Differential Program Analysis

Nimrod Partush
Differential Program Analysis

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Nimrod Partush

Submitted to the Senate
of the Technion — Israel Institute of Technology Tamuz
5776 Haifa August 2016
This research was carried out under the supervision of Prof. Eran Yahav, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:

The Technion’s funding of this research is hereby acknowledged.
# Contents

## List of Figures

## Abstract

## 1 Introduction

### 1.1 Abstract Semantic Differencing via Correlating Programs

#### 1.1.1 Motivation

#### 1.1.2 Problem Definition

#### 1.1.3 Existing Techniques

#### 1.1.4 Key Aspects of the Approach

#### 1.1.5 Main Contributions

### 1.2 Abstract Semantic Differencing via Speculative Correlation

#### 1.2.1 Motivation

#### 1.2.2 Problem Definition

#### 1.2.3 Existing Techniques

#### 1.2.4 Key Aspects of the Approach

#### 1.2.5 Main Contributions

### 1.3 Interprocedural Semantic Change Impact Analysis

#### 1.3.1 Motivation

#### 1.3.2 Problem Definition

#### 1.3.3 Existing Techniques

#### 1.3.4 Key Aspects of the Approach

#### 1.3.5 Main Contributions

### 1.4 Statistical Similarity of Binaries

#### 1.4.1 Motivation

#### 1.4.2 Problem Definition

#### 1.4.3 Existing Techniques

#### 1.4.4 Key Aspects of the Approach

#### 1.4.5 Main Contributions

## 2 Abstract Semantic Differencing via Correlating Programs

### 2.1 Overview

### 2.2 Preliminaries
2.3 Concrete Semantics .............................................. 21
  2.3.1 Concrete State Differencing .............................. 21
  2.3.2 Concrete Correlating Semantics ......................... 23

2.4 Abstract Correlating Semantics ............................... 23
  2.4.1 Abstract Correlating State .............................. 23
  2.4.2 Abstract Correlating Semantics ......................... 24
  2.4.3 Dynamic Partitioning .................................... 25
  2.4.4 Widening ................................................. 26
  2.4.5 Differencing for Abstract Correlating States .......... 27

2.5 Correlating Program ........................................... 29
  2.5.1 Construction of $P \bowtie P'$ ............................ 30

2.6 Evaluation ..................................................... 31
  2.6.1 Prototype Implementation ................................. 31
  2.6.2 Results .................................................... 31

2.7 Related Work .................................................. 37

3 Abstract Semantic Differencing via Speculative Correlation 39
  3.1 Overview ..................................................... 39
    3.1.1 Motivating Example ...................................... 39
    3.1.2 Running Example ......................................... 44
    3.1.3 Uninterpreted Functions ................................. 49
    3.1.4 Path Boundedness ...................................... 50

  3.2 Speculative Correlating Semantics ......................... 50
    3.2.1 Preliminaries ........................................... 51
    3.2.2 Concrete Correlating Semantics ....................... 51
    3.2.3 Abstract Correlating Semantics ....................... 52
    3.2.4 Dynamic Partitioning and Widening .................... 54

  3.3 Speculative Search Algorithm .............................. 54
    3.3.1 Iterative Speculative Correlation ...................... 56
    3.3.2 Speculative Advancement Over All Interleavings ...... 58
    3.3.3 Comparing Abstractions to Find Minimal Difference 58

  3.4 Evaluation ................................................... 59
    3.4.1 Prototype Implementation ............................... 60
    3.4.2 Benchmark Results ....................................... 60
    3.4.3 State of The Art Comparison ............................ 62
    3.4.4 Quality of Semantic Diff ............................... 64

  3.5 Related Work ................................................ 65

4 Interprocedural Semantic Change Impact Analysis ............. 69
  4.1 Overview ..................................................... 69
  4.2 Background .................................................. 71
4.2.1 Programming Language ............................................. 71
4.2.2 Semantics .............................................................. 73
4.3 Problem Statement ...................................................... 74
4.3.1 Representing Changes .............................................. 74
4.3.2 Semantic Change Impact ........................................... 75
4.3.3 Dataflow-based Change Impact Analysis ......................... 76
4.4 Incorporating Change Semantics ..................................... 79
4.4.1 Anytime Algorithm ................................................... 80
4.5 Inferring Equalities .................................................... 82
4.5.1 Product Program Construction .................................... 83
4.5.2 Invariant inference .................................................. 84
4.6 Implementation and Evaluation ..................................... 85
4.6.1 Implementation ...................................................... 85
4.6.2 Evaluation ............................................................. 85
4.6.3 Change-Semantic Aware Analysis ................................. 87
4.6.4 Incremental Analysis ............................................... 87
4.6.5 Anecdotes .............................................................. 89
4.7 Related Work ............................................................ 91
4.8 Related Work ............................................................ 92

5 Statistical Similarity of Binaries ..................................... 95
5.1 Overview ................................................................. 95
5.2 Strand-Based Similarity by Composition .......................... 99
5.2.1 Similarity By Composition ......................................... 99
5.2.2 Procedure Decomposition to Strands ............................ 99
5.2.3 Statistical Evidence ................................................. 101
5.2.4 Local and Global Evidence Scores ............................... 102
5.3 Semantic Strand Similarity .......................................... 102
5.3.1 Similarity Semantics ............................................... 102
5.3.2 Encoding Similarity as a Program Verifier Query ............ 104
5.4 Evaluation ............................................................... 105
5.4.1 Test-Bed Creation and Prototype Implementation ............. 106
5.4.2 Using Vulnerable Code as Queries ............................... 106
5.4.3 Testing Different Aspects of the Problem Separately .......... 107
5.4.4 Evaluating Our Method ........................................... 107
5.4.5 Enabling the Use of Powerful Semantic Tools .................. 108
5.5 Results ................................................................. 109
5.5.1 Finding Heartbleed .................................................. 110
5.5.2 Decomposing Our Method into Sub-methods .................... 111
5.5.3 Comparison of TRACY and Esh ................................. 112
5.5.4 Evaluating BinDiff .................................................. 112
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Two versions of a sign computing procedure (top) and a correlating program for the two procedures (bottom)</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>Two individual abstract states: $\sigma$ for sign and $\sigma'$ for sign' (top) and correlating abstract state $\sigma_{\Delta\triangleleft}$ for the joint program state (bottom)</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Syntactically different yet semantically equivalent procedures.</td>
<td>9</td>
</tr>
<tr>
<td>1.4</td>
<td>(Partial) Graphic description of an iteration of the speculative algorithm over print_numbers and print_numbers'</td>
<td>10</td>
</tr>
<tr>
<td>1.5</td>
<td>Inter-procedural code containing semantic preserving refactoring as well as functionality changes. The highlighted lines are the only ones impacted by the changes.</td>
<td>11</td>
</tr>
<tr>
<td>1.6</td>
<td>Image vs. Code Similarity by Composition. Query image $i_q$ is similar to the target image $t$. Query code $q_2$ is similar to the target code $t$. Images courtesy of Irani et al. [BI06]</td>
<td>15</td>
</tr>
<tr>
<td>2.1</td>
<td>Two implementations of a sign computing procedure ($\text{sign}$ &amp; $\text{sign}'$) and their correlating program ($\text{sign} \triangleleft \Delta\triangleleft \text{sign}'$).</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>Two equivalent versions of a looping program for partial array summation.</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>$\text{sum} \triangleleft \Delta\triangleleft \text{sum}'$</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Single path differentiation candidates</td>
<td>25</td>
</tr>
<tr>
<td>2.5</td>
<td>Original sum, Patched sum’’ and correlating sum $\triangleleft \Delta\triangleleft$ sum’’</td>
<td>26</td>
</tr>
<tr>
<td>2.6</td>
<td>$\Delta$ computation geometrical representation.</td>
<td>29</td>
</tr>
<tr>
<td>2.7</td>
<td>Firefox’s nsGenericDOMDataNode module with patch.</td>
<td>33</td>
</tr>
<tr>
<td>2.8</td>
<td>Difference for Firefox’s nsGenericDOMDataNode, (a) with a single polyhedra; (b) set of polyhedra.</td>
<td>33</td>
</tr>
<tr>
<td>2.9</td>
<td>Original and patched version of Coreutils pr.c’s char_to_clump procedure</td>
<td>34</td>
</tr>
<tr>
<td>2.10</td>
<td>Difference for the char_to_clump procedure from Coreutils’s pr.c</td>
<td>34</td>
</tr>
<tr>
<td>2.11</td>
<td>Two versions of the logicalValue procedure taken from [PDEP08].</td>
<td>35</td>
</tr>
<tr>
<td>2.12</td>
<td>Difference for logicalValue</td>
<td>35</td>
</tr>
<tr>
<td>2.13</td>
<td>Original and patched version of Coreutils md5sum.c’s bsd_split_3 procedure</td>
<td>36</td>
</tr>
<tr>
<td>2.14</td>
<td>Difference for bsd_split_3</td>
<td>36</td>
</tr>
<tr>
<td>3.1</td>
<td>Original (top) and patched (bottom) version of Coreutils seq.c’s print_numbers procedure</td>
<td>40</td>
</tr>
</tbody>
</table>
3.2 (Partial) Graphic description of an iteration of the speculative algorithm over print_numbers and print_numbers'. ................................................................. 42
3.3 get_sha1-basic procedure code fragment taken from Git's sha1_name.c .... 49
3.4 Procedures featuring several (overall changes > k) local syntactic changes (each change < k), along with correlating abstract state (column 3) adhering to executing the blocks from both programs .................................................. 51
3.5 Patch for Git archive-zip.c's write_zip_entry procedure ....................... 61
3.6 Patched version of Firefox's vulnerable SetTextInternal function .......... 62
3.7 Original and patched version of [SSCA13] worked example ................... 63
3.8 Output-equivalent versions of Coreutils seq.c's print_numbers procedure .. 65
3.9 Original and patched (+) version of Coreutils pr.c's char_to_clump procedure 66

4.1 Example. Changes made to Coreutils in files paste.c and sort.c. The lines with − and + represent deleted and added lines respectively. .................. 70
4.2 Inference rules for computing DEPENDONVAR and DEPENDONNODE. The input is a program Prog. ................................................................. 77
4.3 Inference rules for dataflow based change impact analysis. The highlighted antecedents are relevant for change-semantics aware analysis. ............. 78
4.4 Motivating example for anytime algorithm. ........................................... 81
4.5 Pathological example which requires large k ........................................... 81
4.6 Product procedures for inferring equivalences [LMSH13] ....................... 83
4.7 Example illustrating side conditions for inference. .................................... 84
4.8 Comparing results for different analysis depths for the tinyvm project ...... 88
4.9 Change illustrating an extract constant to variable in histo commit c723a4 .... 89
4.10 Change illustrating a failed refactoring in tinyvm commit 716936 ............. 89
4.11 Change illustrating a loop conversion in mdp commit 00c2ad ................. 90
4.12 Change illustrating a goto-elimination refactoring in tinyvm commit 378ce6 .. 90

5.1 Heartbleed vulnerability code snippets. ................................................. 96
5.2 Semantically similar strands. ............................................................... 97
5.3 Syntactically similar but semantically different strands. ......................... 97
5.4 Experiment #1 – Successfully finding 12 procedure variants of Heartbleed. .. 110
5.5 All vs. all experiment result in heat map form (pixel intensity = similarity GES measure). ................................................................. 114
5.6 Wrapper exit_cleanup procedure from sort.c of Coreutils 8.23 ............ 115
5.7 DEFINE_SORT_FUNCTIONS macro for creating “template” procedures in 1s.c of Coreutils 8.23 ................................................................. 115
Abstract

The evolution of software is an emerging research topic, receiving much attention and focus, as every line of code written today is more likely to belong to an existing piece of software. Identifying and modeling the semantic impact of these patches in a sound and precise way is a paramount goal yet to be fulfilled. This work is an effort to generate sound, precise and scalable analysis methods for modeling the differences and measuring the similarity between programs.

This work first focuses on computing semantic differences in numerical programs where the values of variables have no a-priori bounds, and uses abstract interpretation to compute an over-approximation of program differences. Computing differences and establishing equivalence under abstraction requires abstracting relationships between variables in the two programs. Towards that end, a correlating program is first constructed, in which these relationships can be tracked, and then a correlating abstract domain is used to compute a sound approximation of these relationships. The construction of a correlating program relies on syntactic similarity. To better establish equivalence between correlated variables and precisely capture differences, the abstract domain has to represent non-convex information using a partially-disjunctive abstract domain. To balance precision and cost of this representation, the domain over-approximates numerical information while preserving equivalence between correlated variables by dynamically partitioning the disjunctive state according to equivalence criteria.

This approximation of difference can be computed over any interleaving of the two programs. However, the choice of interleaving can significantly affect precision. As the construction of the correlating programs limited the approach to syntactically similar programs, this work further presents a speculative search algorithm that aims to find an interleaving of the two programs with minimal abstract semantic difference. This method is unique as it allows the analysis to dynamically alternate between several interleavings.

Next, to allow a scalable computation of difference for whole programs, change impact analysis (CIA) is employed. CIA refers to the task of determining the set of program statements affected by a program change. In this work, the notion of change impact is formalized in terms of the dynamic trace semantics of two program versions. The approach shows how to leverage differential invariants to make dataflow based CIA change-semantics aware, thereby improving precision in the presence of semantics preserving changes. A novel anytime algorithm allows costly differential invariant generation to be applied incrementally to refine the set of impacted statements.

Finally, the problem of finding similar procedures in stripped binaries is addressed. A new statistical approach is presented for measuring similarity between two procedures. The notion of
similarity makes it possible to find similar code even when it has been compiled using different compilers, or has been slightly modified. The main idea is to use similarity by composition: decompose the code into smaller comparable fragments, define semantic similarity between fragments, and use statistical reasoning to lift fragment similarity into similarity between procedures.

All of the approaches specified in this work are evaluated using real-world code projects. The evaluation shows the benefit of each approach and its ability to produce a sound and precise description of difference and similarity.
Chapter 1

Introduction

In this thesis, the question of identifying, describing and quantifying the difference and similarity between different program versions is addressed. The work is composed of four chapters:

1. Abstract Semantic Differencing via Correlating Programs (Chapter 2) addresses the issue of precisely and soundly characterizing difference over single procedures, while relying on syntactic similarity.

2. Abstract Semantic Differencing via Speculative Correlation (Chapter 3) addresses the question of finding difference over program versions where syntactic similarity is scarce. The syntactic requirement is relaxed by introducing a speculative search algorithm.

3. Inter-procedural Semantic Change Impact Analysis (Chapter 4) addresses the computation of difference over whole programs, through the reduction of the produced result to impacted program lines. The approach leverages semantic information and differential assertions over procedure versions to precisely compute a more minimal set of impacted lines.

4. Statistical Similarity of Binaries (Chapter 5) addresses the problem of finding similarity of source code procedures, after being compiled to binary code and stripped of all syntactic information (procedure and identifier names, etc.). The chapter addresses the challenge of finding the same procedure, or a version of it, even when different compilers are used.

In the rest of this section, we introduce each chapter separately.

1.1 Abstract Semantic Differencing via Correlating Programs

1.1.1 Motivation

Understanding the semantic difference between two versions of a program is invaluable in the process of software development. A developer applying a patch is often interested in answering questions such as: (i) Does the patch add/remove the desired functionality? (ii) Does the patch introduce other, unexpected, behaviors? (iii) Which regression tests should be run? Answering these questions manually is difficult and time consuming.

Semantic differencing has received much attention in classic work (e.g., [Hor90, HPR89, Hoa69]) and has recently seen growing interest for various applications ranging from testing con-
current programs [CGS12] understanding software upgrades [JOX10] to automatic generation of security exploits [BPSZ08].

1.1.2 Problem Definition
We define the problem of semantic differencing as follows: Given a pair of programs \((P, P')\) that agree on the number and type of input and output variables, for every execution \(\pi\) of \(P\) that originates from an input \(i\) and a corresponding execution \(\pi'\) of \(P'\) that originates from the same input \(i\), our goal is: (i) to check whether \(\pi\) and \(\pi'\) have the same output, i.e., are output-equivalent, and (ii) where there is a difference in output variables, to provide a description of the difference.

1.1.3 Existing Techniques
Existing techniques mostly offer solutions based on under-approximation, the most prominent of which is regression testing, which provides limited assurance of behavior equivalence while consuming significant time and compute resources. Other approaches for computing semantic differences [PDEP08, RE11] rely on symbolic execution techniques, may miss differences, and are generally unable to prove equivalence. Previous work for equivalence checking [GS09] relies on unsound bounded model checking techniques to prove (input-output) equivalence of two closely related numerical programs, under certain conditions (see Section 2.7 for more details).

1.1.4 Key Aspects of the Approach
We present an approach based on abstract interpretation [CC77] for producing a sound representation of changed program behaviors and proving equivalence between a program and its patched version. Our method focuses on abstracting relationships between variables in both versions, allowing us to achieve a precise description of the difference. Our solution is sound in the sense that it computes an over-approximation of the difference between the two versions, therefore guaranteeing equivalence when no difference is found.

We focus on output equivalence in the final state. This is sufficient as mid-execution output can be modeled as added variables in the final state. This limitation means that we assume all program executions to be finite (i.e., equivalence/difference holds if indeed both executions terminate).

To address the problem of semantic differencing for infinite-state programs, we employ abstract interpretation. Though the notion of difference is well defined in the concrete case, defining and soundly computing it under abstraction is challenging:

- Differencing requires correlation of different program executions. The abstraction must be able to capture and compare only the input-equivalent executions, and avoid comparing ones that are not input-equivalent.

- Equivalence of abstract output values does not entail concrete value equivalence.

To address these challenges, we introduce two new concepts: (i) correlating program - a single program \(P \bowtie P'\) that captures the behaviors of both \(P\) and \(P'\) in a way that facilitates
abstract interpretation; (ii) correlating abstract domain - a domain for tracking relationships between variables in $P$ and variables in $P'$ using $P \triangleright P'$.

**Correlating Program** We create a single program which captures the behavior of both the original program and its patched version. A correlating program $P \triangleright P'$ contains both programs’ flow and data; however, program flow is arranged so to reflect a (simple) matching between the stages of the two programs. This matching is key for precision as otherwise we will not be able to maintain equivalence throughout the entire run of the program, particularly in the face of loops. Figure 1.1 depicts the correlating program for two versions of a sign computing procedure. The interleaving of the two procedures into a single correlating procedure relies on syntactic similarity, as syntactically equal statements from sign and sign' are coalesced in sign$\triangleright$sign'. The problem of program matching is orthogonal and can be addressed via techniques of varying precision and complexity. These techniques include syntactic diff [HM75] and execution indexing [XSZ08], among others. In this work we employ a simple matching strategy to achieve better precision as described in Section 2.5. We found this technique to be sufficient for our experiments.

```plaintext
int sign(int x) {
  int sgn;
  if (x < 0)
    sgn = -1;
  else
    sgn = 1;
  return sgn;
}

int sign'(int x') {
  int sgn';
  if (x' < 0)
    sgn' = -1;
  else
    sgn' = 1;
  if (x' == 0)
    sgn' = 0;
  return sgn';
}

int sign$\triangleright$sign'(int x) {
  int x' = x;
  int sgn;
  int sgn';
  (x < 0) -> sgn = -1;
  (x' < 0) -> sgn' = -1;
  !(x < 0) -> sgn = 1;
  !(x' < 0) -> sgn' = 1;
  (x' == 0) -> sgn' = 0;
}
```

Figure 1.1: Two versions of a sign computing procedure (top) and a correlating program for the two procedures (bottom)

**Correlating Abstraction** Abstracting relationships allows us to focus on differences while over-approximating (whenever necessary for scalability) equivalent behaviors. We abstract variables of both programs together, starting off by assuming equality over all matched variables (variable matching is discussed in Section 2.3). Thus we can reflect relationships without necessarily knowing the actual value of variables. We focus on numerical programs and use numerical domains such as Octagon [Min06] and Polyhedra [CH78] to capture the relationship between variables. Our current implementation does not track pointer equivalences, but such equivalences can be tracked by a using a correlating shape analysis domain [ARR+07]. To
σ = \{ x < 0, \text{sgn} \mapsto -1 \} \lor \{ x \geq 0, \text{sgn} \mapsto 1 \}

\sigma' = \{ x' < 0, \text{sgn}' \mapsto -1 \} \lor \{ x' > 0, \text{sgn}' \mapsto 1 \} \lor \{ x' = 0, \text{sgn}' \mapsto 0 \}

\sigma_{\triangleleft} = \{ x = x' \mapsto \top, \text{sgn} = \text{sgn}' \mapsto \top \} \lor \{ x = x' = 0, \text{sgn} = 1 \neq \text{sgn}' = 0 \}

Figure 1.2: Two individual abstract states: \( \sigma \) for sign and \( \sigma' \) for sign' (top) and correlating abstract state \( \sigma_{\triangleleft} \) for the joint program state (bottom)

maintain equivalence as much as possible, our domain was designed to represent non-convex information (e.g., so we will not immediately lose equivalence when taking a condition of the form \( x \neq 0 \) into account). We use a power-set domain of convex sub-states. Our domain uses a partitioning strategy that abstracts together states that have the same set of equivalent variables, thus avoiding exponential blowup (as explained in Section 2.4). This strategy helps us preserve equivalence even across widening. Therefore our domain may over-approximate numerical information as long as equivalence between correlated variables is preserved. Figure 1.2 shows a state \( \sigma_{\triangleleft} \) in our abstract domain, which over-approximates the abstract states \( \sigma \) and \( \sigma' \) for each of the programs sign and sign' from Figure 1.1. \( \sigma_{\triangleleft} \) abstracts away specific values of variables, but is able to track equivalence and capture difference for the \( x = x' = 0 \) case.

1.1.5 Main Contributions

Our main contributions in Chapter 2 are:

- We present a novel approach for computing abstract semantic difference between a program \( P \) and a patched version of the program \( P' \). We focus on numerical programs where the values of variables have no a-priori bounds.

- We reduce the problem of analyzing the two programs \( P, P' \) to the problem of analyzing a single correlating program \( P \bowtie P' \) that captures the behavior of \( P \) and \( P' \).

- We present a correlating abstract domain that captures an over-approximation of the difference between \( P \) and \( P' \) by tracking relationships between variables in \( P \bowtie P' \). The domain applies a partitioning strategy for scaling the analysis while maintaining precision in equivalence.

- We implemented our approach in a tool based on the LLVM compiler infrastructure and the APRON numerical abstract domain library, and applied it to several real-world
programs. Our evaluation shows that the tool often manages to establish equivalence, reports useful approximations of semantic differences when differences exist, and reports only a few false differences.

1.2 Abstract Semantic Differencing via Speculative Correlation

1.2.1 Motivation

Understanding the semantic difference between two versions of a program is invaluable in the process of software development [LVH10]. A developer applying a patch is often interested in understanding the effect of the patch in terms of added/removed program behaviors. In particular, proving that a patch does not change observable behavior (i.e., equivalence checking), has numerous applications. These include translation validation [PSS98], verifying super-optimization [SSCA13], and finding protocol bugs [RE11], to name a few.

Semantic differencing is a fundamental problem [Hoa69] that has recently seen growing interest [LVH10, LHKR12, HLP +13, LMSH13, PY13, SSAAC13], with applications including testing of concurrent programs [CGS12], understanding software upgrades [JOX10], differential assertion checking [LMSH13], and automatic generation of security exploits [BPSZ08].

1.2.2 Problem Definition

We define the problem of semantic differencing as follows: Given a pair of programs $(P, P')$ that agree on the number and type of input and output variables, for every execution $\pi$ of $P$ and a corresponding execution $\pi'$ of $P'$ that both originate from the same input our goal is: (i) to check whether $\pi$ and $\pi'$ have the same output, i.e., are output-equivalent, and (ii) when there is a difference in output variables, to provide a description of the difference.

1.2.3 Existing Techniques

The current state of the art applies only to small code and works at the granularity of a single function. We advance the state of the art as we are able to statically produce sound results over complex looping functions with vast syntactic difference—a contribution unmatched in related work. Existing techniques to semantic differencing mostly offer solutions based on under-approximation, of which regression testing is the most prominent and provides limited assurance of behavior equivalence while consuming significant time and compute resources. Other approaches for computing semantics differences [PDEP08, RE11] rely on symbolic execution techniques, may miss differences, and are generally unable to prove equivalence. Previous work for equivalence checking [GS09] rely on unsound bounded model checking techniques to prove (input-output) equivalence of two closely related numerical programs, under certain conditions (see Section 3.5 for more details).

1.2.4 Key Aspects of the Approach

We present an approach based on abstract interpretation [CC77] for computing a sound representation of changed program behaviors and proving equivalence between a program and its patched version. Our method focuses on abstracting relationships between variables in both versions, allowing us to achieve a precise description of the difference and prove equivalence. We produce
a characterization of difference in the form of linear equations over program variables. Our approach computes an over-approximation of the difference between the two versions, therefore guaranteeing equivalence when no difference is found.

**Equivalence under Abstraction** In contrast to techniques that limit loop iterations or the range of inputs [JL94, LHKR12, PDEP08, RE11], our approach uses abstraction to handle infinite-state programs. The main challenge is to abstract program executions in a way that enables capturing difference. In particular, to establish equivalence under abstraction, the states of the two programs have to be abstracted together such that the relationship between them is preserved.

This leads to the notion of a correlating semantics in which \( P \) and \( P' \) are executed together and share a joint program state (in which we can track relationships between their values). Given the correlating semantics, there are many possible (joint) executions of \( P \) and \( P' \), depending on how their steps interleave. Because the programs manipulate disjoint parts of the shared state, the result of each of these different interleavings is exactly the same.

However, under abstraction, these different interleavings may have dramatically different results. The intuitive explanation is that the abstraction of relationships between variables is only able to capture small (or structured) differences. Once the execution of one program goes too far beyond the other program, the abstraction loses precision, and would not be able to preserve equivalence even if it is restored. This is true even without loops, but is of critical importance when loops are present.

One such interleaving is the sequential composition \( P; P' \) used in [LHKR12, PDEP08, RE11] to reason about information flow. Using the sequential composition is typically not feasible under abstraction as the distance between program executions becomes too big for the abstraction to capture with sufficient precision.

The challenge of keeping precise differences under abstraction therefore becomes that of deciding — which interleaving of the executions of \( P \) and \( P' \) we should pick such that the abstraction is able to precisely capture the difference. Figure 1.3 depicts two (input-output) equivalent versions of a sequence printing procedure from Coreutils’s seq.c. There exists an interleaving in which the two programs maintain equivalence, but relying on syntactic cues to find this interleaving would not succeed, as most of the procedures’ lines differ syntactically. Creating a correlating program of the two programs, for instance, would yield a sequential composition.

The interleaving problem is cardinal in equivalence checking, and other approaches include establishing a simulation relation using execution traces [SSCA13], program composition using syntactic similarity [PY13], relying on recursion rules [GS09] and brute-force searching (for non-looping code) [PDEP08, RE11].

One trivial solution is to try all possible interleavings, which is similar to analyzing a concurrent program \( P \parallel P' \). However, this incurs an exponential cost, and severely limits the applicability of the technique. Another trivial solution is to use the sequential composition \( P; P' \) (used e.g., in [TA05] to reason about information flow). However, this is typically not
static void print_numbers (long first, long last, ...) {
  long i;
  for (i = 0; /* empty */; i++) {
    long x = first + i * STEP;
    if (last < x) break;
    if (i) fputs (separator, stdout);
    printf (fmt, x);
  }
  if (i) fputs (terminator, stdout);
}

Coreutils seq.c v6.9

Figure 1.3: Syntactically different yet semantically equivalent procedures.

feasible under abstraction as the distance between program executions becomes too big for the abstraction to capture with sufficient precision.

Speculative Correlation Instead, we use abstraction-guided speculative correlation to find an interleaving of $P$ and $P'$ that tracks differences with sufficient precision. The basic idea is to break the abstract interpretation into short segments in which we speculatively try all interleavings, followed by a greedy choice of the segment that produced the minimal abstract difference as the one to be extended further. Algorithm 1.1 describes a high-level pseudo-code for the speculative correlation algorithms. The pseudo-code consists of two steps running in a fixed-point loop: (i) Speculatively explore interpreting $k$ steps, distributed over both programs ($k$ is a parameter of the algorithm), resulting in $k + 1$ different abstract states and (ii) compare the $k + 1$ results to pick the one with minimal difference. This is a standard chaotic iteration algorithm with a worklist that contains dual-program states. The search for an appropriate interleaving is incorporated as part of a fixed-point abstract interpretation analysis, where the search drives the analysis.

Algorithm 1.1 Speculative Exploration

```plaintext
while worklist ≠ ∅ do
  result ← Speculate(P, P', worklist, state₀ₐ, k)
  (worklist, state₀ₐ) ← Find_Minimal_Diff_Result(results)
end while
```

Figure 1.4 shows a graphic representation of the abstract states produced from analyzing the programs in two possible interleavings. These are (part of) the result of step (i) of Algorithm 1.1. From the two interleavings, step (ii) of the algorithm would pick the interleaving which results in the most precise abstract state.

1.2.5 Main Contributions

Our main contributions in Chapter 3 are:
We present a novel framework for computing abstract semantic differences between a program and its patched version. The main idea is to use a speculative abstract semantics that attempts to find the right correlation between programs such that their equivalence can be captured under a given abstract domain.

We present an abstract correlating semantics that executes a program $P$ and its patched version $P'$ within some bounded window of divergence, and capture relationships between variables in the two programs.

We provide a speculative search algorithm that aims to find an interleaving of the executions of $P$ and $P'$ with the minimal abstract difference. This method is unique as it allows the analysis to dynamically alternate between several interleavings. This is fundamentally different from previous methods, which compute a single interleaving and use it.

We have implemented our approach in a tool called SCORE and applied it to a number of real-world examples. Our evaluation shows that we compute precise approximations of semantic differences, and report only a few false differences.

### 1.3 Interprocedural Semantic Change Impact Analysis

#### 1.3.1 Motivation

Software is omnipresent and is constantly evolving to address the needs of users, add and improve features, eliminate bugs, improve software design, etc. As software evolves faster than ever, it requires rigorous techniques to ensure that a change does not modify existing
behaviors in unintended ways. This need is critical considering the extremely high costs of fixing introduced bugs in program patches.

### 1.3.2 Problem Definition

Given two (inter-procedural) programs \((P, P')\), Change Impact Analysis (CIA) determines the set of program elements that may be impacted by a syntactic change over the two programs. A change can be the removal of lines from \(P\) and/or the addition of lines to \(P'\). Semantic-aware change impact analysis allows the leveraging of semantics, through establishing the equivalence of variables and procedures using semantic tools [LMSH13, PDEP08, BCK11, Ben04, PY14], to reduce the set of impacted lines, while maintaining soundness.

### 1.3.3 Existing Techniques

Several approaches have emerged to ensure that software changes do not modify existing behaviors in unintended ways. Code review [MKAH14], regression testing [RH97, GEM15], test-suite augmentation [PYRK11, PDEP08, MC12], code contracts [JM97, BLS04], regression verification [PMH+14, GS09] and verification modulo versions [LLFB14] are just a few of the approaches used to ensure the quality of a change; they all benefit from change impact analysis.

```c
void print_product_version() {
    int locale = locale_ok();
    print_major_version(locale);
    - print_minor_version(locale, \n');
    + print_minor_version(locale, \0');
}

void locale_ok () {
    - return setlocale(LC_ALL ,""); ? 1 : 0;
    + return !!setlocale(LC_ALL ,"");
}

void print_major_version (int locale) {
    - if (locale) {
    + if (locale) {
        printf("%s",
locale_format("8");
    }
}

void print_minor_version (int locale, char delim) {
    - if (locale) {
    + if (locale) {
        printf("%s%c",
locale_format(".12"),
locale_format("8"),
locale_format("8"),
delim);
    }
}
```

Figure 1.5: Inter-procedural code containing semantic preserving refactoring as well as functionality changes. The highlighted lines are the only ones impacted by the changes.

Traditional approaches are coarse-grained and operate at the level of types and classes [AOH05, AOH04], or files [GEM15] to retain soundness. Fine-grained techniques that aim to work at the level of statements are typically based on performing a dataflow analysis [RHS95] on the whole program to propagate the change along data and control flow edges [BPRT13, Leh11, CS15]. Although such techniques are lightweight, they fail to take the semantics of the change into account. For example, applying such standard CIA on the example in Figure 1.5 would fail to account for the fact that the changes in `locale_ok` and `print_major_version` preserve semantics, and would mark all the lines of these procedures as impacted. This impact will, in turn, flow to all of the lines in the program.

Hence such techniques cannot distinguish between changes that a user expects to have a shallow impact on existing code (e.g., code restructuring or a new feature) from those that
have substantial impact on existing code (e.g., changing the functionality). Producing a sound, tight bound for the impact of a change can help with code review and regression-testing efforts: changes with substantial impact can be prioritized for more rigorous code reviews; regression tests that only execute non-impacted statements can be considered equivalent from the perspective of exercising the changes.

Differential analysis methods that operate at the level of a single procedure, as introduced in Section 1.1 and Section 1.2, cannot soundly capture interprocedural change-impact. For example, applying differential analysis on the procedures in Figure 1.5 would determine all the callees to be semantically equivalent. This would falsely suggest that functionality has been preserved, even though the \texttt{print\_minor\_version} procedure was impacted despite being untouched syntactically.

1.3.4 Key Aspects of the Approach

A key observation is that different changes have different complexities and require different level of scrutiny. \textit{Refactoring} the code is expected to globally preserve the semantics of the original program, while some behavioral changes can be observed locally in the refactored code. \textit{Bug fixes} are expected to change the global behavior of the program but \textit{only} for the faulty behaviors. Adding \textit{new features} should not interfere with the existing features of the software. \textit{Optimizations} should preserve the original behavior. Changes in \textit{libraries and APIs} may impact some of the API functionalities but it is important that they do not impact other APIs. Motivated by this observation, we design an analysis that exploits differential program verification to find, based on the program and change semantics, what parts of the program are impacted by the change.

We propose the use of \textit{differential program verification} \cite{LMSH13, PDEP08, BCK11, Ben04, PY14} to make CIA change-semantics aware. Differential (or relational) verification generalizes program verification and invariant inference to the context of two programs \cite{Ben04, LMSH13}. We use differential invariant inference to construct equivalence relationships between the two program versions, which are then fed to a dataflow-based change propagation algorithm. To integrate program verifiers, we first formalize the notion of change impact in terms of the dynamic trace semantics of two programs. We leverage differential invariants to make dataflow-based CIA change-semantics aware, thereby soundly improving the precision in the presence of changes that (partially) preserve program semantics. In Figure 1.5, differential assertions will capture, that at the point of return from \texttt{locale\_ok} the return value is semantically equivalent before and after the refactoring. The same will be determined for the value of the condition in \texttt{print\_major\_version}. The semantic equivalence will cut off the flow of change-impact which falsely determined the lines of the caller and subsequent callees as impacted.

We propose a novel anytime algorithm that allows applying costly differential invariant generation incrementally to refine the set of impacted statements. We implemented a prototype in \textsc{SymDiff}, and provide results for several real-world changes from open-source projects and several benchmark programs.
1.3.5 Main Contributions

Our main contributions in Chapter 4 are:

1. We are the first to precisely formalize when a statement in a program is *impacted* by a program change, in terms of the dynamic trace semantics of the program.

2. We describe mechanisms to incorporate *semantics of change* into dataflow analysis [RHS95] in the form of various equivalences (Section 4.4).

3. We propose a novel *anytime* algorithm [DB88, ZR95] that allows applying costly differential invariant generation incrementally to refine the set of impacted statements (Section 4.4.1).

4. We have implemented a prototype in SYMDIFF [LHKR12, LMSH13] and evaluated our technique for 322 real-world changes collected from GitHub open-source projects and several standard benchmark programs used in prior research [HFGO94].

1.4 Statistical Similarity of Binaries

1.4.1 Motivation

During December 2014, several vulnerabilities were discovered in the prominent implementation of the network time protocol (NTP) ntpd [Clu]. As this implementation is the de facto standard, many products from major vendors were affected, including RedHat’s Linux distribution, Apple’s OSX and Cisco’s 5900x switches. Because some of these vulnerabilities were introduced many years ago, different versions have already been ported and integrated into many software packages, ready to be deployed in binary form to home computers, enterprise servers and even appliance firmware.

A security savvy individual, or more commonly a security aware company, would want to use a sample of the vulnerable product (in binary form) to search for the vulnerability across all the software installed in the organization, where source-code is mostly not available. Unfortunately, automatically identifying these software packages is extremely challenging. For example, given a sample from the Debian Linux distribution, finding other (recent or older) vulnerable versions of it is already hard. This is because even though older distributions were probably compiled with the same compiler, gcc, they used older gcc versions, producing syntactically different binary code. Trying to find OSX applications is even harder, as they are commonly compiled with a different compiler (CLang), and the firmware of an appliance, using Intel chips, might be compiled with icc (the Intel C compiler), such that the resulting binary procedures differ vastly in syntax. We address this challenge by providing an effective means of searching for semantically similar procedures, at assembly code level.

1.4.2 Problem definition

Given a query procedure \( q \) and a large collection \( T \) of (target) procedures, in binary form, our goal is to quantitatively define the similarity of each procedure \( t \in T \) to the query \( q \). The main challenge is to define a *semantic* notion of similarity that is precise enough to avoid
false positives, but is flexible enough to allow finding the code in any combination of the following scenarios: (i) the code was compiled using different compiler versions; (ii) the code was compiled using different compiler vendors; and (iii) a different version of the code was compiled (e.g. a patch). We require a method that can operate without information about the source code and/or tool-chain used in the creation of the binaries only of procedures in binary form.

