk about the absolute value of the average with respect to some constant) will—from this the student could deduce that something could be going wrong when the average is supposed to be a number *between* 0.0 and 1.0.

6.4.5 Discussion

From these results we conclude that the feedback is already useful given the current implementation, but could be improved in several ways that we will investigate in future work:

- We could improve the inferencer's understanding of complex data structures, allowing it to generate more interesting and expressive invariants.
- We could improve the translation of invarariants into natural language and make the implications of those invariants more clear to the student. Some of the reasoning that we went through for the example outputs above, for instance, seem automatable and something that we could provide to the student directly.
- We could improve the ability to prune useless invariants while keeping the useful ones (noting that this does not necessarily always mean keeping stronger invariants

```
+----+
|Fail|
+----+
The element to search for was the minimum value in the tree == false
The element to search for was the maximum value in the tree == false
The element to search for was the root of the tree == false
The element to search for was not in the tree == false
Your result for the predecessor == 0
The instructor's result for the predecessor != 0
The depth of the predecessor node >= 0
The depth of the original node >= 2
Your result for the predecessor != The instructor's result for the predecessor
The depth of the predecessor node < The depth of the original node

...
```

(a) Snippet of output for Binary Search Tree Predecessor.

```
+----+
|Fail|
+----+
.average() called on the instructor's list != 0
.average() called on your list >= 0.0
size(The input list) >= 1
.average() called on the instructor's list != .average() called on your list
.average() called on the instructor's list > .average() called on your list

+----+
|Pass|
+----+
.average() called on the instructor's list == .average() called on your list
.average() called on the instructor's list == .average() called on your list
.average() called on the instructor's list >= 1.0

+-----------+
|Fail U Pass|
+-----------+
.average() called on the instructor's list != 0
.average() called on your list >= 0.0
.average() called on the instructor's list >= .average() called on your list

...
```

(b) Snippet of output for List Average.

Figure 6.4: Sample output snippets.

and pruning weaker ones). In addition, we could attempt to rank the invariants by how useful we think they will be to the student.

6.5 Related Work

Our method is a combination of automated feedback and program invariant inference, so we describe related work in both areas in this section. We begin with automated feedback. A common theme among related work is that students are given commands to follow or solution fragments to implement rather than the less-revealing feedback that we provide (e.g., [132, 25, 78, 72, 37]).

6.5.1 Automated Feedback

Repair- and synthesis-based methods are among the most interesting in this space, but there are many different ways to provide feedback. We give a broad review in this subsection.

Literature reviews

Keuning et al. [79, 80] provide two literature reviews on automated feedback generation, and we refer the interested reader to these for further information; we provide a shorter such survey below. One interesting note from these surveys is that less than 50% of papers on the subject of feedback generation perform some sort of technical analysis—we go against the majority in this chapter. 60% of feedback methods focus on solution errors, including our own method.

Repair and synthesis

Singh et al. [132] use constraint-based synthesis to compute minimal corrections to students' incorrect solutions to introductory programming assignments. They require an error model describing the potential corrections, whereas our method may be re-used across different kinds of programs. Their tool was able to synthesize corrections for 64% of their corpus. Kaleeswaran et al. [73] use recurrent neural networks to provide repair-based feedback on syntax errors. They formalize the problem of finding fixes for syntax errors in student submissions as a token sequence learning problem using the recurrent neural networks.

Yi et al. [150] try several existing program repair tools out of the box on real student code and find that they do not perform well; they find that allowing for partial repairs improves their results. They perform a user study and discover that "while the graders seem to gain benefits from repairs, novice students do not seem to know how to effectively make use of generated repairs as hints"—this is an interesting argument against popular program-repair based methods that are at the forefront of feedback generation.

Analysis

Program analysis-based methods are another interesting means of observing the semantic meaning of a program. Gulwani et al. [66] target the performance of solutions to introductory programming assignments. Their tool allows a teacher to define an algorithmic strategy by specifying certain key values that should occur during the execution of an implementation, and uses dynamic analysis to test whether a given program matches this specification. In contrast, we do not require any such pre-population of strategies: our method can be useful without initial knowledge of how

students will solve a problem. Blau and Moss [37] use dataflow analysis and AST searching to find "silent flaws" in Java programs—e.g., instance variables that should have been static constants, uses of `public` that should really have been `private`, etc. These are all pre-programmed patterns to search for.

