Fall 2013

Automated Failure Explanation Through Execution Comparison

William Nicholas Sumner
Purdue University

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By William Nicholas Sumner

Entitled
Automated Failure Explanation Through Execution Comparison

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Xiangyu Zhang Chair	
Jan Vitek	
Suresh Jagannathan	
Dongyan Xu	

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Xiangyu Zhang

Approved by: Sunil Prabhakar / William J Gorman — 12 August 2013
Head of the Graduate Program — Date

AUTOMATED FAILURE EXPLANATION THROUGH EXECUTION COMPARISON

A Dissertation

Submitted to the Faculty

of

Purdue University

by

William N. Sumner

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2013

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

My thanks to my advisor Dr. Xiangyu Zhang for serving as my mentor for the past several years, for always engaging in debate, and for pushing me to explore new problems. My thanks also to Dr. Dongyan Xu for helping to guide me in my development as an academic and for many interesting research discussions. And I of course owe a debt to my other committee members, Professors Vitek and Jagannathan, without whom this dissertation could not have happened.

I also owe a debt to all the friends who have helped me or supported me both at Purdue and from afar. To Ian and Alicia, to Kristina, to Henry, to Jayaram, to all of my lab mates, I simply would not have gotten this far without their support, their critiques, and their friendship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Sumner, William N. Ph.D., Purdue University, December 2013. Automated Failure Explanation through Execution Comparison. Major Professor: Xiangyu Zhang.

When fixing a bug in software, developers must build an understanding or explanation of the bug and how the bug flows through a program. The effort that developers must put into building this explanation is costly and laborious. Thus, developers need tools that can assist them in explaining the behavior of bugs. Dynamic slicing is one technique that can effectively show how a bug propagates through an execution up to the point where a program fails. However, dynamic slices are large because they do not just explain the bug itself; they include extra information that explains any observed behavior that might be connected to the bug. Thus, the explanation of the bug is hidden within this other tangentially related information. This dissertation addresses the problem and shows how a failing execution and a correct execution may be compared in order to construct explanations that include only information about what caused the bug. As a result, these automated explanations are significantly more concise than those explanations produced by existing dynamic slicing techniques.

To enable the comparison of executions, we develop new techniques for dynamic analyses that identify the commonalities and differences between executions. First, we devise and implement the notion of a point within an execution that may exist across multiple executions. We also note that comparing executions involves comparing the state or variables and their values that exist within the executions at different execution points. Thus, we design an approach for identifying the locations of variables in different executions so that their values may be compared. Leveraging these tools, we design a system for identifying the behaviors within an execution that can be blamed for a bug and that together compose an explanation for the bug. These explanations are up to two orders of magnitude smaller than those produced by existing state of the art techniques. We also examine how different choices of a correct execution for comparison can impact the practicality or potential quality of the explanations produced via our system.

1 INTRODUCTION

Faulty software is pervasive. It resides in the phones people use to communicate [92], in the cars that people drive [31, 91], and in the medical devices that people rely upon to survive [55]. When the faults or bugs in programs cause those programs to fail or produce unexpected results, the costs can be extensive. A National Institute of Standards and Technology (NIST) report from 2002 found that the financial costs of software failures could reach $60 billion each year [118]. For software that controls devices like cars or pacemakers, these bugs can not only have financial costs but can also cause fatalities [31].

However, fixing bugs to prevent these failures is itself a costly and nontrivial process. The same NIST report estimated that 30-90% of the costs and up to 70% of the time required for successful software projects were spent in testing and debugging. The primary reason reported for this was that the tools available to developers simply were not effective [118]. As a result, there is a desire and a need for new tools to help developers understand the bugs within programs and to correct those bugs.

When trying to fix or debug a fault within a program, developers search through the source code of a program, trying to understand the relationships between different pieces of code [75]. Facing some incorrect output, they reason about the values that the output producing statement depended upon or used and search back through the program to see where those values came from [75]. When they find the statements that produced the incorrect values, the developers repeat the process on these new statements to see why they produced incorrect values, as well. Developers continue this process backward, building an understanding of why the program produced incorrect output, until they reach and understand the original fault or incorrect behavior within the program. Once they understand how the fault propogates through the program, they can start to fix it.

Assisting and automating this searching and understanding process is one of primary goals of research into debugging and debugging tools. In particular, *dynamic slicing* techniques identify the those instructions that influenced the behavior of another instruction during the execution of a program [76]. Developers can use these dynamic slices to automatically identify which instructions affected the incorrect output and avoid trying to reason about these instructions themselves. Furthermore, prior research has shown that slicing allows developers to better understand how a program behaves and to identify and correct faults within small programs more quickly than they otherwise could [47, 74]. We describe these slicing techniques in more detail in Chapter 2.

Dynamic slices can illustrate the relationships between instructions, but they face obstacles. Although prior work has made strides in improving their efficiency, such slices can still be prohibitively large. Thus, even though they show how individual instructions may influence others, they still require the developer to examine too many instructions to realistically help a developer to understand a bug [135]. In part, this happens because many instructions may influence an observed failure even if those instructions are correct themselves. Thus, developers can spend their time searching through these instructions that are irrelevant to the debugging task instead of those that help to explain how the fault propogated through the failing execution [75]. Furthermore, traditional dynamic slicing can only help to explain why something incorrect happened within a failing execution. It cannot help to explain why some missing behavior *did not* happen within an execution [5, 54].

To avoid searching backward through irrelevant instructions within an execution, another approach known as *fault localization* computes a list of the most suspicious statements in a program [29, 34, 67, 100]. These statements are the ones deemed most likely to contain the fault itself, so the developer can potentially identify the fault without necessarily searching through irrelevant instructions of an execution. However, because fault localization techniques do not help developers build an understanding of how a fault propogates through an execution, these techniques require developers to recognize and understand a bug upon simply seeing a faulty statement. Studies show that this is uncommon in practice [95]. Chapter 2 also provides further background on fault localization.

The focus of this dissertation is on techniques for explaining how a fault propagates through a failing execution of a program to an observed failure. These explanations should precisely show not only how a failing execution behaved incorrectly but also how the fault propogated via missing behaviors that the incorrect execution should have performed. Similar to dynamic slicing, such a technique needs to show how some instructions can influence others within an execution to propogate the fault. However, dynamic slices include information about instructions that do not cause the observed failure and can impede a developer's ability to understand and fix the program. These instructions that do not help to explain the bug should be omitted from resulting explanations. Furthermore, the explanations should also highlight instructions that were not performed by the failing execution but should have been to produce correct behavior.

The primary challenge, then, is to identify which instructions within the failing execution helped to cause the failure, so that they can be included in the explanation. This dissertation explores how a failing execution may be compared and contrasted with an execution that does not fail in order to aid in identifying which instructions caused and thus explain the failure. To realize this approach, we identify and address core problems that arise when analyzing relationships across executions. We use these solutions as a platform for dynamic program analysis and derive a means for identifying the instructions that should be included within an explanation for a bug as well as the relationships between these instructions. We then examine the practicality of comparing a buggy execution against a correct execution and examine how the choice of a correct execution could influence the quality of an explanation. Comparing executions enables the efficient automated construction of concise explanations for software failures.

In particular, this dissertation makes the following contributions:

- We note that examining or comparing relationships that exist within or across multiple executions can require a way to identify corresponding instructions across executions, as well. These identities are necessary for determining whether or not a relationship between two instructions in one execution also holds between the same instructions in another execution. Prior research has looked into approaches for identifying instructions within a single execution, but we demonstrate that these approaches are insufficient for comparing multiple executions. This dissertation presents the design and implementation of a technique for efficiently and scalably identifying corresponding instructions.

- Comparing executions can employ not only examining which instructions execute but also the values of variables present within an execution. However, dynamically allocated variables do not have names and may be allocated within different portions of memory in different executions. This impedes identifying the memory that corresponds across different executions and thus makes comparing the values within this dynamically allocated memory difficult. This dissertation further presents a mechanism for identifying such corresponding memory within different executions.

- Using these primitives, we present an approach for contrasting two executions to determine which portions of them behaved differently and then determine which of the differences can be blamed for causing a failing execution to behave differently than a correct execution. Prior work has also examined ways of identifying differences between executions that can be blamed for causing one execution to fail. We identify shortcomings in these approaches and further show that the approach presented within this dissertation does not succumb to these same problems.

- Comparing a failing execution against a correct execution to create explanations will produce different results depending on precisely which correct execution the technique contrasts with the failing execution. Some correct executions may produce desirable explanations that show how the fault propogates through the program and to the failure. Other correct executions will simply produce explanations that reflect the different intent of the correct and incorrect executions. Selecting a correct execution in such a way as to explain the failure and not the different intents of the two executions is thus crucial to the overall effectiveness of comparing two executions to explain failures. This dissertation presents an empirical study of techniques that can select correct executions when given a failing execution as input and examines the costs and benefits of the different techniques that could affect how developers use them in practice.

Building upon these technical contributions, this dissertation shows that contrasting a failing execution with a passing execution can produce concise explanations that illustrate how a fault propogates through an execution. These explanations are more precise than prior techniques for explaining failures while capturing not only the observed buggy behavior of the failing execution but also any missing correct behavior that the failing execution omitted.

Chapter 2 presents additional background and illustrates both what an explanation for a failure is and how the techniques presented in this dissertation may fit together to help construct such explanations. Chapter 3 examines how correspoding or aligning instructions across multiple executions may be identified. Chapter 4 similarly shows how to identify corresponding regions of memory across executions and how the values of variables in one execution may be inserted into another. Chapter 5 examines what it means for one instruction within an execution to cause another when comparing two executions and shows how these causal relationships can be used to build explanations for bugs in software. Chapter 6 then looks at how the correct execution used in comparison may be selected and how it may or may not be a good execution to use for building an explanation for a failure. Finally, Chapter 7 and Chapter 8 look at related work and the conclusions that may be drawn from this dissertation, as well as potential directions for future work.

2 EXPLAINING SOFTWARE FAILURES

Fixing a faulty program first requires understanding why that program is faulty. Developers usually gain this understanding by searching through a program's code and observing how a fault propagates through the program during its execution until that fault becomes an observable failure [75]. However, understanding the behavior of a running program by examining its source code is difficult because it requires developers to execute the code mentally [38,83]. In order to help developers with this task, researchers have put substantial effort into tools that can help developers to explain and understand program behavior [4, 32, 50, 74]. These tools focus primarily on two techniques known as *slicing* and *fault localization*. In this chapter, we examine the existing work in these areas designed to help explain software failures, and we present a design for a new technique for explaining failures that the following chapters shall explore.

2.1 Slicing

Static program slicing was originally introduced by Weiser as an approach for eliminating irrelevant or uninteresting parts of a program when trying to understand program behavior [126]. His technique use dataflow equations to determine which statements in a program were necessary for computing a *criterion*, or the value for a variable at a particular statement. Because the slice captured all statements that could influence the criterion, when the criterion was a buggy value, the static slice represented the possible locations of the bug within the original program. Weiser also noted that developers intuitively computed the slice manually when attempting to understand software failures on their own [127].

Successive work refined and redefined the notion of static slicing through the use of program dependence graphs (PDGs) [44, 59, 93]. These graphs comprise each instruction of a program as a node and the ways that instructions may influence each other as directed edges or dependences between nodes. These dependences may capture either data dependence, where an instruction at a target node uses a value produced by the instruction at a source node, or control dependence, where the instruction at the source node can directly determine whether or not the instruction at the target node executes. The refinements to static slicing examined both how to compute slices by determining whether one node in the graph is reachable from another and how to define slices in terms of subgraphs of the PDG [59, 78, 93, 110].

While static slices allow developers to narrow the amount of code that must be looked at in order to understand a program, they still pose two substantial challenges. Static slices contain all instructions that might influence a criterion within any possible execution of a program. This can be more information than a developer needs to understand one specific execution [76]. In addition, the burden of understanding how that code behaves at runtime still falls upon developers, but understanding the dynamic behavior of static code is difficult [38,83]. Thus, Korel and Laski introduced dynamic slicing to help explain the behavior of a single concrete execution of a program [76]. Whereas static slices comprise instructions and dependences from the original program, dynamic slices comprise instructions and dependences from a trace of a concrete execution of the program. A trace is simply the sequence of instructions performed by a particular execution of a program. Unlike the original program, a single static instruction may appear multiple times as different instruction instances within the trace, and only the executed instructions occur in the trace. The dependences within dynamic slices are the control and data dependences between those individual instruction instances within the trace. Thus, dynamic slices include only information from a particular execution of interest, and they also show why the execution behaves as it does through the dependences within the trace. Following the original paper, additional work also showed how to improve efficiency and correctness when computing dynamic slices from traces [3, 136, 138] An extensive survey by Tip contains further details and history on the development of static and dynamic slicing [119].

User studies have shown that dynamic slices provide an effective means of explaining the faulty behavior of executions during the debugging process [74]. However, these studies examine small programs with short traces, e.g., 500 lines of code (LOC). In practice, programs may be tens of thousands to millions of lines of code, and traces may easily include hundreds of thousands, millions, or even more instructions. As a result, dynamic slices themselves may be long and complex because each instruction can depend on multiple other instructions, thus transitively including them in the slice. The size and complexity of dynamic slices precludes their utility in practice, so additional approaches aim to heuristically organize and prioritize which portions of a slice to examine in order to make slicing more practical [80, 111, 124, 135]. Nonetheless, for these approaches, the developer may have to examine the entire slice in order to understand a bug. Furthermore, understanding a failure may require explaining why some missing behavior does not occur in an execution and thus is not captured by the slice [5, 54, 140]. These slicing techniques do not help to explain such failures.

2.2 Localization

In contrast to slicing, fault localization does not seek to explain how a fault leads to a failure. Instead, localization focuses on identifying possible locations, or root causes, for a bug within a program [29, 34, 67, 100]. The onus is then put upon the developer for interpreting the results to understand which of these possible locations is a real bug and how. Fault localization techniques return a list of possible root causes to the developer, often ranked by the suspected likelihood that each individual statement is the actual cause of the bug, also known as the suspiciousness of the statement [33, 67, 100]. Ideally, the list of potentially buggy statements is small and thus practical for the developer to examine. In practice, the developer can choose a priori the number of statements to return or examine.

There are a variety of categories of techniques for identifying such root cause candidates [128]. For instance, spectra or profile based techniques look at the passing and failing executions within a test suite. In general, they identify properties like statement coverage, execution frequency, likely invariants, and return values that seem to correlate more with failing tests than passing ones [2, 8, 33, 66, 67, 100, 103, 104]. Related to spectra based approaches are techniques that build more detailed statistical models from spectra to refine the ranking process [10, 11, 43, 49, 82, 84]. In contrast to both of these categories, some approaches modify either the source of the program or the values of variables at runtime and then reason about the effects these changes have on program behavior in order to identify possibly buggy statements [24, 28, 63, 134]. Yet another category of localization techniques focus instead on symbolic execution and compare the constraints generated by passing and failing executions [14, 68, 97, 123].

In spite of the extensive research in fault localization, it still faces obstacles. Fault localization focuses on returning potential root causes to developers, yet user studies show that developers do not recognize buggy statements when localization tools present them [95]. Also understanding the context of a bug and how it propagates through a program is crucial to understanding and fixing a bug [47, 74, 75, 95]. In addition, a desirable fix for a bug may not always occur at the root cause because of other engineering concerns [88]. As a result, understanding how the faulty behavior propagates through a program is crucial for identifying and applying alternative corrections for faulty program behavior.