1.4.3 Existing techniques

Previous work on clone detection in binaries [SWP+09] can overcome different instruction selections, but is unable to handle syntactic differences beyond single instructions. Binary code search [DY14] is mostly syntactic and therefore fails to handle differences that result from different compilers. Equivalence checking and semantic differencing techniques [RE11, HLP+13, PY13, PY14] operate at the source-code level and require a matching (labeling of variables) to operate. Furthermore, they do not provide quantitative measures of similarity when procedures are not equivalent. Dynamic equivalence checking techniques [RE11] are hard to apply as they require obtaining sufficient coverage for each procedure. A dynamic approach for binary code search [EWCB14] coerces execution using a randomized environment to achieve coverage, but suffers from a high false positive rate, and does not address or evaluate patching. Data-driven equivalence checking [SSCA13] is aimed at translation validation and is limited to short segments of assembly code.

**Similarity by composition** We draw inspiration from Boiman and Irani’s [BI06] work on image similarity, where the two key ideas are that one image is similar to another if it can be composed using regions of the other image, and that this similarity can be quantified using statistical reasoning.

Figure 1.6(a) and (b) illustrate the idea of similarity by composition for images. Looking at the three images in Figure 1.6(a), we wish to determine which of the two query images, $iq_1$ and $iq_2$, is more similar to the target image, $it$. Our intuition is that $iq_2$ is probably more similar to $it$.

Figure 1.6(b) provides some explanation for this intuition. Looking at this figure, we identify similar regions in $iq_2$ and $it$ (marked by the numbered outlined segments). Although the images are not identical, their similarity stems from the fact that query image $iq_2$ can be composed using significant (and maybe transformed) regions from target image $it$. In contrast, image $iq_1$ shares only two small regions with $it$, and indeed we intuitively consider it to be different. These vague notions are made precise within a statistical framework that lifts similarity between significant regions into similarity between images.

In this chapter, we show that the general framework of “similarity by composition” can also be applied to code. Specifically, we consider assembly code extracted from stripped binaries (with no debug information). Figure 1.6(c), (d), and (e) show partial assembly code of three procedures. Snippets (c) and (e) are taken from the OpenSSL procedure vulnerable to “Heartbleed”, and were compiled using different compilers: gcc 4.9 and CLang 3.5, respectively. Snippet (d) is taken from an unrelated procedure in Coreutils and compiled using gcc 4.9. For
Finding similarity between the procedures using syntactic techniques is challenging, as different compilers can produce significantly different assembly code. Instead, we decompose each procedure to small code segments we call strands (a strand is a basic-block slice, as explained in Section 5.2.2), semantically compare the strands to uncover similarity, and lift the results into procedures.

In Figure 1.6(c) & (e), the query code, \( q_2 \), and the target code, \( t \), share three matching strands, numbered in the figure as 1, 2, and 3. Each strand is a sequence of instructions, and strands are considered as matches when they perform an equivalent computation. In the figure, we mark matching strands using the same circled number. Two syntactically different strands can be equivalent. For example, the strands numbered 2 in \( q_2 \) and \( t \) differ syntactically but are equivalent (up to renaming and ignoring the change of r9). Furthermore, strands need not be syntactically contiguous. This is because they are based on data-flow dependencies.

Figure 1.6: Image vs. Code Similarity by Composition. Query image \( iq_2 \) is similar to the target image \( it \). Query code \( q_2 \) is similar to the target code \( t \). Images courtesy of Irani et al. [BI06]
rather than on syntactic properties. For example, strand 3 in the query procedure $q_2$ (mov r12, rbx; lea rdi, [r12+3]) matches the strand mov r13, rbx; lea rcx, [r13+3] in the target procedure $t$. In contrast, the code in Figure 1.6(d) only matches the single strand 1 in the target.

### 1.4.4 Key Aspects of the Approach

We present a novel notion of similarity for procedures that is based on the following key components:

**Decomposing the procedure into strands:** We decompose procedures into smaller segments we refer to as strands, which are feasible to compare.

**Comparing strands:** We use a program verifier [BCD+05] to check whether two strands are semantically equivalent by assuming input equivalence and checking if intermediate and output values are the same. When they are not equivalent, we define a quantitative notion of strand similarity based on the proportion of matching values to the total number of values in the strand.

**Statistical reasoning over strands:** We present a statistical framework for reasoning about similarity of whole procedures using strand similarity. We compute a global similarity evidence score between procedures using the sum of the local evidence scores ($LES$) between strands. A key aspect of our framework is that we amplify the similarity scores of unique strands, expressed by a high $LES$, and diminish the significance of “common” strands (with respect to the target database examined) as they are less indicative of similarity.

### 1.4.5 Main Contributions

Our main contributions in Chapter 5 are:

- A framework for reasoning about similarity of procedures in stripped binaries. The main idea is to decompose procedures into strands, perform semantic comparison of them, and lift strand similarity into similarity between procedures using statistical techniques.

- A technique for checking input-output equivalence of strands of code, where all variables are unlabeled.

- A statistical model that quantifies the probability of similarity between procedures by examining the similarity of their strands.

- A prototype implementation in a tool called Esh, which is publicly available at github.com/techsrl/esh. We compare Esh to previous binary code-search techniques using challenging search scenarios that combine patched and non-patched versions of real-world vulnerable procedures in binary form, compiled with different compiler versions and vendors. These experiments show that Esh achieves significantly better results.
Chapter 2

Abstract Semantic Differencing via Correlating Programs

In this chapter, we aim to produce sound and precise abstract representation of the difference between two syntactically similar programs. Given two similar procedures \( P \) and \( P' \), which have the same input and output variables, we either prove they are equivalent or produce a description of the semantic difference in the form of linear inequalities [Min06, CMWC09] over program variables. The description is sound i.e. no differences missed, but false positives are possible. The precision of the result will be directly correlated with the size of the syntactic diff [HM75], where the larger the size of the syntactic difference thus the accuracy of the semantic difference produce declines.

Semantic difference has been identified as a key challenge with many opportunities [LVH10] and applications including improving program understanding and debugging, reducing the need for code review, pruning and generating regression tests and automatic generation of exploits.

2.1 Overview

In this section, we provide an informal overview of our approach using a simple illustrating example. In Section 2.6 we show how our approach is applied to real-world programs.

Consider the two versions of a program for computing sign in Figure 2.1, inspired by an example from [RM07]. For these programs, we would like to establish that the output of \( \text{sign} \) and \( \text{sign}' \) differs only in the case where \( x = 0 \) and that the difference is \( \text{sgn} = 1 \neq \text{sgn}' = 0 \).

Separate Analysis is Unsound

As a first naïve attempt to achieve this, one could try to analyze

```c
int sign(int x) {
    int sgn;
    if (x < 0)
        sgn = -1;
    else
        sgn = 1;
    return sgn;
}
```

```c
int sign'(int x') {
    int sgn';
    if (x' < 0)
        sgn' = -1;
    else
        sgn' = 1;
    if (x'==0)
        sgn' = 0;
    return sgn';
}
```

```c
int sign▷sign'(int x) {
    int x' = x;
    guard g1 = (x < 0);
    guard g1' = (x' < 0);
    int sgn;
    int sgn' = sgn;
    if (g1) sgn = -1;
    if (g1') sgn' = -1;
    if (!g1) sgn = 1;
    if (!g1') sgn' = 1;
    guard g2' = (x' == 0);
    if (g2') sgn' = 0;
}
```

Figure 2.1: Two implementations of a sign computing procedure (\( \text{sign} \) & \( \text{sign}' \)) and their correlating program (\( \text{sign}▷\text{sign}' \)).
each version of the program separately and compare the (abstract) results. However, this is clearly unsound, as equivalence under abstraction does not entail concrete equivalence. For example, using an interval domain [CH78] would yield that in both programs the result ranges in the same interval $[-1, 1]$, missing the fact that $\text{sign}$ never returns the value 0 where $\text{sign}'$ does.

Establishing Equivalence under Abstraction To establish equivalence under abstraction, we need to abstract relationships between the values of variables in $\text{sign}$ and $\text{sign}'$. Specifically, we need to track the relationship between the values of $\text{sgn}$ and $\text{sgn}'$. This requires a joint representation in which these relationships can be tracked.

As our approach dictates the joint analysis of two programs for maintaining variable relationships, we need to determine an order in which the different stages of the programs are analyzed. One solution would be to analyze the programs sequentially. However, such an analysis will be forced to retain full path sensitivity, withholding over-approximation, since abstracting together paths will result in a non-restorable loss of equivalence. For example, analyzing $\text{sign}$ first will result in an abstract state where $\sigma = \text{sgn} \mapsto [-1, 1]$. As we continue on towards $\text{sign}'$, we could never restore in $\sigma$ the fact that $\text{sgn}'$ is equal to $\text{sgn}$ for all paths except where $x$ is zero.

Intuitively, establishing equivalence using the sequential composition $P; P'$ requires full path sensitivity, leading to an inherently non-scalable solution. Further, in the presence of loops and widening, applying widening separately to the loops of $P$ and to those of $P'$ does not allow maintaining variable relationships under abstraction.

Correlating Program To address these challenges, we construct a correlating program $P \bowtie P'$ where operations of $P$ and $P'$ are interleaved to achieve correlation throughout the analysis. Figure 2.1 shows the correlating program $\text{sign} \bowtie \text{sign}'$. The programs were transformed to a guarded command language form to allow for interleaving. A key feature of the correlating program for closely related program versions is the ability to keep matched instructions, that appear in both versions, closely interleaved. This allows the analysis to better maintain relationships as the program executions are better aligned. Using the correlating program, we can directly track the relationship between $\text{sgn}$ in $\text{sign}$ and its corresponding variable $\text{sgn}'$ in $\text{sign}'$.

We note that the set of tracked relationships is determined by a matching of $P$ and $P'$ variables denoted $VC$ and defined in Section 2.3. We match variables in the two versions using variable names as we found that these do not vary greatly over patches. However, this matching can also be provided by the user.

We describe the specifics of creating $P \bowtie P'$ in Section 2.5 and only briefly note that the interleaving is chosen according to a syntactic diff process over a guarded command language version of the programs.

Correlating Abstract Domain We introduce a correlating abstract domain that tracks relationships between corresponding variables in $P$ and $P'$ using the correlating program $P \bowtie P'$. Unfortunately, any domain with convex constraints will fail to capture the precise relationship between variables in many cases. For example, using the polyhedra abstract domain [CH78] to analyze the sign example from Figure 2.1, the relationship between the $\text{sgn}$ and $\text{sgn}'$ variables in
the correlating program would be lost, leaving only the trivial \((1 \geq sgn \geq -1, 1 \geq sgn' \geq -1)\) constraint. Although the result soundly reports a difference (as we do not explicitly know that 
\(\equiv_{sgn}\)), we still know nothing about the difference between the programs.

An obvious, but prohibitively expensive, solution to the problem is to use disjunctive completion, moving to a powerset domain where the abstract state is a set of convex objects
(e.g., set of polyhedra). A state in such domain is a set of convex abstract representations (e.g.,
polyhedra [CH78] or octagon [Min06]). For example, analyzing \(sign \bowtie sign'\) using a powerset domain would yield:

\[
\sigma_1 = \{x = x' < 0, sgn = sgn' \mapsto -1\},
\]

\[
\sigma_2 = \{x = x' \mapsto 0, sgn \mapsto 1, sgn' \mapsto 0\}
\]

\[
\sigma_3 = \{x = x' > 0, sgn = sgn' \mapsto 1\}
\]

However, using such domain would significantly limit the applicability of the approach. The desirable solution is a partially disjunctive domain, where only certain disjunctions are kept separate during analysis. The challenge in our setting is in keeping the partition fine enough such that equivalence could be preserved, without reaching exponential blowup. This is accomplished by applying partitioning.

**Partitioning** As the goal of this work is to distinguish equivalent from dissimilar behaviors, using equivalence as criteria for merging paths is apt. The partitioning will abstract together paths that hold equivalence for the same set of variables, allowing for a maximum of \(2^{|VC|}\) disjunctions in the abstract state.

For example partitioning the above-mentioned result of analyzing \(sign \bowtie sign'\) according to our criteria would abstract behaviors \(\sigma_1\) and \(\sigma_3\) together, as they hold equivalence for \(sgn\). The merge would abstract away data regarding \(x\) and represent \(sgn\) as the \([-1, 1]\) interval, losing precision but gaining reduction in state size. This loss of precision is acceptable as it is complemented by the offending state \(\sigma_2\).

\[
\sigma_1 = \{x = x', sgn = sgn' \mapsto [-1, 1]\},
\]

\[
\sigma_2 = \{x' = 0, sgn \mapsto 1, sgn' \mapsto -1\}
\]

To reduce state size, we must perform partitioning dynamically during analysis. This cannot be achieved using a sequential composition \(P; P'\). Intuitively, this is because an operation in \(P\) has to “wait” for its equivalent operation to occur in \(P'\). To overcome this, our correlating program \(P \bowtie P'\) interleaves \(P\) and \(P'\) commands, and informs the analysis when programs have reached a point where correlation may be established by annotating \(P \bowtie P'\) with special markers called correlation points denoted \(CP\) and defined also in Section 2.5.

**Widening** Although we achieved a reduction in state size using partitioning, we have yet to account for programs with loops. Handling loops is where most previous approaches fall short [GS09, KLR10, PDEP08, RE11]. To overcome this, we define a widening operator for our domain, based on the convex sub-domain widening operator (e.g., interval, octagon, polyhedra).
int sum(int arr[], unsigned len) {
    int result = 0;
    for (unsigned i = 1; i < len; i+=2)
        result += arr[i];
    return result;
}

int sum'(int arr[], unsigned len') {
    int result' = 0;
    unsigned i' = 0;
    while (i' + 1 < len') {
        i'++;
        result' += arr'[i'];
        i'++;
    }
    return result';
}

Figure 2.2: Two equivalent versions of a looping program for partial array summation.

The main challenge here, as our state is a set of convex objects belonging to the sub-domain, is finding an optimal pairwise matching between objects for a precise widened result. Ideally, we would like to pair objects that adhere to the same “looping path” meaning we would like to match a path $\pi_i$’s abstraction with a path $\pi_{i+1}$ that results from taking another step in the loop. This requires encoding path information along with the sub-state abstraction. This information is acquired by keeping guard values explicitly, as they appear in our correlating program, inside the state. As guard values (true or false) reflect branch outcomes, they can be used to match sub-states that advanced on the loop by matching their guard values.

We note that the correlating program is crucial to maintaining equivalence over loops. To demonstrate this we perform the simple exercise of checking equivalence of a small looping program with itself. Consider the array summation program in Figure 2.2. Equivalence for these two small programs cannot be established soundly by approaches based on under approximation. To emphasize the importance of the correlating program, we will first show the result of an analysis of $\text{sum;} \text{sum'}$ which will be:

$$\sigma_1 = \{\text{len} = \text{len'} \leq 1, \text{result} = \text{result'} \rightarrow 0\},$$

$$\sigma_2 = \{\text{len} = \text{len'} > 1\}$$

This loss of equivalence occurred due to the inability to precisely track the relationship of result and result' over $\text{sum;} \text{sum'}$. As we widened the first loop to converge, all paths passing through that loop were merged together, losing the ability to be “matched” with the second loop waiting further down the road. Performing the same analysis on $\text{sum} \triangleright \text{sum'}$ instead as seen in Figure 2.3, allows maintaining equivalence, as the loops are interleaved to allow establishing $\equiv_{\text{result}}$ as a loop invariant. This invariant survives the widening process to prove equivalence at the end as the result would be: $\sigma_1 = \{\equiv_{\text{result}}\}$. We note that we implicitly assume equivalence in array content for $\text{sum}$ and $\text{sum'}$.

### 2.2 Preliminaries

We use the following standard concrete semantics definitions for a program:

- $\text{Var, Val, Loc}$ denote the set of program variable identifiers, variable values and program locations respectively. Program locations are also denoted $\text{lab}$ for label. The labels $\text{begin}$...
Figure 2.3: $\text{sum} \triangleright \text{sum}'$

and end mark the start and exit locations of the program.

- A concrete program state $\sigma$ is a tuple $(\text{loc}, \text{values}) \in \Sigma$ mapping the set of program variables to their concrete value at a certain program location $\text{loc}$. The set of all possible states of a program $P$ is denoted $\Sigma_P$.

- We describe an imperative program $P$, as a tuple $(\text{Val}, \text{Var}, \rightarrow, \Sigma_0)$ where $\rightarrow: \Sigma_P \times \Sigma_P$ is a transition relation and $\Sigma_0$ is a set of initial states of the program.

- A program trace $\pi \in \Sigma_P^\ast$, is a sequence of states $\langle \sigma_0, \sigma_1, \ldots \rangle$ describing a single execution of the program. The set of all possible traces for a program is denoted $[P]$. We also define $\text{last} : \Sigma_P^\ast \rightarrow \Sigma_P$ which returns the last state in a trace.

We note that our formal semantics need not deal with errors states therefore we ignore crash states of the programs, as well as inter-procedural programs since our work deals with function calls by either assuming output-equivalence (for functions that were proven to be equivalent) or by inlining them (this work excludes recursion).

2.3 Concrete Semantics

In this section, we define the notion of concrete difference between programs, based on a standard concrete semantics.

2.3.1 Concrete State Differencing

Comparing two programs $P$ and $P'$ under concrete semantics means comparing their traces, but only those that originates from the same input. Towards that end, we first define the difference between two concrete states.

Intuitively, given two concrete states, the difference between them is the set of variables (and their values) where the two states map corresponding variables to different values. As variable names may differ between programs, we parameterize the definition with a mapping that
establishes a correspondence between variables in $P$ and $P'$. Thus concrete state differencing is restricted to comparing values of corresponding variables.

**Definition 2.3.1 (Variable Correspondence).** A variable correspondence $VC \subseteq Var \times Var'$, is a partial mapping between two sets of program variables. The $VC$ mapping can be taken as input from the user however, our evaluation indicates that is sufficient to use a name-based mapping for a program and a patched version:

$$VC_{EQ} \triangleq \{(v, v') | v \in Var \land v' \in Var' \land name(v) = name(v')\}$$

**Definition 2.3.2 (Concrete State Delta).** Given two concrete states $\sigma \in \Sigma_P$, $\sigma' \in \Sigma_{P'}$, and a correspondence $VC$, the concrete state delta is defined as:

$$\triangle S(\sigma, \sigma') \triangleq \{(v, val) | (v, v') \in VC \land \sigma(v) = val \neq \sigma'(v')\}$$

Informally, $\triangle S$ means “the part of the state $\sigma$ where corresponding variables do not agree on values (with respect to $\sigma'$)”. Note that $\triangle S$ is not symmetric. In fact, the direction in which $\triangle S$ is used has meaning in the context of a program $P$ and a patched version of it $P'$. We define $\triangle S^- = \triangle S(\sigma, \sigma')$ which means the values of the state that was removed in $P'$ and $\triangle S^+ = \triangle S(\sigma', \sigma)$ which stands for the values added in $P'$. When there is no observable difference between the states we get that $\triangle S^+(\sigma, \sigma') = \triangle S^-(\sigma, \sigma') = \emptyset$, and say that the states are equivalent denoted $\sigma \equiv \sigma'$.

Consider two concrete states $\sigma = (x \mapsto 1, y \mapsto 2, z \mapsto 3)$ and $\sigma' = (x' \mapsto 0, y' \mapsto 2, w' \mapsto 4)$ and using $VC_{EQ}$ then $\triangle S^- = \{(x \mapsto 1)\}$ since $x$ and $x'$ match and do not agree on value, $y$ and $y'$ agree (thus are not in delta) and $z'$ is not in $VC_{EQ}$. Similarly, $\triangle S^+ = \{(x' \mapsto 0)\}$.

We now use our notion of concrete state difference to define the difference between concrete program traces.

**Definition 2.3.3 (Trace Delta).** Given two traces $\pi \in \llbracket P \rrbracket$ and $\pi' \in \llbracket P' \rrbracket$ that originate from equivalent input states, we define the trace delta as simply the difference between the traces final states. Formally:

$$\triangle T(\pi, \pi') = \{\triangle S(\text{last}(\sigma), \text{last}(\sigma'))\}$$

The definition adheres to our problem definition in Section 1.1, where we defined program difference as difference between matched variables in the terminating state. Since $\triangle T(\pi, \pi')$ is based on state difference, we define $\triangle T^+$ and $\triangle T^-$ similarly to their underlying states difference operations.

Now, we will move past the concrete semantics towards abstract semantics. This is required as it is unfeasible to describe difference based on traces. Before doing so, we must adjust our concrete semantics since a concrete semantics based on individual traces will not allow us to correlate traces that originate from the same input. This is the first formal indication of how a separate abstraction, that considers each of the programs by itself, cannot succeed.
2.3.2 Concrete Correlating Semantics

We define the correlating state and trace which bind the executions of both programs, \( P \) and \( P' \), together and define the notion of delta in this setting. This allows us to define the correlating abstract semantics which is key for successful differencing.

**Definition 2.3.4** (Correlating Concrete State). A correlating concrete state \( \sigma_{\triangleleft} : \text{Var} \cup \text{Var}' \rightarrow \text{Val} \) is a unified concrete state, mapping variables from both programs \((P, P')\) to their values.

**Definition 2.3.5** (Correlating Concrete Trace). A correlating trace \( \pi_{\triangleleft} \), is a sequence of correlating states \( \ldots, \sigma_{\triangleleft i}, \ldots \) describing an execution of \( P \triangleright P' \).

Note that an attribute of the correlating programs (as defined in Section 2.5) is that it restricts to traces that originate from equivalent input states i.e., \( \sigma_{\triangleleft 0} \equiv \sigma'_{\triangleleft 0} \).

We must remember however, that the number of traces to be compared is potentially unbounded which means that the delta we compute may be unbounded too. Therefore we must use an abstraction over the concrete semantics that will allow us to represent executions in a bounded way.

2.4 Abstract Correlating Semantics

In this section, we introduce our correlating abstract domain which allows bounded representation of correlating program state while maintaining equivalence between correlated variables.

2.4.1 Abstract Correlating State

We represent variable information using standard relational abstract domains. As our analysis is path sensitive, we allow for a set of abstract sub-states, each adhering to a certain path in the product program. This abstraction is similar to the trace partitioning domain as described in [RM07].

Our power-set domain records precise state information but does not scale due to exponential blowup. As a first means of reducing state size, we define a special join operation that dynamically partitions the abstract state according to the set of equivalences maintained in each sub-state and joins all sub-states in the same partition together (using the sub-domain join operation). This join criteria allows separation of equivalence preserving paths thus achieving better precision. Second, to allow a feasible bound abstraction for programs with infinite number of paths, we define a widening operator which utilizes the sub-domain’s widening operator but cleverly chooses which sub-states are to be widened, according to path information encoded in state. We start off by abstracting the correlating trace semantics in Section 2.3.2.

In the following, we assume an abstract relational domain \((D^\#, \sqsubseteq_D)\) equipped with operations \( \sqcap_D, \sqcup_D \) and \( \triangledown_D \), for representing sets of concrete states in \( \Sigma_{P \triangleright P'} \). We separate the set of program variables into original program variables denoted \( \text{Var} \) (which also include a special added variable for return value, if such exists) and the added guard variables denoted \( \text{Guard} \) that are used for storing conditional values alone (\( \text{Guard} \) also include a special added guard for return flag). We assume the abstract values in \( D^\# \) are constraints over the variables and guards (we denote \( D^\#_{\text{Guard}} \) for sub-domain abstraction of guards and \( D^\#_{\text{Var}} \) for original variables), and
\[ [v := e]^\sharp \quad l_{\infty} \mapsto \{ (ctx, [v := e]^\sharp_{DF}(data))(ctx, data) \in S \} \]

\[ [g := e]^\sharp \quad l_{\infty} \mapsto \{ ([g := true]^\sharp_{DF}(ctx), [e]^\sharp_{DF}(data))(ctx, data) \in S \}
\cup \{ ([g := false]^\sharp_{DF}(ctx), [-e]^\sharp_{DF}(data))(ctx, data) \in S \} \]

\[ [\text{if}(g) \{ s_0 \} \text{else} \{ s_1 \}]^\sharp \quad l_{\infty} \mapsto \{ ([g = true]^\sharp_{DF}(ctx), [s_0]^\sharp_{DF}(data))(ctx, data) \in S \}
\cup \{ ([g = false]^\sharp_{DF}(ctx), [s_1]^\sharp_{DF}(data))(ctx, data) \in S \} \]

\[ [\text{goto} \ \text{lab}]^\sharp \quad \sigma^\sharp \]

---

Table 2.1: Abstract transformers

---

Do not go into further details regarding the particular abstract domain as it is a parameter of the analysis. We also assume that the sub-domain \( D^\sharp \) allows for a sound over-approximation of the concrete semantics (given a sound interpretation of program operations). In our experiments, we use the polyhedra abstract domain [CH78] and the octagon abstract domain [Min06].

**Definition 2.4.1** (Correlating Abstract State). A correlating abstract program state \( \sigma^\sharp \in \text{Lab} \rightarrow 2^{D^\sharp_{\text{Guard}} \times D^\sharp_{\text{Var}}} \), is a mapping from a correlating program label \( l_{\infty} \) to a set of pairs \( (ctx, data) \), where \( ctx \in D^\sharp_{\text{Guard}} \) is the execution context i.e. an abstraction of guards values via the relational numerical domain and \( data \in D^\sharp_{\text{Var}} \) is an abstraction of the variables.

We separate abstractions over guard variables added by the transformation to Guarded command language (GCL) format (see Section 2.5) from original program variables as there need not be any relationships between guard and regular variables.

### 2.4.2 Abstract Correlating Semantics

Table 2.1 describes the abstract transformers. The table shows the effect of each statement on a given abstract state \( \sigma^\sharp = l_{\infty} \mapsto S \). The abstract transformers are defined using the abstract transformers of the underlying abstract domain \( D^\sharp \). We assume that any program \( P \) can be transformed such that it only contains the operations described in Table 2.1 (this is achieved by the GCL format). We also assume that for \([g := e]^\sharp\) operations, \( e \) is a logical operation with boolean value.

Next, we define the abstraction function \( \alpha : 2^{\Sigma_{P \rightarrow P'}} \rightarrow 2^{D^\sharp \times D^\sharp} \) that abstracts together a set of concrete correlating traces \( T \). As in our domain traces are abstracted together if they share the exact same path, we first define an operation \( \text{path} : \Sigma_{P \rightarrow P'} \rightarrow \text{Lab}^* \) which returns a sequence of labels for a trace’s states i.e. what is the path taken by that trace. We also allow applying \( \text{path} \) on a set of traces to denote the set of paths resulting by applying the function of each of the traces. Finally we define the trace abstraction as follows:

\[
\alpha(T) \triangleq \{ \cup_{\text{path}(\pi) = p} \beta(\text{last}(\pi)) | p \in \text{path}(T) \}
\]

where \( \beta(\sigma) = (\beta_{DF}(\sigma|\text{Guard}), \beta_{DF}(\sigma|\text{Var})) \) i.e. applying the abstraction function of the abstract sub-domain \( \beta_{DF} \) on parts of the concrete state applying to \( \text{Guard} \) (denoted \( \sigma|\text{Guard} \)) and \( \text{Vars} \)
(denoted $\sigma_{|V_{or}}$) separately. Our abstraction partitions trace prefixes $\pi$ by path and abstracts together the concrete states reached by the prefix - $last(\pi)$, using the sub-domain.

Every path in the correlating program will be represented by a single sub-state of the sub-domain. As a result, all trace prefixes that follow the same path to $l_{\infty}$ will be abstracted into a single sub-state of the underlying domain. This abstraction fits semantics differencing well, as inputs that follow the same path display the same behavior and will usually either keep or break equivalence together, allowing us to separate them from other behaviors (it is possible for a path to display both behaviors as in Figure 2.4 and we will discuss how we are able to manipulate the abstract state and separate equivalent behaviors from ones that offend equivalence). Another issue to be addressed is the fact that our state is still potentially unbounded as the number of paths in the program may be exponential and even infinite (due to loops).

### 2.4.3 Dynamic Partitioning

Performing analysis with the powerset domain does not scale as the number of paths in the correlated program may be exponential (we defer the case of unbound paths to widening of loops). We must allow for reduction of state $\sigma^t = l_{\infty} \mapsto S$ with acceptable loss of precision. This reduction via partitioning can be achieved by joining the abstract sub-states in $S$ (using the standard join of the sub-domain). However this can only be accomplished after first deciding which of the sub-states should be joined and then choosing the program locations for the partitioning to occur. To choose a strategy, we start by taking a closer look at the final state of the fully disjunctive analysis of Figure 2.1:

$$\sigma^t(end) = \{(g_1, \neg g_2', \equiv g_1), (x > 0, sgn = 1, \equiv x, sgn)\},$$

$$\{(\neg g_1, \neg g_2', \equiv g_1), (x < 0, sgn = -1, \equiv x, sgn)\},$$

$$\{(\neg g_1, g_2', \equiv g_1), (x = 0, sgn = 0, sgn' = 1, \equiv x)\}\]

One may observe that were we to join the two sub-states that maintain equivalence on $\{x, sgn, g_1\}$, it would result in an acceptable loss of precision (losing the $x$ related constraints). This is achieved by partitioning sub-states according to the set of variables which they preserve equivalence for. This bounds the state size at $2^{|V_{C}|}$, where $V_{C}$ is the set of correlating variables we wish to track. As mentioned, another key factor in preserving equivalence and maintaining precision is the program location at which the partitioning occurs. The first possibility, which is somewhat symmetric to the first proposed partitioning strategy, is to partition at every join point i.e. after every branch converges. Let use examine $\text{sgn} \gg \text{sgn}'$ state after processing the first guarded instruction $\text{if } (g_1) \text{ sgn} = -1$; (we ignored $g2'$ effect at this point for brevity):
int sum(int arr[],
unsigned len) {
    int result = 0;
    for (unsigned i = 1;
         i < len; i += 2)
        result += arr[i];
    return result;
}

unsigned max'' = ...;
int sum''(int arr'', unsigned len'') {
    unsigned len'' = len;
    int arr''[i] = arr;
    int result'' = 0;
    guard r'' = (len'' > max'');
    if (r'') retval'' = -1;
    if (r'') r'' = 0;
    {
        unsigned i = 1;
        unsigned i'' = 1;
        l: guard g = (i < len);
        l'': guard g'' = 0;
        if (r'') g'' = (i'' < len'');
        if (g) result'' += arr''[i];
        if (r'') if (g'') result'' += arr''[i''];
        if (g) i+=2;
        if (r'') if (g'') i''+=2;
        if (g) goto l;
        if (r'') if (g'') goto l'';
    }
}

Figure 2.5: Original sum, Patched sum'' and correlating sum ⊳ sum''

\[\sigma^\sharp = \left[\langle (g_1, \equiv g_1), (x \geq 0, \equiv x, \text{sgn}) \rangle, \langle (g_1, \equiv g_1), (x < 0, \text{sgn}' = -1, \equiv x) \rangle \right]\]

This suggests that partitioning at join points will perform badly in many scenarios, specifically here as we will lose all data regarding sgn. However if we could delay the partitioning to a point where the two programs converge (after the following if \((g_1') \text{ sgn}' = -1;\) line), we will get a more precise temporary result which preserves equivalence. To accomplish this, we define special program locations we name correlating points which present places where programs have likely converged. These are a sub-product of the correlating program construction process described in Section 2.5.

### 2.4.4 Widening

In order for our analysis to handle loops we require a means for reaching a fixed point. As our analysis iterates over a loop, sub-states may be added or transformed continuously, never converging. We therefore need to define a widening operator for our new domain. We have the widening operator of our sub-domain at our disposal, but we are faced with the question of how to lift this operator, i.e., which pairs of sub-states \(\langle \text{ctx}, \text{data} \rangle\) from \(\sigma^\sharp\) should be widened with which. This problem has been addressed in the path in other settings \([BH2006]\), and our approach can be viewed as a specialized form of lifting that is tailored for tracking equivalences. A first viable strategy is to perform an overall join operation on all pairs which will result in a single pair of sub-states and then simply apply the widening to this sub-state using the sub-domain’s \(\triangledown\) operator. If we examine applying this strategy to \(\text{sum} \triangledown \text{sum}'\) from Figure 2.3, we get that it will successfully arrive at a fixed point that also maintains equivalence as all sub-states maintain equivalence at loop back-edges. Now let us try to apply the strategy to the
more complex $\sum \bowtie \sum'$ of Figure 2.5. First we mention that as $\sum'$ introduces a return statement under the $\text{len} > \text{max}$ condition, the example shows an extra $r'$ guard and $\text{retval}'$ variable for representing a return (this exists in all GCL programs but we omitted it so far for brevity). While analyzing, once we pass that first conditional, our state is split to reflect the return effect:

$$\sigma^2 = \{d_1 = \langle \neg r', (\text{len} \leq \text{max}, \text{result} = 0, \equiv_{\text{len}, \text{result}})\rangle, \quad d_2 = \langle r', (\text{len} > \text{max}, \text{retval}' = -1, \text{result} = 0, \equiv_{\text{len}, \text{result}})\rangle\}$$

As we further advance into the loop, $d_1$ will maintain equivalence but $d_2$ will continue to update the part of the state regarding untagged variables (since $r'$ is $\text{false}$), specifically it will change $\text{result}$ continuously, preventing the analysis from reaching fixed point. We would require widening here but using the naive strategy of a complete join will result in aggressive loss of precision, specifically losing all information regarding $\text{result}$. The problem originates from the fact that prior to widening, we joined sub-states which adhere to two different loop behaviors: one where both $\sum$ and $\sum'$ loop together (that originated from $\text{len} < \text{max}$) and the other where $\sum'$ has exited but $\sum$ continues to loop ($\text{len} \geq \text{max}$). Ideally, we would like to match these two behaviors and widen them accordingly. We devised a widening strategy that allows us to do this as it basically matches sub-states that adhere to the same behavior, or loop-paths. This strategy dictates using guards for the matching. If two sub-states agree on their set of guards, it means they represent the same loop path and can be widened as the latter originated from the former (widening operates on subsequent iterations). In our example, using this strategy will allow the correct matching of states after consequent $k, k + 1$ loop iterations:

$$\sigma^k = \{d_1 = \langle \neg r', g, \equiv_g, (\text{len} \leq \text{max}, i = 2k + 1, \equiv_i, \text{len}, \text{result})\rangle, \quad d_2 = \langle r', \neg g, g', (\text{len} > \text{max}, \text{retval}' = -1, \text{result} = 0, i' = 2k + 1, i = 1, \equiv_{\text{len}})\rangle\}$$

And:

$$\sigma^{k+1} = \{d_1 = \langle \neg r', g, \equiv_g, (\text{len} \leq \text{max}, i = 2k + 3, \equiv_i, \text{len}, \text{result})\rangle, \quad d_2 = \langle r', \neg g, g', (\text{len} > \text{max}, \text{retval}' = -1, \text{result} = 0, i' = 2k + 3, i = 1, \equiv_{\text{len}})\rangle\}$$

As we can identify the states predecessors by simply matching the guards. $d_1$ will be widened for a precise description of the difference shown as $\langle \text{len} = \text{len}' > \text{max}', \text{retval}' = -1, \text{retval} = \top \rangle$.

### 2.4.5 Differencing for Abstract Correlating States

Given an abstract state in our correlating domain, we want to determine whether equivalence is kept and if so under which conditions it is kept (for partial equivalence) or determine there is difference and characterize it. As our state may hold several pairs of sub-states, each holding different equivalence data, we can provide a verbose answer regarding whether equivalence holds. We partition our sub-states according to the set of variables they hold equivalence for and report the state for each equivalence partition class. Since we instrument our correlating...
program to preserve initial input values, for some of these states we will also be able to report input constraints thus informing the user of the input ranges that maintain equivalence. When equivalence could not be proved, we report the offending states and apply a differencing algorithm for extracting of the delta. Figure 2.4 shows an example of where our analysis is unable to prove equivalence, although part of the state does maintain equivalence (specifically for $x = 0$). This is due to the abstraction being too coarse. We describe an algorithm that given a sub-state $d \in D^i$, computes the differentiating part of the sub-state (where correlated variables disagree on values) by splitting it into parts according to equivalence. This is done by treating the relational constraints in our domain as geometrical objects and formulating delta based on that.

**Definition 2.4.2 (Correlating Abstract State Delta).** Given a sub-state $d$ and a correspondence $VC$, the correlating state delta $\triangle_A(d)$, computes abstract state differentiation over $d$. The result is an abstract state $\sqsubseteq d$ approximating all concrete values for variables correlated by $VC$, that differ between $P$ and $P'$. Formally, the delta is simply the abstraction of the concrete trace deltas:

$$\triangle_A(d)^+ \triangleq \alpha(\cup_{\text{path}} \triangle_T^+), \triangle_A(d)^- \triangleq \alpha(\cup_{\text{path}} \triangle_T^-)$$

where deltas are grouped together by path and then abstracted.

The algorithm for the extraction of delta from a correlating state, is as follows:

1. $d_\equiv$ is a state abstracting the concrete states shared by the original and patched program. Obtained by computing: $d_\equiv \triangleq d|_{V=V'} \equiv d \cap \bigwedge \{v = v'|(v,v') \in VC\}$.  

2. $\overline{d_\equiv}$ is the negated state i.e. $D^i \setminus d_\equiv$ and it is computed by negating $d_\equiv$ (as mentioned before, all logical operations, including negation, are defined on our representation of an abstract state).