Transition-based

Piech et al. [116] encourage students to transition from one partial solution to another that is closer to the correct solution. They compute how accurately they predict the next transition that a student should take, given a corpus of solution transitions from students to use as ground truth. Similarly, Rivers and Koedinger [124] generate personalized hints for students, even when given states that have not occurred in their training data. This method requires a reference solution and test cases, and computes transitions to the correct solution. Our method differs in that we place no requirements on the shape of a correct solution.

Similarity and clustering

Keuning et al. [78] provide strategy-based feedback: their method recognizes when student solutions are similar to one of the instructor-given models, and incrementally fills in correct statements or adds holes for the student to fill in. Again, we only require one correct solution for our method, rather than several. Kaleeswaran et al. [72] provide feedback by clustering student submissions by solution strategy, and then doing program equivalence checking within each cluster. Feedback is generated based on a correct solution from each cluster.

Other tools

Antonucci et al. [25] create an incremental hint system for assignment. Hints are created in advance from the source code of an exercise's reference solution; there are two kinds of hints ("textual" and "code-revealing"), and they are all created ahead of time by the instructor. D'antoni et al. [51] focus on feedback for automata-construction assignments, and find that "providing either counterexamples or hints is judged as helpful, increases student perseverance, and can improve problem completion time". Our invariant-based method is seen as a mixture of counterexamples and hints, and so this is a promising result. In contrast to these two previous techniques, our method is general and more automatic. Wrenn and Krishnamurthi [148] presents a method for testing the validity of students' handmade input/output examples for a particular program; it requires multiple correct and buggy implementations to be premade to compare against. This method is quite complementary to our own—instead of automatically calculating input/output invariants, this tool requires manually-specified inputs and outputs.

6.5.2 Program Invariants and Their Inference

Daikon

Ernst et al. [55] introduce Daikon, arguably the most well-known dynamic invariant inference tool and method. Instead of requiring programmers to annotate their code, Daikon automatically infers "likely" invariants about a program by observing traces of program executions. The Daikon method calculates invariants about both single variables and combinations of variables at specific program points, and there are many built-in invariants that Daikon checks for. We make use of Daikon in this work.

Daikon has not existed statically, and has been improved over the years. Csallner

et al. [50], for example, present DySy, which employs symbolic execution along with the standard dynamic analysis to produce more appropriate invariants. Demsky et al. [53] harness Daikon to create repair inconsistency properties of data structure due to, e.g., data corruption.

Daikon, however, is not the only contender in this space; we list some other techniques as well as applications below. We note that we chose Daikon because it was easy to use and the first method that we tried—it is possible that some of the techniques or tools listed below could prove useful to our method, and we leave that exploration to future work.

Other popular tools

Flanagan et al. [58] introduce the Extended Static Checker for Java, a tool which allows for automatic verification of annotated source code. The existence of invariants is refuted by program verification over the declared invariants, and other warnings (which can be interpreted as likely invariants) are also generated about the code.

Hangal and Lam [67] introduces the DIDUCE tool: similar to Daikon, it detects likely invariants across many points in a program. Instead of relying on a static set of test cases like Daikon, DIDUCE constantly instruments a running program on real data; this allows for the detection of anomalies in a program's execution.

Machine learning

Brun and Ernst [39] employ machine learning to provide a method for finding program properties that are indicative of errors. Program properties are provided by an analysis and then machine learning is applied to select those properties that will most likely result in an error state.

Applications

Csallner and Smaragdakis [49] infer invariants in the face of interfaces and method overriding. Subtyping introduces consistency issues, and the authors discuss a solution. Baliga et al. [30, 31] learn invariants about Linux kernel data structures, and uses those invariants to detect the presence of a rootkit. Livshits and Zimmermann [96] compute invariants over software revision histories in order to discover common bug fixes and application-specific patterns. Astorga et al. [27] present a method for precondition generation based on dynamic analysis. Observer methods are used to translate advanced properties and/or non-primitive types into primitive types.

Improvements to Supported Invariants

Li et al. [92] study the consistency of dynamic invariant detectors by means of second-order constraints. Second-order constraints are identified and used to prune inconsistent invariants. Nguyen et al. [108] notice that support for invariant inference of disjunctions (e.g., arising from conditional statements) is limited. They harness dynamic invariant inference in order to generate disjunctive invariants over numerical domains, and then verifies the correctness those invariants statically.