2.3 Concise Explanations via Execution Comparison

The advantages and limitations of these approaches to explaining failures are complementary. While dynamic slicing is able to explain how a fault propagates through a buggy execution, the slices produced can be prohibitively large in practice. In contrast, fault localization techniques are concise; they can produce small lists of statements for the developer to examine. However, the returned candidates may not relate to the actual cause of the bug, and even if they are, the developer still bears the burden of recognizing the bug and understanding how it propagates through the program. Contrary to both of these approaches, a desirable explanation for a failure should both be concise and show how a fault propagates through an execution. This section presents a general approach for producing such explanations that the remainder of this dissertation examines in detail.

```
1 inventory = [(Shoes,5); (Hats,0); (Ties,1)]
2 bought = 0
3 for (item, available) in inventory:
4   if bought < 3 and available >= 0 :
5     buy(item)
6     bought += 1
7 print "Bought:",bought
```

(a)

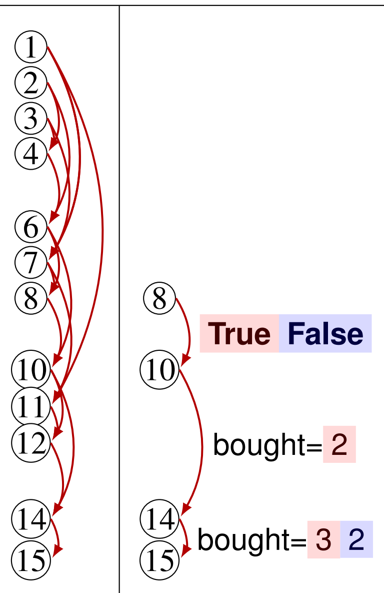
```
1  inventory = . . .
2  bought = 0
3  for (i, a) = (Shoes,5):
4    if True:
5      buy(Shoes)
6      bought = 1
7  for (i, a) = (Hats,0):
8    if True :
9      buy(Hats)
10     bought = 2
11 for (i, a) = (Ties,1):
12   if True:
13     buy(Ties)
14     bought = 3
15 print "Bought:", 3
```

(b)

```
inventory = . . .
bought = 0
for (i, a) = (Shoes,5):
  if True:
    buy(Shoes)
    bought = 1
for (i, a) = (Hats,0):
  if False :


for (i, a) = (Ties,1):
  if True:
    buy(Ties)
    bought = 2
print "Bought:", 2
```

(c)



(d) (e)

Figure 2.1.: An example of explaining a bug using a dynamic slice and a more concise explanation.

One key insight to computing such explanations is that dynamic slices include more information than necessary or desirable to explain a specific bug. Consider the example program in Figure 2.1a. This program maintains an inventory of items in a store along with the available quantities of those items. It iterates through the inventory and buys one of each available item in the inventory. Note that there is a bug on line 4. When checking to see whether an item is available, the program checks whether available >= 0, but this should actually be available > 0. As a result, the program will buy an item even if none of that item is available in the inventory. Figure 2.1b-c present the trace of the buggy execution and the trace of the intended correct execution respectively. Note that in the buggy execution, the program buys Hats even though no Hats are available in the inventory. Thus, the buggy execution prints that it bought three items when it should only have bought two. The correct execution, in contrast, yields **False** on the check for whether Hats are available, so it prints that two items were bought.

Figure 2.1d presents a dynamic slice of the buggy trace in (b) using the printed value of bought as the criterion. Note that most of the statement instances in the trace are also in the slice. In this case, the slice was not able to substantially winnow down the list of statements to examine. Furthermore, because statement instances depende on multiple other statements in the trace, it is difficult to decipher from the slice which dependences were actually buggy versus which were correct. For example, statement 14 in the trace is dependent upon both statement 12 and statement 10. While statement 10 incorrectly increases the value of bought and leads to the failure, statement 12 is actually correct and executed no differently than in the correct execution. Traditional dynamic slicing techniques cannot distinguish between these dependences and must include both of them in the slice, increasing its size and complexity.

Contrasting the traces of the buggy and correct executions, we can identify and elide those dependences that do not help to explain the buggy behavior. That is, we elide statments that behave the same way in both the buggy and the correct traces. This produces an explanation like that presented in Figure 2.1e. This explanation directly identifies the buggy statement instance 8 in the second iteration of the loop. The program incorrectly takes the **True** branch of the **if** statement when checking whether Hats are available. Because of this, it incorrectly increments bought, and this incorrect value propagates throught the next iteration until it is printed out at the end of the execution. Each step of the explanation shows how the fault propagates through the program until the observed failure, and the explanation only presents dependences that help to explain the failure. Contrasting the two executions provided a means of pruning the dynamic slice into a clearer and more concise form. Thus, this explanation for why the buggy and correct executions differ serves as a concise explanation of the bug itself.

```
EXPLAIN($\tau$, $\tau$')
Input:   $\tau$ & $\tau$' - buggy / correct traces of the program
Output:  An explanation of why $\tau$ differs from $\tau$'
 1: output $\leftarrow$ OUTPUTDIFFERENCES($\tau$,$\tau$')
 2: criteria $\leftarrow$ GETDEPENDENCES(output)
 3: explanation $\leftarrow$ $\emptyset$
 4: failurePoint $\leftarrow$ $\bot$
 5: while criteria $\neq \emptyset$ do
 6:     last $\leftarrow$ GETLASTDEFINITION(criteria)
 7:     causePoint $\leftarrow$ ALIGNEDPOINTTHROUGH($\tau$,$\tau$', last)
 8:     possibleCauses $\leftarrow$ CUTSLICEAT($\tau$,$\tau$', criteria, causePoint)
 9:     causes $\leftarrow$ FINDCAUSES($\tau$,$\tau$', causePoint, failurePoint, possibleCauses, criteria)
10:     explanation $\leftarrow$ explanation $\cup$ causes
11:     failurePoint $\leftarrow$ causePoint
12:     criteria $\leftarrow$ causes $\cup$ PRECEDING(criteria, causePoint)
13: return explanation
```

Figure 2.2.: An abstract technique for explaining why two executions differ.

Using this underlying intuition as a model, we then need a means of computing these explanations. In practice, we do not have the corresponding correct execution available, as that would defeat the need for debugging, but we shall first assume the correct execution is available and then revisit this assumption. At a high level, we can compute these explanations by modifying traditional algorithms for dynamic slicing as presented in Figure 2.2. Using the observable output differences in the two executions as slice criteria, lines 1-2, the remainder of this framework works backward through the two executions to produce an explanation containing only those dependences in the slice that explain why the buggy and correct executions produced the differing results in the criteria. Line 3 starts the process with an empty explanation. Line 4 sets the *failure point*, the execution point or dynamic instruction instance for which each iteration of the algorithm examines the criteria and finds the smallest set of dependences that can explain them.

The last starting point or definition for any dependence in the criteria is the first point in a backward traversal of the slice that we may prune the explanation, so line 6 jumps back to that execution point. Note, however, that we can only compare two executions at an execution point that exists within both executions. For example, if one execution takes the **True** branch of an **if** statement and the other does not, we cannot sensibly compare how the executions differ at a statement within the **True** branch. Thus, line 7 finds the *cause point* or the closest point before or at the definition that corresponds or *aligns* across the two executions. The algorithm then identifies the set of dependences in a slice backward from the failure point that are live at the cause point and are sufficient for explaining why one execution failed and one did not. Line 8 first identifies the live dependences in a slice from the failure point. These comprise the possible causes for the criteria at the cause point. Line 9 contrasts and analyzes the two executions two find the minimal set of these possible causes sufficient for explaining why the executions behaved differently. Line 10 adds this pruned set of dependences to the full explanation. Lines 11 and 12 then prepare for the next step backward along the slice by setting the next failure point to the current cause point and by setting the criteria to the identified causes as well as any output differences that preceded the cause point. In this way, each iteration of the algorithm works backward along the cause points, finding the smallest sets of dependences in the slice that are sufficient to explain the bug. Finally, once the slice has been traverse, the loop on line 5 ends, and the algorithm returns the complete explanation on line 13.

Consider again the failing and correct executions in Figure 2.1b-c. The algorithm starts by identifying the output difference on line 15 of both executions as a slice criterion and the print statement itself as the first cause point. The findCauses procedure determines that if the correct execution had the value 3 for bought at line 15, it would have produced identical output to that of the failing execution. Thus, the technique identifies the differing values of bought as an explanation for the bug at line 15. The algorithm then proceeds backward to the definition of bought at line 14. Having the incorrect value of bought at this point is sufficient to produce the incorrect value at line 15, so the different values of bought explain the failure here, as well. The algorithm then jumps back to the incorrect value of bought at line 10. This **True** branch exists only in the buggy execution, so the process proceeds back to line 8. Here, we find that the had the correct execution produced

True for the **if** condition, it would have reproduced the buggy value of bought at line 14. Thus, the differing conditions explain the failure at line 8, and there are no more relevant dependences to traverse back because there are no execution differences before line 8. Thus, the algorithm produces the desired explanation like that present in Figure 2.1e

While this provides a general framework for explaining failures, there remain missing details that must be addressed. Lines 7, 8, and 9 of the algorithm explicitly rely on the notion of corresponding execution points that align across multiple executions. This alignment is necessary in order to contrast the two executions at a point that meaningfully corresponds, yet prior work does not solve the problem. In Chapter 3, this dissertation examines the problem and devises a technique for efficiently labeling every execution point in any execution of a program. These labels are equal for instructions in different executions only if the instructions meaningfully align.

Lines 8 and 9 of the algorithm identify possible causes for the criteria at an execution point and then contrast the buggy and correct executions of a program to winnow these down into only those sufficient for explaining the bug. In practice, this winnowing process in findCauses works by (1) running the correct execution up to the cause point, (2) replacing a portion of the execution's state, or variables and values, with those from the buggy execution, and (3) continuing the execution in order to find out whether the replaced program state was able to reproduce the failure within the correct execution. Two key problems must be addressed in order to perform this task. First, we must be able to identify the memory containing different variables across different executions in order to replace the state of one execution with the state of another. Chapter 4 examines the problem in detail and proposes a solution that enables this state replacement. Prior work explored the process of replacing program state and continuing an execution to identify causes [132], but subtleties in the problem cause existing techniques to yield both inefficient techniques and incorrect results. Chapter 5 explores the problem in detail and derives a new solution that avoids the pitfalls of previous approaches.

We also made the assumption that the correct execution was somehow available so that we could contrast it with the failing one. In practice, this is not the case, as it would obviate the need for debugging. Instead we need to find an execution that behaves approximately as the correct one should have. We can then contrast against these approximately correct executions instead in order to explain bugs. The difficulty, then, lies in finding an approximately correct execution. Chapter 6 explores different possible techniques for finding these executions and compares the quality of the approximations they create.

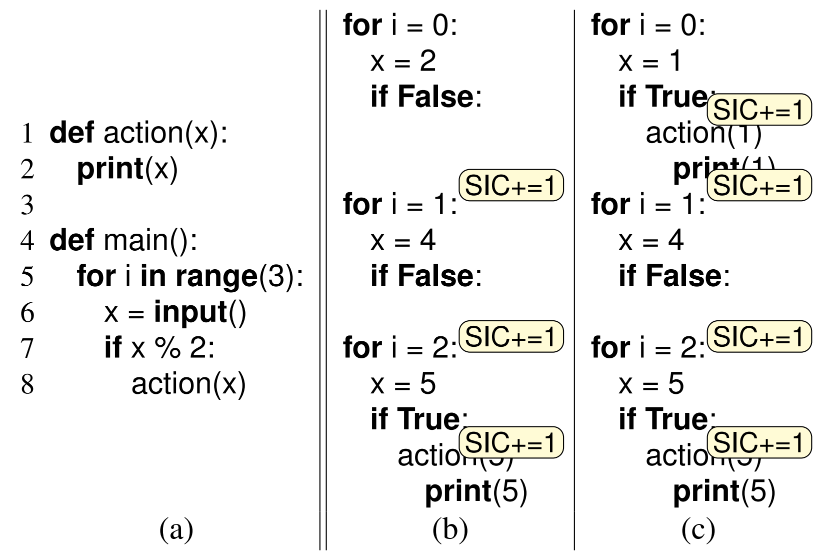
Together, these techniques enable the algorithm in Figure 2.2 to produce concise explanations of real world bugs. The remainder of this dissertation examines these techniques in detail.

3 IDENTIFYING EXECUTION POINTS

Dynamic analyses help developers to identify interesting program behaviors within an execution of a program. As a result, these analyses can simplify or speed up common development tasks like debugging, as presented within this dissertation, and verification [52]. One fundamental task in dynamic analyses is identifying a point within an execution of a program. Such *execution points* are sometimes used to provide feedback to developers [106, 107]. For example, when a tool like Memcheck within Valgrind [107] identifies an invalid memory access, it can provide an execution point that shows the developer where inside the execution this invalid access occurred. The developer can then use this information to help create a fix for the bug. These execution points can also be used as input to additional analyses over executions. For instance, *dual slicing* uses execution points to identify commonly executed instructions across multiple executions [125]. These common behaviors are then used to prune out irrelevant dependences from specialized slices that concisely explain concurrency bugs. The algorithm for explaining bugs in Figure 2.2 uses execution points similarly.

In spite of this pervasiveness, dynamic analyses are inconsistent and imprecise in how they identify and compute these execution points. Many dynamic analyses create their own formulations of *execution point IDs* (EPIDs) without understanding the approaches taken by prior work, and even among the existing techniques, different types of EPIDs have properties that have not been explored. Thus, their strengths and weaknesses are poorly understood and can lead to unexpected limitations of precision or scalability when used by a dynamic analysis. In addition, one definition of execution point may be preferable in one context but undesirable in another, yet these trade offs between the different techniques are presently not well understood.

Consider the program in Figure 3.1a. This program reads in three numbers from the user. If a number is odd, as checked on line 7, then the program calls `action()` to print the number



```
1 def action(x):
2   print(x)
3
4 def main():
5   for i in range(3):
6     x = input()
7     if x % 2:
8       action(x)
```

(a)

```
for i = 0:
  x = 2
  if False:

              SIC+=1
for i = 1:
  x = 4
  if False:
              SIC+=1
for i = 2:
  x = 5
  if True:
    action(5) SIC+=1
      print(5)
```

(b)

```
for i = 0:
  x = 1
  if True:    SIC+=1
    action(1)
      print(1) SIC+=1
for i = 1:
  x = 4
  if False:
              SIC+=1
for i = 2:
  x = 5
  if True:
    action(5) SIC+=1
      print(5)
```

(c)

Figure 3.1.: A program that prints odd numbers and two executions of the program with additional bookkeeping to identify execution points.

out. Notice that line 2 can execute many times because it is called from within the loop. As a result, simply using the line number to identify the execution point is *ambiguous* because the same ID may appear multiple times within the same execution. This is undesirable for many dynamic analyses, as it yields imprecise or incorrect results [26, 116].

One approach, commonly used in the context of record and replay techniques [26, 72, 102, 122], is the *Software Instruction Counter* (SIC). An SIC uses a single integer counter that increments at function calls and loop backedges during the execution of a program. Combining the current counter with the current line number yields a pair (*counter*, *line*) that can uniquely identify an execution point within one execution. For example, the instances of the **if** statement on line 7 of execution (b) are identified by (0,7), (1,7), and (2,7) because of the counter increments on back edges as shown in Figure 3.1b.

Note, however, that these identifiers only work *within a single execution*. That is, the SICs used for execution (b) do not work for the instructions of execution (c). This is because the executions behave differently. In particular, the SIC is incremented at the call to action() in the first iteration of execution (c), so the identifiers for the **if** statements are (0,7), (2,7), and (3,7). Because the SIC was incremented at the first call in (c) but not in (b), the SICs of the two executions diverge and cannot be compared after this point. This is a substantial problem for dynamic analyses that wish to compare information *across multiple executions* [65, 114, 116, 125] because SICs can only precisely identify points within one execution.

To address this problem, and enable comparison across executions, other EPID techniques exploit program structure [6, 130]. Using this information, they are often able to align the corresponding instructions across multiple executions. Unfortunately, these techniques also have limitations that create ambiguous or meaningless relationships when identifying the instructions that align across executions. They also have substantial limitations in usability. In particular, Structural Execution Indexing [130] has difficulties scaling to longer executions, while STAT [6] requires a program core dump at each point that requires an EPID. In addition, SEI can fail to identify useful relationships between EPIDs.

In this chapter we survey five existing approaches used to compute EPIDs for dynamic analyses. The surveyed techniques range in their precision and purpose from only being able to imprecisely identify points even in one execution to uniquely identifying points across multiple executions even in the presence of concurrency and nondeterminism. They range in runtime overhead from none, using only postmortem analysis, to several times the cost of the original execution. Based on the limitations of the existing techniques for cross-execution EPIDs, we observed a need for a new technique that provides meaningful and unambiguous relationships in execution alignment and without the usability and scalability limitations of existing approaches. We introduce a new technique for *Precise Execution Point IDs* (PEPIDs) that addresses these goals and has a runtime overhead only slightly higher than using calling contexts [117].

We have implemented all of these techniques, those surveyed along with PEPID. We evaluated them empirically on SPEC CINT2006 to illustrate their performance. We also provide the first analytical comparison of these different approaches, weighing their costs, their benefits, and the scenarios where one may be more desirable than another. Using this information, a dynamic analysis designer can know in advance which techniques are most appropriate for his or her purposes and avoid inventing or reinventing an approach with known problems. In summary, the contributions of this chapter are:

1. We surveyed and implemented existing techniques for computing EPIDs for dynamic analyses. We analytically examine all of the different techniques and compare them along several spectra in order to weigh their relative costs, merits, and limitations.