3. Eventually: $\triangle_A(d) \triangleq d \cap \overline{d_\equiv}$ abstracts all states in $P \times P'$ where correlated variables values do not match.

4. $\triangle_A(d)^+ = \triangle_A(d)|_{V'}$ is a projection of the differentiation to display values of $P'$ alone i.e. added values.

5. $\triangle_A(d)^- = \triangle_A(d)|_V$ is a projection of the differentiation to display values of $P$ alone i.e. removed values.

Applying the algorithm on Figure 2.4’s $P$ and $P'$ where $d = \{\text{retval}' = 2\text{retval}\}$ will result in the following:

1. $d_\equiv = \langle\text{retval}' = 0, \text{retval} = 0\rangle$.

2. $\overline{d_\equiv} = \{\langle\text{retval}' > 0\rangle, \langle\text{retval}' < 0\rangle, \langle\text{retval} > 0\rangle, \langle\text{retval} < 0\rangle\}$

3. $\triangle_A(d) = \{\langle\text{retval}' = 2\text{retval}, \text{retval}' > 0\rangle, \langle\text{retval}' = 2\text{retval}, \text{retval}' < 0\rangle, \langle\text{retval}' = 2\text{retval}, \text{retval} > 0\rangle, \langle\text{retval}' = 2\text{retval}, \text{retval} < 0\rangle\}$
4. $\triangle_A(d)^+ = [(retval' > 0), (retval' < 0)]$

5. $\triangle_A(d)^- = [(retval > 0), (retval < 0)]$

We note that as a sub-state is basically a conjunction of constraints, negating it by splitting to constraints and negating each individually reflects correctly the effect of negating a conjunction as we are left with a disjunction of negations, as seen in step 2. We also see that displaying the result in the form of projections is ill-advised as in some states differentiation data is represented by relationships on correlated variables alone, thus projecting will lose all data and we will be left with a less informative result. Figure 2.6 depicts a geometrical representation of $\triangle_A$ calculation.

We note that as a sub-state is basically a conjunction of constraints, negating it by splitting to constraints and negating each individually reflects correctly the effect of negating a conjunction as we are left with a disjunction of negations, as seen in step 2. We also see that displaying the result in the form of projections is ill-advised as in some states differentiation data is represented by relationships on correlated variables alone, thus projecting will lose all data and we will be left with a less informative result. Figure 2.6 depicts a geometrical representation of $\triangle_A$ calculation.

From this point forward any mention of delta (denoted $\triangle$) refers to the correlating abstract state delta ($\triangle_A$). We claim that $\Delta$ is a correct abstraction for the concrete state delta which allows for a scalable representation of difference we aim to capture.

2.5 Correlating Program

In this section, we describe how to construct a correlating program $P \bowtie P'$. The process attempts to find an interleaving of programs for a more precise differentiation. The construction also instruments $P \bowtie P'$ with the required correlation points $CP$ which define the locations for our partitioning. We also allow a user defined selection of $CP$. 

Figure 2.6: $\Delta$ computation geometrical representation.
2.5.1 Construction of $P \bowtie P'$

The idea of a correlating program is similar to that of self-composition [TA05], but the way in which statements in the correlating program are combined is designed to keep the steps of the two programs close to each other. Analysis of the correlating program can then recover equivalence between values of correlated variables even when equivalence is temporarily violated by an update in one version, as the corresponding update in the other version follows shortly thereafter.

The correlating program is an optimized reduction over $P \times P'$ where not all pairs of $(\sigma^p, \sigma'^p)$ are considered, but only pairs in a controlled execution, where correlating instructions in $P$ and $P'$ execute adjacently. This allows for superior precision.

The input for the correlation process are two C programs $(P, P')$. The first step involves transforming both programs to a normalized guarded instruction form $(P_G, P'_G)$. Next, a vector of imperative commands $I$ (and $I'$ respectively) is extracted from each program for the purposes of performing the syntactic diff. An imperative command in our GCL format is defined to be either one of $v := e \mid \text{goto } l \mid f(...)$, as they effectively change the program state (variable values, excluding guards) and control. Function calls are either inlined, in case equivalence could not be proven for them, or left as is, in case they are equivalent or are external system calls.

Continuing the construction process, a syntactical diff [HM75] is computed over the vectors $(I, I')$. One of the inputs to the diff process is $VC$ as it is needed to identify correlated variables and the diff comparison will regard commands differing by variable names which are correlated by $VC$ as equal. The result of the last step will be a vector $I_\triangle$ specifying for each command in $I, I'$ whether it is an added command in $P'$ (for $I'$) marked $+$, a deleted command from $P$ (for $I$) marked $-$, or a command existing in both versions marked $\equiv$. This diff determines the order in which the commands will be interleaved in the resulting $P \bowtie P'$ as we will iterate over the result vector $I_\triangle$ and use it to construct the correlating program. We remind that since $I, I'$ contain only the imperative commands, we cannot use it directly as $P \bowtie P'$. Instead we will use the imperative commands as markers, specifying which chunk of program from $P_G$ or $P'_G$ should be taken next and put in the result. The construction goes as follows: iterate over $I_\triangle$ and for every command $c (c')$ labeled $l_c (l'_c)$:

- read $P_G (P'_G)$ up to label $l_c (l'_c)$ including into block $B_c (B'_c)$

- for $B'_c$, tag all variables in the block.

- emit the block to the output.

- delete $B_c (B'_c)$ from $P_G (P'_G)$.

The construction is now complete. We only add that at the start of the process, we strip $P'_G$ of its prototype and add declarations for the tagged input variables, initializing them to the untagged version (thus assuring $P \bowtie P'$ will only co-execute traces that originate from the same input for $P$ and $P'$). As mentioned, $CP$ is also a product of the construction, and it’s defined using $=$ commands: after two $=$ commands are emitted to the output, we add an instrumentation line, telling the analysis of the correlation point. One final observation regarding the correlating
program is that it is a legitimate program that can be run to achieve the effect of running both versions. We plan to leverage this ability to use dynamic analysis and testing techniques such as fuzzing [NS] and directed automated testing [CDE08] on the correlating program in our future work.

It is important to note that the correlating program construction does not support breaking the control flow with break, continue, and goto statements. [GMR15] presented an updated algorithm for the creation of correlating programs, which allows the use of break and continue in loops.

2.6 Evaluation

We evaluated DIZY on a number of real world programs where the patches affect numerical variables. As benchmarks, we used several programs from the GNU core utilities, as well as a few handpicked patches from the Linux kernel and the Mozilla Firefox web browser. We also include results for illustrative examples used throughout the chapter.

2.6.1 Prototype Implementation

We implemented a correlating compiler named CCC which creates correlating programs from any two C programs. We also implemented a differencing analysis for analyzing correlated programs. Both tools are based on LLVM and CLANG compiler infrastructure. We analyze C code directly since it is more structured, has type information and keeps a low number of variables, as opposed to intermediate representation. We also benefit from our delta being computed over original variables. As mentioned in Section 2.5, we normalize the input programs before correlating them. This also allows for a simpler analysis. Our analysis is intra-procedural and we handle function calls by either modularly proving their equivalence and assuming it once encountered or, in case equivalence could not be proved, by inlining. Calls to external system functions do not change local state in our examples and thus were ignored. We used the APRON abstract numerical domain library and conducted our experiments using several domains including Interval, Octagon [Min06] and Polyhedra [CH78]. All of our experiments were conducted running on a Intel(R) Core-i7(TM) processor with 4GB.

2.6.2 Results

Table 2.2 summarizes the results of our analysis. The columns indicate the benchmark name, lines of code for the analyzed program, the number of lines added and removed by the patch, whether it required widening, and the result of each benchmark run alongside its run time in minutes. We included three different setting in the results: with and without partitioning and with an Interval, Octagon [Min06] and Polyhedra [CH78] abstract domains. Generally, the results are ordered in increasing order of precision from left to right. Results marked with ✓ presented abstract states with acceptable precision i.e., mostly variables that indeed differ between variables were reported, and the description of the difference was useful for producing actual values for the differencing variables. As precision increases, the resulting delta was more precise and contained more numerical information describing the difference. Results marked with ✗ produced false positives, reporting equivalent variables as different or providing too
<table>
<thead>
<tr>
<th>Name</th>
<th>#LOC</th>
<th>#P</th>
<th>Widen</th>
<th>Interval Domain</th>
<th>Octagon Domain</th>
<th>Polyhedra Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>With Partitioning</td>
<td>No Partitioning</td>
<td>With Partitioning</td>
</tr>
<tr>
<td>remove</td>
<td>16</td>
<td>4</td>
<td>N</td>
<td>X(0)</td>
<td>X(0)</td>
<td>✓</td>
</tr>
<tr>
<td>copy</td>
<td>44</td>
<td>2</td>
<td>N</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>fmt</td>
<td>42</td>
<td>5</td>
<td>Y</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>md5sum</td>
<td>40</td>
<td>3</td>
<td>Y</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>pr</td>
<td>100</td>
<td>10</td>
<td>N</td>
<td>TO</td>
<td>TO</td>
<td>✓</td>
</tr>
<tr>
<td>saved</td>
<td>86</td>
<td>1</td>
<td>N</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>seq</td>
<td>23</td>
<td>15</td>
<td>Y</td>
<td>X(0.25)</td>
<td>X(2.04)</td>
<td>X(12.21)</td>
</tr>
<tr>
<td>addr</td>
<td>77</td>
<td>1</td>
<td>N</td>
<td>X(0.14)</td>
<td>X(0.46)</td>
<td>✓</td>
</tr>
<tr>
<td>nsGDDN</td>
<td>47</td>
<td>11</td>
<td>N</td>
<td>X(0.02)</td>
<td>X(0.21)</td>
<td>X(0.24)</td>
</tr>
<tr>
<td>sign</td>
<td>8</td>
<td>2</td>
<td>N</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sum</td>
<td>7</td>
<td>5</td>
<td>Y</td>
<td>X(0.03)</td>
<td>X(0.10)</td>
<td>X(1.12)</td>
</tr>
<tr>
<td>nested</td>
<td>10</td>
<td>1</td>
<td>Y</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

abstract of a description of the difference (i.e., ⊤). Results marked in TO represent runs that were stopped after 20 minutes. In either case, the results maintained soundness (equivalence was never reported falsely).

Runs without partitioning presented the most precise results with the most detailed abstract states describing the differencing paths. However this setting could not be applied towards all benchmarks since it leads to state explosion as shown by larger benchmarks that timed out. Applying partitioning allowed us to scale the analysis while maintaining precision. Results from runs that included partitioning described difference with less detail since some numerical data was abstracted away.

As expected, the Interval domain usually produced the fastest, least accurate results, while maintaining soundness as difference was reported for the appropriate variables but numerical data was almost completely abstracted away. In some case, like in the copy benchmark, Interval performed worse than Octagon and Polyhedra (in run time) for runs with partitioning. This is due to the Interval domain’s limited ability to capture variable relationships which led to the partitioning algorithm failing in grouping together the different sub-states (as the equivalences they kept varied greatly). This resulted in a close to $2^{\lceil VC \rceil}$ number of equivalence groups.

Surprisingly, runs using the Octagon domain presented poor performance (run time), even compared to the more expensive Polyhedra domain, with less precision. This is due to the Octagon domain being less successful in capturing equivalences as it is built upon linear inequalities. This meant that more constraints were needed to represent variable equality, resulting in bigger states and a slower analysis.

The addr and nsGDDN benchmarks taken from the net/sunrpc/addr.c module in the Linux kernel SUNRPC implementation v2.6.32-rc6 and Firefox 3.6 security advisory CVE-2010-1196 (adapted to C from C++) respectively. The results produced by DIZY can be directly used towards exploiting known security flaws mentioned in advisories from which these patches originate, as the resulting abstract state describes the difference between versions which is
nsresult SetTextInternal (int textLength, int aCount, int aLength, int aOffset, PRUnichar * aBuffer) {
    PRInt32 newLength = textLength - aCount + aLength;
    PRUnichar * to;
    ...
    if ((unsigned)newLength > (1 << 29))
        return NS_ERROR_DOM_DOMSTRING_SIZE_ERR;
    memcpy (to + aOffset, aBuffer, aLength * sizeof (PRUnichar));
    ...
}

Figure 2.7: Firefox's nsGenericDOMDataNode module with patch.

\[
\begin{array}{lll}
\sigma_1: & \text{returned}' = \text{true} \\
         & \text{returned} = \text{false} \\
         & \text{return value}' = \text{NS_ERROR} \\
\end{array}
\]
\[
\begin{array}{ll}
\sigma_1: & \text{returned}' = \text{true} \\
         & \text{returned} = \text{false} \\
         & \text{newLength} > 536870912 \\
         & \text{return value}' = \text{NS_ERROR} \\
\end{array}
\]
\[
\begin{array}{ll}
\sigma_2: & \text{returned}' = \text{true} \\
         & \text{returned} = \text{false} \\
         & \text{newLength} > -3758096384 \\
         & \text{return value}' = \text{NS_ERROR} \\
\end{array}
\]

(a) (b)

Figure 2.8: Difference for Firefox's nsGenericDOMDataNode, (a) with a single polyhedra; (b) set of polyhedra.

exactly the range of exploitable values.

Non convex delta

Another example, taken from CVE-2010-1196 advisory regarding Firefox's heap buffer overflow on 64-bit systems is shown in Figure 2.7 (vulnerable part of the function only). Firefox 3.5 and 3.6 (up to 3.6.4) contain a heap buffer overflow vulnerability which is caused by an integer overflow. Due to the amount of data needed to trigger the vulnerability (over 8GB), this is only exploitable on 64-bit systems. The vulnerable code is found in /content/base/src/nsGenericDOMDataNode.cpp of the Mozilla code base and was adapted to C for analysis purposes.

Here, we need to describe a non-convex constraint that leads to difference. Running DIZY with partitioning produces the result shown in Figure 2.8 (a) (only the part of the state which breaks equivalence is shown).

The difference in state is described correctly as indeed the only change in values for the patch scenario would be the return value and the early return of the patched version. However, we did not preserve the conditional constraints as they are non-convex. Running the same analysis with no partitioning (this is feasible as the procedure does not loop) produces the result shown in Figure 2.8 (b).

Now we see that the (unsigned)newLength > (1 << 29) constraint has been successfully encoded in two offending states, each holding a part of the problematic range.

Capturing complex delta

Figure 2.9 shows a patch made to the char_to_clump function in version 6.11 of coreutils.
int input_position;

bool char_to_clump(char c) {
    int width;
    ...
    + if (width < 0 && input_position == 0) {
        + chars = 0;
        + input_position = 0;
        + } else if (width < 0 && input_position <= -width) {
            + input_position = 0;
        + } else {
            input_position += width;
        + }
        (∗);
    ...
    return chars;
}

Figure 2.9: Original and patched version of Coreutils pr.c’s char_to_clump procedure

<table>
<thead>
<tr>
<th>σ1:</th>
<th>σ2:</th>
<th>σ3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_position0 = 0</td>
<td>input_position0 &lt; -width</td>
<td>input_position0 &lt; -width</td>
</tr>
<tr>
<td>chars’ = 0</td>
<td>input_position0 &lt; 0</td>
<td>input_position0 &gt; 0</td>
</tr>
<tr>
<td>input_position = width</td>
<td>input_position’ = 0</td>
<td>input_position’ = 0</td>
</tr>
<tr>
<td>input_position &lt; 0</td>
<td>input_position &lt; width</td>
<td>input_position &gt; width</td>
</tr>
<tr>
<td>input_position’ = 0</td>
<td>width ≥ 0</td>
<td>input_position ≤ 0</td>
</tr>
</tbody>
</table>

Figure 2.10: Difference for the char_to_clump procedure from Coreutils’s pr.c

The patch replacing the execution of the line input_position += width, which originally executed unconditionally, with a conditional structure that in the new version, allows the line to execute only under certain complex conditions. Since the variables handled in this patch (the global input_position and return value chars) emit output, describing how their values changed and under which terms is important, especially as the patch cannot be easily parsed by a programmer to understand its meaning. The result of our analysis at the return point is shown in Figure 2.10. These results include information regarding initial values of parameters for improved precision (this is one of DIZY’s features).

The result convey the difference in the output variable values alongside some of conditions under which the difference occurs. The result is composed of three sub-states featuring difference and adhere to two added paths in the patched program. The first sub-state belongs to the first branch in the added conditional: the difference is comprised of (i) the new value of input_position is 0 as opposed to it being width in the former version (the analysis took the input_position += width line into account and incorporated knowing that input_position = 0 from the branch condition). The analysis also deduced that the old input_position is negative under the same input as the branch condition dictates that width is negative. (ii) chars in the new program is 0 under this path. The two other sub-states adhere to the second added path in the conditional and track a difference for input_position alone, basically stating that input_position under this path used to assume values in ranges \([-\infty, width]\) and \([width, 0]\).
int logicalValue(int t) {
    if (!(curr - t >= 100)) {
        return old;
    } else {
        int val = 0;
        for (int i = 0 ; i < data.length; i++)
            val = val + data[i];
        old = val;
        return val;
    }
}

const int THRESHOLD = 100;

int logicalValue(int t) {
    int elapsed = curr - t;
    int val = 0;
    if (elapsed < THRESHOLD) {
        val = 1;
    } else {
        for (int i = 0 ; i < data.length; i++)
            val = val + data[i];
        old = val;
    }
    return val;
}

Figure 2.11: Two versions of the logicalValue procedure taken from [PDEP08].

\[ \sigma_1: \]
\[
\begin{align*}
    \text{curr - t} &< 100 \\
    \text{return value} & = \text{old} \\
    \text{return value'} & = 1
\end{align*}
\]

Figure 2.12: Difference for logicalValue

but now is simply 0. The splitting of this path into two cases is a result of expressing the non-convex input.position \ne 0 condition from the first branch conditional using two sub-states. The result also describes constraints on the procedure’s input under which the difference exist.

Another product of the analysis, which we do not show here, are sub-states describing paths which the patch did not affect.

**Maintaining Equivalence and Reporting Difference in Loops**

Figure 2.11 shows two versions of the java logicalValue() method taken from [PDEP08], adapted to C. This example features semantic preserving refactoring modification (introducing the elapsed variable and THRESHOLD constant, simplifying a conditional and moving the return statement out of branch block) and one semantic change where 1 is returned instead of old in case curr − t < 100). The challenge in this example is proving equivalence over the loop branch and reporting difference for the negated path. Using a separate analysis, we would have to deduce at the following loop invariant: \( \sum_{i=0}^{data.length} data[i] \) in order to show equivalence. However, as our abstraction focuses on variable relationships and our correlating program allows us to interleave the two loops in lock-step, all our analysis needs to deduce is the \( val = val' \) constraint. As we apply widening to converge, the constraint will be kept, allowing us to establish equivalence for the looping path. DIZY reports the state shown in Figure 2.12 state for the exit point of logicalValue(). We note that in [PDEP08] the example was run by unrolling 2 steps of the loop.

In the md5sum benchmark, all paths in the programs contain loops and only some of them maintain equivalence. Figure 2.13 shows part of the benchmark that was patched to disallow 0-length inputs (patch line is marked with ‘+’). The main challenge in this example, is separating
```c
bool bsd_split_3 (char *s, size_t s_len,...) {
    int i = s_len;
    i--;
    if (s_len == 0) return false;
    while (i && s[i] != ')') {
        i--;
    }
    ...
}
```

Figure 2.13: Original and patched version of Coreutils md5sum.c’s bsd_split_3 procedure

<table>
<thead>
<tr>
<th>((\sigma_1))</th>
<th>((\sigma_2) (equivalent))</th>
<th>((\sigma_1) (equivalent))</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_len = 0</td>
<td>s_len' = s_len</td>
<td>s_len' = s_len</td>
</tr>
<tr>
<td>s_len' = 0</td>
<td>i' = i</td>
<td>i = i</td>
</tr>
<tr>
<td>i \leq -1</td>
<td>s_len' - 1 \geq i'</td>
<td>s_len' - 1 \geq i'</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>((\sigma_1))</th>
<th>((\sigma_2) (equivalent))</th>
<th>((\sigma_3) (equivalent))</th>
<th>((\sigma_2) (equivalent))</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_len = 0</td>
<td>s_len' = s_len</td>
<td>s_len' = s_len</td>
<td></td>
</tr>
<tr>
<td>s_len' = 0</td>
<td>i' = i</td>
<td>i = 0</td>
<td></td>
</tr>
<tr>
<td>i \leq -1</td>
<td>s_len' - 1 \geq i'</td>
<td>s_len' \geq 1</td>
<td></td>
</tr>
</tbody>
</table>

(b)

Figure 2.14: Difference for bsd_split_3

the path where \(s._len\) is 0, which results in the loop index \(i\) ranging within negative values (producing an array access out of bounds fault), from the rest of the behaviors that maintain equivalence, throughout the widening process which is required for the analysis to reach a fixed point. As the partitioning maintains equivalence, the path where \(s._len = s._len' \rightarrow 0, ret \rightarrow false, ret' \rightarrow true\) will not be abstracted together with all other paths (that maintain equivalence). The offending path will be widened separately, precisely reporting difference in the final program state for the particular value.

We can see the analysis successfully reports a difference for the singularity point \(s._len = 0\) inside the loop, precisely describing the scenario where \(i'\) is negative. We can also see the other equivalent state existing within the loop which depicts the results of the widened analysis for all other paths (the \(s._len \neq 0\) constraint is not existing there due to partitioning as we will soon show). The differencing sub-state will be omitted once we move past the loop as the \(i \leq -1\) constraint will not allow it to exist beyond the loop body thus we are left with the equivalent state alone after the loop which correctly expresses the fact that the programs are equivalent at this point (since both \(i\)’s converged at 0). We can see that the result at the second differencing point has lost precision since it does not reflect the \(i = 0\) constraint. The loss of this constraint is,
again, due to partitioning as both sub-states that describe exiting the loop and the one describing entering the loop, hold equivalence for all variables and are joined to together and lose the extra constraint information. If we analyze the same example with no partitioning we get the result of Figure 2.14 (b).

Which further separates the paths in the program, allowing for a different sub-state for the \( i = 0 \) and \( i \neq 0 \) substates (again, the \( i \neq 0 \) constraints was lost when joining together the \( i > 0 \) and \( i < 0 \) states as they both adhere to the same path and hold the same guard values). This extra precision is beneficial, but we still managed to supply a satisfactory result using the more scalable partitioning by equivalence technique.

The seq benchmark presented poor results, reporting difference on all variables although the semantic difference is small. This is due to the patch introducing a considerable amount of structural syntactic change to the code. We added the nested benchmark to demonstrate results for a simple nested loop program correlated with itself.

### 2.7 Related Work

Our work has been mainly inspired by recent work identifying program differencing as having vast security implications [BPSZ08, SZS09] as well as advancements made in the field of under-approximations of program equivalence [GS09, KLR10, PDEP08, RE11].

The problem of program differencing is fundamental [Hoa69] and early work mainly focused on computing syntactical difference [HM75]. These solutions are an important stepping stone and we used syntactical diff as a means to achieve interleaving of programs in our correlating program. Another possibility for creating this program is to rely on the editing sequence that creates the new version from the original program [Hor90].

We rely on classic methods of abstract interpretation [CC77] for presenting an over approximating solution for semantic differencing and equivalence. To achieve this we devised a static analysis over a correlating program. The idea of a correlating program is similar to that of self-composition [TA05] except that we compose two different programs in an interleaving designed to maintain a close correlation between them. The use of a correlating construct for differencing is novel as previous methods mainly use sequential composition [GS09, PDEP08, RE11], disregarding possible program correlation.

We base our analysis on numerical abstractions [CH78, Min06] that allow us to reason about variables of different programs. The abstraction is further refined in a way similar to trace partitioning [RM07] with an equivalence-based partitioning criteria.

Jackson and Ladd [JL94] proposed a tool for computing data dependencies between input and output variables and comparing these dependencies along versions of a program for discovering difference. This method may falsely report difference as semantic difference may occur even if data dependencies have not changed. Furthermore, data dependencies offer little insight as to the meaning of difference i.e. input and output values. Nevertheless, this was an important first step in employing program analysis as a means for semantic differencing.

Several works on the problem of equivalence of combinatorial circuits [KK97, MCBE06, CK03] made important contributions in establishing the problem of equivalence as feasible,
producing practical solutions for hardware verification.

Symbolic execution methods [PDEP08, RE11] offer practical equivalence verification techniques for loop and recursion free programs with small state space. These works complement each other in regards to reporting difference as one [PDEP08] presents an over approximating description of difference and the other [RE11] presents an under approximating description including concrete inputs for test cases demonstrating difference in behavior. An interesting question is how could these methods be combined iteratively to achieve better precision. Also, this work can be used to complement our work in cases where equivalence could not be proven and the description of difference can be leveraged for the extraction of concrete input that leads to offending states.

Bounded model checking based work [GS09] presents the notion of partial equivalence which allows checking for equivalence under specific conditions, supplied by the user but are bound by loops. They employ a technique based on theorem provers for proving an equivalence formula which embeds program logic (in SSA form) alongside the requirement for input and output equivalence and user provided constraints.

[ARR+07] introduced a correlating heap semantics for verifying linearizability of concurrent programs. In their work, a correlating heap semantics is used to establish correspondence between a concurrent program and a sequential version of the program at specific linearization points.

In previous work regarding translation validation [PSS98, Nec00, ZPF+02], in order to establish equivalence for a (looping) code fragment being translated or optimized by a compiler, a simulation relation between the basic blocks of the translated code is found. This method is limited in the context of semantic differencing as, for instance, a simulation relation for examples such as Figure 2.2 cannot be automatically established (it needs to be crafted manually as this is not one of the classic transformations). However, the correlating program method we propose is generic enough to establish equivalence for many cases, without requiring special tailoring.
Chapter 3

Abstract Semantic Differencing via Speculative Correlation

In the previous chapter, a technique for producing sound and precise description of difference over two programs was presented. As mentioned, the precision for the description of difference i.e. the tightness of the linear inequalities describing the difference, was heavily affected by the size of the syntactic diff. Figure 1.3 for instance, depicts two semantically equivalent programs which differ vastly syntactically. Analyzing these two programs using techniques from Chapter 2, and specifically generating a correlating program for the two, would sequentially compose the two, which would yield a hugely inaccurate description of difference.

In this chapter we aim to relax the syntactic similarity requirement stated in the problem definition of the previous chapter. Given two programs \((P,P')\), which may differ vastly syntactically, we aim to produce sound and precise description of difference, using the previously defined correlating semantics.

The motivation for this chapter is attributed to real-world examples for program patches, which introduced vast syntactic difference, yet maintained portions of the original functionality. The approach described in this chapter is evaluated on these real-work patches, as well as the benchmarks from the previous chapter. Thus, this chapter motivation is to produce precise semantic difference for any sort of program patching, focusing on changes which largely differ syntactically, yet maintain semantic equivalence for many inputs.

3.1 Overview

3.1.1 Motivating Example

Figure 3.1 shows two versions of the print_numbers function taken from the GNU Coreutils seq.c program. We use primes (’) to denote functions, variables, and program points belonging to the newer version of the program. The functions print out a sequence of numbers, starting from first and ending with last, in intervals of STEP size (the functions have been slightly adapted to an integer-only version, and STEP has been factored out as a constant). Although the syntactic difference between versions is significant, these functions only differ in cases where in print_numbers’ (v6.10), the last element in the sequence is equal (in string form, due to formatting) to the next-to-last element. In this case, the print_extra_number flag will be set to true (line 15’) and the last print would be skipped (by the condition in line 16’). Our goal is to
```c
static void print_numbers (long first, long last, ...) {
    long i;
    for (i = 0; /* empty */ ; i++) {
        long x = first + i * STEP;
        if (last < x) break;
        if (i) fputs (separator, stdout);
        printf (fmt, x);
    }
    if (i) fputs (terminator, stdout);
}
```

```
static void print_numbers (long first, long last, ...) {
    bool out_of_range = (last < first);
    if (!out_of_range) {
        long x0 = x;
        for (i = 1; /* empty */ ; i++) {
            long x = first + i * STEP;
            if (out_of_range) break;
            if (out_of_range) {
                bool print_extra_number = !STREQ(STR(x0), STR(x));
                if (!print_extra_number) break;
            }
            fputs (separator, stdout);
        }
        fputs (terminator, stdout);
    }
}
```

Figure 3.1: Original (top) and patched (bottom) version of Coreutils seq.c’s print_numbers procedure
automatically compute some characterization of this difference, while proving that otherwise the functions are semantically equivalent. To the best of our knowledge, none of the existing techniques [SSCA13, PY13, RE11, GS09, PDEP08] are able to do so.

To precisely characterize the difference for the example of Figure 3.1, it is sufficient to show that at the points of output (line 9 and line 10'), the variable $x$ is equivalent in both versions, in every iteration other than possibly the last iteration, and that the last iteration differs only when the print_extra_number' flag is set (we focus on a single point of output to simplify the explanation).

**Equivalence Check Using Separate Analysis is Unsound** As a naïve attempt to achieve this, one could analyze each version of the program separately and compare the (abstract) results. However, this is unsound, as *equivalence under abstraction does not entail concrete equivalence*. For instance, if we use a standard relational abstraction to track the values of $i$ and $i'$ in the following code fragments separately:

- `for (i=0; i<2*n; i++)
  printf("%d",i);
`  

+ `for (i=0; i<2*n; i=2){
  printf("%d",i);
}
`  

we will arrive at \{0 \leq i \leq 2n\} and \{0 \leq i' \leq 2n'\} which would falsely suggest equivalence, since the left version prints all values in the range, and the right version prints only the even ones. Thus, under abstraction, anything short of *explicit equality* of variables cannot be used to soundly prove equivalence.

**Abstracting Relationships to Prove Equivalence** To establish equivalence under abstraction, we need to abstract relationships between the values of correlating variables in `print_numbers` and `print_numbers'`. Specifically, we need to track the relationship between the values of $x$ and $x'$. This requires a joint representation in which these relationships can be tracked.

To address these challenges, we use a *correlating semantics* $[P, P']$ that maintains a joint shared state for the two programs, and where executions of $P$ and $P'$ are interleaved. We use the $\triangledown$ notation for correlating semantics, states, etc. reflecting the correlation of variables from $P$ and $P'$. The correlating semantics can be then abstracted using numerical domains [CH78, Min06] to obtain shared abstract state. Such abstraction allows us, for instance, to show that when in the previous iteration `print_extra_number' = true`, the relationship between values of $x, x'$ at the point (9, 10') of joint execution can be described by $\sigma^2_{\triangledown} = \{\equiv_{\{\text{first,x, last}\}}, \text{first} \leq x \leq \text{last}\}$. We use the $^2$ superscript to denote abstract entities and $\equiv_{\{\text{first,x, last}\}}$ to denote explicit equality between versions of variables i.e. $\text{first} = \text{first}', x = x'$ and $\text{last} = \text{last}'$. This abstraction proves equivalence (under said clause), even though explicit values of $x$ and $x'$ were abstracted away through widening to overcome the loop (see Section 3.2.4).

**Interleaving Determines Precision** A key aspect of our technique is determining the interleaving in which the programs’ instructions are analyzed. Looking at Figure 3.1 we observe that to produce a precise result, our algorithm needs to analyze the programs together, carefully
Figure 3.2: (Partial) Graphic description of an iteration of the speculative algorithm over print\_numbers and print\_numbers’
interleaving instructions of print_numbers and print_numbers'. We present a speculative algorithm, which performs a correlation-based search for an interleaving that will produce a minimal abstract difference (described in Section 3.3).

The abstraction drives the algorithm: The algorithm chooses an interleaving that minimizes abstract difference. The algorithm speculatively performs analyses in all possible interleavings, and chooses an interleaving that leads to the minimal difference in its abstract state. Figure 3.2 provides some intuition to how interleaving affects equivalence, and we elaborate on it later on as we describe speculative exploration.

This shows a key feature of our technique: interleaving of programs is not static, but is instead computed dynamically in a speculative manner during the analysis. In every iteration, the algorithm re-evaluates equivalence over all interleavings, and chooses a (possibly new) interleaving accordingly. This novel method is fundamentally different from previous methods where a single interleaving is chosen (as a simulation relation for instance) and analyzed. We instead allow for a dynamic interleaving, that may change throughout the analysis. We will first show how the choice of interleaving affects the precision of the resulting abstract difference and then present the algorithm that finds such interleaving.
3.1.2 Running Example

<table>
<thead>
<tr>
<th>print_numbers(...)</th>
<th>print_numbers'(...)</th>
<th>Correlating State $\sigma_{CG}$ adhering to a program counters in print_numbers and print_numbers'</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTRY</td>
<td>ENTRY'</td>
<td>(ENTRY,ENTRY') $\mapsto$ ${\equiv[STEP,first,last]}$</td>
</tr>
<tr>
<td>i long i;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>// loop iteration #0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| i for (i=0; /* empty */; i++){
|   long x = first + i*STEP;
|   if (last<x) break;
|   if (i) fputs(separator,stdout);
| }                   |                                                  |
| ENTRY'            |                   | (8,ENTRY') $\mapsto$ $\{\equiv[STEP,first,last],$ |
|                   |                   | $i = 0, x = first, x \leq last\}$                  |
|                   |                   |                                                  |
| i bool out_of_range' = (last'<x');
| i if (!out_of_range') {
|   long x'=first';
| }                   |                                                  |
|                   |                   | (8,6') $\mapsto$ $\{\equiv[STEP,first,last,x],$ |
|                   |                   | $i = 0, x = first, x \leq last\}$                  |
|                   |                   |                                                  |
| i long i';         |                   | (8,9') $\mapsto$ $\{\equiv[STEP,first,last,x],$ |
| // loop' iteration #0 |                  | $i = 0, i' = 1, x = first$, |
|                   |                   | $x \leq last, x'_0 = x'$                         |
|                   |                   |                                                  |
| i printf(fmt,x);  |                   | (9,10') $\mapsto$ $\{\equiv[STEP,first,last,x],$ |
|                   |                   | $i = 0, i' = 1, x = first$, |
|                   |                   | $x \leq last, x'_0 = x'$                         |
|                   |                   |                                                  |
| i if (out_of_range') break;
| i x' = first' + i' * STEP';
| i out_of_range' = (last' < x');
| i if (out_of_range') {
|   bool print_extra_number'= !STREQ(...);
| }                   |                                                  |
| i if (!print_extra_number) break;
| }                   |                                                  |
|                   |                   | (9,19') $\mapsto$                                   |

44
\[
\begin{align*}
\text{i} &= 0, i' = 1, x' &= x, \\
\text{first} \leq x &= \text{last}, \\
x' &= x + \text{STEP}, \\
\neg \text{print_extra_number'}
\end{align*}
\]

(1++) // loop iteration #1

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>long x = first + i * STEP;</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>if (last&lt;x) break;</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>if (i) fputs(separator,stdout);</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>printf(fmt, x);</td>
<td></td>
</tr>
</tbody>
</table>

(1'++) // loop' iteration #1

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>long x' = x';</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>printf(fmt', x');</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>if (out_of_range') break;</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>x' = first' + i' * STEP';</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>out_of_range' = (last' &lt; x');</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>if (out_of_range') {</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>bool print_extra_number' = !STREQ(...);</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>if (!print_extra_number') break;</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>fputs(separator', stdout');</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{i}' &= 1, i' &= 2, x' &= x, \text{last}, \\
x' &= x + \text{STEP}, \\
\neg \text{print_extra_number'}
\end{align*}
\]
Table 3.1 depicts an analysis over an interleaving that is successful in forming a precise abstraction of equivalence and difference. This interleaving is shown as the solid line in Figure 3.2. Table 3.1 also shows the abstract states throughout the analysis of this interleaving (column 3).

The table is comprised of print.numbers instructions (column 1), print.numbers’ instructions (column 2) and the dual-program correlating state computed so far (column 3). The correlating state in column 3 is mapped to a pair of program counters, denoting values and equivalence of both programs’ variables, at the specified locations. For instance, the correlating state appearing at the bottom of the second row: (8, ENTRY') → {≡{STEP,first,last}, i' = i + 1, x ≤ last, x' = x + STEP, print_extra_number'} expresses that after analyzing up to line 8 in print.numbers, without advancing over print.numbers’, the resulting correlating state holds equivalence for the STEP, first, last variables, alongside other constraints for i, x, first and last. Nothing is known yet for i’, x’, first’ and last’ as none of print.numbers’ lines were analyzed. We omitted most states for brevity, and noted only significant ones, for instance where equivalence

<table>
<thead>
<tr>
<th>// loop iteration #i:</th>
<th>// loop’ iteration #i:</th>
<th>$x' = x + STEP, print_extra_number'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>${≡{STEP,first,last,x}, i' = i + 1, x ≤ last, x' = x + STEP, print_extra_number'$</td>
</tr>
<tr>
<td>9 printf(fmt,x);</td>
<td>10 printf(fmt',x');</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>// loop exit</th>
<th>// loop’ exit</th>
<th>${\equiv{STEP,first,last}, i' = i + 1, x ≤ last, print_extra_number'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 }</td>
<td>19 }</td>
<td>(13,22') → {\equiv{STEP,first,last}, i' = i + 1, x ≤ last, print_extra_number'$</td>
</tr>
<tr>
<td>12 fputs (terminator, stdout);</td>
<td>20 if (i') fputs(terminator', stdout');</td>
<td>\lor {\equiv{STEP,first,last}, i' = i + 1, x ≤ last, print_extra_number'$</td>
</tr>
<tr>
<td>13 }</td>
<td>22 }</td>
<td>-print_extra_number'$</td>
</tr>
</tbody>
</table>

Table 3.1: Analysis order (interleaving) and resulting correlating state for functions taken from Figure 3.1
was restored for a certain variable.