All lies and jest

Still, a man hears what he wants to hear

And strong rejects the rest

Chapter 7

Conclusions and Future Work

In this dissertation, I presented several case studies to support my thesis that there is room for improvement in the realm of program similarity. I made contributions along two different axes:

1. Methods: syntactic similarity across languages, and semantic similarity; and
2. Applications: plagiarism detection, incremental analysis, and student feedback.

As technologies improve, these ideas can be made more useful and can be applied in more areas. In addition, new domains will bring about new applications of similarity.

In the near-term, there are a couple of improvements that can be made to our student feedback work. The first is to provide better English translation of the resulting Daikon invariants—at the moment, we are doing simpler, text-based substitution of the results; this does seem to work well as a starting point, but it is likely that more advanced invariants would benefit from a parsing-based translation approach.

Another topic for investigation is the ordering of the invariants that we output—currently we return them in the order that we get them back from Daikon. Some

invariants are more important than others, though, and a potential solution could involve using statistics to calculate how "surprising" a particular invariant is, relative to the others that we output. We could measure this surprise in terms of how often it appears in other invariant sets (e.g., **Pass**, **Refs**, etc.), or perhaps even in terms of the model count of the formula.

In the long-term, I see two low-hanging areas for improvement, and they are both based off the student feedback project. The first is application-specific feedback. Currently our feedback method is quite general, and this works well for introductory programming classes. Determining what it would take to get a feedback method for an upper-division course (e.g., operating systems) or for a non-general-purpose programming language (e.g., ARM assembly) would be an interesting challenge. Both of those examples would require new kinds of invariants—for example, those that take registers and flags into consideration—as well as new ways to present those invariants to a student in an intuitive manner.

The second long-term project I envision is using dynamic invariants for program similarity. To the best of my knowledge, dynamic program invariants have not been used to detect similarity, though there are ways to compare logical formulae (e.g., SAT solving, model counting, etc.). Dynamic invariants seem like an excellent tool for fingerprinting code, and appear to be a new point in the program similarity space.

Bibliography

[1] 2014. Python implementation of a graph-similarity-grading algorithm. https://stackoverflow.com/questions/12122021/python-implementation-of-a-graph-similarity-grading-\algorithm.

[2] 2017. ANTLR. Retrieved March 17, 2017 from http://www.antlr.org/

[3] 2017. balanced-match. https://github.com/juliangruber/balanced-match.

[4] 2017. chess. https://bitbucket.org/rsb/chesscomnotifier/overview.

[5] 2017. Chrome Extension Developers Under a Barrage of Phishing Attacks. https://tech.slashdot.org/story/17/08/11/221203/chrome-extension-developers-under-a-barrage-of-phishing\-attacks.

[6] 2017. CVE-2013-1624 : The TLS implementation in the Bouncy Castle Java library before 1.48 and C# library before 1.8 does not properly conside. Retrieved March 17, 2017 from http://www.cvedetails.com/cve/CVE-2013-1624/

[7] 2017. dateformat. https://github.com/felixge/node-dateformat.

[8] 2017. emoji-helper. https://github.com/johannhof/emoji-helper/blob/master/src/popup.js.

[9] 2017. Google Removes Chrome Extension Used in Banking Fraud. https://threatpost.com/google-removes-chrome-extension-used-in-banking-fraud/127469/.

[10] 2017. java-lame. Retrieved March 17, 2017 from https://github.com/nwaldispuehl/java-lame

[11] 2017. k-cup-deals. https://addons.mozilla.org/en-US/firefox/addon/keurig-k-cup-deals/versions/.

[12] 2017. lamejs. Retrieved March 17, 2017 from https://github.com/zhuker/lamejs

[13] 2017. Malicious Chrome Extensions Steal Passwords & CPU Power. https://duo.com/decipher/malicious-chrome-extensions-steal-passwords-and-cpu.

[14] 2017. simple-translate. https://github.com/sienori/simple-translate/blob/master/simple-translate/background.js.

[15] 2017. toxiclibs. Retrieved March 17, 2017 from http://toxiclibs.org/

[16] 2017. url-join. https://github.com/jfromaniello/url-join.

[17] 2017. yallist. https://github.com/isaacs/yallist/blob/master/test/basic.js.

[18] 2017. ZXing. Retrieved March 17, 2017 from https://github.com/zxing/zxing

[19] 2018. jpf-symbc/src/examples. https://github.com/SymbolicPathFinder/jpf-symbc/tree/master/src/examples.