2. We observed problems with producing meaningful, unambiguous relationships between EPIDs as well as with usability and scalability in existing techniques for cross-execution EPIDs. To address these problems, we introduce a new cross-execution technique (PEPID) and show that it avoids the limitations of existing work while also having lower runtime overhead.

3. We empirically compare the runtime and space overheads introduced by the different techniques, those surveyed as well as PEPID, and we show that for cross execution EPIDs, PEPID is the most efficient with 25% average overhead. For intra-execution techniques, SICs are the most efficient, with 9% overhead.

4. We illustrate how missing meaningful relationships between inter-execution EPIDs can result in undesirable or incorrect results for dynamic analyses.

3.1 Existing EPID Techniques

In this section, we review the different approaches for computing EPIDs that have appeared historically in the context of dynamic analyses. We consider the intended use cases, design, and requirements of each technique.

3.1.1 Calling Contexts

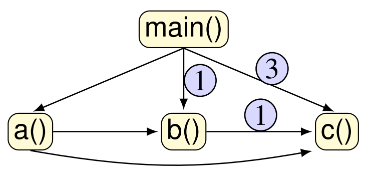
One of the traditional representations of EPIDs is the *calling context* at a point within an execution. The calling context consists of the list of active functions currently on the call stack. Note, similar to using the line number or program counter as an EPID in Figure 3.1, the calling context is an ambiguous representation. The same calling context may appear multiple times even within one execution. As a result, calling contexts are potentially ambiguous EPIDs, but they provide a more detailed representation of the static program behavior than just a line number. In spite of this, calling contexts are already familiar to developers and can be easily collected by walking over the call stack [90, 107]. As a result, many dynamic analysis tools use calling contexts during analysis or while generating reports for developers [94, 106, 107, 132].

In spite of their familiarity, calling contexts were traditionally costly to collect. Walking over the call stack at every point of interest can be costly, which has forced some dynamic analyses to resort to sampling techniques that only analyze portions of an execution [141]. More recently, efforts have focused on efficient means of encoding calling contexts. These include approaches that can probabilistically encode contexts in constant space [18, 19, 89] as well as approaches that can precisely encode calling contexts but can require a flexible encoding size and a slightly higher runtime overhead [117].

In the context of dynamic analysis, more precise information is usually preferred, so in this chapter we only consider the latter work, *Precise Calling Context Encoding* (PCCE) [117]. PCCE works on the principle that calling contexts are equivalent to paths through the call graph of a program. The technique examines the call graph during compilation and numbers all of the acyclic paths present in the graph. It then annotates every edge, or call site, with an arithmetic operation that computes the numerical ID of the current path in the call graph at runtime, similar to Ball-Larus path profiling [12]. Combined with the current instruction, these comprise a calling context. PCCE handles recursion by pushing and popping the acyclic path IDs onto a calling context stack as necessary.

For example, consider the call graph presented in Figure 3.2a. The circle annotations on the call edges denote the amount added to the context ID before each call and subtracted from the ID upon return. Figure 3.2a illustrates this instrumentation for the function b(). Using this example, the calling context main→b→c is captured by the pair (c,2). This reflects the currently executing function, c(), and the numerical ID representing the path in the call graph, 2.

Computing these IDs using PCCE requires that a program be instrumented at compile time, which requires forethought and time not applicable for all dynamic analyses. For instance, if a developer wishes to analyze an already compiled program with a tool that uses PCCE, they have to compile the program again to have the necessary instrumentation added. In addition, the efficiency results achieved by PCCE, 1-3.5% runtime overhead, exploit profile guided instrumentation and additional optimizations for compressing repetitive and recursive calling contexts. Both of these requirements can be avoided by using stack walking to extract the calling context, but, as mentioned before, this induces a higher overhead [19].



(a)

```
def b():
  ...
  contextID += 1
  c()
  contextID −= 1
  ...
```

(b)

Figure 3.2.: (a) An annotated call graph that encodes all calling contexts into unique integers. (b) Example instrumentation of the function b().

3.1.2 Software Instruction Counters

Mellor-Crummey and LeBlanc introduced Software Instruction Counters (SICs) to provide a more precise notion of execution point for profiling and debugging [86]. SICs have since been used in a variety of dynamic analyses, especially in the context of nondeterministic recording and replay [26, 72, 102, 122, 131]. SICs provided the first representation of execution points that was able to uniquely and scalably identify every instruction within a single execution of a program. They work by maintaining a monotonic counter that indicates the progress through an execution. This gives SICs the advantage of only adding a single counter and sparse increment operations to an execution, thus yielding low overhead. While the EPIDs defined by SICs are unambiguous within a single execution, the SIC for a point may change across different executions, and the same SIC may even represent different execution points in two different executions as seen in Figure 3.1 and Section 3.

Computing SICs involves incrementing a counter at every function call and back edge in the control flow graph (CFG) of a program. Figure 3.1b-c show the executions of Figure 3.1a with instrumentation for computing SICs (where **print**, **range**, and **input** are built-in commands). For any point within an execution of a program, the SIC instrumentation creates a pair, (*counter*,*current line*) such that the pair uniquely identifies that execution point. The counter maintains a notion of forward progress within the execution, and it is only incremented at those features within an execution that may cause an instruction to execute multiple times (loops and function calls).

Accurately placing instrumentation on back edges requires static analysis or some additional dynamic analysis to detect loops within individual functions. This mandates either forethought for the static analysis, just like PCCE, or additional runtime, space, and complexity overhead for dynamic loop detection. Instead of instrumenting back edges in the CFG, many analyses alternatively instrument the branch points within a program [72, 122]. For executions that terminate or have side effects, these are equivalent and have the advantage that branch instructions can be easily identified and instrumented by dynamic instrumentation or virtualization tools [85, 107].

3.1.3 Structural Execution Indexing

While the EPIDs provided by SIC suffice for *intra-execution* analyses, we saw in Section 3 that an EPID defined by SIC might correspond to the first iteration of a loop in one execution and the last iteration of a loop in another execution. Indeed, the alignment that SICs create between the instructions of two different executions can match instructions at the *beginning* of one execution with instructions at the *end* of a second. For dynamic analyses that perform *inter-execution* analysis, e.g. execution comparison, this can lead to meaningless results. Intuitively, when there is no relationship between instructions with the same EPIDs across the two executions, comparing them is uninformative.

```
 1 def action(x):
 2   print(x)
 3
 4 def main():
 5   for i in range(3):
 6     push((5, 14))
 7     x = input()
 8     if x % 2:
 9       push((8, 13))
10       push((11, 12))
11       action(x)
12       pop(12)
13     pop(13)
14   pop(14)
```

(a)

Possible Calls to action():

$\langle (5,14)(8,13)(11,12), 11 \rangle$

$\langle (5,14)(5,14)(8,13)(11,12), 11 \rangle$

$\langle (5,14)(5,14)(5,14)(8,13)(11,12), 11 \rangle$

(b)

Figure 3.3.: (a) The program from Figure 3.1 instrumented for computing SEIs. The consecutive pushes at 9 and 10 are discussed in the text below. (b) EPIDs for potential calls to action().

This provided the motivation for *Structural Execution Indexing* (SEI) from Xin et al [130]. They observed that some dynamic analyses compare execution points across executions, but the way that analyses identified execution points led to meaningless correspondences, like those established by SIC in Section 3 [114, 130]. They instead sought to use the semantic structure of underlying programs to determine which program points corresponded. They observed that the control structures of a program along with the dynamic control dependence [44] at runtime established a semantic identity for execution points even across different executions, so they used these to uniquely identify instructions at runtime. The technique maintains a stack that keeps track of the currently active control structures while a program executes. This stack then acts as the EPID.

The process of computing an SEI based ID for an execution point is similar to manually maintaining a call stack at runtime, except that dynamic control dependence information is also included in the stack. At every branch (or call) instruction, the instruction ID is pushed onto an indexing stack along with the ID of its postdominator (or return instruction). This *(ID, postdominator)* pair identifies the region of code that is control dependent upon the branch (or call) instruction. Upon encountering a postdominator (or return), all entries in the stack postdominated by that instruction are popped from stack. Applying this process to the code from Figure 3.1 yields the new program in Figure 3.3a. Note, for each dynamic iteration of the loop, an *(ID, postdominator)* token is pushed on the stack at line 6. As seen in Figure 3.3b, showing the EPIDs for each call to action(), these tokens track the monotonic progress of an execution through a loop until the loop finishes and all iteration tokens are popped at line 14. The pop(x) operation removes all tokens with the postdominator x. The push and pop on lines 9 and 13 bound a region of code control dependent upon line 8 [44], while those on lines 10 and 12 identify the call on line 11. Uniquely identifying function calls is crucial because the same function may be called multiple times, and not differentiating the call sites would lead to ambiguous EPIDs.

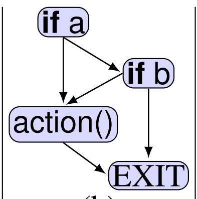
The complete algorithm also contains additional operations for optimizing simple loops using counters and for eliminating pushes onto the stack that can be inferred based purely on where an instruction lies within the CFG. For example, the push for each loop iteration on line 6 can be replaced by a counter increment, since the loop has a single conditional guard. Also, executing the body of the **if** statement in Figure 3.3 automatically implies that the **if** statement on line 8 was executed and the **True** branch taken. Thus, the pushes and pops on lines 9 and 13 can be safely elided.

The intuition that control dependence creates a semantic relationship across executions had previously been used for trace similarity metrics [53] and has proven effective enough that SEI has gained traction in analyses that examine *inter*-execution relationships. It has since been used for tasks ranging from automated debugging [116] to concurrent profiling [139] to identifying causes of security vulnerabilities [65]. In spite of this, tracking control dependence can require $O(N)$ space where N is the length of an execution, which does not scale for some programs. The aforementioned optimization heuristics can mitigate this problem in practice, but they do not eliminate it. We explore the space and runtime overheads in Section 3.4. In addition SEI requires that a program be instrumented at compile time to accurately identify postdominators.

As previously noted, SEI was designed to guarantee EPIDs across executions could only be equal for execution points that correspond across the executions. In some cases it

```
1 if a || b:
2    action()
```

(a)



(b)

Calls to action():
$\langle(1a, EXIT)(2, EXIT), 2\rangle$
$\langle(1a, EXIT)(1b, EXIT)(2, EXIT), 2\rangle$

(c)

Figure 3.4.: (a) A simple program where structural execution indexing is dependent upon the execution path. (b) The CFG with two paths to action(). (c) EPIDs for the call to action().

is too aggressive in achieving this goal and can create different EPIDs *even when execution points meaningfully correspond across executions*. Consider the code in Figure 3.4a. A short-circuiting **or** operation creates the CFG in Figure 3.4b with two branches and two paths to action() on line 2. Note that the paths through the program split based on the values of a and b, but the paths that call action() merge together again before this call. Intuitively, the calls to action() occur at the same execution point even in different executions, so their EPIDs should be the same. In spite of this, because SEI bases EPID construction on the control dependence of the execution point, *different paths to the same point can have different EPIDs*. In this case, as Figure 3.4c shows, one EPID encodes a path where a is **True** and the call is control dependent on $1a$. The other encodes the path where just b is **True**, and the call is depends first on $1b$ which transitively depends on $1a$ [44]. We show later in Section 3.4.3 that this counterintuitive relationship leads to undesirable results for dynamic analyses.

3.1.4 STAT Ordering

The techniques presented thus far have all required either static or dynamic program instrumentation. In some cases however, such as when analyzing a deployed program or when analyzing a program whose behavior changes when it is instrumented, it is necessary to avoid any instrumentation whatsoever. This motivated the EPID technique presented by Ahn et al. as a part of their *Stack Trace Analysis Tool* (STAT) [6]. STAT was designed for debugging high performance computing applications with multiple processes. In order to better classify and group equivalent processes that represented failures, they developed a technique for analyzing core dumps of programs in order to extract the execution point where a program failed. These core dumps are essentially snapshots of program memory and contain not only the call stack of the execution at the point of failure but also the values of all variables on the stack or heap at that point. In addition to producing an EPID from the core dumps, STAT produced a partial ordering of execution points across different executions. This partial order was particularly important in the context of analyzing parallel code that involved multiprocess communication. STAT was the first EPID technique we are aware of that observed that a partial ordering of EPIDs could be useful for analyses.

EPIDs produced by STAT are also stack based, similar to those produced by SEI. However, STAT does not have any of the control dependence or postdominance information used by SEI. Instead, STAT infers as much as possible about the identity of an execution point from the core dump. In particular, EPIDs from stat interleave (1) the call stack of the execution point and (2) values of certain local variables that show the monotonic progress of an execution through loops. The call stack of an execution point can be extracted from the core dump using stack walking methods previously mentioned in Section 3.1.1, but finding variables that show loop progress is more difficult. Indeed, such variables do not even always exist, so STAT makes no guarantee that EPIDs it produces are unambiguous. To approach the problem pragmatically, STAT defines *loop order variable*s (LOVs) that can easily be recognized and extracted as indicators of loop progress when present. LOVs must (1) be defined at least once each iteration, (2) be given strictly increasing or decreasing values over a loop's lifetime, and (3) be given an identical value each particular iteration across all possible executions. Informally, these variables are given a strictly ordered and predefined sequence of values. STAT also defines a static analysis for identifying when these variables are available.

Consider the simple program in Figure 3.5a. This program contains two loops, one that iterates over a fixed range of integers on lines 1 & 2 and another that iterates over a linked list on lines 3 & 4. Suppose that the linked list contains two elements. The EPIDs computed

```
1 for i in range(3):
2   process_int(i)
3 for node in linkedList:
4   process_node(node)
```

(a)

Possible Calls to process_int() and process_node():

$\langle(1, \mathsf{i} \mapsto 0) \to (2, \mathsf{process\_int})\rangle$
$\langle(1, \mathsf{i} \mapsto 1) \to (2, \mathsf{process\_int})\rangle$
$\langle(1, \mathsf{i} \mapsto 2) \to (2, \mathsf{process\_int})\rangle$
$\langle(4, \mathsf{process\_node})\rangle$
$\langle(4, \mathsf{process\_node})\rangle$

(b)

Figure 3.5.: Example of using STAT to identify EPIDs at the calls to process_int() and process_node().

by STAT for each call to process_int() or process_node() are shown in Figure 3.5b. For the first loop, STAT is able to identify that i is a LOV, so its value inside the loop is extracted and included in the EPID of each function call. This makes the EPID for each call to process_int() unique. However, for the second loop, there is no LOV, as the loop iterates over a linked list. As a result, the EPID contains only the call to process_node(), and the EPIDs are ambiguous.

In contrast to previous techniques, STAT does not require program instrumentation and thus does not induce additional overhead on an analyzed application. However, it can only extract an EPID at a location where the program produced a core dump, e.g. a crashing failure. In practice, this meant that STAT was strictly a post-mortem technique; it could not produce EPIDs on the fly as a program was executing. While this limitation can be worked around by explicitly producing core dumps, both the runtime and space overhead of producing core dumps can be prohibitive. Also note that STAT makes use of static analysis for identifying the LOVs whose values it captures. Performing this analysis precisely requires access to the CFG and variable information available at compile time, but it can also be approximated through binary static analysis, thus avoiding the need for any compile time information.

3.1.5 Lightweight Execution Indexing

While SEI offers an approach for computing EPIDs online at runtime, the potential overhead can cause scalability problems and interfere with the program being analyzed. This occurs when loops have multiple guarded exits. Consider the loop in Figure 3.6. SEI pushes a token onto the stack every time lines 1 or 3 execute because they branch the control flow, but those tokens will not be popped off the stack until the loop finishes because the branches are postdominated by a statement outside the loop. In order to avoid the overhead of SEI, some analyses instead use information about the number of times an instruction has been seen within a particular calling context, a particular function invocation, or invocations at a certain depth of the call stack [28, 69]. A canonical example of this is *Lightweight Execution Indexing* (LEI), which was used to identify allocated objects in order to help expose potential deadlocks in concurrent Java programs [69].

```
1 while a:
2   ...
3   if b: break
4   ...
```

(d)

Figure 3.6.: A loop with linear SEI growth.

The approach of LEI is to maintain a counter for each depth of the call stack. This counter keeps track of how many times a particular method has been called at that depth. For instance, the first time that the method foo() is called at a depth of 3 on the stack, its hit counter for the depth 3 is 0. The next time it is called at the depth of 3 on the stack, its hit counter for that depth is 1. The counter for each method at each depth is maintained independently. The LEI for a given execution point then comprises the current calling context along with the hit counts of every call site within the context as well as the hit count and identity of the currently executing statement.