The analysis starts at the entry points of the two functions (ENTRY, ENTRY'), assuming equivalence over input. It then advances towards the line preceding the printing point of print_numbers at locations (ENTRY, 8') and computes an abstract state where \( x = \text{first} \). At this point, the analysis alternates to print_numbers', interpreting instructions up to the printing point in line 10', where equivalence is restored (\( x = x' = \text{first} \)). It is now “safe” to advance towards the output points, as described by the subsequent row in the table.

As mentioned, the choice of interleaving affects precision: had the interleaving instead included further instructions from print_numbers, the analysis would report the loss of equivalence for variable \( x \) at line 9. This demonstrates how interleaving affects precision, as also observed by other approaches [PY13, SSCA13].

The analysis continues from the output locations at (9, 10') to analyze the rest of print_numbers' loop body and iteratively alternates between the versions', keeping a one-to-one loop iteration ratio, reflecting that analyzing both loops in a controlled manner is key for maintaining precision. Paths that break from the loop, when print_extra_number' is true, are abstracted by states mapped to (9, 19') (as they break from the loop). These paths are added to paths where print_extra_number' = true at the end of the analysis, as one can see from the abstract state mapped to (13, 22'). We discuss partial disjunction later in this section.

We emphasize that our algorithm does not rely on advancing towards output locations (as one may suspect from Table 3.1). It searches for an optimal interleaving based on minimizing difference alone. The interleaving in Table 3.1 was chosen as it simplifies the explanation. Interleavings that do not alternate on output locations are equally valid (as long as they minimize difference).

Speculative Exploration in Search of Equivalence We now describe our speculative exploration algorithm which iteratively computes interleavings. We present an algorithm which computes analyses over all possible interleavings, within a given speculation window. It then proceeds to greedily select the interleaving that results in an abstract state with minimal abstract difference. Because our analysis is sound, and can never miss a difference, picking the minimal abstract difference necessarily means better precision. This process iteratively continues, until a fixed-point is reached.

The exploration process is essentially performing all possible analyses, in all possible interleavings, and comparing them to pick the one with minimal abstract difference. Each analysis advances an overall \( k \) number of steps over both programs, in a different interleaving from all other analysis, where \( k \) is the parameterized speculative windows size. We note that interleaving need not be balanced, i.e. one analysis can perform all \( k \) steps of \( P \) and zero steps over \( P' \). We show that this novel dynamic method is superior to previous methods as it does not rely on the syntactic structure of the program [PY13] nor does it require any concrete data [SSCA13] to help establish a matching of program points.

Figure 3.2 shows a (partial) graphical representation of an iteration of the speculative algorithm, running from the start of the programs (4, 4') up to the first printing point (9, 10') (this
is in fact the first iteration of the algorithm). The vertical and horizontal axes correspond to instructions of \texttt{print\_numbers} and \texttt{print\_numbers'} respectively, i.e. moving up means advancing over a line from \texttt{print\_numbers} and moving right means advancing over \texttt{print\_numbers'}. The circles in the graph denote abstract states computed by the analysis, which correspond to program locations according to the numbering on the axis. Equivalent and differing states are colored differently. Areas in the graph marked by $\equiv_v \{v\}$ denote states where $v$ is equivalent ($v = v'$). Lastly, the arrows in the graph represent an advancement over a line of either program, interpreting that line and reaching a new abstract state at the new location. A path from $(4, 4')$ to $(9, 10')$ represents an interleaving of the two programs and the states over that path are the result of analyzing the programs in that interleaving.

Figure 3.2 features two interleavings: a less precise one, marked by the dotted line, as it travels through more non-equivalent states and will falsely report difference at line 9 (as it reaches $(5, 9')$, equivalence for $x$ has yet to be restored). The second interleaving, marked by a solid line, is a more precise one, and in fact corresponds to the interleaving shown in Table 3.1, up to the first printing points.

During an iteration of the speculative algorithm, it advances over all possible interleavings, up to the speculation window, effectively analyzing the two programs in all orders. It then assigns values to each interleaving according to equivalence criteria. Graphically, this would show as having all the possible paths of length $k = 12$, originating from $(4, 4')$ in Figure 3.2. Some paths will reach beyond the boundaries of the graph since an interleaving can perform all steps over just one of the programs. Paths will also be limited by the programs structure (if $P$ reached an exit point, the path will no longer advance over it).

The value assigning metric mainly relies on equivalence, but also contains a weighing element as certain program points (like labels containing back-edges) receive a higher score for equivalence (see Section 3.3). When equivalence is broken, the size of the difference is used, where the interleaving(s) with the minimal diff are scored.

Finally, once an interleaving is selected, the iteration completes and another speculative phase of the algorithm starts. For example, after the iteration of Figure 3.2 is completed and the precise interleaving is chosen, the abstract state computed by the analysis will include all of the states along the chosen interleaving, and the next iteration will begin from $(9, 10')$, i.e., where the last iteration left off.

**Partial Order Reduction** Speculatively exploring for all possible program interleavings up to $k$ steps results in performing $2^k$ analyses. To allow scalability, we employed a partial order reduction [Val89, WG93]. Given a speculative window value $k$, instead of exploring all interleavings, we choose a single representative adhering to the number of steps taken in each program. For example, if $k = 3$ then instead of exploring all 8 interleavings, only 4 will be analyzed: (i) 0 steps over $P$ and 3 steps over $P'$, (ii) 1 step over $P$ and 2 steps over $P'$, (iii) 2 steps over $P$ and 1 step over $P'$ and (iv) 0 steps over $P$ and 3 steps over $P'$. This reduces the number of analyses in each speculative step to $k + 1$ resulting in a more scalable analysis. The reduction allows experimenting with larger values for $k$ with minor loss of precision,
since the (precision losing) partition operation is performed only in between speculative steps 
(Algorithm 3.1). The reduction is implemented in Function 3.2. Another means of achieving 
scalability is advancement over program blocks instead of lines. A block is a subsequent, 
non-branching group of instructions in the program. Thus, advancing over a speculative window 
of $k$ means advancing over $k$ blocks which translates to approximately $4k$ lines of code, in our 
experiments.

Separating Equivalent Paths From Differing Paths As one can see from the last row in Ta-
ble 3.1, we use a partially disjunctive abstract domain that maintains a set of abstract numerical 
sub-states. This allows us to maintain several sub-states based on the set of equivalences they 
maintain. The ability to hold separate states for equivalent and differing paths is crucial for 
precision. Maintaining a fully disjunctive domain with complete path sensitivity does not scale, 
especially in the face of loops. Therefore we use a partitioning technique to abstract together 
sub-states according to an equivalence criteria. Sub-states that prove equivalence for the same 
set of variables will be abstracted (joined and widened) together.

The benefits of using equivalence-based partitioning can be seen in Table 3.1 (row 5 onwards) 
where in the computed abstract state, differing paths are separated from equivalent paths. States 
where print_extra_number' is false are gathered at the loop exit (as they cause a break in 
the loop) alongside equivalent states where print_extra_number' is true. As described earlier, 
we use an equivalence criteria to partition and widen sub-states together, thus all the looping 
differing states are widened together, separately from the equivalent states, producing a precise 
result as seen in the last row of the table. Details of our partially disjunctive domain are discussed 
in Section 3.2.

Note that relying on syntactic methods to find such interleaving can prove to be challenging 
as print_numbers’ includes refactoring as well as the actual patch that changes behavior, thus 
previous approaches [PY13] will fail in reaching this result.

3.1.3 Uninterpreted Functions

```c
static int
get_sha1_basic(const char *str, int len, 
unsigned char *sh1, 
int warn_ambiguous_refs) {
    ...
    if (len > 0 && str[len-1] == '}') {
        for (at = len - 2; at >= 0; at--) {
            if (str[at] == '@' && str[at+1] == '{') {
                if (upstream_mark(str + at, len - at) > 0) {
                    reflog_len = (len-1) - (at+2);
                    len = at;
                }
            }
        }
        break;
    }
    ...
}
```

Figure 3.3: get_sha1_basic procedure code fragment taken from Git’s sha1_name.c
Figure 3.3 depicts a code fragment, taken from the Git project sha1_name.c file. This fragment features a challenging set of operations, including array access by index and function calls. In order to successfully reason over such rich and complex code features, SCORE uses an uninterpreted functions modeling technique [GS09, LHKR12].

Any operation that is not supported by the underlying APRON domain, including operations on non-integer data, is modeled by the analysis as an uninterpreted function. For example, Figure 3.3 code fragment’s array indexing and function call operations will be modeled as external functions and the following equivalence deduction rule will be applied during the analysis:

\[
\begin{align*}
\text{ARRAY INDEXING} & : \\
array & \equiv idx \\
\Rightarrow array[idx] & \equiv up\text{stream}\_mark(v_1,v_2)
\end{align*}
\]

We assume initial equivalence for arrays by denoting explicit equality on the array variable (this satisfies the \(\equiv array\) predicate in the rule). Once equivalence is broken for a certain index, for instance by assigning non equal values to it, equivalence is broken for the entire array. In case of actual functions, like \(up\text{stream}\_mark\), equivalence is assumed only if the function did not (syntactically) change over versions, or if equivalence was proven for it.

3.1.4 Path Boundedness

**Analyzing Unbounded Paths with \(k\)-Bounded Syntactic Divergence** We emphasize that \(k\) (our notation for speculative window size) bounds only the size of syntactic divergence explored throughout the analysis. It is in no way a bound for the length of paths analyzed by out approach. The analysis depicted in Table 3.1 shows an analysis for unbounded paths in looping code. The speculative window only determines how far can the analysis advance over one program before advancing over the other. For instance, in Figure 3.2 the precise interleaving advanced four lines of code over the first program and only then turns to analyze five lines of the second program, etc. Furthermore, \(k\) only bounds the size of local syntactic change.

Consider the two procedures in Figure 3.4 exhibiting several syntactic differences, each localized to a few consecutive lines of code. In this example, all lines hidden by ellipses are equal, while differing lines remain. The semantic effect of these lines was also omitted for brevity. The speculative analysis will produce a precise description of difference with \(k\) that is bounded only by the largest consecutive syntactic change. The local speculative exploration at each of the changed code fragments will result in a precise description of the change, as depicted by the linear equations in the third column. These are produced throughout the speculative analysis, as further described in Section 3.3. Therefore even though there are overall many syntactic changes on paths longer than \(k\), the speculative analysis produces a precise result.

3.2 Speculative Correlating Semantics

In this section, we introduce our speculative correlating semantics which allows bounded representation of program difference through correlation of variables, within some divergence
Figure 3.4: Procedures featuring several (overall changes > k) local syntactic changes (each change < k), along with correlating abstract state (column 3) adhering to executing the blocks from both programs.

window. In Section 3.2.2, we define a concrete correlating semantics that tracks relationships between variables of both programs. Then, in Section 3.2.3, we show how to abstract this semantics using numerical domains. This sets the scene for the algorithm we present in Section 3.3.

3.2.1 Preliminaries

We use the following standard definitions for a program: \( \text{Var}, \text{Val}, \text{Loc} \) denote the set of program variable identifiers, variable values and program locations respectively.

A concrete program state \( \sigma \) is a pair \( l \mapsto \text{values} \), mapping the set of program variables to their concrete value \( \text{values} : \text{Var} \rightarrow \text{Val} \), at a certain program location \( l \in \text{Loc} \). The set of all possible states of a program \( P \) is denoted by \( \Sigma_P \). We also define \( \text{loc} : \Sigma_P \rightarrow \text{Loc} \) which returns the program location of a state.

A program trace \( \pi \in \Sigma^*_P \), is a sequence of states \( \langle \sigma_0, ..., \sigma_n \rangle \) describing a single execution of the program. The set of all possible traces for a program is denoted by \([P]\). We also define \( \text{last} : \Sigma^*_P \rightarrow \Sigma_P \) which returns the last state in a trace.

3.2.2 Concrete Correlating Semantics

We define a concrete semantics that (i) maintains direct relationships between variables in \( P \) and \( P' \), using a correlating concrete state and (ii) define a notion of dual execution of \( P \) and \( P' \) within a restricted window of divergence, using k-diverging correlating concrete traces (by k-diverging we mean that one program may be at most k-steps ahead of the other).

**Correlating Concrete State** A correlating concrete state \( \sigma_{\text{corr}} \) is a pair \( (l, l') \mapsto \text{values}_{\text{corr}} \), where \( \text{values}_{\text{corr}} : (\text{Var} \cup \text{Var}') \rightarrow \text{Val} \) is a joint mapping of variables, from both programs \( (P, P') \) to their values at program locations \( (l, l') \in \text{Loc} \times \text{Loc}' \). The set of all correlating states is

\[
\{ x, y, w, o, z' = z - 1 \} \\
\{ y, w, o, z' = z - 1, x = 2x' \} \\
\{ y, w, o, y \leq 42, z' = z - 1, x = 2x' \} \lor \{ y, w, 47 \geq y > 42, z' = z - 14, o' = z' + \text{zoo}(3), ... \} \lor \{ y, z, w, 47 > y > 47, o = o' + z, x = 2x' - 2 \}
\]
denoted by $\Sigma_{P,P'}$.

**k-Diverging Correlating Concrete Trace** A $k$-diverging correlating concrete trace $\pi_{\triangleleft k,0} \in \Pi_{\triangleleft k,0}$, is a sequence of correlating concrete states $\langle \sigma_{\triangleleft 0}, \ldots, \sigma_{\triangleleft n} \rangle$ describing a bounded dual execution of $P$ and $P'$ where at any given point in the trace, the number of $P$ instructions performed cannot exceed those of $P'$ by more than $k$. We denote the set of $k$-diverging correlating traces of $P$ and $P'$ as $[\Pi_{\triangleleft 0}]_{k,0}$. Informally, this can be thought of as a trace of a parallel execution $P || P'$ where one thread cannot go more than $k$ steps ahead of the other.

**Concrete Semantic Difference** We define concrete semantic difference of a correlating state $\Delta_{\sigma}(\text{varcor}, \sigma_{\triangleleft 0}) \subseteq \sigma_{\triangleleft 0}$ to be all the mapped variables in $\text{values}_{\text{cor}}$ that do not agree on a value with their correlated variable under a matching $\text{varcor} : \text{Var} \times \text{Var}'$. We assume $\text{varcor}$ to be either a matching by variable name, or supplied by the user. Next, we define concrete trace semantic difference. We further restrict our semantics and only consider traces that originate from equivalent input states, as we are interested in comparing executions of $P$ and $P'$ which originate from the same input values. Concrete semantic difference of a $k$-diverging correlating concrete trace $\Delta_{\pi}(\text{labels}, \text{varcor}, \pi_{\triangleleft k,0})$ is defined by the semantic difference of states along the trace of which the pair of labels $(l,l')$ are in $\text{labels}$. $\Delta_{\pi}$ produces a sub-trace of $\pi_{\triangleleft k,0}$ according to matched labels. $\text{labels}$ usually holds the pair of exit labels, along with pairs of labels where output is emitted. We assume $P, P'$ have the same number of output locations and thus can be matched by order, as this was the case in our experiments.

The semantics $[\Pi_{\triangleleft 0}]_{k,0}$ is non-computable. In the next section, we define an abstract semantics that over-approximates it.

### 3.2.3 Abstract Correlating Semantics

In this section, we introduce our correlating abstract domain which allows bounded representation of program state while maintaining equivalence between correlated variables. We use a disjunctive domain to allow separate representation of different paths. Therefore, an abstract state in our domain is a set of abstract sub-states. This abstraction is similar to the trace partitioning domain [RM07].

We use relational abstract domains to hold variable information in the sub-states. In the following we assume an abstract relational domain $(D^2, \sqsubseteq_D)$ equipped with join $\sqcup_D$, meet $\sqcap_D$ and widening $\nabla_D$ operations, for abstracting sets of concrete correlating states. We assume the abstract values in $D^2$ are constraints over the $P$ and $P'$ variables, and do not go into further details regarding the particular abstract domain as it is a parameter of the analysis. In our experiments, we use the polyhedra abstract domain [CH78] and the octagon abstract domain [Min06].

**Correlating Abstract State** A correlating abstract program state $\sigma_{\triangleleft 0}^\pi \in (\text{Loc} \times \text{Loc}') \rightarrow 2^{D^2}$, is a mapping from a pair of program locations to a set of sub-domain factoids, each representing a relational abstraction of the variables.

**Abstract Transformers** Our domain’s abstract transformers, which define how each program statement affects an abstract state, are based on the abstract transformers of the underlying
domain. Applying a statement \(s\) on an abstract correlating state \(\sigma_{\text{cor}}^k = (l, l') \mapsto \{d_1 \lor \ldots \lor d_n\}\) will result in the application of the statement on each of the sub-states using the sub-domains’s transformer i.e. \([s]^{2\Pi_1^k}_{\text{cor}}(\sigma_{\text{cor}}^k) = (l, l') \mapsto \{[s]^{2\Pi_1^k}_{D_1}(d_1) \lor \ldots \lor [s]^{2\Pi_1^k}_{D_n}(d_n)\}\)

**Abstract Semantic Difference** Given the abstraction, we define the abstract semantic difference \(\Delta(\text{labels}, \text{varcor}, \alpha^k)\) as the maplets \((l, l') \mapsto \{d_1 \lor \ldots \lor d_n\} \in \alpha^k(T)\) such that:

(i) \((l, l') \in \text{labels}\)

(ii) \(\exists d_i, (v, v') \in \text{varcor}. d_i \not\simeq (v = v')\)

This singles out abstract states where some factoids \(d_i\) cannot prove explicit equivalence \(v = v'\) for matched variables under \(\text{varcor}\). Note that using \(d_i \not\simeq v \neq v'\) as the criteria is insufficient since as mentioned, anything short of explicit equality under the abstraction is unsound for

\[\alpha^k(T) \triangleq \bigvee_{(l, l') = \text{loc}(\text{last}(\pi_{\text{cor}}^k))} \bigcup_{\pi_{\text{cor}}^k \in \text{path}(T)} \alpha_{\text{D}_i}(\text{last}(\pi_{\text{cor}}^k))\]

where \(\alpha_{\text{D}_i}\) is the abstraction function of the sub-domain.

We break down the definition as follows:

1. \(\alpha^k\) groups together all trace prefixes \(\pi_{\text{cor}}^k\) from \(T\) that share the same path, as denoted by \(\text{path}(\pi_{\text{cor}}^k) \in \text{path}(T)\).

2. It then abstracts together (using the underlying join operation \(\bigcup\)) the concrete states at the end of the grouped traces as denoted by \(\alpha_{\text{D}_i}(\text{last}(\pi_{\text{cor}}^k))\).

3. Finally, it disjuncts all paths that arrive at the same end locations \((l, l')\) denoted by \((l, l') \mapsto \text{loc}(\text{last}(\pi_{\text{cor}}^k))\).

Every pair of paths in \(P\) and \(P'\), will be represented by a single factoid of the sub-domain (denoted \(d_i\)), and all path pairs that arrive at locations \((l, l')\) will be represented by the abstract state \((l, l') \mapsto \{d_1 \lor \ldots \lor d_n\}\). The run of Algorithm 3.1 (Section 3.3) with parameters \((P, P', k)\) effectively returns \(\alpha^k(T_{\text{P,P'}})\) where \(T_{\text{P,P'}}\) is the set of traces of a dual execution of \(P\) and \(P'\). Next, we address the fact that our abstract state may still be potentially unbounded as the number of paths in the program may be exponential and even infinite (due to loops).

**Abstract Semantic Difference** Given the abstraction, we define the abstract semantic difference \(\Delta(\text{labels}, \text{varcor}, \alpha^k)\) as the maplets \((l, l') \mapsto \{d_1 \lor \ldots \lor d_n\} \in \alpha^k(T)\) such that:

(i) \((l, l') \in \text{labels}\)

(ii) \(\exists d_i, (v, v') \in \text{varcor}. d_i \not\simeq (v = v')\)

This singles out abstract states where some factoids \(d_i\) cannot prove explicit equivalence \(v = v'\) for matched variables under \(\text{varcor}\). Note that using \(d_i \not\simeq v \neq v'\) as the criteria is insufficient since as mentioned, anything short of explicit equality under the abstraction is unsound for
proving equivalence. We assume the underlying domain $D$ provides a means for performing these checks.

We note that in terms of fixed-point analysis, this means that our join operation $\sqcup$, that abstracts together states of converging paths is simply a disjunction operation, and the only application of the sub-domains $\sqcup^D$ operation will occur during partitioning, as we will next describe.

### 3.2.4 Dynamic Partitioning and Widening

As a first means of reducing state size, we define a special operation $\sqcup$, that dynamically partitions the abstract state according to the set of equivalences maintained in each sub-state and joins all sub-states in the same partition class together (using the sub-domain join operation). This join criteria allows separation of equivalence preserving paths from differing ones, thus achieving better precision. Second, to allow a feasible bounded abstraction for programs with infinite number of paths, we define a widening operator $\nabla$, which utilizes the sub-domain’s widening operator but chooses which sub-states are to be widened, according to the same equivalence criteria.

**State Size Reduction Using Equivalence-Based Partitioning** As mentioned, we must allow for reduction of state $\sigma_{\nabla} = (l, l') \mapsto \bigvee d_i$ with acceptable loss of precision. This reduction is achieved by joining the abstract sub-states in $\bigvee d_i$ (using the standard join of the sub-domain). However this can only be accomplished after first deciding which of the sub-states should be joined and then choosing the program locations for the partitioning to occur. Our partitioning strategy abstracts together sub-states according to the set of variables which they preserve equivalence for. This bounds the state size at $2^{|\text{varcor}|}$, where $\text{varcor}$ is the set of correlating variables we wish to track.

In order for our analysis to handle loops we require a means for abstracting an infinite number of paths in a bounded way. As our analysis iterates over a loop, sub-states may be added or transformed continuously, never converging. We therefore need to define a widening operator for our new domain. This problem has been addressed in other settings [BHZ06], and our approach can be viewed as a specialized form of widening that is tailored for tracking equivalences. We found that our partitioning strategy works well for widening, as it allows separation of equivalent looping paths from differing paths. Given two subsequent abstract states $\sigma^2_{\nabla^{POP}} = (l, l') \mapsto \bigvee d^0_i$ and $\sigma^2_{\nabla^{P1}} = (l, l') \mapsto \bigvee d^1_j$ corresponding to two subsequent iteration of a loop, the result of applying widening will be as follows:

$$\nabla (\sigma^2_{\nabla^{POP}}, \sigma^2_{\nabla^{P1}}) \triangleq (l, l') \mapsto \bigvee d^{D} (d^0_i, d^1_j)$$

Where $\equiv (d)$ returns the set of variables (from $\text{varcor}$) that are equivalent in $d$.

### 3.3 Speculative Search Algorithm

In this section, we present our speculative search algorithm. Given a speculation window size $k$, the algorithm explores all possible interleavings of the two programs up to $k$ steps, and then
employs an equivalence-based scoring heuristic to greedily select the abstract state with the minimal abstract difference. This abstract state is used as a basis for the next iteration of the speculative algorithm, which proceeds until reaching a fixed point. This novel dynamic approach allows to dynamically explore different interleavings on each step of the analysis, instead of deciding on a fixed interleaving a priori.

### Algorithm 3.1 Compute abstract difference

**Require:** Two programs $P, P'$, Speculation window size $k$, Partition interval $p$, A matching $varcor : Var \leftrightarrow Var'$ of the correlating variables in both programs.

**Ensure:** A mapping $statespace : Loc \times Loc' \mapsto \sigma^d_{\mathcal{F}}$ from pairs of prog. locations to correlating abstract state.

```plaintext
num_speculations ← 0
workset ← \{(entry, entry')\}
statespace ← \{(entry, entry') ↦ \{v \equiv v'|(v, v') \in varcor \cap (In \times In')\}\}
while workset \neq ∅ do
    solutions ← Speculate ($P, P'$, workset, statespace, $k$)
    num_speculations += 1
    (workset, statespace) ← FindMinDiffSolution ($P, P'$, varcor, solutions)
    if num_speculations mod $p = 0$ then
        statespace ← Partition (statespace)
    end if
    statespace ← Widening (statespace)
end while
return statespace
```

### Algorithm 3.2 Speculate ($P, P'$, workset, statespace, $k$)

**Require:** As specified in Algorithm 3.1.

**Ensure:** A set $solutions : \{(workset_1, statespace_1), ..., (workset_n, statespace_n)\}$ specifying the work-set and state-space generated from advancing $k$ steps over $P, P'$ in all possible interleavings, under a partial order reduction

```plaintext
(workset_0, statespace_0) ← (workset, statespace)
for ((i, j) ← (0, k); i ≤ k; i++, j -) do
    for (t = 0; t < i; t++) do
        (workset, statespace) ← Step ($P, workset, statespace$);  
    end for
    for (t = 0; t < j; t++) do
        (workset, statespace) ← Step ($P', workset, statespace$);  
    end for
    solutions ← solutions ∪ \{(workset, statespace)\};
    (workset, statespace) ← (workset_0, statespace_0);
end for
return solutions
```
Algorithm 3.3  Step \((P, workset, statespace)\)

Require: As specified in Algorithm 3.1.
Ensure: A solution \((workset, statespace)\) adhering to advancing a step over \(P\)

\[
\begin{align*}
\text{workset}_\text{res} & \leftarrow \emptyset \\
\text{for } (\text{loc}, \text{loc}') \in \text{workset} & \text{ do} \\
\text{if } \text{loc} \in P & \text{ then} // \text{advancing over first graph} \\
\text{for } \text{succ} \in \text{successors(\text{loc})} & \text{ do} \\
\sigma & \leftarrow \text{statespace}(\text{loc}, \text{loc}') \\
\text{block} & \leftarrow \text{blocks}(P, \text{loc}, \text{succ}) \\
\text{statespace}(\text{succ}, \text{loc}') & \leftarrow \text{statespace}(\text{succ}, \text{loc}') \lor [\text{block}]^B(\sigma) \\
\text{workset}_\text{res} & \leftarrow \text{workset}_\text{res} \cup \{(\text{succ}, \text{loc}'){\}} \\
\text{end for} \\
\text{else if } \text{loc}' \in P & \text{ then} // \text{advancing over second graph} \\
\ldots // \text{the symmetrical case} \\
\text{end if} \\
\text{end for} \\
\text{return } (\text{workset}_\text{res}, \text{statespace})
\end{align*}
\]

3.3.1 Iterative Speculative Correlation

Algorithm 3.1 produces an abstraction of program difference in a form of a mapping from pairs of program locations of \(P, P'\) to correlating abstract states.

The input to the algorithm is: (i) two programs \(P\) and \(P'\), (ii) a speculation window \(k\) which determines how many speculative steps the algorithm may take, (iii) a mapping \(\varcor\), matching variable names between variables of \(P\) and \(P'\), describing which variables should be correlated. In our experiments, we show that matching variables that have the same name and appear in both programs is sufficient for producing precise results. However, in general, this mapping can be provided by the user or by other methods such as using data traces [SSCA13].

The algorithm operation is similar to standard abstract interpretation fixed-point algorithms [CC77]. It starts by adding the entry point of the two programs, denoted by \(\text{entry}\) and \(\text{entry}'\), to the empty \(\text{workset}\), and initializes the \(\text{statespace}\) by mapping these locations to correlating state that assume equivalence over input variables, denoted by \(\text{In}\) and \(\text{In}'\). The algorithm iteration then starts, as it interprets program lines from the \(\text{workset}\), creating and updating the \(\text{statespace}\) with abstract states mapped to pairs of program locations, while adding lines where state has changed back to the \(\text{workset}\), until a fixed-point is reached. Note that Algorithm 3.1 differs from standard algorithms as it: (i) Operates over two programs instead of one. (ii) Performs \(k\) steps of the analysis instead of just one, i.e. it interprets \((0, k), (1, k - 1), \ldots (k, 0)\) lines of the programs through \text{Speculate} resulting in \(k + 1\) solutions (pairs of \(\text{workset}\) and \(\text{statespace}\)). (iii) Dynamically chooses the solution with the minimal abstract difference by applying \text{FindMinDiffSolution} which surveys the \(\text{statespace}\) of each solution and assigns a score to it according to an equivalence based weighed metric. (iv) Partitions \(\text{statespace}\) according to partition interval \(p\) (v) Applies widening to states mapped to blocks with back-edges.

Greedy Dynamic Selection of Interleaving We again emphasize an important dynamic feature
Algorithm 3.4 FindMinDiffSolution \((P, P', vc, solutions)\)

**Require:** As specified in Algorithm 3.1.

**Ensure:** A pair \((workset, statespace)\) containing the work-set and state-space of the most precise solution

\[
\text{factor} \leftarrow |\text{Loc}| \cdot |\text{Loc}'|
\]

**for** \((l, l') \in P \times P'\) do

- equivalence\((l, l') \leftarrow \text{false}\)
- \(\text{min\_delta} \leftarrow \infty\)
- \(\text{min\_delta\_solutions} \leftarrow \emptyset\)

**for** \((\text{workset}, \text{statespace}) \in \text{solutions}\) do

- score \(\leftarrow 0\)

  **if** \(\equiv_{\text{varcor}}(\text{statespace}(l, l'))\) **then**

  - equivalence\((l, l') \leftarrow \text{true}\)

  **if** \(\text{has\_backedge}(l) \vee \text{has\_backedge}(l')\) **then**

    - score\((\text{workset}, \text{statespace}) \leftarrow \text{factor}^2\)

  **else**

    - score\((\text{workset}, \text{statespace}) \leftarrow \text{factor}\)

**end if**

**end if**

**end for**

**if** \(\neg\text{equivalence}(l, l')\) **then**

**for** \((\text{workset}, \text{statespace}) \in \text{solutions}\) do

  - if \(|\Delta(\text{statespace}(l, l'))| = \text{min\_delta}\) **then**

    - \(\text{min\_delta\_solutions} \leftarrow \text{min\_delta\_solutions} \cup \{(\text{workset}, \text{statespace})\}\)

  **else if** \(|\Delta(\text{statespace}(l, l'))| < \text{min\_delta}\) **then**

    - \(\text{min\_delta} \leftarrow |\Delta(\text{statespace}(l, l'))|\)

    - \(\text{min\_delta\_solutions} \leftarrow \{(\text{workset}, \text{statespace})\}\)

**end if**

**end for**

**end if**

**for** \((\text{workset}, \text{statespace}) \in \text{min\_delta\_solutions}\) do

  - score\((\text{workset}, \text{statespace}) \leftarrow 1\)

**end for**

**end for**

\(\text{max} \leftarrow \text{Max}(\text{Range}(\text{score}))\)

**return** \((\text{workset}, \text{statespace})\) s.t. \((\text{workset}, \text{statespace}) \in \text{score}^{-1}(\text{max})\)

---

of our speculative run: picking an interleaving over a range of program locations in one stage, does not determine the interleaving over these locations for the rest of the analyses. Speculate will always return the analysis of \(k + 1\) interleavings, and the one with minimal difference will be picked by FindMinDiffSolution. Therefore, the decision could vary in different stages of the run, based on the size of difference in the candidate solutions. The importance of this is exemplified in Table 3.1, where the order in which we analyze lines 4 through 9 in print_number and lines 4’ to 11’ in print_number’ changes between iterations of the algorithm and the rows of the table. Next, we will describe the Speculate function and explain how an interleaving is explored and how all interleavings are returned in the forms of solutions.
State Size Reduction With Partitioning

The number of speculative steps taken is kept in the \texttt{num_speculations} variable which is used to determine when to partition. Partitioning is crucial for performance as it reduces the size of the disjunction. We found that partitioning at parameterized constant intervals, is sufficient in producing scalable precise results. We experimented with different values for \( p \) to refine the result and achieve better precision. More complicated strategies for picking partition location include using the syntactic structures of the programs to find locations where these “converge” [PY13] however we did not find the need to apply other methods as precision was satisfactory.

Widening

We maintain the number of visits to each pair of program locations and perform widening once a predetermined threshold has been reached. Widening is performed only if one of the lines in the pair (at least) has a back-edge reaching it, this is sufficient for reaching a fixed point as further discussed in Section 3.2.

3.3.2 Speculative Advancement Over All Interleavings

The \texttt{Speculate} function produces \( k + 1 \) solutions, representing the result of analyzing \( P, P' \) in by advancing \( k \) steps over both programs, in different distributions of steps. The functions receives the two programs \( P, P' \), a speculative window size \( k \) and the state of the analysis so far captured by \((\text{workset}, \text{statespace})\). It then proceeds to extend the solution by performing \( k \) steps over both programs, starting with 0 steps over the first and \( k \) over the second, continuing to perform \((1, k - 1)\) steps, and so on as can be seen by the procedure’s main loop.

Advancing a step over either program \( P'(l') \) is performed by taking all location pairs in the \texttt{workset} and advancing over \( P'(l') \) locations as captured by the \texttt{Step} procedure. Step employs the abstract transformer described in Section 3.2.3 over the locations successors. For instance, if \((l, l') \in \text{workset} \) and \( l_{\text{succ}} \) is a successor of \( l \) (in \( P \)), then the effects of advancing from \( l \) to \( l_{\text{succ}} \) will be interpreted by: (i) Retrieving the state adhering to \((l, l')\) from \texttt{statespace}. (ii) Retrieving the basic block (i.e. the edge) between \( l \) and \( l_{\text{succ}} \) by applying \texttt{blocks}(\( P, l, l_{\text{succ}} \)). (iii) Applying the abstract transformer of the block on the retrieved state and joining with the state at \texttt{statespace}(\( l_{\text{succ}}, l' \)) (if such exists, otherwise \texttt{statespace}(\( l_{\text{succ}}, l' \)) will only hold the result). (iv) Adding \((l_{\text{succ}}, l')\) to the resulting \texttt{workset_res}. The case of advancing from \((l, l')\) to \( l'_{\text{succ}} \) in \( P' \) is symmetrical.

3.3.3 Comparing Abstractions to Find Minimal Difference

The final stage of the speculative iteration is the selection of a single solution out of the set of \texttt{solutions}, as performed by the function \texttt{FindMinDiffSolution}. Our goal is to find a solution, whose states differ minimally, since it necessarily means a more precise result (due to the soundness of the analysis). \texttt{FindMinDiffSolution} defines a scoring heuristic which ranks abstractions based on equivalence and minimal difference. It surveys all the possible pairs \((l, l') \in \text{Loc} \times \text{Loc}'\) and compares the abstract state mapped to that pair in \texttt{statespace}(\( l, l' \)) against all other states, awarding points according to the size of the difference.

For each pair of program locations \((l, l')\), the algorithm tries to find a solution which “proves equivalence” i.e. for all matched variables in \((v, v') \in \text{varcor}, \text{statespace}(l, l') \vdash v = v'\). This
means that in all sub-states in the disjunctive abstract state $statespace(l, l')$, $v$ is equal to $v'$. In the algorithm this is denoted by $\equivvarcors\ (statespace(l, l'))$. In case such a solution is found, $(l, l')$ is flagged using the equivalent flag and other solutions will be given points there only if they also prove equivalence. This reflects the fact that the existence of equivalence for a pair of lines in one interleaving invalidates differences for that pair in other interleavings. Also, this is a design choice aimed at improving performance, as the computation and comparison of difference can be costly and will be omitted once equivalence is found.

In case no such solution is found for $(l, l')$, the function continues on to compare the size of difference. We experimented with different ways of comparing difference and arrived at using the number and (textual) size of the offending states in the disjunction as criteria, denoted by $|\Delta (statespace(l, l'))|$. We found this heuristic to be efficient yet accurate enough to produce precise results in reasonable analysis time. We also experimented with a formal method for comparing difference, via an algorithm for manipulating abstract states $\sigma^\Delta$ to extract disjunctive states that describe difference and can be compared over a lattice using the $\sqsubseteq$ operation. However, since this algorithm is computationally expensive and does not improve precision dramatically, we refrained from using it.

The existence of equivalence awards the solution score a factor $|Loc| \cdot |Loc'|$ amount of points, while the solution with minimal difference (in case equivalence was not found) is given a single point. This reflects that a solution with one equivalent state is preferred over a solution where all states have minimal difference. Solutions that did not arrive at $(l', l)$ are not considered and do not get points for it.

We added a weighing component to our metric, where locations with entering back-edges (denoted by the $has\backedge\ (l)$ predicate), receive $factor^2$ points, in case they prove equivalence. This change in order of magnitude was added to help direct the search towards lines with entering back-edges, that prove equivalence. Equivalence at these locations is important as widening is performed there. Since widening over-approximates specific numeric data, widening correlating states that do not already prove equivalence, will likely result in a non-restorable loss of equivalence. Thus, we will prefer solutions where abstract states at back-edge locations prove equivalence. This lowers the risk of losing equivalence once widening occurs.

### 3.4 Evaluation

In this section, we evaluate SCORE using a number of real world programs, and compare it to the state of the art in equivalence checking and semantic differencing. Our goal is to assess how SCORE measures against challenges presented by each method, and whether it can be applied to produce useful results. As benchmarks, we used several programs drawn from the Git source code [Git], GNU Coreutils, as well as a few patches from the Linux kernel and the Mozilla Firefox web browser. We also collected benchmarks used as motivating examples in state of the art work, applied SCORE to them and present the comparison here.
3.4.1 Prototype Implementation

We implemented speculative correlation, as described in Algorithm 3.1, based on the LLVM and CLANG compiler infrastructure. We chose to analyze C code directly as it is more structured, has type information and keeps a low number of variables, as opposed to intermediate representation. Our analysis is intra-procedural and handles function calls, as well as operations not modeled by the numerical domain, as uninterpreted functions (Section 3.1.3). We used the APRON abstract numerical domain library and conducted our experiments using the Polyhedra domain [CH78]. We ran our experiments on an Intel(R) Core(TM) i7-2620M CPU @ 2.70 Ghz machine with 4GB installed RAM.