[20] 2020. UndefinedBehaviorSanitizer — Clang 11 documentation. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[21] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. 2015. An exact graph edit distance algorithm for solving pattern recognition problems. In *4th International Conference on Pattern Recognition Applications and Methods 2015*.

[22] Farouq Al-Omari, Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. 2012. Detecting Clones Across Microsoft .NET Programming Languages. In *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*. 405–414. https://doi.org/10.1109/WCRE.2012.50

[23] Stephen F. Altschul. 2011. Global and Local Sequence Alignment. Lecture Notes.

[24] Stephen F. Altschul and Bruce W. Erickson. 1986. Optimal sequence alignment using affine gap costs. *Bulletin of Mathematical Biology* 48, 5 (1986), 603–616. https://doi.org/10.1007/BF02462326

[25] Paolo Antonucci, Christian Estler, Durica Nikolić, Marco Piccioni, and Bertrand Meyer. 2015. An incremental hint system for automated programming assignments. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 320–325.

[26] Steven Arzt and Eric Bodden. 2014. Reviser: Efficiently Updating IDE-/IFDS-based Data-flow Analyses in Response to Incremental Program Changes. In *ICSE 2014*. 288–298.

[27] Angello Astorga, P Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. 2019. Learning stateful preconditions modulo a test generator. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 775–787.

[28] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-based model counting for string constraints. In *International Conference on Computer Aided Verification*. Springer, 255–272.

[29] Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. 2018. Parameterized model counting for string and numeric constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 400–410.

[30] A. Baliga, V. Ganapathy, and L. Iftode. 2008. Automatic Inference and Enforcement of Kernel Data Structure Invariants. In *2008 Annual Computer Security Applications Conference (ACSAC)*. 77–86. https://doi.org/10.1109/ACSAC.2008.29

[31] A. Baliga, V. Ganapathy, and L. Iftode. 2011. Detecting Kernel-Level Rootkits Using Data Structure Invariants. *IEEE Transactions on Dependable and Secure Computing* 8, 5 (Sept. 2011), 670–684. https://doi.org/10.1109/TDSC.2010.38

[32] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S Păsăreanu, and Tevfik Bultan. 2016. String analysis for side channels with segmented oracles. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 193–204.

[33] Lucas Bang, Nicolás Rosner, and Tevfik Bultan. 2018. Online Synthesis of Adaptive Side-Channel Attacks Based On Noisy Observations. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 307–322.

[34] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 368–377.

[35] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering* 33, 9 (2007).

[36] Philip Bille. 2005. A Survey on Tree Edit Distance and Related Problems. *Theor. Comput. Sci.* 337, 1-3 (June 2005), 217–239. https://doi.org/10.1016/j.tcs.2004.12.030

[37] Hannah Blau and J Eliot B Moss. 2015. FrenchPress gives students automated feedback on java program flaws. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 15–20.

[38] Tegan Brennan, Nestan Tsiskaridze, Nicolás Rosner, Abdulbaki Aydin, and Tevfik Bultan. 2017. Constraint normalization and parameterized caching for quantitative program analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 535–546.

[39] Y. Brun and M.D. Ernst. 2004. Finding latent code errors via machine learning over program executions. In *Proceedings. 26th International Conference on Software Engineering*. IEEE Comput. Soc, Edinburgh, UK, 480–490. https://doi.org/10.1109/ICSE.2004.1317470

[40] Martin D. Carroll and Barbara G. Ryder. 1988. Incremental data flow analysis via dominator and attribute update. In *POPL '88*. ACM, 274–284.

[41] Xiao Cheng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. 2016. On the feasibility of detecting cross-platform code clones via identifier similarity. In *Proceedings of the 5th International Workshop on Software Mining*. ACM, 39–42.

[42] Xiao Cheng, Zhiming Peng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. 2016. Mining revision histories to detect cross-language clones without intermediates. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 696–701.

[43] Michel Chilowicz, Etienne Duris, and Gilles Roussel. 2009. Syntax tree fingerprinting for source code similarity detection. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. IEEE, 243–247.

[44] Christopher L. Conway, Kedar S. Namjoshi, Dennis Dams, and Stephen A Edwards. 2005. Incremental algorithms for inter-procedural analysis of safety properties. In *International Conference on Computer Aided Verification*. Springer, 449–461.