This approach bounds the size of the an EPID to twice the size of the calling context. In addition, it maintains a notion of forward progress through depth counters, and this notion of progress is structured by the call stack. As a result, each EPID is unique and unambiguous within one particular execution. Unfortunately, exactly as with SIC, the values of counters seen in one execution have no guaranteed relationship with the counters seen in other executions. As a result, Lightweight execution indexing can provide EPIDs within one execution, but it cannot provide meaningful EPIDs across executions.

Also similar to SIC, LEI does not inherently require that a program be analyzed or rewritten at compile time. The counters associated with each function and statement of interest at every depth of the call stack can be entirely constructed using dynamic instrumentation without a need for prior planning.

3.2 Precise Execution Point IDs

Dynamic analyses comparing multiple executions are increasingly common [65, 116, 125], so having a robust, efficient EPID technique that works across executions is important. Such *inter-execution* techniques create EPIDs that are only equal when their corresponding execution points are equivalent. Prior work has called this the *execution correspondence* criterion [130]. In spite of this problem's importance, we see that there are only two existing techniques that can provide EPIDs across executions: SEI and STAT. Both techniques have limitations that can prevent them from being practical or useful for particular dynamic analyses. In particular, we desire an inter-execution EPID technique that is:

- **Online** - An analysis should be able to construct the EPID for the current point in the execution and as often over the lifetime of an execution as necessary.
- **Low Overhead** - An execution running with an EPID technique should require as little additional runtime and memory as possible.
- **Scalable** - Neither the duration of an execution nor the size of its workload should significantly affect the runtime or space requirements of the EPID technique.
- **Unambiguous** - Every instruction or statement within an execution should have a unique EPID.

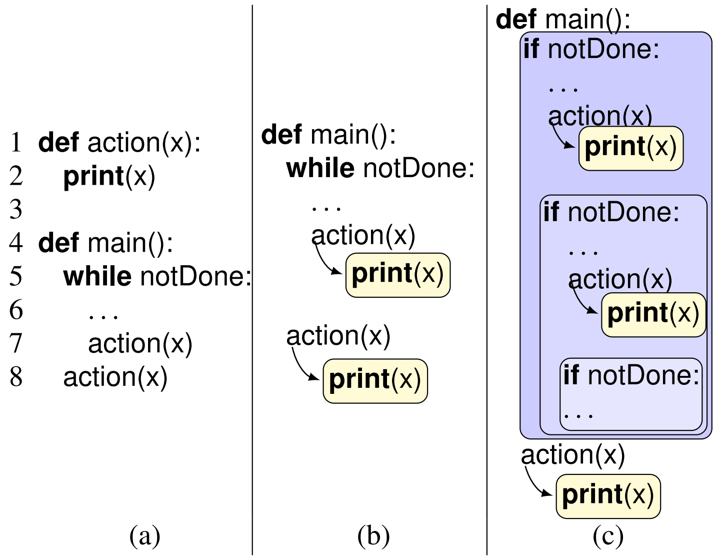
- **Comprehensive** - As a dual to satisfying the execution correspondence criterion, equivalent execution points should also yield equal EPIDs.

Neither SEI nor STAT is able to satisfy all of these requirements. SEI is not low overhead, scalable, or comprehensive, and STAT is not online, unambiguous, or comprehensive. In this section, we introduce a new EPID technique, *Precise Execution Point ID*s (PEPID), that targets all of these criteria. We start by building an intuition about which points *should* correspond across executions in order to provide unambiguity and comprehensiveness. We then devise a technique for computing EPIDs that produces this correspondence efficiently online.

3.2.1 Which Points Correspond?

Because we desire an *inter*-execution EPID technique, we must first decide which execution points should correspond or align across executions. The intuition used by SEI was that the path taken by an execution helped to determine which execution points were equivalent, and SEI used control dependence to codify this relationship. STAT, in contrast, used the intuition that loop control variables captured a notion of forward progress through the loop iterations of an execution. But, as we saw before, control dependence prevents comprehensiveness, and focusing on loop control variables leads to ambiguity. In contrast, we base PEPID on the idea that *execution points at the same position in a sufficiently inlined and unrolled CFG are equivalent*.

Consider a simple program with an acyclic CFG and no function calls. Each instruction inside the program can be executed at most once, so an instruction's position within the CFG can unambiguously identify the instruction within an execution. In addition, the same instruction will trivially have the same EPID across all possible executions, thus guaranteeing comprehensiveness. Unfortunately, this model is unrealistic in general; real programs have both function calls and back edges in their CFGs, both of which can cause instructions to execute more than once and thus introduce ambiguity. However, we can extend the intuition of equivalent points in the CFG to handle those cases as well.



```
1 def action(x):
2   print(x)
3
4 def main():
5   while notDone:
6     ...
7     action(x)
8   action(x)
```

```
def main():
  while notDone:
    ...
    action(x)
      print(x)

  action(x)
    print(x)
```

```
def main():
  if notDone:
    ...
    action(x)
      print(x)
    if notDone:
      ...
      action(x)
        print(x)
      if notDone:
      ...
  action(x)
    print(x)
```

Figure 3.7.: (a) A small program. (b) The program with calls logically inlined. (c) The program with calls inlined and loops unrolled.

First, consider programs that also include function calls. A function may be called from multiple locations, thus executing its body multiple times and making the CFG location an ambiguous EPID. A simple solution to this in most cases would be to inline every function call. If every call were inlined, then function bodies would be duplicated at every call site, once again ensuring uniqueness. Thus, the position of an instruction within this *fully inlined* CFG serves as an unambiguous EPID (ignoring loops). This can be seen in Figure 3.7a-b. This simple program makes calls to action() both inside and outside of the loop. Using the position in the CFG alone would make these calls to **print**() on line 2 ambiguous, however, once action() is inlined, the calls from inside the loop are clearly distinguished from those outside of the loop. Of course, this cannot be done in practice because (1) recursive calls would require an undecidable degree of inlining and (2) inlining every function call would simply increase a program's size too much to be pragmatic. However, we only need to perform this operation *logically* for now. We shall later show that the same correspondence can be computed without actually inlining any functions at all.

Next, we must handle back edges in the CFGs of a program's functions. Back edges create loops or general cycles in a CFG and can thus cause instructions to execute multiple times, again making an instruction's position in the CFG ambiguous as an EPID. One approach used by bounded model checkers is to *unroll* the loops of a program [15, 27]. Each iteration of a loop is peeled of into the guarded body of an **if** statement, and each successive iteration is nested within the body of the preceding iteration. Figure 3.7c illustrates this unrolling in combination with the inlining of function calls. Again, unrolling a loop sufficiently for all executions is not possible in practice, but we shall show that this limitation is irrelevant in the next section.

Using this combination of unrolling and inlining, we are able to define how execution points relate across executions:

Definition 3.2.1 (Alignment) *Given two execution points, p_1 and p_2 from executions e_1 and e_2 of program p respectively, let G be CFG of p sufficiently unrolled and inlined to contain both execution points. Points e_1 and e_2 align iff they occur at the same instruction in G.*

This *alignment* of execution points determines exactly which points are equivalent and must have equal EPIDs even across different executions. Observe, in this transformed program G, execution points p_1 and p_2 can each be performed at most once in any execution, as guaranteed by the acyclic structure of the unrolled and inlined CFG. Thus, the transformed program guarantees that the position in the control flow graph of the program provides an unambiguous EPID, and the control flow graph correspondence maintains comprehensiveness as before. This means that PEPID avoids the problems with SEI presented in Figure 3.4 and Figure 3.6.

3.2.2 Efficiently Computing PEPIDs

As discussed in the last section, inlining all function calls and unrolling all loops is impractical and even undecidable in general, so we must compute this equivalence another way. Instead of actually performing these program transformations, PEPID executes the original program without any extra inlining or unrolling but at the same time keeps track of the inlining and unrolling operations *that would have occurred* in order to identify the

```
INSTRUMENT($P$)
Input:  A program $P$
1: for each loop $l$ in $P$ do
2:     insert pushLoopCounter before the loop header of $l$
3:     insert incrementLoopCounter before loop latches of $l$
4:     insert popLoopCounter on loop exits of $l$.
5: for each call $c$ in $P$ do
6:     insert pushCallSiteID before $c$
7:     insert popCallSiteID after $c$
```

Figure 3.8.: INSTRUMENT takes in a program P and modifies it to maintain a PEPID online. This is the unoptimized instrumentation.

current execution point. We keep track of these operations on an ID stack, similar to those used in SEI and STAT. This stack is then what PEPID uses to produce EPIDs.

In particular, we push an entry onto the stack to identify the call site of every function invocation, popping it as the function returns (or unwinds for exceptional control flow). This tracks the inlining operations for all function invocations. We also need to track all unrolling operations for backedges. We first consider only natural loops, loops with a single entry node or *loop header*, but we extend this to irreducible loops in the next section. We compactly record the unrolling of natural loops by pushing a counter for the loop upon loop entry and popping the counter upon loop exit. We increment the counter upon every iteration of the loop by instrumenting the loop latches, or the edges in the CFG that lead back to the loop header. The stack also naturally handles nested loops.

Figure 3.8 shows a naïve instrumentation algorithm for PEPID. It does not cover exceptional control flow, but we handle exceptions by saving the ID stack height before a call that might throw an exception and pruning the stack to that height if an exception was thrown. Note that the entries in the stack related to inlining and the entries related to unrolling may be maintained independently because they can be unambiguously recombined. This stems from the fact that, given an instruction i, the number of static loops containing i may be readily identified. As a result, a PEPID can be broken down into (1) the inlining ID stack, (2) the unrolling ID stack, and (3) the current instruction ID. Observe, though, the inlining ID stack is precisely equal to the calling context. PCCE already provides a means of encoding the calling context that is more efficient than explicitly pushing and popping at each call site, so we can exploit this to make PEPID computation more efficient. At any point during the execution, a dynamic analysis can call GETCURRENTPEPID() to yield an EPID of the form:

$$\langle \textit{PCCE calling context, unrolling ID stack, current instruction} \rangle$$

This tuple comprises an EPID that provides comprehensiveness and uniqueness based on the prior construction.

Like SEI and STAT, PEPID requires compile-time knowledge about a program. In particular, efficiently computing PCCE calling contexts requires the call graph, and the unrolling stack requires loops to be identified. For programs with only natural loops, it is also relatively compact. The PCCE context is bounded in the worst case by the calling context depth, and the loop unrolling stack is bounded by the number of nested loops that may be active at one time within a program. We show in Section 3.4 that this instrumentation scheme allows PEPID to scale with low overhead.

3.2.3 Handling Irreducible Loops

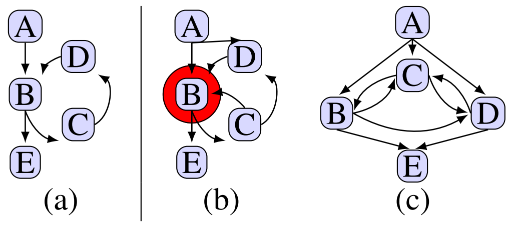


Figure 3.9.: (a) A natural loop. (b-c) Irreducible loops.

Counting iterations is effective for natural loops, which have a unique headers or entry nodes. In that context, unrolling loops is well defined and corresponds to actions upon the unrolling ID stack. Programs can also have unnatural or *irreducible* loops, which have multiple entry points. Indeed, half the SPEC CINT2006 benchmarks have such loops. Figure 3.9 shows some natural and unnatural loops. With multiple headers, distinguishing a loop body from a nested loop is difficult. We use Steensgaard's generalized loop forest recognition to identify irreducible loops and their bodies [112]. Steensgaard's approach is preferable to other loop extraction techniques in that it produces consistent results regardless of how a CFG is traversed [98]. Both (b) and (c) are individual (irreducible) loops under this approach with headers B and D for (b) and B, C, and D for (c).

Sometimes, using an iteration counter can still work for irreducible loops. Given a loop, if there exists a header h of the loop such that every path from each header h' through the loop body back to h' must pass through h, then we say that the header h *naturalizes* the loop. This is because there exists a traversal of the CFG such that every backedge in the loop has h as its destination. Thus, we can use a counter as before and simply increment it on every loop edge that targets h. An alternative intuition is that breaking only edges to h would destroy all cycles in the loop, so a counter incremented on h will uniquely identify instances of this acyclic subregion. Node B in loop (b) is one such naturalizing header. Note that this is just a generalization of natural loops, where the unique header always naturalizes the loop body. We identify naturalizing headers using simple static analysis.

Without a naturalizing header, edges to *multiple* headers must increment the counter to avoid ambiguity. Conservatively, *all* headers may need to increment. This can yield unintuitive results. For example, the path ABDCDCDC in loop (c) would have the EPID $\langle \text{Entry}, \{6\}, C\rangle$ if edges to header nodes increment the loop counter, but so would the path ABCBCBDC. Here, Entry is the calling context, and $\{6\}$ is the unrolling ID stack. Technically, there exists an unrolling of (c) that produces these IDs, but it is unclear how meaningful this is in practice. Alternatively, we can use the same approach as SEI for only this small portion of the program. We push the IDs of predicates in the loop that the headers are control dependent upon and pop them upon their postdominators. This produces the EPIDs $\langle \text{Entry}, \{(B,E)(D,E)(C,E)(D,E)(C,E)(D,E)\}, C\rangle$ and $\langle \text{Entry}, \{(B,E)(C,E)(C,E)(C,E)(B,E)(D,E)\}, C\rangle$, which show the different paths. Both approaches produce unambiguous inter-execution EPIDs. They merely use different approaches for unrolling these degenerate irreducible loops. In fact, an analysis can correctly select either. If overhead is more important, then incrementing on the edges to all headers is preferable. If disambiguating the different paths through these loops is important, then using the localized pushing and popping from SEI is preferable.

3.3 Analytical Comparison

In this section, we examine some of the analytical properties of the different techniques surveyed and how they impact which techniques are preferable in different situations. Figure 3.1 summarizes the results, and we discuss them in detail below.

Availability- Many dynamic analyses require that EPIDs be available online, e.g. for identifying events like allocation or synchronization during an execution. Most of the techniques provide EPIDs online, although STAT does not. However, for analyses that are interested in execution points at the point a program crashes, STAT can still be a useful choice because it alone avoids the need for any program instrumentation.

Requirements & Instrumentation- The requirements and time of instrumentation for the techniques can sometimes create more work for analyses or developers that depend on EPIDs. For example, STAT places the lowest instrumentation burden on users and client analyses because it does not modify the underlying program. As a result, it is easy for STAT to be used with an already compiled program. Because it imposes no overhead, it could even be used on deployed software. Techniques like SIC and LEI that use local counters can be implemented using runtime instrumentation alone, so they also impose little burden on users, but they may not be appropriate for deployed software. Finally, the remaining techniques all require that programs are recompiled with additional static instrumentation. This requires the most work and planning on the part of the developer or client analysis.

Independent of instrumentation, the techniques can also require additional source level information to be precise. PCCE, SEI, and PEPID all require additional compilation information, which is expected since they also require static instrumentation. However, STAT also requires some compile time information in order to identify LOVs. This requirement holds in spite of the fact that STAT performs no instrumentation.

Table 3.1: Analytical properties of the different EPID techniques.

Properties	PCCE	SIC	SEI	STAT	LEI	PEPID
Availability	online	online	online	offline	online	online
Requirements	Call Graph	None	Control Dependence Loops	Loop Order Variables	None	Call Graph Loops
Instrumentation	static	dynamic	static	none	dynamic	static
Ambiguous	yes	no	no	yes*	no	no
Inter-execution	no	no	yes	no*	no	yes
Comprehensive	no	no	no	no*	no	yes
Ordering	none	intra	inter	inter	intra	inter
Space Overhead	O(call stack)	O(1)	O(path length)	none	O(call stack)	O(call stack + unrolling stack)

Ambiguousness & inter-execution IDs- Ambiguous EPIDs do not necessarily confer much information about where an execution point occurs temporally. Thus, ambiguous techniques may be useful for attaching a lightweight notion of local execution context to an execution point, but they cannot be used for more fine grained execution comparison based techniques [125]. Note, though, that while both PCCE and STAT are listed as ambiguous, STAT is unambiguous for programs in which all loops have identifiable LOVs (hence the '*' in the table).

The major differentiating feature of inter-execution techniques is that they are able to align loop iterations across different executions. As a result, techniques that do not track the progress through each loop independently are unable to provide inter-execution IDs. This effectively leaves only SEI and PEPID as viable techniques for analyses requiring such EPIDs. Note, however, that STAT can also provide this under the same assumptions of LOVs as before (*).