3.4.2 Benchmark Results

Table 3.2 summarizes the results of our analysis. The columns indicate the function name, length in lines of code, the number of lines added and/or removed by the patch, total number of loops, and the run time in minutes along with the minimal speculative window k and partition interval p values that produced minimal difference. The results are separated (by a horizontal line) according to project the function was taken from, as follows: GNU Coreutils, Linux kernel, Firefox, Git and related work representatives.

Results produced by SCORE are abstractions of the data dependencies and equivalence of variables in the program. SCORE supplies the resulting state space, as a mapping from all pairs of
```c
static int write_zip_entry(int zip_dir,
    int zip_dir_size,
    int zip_dir_offset,
    int pathlen, ...) {
    ...
    /* make sure we have enough
       free space in the dictionary */
    int direntsize = ZIP_DIR_HEADER_SIZE +
                    pathlen + ZIP_EXTRA_MTIME_SIZE;
    bool flag = false;
    while (zip_dir_size < zip_dir_offset + direntsize) {
        zip_dir_size += ZIP_DIRECTORY_MIN_SIZE;
        - zip_dir = xrealloc(zip_dir,zip_dir_size);
        + flag = true;
    }
    + if (flag)
    +     zip_dir = xrealloc(zip_dir,zip_dir_size);
    ...
```

Figure 3.5: Patch for Git archive.zip.c’s write_zip_entry procedure

program locations to correlating state for those pairs (as described if Section 3.3). SCORE further reports whether differences were found in observable program locations (output and exit points) and prints out such states holding difference in textual form.

We experimented with a speculative window $k$ of up to 4. We chose increasing partition interval sizes $p$ as well, up to a maximum of 5. We capped run time at two hours for each SCORE run and reported the minimal $p$ and $k$ which resulted in the smallest difference ($\Delta$) at output and exit locations. The size of speculation window $k$ could usually be predicted by the size of the biggest addition or deletion of subsequent lines to the code. In some cases, like the get_sha1_basic v2 benchmark, vast semantic differences required a large speculation window, and reported differences for smaller $k$ sizes were unusable. The benchmark was therefore reported to time out with no acceptable precision output given.

Another interesting case is the 340-line long write_zip_entry benchmark depicted in Figure 3.5. The patch here involves changing a few lines of code where the memory allocation to zip_dir is hoisted out of the loop (line marked with - in Figure 3.5) and instead performed only after zip_dir_size has been determined (lines marked with + in Figure 3.5). The appropriate lookahead window here is $k = 2$, to overcome additions by the patch. Running SCORE while partitioning at $p = 1$ (after each speculation) yields equivalence. Interestingly enough, SCORE was able to maintain equivalence for zip_dir, although the value for it was widened to overcome the loop. This is due to partitioning and widening both operating by equivalence: at the loops end, the analysis accumulates states holding the $zip_dir = xrealloc(...), flag = true$ predicates, alongside states withholding these predicates but instead hold equivalence for $zip_dir$ (states that did not enter the loop). These two categories of states were partitioned and widened separately due to equivalence criteria and equivalence was restored for zip_dir once the patched lines were reached. This demonstrates one of SCORE’s strong points: the ability to restore equivalence from equivalence classes.
Our experiments included other numerical domains such as Octagon [Min06], however we did not include the results as they exhibit poor precision and performance. Moving from Octagon to Polyhedra domain, a notable increase in precision was shown as the Polyhedra domain is able to capture more complex data relationships. Surprisingly, runs using the Octagon domain presented poor performance (run time) compared to the more expensive Polyhedra domain, with less precision. This is due to the domain’s being less successful in capturing equivalences as it is built upon simpler linear inequalities. This means that more constraints were needed to represent variable equality, resulting in bigger states and a slower analysis.

SCORE produced results with high precision where only variables that indeed differ between versions were reported, and the description of the difference was useful for producing actual values for the differing variables. In cases where equivalence holds, no difference was reported.

The addr and SetTextInternal benchmarks were taken from the net/sunrpc/addr.c module in the Linux kernel SUNRPC implementation v2.6.32-rc6 and Firefox 3.6 security advisory CVE-2010-1196 (adapted to C from C++) respectively. The results produced by SCORE can be directly used towards exploiting known security flaws mentioned in advisories from which these patches originate. Figure 3.6 shows the patch made in the CVE-2010-1196 advisory, fixing a heap buffer overflow in the Firefox browser. Running SCORE on the SetTextInternal function yields the following output:

$$\Delta[EXIT,EXIT] =$$

$$\{ Ret = 0, Ret' = 1, newLength > 536870912 \} \lor$$

$$\{ Ret = 0, Ret' = 1, 0 > newLength > -3758096384 \}$$

The Ret variable, was added by SCORE to instrument function return. Also, special handling was added for interpreting integer casting. The produced result is useful to the programmer, to ensure that vulnerable ranges have been covered and that the function ends for these ranges. The results could also be used by an attacker to deduce the vulnerability fixed by the patch.

### 3.4.3 State of The Art Comparison

**Syntactic Correlation Based Techniques** The approach described in [PY13] aims at proving C-code function equivalence and producing a textual representation of difference for equivalence
checking, patch debugging etc. We evaluated SCORE on several benchmarks taken from [PY13].

Results show that SCORE yields results at similar or higher precision. One downside of this previous work, is illustrated by our motivating example from Figure 3.1. The seq.c benchmark introduces considerable amounts of textual change which defeats the syntactic reliant method suggested in [PY13]. An integral part of this method involves creating a unification program, containing both versions, to be used by the analysis. The precision of the analysis relies on this unified program and its ability to bring together (syntactically) instructions that are “equivalent” in both programs. However, for the seq.c benchmark, this correlating program will be poorly formed, unable to syntactically match the versions, which will result in an imprecise result. Another shortcoming of this method is the addition of guards and the need to syntactically transform C code to guarded command form, a process proved to be challenging and erroneous. These guard variables are also incorporated into the domain, resulting in a more expensive analysis.

Data Driven Techniques Figure 3.7 depicts the worked example from [SSCA13], describing a semantic preserving compiler optimization. This work aims to prove equivalence for looping assembly code segments for translation validation purposes. This dynamic method uses data traces in order to establish a simulation relation between code segments and then attempts to prove this relation by using off-the-shelf SAT solvers.

We ran SCORE on Figure 3.7 to see whether equivalence can be established. SCORE reported the following result as the abstract state at the exit point of both programs, within 13.3 seconds ($k = 1, p = 8$): \(\{\equiv_{\{x, i\}}, i = n, z = 5z', i \leq z\}\). SCORE was able to prove equivalence for \(x\) and capture the exact relationship of \(z\) and its patched counterpart \(z'\): \(z = 5z'\). The strongpoint of [SSCA13] is the ability to produce this result on machine code, where syntactic differences are bigger.

Recursion Rule Based Techniques [GS09] applies a recursion rule to verify equivalence of recursive functions. This work uses recursive calls within candidate functions and assumes their equivalence as the basis of the recursive verification rule. It then tries to inductively prove equivalence by showing that all paths to the recursive call in both versions are equivalent, using bounded model checkers [CKSY04]. Although this technique is able to deal with recursion, it requires the recursive call to be nested under the exact same conditions in both programs, disallowing the use of the recursive rule in many cases. Our motivating example requires complex
manual rewrite to adhere to such form, as well as many other benchmarks. As mentioned, SCORE assumes modular equivalence of function calls it encounters, thus, we were able to adapt it to implement the recursion rule from [GS09], by simply assuming equivalence on recursive calls and proving equivalence by showing equivalence over the recursive call parameters. The benchmark appears as RegVer in Table 3.2. [LHKR12] applies a similar path-proving technique, however they do not use the recursive rules as they do not handle loops. The strong point of this method, is that it handles input programs written in the Boogie verification language [BCD+05]. Boogie has translations from C, C# and x86 assembly, making [LHKR12] language agnostic. The benchmark appears under SymDiff in Table 3.2.

Directed Symbolic Execution Based Techniques

Work such as [PDEP08] and [RE11] uses symbolic execution testing tools, such as KLEE [CDE08] and DART [GKS05], to find equivalence bugs. To prove equivalence between $P$ and $P'$, these techniques sequentially compose the programs (while assuring the same input) and then add an assertion checking if return values of both versions are equal. The new program is then fed to an automatic test generation tool which in turn tries to explore all paths in the program, effectively exploring all path compositions of $P$ and $P'$ and checks if any of these paths break the assertion. This technique heavily relies on the effectiveness of the test generation tool and benefits from its features, as such the ability to reason over pointer and heap data. In addition to this method being dynamic, it is fundamentally unable to reason over looping programs, and loops are mitigated by a simple unrolling to some constant number of iterations which is of course un-sound. We drew one benchmark from [PDEP08], which appears as DSE, containing a loop. [RE11] benchmarks were omitted as they focused on pointer and heap data.

3.4.4 Quality of Semantic Diff

Next we examine the quality and usefulness of SCORE output. We compare the semantic diff produced by SCORE to several other methods, focusing on the most prominent tool for comparing programs—syntactic diff.

Equivalence of Refactored Looping Code

Figure 3.8 depicts a modified version our motivation example (Figure 3.1) where only the refactoring was applied to change code structure, while the semantic changing part was removed. The two versions of the print_numbers procedure are output equivalent as the printed variable $x$ is equivalent at the print location. Running diff on the two versions would produce the entire text of the programs. This provides no insight into change semantics. SCORE output consists of: (i) an abstract description of versions state, in the form of linear equations, at all joint program locations and specifically at the output location (ii) for each variable in said state, the equivalence status is reported (all equivalent variables is explicitly reported as program equivalence). Thus, the output of SCORE for Figure 3.8 is shown in Table 3.3. The output correctly describes the change in the programs, emphasizing the fact that $x$ is equivalent at the output location. The state further holds the loop invariant for $x$ value which is useful for program understanding. Previous techniques (described in Section 3.4.3) are unable to provide a useful description of
Figure 3.8: Output-equivalent versions of Coreutils seq.c’s print_numbers procedure

<table>
<thead>
<tr>
<th>Location (9,10):</th>
<th>Equivalent: first, last, x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Equivalent:</td>
<td>i, x₀, out_of_range</td>
</tr>
<tr>
<td>State = {first′ − x₀′ + 2i − 2 = 0, first′ + 2i − x = 0, first′ − x′ + 2i = 0, first′ − 2i + last ≥ 0, i − 1 ≥ 0}</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: SCORE Output for Figure 3.8

the change since they are either inherently incapable of handling loops [PDEP08, RE11]; rely on specific loop structure [GS09, LHKR12]; rely on syntactic similarity [PY13] or work on smaller lower-level code [SSCA13].

Previous Path Knowledge

In many cases, a locally applied patch could be affected by previously executed code as shown in Figure 3.9 depicting the pr benchmark code taken from Coreutils. The lines added by the patch are marked by a boldface plus sign (+). Most of the benchmark code was omitted for brevity, except for the patch and a preceding loop involving values affected by the patch. Here, the semantics of the patch are related to previously executed code over many execution paths. Referring locally to the patch itself, as the result of a diff would, provides no information towards the change in variable values affected by the patch. SCORE instead provides a useful description of the change in variable values due to patching (Table 3.4). The output features variable values adhering to the two paths affected by the change. This result is more conducive towards helping the programmer understand and verify their patch.

3.5 Related Work

Equivalence checking and semantic differencing has received much attention lately where more and more works have been applied towards advancing the state of the art in some aspect. Similar to many verification problems, this problem has been presented nearly sixty years ago,
static int input_position = 0;
static int char_to_clump (int chars_per_c, int width, bool untabify_input, ...) {
    int chars = 0, i;
    char *s;
    ...
    if (untabify_input) {
        i = width;
        while (i > 0) {
            *s++ = '␣';
            --i;
        }
        chars = width;
    }
    ...
    + if (width < 0 && input_position == 0) {
    +     chars = 0;
    +     input_position = 0;
    + } else if (width < 0 && input_position <= -width) {
    +     input_position = 0;
    + } else {
    +     input_position += width;
    + }
    return chars;
}

Figure 3.9: Original and patched (+) version of Coreutils pr.c's char_to_clump procedure

---

At Location (EXIT,EXIT'):
Equivalent: width,chars_per_c,untabify_input,...
Non-Equivalent: input_position,chars
State = {width = width' = −1, chars = 1, chars' = 0, input_position = −1, input_position' = 0, ...
\text{ \lor }
{width = width' = −1, chars = chars' = 1, input_position \leq 0, input_position' = 0, ...}

Table 3.4: SCORE Output for Figure 3.9

66
with few mentions in the years following [Hoa69, HM75, HPR89, Hor90] as it is considered undecidable. Recently, work identifying program differencing as having vast security implications [BPSZ08, SZS09] has surfaced, bringing the problem back to mainstream. Also, the considerable advancements made in the field of SAT solvers and its application to various verification problems, lead to a plethora of work leveraging off-the-shelf SAT solvers towards program equivalence [PDEP08, GS09, RE11, LHKR12, SSCA13]. Several works on the problem of equivalence of combinatorial circuits [KK97, MCBE06, CK03] made important contributions in establishing the problem of equivalence as feasible, producing practical solutions for hardware verification.

The idea of a correlating semantics and reasoning about correlated execution has been investigated in many contexts before (e.g., [TA05, ARR+07]). In fact, our approach can be seen as an invariant generation technique for relational logics such as Relational Hoare Logic (RHL) [Ben04].

An early work presenting a usable tool for semantic diff is [JL94]. They present a tool for computing data dependencies between input and output variables and comparing these dependencies along versions of a program for discovering difference. This method may falsely report difference as semantic difference may occur even if data dependencies have not changed. Furthermore, data dependencies offer little insight as to the meaning of difference i.e. input and output values. Nevertheless, this was an important first step in employing program analysis as a means for semantic differencing.

Aiken et. al. [SSCA13] present a data driven approach for translation validation. This approach requires the running of the program and data traces to help establish a simulation relation to later be proved by SMT solvers. Also, as far as we could discern, they are limited to small segments of assembly code, unable to handle bigger, higher language code. We compare to this work in our evaluation.

David et. al. [DY14] address equivalence of short non-branching sections of binary code (i.e. ‘tracelets’) in the context of binary code search. They define a notion of equivalence for binary code sections and apply rewriting and constraint solving techniques for finding equivalence as a basis for whole function similarity. These techniques can be further extended using speculative exploration to overcome syntactic obstacles and to apply to looping code. This can help find similarly even for highly re-factored, optimized or patched code.

Other work regarding translation validation [PSS98, Nec00, ZPF+02], require establishing a simulation relation between the basic blocks of the translated code is found. This method is limited in the context of semantic differencing as, for instance, a simulation relation for examples such as Figure 3.1 cannot be automatically established (it needs to be crafted manually as this is not one of the classic transformations).

Symbolic execution based methods [PDEP08, RE11] offer practical equivalence verification techniques for loop and recursion free programs with small state space. These works complement each other in regards to reporting difference as one [PDEP08] presents an over approximating description of difference and the other [RE11] presents an under approximating description including concrete inputs for test cases demonstrating difference in behavior. These cannot be
applied towards our setting as they handle loop free, finite state programs only.

[GS09] presents the notion of partial equivalence which allows checking for equivalence under specific conditions, using a recursive rule. They employ a technique based on theorem provers for proving an equivalence formula which embeds program logic (in SSA form) alongside the requirement for input and output equivalence and user provided constraints. As mentioned in Section 3.4, this work requires the rewrite of programs and loops recursive constructs, and applies only when the path conditions leading to the recursive call are the same in both candidate version, making it limited and inapplicable to our setting.
Chapter 4

Interprocedural Semantic Change Impact Analysis

In previous chapters we presented techniques for computing sound and precise abstract difference for procedures. However, we did not apply these techniques to whole, inter-procedural programs as generally, performing whole program inter-procedureal analysis does not scale to more than a few hundred lines.

In this chapter, we produce an algorithm for performing inter-procedural differential program analysis, while leveraging previous knowledge of semantic equivalence at the procedure level. Given a program $P$, consisting of a number of procedures which invoke each other, and a list of syntactic changes, we aim to produce a tight bound over the set of semantically impacted program statements. In order to be scale for whole programs with thousands of lines, we do not perform abstract interpretation analysis over the whole program, but instead perform a change-impact analysis which allows the approach to scale.

The impact of reducing the set of impacted program lines can be directly and clearly quantified. Reducing change impact can be applied to reduce the set of regression tests which need to be run, or reduce the set of modules which need to be reviewed in the code review process.

4.1 Overview

Figure 4.1 shows a running example written in C with several procedures. The example is inspired by real commits to Coreutils, in files paste.c [corb] and sort.c [corc]. The program has three changes. Two semantics-preserving changes: (i) extracting the literal '$n'$ into the variable line_delim in the procedure print_product_info and (ii) replacing the branch with a double negation in locale_ok. A third change affects semantics by setting the line_delim variable to '$0$' in the procedure print_product_info which impacts statements in print_minor_version. We intuitively claim that the only (unchanged) line that is impacted by the change is the highlighted line in print_minor_version due to the semantic change at the callsite. All the statements in print_header, print_name and print_major_version() are not impacted by the changes. For brevity, the example does not contain the definitions of the setlocale and locale_format procedures as well as the LC_ALL and HEADER constants as they are not impacted and irrelevant. We will now analyze the change through the lens of dataflow analysis and equivalence checking, and then sketch our technique.

A dataflow analysis technique will start with the source of changes and then propagate them...
+ static unsigned char line_delim = '\n';

int print_product_info(int name, int version) {
    int locale, printed = 0;
    - print_header('\n');
    + print_header(line_delim); // spurious arg taint
    locale = locale_ok(); // spurious summ taint
    if (name) {
        printed = print_name(locale);
    }
    if (version && printed) {
        printed = print_major_version(locale));
        + line_delim = '\0';
        + printed = print_minor_version(locale, line_delim));
        - printed = print_minor_version(locale, '\n'));
    }
    return printed;
}

void print_header(char delim) {
    printf("%s%c",HEADER,delim);
}

int locale_ok() {
    - return setlocale (LC_ALL,"") ? 1 : 0;
    + return !!setlocale (LC_ALL,"";?></span></pre><p><span class="highlighted">Figure 4.1: Example. Changes made to Coreutils in files paste.c and sort.c. The lines with − and + represent deleted and added lines respectively.</span></p></div><div class="highlighted" style="color: #555555; font-family: monospace; font-size: 16px; white-space: pre-wrap;">through data and control edges (typically in the changed program). Data-flow techniques are not aware of the change-semantics, and thus cannot exploit equivalence preserving changes. Initially, the call to print_header has a change which will mark all of the statements in this procedure as impacted. Next, the call to locale_ok will impact the locale variable because of the change to the body of locale_ok. This in turn will mark the input of print_name as impacted, which in turn flows to its output (locale affects the return value). (A context-insensitive analysis will further impact the return at all call-sites to print_name(), not shown in the figure.) This will</div>
propagate to the call to \texttt{print_major_version()} and \texttt{print_minor_version()} and impact all the statements in these procedures as well as all the returns at all call-sites. Finally, the call to \texttt{print_minor_version()} will correctly impact all of the callee statements, even though the functionality of the \texttt{print_minor_version()} has not changed. A context sensitive (relational) analysis will not help either since the body of \texttt{locale.ok()} changes, which implies that the return value may change across the two versions. This is \textit{imprecise} since the analysis will be unable to determine that the statements in \texttt{print.name()} and \texttt{print_major_version()} are not impacted.

A traditional interprocedural equivalence checking [GS09, LHKR12] (which only checks if two procedures have identical input-output behavior) will determine that \texttt{locale.ok()}, \texttt{print_header()}, \texttt{print.name()}, \texttt{print_major_version()} and \texttt{print_minor_version()} have identical summaries. This is \textit{unsound} with respect to impact analysis, as the statement of \texttt{print_minor_version()} is actually impacted due to the change of print delimiter. This illustrates the difference between CIA and equivalence checking, two procedures can be equivalent, but still impacted, because they may get called under different contexts.

In this work, we present a \textit{change semantics aware} CIA that works as follows: We use differential invariants to determine that the arguments in call-sites to \texttt{print.name()} and \texttt{print_major_version()} are equal across both versions at all call-sites, not propagating impacts through the arguments to them. Further \texttt{locale.ok()} has an equivalent summary in the two versions (by using equivalence checking) — this ensures that if two call-sites with equal arguments return equal results. Together, we can conclude (by simple dataflow) that arguments to \texttt{print.name()} and \texttt{print_major_version()} are unchanged and the statements in \texttt{print.name()} and \texttt{print_major_version()} are not impacted. This way, our approach will precisely identify and highlight the only unchanged line that is indeed impacted.

4.2 Background

In this section, we formalize the programming language and notations.

4.2.1 Programming Language

A program \texttt{Prog} \in \texttt{Programs} consists of (i) a set of procedures \texttt{Procs}_\texttt{Prog} \subseteq \texttt{Procs}, (ii) an entry procedure \texttt{main} and (iii) a mapping \texttt{StmtAt}_\texttt{Prog} : \texttt{N} \rightarrow \texttt{Stmts} that maps a node \texttt{n} \in \texttt{N} in a procedure \texttt{f} to a statement. \texttt{main} is the (unique) entry procedure from which the program execution starts. Note that we distinguish the node’s identifier (which we will usually denote \texttt{n}) from the node’s statement \texttt{StmtAt(n)}. (We omit \texttt{Prog} from \texttt{Procs}_\texttt{Prog} and \texttt{StmtAt}_\texttt{Prog} when the program is clear from the context.)

A procedure \texttt{f} \in \texttt{Procs} is a tuple \((\texttt{N}_f, \texttt{E}_f, \texttt{In}_f, \texttt{Out}_f, \texttt{Vars}_f, \texttt{n}_1, \texttt{n}_2)\), where:

- \texttt{N}_f is a set of control-flow locations in \texttt{f},
- \texttt{E}_f \subseteq \texttt{N}_f \times \texttt{N}_f is a set of edges over \texttt{N}_f denoting control-flow,
- \texttt{In}_f is a vector of input formals of \texttt{f},
- \texttt{Out}_f is a vector of output formals of \texttt{f},
• \( \text{Vars}_f \) is a set of variables of \( f \) and includes \( \text{In}_f, \text{Out}_f \) and local variables of \( f \),

• \( n^e_f \in N_f \) is the unique entry node of \( f \),

• \( n^x_f \in N_f \) is the unique exit node of \( f \),

Let \( N = \bigcup_{f \in \text{Procs}} N_f \) and \( \text{Vars} = \bigcup_{f \in \text{Procs}} \text{Vars}_f \). Nodes and variables in a procedure \( f \) are often denoted by \( n_f \) and \( x_f \) respectively. For simplicity of exposition, we assume (without loss of generality) that the program does not have any global variables — global variables (including the heap) are passed as additional input and output formals of a procedure. However, our implementation supports explicit globals.

A statement \( st \in \text{Stmts} \) can be one of:

• An \textit{assign} statement of the form \( x := e \) where \( x \in \text{Vars}_f \) and \( e \) is a type-consistent expression,

• An \textit{assume} statement of the form \( \text{assume} e \) where \( e \) is a Boolean-valued well-scoped expression,

• A \textit{skip} statement of the form \( \text{skip} \),

• A \textit{call} statement \( \text{call} x_1, x_2, \ldots, x_k := g(e_1, e_2, \ldots, e_m) \), where \( g \in \text{Procs} \), \( e_i \) are expressions representing the actuals and \( x_j \in \text{Vars}_f \) capture the output formals of \( g \).

For any procedure \( f \), we assume that \( \text{StmtAt}(n^f_f) = \text{skip} \). For any node \( n \in N_f \), we define the \textit{readset} \( \text{RVars}(n) \) and \textit{writeset} \( \text{WVars}(n) \) as the set of variables that are read and written to respectively in the statement at \( n \).

An expression \( e \in \text{Exprs} \) in the language is built up from:

• \textit{Constant} values of different types such as \{ \text{true}, \text{false} \} for Booleans, \{ \ldots, -1, 0, 1, \ldots \} for integers etc.,

• Variables from \( \text{Vars} \) in scope,

• Operations over expressions \( \text{op}(e_1, \ldots, e_k) \), where \( \text{op} \) is a function or predicate symbol, and \( e_i \) are expressions. The function and predicate symbols can be uninterpreted or interpreted by some theories (e.g., \{ +, -, *, \leq, \geq, \ldots \} by the theory of arithmetic, and \{ sel, update \} by the theory of arrays [BST10]).

We keep the expression language open for this paper, and assume that expressions in the language are well-typed according to some type checking rules. We represent a vector of variables and expressions using \( \overline{x} \) and \( \overline{e} \), respectively.

We use several forms of syntactic sugar to keep the examples readable. First, we use the notation \( n_1 : st; \text{goto} n_2, n_3 \) to express that \( \text{StmtAt}(n_1) = st, \{ (n_1, n_2), (n_1, n_3) \} \subseteq E_f \). We use \( n_f : \text{return} \) to abbreviate \( \text{StmtAt}(n_f) = \text{skip} \) and \( (n_f, n^x_f) \in E_f \). Conditional statements \( \text{if} (e) st_1 \text{ else } st_2 \) are modeled by using \text{goto} and \text{assume} constructs as further discussed in Section 4.3.1 and [BCD+05].
The language also accommodates richer data types such as arrays and maps, where an array read \( x[e] \) is modeled using \( \text{sel}(x, e) \) and a write \( x[e] := e_2 \) is modeled using \( x := \text{update}(x, e_1, e_2) \). Here \( \text{sel}, \text{update} \) are functions interpreted by the theory of arrays in the Satisfiability Modulo Theories (SMT) [BST10] framework. Arrays are used to model the heap in programs and such abstractions are fairly standard in several tools [FLL+02, CHLQ09]. Additional internal non-determinism (e.g. reads from file, network) is lifted as reads from immutable input arrays of main, making programs deterministic in our language [LHKR12].

### 4.2.2 Semantics

Let \( \mathcal{V} \) denote the set of values that variables and expressions can evaluate to. Let \( \theta \in \Theta \) be a store mapping variables to values in \( \mathcal{V} \). For \( x \in \text{Vars} \), we define \( x \in \theta \) if \( x \) is a variable in the domain of \( \theta \). For \( x \in \theta \), \( \theta(x) \) denotes the value of variable \( x \). The store \([x \mapsto \nu]\) represents a singleton store that maps \( x \) to \( \nu \). The store \( \theta|_{\text{Vars}_1} \) restricts the domain of the store to variables in \( \text{Vars}_1 \). For stores \( \theta_1 \) and \( \theta_2 \), the store \( \theta_3 = \theta_1 \oplus \theta_2 \) is defined as follows for any variable \( x \in \theta_1 \) or \( x \in \theta_2 \):

\[
\theta_3(x) = \begin{cases} 
\theta_2(x), & \text{if } x \in \theta_2 \land \neg (x \in \theta_1) \\
\theta_1(x), & \text{if } x \in \theta_1
\end{cases}
\]

The value of an expression \( e \in \text{Exprs}(\theta(e)) \) is defined inductively on the structure of \( e \) (we omit it for brevity as it is fairly standard).

Let \( cs \in (N \times \text{Vars}^* \times \Theta)^* \) be a call stack that is a sequence of tuples \(((n_0, r_0, \theta_0), (n_1, r_1, \theta_1), \ldots)\), where \( n_i \) is the \( i \)-th call-site on the call stack (\( n_0 \) is the most recent), \( r_i \) and \( \theta_i \), respectively, are the vector of return actuals and the valuation of the local variables of the caller, at the corresponding call-site. Let \( CS \) denote the set of all such call stacks, \( e \) denotes an empty stack, and \((n, r, \theta) :: cs\) denotes the concatenation operator.

A state \( \sigma \in \Sigma \) is a tuple \((n, \theta, cs) \in N \times \Theta \times CS \) that denotes a point in program execution where \( n \) is the current program counter in a procedure \( f \), \( \theta \) is the valuation of variables in \( \text{Vars}_f \) and \( cs \) is the current call stack.

A state transition denoted as \((n_f, \theta_1, cs_1) \rightarrow (n_2, \theta_2, cs_2)\) is a relation over \( \Sigma \times \Sigma \) holds only if:

1. \( \text{StmtAt}(n_f) = x := e, n_2 \in N_f, \theta_2 = \theta_1 \oplus [x \mapsto \theta_1(e)], (n_f, n_2) \in E_f, \text{ and } cs_1 = cs_2, \) or

2. \( \text{StmtAt}(n_f) = \text{assume } e, n_2 \in N_f, \theta_1(e) = \text{true}, (n_f, n_2) \in E_f, \theta_1 = \theta_2 \text{ and } cs_1 = cs_2, \) or

3. \( \text{StmtAt}(n_f) = \text{skip}, n_f \neq n_f^f, n_2 \in N_f, (n_f, n_2) \in E_f, \theta_1 = \theta_2 \text{ and } cs_1 = cs_2, \) or

4. \( \text{StmtAt}(n_f) = \text{call } r := g(\pi). \) Let \( n \) be the unique successor of \( n_f \) in \( f \), and \( \pi \) be the vector of input formals for \( g \) in \( n_2 = n_2^x, cs_2 = (n, r, \theta_1) :: cs_1 \) and \( \theta_2 = [\pi \mapsto \theta_1(\pi)] \), or

5. \( \text{StmtAt}(n_f) = \text{skip}, n_f = n_f^f, cs_1 = (n_x, r, \theta_3) :: cs_3. \) Let \( \pi \) be the vector of output formals for \( f \) in \( n_2 = n_x, \theta_2 = (\theta_3 \oplus [\pi \mapsto \theta_1(\pi)])|_{\text{Vars}_x}, cs_2 = cs_3. \)
A transitive edge $\sigma_0 \rightsquigarrow^* \sigma_n$ exists if $\sigma_n \equiv \sigma_0$ or there exists a sequence of transitions $\sigma_0 \rightsquigarrow \sigma_1, \ldots, \sigma_{n-1} \rightsquigarrow \sigma_n$, where $\sigma_i \rightsquigarrow \sigma_{i+1}$, for all $i \in [0, \ldots, n)$. For a procedure $f$, we denote the input-output transition relation $\Omega_f = \{(\theta_1, \theta_2) \mid (n^f_1, \theta_1, \epsilon) \rightsquigarrow^* (n^f_2, \theta_2, \epsilon)\}$.

An execution trace $\tau$ is a (possibly infinite) sequence of states $\langle \sigma_0, \sigma_1, \ldots \rangle$, where $\sigma_i \rightsquigarrow \sigma_{i+1}$, for any adjacent pair of states in the sequence. For a trace $\tau$ and a node $n \in N$, $\tau_n$ denotes the (maximal) subsequence of $\tau$ containing states of the form $(n, \_, 
olinebreak\_)$. For a sequence of states $\sigma$ and a variable $x \in Vars, \sigma|_x \in (V \cup \{\bot\})^*$ denotes the sequence of values $\sigma$ with the same length as $\sigma$, and

$$\nu_i = \begin{cases} \theta(x), & \sigma_i = (\_, \theta, \_), \text{ and } x \in \theta \\ \bot & \text{otherwise} \end{cases}$$

For such a trace $\tau$ of length at least $i + 1$, $\tau[i]$ denotes the state at position $i$ (namely $\sigma_i$). For any procedure $f$, let $\Gamma_f$ be the set of maximal traces of $f$. That is, $\Gamma_f$ is the set of all traces $\tau$ such that (i) $\tau[0] \equiv (n^f_1, \_, \epsilon)$, and (ii) either (a) the final state $\sigma_n$ has no successors, or (b) the trace is non-terminating. Traces with no successors can either terminate normally in a state $(n^f_1, \_, \epsilon)$, or could be blocked due to no successors in $E$ or due to an unsatisfied assume statement.

For a store $\theta \in \Theta$, we denote $\tau_f(\theta)$ as the maximal trace (due to determinism) of $f$ that starts in a store $\theta$.

### 4.3 Problem Statement

In this section, we formalize the problem of semantic change impact analysis and provide a simple solution based on dataflow-based static analysis.

#### 4.3.1 Representing Changes

We denote $Prog^1, Prog^2 \in Programs$ as two versions of a program $Prog$. Similarly $\sigma^i, \theta^i, \tau^i$ denote entities for $Prog^i$, without making $Prog^i$ explicit.

To simplify the formulation and avoid additional mappings between two versions, we assume the two programs in a normalized form, where (i) the set of procedures $Procs$ are identical (their signatures), and (ii) for each $f \in Procs$, the vector of variables in $Vars_f$, and the set of nodes $N_f$ (but not necessarily $E_f$) are identical. We can easily preprocess the programs to obtain their normalized forms, by introducing additional procedures, variables (uninitialized) and nodes. Finally, for any missing node $n$, we add an unreachable node in $N_f$ with skip statement and empty successor list.

We now describe how we represent the syntactic change in a procedure, which forms the source or seed of the impact. Recall that our normalization ensures that the set of nodes in $N_f$ is identical across the two versions. For each procedure $f$, we identify a subset of nodes in $N_f$ that have identical statement and identical set of successor identifiers, as the set of mapped (unchanged) nodes. Note that the successors nodes need not be mapped themselves, as we are only defining the syntactic change to the node itself. The remaining set of nodes in $N_f$ over-approximate the sources of syntactic change.

**Definition 4.3.1 (MAPPED).** We denote a node $n_f \in N_f$ as a mapped node if both $StmtAt(n_f)$
and the set of successor node identifiers of $n_f$ in $E_f$ are identical in both programs. We define the set \text{MAPPED} as the set of mapped nodes.

Let us informally state an important property relating the mapped nodes. For any pair of executions of the deterministic programs $\text{Prog}^1$ and $\text{Prog}^2$ starting in identical input states and identical mapped nodes, if the sequence of nodes in the two executions solely consists of mapped nodes, then the two executions yield identical sequence of states. The intuition is as follows: In the absence of any internal non-determinism, a mapped node either has no successors or a unique successor node for a given state (either the \textit{then} or the \textit{else} branch). If the successor nodes are also mapped, then they have to be identical as well, and would execute the same statement in the successors, and so on. This implies that two executions starting at identical states and identical nodes need to have at least one non-mapped node to illustrate a difference. In other words, the source of impact is limited to the non-mapped nodes, although change impact can flow to mapped nodes from an unmapped node.

Thus, the mapped nodes \text{MAPPED} underapproximate the set of nodes that are syntactically unchanged. Let $\text{Procs}^\Delta \subseteq \text{Procs}$ be the set of procedures that have undergone some syntactic change. It is easy to establish that the set of non-mapped nodes is an upper bound of the set of syntactically changed nodes. For a procedure $f \in \text{Procs} \setminus \text{Procs}^\Delta$, $N_f$ are trivially mapped as their control-flow graphs are identical in two versions. For a procedure $f \in \text{Procs}^\Delta$, the goal of any diff algorithm is to maximize the set of mapped nodes for the procedures. In practice, the diff algorithm starts with two control-flow graphs with no identifier names and assigns node identifiers by matching the control-flow graphs of the two versions of a procedure $f$. Our formulation is thus parameterized by such a diff algorithm that can either be based on text \cite{Yan91} or more sophisticated notions such as abstract syntax trees \cite{FMB+14} or program-dependency-graphs \cite{Kri01}.

Finally, we assume the input programs in a simplified form for the static analysis as follows: (i) Each node $n \in N_f$ has at most two successor nodes in $E_f$, where nodes with two successors correspond to conditional statements. (ii) The only use of an \texttt{assume} statement is to model such a conditional statement. (iii) We further assume that such a conditional statement is transformed into the form $n_1: c := e; \texttt{goto } n_2, n_3: \texttt{assume } c; \ldots$, and $n_3: \texttt{assume } \neg c; \ldots$ where a fresh Boolean variable $c$ captures the value of the condition $e$. We refer to $n_1$ as a \textit{branching} node with two successors in $E$ with complementary expressions in \texttt{assume} statements. The transformation simplifies determining if control flow is impacted by looking at branching nodes only.

### 4.3.2 Semantic Change Impact

We can now state the meaning of a node being impacted by a program change, in terms of concrete semantics of the two programs and the set \text{MAPPED}.

**Definition 4.3.2 (Impacted nodes).** Given $\text{Prog}^1, \text{Prog}^2$ and \text{MAPPED}, a node $n \in N$ is impacted if:

1. $n \not\in \text{MAPPED}$, or
Table 4.1: Predicates used for dataflow analysis.

<table>
<thead>
<tr>
<th>Predicate name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRANCHING_NODE(n)</td>
<td>if ( n ) is a branching node</td>
</tr>
<tr>
<td>CONTROL_DEPENDENT(n2, n1)</td>
<td>if ( n_2 ) is control-dependent on ( n_1 ) [FOW87]</td>
</tr>
<tr>
<td>CALLSITE(n, f, g)</td>
<td>if ( \text{StmtAt}(n) ) is a call to ( f ) within a caller ( g ).</td>
</tr>
<tr>
<td>IN_FORMAL(x, i, f)</td>
<td>if ( x ) is the ( i )-th input formal of ( f )</td>
</tr>
<tr>
<td>OUT_FORMAL(e, i, f)</td>
<td>if the expression ( e ) is the ( i )-th actual argument to a call to ( f ) at a callsite.</td>
</tr>
<tr>
<td>IN_ACTUAL(e, i, f, n)</td>
<td>if the expression ( e ) is the ( i )-th output formal of ( f ) at a callsite.</td>
</tr>
<tr>
<td>OUT_ACTUAL(r, i, f, n)</td>
<td>if the variable ( r ) receives the ( i )-th output formal to a call to ( f ) at a callsite.</td>
</tr>
</tbody>
</table>

Intuitively, we conservatively treat any unmapped node as impacted. A mapped node \( n \) is not impacted if the sequence of values of variables in \( RVars(n) \) is identical for any two execution traces \( \tau^1 \) (in \( Prog^1 \)) and \( \tau^2 \) (in \( Prog^2 \)) starting from a common input store \( \theta \) to \( \text{main} \).