[45] Keith Daniel Cooper. 1983. *Interprocedural Data Flow Analysis in a Programming Environment*. Ph.D. Dissertation. Rice University, Houston, TX, USA. AAI8314924.

[46] Keith D. Cooper and Ken Kennedy. 1984. Efficient Computation of Flow Insensitive Interprocedural Summary Information. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*. 247–258.

[47] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2001. An Improved Algorithm for Matching Large Graphs.

[48] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*. 238–252.

[49] Christoph Csallner and Yannis Smaragdakis. 2006. Dynamically discovering likely interface invariants. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*. ACM Press, Shanghai, China, 861. https://doi.org/10.1145/1134285.1134435

[50] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*. ACM, 281–290.

[51] Loris D'antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. 2015. How can automatic feedback help students construct automata? *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 9.

[52] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[53] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. 2006. Inference and enforcement of data structure consistency specifications. In *Proceedings of the 2006 international symposium on Software testing and analysis - ISSTA'06*. ACM Press, Portland, Maine, USA, 233. https://doi.org/10.1145/1146238.1146266

[54] Kyle Dewey, Lawton Nichols, and Ben Hardekopf. 2015. Automated data structure generation: Refuting common wisdom. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 32–43.

[55] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123.

[56] Antonio Filieri, Corina S Păsăreanu, and Willem Visser. 2013. Reliability analysis in symbolic pathfinder. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 622–631.

[57] Antonio Filieri, Corina S Pasareanu, and Guowei Yang. 2015. Quantification of software changes through probabilistic symbolic execution (N). In *Automated*

Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on. IEEE, 703–708.

[58] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2013. PLDI 2002: Extended Static Checking for Java. *SIGPLAN Not.* 48, 4S (July 2013), 22–33. https://doi.org/10.1145/2502508.2502520

[59] Enrique Flores, Alberto Barrón-Cedeno, Paolo Rosso, and Lidia Moreno. 2012. DeSoCoRe: Detecting source code re-use across programming languages. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Demonstration Session*. Association for Computational Linguistics, 1–4.

[60] Deqiang Fu, Yanyan Xu, Haoran Yu, and Boyang Yang. 2017. Wastk: A weighted abstract syntax tree kernel method for source code plagiarism detection. *Scientific Programming* 2017 (2017).

[61] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*. ACM, 321–330.

[62] Debin Gao, Michael K Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*. Springer, 238–255.

[63] Jaco Geldenhuys, Matthew B Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 166–176.

[64] Vida Ghodssi. 1983. *Incremental analysis of programs*. Ph.D. Dissertation. University of Central Florida.

[65] David Gitchell and Nicholas Tran. 1999. Sim: a utility for detecting similarity in computer programs. In *ACM SIGCSE Bulletin*, Vol. 31. ACM, 266–270.

[66] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2014. Feedback generation for performance problems in introductory programming assignments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 41–51.

[67] S. Hangal and M. S. Lam. 2002. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. 291–301. https://doi.org/10.1145/581376.581377

[68] Manuel Hermenegildo, German Puebla, Kim Marriott, and Peter J Stuckey. 2000. Incremental analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 2 (2000), 187–223.

[69] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.

[70] Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 81–92.

[71] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. 2010. Code similarities beyond copy & paste. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 78–87.

[72] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 739–750.

[73] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. 2014. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 266–276.

[74] D. Kalman, M. Pistoia, G. Podjarny, O. Tripp, and O. Weisman. 2012. Incremental static analysis. `https://www.google.com/patents/US20120054724` US Patent App. 12/873,219.

[75] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[76] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2013. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware.. In *USENIX Security Symposium*.

[77] Vineeth Kashyap and Ben Hardekopf. 2014. Security signature inference for javascript-based browser addons. In *CGO*. ACM, 219.

[78] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2014. Strategy-based feedback in a programming tutor. In *Proceedings of the Computer Science Education Research Conference*. ACM, 43–54.

[79] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 41–46.

[80] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)* 19, 1 (2018), 3.

[81] Daeyoung Kim, Amruta Gokhale, Vinod Ganapathy, and Abhinav Srivastava. 2016. Detecting plagiarized mobile apps using API birthmarks. *Automated Software Engineering* 23, 4 (2016), 591–618.

[82] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. 2011. MeCC: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 301–310.

[83] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. F a C o Y: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 946–957.