Comprehensiveness- One of the large limitations of SEI was that it was not comprehensive. While its EPIDs always established a correspondence across executions, it also created different EPIDs for execution points that did correspond (see Figure 3.4). Note, for programs with LOVs (*), STAT actually *is* comprehensive. However, in contrast to both, PEPID provides comprehensive inter-execution EPIDs in general, making it a preferable choice when instrumentation is possible.

Ordering- Some analyses require that EPIDs be ordered. For example, record and replay techniques require that EPIDs be ordered within one execution (intra) [102]. Some analyses require stricter orders, where EPIDs are partially ordered even across executions (inter) [6, 116, 125]. Most of the techniques are able to provide intra-execution ordering among EPIDs, except for calling contexts with PCCE. SEI, STAT, and PEPID provide stronger inter-execution ordering as well through happens-before relationships among their EPIDs [79].

Space overhead- The size of EPIDs is also an important concern. The required space ranges from none or a constant word, STAT and SIC respectively, to proportional to the length of an execution in the worst case for SEI. All other techniques, however, have EPIDs that grow roughly proportional to the size of the call stack. We examine later how the sizes of the EPIDs produced by these techniques compare in practice.

3.4 Empirical Evaluation

In order to compare these different EPID techniques in practice, we implemented all of them using LLVM 3.2 as a program instrumentation platform and compared them on the SPEC CINT2006 benchmarks. The implementations cover all basic program behavior covered by these benchmarks, including exceptional control flow. In this section, we look closely at the compile time properties as well as the runtime and space overheads induced by these techniques. We conclude by looking at a particular case study that illustrates why comprehensiveness is important in practice.

Note that neither the runtime nor space overhead comparisons include STAT. This is because STAT performs no instrumentation and thus has no overhead. However, the effectiveness of STAT, as discussed in the last section, depends heavily on the ability to produce core dumps and identify LOVs. To gauge whether or not these variables can be found in practice, we compiled the SPEC benchmarks and counted the total number of static loops as well as the number of static loops for which a LOV could be identified. Table 3.2 contains the results.

Overall, a median of 34% of loops had identifiable LOVs across the different benchmarks, and 31% of all loops had such variables. This indicates that relying on LOVs may not be practical in general. However, STAT was originally designed for analyzing high performance computing programs. For programs in that domain, the structure of the programs may make relying on LOVs practical [6].

3.4.1 Runtime Efficiency

For each of the techniques except STAT, we ran the SPEC CINT2006 benchmarks using ‘reference’ workloads 5 times and computed the median and 95% confidence interval for the mean. We ran all experiments on a 64-bit Intel i5 machine with 8GB RAM running

Table 3.2: LOV identification for SPEC CINT2006.

Program	# Loops	# LOVs	% with LOVs
400.perlbench	2151	251	12%
401.bzip2	324	80	25%
403.gcc	7344	1816	25%
429.mcf	57	8	14%
445.gobmk	1444	1090	75%
456.hmmer	425	218	51%
458.sjeng	364	140	38%
462.libquantum	78	60	77%
464.h264ref	1526	1192	78%
471.omnetpp	913	280	31%
473.astar	101	65	64%
483.xalancbmk	8637	1938	22%
total	23364	7138	31%

Ubuntu 13.04. Figure 3.10 presents the normalized median of each technique compared to uninstrumented trials of the benchmark suite. We also present the geometric means of the normalized results for each technique. Error bars indicate the 95% confidence intervals of the means.

PCCE and SIC usually have the lowest overhead on average, 8% and 9% respectively. The next closest is PEPID with 25%, then LEI with 70% and SEI with 314%. We immediately see that in comparison to the other inter-execution technique, SEI, PEPID consistently produces lower overhead. The original SEI paper produced overhead near 42% on average, which differs the results we find. While we used `clang`, SEI used `Diablo/FIT` with link time optimization [121], yielding optimization differences. The original evaluation of SEI also used SPEC CPU95 and CPU2000 benchmarks with smaller workloads than those present in the 2006 benchmarks. When we used the ‘test’ workload, the smallest that SPEC provides, SEI improved to 90% overhead. This illustrates that scalability was indeed a problem for SEI. One of the benchmarks, 471.omnetpp, would not even run using SEI on the reference workload because the stack used for EPIDs consumed all memory and

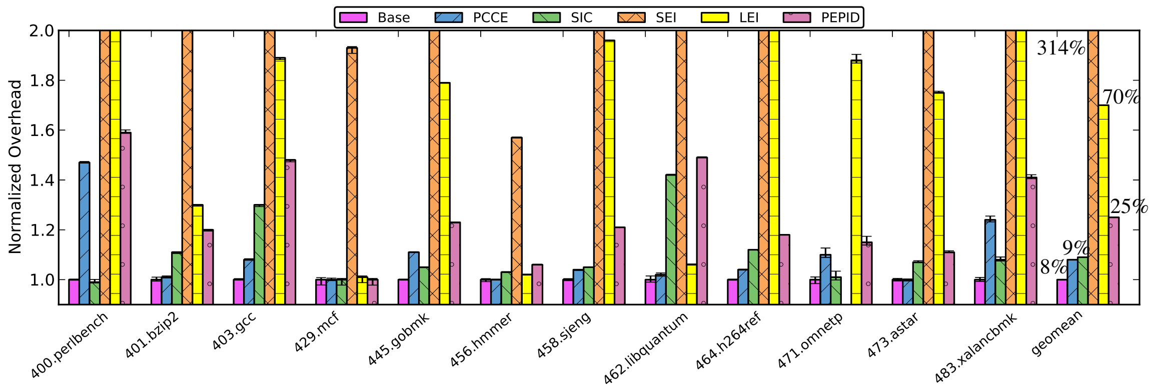


Figure 3.10.: Median runtime overhead normalized against the uninstrumented benchmark of the different EPID techniques on SPEC CINT2006 benchmarks. Error bars show the 95% confidence intervals for the mean of each technique.

crashed the program before completion. In contrast, PEPID's overhead was always closer to SIC and PCCE, in spite of the fact that it provides a more informative form of EPID.

We also note that the original PCCE paper reports overhead closer to 3%. The work used profile guided instrumentation to achieve low runtime overhead, but we did not use profile guided instrumentation in our LLVM based implementation. Also, while we used `clang` to compile programs, PCCE used `gcc`, which optimizes programs differently. This does not affect our comparison because *all* techniques in this chapter were compiled using `clang`. In addition, using profile guided optimizations for PCCE would just strengthen the results of PEPID, since PEPID relies on PCCE as a subtask.

3.4.2 Space Overhead

Maintaining the current EPID consumes memory for each technique except STAT. Table 3.3 lists the maximum memory overhead for each benchmark and technique as well as the mean across all benchmarks. SIC and STAT require a single word or no overhead, respectively, which may be preferable if memory must be conserved. Even though PCCE compactly encodes the calling context, it still takes 51.1KiB on average because some benchmarks have deeply nested calls. For instance, 403.gcc has a maximum depth of 21100

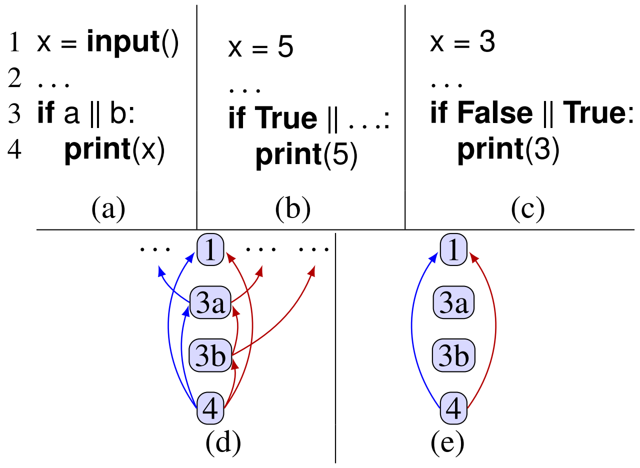
Table 3.3: Worst case memory overhead of EPID techniques.

Program	PCCE	SIC	SEI	STAT	LEI	PEPID
400.perlbench	197KiB	8B	59.8MiB	0	110KiB	262KiB
401.bzip2	8B	8B	238MiB	0	5KiB	112B
403.gcc	165KiB	8B	885MiB	0	1.1MiB	496KiB
429.mcf	232B	8B	626MiB	0	5.3KiB	488B
445.gobmk	2.7KiB	8B	16.3MiB	0	126KiB	5.4KiB
456.hmmer	32B	8B	255KiB	0	6.1KiB	96B
458.sjeng	368B	8B	21.3KiB	0	20.4KiB	856B
462.libquantum	8B	8B	40MiB	0	5.2KiB	48B
464.h264ref	24B	8B	121KiB	0	9.7KiB	168B
471.omnetpp	1.9KiB	8B	>7GiB	0	22.4KiB	1.9KiB
473.astar	8B	8B	1.3MiB	0	5.6KiB	64B
483.xalancbmk	246KiB	8B	2.86GiB	0	12.6MiB	431KiB
mean	51.1KiB	8B	436MiB	0	1.2MiB	99.9KiB

calls. Profile guided instrumentation can help reduce this. However, even the worst case overhead of PEPID, which uses PCCE, is relatively low, around 100 KiB on average. It is almost always smaller than LEI and is orders of magnitude smaller than SEI in spite of its precision. This makes PEPID a preferable technique for analyses needing inter-execution EPIDs.

3.4.3 Client Impact

We now show how a comprehensive technique like PEPID is preferred over a non-comprehensive technique like SEI for a particular dynamic analysis. We consider an analysis known as dual slicing. Dual slicing is a backward slicing technique that contrasts two executions [125]. Instead of including all backward dependences for a slice criterion, it includes only those dependences that either (1) exist in only one of the executions or (2) exist in both executions but define different values. In this way, dual slicing produces a notion of explanation for why two executions differ, which can be useful for debugging [125] or for security analysis [65]. Backward slicing techniques traditionally include too many



```
1 x = input()
2 ...
3 if a || b:
4   print(x)
```

(a)

```
x = 5
...
if True || ...:
  print(5)
```

(b)

```
x = 3
...
if False || True:
  print(3)
```

(c)

Figure 3.11.: (a) A program that can lead to bad dual slices using SEI. (b) A trace where a is **True**. (c) A trace where b is **True**. (d) A dual slice using SEI. (e) A dual slice using PEPID.

dependences to be practical [135], so dual slicing is particularly useful because it prunes away irrelevant dependences as it contrasts two executions.

EPID techniques like SEI form the foundation of dual slicing. EPIDs determine whether a dependence in one execution exists in another. Unfortunately, when noncomprehensive EPIDs are used, they can include unnecessary dependences in the slice, defeating one of the main goals of the technique.

Consider the program in Figure 3.11a. This program reads an integer x from the user and prints it if either a or b is **True**. Suppose there are two different executions of the program, one where the program prints 5, and the other prints 3 as shown in Figure 3.11b-c. Note that a is **True** in one execution, but only b is **True** in the other. This matches the case we considered earlier in Section 3.1.3, meaning that the print statements in the two executions have different EPIDs under SEI. Because the EPIDs differ, dual slicing considers them different statements and also includes their control dependences. The dual slice includes the different values of a and b via control dependence, even though they do not actually affect the output differences. These irrelevant dependences get in the way and impede the user's ability to understand why the executions printed different numbers as shown in Figure 3.11d. Here, the arrows denote dependences in the dual slice. In contrast, a comprehensive technique like PEPID is able to identify that the print statements occur at the same execution point and identify that the differing user input for `x` caused the different output. Figure 3.11e shows the dual slice when using PEPID and clearly identifies how the input difference directly caused the output difference.

3.5 Related Work

We examined several approaches from literature that compute EPIDs for dynamic analyses [6, 69, 86, 117, 130]. Each of these techniques has been used to solve real problems in dynamic analysis ranging from informing replay techniques [102] to fine-grained execution comparison [125]. The comparison of these techniques along with our new EPID computation technique, PEPID, is one of the core contributions of this work.

In developing PEPID, we based our system around the notion that the position within an unwound and unrolled CFG provides a notion of identity for execution points. This was inspired in part by bounded model checking [27], but model checkers do not need to consider the alternative high-level semantics for unrolling degenerate irreducible loops. Similar notions of identifying execution points also exist within static analysis, where k-CFA provides a statically bounded approximation of execution points using a similar intuition [109].

3.6 Conclusion

In this chapter, we examined several techniques for computing execution point IDs (EPIDs) and considered their strengths, weaknesses, and limitations. To address limitations of inter-execution EPIDs, we introduced a new technique, PEPID, that is able to comprehensively compute inter-execution EPIDs with significantly less space and runtime overhead than existing techniques. PEPID also produces more meaningful relationships between EPIDs in different executions. We also showed that establishing these meaningful relationships is useful in the context of real world dynamic analyses. PEPID forms a foundation for contrasting instructions across two executions and thus drives the debugging techniques presented within this dissertation.

4 IDENTIFYING AND REPLACING MEMORY ACROSS EXECUTIONS

The SEI and PEPID introduced in Chapter 3 provide ways to identify the instructions that align across different executions. This alignment allows analyses to compare different executions. Comparing executions is a fundamental challenge in dynamic program analysis with a wide range of applications. For instance, comparing executions of two versions of a program with the same input can be used to isolate regression faults [57] and analyze the impact of code changes [99]. Comparing program state at different points within executions can also be used to normalize and cluster execution traces, simplifying analyses that use those traces as input [37]. Comparison also provides unique advantages in program deobfuscation [42] and debugging compiler optimizations, where aggressive transformations make static comparison less effective. Two executions from the same concurrent program can be generated with schedule perturbations to confirm harmful data races [130], and real deadlocks [70].

However, EPIDs only solve one dimension of the problem – aligning corresponding instructions or control flow. The algorithm in Figure 2.2 requires more. The other unsolved dimension, orthogonal to control flow, is memory. In the presence of program differences, input differences, or non-determinism, corresponding memory regions or structures in the heap are allocated in different places across runs. Therefore, although executions are aligned along control flow paths, if memory regions are not also aligned, comparing the values of variables is not meaningful.

Existing techniques rely on sub-optimal solutions [99, 114, 132], such as identifying memory using symbolic names. In particular, to compare memory states of two executions, reference graphs [17] are first constructed in which global and local variables are roots, and memory regions, especially heap regions, are connected by reference edges. Roots align by their symbolic names, and other memory regions align by their reference paths, which consist of variable and field names. We call this approach *symbolic alignment*. However, symbolic alignment of memory is problematic in the presence of aliasing. A detailed discussion of the issue can be found in Section 4.1.

In this chapter, we propose a technique called *memory indexing* (MI). The central idea is to canonicalize memory addresses such that each memory location is associated with a canonical value called its *memory index*. Memory locations across multiple executions align according to their indices. Pointers are compared by comparing the indices of their values. Memory indices are maintained along an execution such that they can be directly accessible or computable.

Overall, we make the following contributions.

- We formally present the memory indexing problem. We identify key properties of valid solutions.
- We discuss two semantics for memory indexing. The first one is an online semantics that computes indices on the fly during execution and handles pointer arithmetic. The second is a lazy semantics that computes indices on demand. It has lower cost and is more suitable for languages without pointer arithmetic.
- We introduce a practical design that uses a tree to allow multiple indices to share their common parts. Optimizations remove redundant tree construction and maintenance.
- We illustrate how memory indexing facilitates computing cause transitions for failures. Novel memory comparison and substitution primitives resolve limitations of existing solutions. They allow robust mutation of a passing run to a failing run by copying state across runs.
- We evaluate the proposed MI scheme. It causes a 41% slow down and 213% space overhead on average. The results of two client studies show that MI is able to canonicalize address traces across runs, and it scales cause transition computation to programs with complex heap structures.

The ideas presented in this chapter have been previously published by the author in the proceedings of FSE 2010 [115].

4.1 Motivation

Execution comparison not only requires alignment of the control flow of executions, but also the memory. Aligning and comparing memory snapshots across runs is a key challenge. Previous techniques do not provide satisfactory solutions to the following challenges.