To get an intuition of our definition, when a statement reads different values in one program version than in the other, the behavior of that statement can change, e.g., a previously unseen division by zero or null pointer exception may emerge; if the values read by a statement are the same in both program versions the behavior of that statement cannot change. Note that for our low-level language, the \( RVars(n) \) of a statement often includes the state of the heap and address being written to. For example, the C# statement \( x.length = y \) gets translated to \( n : \text{Length} := \text{update}(\text{Length}, x, y) \), (where \( \text{Length} \) is an array variable representing the state of \( \text{length} \) field/attribute in all objects) with \( RVars(n) = \{ \text{Length}, x, y \} \).

The non-impacted nodes are useful for regression testing. Consider a deterministic (regression) test \( t \) that executes \( \text{main} \) with an argument \( \theta \). We can statically guarantee that one cannot distinguish the set of values computed at a non-impacted node \( n \) when \( t \) is executed on either \( Prog^1 \) or \( Prog^2 \). This has application in both selecting relevant regression tests for a software change [YH12, GEM15, RH97] and identifying opportunities for test-suite augmentation [PDEP08, PYRK11].

### 4.3.3 Dataflow-based Change Impact Analysis

In this section, we describe *Dataflow-based Change Impact Analysis (DCIA)*, a *change semantics unaware* static analysis to provide a conservative estimate of the set of impacted nodes. The static analysis is an interprocedural dataflow analysis [RHS95] that starts with a program \( Prog^i \) \((i \in 1,2)\) and a conservative estimate of the syntactically-changed nodes, nodes not in \( \text{MAPPED} \), and returns an upper bound on the set of (a) impacted nodes, (b) impacted variables, and (c) output variables whose summary may have changed.
We assume that each procedure is transformed into a Static Single Assignment (SSA) form [FOW87], where a variable (instance) is assigned at exactly one program node. The SSA form allows us to use a flow-insensitive analysis within a procedure without sacrificing flow-sensitivity. We extend Vars to include the SSA-renamed instances of existing variables. Due to the SSA form and the lack of global variables, a variable \( x \) uniquely identifies the enclosing procedure and the node where the variable is written to.

Table 4.1 defines a few predicates that are used in the inference rules. Most predicates are straightforward; for CONTROLDEPENDENT\((n_2, n_1)\), a node \( n_2 \) is control-dependent on node \( n_1 \) iff (i) there exists a path from \( n_1 \) to \( n_2 \) s.t. every node in the path other than \( n_1 \) and \( n_2 \) is post-dominated by \( n_2 \), and (ii) \( n_1 \) is not post-dominated by \( n_2 \) [FOW87].

\[
\begin{align*}
\text{DEPENDS-ENTRY} & \quad x \in \text{Vars} \\
& \quad \text{DEPENDSONVAR}(x, x, f) \\
\text{DEPENDS-WRITE} & \quad x \in \text{RVars}(n) \\
& \quad y \in \text{WVars}(n) \\
& \quad n \in N_f \\
& \quad \text{DEPENDSONVAR}(y, x, f) \\
\text{DEPENDS-TRANSITIVE} & \quad \text{DEPENDSONVAR}(y, x, f) \quad \text{DEPENDSONVAR}(x, z, f) \\
& \quad \text{DEPENDSONVAR}(y, z, f) \\
\text{DEPENDS-NODE} & \quad x \in \text{WVars}(n) \\
& \quad n \in N_f \\
& \quad \text{DEPENDSONVAR}(y, x, f) \\
& \quad \text{DEPENDSONODE}(y, n, f) \\
\text{CONTROL-DEPENDS} & \quad \text{BRANCHINGNODE}(n_1) \\
& \quad x \in \text{RVars}(n_1) \\
& \quad \text{CONTROLDEPENDENT}(n_2, n_1) \\
& \quad y \in \text{WVars}(n_2) \\
& \quad \text{DEPENDSONVAR}(y, x, f) \\
\text{SUMMARY-DEPENDS} & \quad \text{CALLSITE}(n, f, g) \\
& \quad \text{OUTACTUAL}(r, i, f, n) \\
& \quad \text{OUTFORMAL}(y, i, f) \\
& \quad \text{INFORMAL}(x, j, f) \\
& \quad \text{DEPENDSONVAR}(y, x, f) \\
& \quad \text{INACTUAL}(e, j, f, n) \\
& \quad w \in \text{RVars}(e) \\
& \quad \text{DEPENDSONVAR}(r, w, g)
\end{align*}
\]

Figure 4.2: Inference rules for computing \( \text{DEPENDSONVAR} \) and \( \text{DEPENDSONODE} \). The input is a program \( \text{Prog} \).

Figure 4.2 describes a set of inference rules to compute two relations \( \text{DEPENDSONVAR} \) and \( \text{DEPENDSONODE} \). For a pair of variables \( x, y \in \text{Vars} \) such that \( y \) is either data- or control-dependent on \( x \), then \( \text{DEPENDSONVAR}(y, x, f) \) holds. Similarly, a node \( n \in N_f \) and a variable \( y \) such that \( y \) is data or control dependent on a variable \( x \) that is updated at \( n \), \( \text{DEPENDSONODE}(y, n, f) \) holds. An inference rule (e.g. \( \text{DEPENDS-NODE} \)) lists a set of antecedents (above the line) and the consequent (below the line). Applying an inference rule results in growing the relation in the consequent (e.g. \( \text{DEPENDSONODE} \)) to satisfy the consequent. The inference rules are applied repeatedly until a fix-point is reached.

Most of the inference rules are straightforward encoding of program data and control flow, we only explain the last two rules. The rule \( \text{CONTROL-DEPENDS} \) expresses that if \( n_1 \) is a
branching node, whose condition depends on \( x \) and \( y \) is written in a control-dependent node \( n_2 \), then \( y \) depends on \( x \). The rule \textbf{SUMMARY-DEPENDS} captures the dependency of an actual return variable \( r \) on \( w \) in a caller \( g \) where \( w \) indirectly flows to \( r \) through a procedure call to \( f \). For this callsite, the \( i \)-th output formal \( y \) (that is assigned to \( r \)) is dependent on the \( j \)-th input formal \( x \), which in turn is assigned the actual \( e \) at the callsite.

Figure 4.3: Inference rules for dataflow based change impact analysis. The highlighted antecedents are relevant for change-semantics aware analysis.

Finally, Figure 4.3 describes a set of inference rules to compute the set of nodes that are impacted in either program. For this section, we will ignore the highlighted antecedents (they become relevant in Section Section 4.4). The rules take as input a program (either \( \text{Prog}^1 \) or \( \text{Prog}^2 \)), the set of mapped nodes \( \text{MAPPED} \) and precomputed relations \( \text{DEPENDEDSITE} \) and \( \text{DEPENDEDSITE} \) for the particular program. It computes the relations \( \text{IMPACTEDNODE}, \text{IMPACTEDVAR} \) and \( \text{IMPACTEDSUMM} \) that represent an upper bound on the set of impacted nodes, impacted variables and impacted variable summaries, respectively. Most rules are self-
explanatory; we describe the main rules relevant to the interprocedural reasoning. Note that we do not have a special rule for control-flow impact, since \textsc{DependsOnVar} already captures control flow dependency.

For an output formal \( y \in \text{Out}_f \), the summary (input-output relation) may change either when (i) \( y \) depends on a variable updated at an unmapped node \( n \in N_f \) (expressed by \textsc{Impact-Summary}), or (ii) \( y \) depends on a variable \( w \) that stores the output formal \( x \) of \( g \) at a callsite in node \( n \), and the summary of \( x \) has changed in \( g \) (expressed by \textsc{Impact-Summary-Prop}).

The \textsc{Call-Impact} rule says that an input formal \( x \) in \( f \) can be impacted if the corresponding actual argument \( e \) at a callsite is impacted. The \textsc{Return-Impact} and \textsc{Summary-Impact} rules enumerate the cases when the return value in \( r \) at a callsite to \( f \) (in \( g \)) can be impacted. \textsc{Return-Impact} considers the case when the variable summary for the corresponding output formal \( y \) is impacted. \textsc{Summary-Impact} considers the case when the actual argument expression \( e \) (passed for the formal \( x \) of \( f \)) is impacted, and \( y \) depends on the value of \( x \) in \( f \). Note that we do not have a rule that impacts a callsite actual return value if the corresponding output formal is impacted — this will make the analysis context insensitive as impacts from a single callsite will impact all callsites.

The algorithm \textsc{DCIA} does the following:

1. Takes as input \( \text{Prog}^1, \text{Prog}^2 \) and \text{MAPPED}.
2. Applies the inference rules in Figure 4.3 on \( \text{Prog}^i \) to generate \text{ImpactedNode}^i, \text{ImpactedVar}^i, \text{ImpactedSumm}^i.
3. Returns the tuple \((\bigcup_i \text{ImpactedNode}^i, \bigcup_i \text{ImpactedVar}^i, \bigcup_i \text{ImpactedSumm}^i)\).

The following theorem states the soundness of the dataflow analysis \textsc{DCIA}.

\textbf{Theorem 4.1} (Soundness). Given \( \text{Prog}^1, \text{Prog}^2 \in \text{Programs} \) and \text{MAPPED} \subseteq N, \( (a) \) \text{DCIA} terminates, and \( (b) \) for any \( n \notin \text{ImpactedNode}, n \) is not an impacted node with respect to MAPPED (according to Definition 4.3.2).

\subsection{4.4 Incorporating Change Semantics}

In this section, we make the \textsc{DCIA} algorithm change-semantics aware. In other words, the analysis takes into account also the exact semantics of the change, in addition to the set of nodes \text{MAPPED} that may have been syntactically changed. We inject the change-semantics by leveraging equivalence relationships between variables and procedure summaries in the two programs \( \text{Prog}^1 \) and \( \text{Prog}^2 \).

Let us define the following semantic equivalences for a variable over \( \text{Prog}^1 \) and \( \text{Prog}^2 \).

\textbf{Definition 4.4.1} (\textsc{PreEquiv}(x, f)). \textsc{PreEquiv}(x, f) holds for an input formal \( x \in \text{In}_f \) if for all stores \( \theta \), and for every pair of traces \( \tau^1 \vdash \tau^1_{\text{main}}(\theta) \) and \( \tau^2 \vdash \tau^2_{\text{main}}(\theta) \) \( (\tau^1|_{\text{in}})\}_{x} = (\tau^2|_{\text{in}})\}_{x} \).

Intuitively, \textsc{PreEquiv}(x, f) holds for an input formal \( x \) of \( f \) if any two executions starting from the same input \( \theta \) to \text{main} call \( f \) with the same sequence of values of \( x \).
Let us define $\text{Deps}(y)$ as the set of variables $x$ such that $\text{DEPENDS}\text{ONVAR}(y, x, f)$ in either $\text{Prog}^1$ or $\text{Prog}^2$. For two stores $\theta_1$ and $\theta_2$ defined over same set of variables, we denote $\theta_1 =_{\text{Vars}_1} \theta_2$ to mean $\theta_1(x) = \theta_2(x)$ for every $x \in \text{Vars}_1$.

**Definition 4.4.2 (SUMMARY\text{EQUIV}(y, f))**. $\text{SUMMARY}\text{EQUIV}(y, f)$ holds for an output formal $y \in \text{Out}_j$ if $(\theta_1, \theta_2) \in \Omega_j$ in $\text{Prog}^i$ and $\theta_1 =_{\text{Deps}(y)} \theta_3$, then $(\theta_3, \theta_4) \in \Omega_j$ is in $\text{Prog}^j (j \neq i)$ and $\theta_4(y) = \theta_4(y)$.

Intuitively, if the versions of $f$ are executed from stores $\theta_1$ and $\theta_3$ where $\theta_1 =_{\text{Deps}(y)} \theta_3$, then either both procedures do not terminate, or the value of $y$ after executing $f$ is identical on exit.

Figure 4.3 with the highlighted parts provides a refinement to the dataflow analysis to incorporate change semantics. In addition to the MAPPED, the algorithm now takes as input pre-computed relations $\text{PRE}\text{EQUIV}$ and $\text{SUMMARY}\text{EQUIV}$. In this section, we assume an oracle that provides these relations; we provide one implementation later (Section 4.5). The highlighted facts strengthen the antecedent of a rule and prevent it from being applicable in some contexts. For example, the strengthened CALL-IMPACT prevents an input formal $x$ from being impacted if $\text{PRE}\text{EQUIV}(x, f)$ holds. Similarly, the strengthened IMPACT-SUMMARY prevents a summary for $y$ from impact if we know that $\text{SUMMARY}\text{EQUIV}(y, f)$ holds. The strengthened SUMMARY-IMPACT is now applicable only when either (i) the formal $x$ does not satisfy $\text{PRE}\text{EQUIV}$ or (ii) the summary for $y$ does not satisfy $\text{SUMMARY}\text{EQUIV}$. Note that we only strengthen the interprocedural impact propagation rules since we do not assume any mapping of callsites, or arbitrary expressions in the procedure bodies, for the two versions.

We denote the new change-semantics aware algorithm as Semantic Dataflow-based Changed Impact Analysis (SEM-DCIA).

**Theorem 4.2 (Soundness)**. Given $\text{Prog}^1, \text{Prog}^2 \in \text{Programs}$, MAPPED, $\text{PRE}\text{EQUIV}$, and $\text{SUMMARY}\text{EQUIV}$, (i) SEM-DCIA terminates, and (ii) for any $n \notin \text{IMPACTED}\text{NODE}$, $n$ is not an impacted node with respect to MAPPED(according to Definition 4.3.2).

### 4.4.1 Anytime Algorithm

The SEM-DCIA algorithm assumes an oracle to compute the $\text{PRE}\text{EQUIV}$ and $\text{SUMMARY}\text{EQUIV}$ relations. Computing such equalities typically require constructing product of the two programs $\text{Prog}^1$ and $\text{Prog}^2$ and performing invariant inference over the product program [LMSH13]. Such invariant inference algorithms typically have high complexity and therefore it is wise to apply them prudently. In this section, we make a simple observation that allows us to interleave SEM-DCIA and inference of $\text{PRE}\text{EQUIV}$ and $\text{SUMMARY}\text{EQUIV}$ in a single framework.

To exploit the change semantics, it is often useful to apply differential reasoning only in the vicinity of actual syntactic changes.

Consider the example in Figure 4.4 to make the intuition clear. Applying DCIA will result in impacting all the nodes in the program as follows. The modified call node for $f_1$ in $\text{main}$ is not mapped, which impacts input formal $x$ of $f_1$. This in turn impacts the call to $f_2$ and so on. On the other hand, we can determine manually that $\text{PRE}\text{EQUIV}$ and $\text{SUMMARY}\text{EQUIV}$ hold for each of the procedures because the change does not escape the statement.
We observe that for this example it suffices to infer the equivalences for `main` by hiding the rest of the programs from analysis. Analyzing `main` in the two programs with the knowledge that no additional callsites of `main` and `f_1` exist in either program, one can still determine that \( \text{PREEQUIV}(x, f_1) \) holds. This information can be fed to `SEM-DCIA` which will prune the impact for the input parameter of `f_1` which will prune the remaining impacts when performing a pure dataflow analysis. Thus, we obtain a precise change impact analysis by applying the equivalence inference only on a small subset of procedures in the program.

Algorithm 4.1 `SEM-DCIA-ANYTIME`

Require: \( \text{Prog}^1, \text{Prog}^2 \in \text{Programs} \)

Require: \( \text{Procs}^\Delta \subseteq \text{Procs} \)

Ensure: \( \text{impNds} \subseteq N \)

1: \( k \leftarrow 0 \)
2: \( \text{EQ} \leftarrow (\emptyset, \emptyset) \)
3: \( (\text{impNds}, \text{impVars}, \text{impSumms}) \leftarrow \text{SEM-DCIA}(\text{Prog}^1, \text{Prog}^2, \text{MAPPED}, \text{EQ}) \)
4: \( \text{Procs}' \leftarrow \text{Procs}^\Delta \)
5: while \( \text{Procs}' \subset \text{Procs} \) do
6: \( \text{prEQ} \leftarrow \{ (x, f) \mid x \in \text{In}_f \text{ and } x \notin \text{impVars} \} \)
7: \( \text{smEQ} \leftarrow \{ (x, f) \mid x \in \text{Out}_f \text{ and } (x, f) \notin \text{impSumms} \} \)
8: \( \text{EQ} \leftarrow \text{EQ} + (\text{prEQ}, \text{smEQ}) \)
9: \( \text{Procs}' \leftarrow \text{ProcsWithin}(\text{Procs}^\Delta, \text{Prog}^1, \text{Prog}^2, k) \)
10: \( \text{Prog}^1_k \leftarrow \text{HideProcs}(\text{Prog}^1, \text{Procs} \setminus \text{Procs}') \)
11: \( \text{Prog}^2_k \leftarrow \text{HideProcs}(\text{Prog}^2, \text{Procs} \setminus \text{Procs}') \)
12: \( \text{EQ} \leftarrow \text{InferEquivs}(\text{Prog}^1_k, \text{Prog}^2_k, \text{EQ}) \)
13: \( (\text{impNds}, \text{impVars}, \text{impSumms}) \leftarrow \text{SEM-DCIA}(\text{Prog}^1, \text{Prog}^2, \text{MAPPED}, \text{EQ}) \)
14: \( k++ \)
15: end while
16: return \( \text{impNds} \)

Algorithm 4.1 (`SEM-DCIA-ANYTIME`) provides an anytime algorithm that performs the
introduction. The algorithm takes as an additional input $\text{Procs}^\Delta$, the set of syntactically changed procedures. It outputs a set of nodes $\text{impNds}$ that overapproximates the set of impacted nodes. We term the algorithm anytime [DB88, ZR95] because the algorithm can be stopped at any time after the first call to $\text{SEM-DCIA}$ to obtain a conservative bound for the impacted nodes.

The algorithm starts with invoking $\text{SEM-DCIA}$ on the two programs with an empty set of equivalences in $\text{EQ}$ (line 3); this is identical to calling $\text{DCIA}$. The return values provide a conservative measure on impacted variables, nodes and summaries respectively (Theorem 4.1). The algorithm implements a loop (line 5) where it increases the frontier of procedures $\text{Procs}'$ around $\text{Procs}^\Delta$ that are analyzed for inferring equivalences in $\text{InferEquivs}$ (line 12). Lines 6 and 7 construct equivalences from the provably non-impacted variables and summaries. These equivalences are added to $\text{EQ}$ in line 8. $\text{ProcsWithin}$ returns all procedures that can reach or be reached from $\text{Procs}^\Delta$ within a call stack of depth $k$; $k$ is incremented with each iteration of the loop. $\text{HideProcs}$ hides all procedures outside $\text{Procs}'$; it only retains the knowledge of whether any procedure $f \in \text{Procs}'$ has additional call sites outside $\text{Procs}'$. $\text{InferEquivs}$ is invoked with a set of equivalences in $\text{EQ}$ on the smaller programs $\text{Prog}_i^k$. The final call to $\text{SEM-DCIA}$ is used to compute the more refined set of impacted variables, nodes and summaries based on the equivalences discovered from $\text{InferEquivs}$. The loop terminates when $\text{Procs}'$ consists of the entire program; at this point $\text{InferEquivs}$ has already looked at the entire program and no new equivalences will be discovered in line 12.

Let us denote $\text{SEM-DCIA}_k$ as an instantiation of the algorithm $\text{SEM-DCIA-ANYTIME}$ that terminated after the loop is executed exactly $k + 1$ times. We also denote $\text{SEM-DCIA}_\infty$ if the loop terminates normally after $\text{Procs}'$ equals $\text{Procs}$.

**Theorem 4.3** (Soundness). Given $\text{Prog}_1^1, \text{Prog}_2^2 \in \text{Programs, MAPPED, and Procs}^\Delta$, if $\text{SEM-DCIA}_k$ terminates then for any $n \notin \text{impNds}$, $n$ is not an impacted node with respect to $\text{MAPPED}$ (according to Definition 4.3.2).

It is worth highlighting that small bounds for $k$ may not always suffice to achieve precise results, as illustrated by the example in Figure 4.5. For this example $\text{Procs}^\Delta = \{\text{main}\}$; The set $\text{Procs}'$ grows as follows across iterations: $\{\text{main}\}$, $\{f_{2n}, \text{main}, f_1\}$, $\{f_{2n-1}, f_{2n}, \text{main}, f_1, f_2\}$, and so on. It is not hard to see that until $\text{Procs}' = \text{Procs}$, $\text{InferEquivs}$ will not be able to infer that the $\text{PREEQUIV}(x, f_1)$ holds.

### 4.5 Inferring Equalities

In this section, we will describe one particular implementation of $\text{PREEQUIV}$ and $\text{SUMMARYEQUIV}$ using a product construction in $\text{SYMDIFF}$ [LMSH13]. We choose the product construction in $\text{SYMDIFF}$ because it allows for precise interprocedural differential reasoning over the product of two programs; most other product constructions are limited to single procedures with loops. We start by briefly summarizing the product construction (detailed description is outside the scope of this work) and the inference of equivalences over the product. We next identify restrictions under which these equivalences serve as valid $\text{PREEQUIV}$ and $\text{SUMMARYEQUIV}$ and augment the product construction to ensure these restrictions.
4.5.1 Product Program Construction

```plaintext
proc f₁×₂(x¹, x²) : (r¹, r²)
{
  //-- initialize call sequences ---
  var callHist¹, callHist² := ∅, ∅;
  //-- Start of f¹ body
...
  call r¹ := g¹(π¹); ///f¹ ∩ RVars(π¹) = ∅
  callHist¹ += (n₁, g¹, r¹); // record call
...
  //-- End of f¹ body
  //-- Start of f² body
...
  call r² := h²(π²); ///f² ∩ RVars(π²) = ∅
  callHist² += (n₂, g², r²); // record call
...
  //-- End of f² body
  //-- constrain calls
  foreach (var g in Procs) {
    foreach ((n₁, g, r¹, y¹) in callHist¹) {
      foreach ((n₂, g, r², y²) in callHist²) {
        //replay inputs
        call σ¹, σ² := g¹×₂(π¹, π²);
        //constrain outputs
        assume (σ¹ == y¹ ∧ σ² == y²);
      }
    }
  }
  assert callHist¹ ==procs callHist²;
}
proc main¹×₂(x¹, x²)
free requires x¹ == x²
{ ... }
```

Figure 4.6: Product procedures for inferring equivalences [LMSH13].

Given Prog¹ and Prog², SYMDIFF generates a product program Prog¹×₂ that defines a procedure f¹×₂ for every f ∈ Procs. Loops are already assumed to be present a tail-recursive procedures in Procs. Figure 4.6 shows the high-level structure of a product procedure; we will ignore the highlighted statement for now. The input and output formals of f¹×₂ is a concatenation of the inputs and outputs of f¹ and f², suitably renamed. The body of f¹×₂ consists of 3 parts. (1) Line 4 initializes two variables callHist¹ that capture the sequence of callsites in a trace through the body of f¹. (2) Lines 5..10 and lines 11.16 contain the bodies of f¹ and f² respectively. Any callsite in either f¹ (underlined) is instrumented to track (a) the arguments (b) returns and (c) callsite (e.g. line 8). (3) Finally, lines 18..27 iterate through all pair of callsites in a path in f¹ and f² with the same callee (say g), and invokes the product procedure g¹×₂ with corresponding arguments (line 22) and constrains the outputs to be outputs from the respective callsites (line 24).

The product procedure main¹×₂ that is the entry procedure for Prog¹×₂ has a precondition
to constrain the input arguments $x^1$ and $x^2$ to start in equal states.

For a procedure $f$ with inputs $x$ and outputs $r$, a boolean expression $\varphi(x) \Rightarrow \psi(x,r)$ is a summary for $f$ if for any $(\theta_1, \theta_2) \in \Omega_f$, if $\theta_1$ satisfies $\varphi$, then $\theta_2$ satisfies $\psi$. The following theorem relates $f^{1\times2}$ with $f^1$ and $f^2$.

**Theorem 4.4 ([LMSH13])**. If $\varphi(x^1, x^2) \Rightarrow \psi(x^1, x^2, r^1, r^2)$ is a summary for a product procedure $f^{1\times2}$, then for any $(\theta_1, \theta_2) \in \Omega^1_f$ and $(\theta_3, \theta_4) \in \Omega^2_f$, if $\theta_1 \oplus \theta_3$ satisfies $\varphi$ then $\theta_2 \oplus \theta_4$ satisfies $\psi$.

### 4.5.2 Invariant inference

For the product program $Prog^{1\times2}$, one can leverage any of the (single program) invariant generation techniques based on abstract-interpretation [CC77], or interpolants [McM04] to infer preconditions, postconditions (including two-state postconditions) on $f^{1\times2}$. Such invariants are relational or differential in that they are over the state of two programs $Prog^1$ and $Prog^2$. Equivalences are special classes of such relational invariants and therefore we leverage the invariant generation over $Prog^{1\times2}$ to infer $Prequiv$ and $SummEquiv$. In the next few paragraphs, we identify conditions under which such relational invariants can yield our desired equivalences.

For an output formal $r \in Out_f$, let $SummaryEq(r^1, r^2)$ be the predicate $\left( \land_{x \in In_f \text{ and } x \in Deps(r)} x^1 \equiv x^2 \right) \Rightarrow r^1 \equiv r^2$.

**Theorem 4.5.** For a product program $Prog^{1\times2}$ and a procedure $f^{1\times2}$ and output parameter $r \in Out_f$, if $SummaryEq(r^1, r^2)$ is a summary of $f^{1\times2}$, then $SummEquiv(r,f)$ holds, provided all executions in $f^1$ and $f^2$ are terminating.

**Theorem 4.6.** For a product program $Prog^{1\times2}$ and a procedure $f^{1\times2}$ and input parameter $x \in In_f$, if $x^1 \equiv x^2$ is a precondition of $f^{1\times2}$, then $Prequiv(x,f)$ holds, provided:

1. All executions in $Prog^1$ and $Prog^2$ are terminating,

2. For any procedure $h \in Procs$, and for any pair of traces $\tau^1_h(\theta)$ and $\tau^2_h(\theta)$ starting in $\theta$, the sequences of procedures calls are identical.

```c
void main()
{
    f(1, 2);
}
```

```c
int f(int x, int y)
{
    int r = x + 2;
    f(x, y);
    return r;
}
```

Figure 4.7: Example illustrating side conditions for inference.

84
Example Figure 4.7 illustrates the need for the side conditions when working with the product program. The change in \( f \) adds an infinite loop in \( \text{Prog}^2 \) by calling itself. Here \( \text{Deps}(r) = \{ x \} \) for \( f \) in both programs. Since \( f \) does not terminate, neither does \( f^1 \times 2 \). Hence, \( (x^1 == x^1) \Rightarrow r^1 == r^2 \) is a vacuous summary of \( f^1 \times 2 \), although \( \text{SUMMARYEQUIV}(y, f) \) does not hold. On the other hand, since \( f \) does not invoke itself in \( \text{Prog}^1 \), no calls to \( f^1 \times 2 \) are made from within \( f^1 \times 2 \). This means that \( x^1 == x^2 \) is a precondition for \( f^1 \times 2 \), although \( \text{PREEQUIV}(x, f) \) does not hold.

Termination is a desirable property for a program and can be verified separately (e.g. establishing ranking functions for loops and recursive procedures. In fact, we can relax the need for every execution to terminate by only requiring equitermination (an input \( \theta \) leads to a terminating execution in \( \text{Prog}^1 \) if an only if \( \text{Prog}^2 \) terminates on the same input) [HKLR13]; this is an area of future work. For this work, we (unsoundly) assume that the changes do not introduce or fix non-termination.

Satisfying the second side condition for Theorem 4.6, unfortunately, cannot be reduced to a standard program verification task. We informally describe a modification to the product construction (Figure 4.6) that provides a sufficient check for ensuring that call sequences from a common input are identical. We reuse the history variables \( \text{callHist}^i \) that capture the call sequences along any path in either procedures (lines 8 and 18). We add an assertion (highlighted in line 0) that checks if call sequence through \( f^1 \) is identical to \( f^2 \) (assuming equitermination); the equality \( == \text{procs} \) compares the two sequences after projecting on to the second element (procedure name) of each tuple. If every product procedure satisfies this assertion, starting from only the assumption that arguments to \( \text{main} \) are identical, then one can ensure the second side condition for Theorem 4.6. Our implementation currently does not handle this modification.

4.6 Implementation and Evaluation

4.6.1 Implementation

We implemented our \( \text{SEM-DCIA} \) analysis for C programs, but all our analysis works on the intermediate verification language Boogie [BCD+05]. We leverage SMACK [RE14] to convert LLVM bytecode to Boogie programs. We modified \text{SMDiff} to add candidates for inferring summaries, and take as input equalities inferred from \text{DCIA}. For our initial implementation, we leveraged the output of \text{diff} on procedure bodies in C to infer the source of changes \text{MAPPED}. However, this leads to several sources of unsoundness from changes in macros, data structures, control-flow changes etc.; we therefore conservatively consider all nodes in a changed procedure as source of impacts. Although this can grossly overapproximate the initial source of impact, the use of equivalences in \( \text{SEM-DCIA} \) allows us to prune the spurious impacts from escaping the syntactically-changed procedures; this approach avoids the unsoundness of performing \text{diff} in an heuristic, best-effort fashion and hence preserves our soundness guarantees.

4.6.2 Evaluation

In this section we demonstrate the effectiveness of our approach on real program changes and standard benchmark programs. We show that our semantic based analysis, \( \text{SEM-DCIA} \), is useful
<table>
<thead>
<tr>
<th>Project Name</th>
<th># Version Pairs</th>
<th>SLOC min</th>
<th>SLOC max</th>
<th>LOC Changed min</th>
<th>LOC Changed max</th>
</tr>
</thead>
<tbody>
<tr>
<td>flingfd</td>
<td>2</td>
<td>142</td>
<td>146</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>histo</td>
<td>8</td>
<td>617</td>
<td>624</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>mdp</td>
<td>91</td>
<td>135</td>
<td>1616</td>
<td>1</td>
<td>402</td>
</tr>
<tr>
<td>theft</td>
<td>2</td>
<td>1672</td>
<td>1838</td>
<td>2</td>
<td>328</td>
</tr>
<tr>
<td>tinyvm</td>
<td>61</td>
<td>425</td>
<td>903</td>
<td>1</td>
<td>328</td>
</tr>
<tr>
<td>print_tokens</td>
<td>5</td>
<td>478</td>
<td>480</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>10</td>
<td>397</td>
<td>402</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>replace</td>
<td>32</td>
<td>509</td>
<td>516</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>schedule</td>
<td>9</td>
<td>290</td>
<td>294</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>space</td>
<td>38</td>
<td>6180</td>
<td>6205</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>tcas</td>
<td>41</td>
<td>136</td>
<td>140</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>tot_info</td>
<td>23</td>
<td>346</td>
<td>347</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.2: Summary of projects used as evaluation subjects

in reducing the size of the impacted set compared to the non-semantic analysis, DCIA.

We applied our analysis over 164 development changes consisting of refactorings, feature additions, buggy changes, and bug fixes from 5 projects from the GitHub repository. The projects, number of versions used, their size in source lines of code (SLOC), and corresponding changes sizes (in number of C source lines changed) are summarized in Table 4.2. Our subjects are all applications written in C such as a virtual machine program (tinyvm), a histogram creator (histo), a markdown presentation tool (mdp), a file-descriptor management library (flingfd) and a test-generation library (theft). These applications are open-source projects on GitHub and the changes made are both maintainence tasks and active feature additions that developers made independently without considering or being aware of our tool. In addition, we also include 6 standard benchmark programs widely used by prior research on regression testing [HFGO94]. These benchmarks consist of 158 manually introduced changes to represent non-trivial and hard to detect bugs. While these are changes that create semantic impact, we demonstrate that even under these circumstances, our technology can still be useful in pruning the set of impacted statements. Our projects are sized between 142 lines of source code and 6205 (SLOC). The changes in our projects vary in size between very small changes, consisting of single line changes and large ones, consisting of over 400 lines (most of our changes are on the small end of this spectrum).

For our experiments, we first compare SEM-DCIA against DCIA to study the impact of adding change-semantics to the impact analysis (Section 4.6.3). Next, we evaluate the cost-precision tradeoff of the anytime algorithm SEM-DCIA-ANYTIME (Section 4.6.4). Finally, we present interesting anecdotes discovered while applying our tool (Section 4.6.5).
Table 4.3 shows for all projects the results of varying the bound on $k$ (we used a timeout of 1 hour). Table 4.4 shows the first 4 levels for SEM-DCIA.

### 4.6.3 Change-Semantic Aware Analysis

Table 4.3 shows the results of running our SEM-DCIA analysis on the each of our subjects. For each change, we measure the number of lines reported as impacted by dataflow analysis (columns DCIA Impact) and also by SEM-DCIA (columns SEM-DCIA$_\infty$). The columns SEM-DCIA$_i$ denote various bounds for the SEM-DCIA-ANYTIME and will be discussed in Section 4.6.4. We report for each project the minimum and maximum number of impacted lines (min, max), and for the SEM-DCIA analysis we report also the average reduction of the size of the impacted set. Note that SEM-DCIA analysis will always report a subset of the set reported by the non-semantic analysis. We also report the average analysis time in seconds for the non-semantic analysis and for the SEM-DCIA analysis.

Our evaluation shows that on average, the change-aware analysis reduces the size of the impacted set by 35%. The overhead of performing full semantic analysis on the entire program is on median 19x, ranging between 3x and 67x. While the semantic analysis results at $\infty$ level represent the most precise analysis our technique achieves, it is quite expensive, and it even times-out for our largest program (e.g. space). For example in the theft project the reduction achieved by SEM-DCIA$_\infty$ is 77% but with a 64x overhead. This indicates the need for an incremental analysis, whose results can be obtained faster.

### 4.6.4 Incremental Analysis

Table 4.3 shows for all projects the results of varying the bound on $k$ for the SEM-DCIA-ANYTIME. The first iteration SEM-DCIA$_0$ corresponds to semantically analyzing only the syntactically-changed procedures; the second iteration SEM-DCIA$_1$ corresponds to analyzing the procedures at distance at most one from the syntactically changed procedures. The results show that even SEM-DCIA$_0$ provides benefits, pruning the impacted set by 22% on average. The overhead is reduced compared to the full analysis (9x). Similarly, results for SEM-DCIA$_1$ shows the analysis is effective. For the case of theft we notice that the precision improves from 61% (SEM-DCIA$_0$) to 77% (SEM-DCIA$_\infty$), at the cost of overhead increase from 8x to 64x.

In general, we foresee that the anytime analysis is most useful for cases where it is prohibitive to run the full algorithm because of time constraints. This is best illustrated for the case of space (we used a timeout of 1 hour). Table 4.4 shows the first 4 levels for space (2 more
<table>
<thead>
<tr>
<th>Analysis</th>
<th>Min</th>
<th>Max</th>
<th>Reduction</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCIA</td>
<td>20</td>
<td>2851</td>
<td>n.a.</td>
<td>59.45</td>
</tr>
<tr>
<td>SEM-DCIA(_0)</td>
<td>14</td>
<td>2816</td>
<td>31.87%</td>
<td>798.96</td>
</tr>
<tr>
<td>SEM-DCIA(_1)</td>
<td>14</td>
<td>2816</td>
<td>36.71%</td>
<td>895.14</td>
</tr>
<tr>
<td>SEM-DCIA(_2)</td>
<td>14</td>
<td>2816</td>
<td>40.56%</td>
<td>1300.43</td>
</tr>
<tr>
<td>SEM-DCIA(_3)</td>
<td>14</td>
<td>2808</td>
<td>43.96%</td>
<td>1900.03</td>
</tr>
<tr>
<td>SEM-DCIA(_\infty)</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>timeout</td>
</tr>
</tbody>
</table>

Table 4.4: Analysis results for space

iteration beyond the ones in Table 4.3); performing the analysis incrementally is still valuable even upto \( k = 3 \); the first iteration already provides big benefits on top of the non-semantic analysis, while the following iterations display a smooth improvement with each iteration. We believe this highlights the benefits of our anytime algorithm, giving the user control over the tradeoff between precision and analysis-time. We believe this is a crucial knob in making analyses that use verification usable.

Figure 4.8: Comparing results for different analysis depths for the tinyvm project
void draw_histogram(int data[], int len) {
    int xbarw = 5;
    while (y--) {
        term_move_to(x * 5 + xpad + 3,
                     y - 1 + h + ypad);
    }
}

Figure 4.9: Change illustrating an extract constant to variable in histo commit c723a4

if ((float) ++htab->num_nodes / htab->size > 0.7)
+if((float)(++htab->num_nodes / htab->size) > HtabLoadFactor)
.