[84] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *International static analysis symposium*. Springer, 40–56.

[85] Danai Koutra, Ankur Parikh, Aaditya Ramdas, and Jing Xiang. 2011. Algorithms for Graph Similarityand Subgraph Matching. `https://www.cs.cmu.edu/~jingx/docs/DBreport.pdf`. (2011).

[86] Nicholas A Kraft, Brandon W Bonds, and Randy K Smith. 2008. Cross-language Clone Detection. In *SEKE*. 54–59.

[87] Andreas Krall and Thomas Berger. 1994. Incremental Flow Analysis.

[88] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 301–309.

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

[90] S. Kullback and R. A. Leibler. 1951. On Information and Sufficiency. *Ann. Math. Statist.* 22, 1 (03 1951), 79–86. `https://doi.org/10.1214/aoms/1177729694`

[91] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL 2012*. 96.

[92] Kaituo Li, Christoph Reichenbach, Yannis Smaragdakis, and Michal Young. 2013. Second-order constraints in dynamic invariant inference. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. ACM Press, Saint Petersburg, Russia, 103. https://doi.org/10.1145/2491411.2491457

[93] Yujian Li and Zhang Chenguang. 2011. A metric normalization of tree edit distance. *Frontiers of Computer Science in China* 5, 1 (2011), 119–125. https://doi.org/10.1007/s11704-011-9336-2

[94] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 872–881.

[95] Ben Livshits and Salvatore Guarnieri. 2010. *Gulfstream: Incremental Static Analysis for Streaming JavaScript Applications*. Technical Report. Microsoft Research.

[96] Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 296–305. https://doi.org/10.1145/1081706.1081754

[97] Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. 2014. Verification Modulo Versions: Towards Usable Verification. In *PLDI '14*. 294–304.

[98] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. *An Incremental Points-to Analysis with CFL-Reachability*. Springer Berlin Heidelberg, Berlin, Heidelberg, 61–81.

[99] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering* 12 (2017), 1157–1177.

[100] Andrian Marcus and Jonathan I Maletic. 2001. Identification of high-level concept clones in source code. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*. IEEE, 107–114.

[101] Victor J. Marin and Carlos R. Rivero. 2018. Towards a Framework for Generating Program Dependence Graphs from Source Code. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics (SWAN 2018)*. ACM, New York, NY, USA, 30–36. https://doi.org/10.1145/3278142.3278144

[102] Thomas J. Marlowe and Barbara G. Ryder. 1990. An Efficient Hybrid Algorithm for Incremental Data Flow Analysis. In *POPL '90*. 184–196.

[103] Vítor T Martins, Daniela Fonte, Pedro Rangel Henriques, and Daniela da Cruz. 2014. Plagiarism detection: A tool survey and comparison. In *OASIcs-OpenAccess Series in Informatics*, Vol. 38. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[104] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. 2013. Scalable and Incremental Software Bug Detection. In *FSE 2013*. 554–564.

[105] Lefteris Moussiades and Athena Vakali. 2005. PDetect: A clustering approach for detecting plagiarism in source code datasets. *The computer journal* 48, 6 (2005), 651–661.

[106] Rashmi Mudduluru and Murali Krishna Ramanathan. 2014. *Efficient Incremental Static Analysis Using Path Abstraction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 125–139.

[107] Sandhya Narayanan and S Simi. 2012. Source code plagiarism detection and performance analysis using fingerprint based distance measure method. In *Computer Science & Education (ICCSE), 2012 7th International Conference on*. IEEE, 1065–1068.

[108] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. Using Dynamic Analysis to Generate Disjunctive Invariants. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 608–619. https://doi.org/10.1145/2568225.2568275 event-place: Hyderabad, India.

[109] José Oncina and Pedro Garcia. 1992. Identifying regular languages in polynomial time. In *Advances in structural and syntactic pattern recognition*. World Scientific, 99–108.

[110] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.

[111] Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* 20, 3 (2013), 391–425.

[112] Mateusz Pawlik and Nikolaus Augsten. 2015. Efficient Computation of the Tree Edit Distance. *ACM Trans. Database Syst.* 40, 1 (2015), 3:1–3:40. https://doi.org/10.1145/2699485

[113] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Inf. Syst.* 56 (2016), 157–173. https://doi.org/10.1016/j.is.2015.08.004

[114] David M Perry, Dohyeong Kim, Roopsha Samanta, and Xiangyu Zhang. 2019. SemCluster: clustering of imperative programming assignments based on quantitative semantic features. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 860–873.