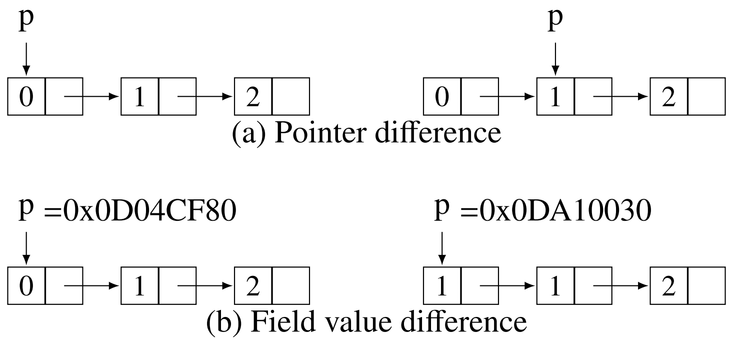


Figure 4.1.: Pointer comparison. Linked lists represent the snapshots of different executions. Each node has two fields: *val* and *next*.

Support for Pointer Comparison. Many applications require the ability to compare pointers across runs. For example, regression debugging [57] and computing cause transitions [132] rely on contrasting variable values in a passing run and a failing run to identify faulty values. For pointer related failures, it is critical to identify when a pointer contains a faulty value. However, due to semantic differences or non-determinism, even pointers that point to the same data structure can have different values across runs, so they are not directly comparable. Most existing techniques do not support pointer comparison. Instead, non-pointer field values, such as $p \rightarrow val$ and $p \rightarrow next \rightarrow val$ in Figure 4.1, are compared following their symbolic reference paths. For the case in Figure 4.1 (b), such comparison yields the right result. That is, only $p \rightarrow val$ has different values across executions. Whereas in case (a), the conclusion is that $p \rightarrow val$ and $p \rightarrow next \rightarrow val$ have different values, implying the definitions to these fields are faulty in a debugging application, which is not true. A more appropriate conclusion is that only the pointer *p* has different values, all other differences are manifestations of the pointer difference.

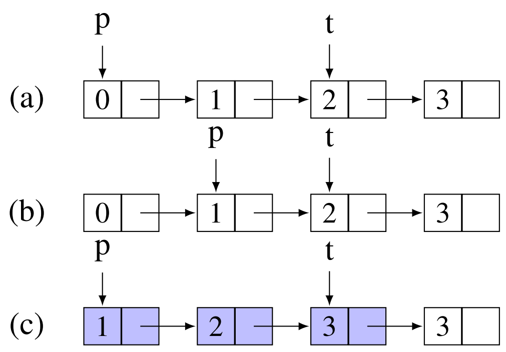


Figure 4.2.: Destructive state mutation. (a) Snapshot in run one. (b) In run two. (c) Mutating (a) to (b).

Destructive State Mutation. Uses of execution comparison such as computing cause transitions [132] compare memory snapshots from a passing run and a failing run. A variable having different values in the two respective runs is called a *difference*. In order to reason about the causal relevance of differences with the failure, values of difference subsets are copied from the failing run to the passing run to see if the failure is eventually triggered in the mutated passing run. However, using symbolic alignment causes a *destructive mutation* problem in the presence of aliasing. In particular, a memory location may have multiple reference paths. It may be classified as a difference when it is compared under one path but not along another path. Mutation along one path destroys the semantic constraints along other paths and can lead to undesirable effects. Consider the example in Figure 4.2. Two snapshots are shown in (a) and (b) with p pointing to different locations in each. With symbolic alignment, the root p is aligned first, followed by nodes along paths from p. As a result, the first node in (a) is aligned with the second node in (b), the second node in (a) with the third node in (b), and so on. Comparing the non-pointer fields of the aligned nodes yields the following reference paths denoting differences: $p \to val$, $p \to next \to val$ and $p \to next \to next \to val$. They are fields in (b) having values different than those in (a). Suppose we try to mutate (a) to (b) by copying values from (b) to (a) following the paths of differences. The resulting state is shown in (c). Observe that t's value is undesirably destroyed.

Lost Mutation. If multiple differences alias, they may result in *lost mutation* when they are applied together. Specifically, the differences applied earlier may be overwritten undesirably by differences applied later. This may lead to incorrect conclusions about the relevance of differences. Consider the example in Figure 4.3. Assume the failure is that $(p \rightarrow val) + (t \rightarrow val)$ has the wrong value. Pointer t points to the wrong place, and the node pointed to by p has the wrong value. These together cause the failure. Symbolic alignment and non-pointer value comparison identifies two differences denoted by their reference paths: $p \rightarrow val$ and $t \rightarrow val$. However, as p and t alias in (a), when the differences are applied to (a) in the order of p first and then t, the rightmost leaf first has the value 2 and then 1. The mutated state does not lead to the expected failure. Hence, we mistakenly conclude that the two differences are not relevant to the failure.

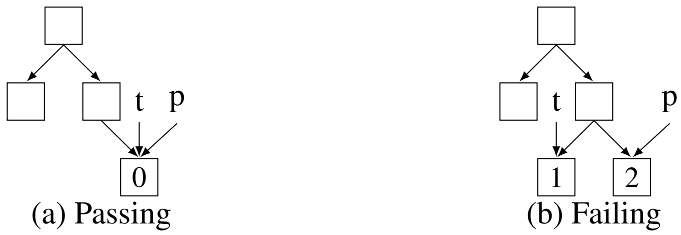


Figure 4.3.: Lost mutation.

4.2 Problem Statement and Overview

To overcome the aforementioned problems and provide robust support for memory comparison and mutation, we propose a novel technique called *memory indexing*. The basic challenge is to *associate each memory location with a canonical value such that locations across runs are aligned by their canonical values; pointers can be compared by their canonical values*. Such values are also called *memory indices* because they essentially provide an indexing structure for memory.

The idea is illustrated by Figure 4.4, which revisits the example in Figure 4.3. Focus on the parts inside the boxes for now. Each node is associated with a canonical value (index) circled at a corner. Nodes are aligned by their indices. Hence, we can see the root nodes

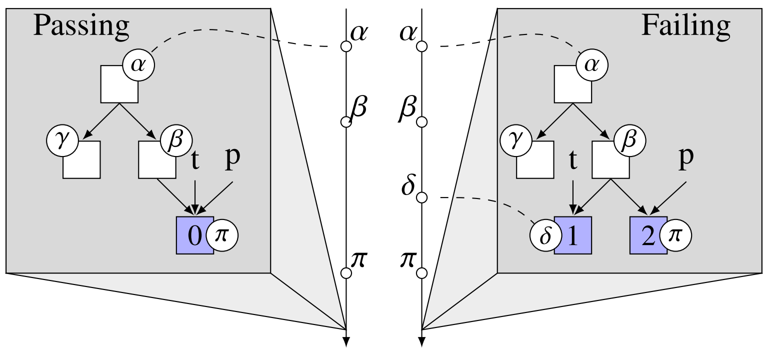


Figure 4.4.: Overview of memory indexing.

align as they have the same index α. The node with index δ on the right does not align with any node on the left. Besides its concrete value, pointer p also has a canonical value π in both runs. Pointer t has π on the left but δ on the right. With memory indexing, the differences of the two states can be correctly identified: pointer t has a different pointer value and the nodes pointed to by p have different field values. When mutation occurs, t is set to the location with index δ, which is not present in the passing run and thus requires allocation. p's field value is changed to 2. Such mutation properly induces the failure.

A valid memory indexing scheme should have the following property: *at any execution point, each live memory location must have a unique index.* We call this the *uniqueness property*. If this property is not satisfied, multiple locations may share the same index or one location may have multiple indices, which makes proper alignment across runs impossible. Symbolic alignment does not always satisfy this property and is thus not a good indexing scheme.

A good indexing scheme should have the following additional feature: *locations across runs that semantically correspond to each other should share the same index*. We call this property *alignment*. Using the concrete address of a memory location is an indexing scheme that provides uniqueness, but it does not deliver good alignment.

Inappropriateness of Graph Matching. Finding the most appropriate memory alignment concerns program semantics and is thus, in general, not a concretely knowable problem. As pointed out in [132], one possible approximate solution in the general case is to represent the memory snapshots under comparison as reference graphs and formulate the alignment problem as a graph matching problem [20]; the goal of which is to produce a match with the minimal number of graph differences. However, this solution is too expensive (NP complexity) to be practical [132]. More important, we observe that it fails to deliver desirable alignment in many cases because it does not capture semantic differences. Consider the example in Figure 4.5. The failure in (b) occurs because the value field passed to the node constructor is incremented by one. With a graph matching algorithm, to minimize graph differences, the second node in (a) aligns with the first node in (b), the third node in (a) aligns with the second node in (b), and so on. As highlighted in the figure, the graph differences, namely the graph operations needed to mutate (a) to (b), are: add (b)'s tail to (a); add the edge to the added node; remove the head in (a). However, such differences imply that the shape of the linked list is faulty, which is not true. The most appropriate alignment matches the corresponding nodes in the lists, resulting in four field value differences that precisely reflect the semantic differences.

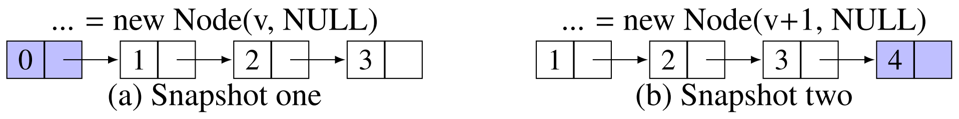


Figure 4.5.: Graph matching may be undesirable.

Our Indexing Scheme. We propose to use *the execution point where the allocation of a memory region occurs as the index of the region.* We leverage the observation that semantic equivalence between executions often manifests itself through control flow, as used by EPIDs. Hence, semantically equivalent memory regions are often allocated at corresponding execution points. Figure 4.4 presents an overview. The two lines in the middle represent the control flows of the executions. The memory indices of regions in memory are essentially canonical representations of the allocation points. For instance, the root nodes share the memory index α, indicating they are allocated at the same point α. In contrast, index δ's presence in only the failing run means that the allocation does not occur in the passing run. Our indexing scheme satisfies the uniqueness property and provides high quality alignment of memory regions in practice.

4.3 Semantics

In this section, we present two semantics for memory indexing. The first is for low level languages such as C. It supports pointer arithmetic by updating indices on the fly. This is called the *online semantics*. The other semantics computes indices on demand and does not need interpretation of pointer arithmetic. We call this the *lazy semantics*.

Our semantics canonically represents each memory location by a pair (*region*, *offset*), with *region* as the canonical representation its containing allocated region and *offset* as its offset inside the region. The canonical representation of a region is captured when the region is allocated and serves as a birthmark of the region during its lifetime. We provide a function MI() that maps a concrete address to its index. We also maintain a function PV() that maps a pointer variable to the index of the value stored in the pointer. In the lazy semantics, PV(p) is lazily computed from MI(p), whereas in the online semantics PV(p) is updated on the fly through pointer manipulations on pointer p. Hence, PV(p) may be different than MI(p) in the online semantics. As we later show, separating PV from MI allows us to precisely handle pointer arithmetic, which is desirable for certain uses.

4.3.1 Online Semantics

In this subsection, we discuss the online semantics for memory indexing.

Indexing Global Memory. We consider global memory locations as part of a *global region*. Hence, the memory index of a global location is its offset in the global region (Rule 5 of Figure 4.6). In our terminology, &g denotes the concrete address of a variable g. If executions from different program versions are considered, e.g. in comparing regressing executions, symbolic names of variables are used instead of their offsets. It is easy to see the uniqueness property is satisfied.

Rule	Event	Instrumentation
(5)	Prog. starts	for each global variable g: MI(&g)= (nil, `global_offset`(g))
(6)	Enter proc. X	for each local variable lv of X: MI(&lv)= (CS, `local_offset`(lv))

(7)	pc: p = `malloc`(s)	for i=0 to s-1:
		MI($p + i$)= ([EPID, pc], i)
		PV(p)= MI(p)

(8)	$p = \&v$	PV(p)= MI(&v)
(9)	$p = q$	PV(p)= PV(q)
(10)	$p = q \pm offset$	PV(p)=(PV(q).first, PV(q).second$\pm$ $offset$)

Figure 4.6.: Online semantics. A memory index MI(a) represents the memory index of an address a, which is a pair comprising a region identifier and an offset. CS represents the current call stack. pc represents the program counter. EPID represents the EPID of the current execution point. PV(p) represents the memory index of the address value stored in p.

Indexing Stack Memory. We consider stack memory to be allocated upon function entry. The allocated region is the stack frame of the function. Hence, we use a stack frame identifier and the stack frame offset of a location to represent its index. Recursive calls allow multiple instances of the same function to exist in the call stack at an execution point so that we have to use the call path of a stack frame as its id. Such stack indices trivially satisfy the uniqueness property and provide meaningful alignment. This is presented in Rule 6 of Figure 4.6. Some programs perform dynamic allocation on the stack, which makes stack variables have varying offsets. We identify such variables through static analysis and use our own IDs to replace the offsets.

Indexing Heap Memory. The essence of our technique is to create a birthmark of a memory location as its canonical representation. The birthmarks of heap locations are more tricky. Using the program counter (PC) of the allocation point is not sufficient because multiple live heap regions may be allocated at the same PC. The calling context of the allocation point is not sufficient either. For example, the code in Figure 4.7 (a) creates a linked list in the loop on lines 1 and 2. All allocations occur in the same context (statement

2 inside `F()`). Adding an instance count does not help either because different executions may take different paths so that the same count does not imply correspondence.

```
    def F():
 1    for i in 0 to 2:
 2      h = new Node(..., next=h)
 3    p = r = h
 4    C = (p−>val > 0)  # Should be >= 0
 5    if C:
 6      r = r−>next
 7      h = new Node(val=0, next=h)
 8    else:
 9      p = p−>next
10    p = p−>next
11    print(p−>val + r−>val)
```

(a) Code

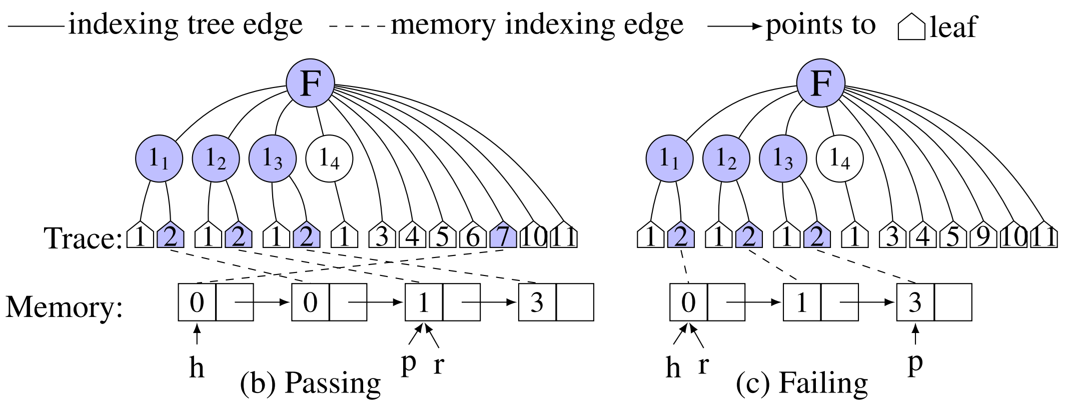


Figure 4.7.: Example for heap indexing. The code constructs a linked list of three nodes with values of 0, 1 and 3. Initially, the three pointers *h*, *p*, and *r* all point to the head of list. There is a regression bug at line 4 in computing the predicate. As a result, the failing run takes the false branch, making *p* point to its second node. Pointer *p* further advances to the third node at line 10. In contrast, the passing run takes the true branch, eventually resulting in both *p* and *r* pointing to its third node. The failure is observably wrong output. The memory snapshots are before the failure at statement 11.

To index heap memory, we use the EPID of the allocation point as the id of an allocated region to compose the memory index. The uniqueness of EPIDs ensures the uniqueness of heap indices. The alignment of the memory indexing scheme also originates from the fact that EPIDs identify equivalent allocation points across executions. In particular, heap indices are set when a region is allocated (Rule 7 in Figure 4.6). A heap index consists of the current EPID and the allocation site *pc*. Besides setting the memory indices, the rule also sets the canonical value of the pointer variable, i.e. PV(p), to the memory index of the head of the region. Such a canonical value will be used in pointer manipulation. For example, in the second iteration of the loop in Figure 4.7, after the allocation in statement 2, PV(h)= ($\langle$F, $\{2\}$, 2$\rangle$, 0)

Memory locations across multiple runs are aligned by their indices. By this criterion, in Figure 4.7, the head of the list in (b) does not align with anything, but the remaining three nodes align with the list in (c).