Figure 4.10: Change illustrating a failed refactoring in tinyvm commit 716936

To gain better insight on how the depth of our analysis, i.e., the number of iterations run, influences the impacted set, we highlight how the impacted set is distributed across the program with respect to the syntactically changed procedure. We show that the impacted statements are distributed across the program and how more iterations are able to prune impacted statements that are farther away from the syntactic change. For illustration purposes, in Figure 4.8 we show the distribution of the impacted set, across the program procedures, by distance from the syntactically changed procedure; the x axis shows the distance from the syntactically changed procedure and the y axis shows the number of nodes impacted at the corresponding distance. We show results for the first, second, and third iteration of the analysis. Blue lines show information for non-semantic analysis, while the red one shows results for the semantic analysis. For example, for the first iteration, corresponding to depth 0, there are 29 lines impacted at distance 0 from the change, i.e., in the syntactically changed procedure itself; 32 lines at distance 1 by the syntactic analysis and similarly for the semantic analysis. The three figures convey the intuitive idea that as more iterations are performed, impacts from farther away from the change are pruned.

4.6.5 Anecdotes

A detailed study of the results indicates that the improvement in precision in SEM-DCIA comes from two fronts. First, it compensates for the price we paid for soundness by considering entire procedures as source of impact. The semantic analysis, although does not reduce the impacted set for the Procs^Δ but reduces the impacts from callers and callees transitively. Second, the reduction in impact happens from many actual refactorings that a pure dataflow analysis cannot take into account. We show a few interesting patterns that were discovered while applying the tool (for brevity we only describe the change briefly).

Variable Extraction: Figure 4.9 shows a refactoring to extract a constant to a variable. A
non-semantic technique will create impacts in term_move_to through the first argument, since it will not be able to find that the value flowing into the first argument is the same in both versions and in all executions. Our SEM-DCIA technique will be able to prove the mutual precondition necessary to show the equality in both versions, and hence cut impacts that would propagate through the first argument.

Non-Refactoring: Figure 4.10 shows an example of a failed refactoring The developer tries to refactor the code to use a variable rather than a constant and adds some parentheses as a matter of style to the expression. Unfortunately, parenthesesizing the entire expression leaves the division operating over integers and performing truncation, and only converts the final result to a float. This was not the behavior before the change, and this change actually breaks the program, introducing a bug, which was fixed by another developer 5 months later (by removing the parentheses). SEM-DCIA finds that this change impacts many other portions of code and hence warns the developer that the change is not an actual refactoring.

Loop Refactoring: Figure 4.11 shows a change from a while loop to a for loop. Input-output equivalence checking would not prevent the impact of the argument c to the callee inside the loop (nor would dataflow analysis). Remember that we extract loops as tail recursive procedures.

Control-Flow Equivalence: Figure 4.12 shows a change to replace a goto with return statements. This is a change in the project tinyvm. The goto statements were all redirecting control-flow to a return statement, so the developer replaced the goto with the target return statement.
4.7 Related Work

Our work is closely related to work on change-impact analysis, regression verification, regression test generation, and relational analysis.

**Change impact analysis:** Change Impact Analysis has been widely explored in static and dynamic program analysis context [Leh11, CS15, RST+04, RT01, LR03]. Previous work informally characterized the set of impacted statements as the set of statements that are data- or control- dependent on a changed statement; past techniques ignored the change-semantics. We differ from this line of work by being the first to formally define the notion of impact at the level of statements; further, we are the first to propose a general framework for incorporating and exploiting change-semantics when determining the impact of a change.

Most previous works perform the analysis at a coarse-grain level (classes and types) to retain soundness of analysis [AOH05, AOH04, OAH03, LP14] which can result in coarse results. JDiff [AOH04] addresses some of the challenges of performing both a diff and computing a mapping between two programs in the context of Java object-oriented programs. Both Gama [OAH03] and the Execute-After-based techniques [AOH05] use dynamic analysis to compute a dynamically impacted set. These techniques conceptually aim to prune the overly-conservative dataflow based techniques by using dynamic information. We share the goal of removing imprecision in CIA, but use program semantics and change semantics to reason over all execution traces whereas the dynamic techniques only provide guarantees over the execution.

**Regression verification:** SymDiff [LHKR12, LMSH13] is a platform that provides general facilities for differential reasoning about programs. Verification-Modulo-Versions [LLFB14] aims to perform semantic baselining to filter out static analysis alarms that are not related to the change, but require a set of assertions. Regression verification [GS09, PDEP08] and its implementations [LHKR12] aim at proving summary equivalence interprocedurally, but does not help with the CIA directly. The work by Bakes et al. [BPRT13] performs equivalence checking by computing summaries using symbolic execution. The approach is non-modular (does not summarize callees), bounded (unrolls loops and recursion), and does not improve the underlying change-impact analysis. The technique leverages CIA to avoid performing equivalence checking on non-impacted procedures (computed by standard dataflow analysis). These approaches are useful for equivalence-preserving changes; when the changes are non-equivalent they do not provide meaningful help to users/downstream tools. Our approach aims to work with equivalent and non-equivalent changes, providing help to users/downstream tools even in such changes.

**Regression:** Person et al. using change directed symbolic execution to generate regression tests [PYRK11]. Our technique can be used to prune the space of statements for which regression tests need to be generated. In addition, there is research on relational verification using product construction [PY13, BCK11, BCK13, Ben04], but most approaches are not automated and do not consider changes across procedure calls.

**Relational Analysis:** Research has explored relational analyses, aiming to find and prove properties over two programs [BCK11, CKMR12, BCK13] such as equivalence, non-interference, smoothness of programs, approximate bounds for loop perforation. These technique
allow for expressive properties to be proven, but are not automatic. These techniques also do not handle procedure calls. In contrast, we design an automatic technique that aims to reason precisely about two program versions and their relation in terms of impacted statements.

4.8 Related Work

Our work is closely related to work on change-impact analysis, regression verification, regression test generation, and relational analysis.

Change impact analysis: Change Impact Analysis has been widely explored in static and dynamic program analysis context [Leh11, CS15, RST+04, RT01, LR03]. Previous work informally characterized the set of impacted statements as the set of statements that are data- or control- dependent on a changed statement; past techniques ignored the change-semantics. We differ from this line of work by being the first to formally define the notion of impact at the level of statements; further, we are the first to propose a general framework for incorporating and exploiting change-semantics when determining the impact of a change.

Most previous works perform the analysis at a coarse-grain level (classes and types) to retain soundness of analysis [AOH05, AOH04, OAH03, LP14] which can result in coarse results. JDiff [AOH04] addresses some of the challenges of performing both a diff and computing a mapping between two programs in the context of Java object-oriented programs. Both Gama [OAH03] and the Execute-After-based techniques [AOH05] use dynamic analysis to compute a dynamically impacted set. These techniques conceptually aim to prune the overly-conservative dataflow based techniques by using dynamic information. We share the goal of removing imprecision in CIA, but use program semantics and change semantics to reason over all execution traces whereas the dynamic techniques only provide guarantees over the execution.

Regression verification: SymDiff [LHKR12, LMSH13] is a platform that provides general facilities for differential reasoning about programs. Verification-Modulo-Versions [LLFB14] aims to perform semantic baselining to filter out static analysis alarms that are not related to the change, but require a set of assertions. Regression verification [GS09, PDEP08] and its implementations [LHKR12] aim at proving summary equivalence interprocedurally, but does not help with the CIA directly. The work by Bakes et al. [BPRT13] performs equivalence checking by computing summaries using symbolic execution. The approach is non-modular (does not summarize callees), bounded (unrolls loops and recursion), and does not improve the underlying change-impact analysis. The technique leverages CIA to avoid performing equivalence checking on non-impacted procedures (computed by standard dataflow analysis). These approaches are useful for equivalence-preserving changes; when the changes are non-equivalent they do not provide meaningful help to users/downstream tools. Our approach aims to work with equivalent and non-equivalent changes, providing help to users/downstream tools even in such changes.

Regression: Person et al. using change directed symbolic execution to generate regression tests [PYRK11]. Our technique can be used to prune the space of statements for which regression tests need to be generated. In addition, there is research on relational verification using product construction [PY13, BCK11, BCK13, Ben04], but most approaches are not automated and do not consider changes across procedure calls.
Relational Analysis: Research has explored relational analyses, aiming to find and prove properties over two programs [BCK11, CKMR12, BCK13] such as equivalence, non-interference, smoothness of programs, approximate bounds for loop perforation. These techniques allow for expressive properties to be proven, but are not automatic. These techniques also do not handle procedure calls. In contrast, we design an automatic technique that aims to reason precisely about two program versions and their relation in terms of impacted statements.
Chapter 5

Statistical Similarity of Binaries

In this final chapter, we focus on differential analysis of compiled binaries. Previous chapters mainly applied to source code, however an important aspect of program similarity is measuring similarity of assembly code procedures. The compilation process, which can be applied using different compilers, yields vast syntactic difference for the exact same procedure. Providing techniques for finding the same procedure, or a similar one, after being compiled by different compilers is challenging.

Given a query procedure in stripped (without any symbol or identifier names) binary form \( q \), compiled by an unknown compiler, we aim to find the same (target) procedure \( t \) (originating from the same source code) or a patched version of the procedure, after it has been compiled by another different and unknown compiler. We produce a numeric score which measures the similarity of each pair of procedures in the context of an entire corpus. As in previous chapters, the measure of similarity declines as the procedures differ in functionality.

Finding similarity in binary procedures has applications in vulnerability detection, where a vulnerable procedure could have been ported and compiled using an unknown compiler, and embedded in some executable within an organization making it vulnerable to malicious attacks. Binary similarity has further applications in detecting software plagiarism and intellectual property theft, as identifying a stolen piece of software after being compiled and stripped is extremely challenging.

5.1 Overview

In this section, we illustrate our approach informally using an example.

Given a query procedure from a stripped binary, our goal is to find similar procedures in other stripped binaries. To simplify presentation, we illustrate our approach on two code snippets instead of full procedures. Consider the assembly code of Figure 5.1(a). This snippet is taken from a version of OpenSSL, which is vulnerable to the Heartbleed bug [Hea]. Our goal is to find similar vulnerable code snippets in our code-base. We would like to define a notion of similarity that can find matches even when the code has been modified, or compiled with a different compiler vendors and versions. For example, the code of Figure 5.1(a) was compiled using gcc v4.9, and the code of Figure 5.1(b), originating from the same source, was compiled using icc v15.0.1. We would like our approach to find the similarity of these two snippets despite their noticeable syntactic difference.
We focus on similarity rather than equivalence, as we would like our approach to apply to code that may have been patched. Towards that end, we compute a similarity score that captures similarity between (small) partial computations performed by each procedure.

The main idea of our approach is to decompose the code to smaller fragments, for which similarity is easy to compute, and use statistical reasoning over fragment similarity to establish the global similarity between code snippets. Towards that end, we have to answer three design questions:

- What is the best way to decompose the code snippets?
- How should the decomposed fragments be compared?
- How can fragment similarity be lifted into find snippet similarity?

### Decomposition into strands
We decompose a procedure into fragments which are feasible to compare. In this work, we use *strands*—partial dependence chains, as the basic unit. Figure 5.2 shows two strands obtained from the code snippets of Figure 5.1. For now, ignore the assume and assert operations added around the strands. The strands in the figure have been transformed to an Intermediate Verification Language (IVL), by employing tools from [BJAS11, RE14]. The IVL abstracts away from specific assembly instructions, while maintaining the semantics of the assembly code. A fresh temporary variable is created for every intermediate value computed throughout the execution. The flow of data between registers is always through these temporaries. Furthermore, the IVL always uses the full 64-bit representation of registers (e.g. rax and not eax) and represents operations on part of the register using temporaries and truncation (e.g. mov rbx, al will be v1 = truncate(rax,8); rbx = v1;). The strands in the figure have been aligned such that similar instructions of the two strands appear side by side. This alignment is only for the purpose of presentation, and our comparison is based on semantically comparing

---

(a) gcc v.4.9 -O3
(b) icc v.15.0.1 -O3

**Figure 5.1:** Heartbleed vulnerability code snippets.

96
\begin{align*}
    \text{assume } r12_q &= rbx_t \\
    v1_q &= r12_q \\
    v2_q &= v1_q + 1 \\
    v3_q &= \text{int}_\text{to}_\text{ptr}(v2_q) \\
    r14_q &= v3_q \\
    v4_q &= 18h \\
    rsi_q &= v4_q \\
    v5_q &= v4_q + v3_q \\
    rax_q &= v5_q \\
    v1_t &= 13h \\
    r9_t &= v1_t \\
    v2_t &= rbx_t \\
    v3_t &= v2_t + v1_t \\
    v4_t &= \text{int}_\text{to}_\text{ptr}(v3_t) \\
    r13_t &= v4_t \\
    v5_t &= v1_t + 5 \\
    rsi_t &= v5_t \\
    v6_t &= v5_t + v4_t \\
    rax_t &= v6_t \\
\end{align*}

\begin{align*}
    \text{assert } v1_q &= v2_t, v2_q &= v3_t, v3_q &= v4_t, r14_q &= r13_t, \\
    v4_q &= v5_t, rsi_q &= rsi_t, v5_q &= v6_t, rax_q &= rax_t
\end{align*}

Figure 5.2: Semantically similar strands.

\begin{align*}
    v1_q &= r14_q \\
    v2_q &= v1_q + 1 \\
    v3_q &= \text{xor}(v2_q, v1_q) \\
    v4_q &= \text{and}(v3_q, v2_q) \\
    v5_q &= (v4_q < 0) \\
    \text{FLAGS}[^{OF}]_q &= v5_q \\
    v1_t &= r14, \\
    v2_t &= v1_t + 16 \\
    v3_t &= \text{xor}(v2_t, v1_t) \\
    v4_t &= \text{and}(v3_t, v2_t) \\
    v5_t &= (v4_t < 0) \\
    \text{FLAGS}[^{OF}]_t &= v5_t
\end{align*}

Figure 5.3: Syntactically similar but semantically different strands.

the strands. We added \( q \) and \( t \) postfixes to the strands’ variables to separate the name spaces of variables in the two strands and specify one as the query and the other as the target. This allows us to create a joint program that combines the variables from the two strands and makes assumptions and assertions about their equality.

**Comparing a pair of strands** To compare a pair of strands, we create a joint program that combines them, but has a separate name space for the variables of each strand. We then explore the space of equality assumptions on the different inputs, and check how these assumptions affect the equality assertions on the outputs. For example, one choice for assumptions and assertions is shown in Figure 5.2. Technically, given a pair of strands, we perform the following steps: (i) add equality assumptions over inputs of the two strands, (ii) add assertions that check the equality of all output variables (where output variables also include temporaries), and (iii) check the assertions using a program verifier and count how many variables are equivalent. Choosing which variables to pair when assuming and asserting equality is solved by searching the space of possible pairs. The choice of the strand as a small unit of comparison (with a relatively small number of variables), along with verifier based optimizations (described in Section 5.4.5), greatly reduce the search space, making the use of a verifier feasible.

**Match Probability:** We define an asymmetric similarity measure between a query strand, \( s_q \), and a target strand, \( s_t \), as the percentage of variables from \( s_q \) that have an equivalent counterpart in \( s_t \). We denote this measure by \( VCP(s_q, s_t) \). We later use this measure as a basis for computing the probability \( Pr(s_q|s_t) \) that a strand \( s_q \) is input-output equivalent to a strand \( s_t \) (see Section 5.2.3).
For example, taking $s_q$ to be the strand on the left-hand side of Figure 5.2, and $s_t$ to be the strand on the right-hand side, $VCP(s_q, s_t) = 1$, because all 8 variables from the left-hand side have an equivalent variable on the right-hand side. However, in the other direction, $VCP(s_t, s_q) = 8/10$. We note that no previous approach is able to produce such a matching, as the equivalent values are computed using different instructions.

In contrast to Figure 5.2, the strands of Figure 5.3 (taken from our corpus as specified in Section 5.4.1) are very similar syntactically (differ in just one character), but greatly differ semantically. Syntactic approaches would typically classify such pairs as matching, leading to a high rate of false positives. Our semantic comparison identifies that these two strands are different, despite their significant syntactic overlap, and yields $VCP = 1/6$, expressing the vast difference.

Local Evidence of Similarity: Modeling strand similarity using probability allows us to express other notions of similarity in a natural manner. For example, given a target procedure $t$ and a query strand $s_q$, we can capture how well $s_q$ can be matched in $t$ by computing the maximal probability of $Pr(s_q | s_t)$ over any possible strand $s_t$ in $t$.

We further define the probability, $Pr(s_q | H_0)$, of finding a matching strand for $s_q$ at random (where $H_0$ represent all possible strands). The significance of finding a match for $s_q$ in $t$ can be then defined as:

$$LES(s_q | t) = \log \max_{s_t \in t} \frac{Pr(s_q | s_t)}{Pr(s_q | H_0)}.$$  

$LES$ provides a measure of the significance of the matching of $s_q$ with $t$ by comparing it to the matching of $s_q$ with the random source $H_0$. It is important to measure the significance of a match, because many strand matches may be due to common strands introduced by the compiler (e.g., prolog/epilog), and therefore not significant in determining a match between procedures.

Lifting strand similarity into procedure similarity We say that two procedures are similar if one can be composed using (significantly similar) parts of the other. Given a query procedure $q$ and a target procedure $t$, the $LES(s_q | t)$ measure indicates how significant the match of $s_q$ is in $t$. We can therefore define global evidence of similarity (GES) between the procedures $q$ and $t$ by summing $LES(s_q | t)$ over all strands $s_q$ in $q$ (as further explained in Section 5.2.1).

The local and global similarity evidence allow us to lift semantic similarity computed between individual strands into a statistical notion of similarity between procedures.

Key Aspects of our approach

- Similarity by composition: we decompose procedures into strands, semantically compare strands, and lift strand similarity into procedure similarity using statistical techniques.

- Using strand similarity allows us to establish similarity between procedures even when the procedures are not equivalent, but still contain statistically significant similar strands.

- By performing semantic comparison between strands, we are able to find similarity across different compilers versions and vendors (without knowledge of the specifics of the compilers). In Section 5.4 we compare our approach to previous work and show the importance of semantic comparison.


98
5.2 Strand-Based Similarity by Composition

In this section, we describe the technical details of our approach. In Section 5.2.1, we describe the steps for generating statistical procedure similarity. In Section 5.2.2 we illustrate how we decompose procedures into the single-path units of execution we call strands (step 1). We defer the exact details of computing similarity between strands (step 2) to Section 5.3, and assume that such a similarity measure is given. In Section 5.2.3 we define the likelihood and statistical significance of a strand, allowing us to quantify similarity using these probabilistic tools (step 3). In Section 5.2.4, we describe how whole procedure similarity is computed from statistical similarity of strands, using global and local evidence scores (step 4).

5.2.1 Similarity By Composition

We determine that two procedures are likely to be similar if non-trivial strands from one can be used to compose the other, allowing for some (compiler or patch related) transformation. We decompose the procedure to basic blocks, and then further split each block into strands using a slicing technique, as described in Section 5.2.2. We check for strand similarity with the aid of a program verifier, as further shown in Section 5.3. This gives us the flexibility to determine similarity between two strands, i.e., that they are (partially) input-output equivalent, even if they produce their result using different instructions, register allocation or program ordering.

We then define a Global Evidence Score, $GES(q|t)$, representing the likelihood that a query procedure $q$ is similar to a target procedure $t$. This global score is based on the composition of multiple Local Evidence Scores (denoted $LES(s_q|t)$) representing the sum of likelihoods that each strand $s_q \in q$ has a similar strand $s_t \in t$:

$$GES(q|t) = \sum_{s_q \in q} LES(s_q|t) = \sum_{s_q \in q} \log \frac{\max_{s_t \in t} Pr(s_q|s_t)}{Pr(s_q|H_0)}. \tag{5.1}$$

The right-hand side of Equation 5.1 shows that the LES for strand $s_q$ is calculated using the ratio between two factors: (i) the probability, $Pr(s_q|s_t)$, that $s_q$ is semantically similar to one of the strands $s_t \in t$, where $s_t$ is the strand that produces the highest probability (i.e., is most similar to $s_q$), and (ii) $Pr(s_q|H_0)$, the probability that this strand matches a random source. In our context, this means that the strand is commonly found in the corpus binaries and does not uniquely identify the query procedure. This is further described in Section 5.2.3.

5.2.2 Procedure Decomposition to Strands

We use a standard control flow graph (CFG) representation for procedures. We decompose the procedure by applying slicing [Wei81] on the basic-block level. A strand is the set of instructions from a block that are required to compute a certain variable’s value (backward slice from the variable). Each block is sliced until all variables are covered. As we handle each basic block separately, the inputs for a block are variables (registers and memory locations) used before they are defined in the block. Figure 5.2 is an example of a pair of strands extracted from the blocks in Figure 5.1.

Strands as partial program dependence graphs (PDGs) Strands are a practical compromise
over enumerating all paths in the PDG [FOW87]. All strands are obtained by decomposing the
PDG at block boundaries. This means that strands only contain data dependencies, as control
dependencies exist only over block boundaries, while severed data dependencies (e.g., values
created outside the block) are marked as such and used in the comparison process (these are
the inputs, as explained later in Algorithm 5.1). This approach of decomposing graphs while
making use of loose edges to improve performance is similar to the “extended graphlets” in
[KMA13], yet there the edge contains less information as it is not connected to a certain variable.
Decomposing at block level yielded precise results for most of our benchmarks. However, using
longer paths can be advantageous when handling small procedures, where breaking at block
level results in a small number of short blocks that can be easily matched with various targets.
Section 5.5.6 further discusses benchmarks for which a the small number of blocks yields low
accuracy for Esh.

Algorithm 5.1 Extract Strands from a Basic Block

Require: $b$ - An array of instructions for a basic-block
Ensure: strands - $b$’s strands, along with their inputs

```
unusedInsts ← \{1, 2, ..., |b|\}
strands ← []
while unusedInsts ≠ ∅ do
    maxUsed ← max(unusedInsts)
    unusedInsts \= maxUsed
    newStrand ← [b[maxUsed]]
    varsRefed ← Ref(b[maxUsed])
    varsDefed ← Def(b[maxUsed])
    for i ← (maxUsed − 1)..0 do
        needed ← Def(b[i]) \ varsRefed
        if needed ≠ ∅ then
            newStrand += b[i]
            varsRefed ⊇ Ref(b[i])
            varsDefed ⊇ needed
            unusedInsts \= i
        end if
    end for
    inputs ← varsRefed \ varsDefed
    strands += (newStrand, inputs)
end while
```

Algorithm 5.1 uses standard machinery to extract strands from a basic block, and uses the
standard notions of $\text{Def}$ and $\text{Ref}$ for the sets of variables defined and referenced (respectively) in
a given instruction.

The process starts by putting all instructions in $\text{unusedInsts}$, and will end only when this list
is empty, i.e., when every instruction is marked as having been used in at least one extracted
strand. The creation of a new strand begins by taking the last non-used instruction, as well as
initializing the list of variables referenced in the strand – $\text{varsRefed}$, and the variables defined

100
in the strand – \textit{varsDefed}. Next, all of the previous instructions in the basic-block are iterated \textit{backwards}, adding any instruction which defines a variable referenced in the strand so far (is in \textit{varsRefed}) and updating \textit{varsRefed} and \textit{varsDefed} with every instruction added. When the \texttt{for} loop is finished the new strand is complete, as every instruction needed to calculate all of the variables defined inside the basic-block is present. This does not include the \textit{inputs}, which are any variables used in the calculation and not defined in the basic-block. Note that the backward iteration is crucial for minimizing the number of strands.

5.2.3 Statistical Evidence

Given two sets of strands obtained from two procedures for comparison, we assume a mechanism for computing a similarity measure between two strands based on the proportion of output variables they agree on when given equivalent inputs. We denote this similarity metric by $VCP$, and defer its formal definition and computation to Section 5.3. In this section, we assume that the $VCP$ between two strands, $s_q$ and $s_t$, is given, and transform it into a probabilistic measure for strand similarity $Pr(s_q|s_t)$. We then describe how to compute $Pr(s_q|H_0)$, the probability of each strand to match a random process. Using these values we compute the likelihood-ratio, $LR(s_q)$, from which the $GES$ value is composed.

\textbf{Strand Similarity as a Probability Measure}

We denote $Pr(s_q|t)$ as the likelihood that a strand, $s_q$, from the query procedure, $q$, can be “found” in the target procedure $t$, i.e., that we can find an equivalent strand $s_t \in t$.

$$Pr(s_q|t) \triangleq \max_{s_t \in t} Pr(s_q|s_t).$$ \hspace{1cm} (5.2)

The likelihood $Pr(s_q|s_t)$ that two strands are input-output equivalent is estimated by applying a sigmoid function (denoted $g()$) over the $VCP$ of the two strands (we set the sigmoid midpoint to be $x_0 = 0.5$ as $VCP(s_q, s_t) \in [0, 1]$):

$$Pr(s_q|s_t) \triangleq g(VCP(s_q, s_t)) = 1/(1 + e^{-k(VCP(s_q, s_t) - 0.5)}).$$ \hspace{1cm} (5.3)

The use of the logistic function allows us to produce a probabilistic measure of similarity, where $Pr(s_q|t)$ is approximately 1 when $VCP(s_q, s_t) = 1$ and nears 0 when $VCP(s_q, s_t) = 0$. We experimented with different values to find the optimal value for the steepness of the sigmoid curve parameter, $k$, and found $k = 10.0$ to be a good value.

Our use of the sigmoid function is similar to its application in logistic regression algorithms for classification problems [KK10]. The hypothesis $h_\theta(x)$ is set to be the sigmoid function of the original hypothesis $\theta^T x$, resulting in the hypothesis being a probability distribution, $h_\theta(x) \triangleq Pr(y = 1|x; \theta) = g(\theta^T x)$, which reflects the likelihood of a positive classification ($y = 1$) given a sample $x$. This correlates to $Pr(s_q|s_t)$ representing the likelihood that $s_q$ and $s_t$ are a positive match for performing the same calculation.
The Statistical Significance of a Strand

In order to find procedure similarity in binaries, we require that non-trivial strands of code be matched across these binaries. This stands in contrast to Equation 5.3, where smaller pieces of code will receive a high likelihood score, since they perform trivial functionality that can be matched with many strands. Thus a query strand need not only be similar to the target, but also have a low probability to occur at random. For this we introduce the Likelihood Ratio measure:

\[ LR(s_q|t) = \frac{Pr(s_q|t)}{Pr(s_q|H_0)}. \] (5.4)

This measure represents the ratio between the probability of finding a semantic equivalent of \( s_q \) in \( s_t \) vs. the probability of finding a semantic equivalent at random (from the random process \( H_0 \)). \( Pr(s_q|H_0) \) in fact measures the statistical insignificance of a strand, where a higher probability means low significance. We estimate the random hypothesis \( H_0 \) by averaging the value of \( Pr(s_q|s_t) \) over all targets (as it needs to be computed either way), i.e.,

\[ Pr(s_q|H_0) = \frac{\sum_{s_t \in T} Pr(s_q|s_t)}{|T|} \]

where \( T \) is the set of all target strands for all targets in the corpus.

5.2.4 Local and Global Evidence Scores

After presenting the decomposition of procedures \( q \) and \( t \) to strands, and defining the likelihood-ratio for each strand \( s_q \) and target \( t \), we can define the Local Evidence Score as the log of the likelihood-ratio:

\[ LES(s_q|t) = \log LR(s_q|t) = \log Pr(s_q|t) - \log Pr(s_q|H_0). \] (5.5)

This local score reflects the level of confidence for \( s_q \) to have a non-trivial semantic equivalent strand in \( t \). The global score \( GES(q|t) \) is simply a summation (Equation 5.1) of all \( LES(s_q|t) \) values after decomposing \( q \) to strands, which reflects the level of confidence that \( q \) can be composed from non-trivial parts from \( t \) and is in fact similar to it.

5.3 Semantic Strand Similarity

In the previous section, we assumed a procedure that computes semantic strand similarity. In this section, we provide such a procedure using a program verifier. Given two strands to compare, the challenge is to define a quantitative measure of similarity when the strands are not equivalent. We first provide the formal definitions on which the semantics of strand similarity are based, and then show how we compute strand similarity using a program verifier.

5.3.1 Similarity Semantics

Preliminaries We use standard semantics definitions: A program state \( \sigma \) is a pair \((l, values)\), mapping the set of program variables to their concrete value \( values : Var \to Val \), at a certain program location \( l \in Loc \). The set of all possible states of a program \( P \) is denoted by \( \Sigma_P \). A program trace \( \pi \in \Sigma^*_P \) is a sequence of states \( \langle \sigma_0, ..., \sigma_n \rangle \) describing a single execution of the program. The set of all possible traces for a program is denoted by \([P]\). We also define \( first : \Sigma^*_P \to \Sigma_P \) and \( last : \Sigma^*_P \to \Sigma_P \), which return the first and last state in a trace.
respectively. A strand $s \in P$ is therefore a set of traces, $s \subseteq [P]$ – the set of all traces generated by all possible runs of $s$, considering all possible assignments to $s$ inputs. We will use this abstraction to further define strand equivalence and VCP.

**Variable correspondence** A variable correspondence between two states, $\sigma_1$ and $\sigma_2$, denoted $\gamma : \text{Var}_1 \rightarrow \text{Var}_2$, is a (partial) function from the variables in $\sigma_1$ to the variables in $\sigma_2$. Note that several variables can be mapped to a single variable in $\text{Var}_2$. $\Gamma(P_1,P_2)$ denotes the set of all variable correspondences for the pair of programs $(P_1,P_2)$. This matching marks the variables as candidates for input-output equivalence to be proven by the verifier.

**State, trace equivalence** Given two states and a correspondence $\gamma$, if $\forall(v_1,v_2) \in \gamma : \sigma_1(v_1) = \sigma_2(v_2)$, then we say that these states are equivalent with respect to $\gamma$, and denote them $\sigma_1 \equiv_{\gamma} \sigma_2$.

Given two traces and a correspondence $\gamma$ between their last states, if $\text{last}(\pi_1) \equiv_{\gamma} \text{last}(\pi_2)$, then we say that these traces are equivalent with respect to $\gamma$, and denote them $\pi_1 \equiv_{\gamma} \pi_2$.

**Definition 5.3.1 (Strand equivalence).** Given two strands (remembering that each strand has inputs as defined in Section 5.2.2 and denoted $\text{inputs}(s)$) and a correspondence $\gamma$, we say that these strands are equivalent with respect to $\gamma$, denoted $s_1 \equiv_{\gamma} s_2$ if: (i) every input from $s_1$ is matched with some input from $s_2$ under $\gamma$, and (ii) every pair of traces $(\pi_1, \pi_2) \in (s_1, s_2)$ that agree on inputs ($\forall(i_1,i_2) \in (\gamma \cap (\text{inputs}(s_1) \times \text{inputs}(s_2))) : \text{first}(\pi_1)(i_1) = \text{first}(\pi_2)(i_2)$) is equivalent $\pi_1 \equiv_{\gamma} \pi_2$. This expresses input-output equivalence.

**Definition 5.3.2 (State, trace variable containment proportion).** We define the VCP between a query state $\sigma_q$ and a target state $\sigma_t$ as the proportion of matched values in $\sigma_q$, denoted $VCP(\sigma_q, \sigma_t) \triangleq \frac{|\gamma_{\max}|}{|\sigma_q|}$, where $\gamma_{\max}$ is the maximal variable correspondence (in size) for which the two states are equivalent, i.e., $\sigma_q \equiv_{\gamma_{\max}} \sigma_t$, considering all possible gammas. We define the VCP between two traces, $VCP(\pi_q, \pi_t)$, as $VCP(\text{last}(\pi_q), \text{last}(\pi_t))$.

For instance, given $\text{values}_q = \{x \mapsto 3,y \mapsto 4\}$, $\text{values}_t = \{a \mapsto 4\}$, the maximal correspondence is therefore $\gamma_{\max} = \{y \mapsto a\}$ as it matches the most possible variables. Therefore $VCP(\sigma_q, \sigma_t) = \frac{1}{2}$. We note that it is possible for several maximal correspondences to exist, and in these cases we simply pick one of the said candidates.

**Definition 5.3.3 (Strand VCP).** We define the VCP between two strands as the proportion of matched variables in the $\gamma$ that induces the maximal containment proportion over all pairs of traces, as follows:

$$VCP(s_q,s_t) \triangleq \frac{\max\{|\gamma|\|\forall(\pi_q, \pi_t) \in (s_q, s_t) : \pi_q \equiv_{\gamma} \pi_t\}}{|\text{Var}(s_q)|}.$$

An important observation regarding the VCP is that it can produce a high matching score for potentially unrelated pieces of code, for instance if two strands perform the same calculation but one ends by assigning 0 to all outputs, or if the result of the computation is used for different purposes. We did not observe this to be the case, as (i) compiler optimizations will eliminate
such cases where a computation is not used, and (ii) even if the code is used for different purposes – it may still suggest similarity, if for example a portion of the query procedure was embedded in the target.

5.3.2 Encoding Similarity as a Program Verifier Query

Next we show how we compute a strand’s $VCP$ (Definition 5.3.3) by encoding input-output equivalence, along with procedure semantics as a program verifier query. The query consists of three parts, including (i) assuming input equivalence over the inputs in the variable correspondence ($\gamma$), (ii) expressing query and target strand semantics by sequentially composing their instructions, and (iii) checking for variable equivalence, over all possible traces, by adding equality assertions to be checked by the program verifier.

**Program verifiers** For the purposes of this chapter, we describe a program verifier simply as a function denoted $Solve : (\text{Proc}, \text{Assertion}) \rightarrow (\text{Assertion} \rightarrow \{\text{True}, \text{False}\})$, that given a procedure $p \in \text{Proc}$ with inputs $i_1, \ldots, i_n$ and a set of assertion statements $\Phi \subseteq \text{Assertion}$, is able to determine which of the assertions in $\Phi$ hold, for any execution of $p$, under all possible values for $i_1, \ldots, i_n$. The assertions in $\Phi$ are mapped to a specific location in $p$ and specify a property (a formula in first-order logic (FOL)) over $p$’s variables that evaluates to $\text{True}$ or $\text{False}$ according to variable value. $Solve$ will label an assertion as $\text{True}$ if for all variable values under all input values, the assertion holds. Verifiers usually extend the program syntax with an $\text{assume}$ statement, which allows the user to specify a formula at desired program locations. The purpose of this formula is to instruct the verifier to assume the formula to always be true at the location, and try to prove the assertions encountered using all the assumptions encountered in the verification pass. In this work, we used the Boogie program verifier. ([BCD+05] provides the inner workings of the verifier.) To allow the use of the verifier, we lifted assembly code into a non-branching subset of the (C-like) Boogie IVL (translation details described in Section 5.4.1). The details of BoogieIVL are thoroughly described in [Lei].

**Procedure calls** We treat procedure calls as uninterpreted functions while computing similarity because, (i) an inter-procedural approach would considerably limit scalability, as the verifier would need to reason over the entire call tree of the procedure (this could be unbounded for recursive calls), and (ii) we observed that the semantics of calling a procedure is sufficiently captured in the code leading up to the call where arguments are prepared, and the trailing code where the return value is used. Calls to two different procedures that have the exact same argument preparation process and use return values in the same way would be deemed similar by our approach. We note, however, that our approach does not rely on knowing call targets, as most are omitted in stripped binaries.

**Calculating strand $VCP$ using a program verifier** Next, we describe how we encode strand similarity as a Boogie procedure. We present a simplified version of the algorithm for clarity and brevity, and further describe optimizations in Section 5.4.5. As our compositional approach alleviates the need to reason over branches, we can define the encoding assuming single-path programs. For the rest of the chapter we separate procedure variables to $Vars(p)$, denoting
Algorithm 5.2 Compute Strand VCP

Require: Query \(p^q\), Target \(p^t\) in BoogieIVL
Ensure: \(VCP(p^q, p^t)\)

\[
\text{maxVCP} \leftarrow 0 \\
\text{for } \gamma \in \Gamma(p^q, p^t) \text{ do} \\
\quad p \leftarrow \text{NewProcedure}(\text{Inputs}(p^q) \cup \text{Inputs}(p^t)) \\
\quad \text{for } (i^q, i^t) \in (\gamma \cap (\text{Inputs}(p^q) \times \text{Inputs}(p^t))) \text{ do} \\
\quad\quad p\.body\.Append(\text{assume } i^q == i^t) \\
\quad\text{end for} \\
\quad p\.body\.Append(\text{body}(p^q); p\.body) \\
\quad \text{for } (v^q, v^t) \in \left((\text{Vars}(p^q) \times \text{Vars}(p^t)) \cap \gamma\right) \text{ do} \\
\quad\quad p\.body\.Append(\text{assert } v^q == v^t) \\
\quad\text{end for} \\
\quad \text{Solve}(p) \\
\quad \text{if } p^q \equiv p^t \text{ then} \\
\quad\quad \text{maxVCP} \leftarrow \max(|\gamma|/|\text{Vars}(p^q)|, \text{maxVCP}) \\
\quad\text{end if} \\
\text{end if} \\
\text{end for}
\]

all non-input variables in the procedure, and \(\text{Inputs}(p)\), denoting only inputs. Algorithm 5.2 receives a pair of Boogie procedures \(p^q, p^t\) representing the strands \(q\) and \(t\), after renaming of variables to avoid naming collisions. It then proceeds to enumerate over all possible variable correspondences \(\gamma \in \Gamma(p^q, p^t)\), where all of \(p_q\)'s inputs are matched in compliance with Definition 5.3.1. For each correspondence, a new Boogie procedure \(p\) is created. We start building the procedure body by adding assumptions of equivalence for every pair of inputs in \(\gamma\). This is crucial for checking input-output equivalence. Next, we append the bodies of the query and target procedures sequentially, capturing both strands’ semantics. Lastly, a series of assertion statements are added, whose goal is to assert the exit state equivalence by adding an assertion for all variable pairs matched by \(\gamma\). The resulting procedure \(p\) is then given to the \(\text{Solve()}\) function, which uses the program verifier to check assertion correctness. If all the assertions were proven, the current \(VCP\) is calculated and compared against the best \(VCP\) computed so far, denoted \(\text{maxVCP}\). The higher value is picked, resulting in the maximal \(VCP\) at the end of the loop’s run.