[115] Quoc-Sang Phan, Lucas Bang, Corina S Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of adaptive side-channel attacks. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 328–342.

[116] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*. ACM, 195–204.

[117] Li ping Zhang and Dong sheng Liu. 2013. AST-based multi-language plagiarism detection method. In *Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference on*. IEEE, 738–742.

[118] L. L. Pollock and M. L. Soffa. 1989. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering* 15, 12 (1989), 1537–1549.

[119] Bruno Prado, Kalil A Bispo, and Raul Andrade. 2018. X9: An Obfuscation Resilient Approach for Source Code Plagiarism Detection in Virtual Learning Environments.. In *ICEIS (1)*. 517–524.

[120] Lutz Prechelt, Guido Malpohl, and Michael Phlippsen. 2000. JPlag: Finding plagiarisms among a set of programs. (2000).

[121] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[122] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165 – 1199.

[123] Matthias Rieger. 2005. *Effective clone detection without language barriers*. Ph.D. Dissertation. University of Bern.

[124] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64.

[125] Chanchal Kumar Roy and James R Cordy. 2007. *A survey on software clone detection research*. Technical Report 541. Queen's School of Computing.

[126] C. K. Roy and J. R. Cordy. 2008. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *2008 16th IEEE International Conference on Program Comprehension*. 172–181. https://doi.org/10.1109/ICPC.2008.41

[127] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. 2018. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 354–365.

[128] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Di Yang, Pedro Martins, Hitesh Sajnani, Pierre Baldi, and Cristina V Lopes. 2019. Towards automating precision studies of clone detectors. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 49–59.

[129] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877

[130] Yutaka Sasaki et al. 2007. The truth of the F-measure. (2007).

[131] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 76–85.

[132] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. *Acm Sigplan Notices* 48, 6 (2013), 15–26.

[133] T.F. Smith and M.S. Waterman. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* 147, 1 (1981), 195 – 197. https://doi.org/10.1016/0022-2836(81)90087-5

[134] Jeong-Woo Son, Seong-Bae Park, and Se-Young Park. 2006. Program Plagiarism Detection Using Parse Tree Kernels. In *PRICAI 2006: Trends in Artificial Intelligence*, Qiang Yang and Geoff Webb (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1000–1004.

[135] A. L. Souter and L. L. Pollock. 2001. Incremental call graph reanalysis for object-oriented software maintenance. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*. 682–691.

[136] Kathryn T Stolee, Sebastian Elbaum, and Matthew B Dwyer. 2016. Code search with input/output queries: Generalizing, ranking, and assessment. *Journal of Systems and Software* 116 (2016), 35–48.

[137] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. 2016. Code relatives: detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 702–714.

[138] Lisan Sulistiani and Oscar Karnalim. 2019. ES-Plag: Efficient and sensitive source code plagiarism detection tool for academic environment. *Computer Applications in Engineering Education* 27, 1 (2019), 166–182.

[139] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. [n. d.]. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 476–480.

[140] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 320–331.

[141] Narjes Tahaei and David C Noelle. 2018. Automated Plagiarism Detection for Computer Programming Exercises Based on Patterns of Resubmission. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, 178–186.

[142] Ankur Taly, John C Mitchell, Mark S Miller, Jasvir Nagra, et al. 2011. Automated analysis of security-critical javascript apis. In *2011 IEEE Symposium on Security and Privacy*. IEEE, 363–378.

[143] Omer Tripp, Pietro Ferrara, and Marco Pistoia. 2014. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 49–59.

[144] David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *ICFP (ICFP '10)*. ACM, New York, NY, USA, 51–62.

[145] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. 2007. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica* 48, 1 (2007), 37–66.

[146] Willem Visser. 2016. What makes killing a mutant hard. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 39–44.

[147] Willem Visser and Corina S Păsăreanu. 2017. Probabilistic programming for Java using symbolic execution and model counting. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists*. ACM, 35.

[148] John Wrenn and Shriram Krishnamurthi. 2019. Executable examples for programming problem comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. 131–139.

[149] Wuu Yang. 1991. Identifying syntactic differences between two programs. *Software: Practice and Experience* 21, 7 (1991), 739–755.

[150] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 740–751.

[151] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 141–151.