A key feature of memory indexing is pointer value comparison across runs. Besides a concrete memory address, a pointer variable is also associated with a canonical value. Canonical pointer values are updated on the fly in the online semantics, as specified by Rules 8-10. For brevity, we assume a simple syntax for pointer operations. In particular, if the address of a variable v is retrieved and assigned to a pointer, the canonical value of the pointer is the memory index of v's address (Rule 8). If a pointer variable is copied to another variable, the canonical value gets copied too (Rule 9). For pointer arithmetic expressions $p = q \pm offset$, variable p's canonical value is computed by copying the region identifier of q and adding r to the offset of q (Rule 10). For brevity, our semantics assumes type information has been processed so that offset variables are identified at the unit of bytes.

4.3.2 Lazy Semantics

For high level languages in which pointer arithmetic is not permitted, or when client applications do not require considering the effects of pointer arithmetic, we can derive PV values on demand to allow a more efficient implementation. The semantics is called the *lazy semantics*. The observation is that canonical values of pointers can be lazily inferred from their concrete values. That is, given a pointer p, PV(p)=MI(p). Recall that in the online semantics, PV is computed by interpreting pointer arithmetic (Rule 10) and hence PV(p) is not necessarily equivalent to MI(p).

Rule	Event	Instrumentation
(11)	*pc*: *p* = `malloc`(s)	MI(*p*)=([EPID, *pc*], 0)

(12)	Query the index of heap address *a*	*t* = *a*
		while (MI(*t*)≡ nil) *t*=*t*-1
		return (MI(*t*).first, *a*-*t*)

Figure 4.8.: Lazy semantics.

The new rules are presented in Figure 4.8. On the fly computation is only needed upon heap allocation (Rule 11): the current EPID is assigned to the region base address, but not to the other cells in the region. When the MI value of a heap address is queried (Rule 12), the algorithm scans backwards from the given address to find the first address with a non-empty index. Alternatively, a binary search tree can be used to track the addresses with indices. The memory index of the given address consists of the region denoted by the non-empty index and the offset inside the region. For large heap regions, we can set the MI for a number of addresses at set intervals besides the base address such that a linear scan can quickly encounter a non-empty MI. No on-the-fly computation is needed for global and stack memory. The MI values of global and stack addresses can similarly be computed on demand.

Precision Lost in the Lazy Semantics. In languages with pointer arithmetic, the lazy semantics does not instrument pointer operations or track the original regions of pointers. The looser coupling with program semantics may lead to undesirable imprecision in certain applications.

```
def F():
   s = 100  # should be 500
   A = malloc(s)
   B = malloc(200)
   ...
   p = A
   p = p + 200
```

Figure 4.9.: The advantage of the online semantics.

Consider the example in Figure 4.9. It is a simplification of a real bug in `bc-1.06`. Assume there is a regression error in the program; variable s should be 500 whereas it is 100 in the faulty version. For simplicity, we also assume the A and B regions are adjacent in memory. In the failing run, buffer A has size 100 and pointer p points to a location in B due to overflow, although it originally points to A. According to the lazy semantics, at the end of the failing execution, PV(p)= MI(p)=($\langle F, \{\}, 3\rangle$, 100), which is offset 100 in the B region. In the passing run, PV(p)= ($\langle F, \{\}, 2\rangle$, 200), which is offset 200 in the A region and hence pointer p is considered a difference. In contrast, following the online semantics, both the passing and the failing runs have PV(p)= ($\langle F, \{\}, 2\rangle$, 200), and hence p is not considered a difference. Instead, the online semantics only reports variable s to be a difference, which precisely reflects that the program is faulty in the allocation size instead of the pointer arithmetic.

In practice, one can choose the right semantics based on the application. For instance, the online semantics is more desirable when out-of-bound accesses are involved, such as when debugging a segmentation error. It also handles dangling pointers better because a PV value has the same lifetime as the pointer regardless of the status of the deallocated memory, whereas in the lazy semantics a dangling pointer is no longer dangling when the memory is re-allocated.

4.4 Design and Optimizations

The semantics in the previous section are conceptual. They model an index as a sequence of symbols (the region) and an integer (the offset). This is too expensive to operate with in practice. In our design, we explicitly maintain an index tree for heap memory and represent a heap region as a reference to some leaf in the tree. The full index of a heap location can be acquired by traversing bottom-up from the leaf. Rules 15 and 16 show the tree based instrumentation for the lazy semantics. Upon heap allocation (Rule 15), a leaf node representing the allocation is created and inserted into the tree by calling *Tree_Insert()*. The function first checks if the current EPID is part of the tree. If not, it adds the EPID to the tree before it inserts the leaf node. At the end, the instrumentation sets the MI of the region base address to the leaf. Upon deallocation (Rule 16), *Tree_Remove()* is called with the leaf node corresponding to the to-be-freed region. We perform recursive tree elimination, meaning that removing a leaf node may lead to removing its ancestors if they have no other children. Shaded subtrees in Figure 4.7 are example heap index trees. Dotted edges link leaf nodes to memory regions. We have the following optimizations to make our design practical.

Rule	Event	Instrumentation
(15)	*pc*: *p* = `malloc`(s)	*l*=new Leaf(*pc*, *p*, *s*) *Tree_Insert* (EPID, *l*) MI(*p*)=(*l*, 0)
(16)	`free` (*p*)	*Tree_Remove* (MI(*p*).first)

Figure 4.10.: Tree based indexing in lazy semantics.

Removing Redundant Instrumentation. We have two observations that help remove redundant instrumentation. The first one is that we only need a partial indexing tree to index heap allocations. Hence we can *avoid instrumentation that maintains irrelevant EPIDs.* If a function does not allocate heap memory, it is not necessary to compute EPIDs inside that functions. More formally, a function or a loop is *relevant* to heap allocation if and only if a heap allocation can directly or transitively occur in its body. Irrelevant functions and loops are not instrumented.

The second observation is that we do not need to instrument all relevant functions or loops. More specifically, given a relevant function other than `main` (loop) *n*, if all index paths from any of *n*'s parents to a heap allocation inside *n*'s body have to go through *n*, we don't need to instrument *n*. We call *n* a *dominant* function (loop). Intuitively, we do not need to instrument if we can infer the presence of *n* on an index path given the allocation site and the parent node. We have developed static analyses to identify relevant but not dominant functions and predicates. They are analyses on call graphs and control flow graphs. Details are elided. Note that such optimizations are not applicable to general computation of EPIDs because they leverage heap allocation information.

The space consumption is dominated by the tree, whose size depends on its shape and the number of live heap regions. A pessimistic bound is O(*maximum tree depth* × *maximum live heap regions*). In theory, the tree depth is unbounded because it is tied to the depth of recursion. In practice, because we are only interested in the partial tree for allocations, the tree depth is well bounded such that the space overhead is feasible (see Section 4.6).

4.5 Robust Memory Comparison and Replacement

Cause transition computation [28, 132] produces a possible causal explanation for a software failure. The technique takes two executions: one failing and the other passing that closely resembles the failing one. The passing run can be generated by selecting an input similar to the failing input. The overall idea is to compare memory snapshots of the two runs at selected execution points. The technique constructs a reference graph [17] to represent a snapshot and reduces memory comparison to graph comparison driven by symbolic reference paths. It then performs causality testing to isolate a minimal subset of graph differences relevant to the failure. More specifically, it enumerates subsets of graph differences through the delta debugging algorithm. A subset is considered relevant if *replacing the program state specified by the subset in the passing run with the corresponding values in the failing run produces the failure.* The minimal subsets computed at the selected execution points are chained together to form an explanation. Memory comparison and replacement is driven by symbolic paths, so it faces the issues mentioned in Section 4.1.

Consider the example in Figure 4.7. Using symbolic alignment and comparing only non-pointer values, if only heap memory is considered, the set of differences (failing - passing) $\Delta=\{p \rightarrow val, r \rightarrow val, r \rightarrow next \rightarrow val, r \rightarrow next \rightarrow next \rightarrow val, h \rightarrow next \rightarrow val, h \rightarrow next \rightarrow next \rightarrow val\}$. None of the subsets, including the Δ set itself, can induce the same failure. For instance, applying subset $\{p \rightarrow val, r \rightarrow val\}$ does not work due to the lost mutation problem. As a result, the delta debugging algorithm terminates without finding the minimal failure inducing subset. Since aliasing is very common in general programs, we need to perform robust memory comparison and replacement.

With MI, we are able to develop two robust primitives: comparison of memory snapshots with MEM_COMP() and application of a memory difference with DIFF_APPLY(), i.e., copying a value from one memory snapshot to the other across executions.

For the comparison primitive, we first align snapshots via their indices and then conduct value comparison at aligned memory locations. Memory locations with non-pointer types are compared by their concrete values. Locations with pointer types are compared by their canonicalized values. Differences are presented as a set of indices, denoting that the corresponding locations are different.

Consider the two snapshots in Figure 4.7. Global variables C, h, p, and r align across the executions. Since C has a boolean type, we compare its values and classify them as differences. In contrast, we conduct canonical value comparison for pointer variables h, p and r. It is easy to see that they are different. We compare heap memory is compared via the index trees. The region pointed to by h in (b) is identified as the only tree difference. Hence, if we compute the difference set (passing - failing), the result is {(nil, offset(C)), (nil, offset(h)), (nil, offset(p)), (nil, offset(r)), ($\langle F, \{\}, 7\rangle$, *)}. The symbol '*' in the last index signifies that the entire region is different. It is smaller than the symbolic results.

The second primitive is the application of a unit difference[1] represented as an index, from which the corresponding concrete memory location in both snapshots can be identified. The value is copied from the source snapshot to the target snapshot. If the value is a pointer, we cannot simply copy the concrete address. Instead, we identify the proper concrete address in the target snapshot following the canonical value of the pointer. If the region is not present in the target snapshot, it is first allocated.

Function DIFF_APPLY() in Figure 4.11 describes how to apply a heap unit difference. In the algorithm, the source and target heaps are indexed by trees rooted at T' and T, respectively. Variable δ represents the unit difference. Lines 3 and 4 identify the heap region denoted by δ in T' and T. In lines 5 and 6, the concrete addresses are computed.

[1]A unit difference is a difference regarding a specific memory location instead of a region.

DIFF_APPLY(T, T', δ)

Input:	T - source execution T' - target execution δ - difference to apply
Description:	Copy the value in location δ from T' to T. Leaf node is of the type (*pc*, *base*, *size*).

```
1: let $\delta$ be ($path$, $offset$)
2: let (-, $base'$, -) be the leaf node in $T'$ along $path$
3: let (-, $base$, -) be the leaf node in $T$ along $path$
4: $a \leftarrow base + offset$
5: $a' \leftarrow base' + offset$
6: if $*(a')^{T'}$ is NOT a pointer then
7:     $*(a)^{T} \leftarrow *(a')^{T'}$
8: else
9:     let $\mathrm{PV}(a')^{T'}$ be ($p'$, $f'$)
10:    if $T$ does not have path $p'$ then
11:        REGION_COPY ($T$, $T'$, $p'$)
12:    let (-, $b$, -) be the leaf node in $T$ following $p'$
13:    $*(a)^{T} \leftarrow b + f'$
```

REGION_COPY(T, T', $path'$)

Input:	T - source execution T' - target execution $path'$ - region to copy
Description:	Copy region $path'$ in T' to T.

```
1: let (-, $base'$, $size'$) be the leaf in $T'$ along $path'$
2: $r \leftarrow$ allocate($size'$) in the run denoted by $T$
3: insert $path'$ to $T$
4: set the leaf node following $path'$ in $T$ to (-, $r$, $size'$)
5: for $i$=0 to $size' - 1$ do
6:     DIFF_APPLY($T$, $T'$, ($path'$,$i$))
```

Figure 4.11.: Apply a heap difference.

At line 7, the algorithm tests if the computed address is a pointer (the superscript specifies where the dereference occurs). If not, the algorithm copies the value (line 8). If so, it tests if the region pointed-to is present in T (line 11). If not, it copies the region (line 12). Finally at line 14, the concrete address stored to the pointer is set to a location in the region (in T) aligned with the source region (in T').

Function REGION_COPY() copies a region denoted by the parameter $path'$ from T' to T. It first locates the region in T' (line 2) and allocates a region of the same size in T (line 3). The $path'$ is inserted to T and a leaf node is created to represent the allocated region (lines 4-5). This avoids allocating the same region again. Finally, individual fields are copied from T' to T by calling DIFF_APPLY() (lines 6-7). Note that this process may transitively copy more regions from T' to T through line 11 of DIFF_APPLY().

Applying stack and global differences is similarly defined.

Example. Consider the example in Figure 4.7. Assume we want to apply the differences of p and r to the passing run. Observe that p points to the third node in the failing run and PV$(p)^{fail}$= $(\langle F, \{2\}, 2\rangle, 0)$. During the p difference application, following the path, the concrete address of the fourth node *in the passing* run is identified and assigned to p. Similarly, after applying the r difference, r holds the concrete address of the second node in the passing run. Note that, by applying these two differences, the same failure can be produced. Applying other differences, such as C, at this point (before statement 11) has no impact on the failure. The minimal failure inducing difference subset including p and r is emitted as one cause transition.

The same memory comparison and difference minimization is further performed at aligned instructions 10 and 5; it stops at 4 as no state difference is identified. The chain of cause transitions is: C has the incorrect value false at 5, then p and r point to the wrong places at 10 and 11, and finally the failure. These transitions compose a failure explanation.

Next, we define the composability property and show that it holds for the proposed primitive.

Definition 4.5.1 (Composability) *A scheme for memory differencing and replacement is composable iff given a set of unit differences* Δ=$\{\delta_1, \delta_2, ..., \delta_n\}$ *and the universal set* $\mathcal{U}$ *of all differences, after applying the differences in* Δ *from* T' *to* T*, the differences between* T' *and the mutated* T *is* $\mathcal{U} - \Delta$.

Composability is very important for cause transition computation, it ensures that the delta debugging algorithm is able to make progress, because it mandates that the effect of applying a set of differences must subsume the effect of applying a subset of the differences [132]. If a replacement scheme is not composable, applying the universal set of differences may even fail to convert T to T'. The symbolic path based scheme is not necessarily composable. As shown in Figure 4.7, applying the two differences of $p \rightarrow val$ and

$r \rightarrow val$ from the failing run to the passing run results in a state in which $p \rightarrow val$ still manifests itself as a difference.

Property 1 *The proposed MI based memory differencing and replacement primitive is composable.*

From Figure 4.11, we observe that for non-pointers, the primitive faithfully copies values; hence the property is trivially true. For pointers, the primitive either allocates a region when it is not present in the index tree or simply assigns the address if the region is present. Such behavior does not lead to additional differences that were not present in the original difference set or mask any other existing differences.

4.6 Evaluation

The implementation consists of both semantics and two client studies. It is based on the CIL infrastructure and has 3500 lines OCaml, 3500 lines C and 3000 lines Python. For these experiments, we used the SEI implementation of EPIDs.

4.6.1 Efficiency

The first experiment focuses on cost. The evaluation is on SPECint 2000 benchmarks. We excluded `252.eon` and `253.perlbmk` because they were not compatible with whole program analysis in CIL. All experiments were executed on an Intel Core 2 2.1GHz machine with 2 GB RAM and running Ubuntu 9.04.

Table 4.1 shows the instrumentation needed and characteristics of allocations. All executions are acquired on reference inputs. The second column shows the number of instrumented functions (after optimizations) and their percentage over all functions. The third column shows the same data for predicates. The fourth column shows the numbers of static allocation sites and dynamic allocations. The fifth column shows the average size of each allocation. The last column shows the maximum depth of the memory index tree. We observe that some programs make a lot of allocations with various sizes (`gcc` and `twolf`) and

Table 4.1: Instrumentation and allocation.

program	instmt. func	instmt. branch	# of alloc stat/ dyn.	avg. alloc size	tree dep.
164.gzip	11 (12%)	17 (1.8%)	5 / 436k	28 KB	130
175.vpr	100 (37%)	202 (7.5%)	3 / 107k	481 B	59
176.gcc	1282 (57%)	17774 (24.9%)	236/ 10.2m	5 KB	700
181.mcf	5 (19%)	6 (2%)	4 / 3	33 MB	8
186.crafty	8 (7.3%)	158 (2.9%)	12 / 37	23 KB	6
197.parser	2 (0.6%)	41(1.3%)	1 / 1	31 MB	292
254.gap	596 (70%)	4109 (19%)	2 / 2	100 MB	10
255.vortex	672 (73%)	3400 (19%)	9 / 258k	399 B	365
256.bzip2	8 (11%)	7 (1.1%)	10 / 36	16 MB	51
300.twolf	59 (31%)	218 (3.5%)	3 / 574k	31 B	28

some make very few but large allocations (`mcf` and `bzip2`). They have different impacts on the performance. Programs `parser` and `gap` allocate a memory pool at the beginning and then rely on their own memory management systems. Our current system does not trace into memory pool management. We leave it for future work. Observe, the maximum tree depth is not high with respect to the structural complexity of programs.