5.4 Evaluation

Our method’s main advantage is its ability to perform cross-compiler (and cross-compiler-version) code search on binary methods, even if the original source-code was slightly patched. We will harness this ability to find vulnerable code; we demonstrate our approach by using a known (to be) vulnerable binary procedure as a query and trying to locate other similar procedures in our target database that differ only in that they are a result of compiling slightly patched code or using a different compilation tools. These similar procedures become suspects for being vulnerable as well and will be checked more thoroughly. We designed our experiments to carefully test every major aspect of the problem at hand. Finally, we will also show how our
method measures up against other prominent methods in this field.

In Section 5.4.1 we describe the details of our prototype’s implementation. In Section 5.4.2 we detail the vulnerable and non-vulnerable code packages that compose our test-bed. In Section 5.4.3 we show how our test-bed was built and how this structure enabled us to isolate the different aspects of the problem domain. In Section 5.4.4 we explain the ROC & CROC classifier evaluation tools that we used to evaluate our method’s success. In Section 5.4.5 we detail the different heuristics we employed in our prototype to improve performance without undermining our results.

5.4.1 Test-Bed Creation and Prototype Implementation

We implemented a prototype of our approach in a tool called Esh. Our prototype accepts a query procedure and database of procedures (the targets), residing in executable files, as input in binary form. Before the procedures can be compared using our method, we need to divide them (from whole executables into single procedures) and “lift” them into the BoogieIVL to enable the use of its verifier.

Lifting Assembly Code into BoogieIVL

Each binary executable was first divided to procedures using a custom IDA Pro (the Interactive DisAssembler) [IDA] Python script, which outputs a single file for every procedure. BAP (Binary Analysis Framework) [BJAS11] “lifts” the binary procedure into LLVM IR [LA04] code, which manipulates a machine state represented by global variables. An important observation, and this was shown in Figure 5.2 & Figure 5.3, is that translated code is in Single Static Assignment (SSA) form, which is crucial for an effective calculation of the VCP (Definition 5.3.3). The SMACK (Bounded Software Verifier) [RE14] translator is used to translate the LLVM IR into BoogieIVL. Finally, strands are extracted from the blocks of the procedure’s CFG.

Alongside Esh, we performed all the experiments on the prototype implementation of [DY14] called TRACY. Section 5.5.3 shows a detailed analysis of these experiments.

Our prototype was implemented with a mixture of C# (for Boogie framework interaction) and Python. The full source code & installation guide are published on Github, and the prototype was deployed on a server with four Intel Xeon E5-2670(2.60GHz) processors, 377 GiB of RAM, running Ubuntu 14.04.2 LTS.

5.4.2 Using Vulnerable Code as Queries

To make sure the experiments put our method to a real-world test scenario, we incorporated eight real vulnerable code packages in the test-bed. The specific Common Vulnerabilities and Exposures (CVEs) are detailed in Table 5.1. The rest of the target database was composed from randomly selected open-source packages from the Coreutils [Cora] package.

All code packages were compiled using the default settings, resulting in most of them being optimized using the -O2 optimization level while a few, like OpenSSL, default to -O3. All executables were compiled to the x86_64 (64-bit) architecture as default. Following this we decided to focus our system’s implementation efforts on this architecture. Note that our system can be easily expanded to support x86 (32bit) and even other chip-sets (assuming our tool-chain
provides or is extended to provide support for it). After compilation we removed all debug information from the executables. After compiling the source-code into binaries, our target corpus contained 1500 different procedures.

5.4.3 Testing Different Aspects of the Problem Separately

Many previous techniques suffered from a high rate of false positives, especially as the code corpus grew. We will show one cause for such incidents using a general example. [RMZ11] shows that the compiler produces a large amount of compiler-specific code, such as the code for its control structures, so much so that the compiler can be identified using this code. This is an example of a common pitfall for methods in the fields of binary code search: if the comparison process is not precise enough when comparing procedures compiled by a different compiler, a low similarity score can be wrongly assigned for these procedures on the basis of the generating compiler alone. This is due to the compiler-centric code taking precedence over the real semantics of the code, instead of being identified as common code and having its importance reduced in the score calculation.

In our context, the problem can be divided into three vectors: (i) different compiler versions, (ii) different compiler vendors, and (iii) source-code patches.

Different versions of the same compiler Before attempting cross-compiler matching, we first evaluated our method on binaries compiled using different versions of the same compiler. We therefore compiled the last vulnerable version of each package mentioned in Section 5.4.2 using the gcc compiler versions 4.6,8,9. Next we performed the same process with the CLang compiler versions 3.4,5, and again with the icc compiler versions 15.0.1 and 14.0.4.

Cross-compiler search Next, we evaluated our approach by searching procedures compiled across different compilers. An important aspect of this evaluation process was alternating the query used, each time selecting the query from a different compiler. This also ensured that our method is not biased towards a certain compiler. As explained in Section 5.2, our matching method is not symmetric, and so examining these different scenarios will provide evidence for the validity of this asymmetric approach.

Patched source-code The last vector we wish to explore is patching. We define a patch as any modification of source-code that changes the semantics of the procedure. The common case for this is when the procedure’s code is altered, yet changes to other procedures or data-structures can affect the semantics as well. We predict that precision will decline as the size of the patch grows and the procedures exhibit greater semantic difference.

5.4.4 Evaluating Our Method

A naive approach to evaluating a method which produces a quantitative similarity score is to try and find a “noise threshold”. This threshold transforms the quantitative method into a binary classifier by marking all pairs of procedures with a score above the threshold as a match, and the rest as a non match.

Alas, for most cases there is no clear way to compute or detect one threshold which creates a clear separation between true and false positives for all experiments. As this is also true for
our method, we will evaluate our tool by examining the results of our experiments as a ranked list, and use a measure which reflects whether the true positives are ranked at the top of that list.

**Evaluating classifiers with ROC** The receiver operating characteristic (ROC) is a standard tool in evaluation of threshold based classifiers. The classifier is scored by testing all of the possible thresholds consecutively, enabling us to treat each method as a binary classifier (producing 1 if the similarity score is above the threshold). For binary classifiers, accuracy is determined using the True Positive (TP, the samples we know are positive), True Negative (TN, the samples we know are negative), Positive (P, the samples classified as positive) and Negative (N, the samples classified as negative) as follows: \( \text{Accuracy} = \frac{TP + TN}{P + N} \). Plotting the results for all the different thresholds on the same graph yields a curve; the area under this curve (AUC) is regarded as the accuracy of the proposed classifier.

**Evaluating classifiers with CROC** Concentrated ROC [SADB10] is an improvement over ROC that addresses the problem of “early retrieval” – where the corpus size is huge and the number of true positives is low. The idea behind the CROC method is to better measure accuracy in a scenario with a low number of TPs. The method assigns a higher grade to classifiers that provide a low number of candidate matches for a query (i.e., it penalizes false positives more aggressively than ROC). This is appropriate in our setting, as manually verifying a match is a costly operation for a human expert. Moreover, software development is inherently based on re-use, so similar procedures should not appear in the same executable (so each executable will contain at most one TP).

5.4.5 Enabling the Use of Powerful Semantic Tools

**Esh Performance** Our initial experiments showed the naive use of program verifiers to be infeasible, resulting in many hours of computation for each pair of procedures. In the following subsection we describe various optimizations to our algorithm which reduced the time required for comparing a pair of procedures to roughly 3 minutes on average on an 8-core Ubuntu machine. We emphasize that our approach is embarrassingly parallelizable, as verifier queries can be performed independently, allowing performance improvements linearly to the number of computation cores.

**Algorithm 5.2 optimizations** We first presented the VCP computation algorithm in a simplified form with no optimizations. To avoid enumerating over all variable correspondences in \( \Gamma(p^q, p^t) \), in our optimized implementation we enumerated over inputs only \( (\text{Inputs}(p^q) \times \text{Inputs}(p^t)) \). Furthermore, we did not allow multiple inputs in \( p^q \) to be matched with a single input in \( p^t \) (\( \gamma \) was one-to-one) and only allowed for correspondences that matched all of \( p^q \) inputs. This reduced the number of outer loop iterations to \( \max(|I_q|, |I_t|) \). We further reduced this by maintaining typing in matches.

For each matching of the inputs, the non-input variable matching part of \( \gamma \) starts out simply as \( \text{Var}(p^q) \times \text{Var}(p^t) \), while maintaining types. We further perform a data-flow analysis to remove variable pairs that have no chance of being matched, as their calculation uses inputs that were not matched with an initial assumption (were not in \( \gamma \)). Allowing for all possible matchings
(up to typing and dataflow analysis) means that we check all possible correspondences for non-inputs at once. We do this by parsing the output of the Boogie verifier that specifies which of the equality assertions hold and which fail. Unmatched variables are removed, leaving only equivalent pairs in $\gamma$. Finally, multiple matchings for variables are removed ($\gamma$ must be a function over $q$’s variables according to Definition 5.3.3) and the $VCP$ is calculated.

**Discarding trivial strands and impractical matches** As mentioned in Section 5.2.3, trivial strands receive low $LES$ similarity scores. Thus smaller strands are less likely to generate evidence for global similarity $GES$. We therefore established a minimal threshold for the number of variables required in a strand in order that it be considered in the similarity computation (5 in our experiments), and did not produce verifier queries for strands smaller than the said threshold. We further avoided trying to match pairs of strands which vary greatly in size. We used a ratio threshold, set to 0.5 in our experiments, meaning that we will attempt to match a query strand only with target strands that have at least half the number of variables (i.e. a minimal value of $VCP = 0.5$ is required) or at most twice the number of variables (avoiding matching with “giant” strands, which are likely to be matched with many strands).

**Batching verifier queries** To further save on the cost of performing a separate verifier query for each input matching (which usually contains a small number of assertions), we batch together multiple queries. As we wanted to allow procedure knowledge gathered by the verifier to be re-used by subsequent queries, we embedded several possible $\gamma$ correspondences in the same procedure by using the non-deterministic branch mechanism in Boogie (further explained in [Lei]) to make the verifier consider assumptions and check assertions over several paths. We also batched together different strand procedures up to a threshold of 50,000 assertions per query.

### 5.5 Results

<table>
<thead>
<tr>
<th>#</th>
<th>Alias/Method</th>
<th>CVE/Stats</th>
<th>#BB</th>
<th>#Strands</th>
<th>FP</th>
<th>ROC</th>
<th>CROC</th>
<th>FP</th>
<th>ROC</th>
<th>CROC</th>
<th>FP</th>
<th>ROC</th>
<th>CROC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Heartbleed</td>
<td>2014-0160</td>
<td>15</td>
<td>92</td>
<td>107</td>
<td>.967</td>
<td>.814</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>Shellshock</td>
<td>2014-6271</td>
<td>136</td>
<td>430</td>
<td>246</td>
<td>.866</td>
<td>.552</td>
<td>3</td>
<td>.999</td>
<td>.995</td>
<td>3</td>
<td>.999</td>
<td>.996</td>
</tr>
<tr>
<td>3</td>
<td>Venom</td>
<td>2015-3456</td>
<td>13</td>
<td>67</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>Clobberin’ Time</td>
<td>2014-9295</td>
<td>41</td>
<td>233</td>
<td>351</td>
<td>.797</td>
<td>.343</td>
<td>66</td>
<td>.987</td>
<td>.924</td>
<td>19</td>
<td>.993</td>
<td>.956</td>
</tr>
<tr>
<td>5</td>
<td>Shellshock #2</td>
<td>2014-7169</td>
<td>88</td>
<td>294</td>
<td>175</td>
<td>.889</td>
<td>.541</td>
<td>40</td>
<td>.987</td>
<td>.920</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>ws-snmp</td>
<td>2011-0444</td>
<td>6</td>
<td>98</td>
<td>42</td>
<td>.981</td>
<td>.879</td>
<td>5</td>
<td>.999</td>
<td>.990</td>
<td>1</td>
<td>1.00</td>
<td>.997</td>
</tr>
<tr>
<td>7</td>
<td>wget</td>
<td>2014-4877</td>
<td>94</td>
<td>181</td>
<td>332</td>
<td>.885</td>
<td>.600</td>
<td>11</td>
<td>.998</td>
<td>.988</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>8</td>
<td>ffmpeg</td>
<td>2015-6826</td>
<td>11</td>
<td>87</td>
<td>222</td>
<td>.921</td>
<td>.659</td>
<td>97</td>
<td>.981</td>
<td>.895</td>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 5.1: The ROC, CROC and false positives (FP) for our query search experiments
5.5.1 Finding Heartbleed

To better understand our evaluation process and test-bed design, we start by walking through experiment #1 from Table 5.1. The query for this experiment was the “Heartbleed” vulnerable procedure from openssl-1.0.1f, compiled with CLang 3.5. Each bar in Figure 5.4 represents a single target procedure, and the height of the bar represents the GES similarity score (normalized) against the query. The specific compiler vendor and version were noted below the graph (on the X axis) the source package and version above it. Bars filled green represent procedures originating from the same code as the query (i.e. “Heartbleed”) but vary in compilation or source code version (the exact code version openssl-1.0.1{e,g,f} is specified over the bar itself). All unrelated procedures were left blank.

As we can see from the results in Figure 5.4, our method gives high scores to all other similar versions of the “Heartbleed” procedure, despite them being compiled using different compilers, different compiler versions or from a patched source code. A gap of 0.08 in the GES score exists between the true positives from the rest of the procedures. (0.419 for the icc 15 compiled procedure of openssl-1.0.1g “Heartbleed” vs. 0.333 for the bash 4.3 “ShellShock” procedure compiled with CLang 3.5). It is important to note that we will not try to establish a fixed threshold to evaluate the quality of these results. As mentioned, this clean separation between the true positives and the false positives is not always possible. Instead, this result and others, as shown in the following sections, are evaluated according to the produced ranking. The result in Figure 5.4 receives a \( ROC = CROC = 1.0 \) score as it puts all of the true positives in the top of the ranking.

Figure 5.4: Experiment #1 – Successfully finding 12 procedure variants of Heartbleed.
5.5.2 Decomposing Our Method into Sub-methods

When examining our method bottom up, we can divide it into three layers:

- **S-VCP**: The first layer of our method is the way we calculate the VCP between strands. Without the use of the statistical processing, we still define a similarity score as:
  \[
  \sum_{s_t \in T} \max_{s_q \in Q} (VCP(s_t, s_q)).
  \]
  This approach attempts to generalize the VCP from a pair of strands to a pair of procedures by counting the maximal number of matched variables in the *entire* procedure.

- **S-LOG**: The next layer of our approach incorporates the statistical significance of every query strand, by using local and global significance. By alternatively defining
  \[
  Pr(s_q, s_t) = VCP(s_q, s_t)
  \]
  and applying it to the LES and GES equations, we can see how our method looks without applying the sigmoid function to the VCP.

- **Esh**: Adding the use of the sigmoid function results in our full method as described in Section 5.2

To fully justify and explain each layer of our method, Table 5.1 shows the results of each sub-method compared with our full method, in terms of (i) false positives (FT), and (i) ROC & CROC (explained in Section 5.4.4). Note that we count the number of false positives as determined by a human examiner who receives the list of procedures sorted by similarity scores, and we define the number of false positives as the number of non-matching procedures the human examiner will have to test until all the true similar procedures are found. The effectiveness of a method can be measured more precisely and quickly by using CROC. We also included additional information about the number of basic-blocks and the number of strands extracted from them, and the CVE for every vulnerable procedure we attempted to find.

These results clearly show that each layer of our method increases its accuracy. Comparison of the different experiments shows that CROC & ROC scores do more than simply count the number of false positives for every threshold they compare the *rate* by which the false positive rate grows. (Informally, this may be regarded as a prediction of the number of attempts after which the human researcher will give up.) An important point is that the size of the query, in terms of the number of basic blocks or strands, does not directly correlate with easier matching.

Experiment #3 shows an interesting event, where even S-VCP gets a perfect score. Upon examination of the query procedure, we discovered that this occurs because the said procedure contains several distinct numeric values which are only matched against similar procedures. (These are used to explore a data structure used to communicate with the QEMU floppy device.)

Examining the results as a whole, we see that in more than half of the experiments the use of S-VCP, which doesn’t employ the statistical amplification of strands, results in a high number of false positives. To understand this better, we performed a thorough analysis of experiment #5. When we examine the values of \( Pr(s_q|H_0) \), which express the frequency of appearance of strand \( s_q \), we see that several strands get an unusually high score (appear more frequently). One of these strands was found to be a sequence of `push REG` instructions, which are commonplace for a procedure prologue.
5.5.3 Comparison of TRACY and Esh

<table>
<thead>
<tr>
<th>Versions</th>
<th>Cross</th>
<th>Patches</th>
<th>TRACY (Ratio-70)</th>
<th>Esh</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td></td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.6764</td>
<td>1.0000</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>0.5147</td>
<td>1.0000</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>0.4117</td>
<td>1.0000</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>0.8230</td>
<td>1.0000</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>0.3529</td>
<td>1.0000</td>
</tr>
</tbody>
</table>

Table 5.2: Comparing TRACY and Esh on different problem aspects.

Following our explanation of the different problem aspects in Section 5.4.3, we tested both tools, with a focus on experiment number one in Table 5.1. Each line in Table 5.2 represents a single experiment; the ✓ in one of the columns specifies that this specific problem aspect is applied: (i) Compiler version from the same vendor, (ii) Cross-compiler, meaning different compiler vendors, and (iii) applying patches to the code. For example, when (i) and (ii) are checked together this means that all variations of queries created using all compiler vendors and compiler versions were put in the target database.

As we can see, because TRACY was designed to handle patches, it achieves a perfect grade when dealing them. Moreover, TRACY can successfully handle the different compiler versions on their own. However, when it is used in a cross-compiler search, its accuracy begins to plummet. Furthermore, when any two problem aspects are combined, and especially when all three are addressed, the method becomes practically unusable.

5.5.4 Evaluating BinDiff

BinDiff [Bin] is a tool for comparing whole executables/libraries by matching all the procedures within these libraries. It works by performing syntactic and structural matching relying mostly on heuristics. The heuristic features over procedures which are the basis for similarity include: the number of jumps, the place of a given procedure in a call-chain, the number of basic blocks, and the name of the procedure (which is unavailable in stripped binaries). Detailed information about how this tool works can be found in [Und], along with a clear statement that BinDiff ignores the semantics of concrete assembly-level instructions.

Table 5.3 shows the results of running BinDiff on each of the procedures in Table 5.1. As BinDiff operates on whole executables/libraries, the query/target was a whole library containing the original vulnerability. We only compared one target for each query, the same library compiled with a different vendor’s compiler, and patched (for the queries where patching was evaluated). BinDiff failed to find the correct match in all experiments but two. The experiments in which BinDiff found the correct match are those where the number of blocks and branches remained the same, and is relatively small, which is consistent with [Und].

112
Table 5.3: Evaluating BinDiff

<table>
<thead>
<tr>
<th>Alias</th>
<th>Matched?</th>
<th>Similarity</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heartbleed</td>
<td>✗</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Shellshock</td>
<td>✗</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Venom</td>
<td>✗</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Clobberin’ Time</td>
<td>✗</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Shellshock #2</td>
<td>✗</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ws-snmp</td>
<td>✓</td>
<td>0.89</td>
<td>0.91</td>
</tr>
<tr>
<td>wget</td>
<td>✗</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>✓</td>
<td>0.72</td>
<td>0.79</td>
</tr>
</tbody>
</table>

5.5.5 Pairwise Comparison

Figure 5.5 shows the similarity measures produced in an all-vs-all experiment, where 40 queries were chosen at random from our corpus and compared. The result is shown in heat-map form, where the axes represent individual queries (X and Y are the same list of queries, in the same order) and each pixel’s intensity represents the similarity score $GES$ value (Equation 5.1) for the query-target pair. Queries that originate from the same procedure (but are compiled with different compilers, or patched) are coalesced. Different procedures are separated by ticks. We included at least two different compilations for each procedure. The first procedure (leftmost on the X, bottom on Y axis) is `ftp_syst()` from `wget 1.8`, queried in 6 different compilations. The second is taken from `ffmpeg 2.4.6`, queried in 7 different compilations. The rest are taken from `Coreutils 8.23`. The average $ROC$ and $CROC$ values for the experiment were 0.986 and 0.959 respectively.

Several observations can be made from Figure 5.5:

1. The graph’s diagonal represents the “ground truth” i.e., each query is matched perfectly with itself.
2. Our $GES$ measure is not symmetrical (as it is based on the asymmetrical $VCP$ metric).
3. $Esh$ provides a very clear distinction for `ffmpeg-2.4.6`’s `ff_rv34_decode_init_thread_copy()`, marked with a dashed region numbered 1, where all compiled queries of the procedure receive high $GES$ scores when compared with each other and low ones when compared with random code. In general, $Esh$ correctly matches procedures compiled with different compilers, as the pattern of “boxes” along the diagonal shows.
4. Groups of queries that originate from the same procedure generally receive similar $GES$ values (i.e. similar shade) when compared to groups of targets that originate from the same (but different from the queries’) code.

Although some procedures, such as `default_format()`, seem to perform poorly as they have high $GES$ matches with wrong targets (marked with a dashed region numbered 2), the correct
evaluation of the matching should be relative to the correct matching. For `default_format()`, the `GES` values of the correct matching is high (specified by dark pixels around the diagonal), so relative to that, the set of wrong matchings (somewhat dark pixels in the middle) becomes less significant, which is reflected by a `ROC = .993` and `CROC = .960` AUC score.

5.5.6 Limitations

In this subsection we discuss the limitations of our approach using study cases of procedures where `Esh` yields low recall and precision.

**“Trivial” procedures and Wrappers** Our approach relies on matching semantically similar, non-trivial fragments of execution. Thus, when matching a very small query (i.e. a “trivial” procedure), we are forced to rely on the statistical significance of a very small number of relatively short strands, which yields poor results. Wrappers are procedures that mostly contain calls to other procedures, and hold very little logic of their own e.g. the `exit_cleanup()` procedure in Figure 5.6. As we operate on stripped binaries, we cannot rely on identifiers like procedure names to try and match strands based on procedure calls.

**Generic procedures** The C concatenation preprocessor directive (`##`) is sometimes used as a mechanism for generics, where “template” procedures are created, that have similar structure but vary in type, or use a function pointer to provide different functionality as shown in Figure 5.7, where different string comparison procedures are created to compare creation, modification and access time of a file. As these hold the same general structure (and are specifically small), `Esh` deems the whole set of procedures similar. We do not consider these matches to be strict false positives, as the procedures are in fact similar in structure. Note that this problem only arises when these types of procedures are used as queries, as it is not clear whether the different
```c
static void exit_cleanup (void) {
    if (temphead) {
        struct cs_status cs = cs_enter ();
        cleanup ();
        cs_leave (cs);
    }
    close_stdout ();
}
```

Figure 5.6: Wrapper exit_cleanup procedure from sort.c of Coreutils 8.23

variants should be considered as true positives (they perform similar semantics) or false positives (as they might operate on different data structures).

```c
#define DEFINE_SORT_FUNCTIONS(key_name, key_cmp_func) \
    static int strcmp_##key_name (V a, V b) \
    { return key_cmp_func (a, b, strcmp); } \
...
DEFINE_SORT_FUNCTIONS (ctime, cmp_ctime)
DEFINE_SORT_FUNCTIONS (mtime, cmp_mtime)
DEFINE_SORT_FUNCTIONS (atime, cmp_atime)
```

Figure 5.7: DEFINE_SORT_FUNCTIONS macro for creating “template” procedures in ls.c of Coreutils 8.23

5.6 Related Work

In this section, we briefly describe closely related work.

**Equivalence checking and semantic differencing** [PY14, PY13, RE11] are aimed at proving equivalence and describing differences between (versions of) procedures in high-level code, and do not apply to searching for similarity in machine code. The authors of [PY13, PY14] offer some handling for loops, but apply computationally expensive abstract domain libraries, which do not scale in our setting. Sharma et al. [SSCA13] present a technique for proving equivalence between high-level code and machine code, with loops, by trying to find a simulation relation between the two. However the search is computationally demanding and will not scale. Lahiri et al. [LSH15] narrow the scope to assumed-to-be but cannot-be-proved-to-be-equivalent snippets of binary code, and attempts to find a set of changes (add or delete) to complete the proof. While having interesting implications for our problem, the assumption of a variable mapping encumbers the transition. Ng and Prakash [NP13] use symbolic execution for similarity score calculation, but it is not geared towards cross-compiler search, and is limited to handling patched procedures (specifically, only handles one calling convention and rejects procedures based on the number of inputs.)

**Compiler bug-finding** Hawblitzel et al. [HLP+13] present a technique that handles compiled
versions of the same procedure from different compilers. Their goal is to identify root causes of compiler bugs, and their approach cannot be directly applied to our setting as they: (i) require strict equivalence and thus even a slight change would be deemed a bug, (ii) know the IL originated from the same code allowing them to easily match inputs and outputs (i.e. these are labeled) for solving, which is not the case for machine code, and (iii) offer no method for segmenting procedures and thus are limited in handling loops (they use loop unrolling up to 2 iterations).

**Dynamic methods** Egele et al. [EWCB14] present a dynamic approach which executes the procedure in a randomized environment and compares the side effects of the machine for similarity. As they base the similarity on a single randomized run, similarity may occur by chance, especially since they coerce execution of certain paths to achieve full coverage. Their evaluation indeed outperforms [Bin]. However, it is still wanting as they rank similar functions in the top 10 in only 77% of the cases, and do not evaluate over patched versions.

An interesting method by Pewny et al. [PGG+15] uses a transition to an intermediate language (VEX-IR), a simplification using the Z3 theorem prover, sampling and a hashing-based comparison metric. In their results they report several problems with false positives. We believe that this is because sampling alone is used to match two basic-blocks without proving the match, and that basic-blocks are not weighted against how common they are in the corpus (and these basic-blocks might be a compilation by-product). Moreover, the goal of their method is to find clones for whole binaries. Thus, it might be hard to apply in situations where patching was performed.

**Structure-based static methods** Jacobson et al. [JRM11] attempt to fingerprint binary procedures using the sequence of system calls used in the procedure. This approach is unstable to patching, and is only applicable to procedures which contain no indirect calls or use system calls directly.

Smith and Horwitz [SH09] recognized the importance of statistical significance for similarity problems, yet their method is geared towards source-code and employs n-grams, which were shown ([DY14]) to be a weak representation for binary similarity tasks.

The authors of [PSB+14] show an interesting approach for finding similarity using expression trees and their similarity to each other, but this approach is vulnerable to code-motion and is not suited to cross-compiler search as the different compilers generate different calculation “shapes” for the same calculation.

**Detecting software plagiarism** Moss [Aik] (Measure Of Software Similarity) is an automatic system for determining the similarity of programs. To date, the main application of Moss has been in detecting plagiarism in programming classes. Moss implements some basic ideas that resemble our approach: (i) it decomposes programs and checks for fragment similarity and (ii) it provides the ability to ignore common code. However, in Moss the code is broken down to lines and checked for an exact syntactic match, while Esh decomposes at block level and checks for semantic similarity. Furthermore, in Moss the ignored common code must be supplied by the user as the homework base template code, which is expected to appear in all submissions,
while Esh finds common strands by statistical analysis. All the ideas implemented in Moss are preliminary ideas that bear resemblance to ours but are not fully developed in a research paper or evaluated by experiments over binaries.
Chapter 6

Conclusion and Open Questions

6.1 Conclusion

The first goal of this work was to apply abstract interpretation [CC77] techniques towards the goal of soundly and precisely characterizing differences between programs. Chapter 2 presented an abstract interpretation approach for program equivalence and differencing on the level of a single procedure while relying on syntactic similarity. This chapter included two contributions: (i) The definition of the correlating program construct, which allows reasoning over both programs and establishing of equivalence, and was also used and extended in other work [GMR15]. (ii) The definition of a correlating abstract domain, which allows us to maintain variable relationships. This partially disjunctive domain allows us to differentiate equivalent from differencing paths, and we introduced a dynamic partitioning strategy to abstract together paths according to equivalence criteria and avoid exponential blowup. We also defined a widening operator for the disjunctive domain, which over-approximates looping paths and is able to maintain equivalences for programs with unbounded loops. This contribution is key for the use of abstract interpretation for equivalence reasoning over two programs. The evaluation showed that this approach is feasible and can be applied successfully to challenging real world patches.

The results and insights in Chapter 2 fueled the creation of a new abstract interpretation approach for program equivalence and differencing, presented in Chapter 3. The need to reason over equivalence and difference of programs without relying heavily on syntactic similarity to establish program correlation motivated the novel approach of speculative correlation. This algorithm guides the interleaving of the two programs based on the abstract difference between them. The algorithm is instantiated over a previously defined correlating abstract domain. The domain uses a powerful numerical abstract sub-domain to capture abstract program states (and differences) as linear inequalities between variables. The speculative correlation approach was evaluated on previous benchmarks, as well as many new ones where the previous approach falls short. Chapter 3 shows that this approach is feasible and can be applied successfully to challenging real-world patches.

The next goal of this work was to apply differential analysis at the level of whole programs, interprocedurally. As applying purely semantic approaches interprocedurally is known not to scale [RHS95], a data-flow based approach was used as the initial approach, while semantic
tools were introduced in a level which maintains feasibility and scalability. Chapter 4 presented a framework for incorporating change semantics while performing change impact analysis. Semantic-aware change impact analysis demonstrates how to leverage differential invariants in the framework of change analysis and provides a scalability-precision tradeoff with SEM-DCIA-ANYTIME, which is crucial for applying such analyses to large projects. The use of program verifiers [BCD+05] to (i) prove individual procedure equivalence and (ii) bound the impact of a syntactic change to relevant variables using differential assertions was shown to successfully reduce the set of impacted lines across entire programs. An important lesson of Chapter 4 is that applying program verifiers implicitly to improve software engineering tasks such as CIA can go a long way in the adoption of verifiers by developers.

Finally, Chapter 5 introduced a framework for reasoning about the similarity of procedures, post compilation, in stripped binaries. This chapter presented a new statistical technique for measuring similarity between procedures. The main idea is to decompose procedures to smaller comparable fragments, define semantic similarity between them, and use statistical reasoning to lift fragment similarity into similarity between procedures. The technique was used to find various prominent vulnerabilities across compilers and versions. The evaluation shows that the approach produces high accuracy results, with few to no false positives, a crucial factor in the scenario of vulnerability search in stripped binaries, which motivated this research. A key contribution is the statistical notion of similarity presented, as it is general, and not limited to binaries. It was applied to binaries where the utility of other techniques (that use the structure of the code) is severely limited.

6.2 Open Questions

Abstract Semantic Differencing of Probabilistic Programs Probabilistic programs are a prevalent new approach for simulating probabilistic processes. Examples include programs used in risk analysis, medical decision making and cyber-physical systems. The probabilities of events in these programs are queried in the form of assertions over program variables. So far these probabilities have been estimated either by sampling, or more lately by static analysis that provides guarantees over assertions bounds. However, whenever a probabilistic program changes, the entire process of sampling and analysis needs to be repeated, even when the programmer applies small changes to noise bounds or conditionals, which is an integral part of using probabilistic programs. An interesting question would be how the techniques in Chapter 2 and Chapter 3 might be used to allow for fast re-inference of probabilities for changes in noise bound conditionals over noise variables, dramatically reducing computation time.

Abstract Semantic Differencing of Spreadsheets Spreadsheets often operate as de facto programs for manipulating data. The significance of this computation, or more specifically an error in it can have vast repercussions [RR10]. Despite the prevalence of spreadsheets and their importance, they are the focus of very little analysis and verification research [BGB14]. A promising direction would be to leverage differential analysis techniques from this work towards checking and describing semantic difference induced by a syntactic change in a spreadsheet.
Thread-Dependant Equivalence Checking for GPU Kernels An important future challenge is to explore the equivalence of multi-threaded programs. GPU kernel programs are a promising candidate for such research. GPU kernel equivalence is challenging and thwarts existing approaches as it requires (i) analyzing a group of threads in each program and (ii) reasoning over array data. To overcome these challenges, the correlating semantics presented in Chapter 2 and Chapter 3 can be adapted to a multi-threaded correlating semantics, able to reason over variables and arrays in groups of threads over both programs. Due to the nature of GPU kernels, which operate over arrays of shared memory, the correlating abstract domain will need to capture relationships of variables and arrays of unbounded size over the two programs. An exciting result of this research could be a search algorithm for synthesizing thread configuration for equivalence: Given two kernels $P$ and $P'$, this algorithm will enumerate all possible groups of threads in $P$ and $P'$, up to a bounded size, exhaustively searching for the setting featuring minimal difference. This could allow for the automatic resizing and optimization of kernels – a complex and error-prone task, currently performed manually.

Interprocedural Differential Test Generation Chapter 4 leverages the equivalence of arguments at call points, as well as the equivalence of procedures to reduce the set of program lines, impacted by a syntactic change. An interesting question would be how to leverage the difference between procedures. One approach can be to use the description of difference computed in Chapter 2 and Chapter 3, and specifically the inputs that lead to this difference, in order to produce caller inputs that trigger differentiating behavior. This may be used to automatically generate interprocedural tests that apply to the changed behaviors, a challenging task performed manually by coders.

Interprocedural Differential Test Absorption A counterpoint to the previous open question would be: Given a description of output difference for procedures $(P, P')$, are there call points to said procedures, say $(G, G')$ in which the differentiating behavior does not manifest, i.e., the difference of $(P, P')$ is absorbed by the calling environment? Identifying such instances can be of use for program understanding: programmers can know the propagation of the change they made, as well as prune away regression tests for $(G, G')$ as the calling environment was not tainted by the callee.

Statistical Similarity of Programs In Chapter 5 a general notion of statistical similarity was defined, based on procedure decomposition, pairwise similarity and statistical evidence. This general notion can be applied at source code level, to find similarities of source procedures. This may also be applied to find similarity across different programming languages.

Statistical Clustering of Binaries In Chapter 5, we used our notion of similarity for retrieval. An interesting question is: How can the technique be used for clustering and classification of binary procedures?
Bibliography


Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy


קוד פיתוח הנצ cerco בשימוש נרחב. כל פרק מוכיל תתרגים העוסק בנטייה והטיעון. הערכות הביאISISו מסייעות את הניסוח האגודה המשולשת בשוני השונות בין פעולות מתכאות ווכלי التركيนักเตะ
וישימור ליקוד מקרוב בעולמם האמיתית. כנכינה להחייה ואיפור שולגון, שיקולות דמויים בר хочי

iii
לפי בדיקת.InputStreamReader, ניתן להשתתף בפרויקט של מחקר זה, תוך מתן תודעה לשונות שונות. 

לפי מחקר זה, נמצאה ששטחי התוכן, שהצבתו הוא ייצוגים של התוכן, יכולות להיעשות באמצעות שיפורים адונטיים. 

לפי המחקר, התוכן המופיעה בשימוש בקוד מתאים לפעילות התוכן, ניתן להיעשות באמצעות שיפורים адונטיים. 

לפי מחקר זה, התוכן המופיעה בשימוש בקוד מתאים לפעילות התוכן, ניתן להיעשות באמצעות שיפורים адונטיים. 

לפי מחקר זה, התוכן המופיעה בשימוש בקוד מתאים לפעילות התוכן, ניתן להיעשות כך באמצעות שיפורים адונטיים. 

לפי מחקר זה, התוכן המופיעה בשימוש בקוד מתאים לפעילות התוכן, ניתן להיעשות כך באמצעות שיפורים адונטיים. 

לפי מחקר זה, התוכן המופיעה בשימוש בקוד מתאים לפעילות התוכן, ניתן להיעשות כך באמצעות שיפורים адונטיים. 

לפי מחקר זה, התוכן המופיעה בשימוש בקוד מתאים לפעילות התוכן, ניתן להיעשות כך באמצעות שיפורים адונטיים. 

לפי מחקר זה, התוכן המופיעה בשימוש בקוד מתאים לפעילות התוכן, ניתן להיעשות כך באמצעות שיפורים адונטיים. 

לפי מחקר זה, התוכן המופיעה בשימוש בקוד מתאים לפעילות התוכן, ניתן להיעשות כך באמצעות שיפורים адונטיים.
 lakshmanasivan 2021-05-18
המחקר السلطנות על פורפורה ערב ייח, בפקולטה למדעי המחשב.

חלק מההתוצאות בחיבור זה פורסמו כ מאמרים מאית המחבר ושוחחו למחוקר בכונסיו ובכתביהם.

בماتל חקוקчет מחקר הדוקטורט של המחבר, אשר-transparentהundyכיסות ייחו: 

הכרת תודה מסורה לטעני על פימוק מחקר זה.
ניתוח סטטיסטי דיפרנציאלי של תוכנה

תובנות על מחקר

לשכם מילוי חלקי של הדרישות לקבלת התואר
dוקטור לפילוסופיה

נמרוד פרטוש

כומר פרטוש

הוא חדשנט הטכנולוגיה – מכון טכנולוגיה לישראל
tמו התחתון קיבוץ אוגוסט 2016
ניתוח סטטיסטי דיפרגניציאל של תוכנה

נמרוד פרטוש