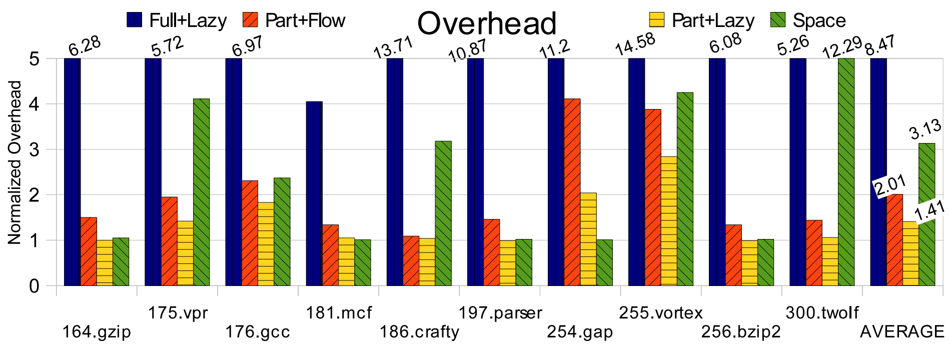


Figure 4.12.: Normalized runtime and space overheads of memory indexing with and without optimizations.

The overhead can be seen in Figure 4.12, in which `Full` represents implementation without removing redundant instrumentation; `Part` removing redundant instrumentation; `Flow` the online semantics; and `Lazy` the lazy semantics. The figure presents the performance overhead for a number of combinations. In practice, `Part+Lazy` is desirable for most applications, as illustrated by later client studies. `Space` represents the space overhead for `Part+Lazy`. All data is normalized against original runs without instrumentation.

We observe first that the `Full+Lazy` approach has substantially more runtime overhead than the `Part+Lazy` approach. `Part+Flow` is slightly more expensive than `Part+Lazy` due to instrumentation on pointer operations. The overhead of `Part+Lazy` is low (41%). Next, observe that in half the benchmarks, there is little space overhead. This is because the number of allocations and the tree depth are relatively low regarding the size of each allocation. In contrast, `300.twolf` had the most overhead. It performs a large number of very small allocations, <32 bytes on average, so on average maintaining the index for each allocation is more costly[2]. Nonetheless, the average space overhead is 213% (111% without `twolf`). The conclusion is that the cost of MI is feasible for many applications.

4.6.2 Trace Canonicalization

Trace canonicalization is the alignment of control flow and memory accesses across traces from two executions. It plays a part in debugging and regression analyses [42,57,99], among others. With MI, an important question can be answered, *given two address entries in two respective traces, should they be considered differences?* Note, two accesses at the corresponding points in the two traces do not mean that they operate on the same data; the accessed addresses being different does not mean they do not semantically correspond.

The study is on three common, open source programs, `make`, `gawk`, and `dot`. We reported the number of address differences before and after MI canonicalization. We turned off all memory layout randomization. To avoid comparing trace entries that do not correspond, we aligned the execution points of memory accesses in both executions and only compared accesses that occurred at aligning points.

We generated traces from the programs' provided test suites or, in the case of `dot`, the provided examples in the documentation. Each full trace was compared with traces generated by a fixed percentage of the input, i.e., removing part of the inputs. The results

[2]In our implementation, we use 22 bytes for each tree node.

are shown in Fig. 4.13. For each percentage of input similarity, we present the percentage of matched stack and heap memory accesses before and after canonicalization. For MI, these are '`MI locals`' and '`MI allocs`' respectively, while for the addresses without canonicalization, they are '`Addr allocs`' and '`Addr locals`'. We furthermore present the percentage of control flow correspondence ('`Control Flow`').

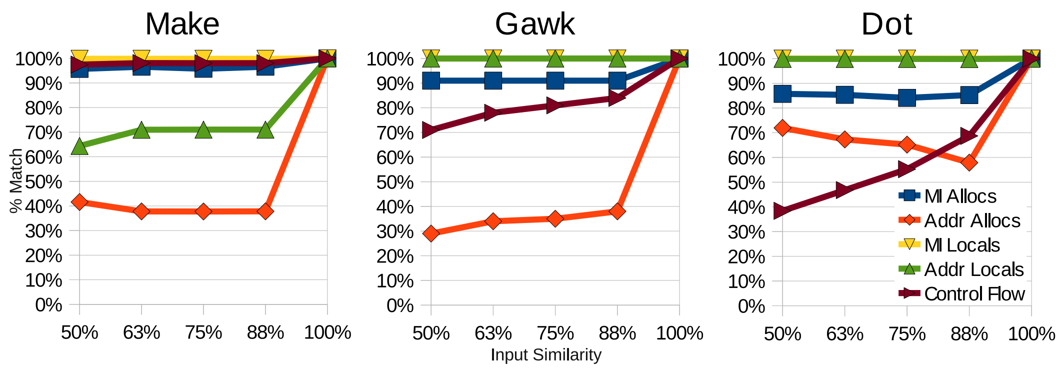


Figure 4.13.: Percent of corresponding memory accesses w. and w/o MI.

Observe first that MI provides a substantially higher level of heap access correspondence (50% more for `make` and 50-60% more for `gawk`, and 15-30% more for `dot`). Less benefit was observed in `dot` because `dot`'s dynamic allocations are largely on fixed buffers that do not change according to inputs. MI was able to find more corresponding addresses for local variables too. Observe that stack local allocation and variable sized objects on the stack make it more difficult to find correspondences without MI (e.g. the `make` case).

The control flow similarity increases as the input similarity increases. Note that the address correspondence without canonicalization stays roughly the same or even decreases as the control flow similarity increases (like in `dot`). The decrease happened because greater control flow similarity allows more (different) addresses to be compared. In contrast, the correspondence found by MI is roughly consistent.

4.6.3 Cause Transition Computation

This experiment evaluates the impact of MI on computing cause transitions. The algorithm in [114] was implemented as a platform on which we tested two versions of the memory comparison and replacement primitive: one is symbolic path based, used in [28, 114]; the other is the new MI-based. The study is on several real bugs in open source programs, including `gcc`, `make`, and `gawk`, that have non-trivial heap behavior and aliasing. The failing runs are generated according to the bug reports. The passing runs are acquired from the correct inputs in the reports if provided; previous non-regressing versions; or using an automated patching technique [134]. Note that acquiring passing runs is an orthogonal problem out of the scope of this chapter. Other patching techniques such as [62] can also be used.

Results are summarized in Table 4.2. The `Program` column contains the buggy programs. `Bug ID` presents the bug id, through which one can identify the report online, or the publication date on the mailing list. `Bug` describes each bug. `Passing Run` shows the sources of the passing runs: inputs provided in reports (`correct input`), non-regressing versions (`non-regressing`), and dynamic patching (`predicate switch`). The maximum number of differences (present in failing and absent in passing) found using symbolic differencing is presented in `Sym Diffs`, and the maximum when using MI is in `MI Diffs`. In `Sym Diffs`, we report one memory cell only once although it may be a difference along multiple paths. Of further interest is the number of differences with aliases (in column `Aliases`), or multiple symbolic paths. They can cause issues as discussed in Section 4.1. Column `Issue` presents the exhibited problems when using the symbolic path based primitive. We also present the number of transitions and the average number of differences included in each transition in `Trans/Diffs`, along with time required (in seconds) in `Time` when using the MI version.

Observe that in every case, the number of symbolic differences is substantially, 2-50 times, larger than the number of differences when using MI, because the proper memory correspondence cannot be found. Furthermore, the `Aliases` column shows that substantial aliasing is common, creating a lot of difficulties for the symbolic method. As seen in the `Issue` column, in most cases, symbolic path based computation would not terminate within 12 hours. This stems in part from the large number of differences found. For example, in the first `gcc` case, it may be possible that all the enumerated subsets of the 8365

Table 4.2: Cause transition computation for failures.

Program	Bug ID	Bug	Passing	Sym. Diffs	MI Diffs	Aliases	Issue	Trans/Diffs	Time (s)
gcc 2.95.2	529	-Wshadow warns on functions	predicate switch	8365	233	8105	>12h	8/1	4559
gcc 2.95.2	776	Large array size causes abort	predicate switch	10101	230	9027	>12h	2/1	379
gcc 2.95.2	2771	-O1 breaks strength-reduce	provided input	11095	284	10254	>12h	4/1	1797
make 3.81	16958	.PHONY targets are unrecognized	non-regressing	2699	184	33	>12h	9/1	740
make 3.81	18435	Parentheses break make targets	provided input	3301	336	356	>12h	29/2	645
make 3.81	19133	./ prevents self remake	provided input	3728	550	187	>12h	5/2	235
make 3.80	112	Rules cannot handle colons	provided input	3309	645	217	>12h	11/6	685
gawk 3.1.5	1/20/06	Deallocate bad pointer	provided input	630	22	509	early term.	8/1	56

differences need to be tested. In `gawk`, the algorithm simply terminated early, unable to produce relevant transitions for the failure.

A Case Study on Detailed Comparison. We performed a separate test focusing on `make` bug 18435 from Table 4.2. We selected 10 sample points at 10% intervals along the part of the passing run that is beyond the first divergence of the two runs. At each sample point, we compared the memory snapshots across the two runs and then mutated the memory in the passing to that in the failing by applying the universal set of differences, alternatively using the symbolic path based primitive [28, 114] and the MI based primitive. Then we collected the trace after the mutation and compared it to that of the failing run. We performed trace comparison by aligning the corresponding instructions and memory addresses across the executions. According to the discussion in Section 4.5, the traces should be identical if the primitives are completely composable.

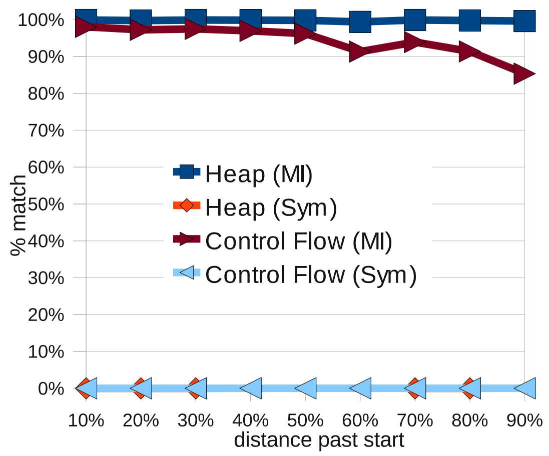


Figure 4.14.: Heap accesses and control flow trace similarity after state mutation in the execution of `make`.

The results are shown in Figure 4.14. '`Heap (MI)`' and '`Control Flow (MI)`' represent the similarities of heap access and control flow traces using the MI based primitive, and '`Heap (Sym)`' and '`Control Flow (Sym)`' represent those using the symbolic primitive. Observe, the access similarity when using MI is consistently near 100%, and the control flow similarity is consistently above 90% until the end. This means the mutation is mostly successful in turning the passing run to the failing run. The similarity is not 100%

because we currently do not model external state such as file IDs, process IDs, etc. Thus, such states are not eligible for meaningful comparison and replacement. In contrast, when the symbolic primitive is used, the execution quickly diverges from the expected control flow, having near 0% similarity, and it has near 0% similarity for accessing the heap. In fact, the mutated run often quickly crashes due to destructive mutation (Section 4.1). This supports that the MI primitive is composable, but the symbolic primitive is not.

In summary, MI allows cause transition computation to be more precisely realized, reflected by our success of scaling to programs like `gcc` with full automation. Note, although a `gcc` case was presented in [132]. It was conducted with human intervention.

4.7 Related Work

Trace normalization [37] divides traces into segments. Segments with the same starting and ending state are considered equivalent. Client applications using such traces only need to look at a consistent representative segment from each equivalence class. The outcome is reduced workload and increased precision. Memory indexing is complementary in that it provides a robust way of comparing program state across executions and hence helps identify equivalent trace segments.

Recent work has examined comparing executions for debugging regression faults [57], analyzing impact of code changes [99], and finding matching statements across program versions [42]. These techniques are able to construct a symbolic mapping of variables across program versions through profiling, such as pointer x in version one being renamed to y in version two. The constructed mapping is static. In comparison, we focus on comparing dynamic (address) values of corresponding variables, answering questions like "does x point to the corresponding address in the two executions". Furthermore, trace canonicalization facilitated by MI would improve the precision of these analyses.

Many debugging techniques [21,25,60,64,66,81] compute fault candidates by looking at a large number of executions, both passing and failing. In these techniques, execution profiles are collected and analyzed statistically. Some debugging techniques compare a simple profile of a failure with a small number of correct runs (usually one) [53]. They use control flow paths and code coverage. MI is complementary to these techniques by providing a way to canonicalize profiles before they are analyzed to achieve better precision, especially for pointer related bugs. We have demonstrated in this chapter that MI is able to drive cause transition computation that is highly sensitive to memory alignment.

Compared to the recent advances on generating causal explanations of failures [25, 64], The proposed robust, fine-grained memory differencing and substitution primitives make it feasible to extract succinct and in-depth information about failures. For instance, it is easier for us to reason about whether a value at a given execution point is relevant to a failure. Furthermore, MI improves cause transition computation [28, 114] by allowing alignment along the memory dimension, which substantially improves robustness in the presence of aliasing.

Abstractions for memory regions used in static analysis also have the notion that small, bounded chains of the control dependence of a region's creation provide a notion of identity for that region [109]. In contrast, we extend this into the dynamic domain, efficiently capturing precise identities online instead of just over-approximations.

Joshi et al. use execution points to locate locks across executions of Java programs [70], but the approach is not as generalized or optimized as memory indexing.

4.8 Conclusions

We present a novel challenge in dynamic program analysis: aligning memory locations across executions. We propose a solution called memory indexing (MI), which provides a canonical representation for memory addresses such that memory locations across runs can be aligned by their indices. Pointer values can be compared across runs by their indices. The index of a memory region is derived from the canonical control flow representation of its dynamic allocation site such that control flow correspondence is projected to memory correspondence. Enabled by MI, we also propose a novel memory substitution primitive that allows robustly copying states across runs. We evaluate the efficiency of two memory indexing semantics. Our results show that the technique has 41% runtime overhead and 213% space overhead on average. We evaluate effectiveness through two client studies: one is trace canonicalization and the other is cause transition computation on failures. The studies show that MI reduces address trace differences by 15-60%. It also scales cause transition computation to programs with complex heap structures.

5 EXPLAINING WHY EXECUTIONS DIFFER

Explaining *why* something happened is a subtle task; philosophers have debated the notion of causation for centuries [40,73,87]. One common thread among the myriad approaches is that they involve comparing a world in which that something happened to others in which it did not. Many software engineering techniques take similar approaches in explaining software behavior. For example, in probabilistic fault localization, a set of failing runs is contrasted with a set of passing runs [11, 66] to provide probabilistic insights into the cause of the failures. Compared to techniques that do not rely on comparison to explain software behavior, such as program slicing [136], these techniques are more precise as they use comparison to trim unnecessary information.

One classic fine-grained comparative technique for identifying causes when one execution (e.g., a buggy execution) differs from another (e.g. a similar correct execution) is Zeller's delta debugging approach [132]. It is capable of reasoning about causality at the granularity of individual instructions and variables, generating much more informative and precise failure explanations compared to other techniques [115]. The technique involves replacing part of the state in the correct execution with that from the buggy execution and determining whether such replacement induces the failure in the modified execution. However, due to the complexity of program state (e.g. inter-connected data structures in the heap, pointers, and external resources), it faces many problems in practice. In particular, entangling the states from both executions allows them affect each other in undesirable and unexpected ways, leading to poor failure explanations. More discussion of the limitations of the technique can be found in Section 5.1.

In this chapter, we propose a novel fine-grained causal inference technique that illustrates the behavior of the findCauses() function in Figure 2.2. Given two executions and some observed differences between them, the technique can precisely reason about the causes of such differences. While the technique reasons about causality through state replacement, it makes three key advances. It features a novel execution model that avoids undesirable entangling of the replaced state and the original state such that the precision of causal inference can be substantially improved. It is capable of handling execution omission errors by analyzing both executions symmetrically. It also leverages an existing slicing technique called dual slicing [125] to limit the scope of causality testing while ensuring no relevant state differences can be missed. As a result, the efficiency is substantially improved.

Our main contributions are highlighted as follows.

- We first thoroughly discuss the limitations of the state of the art fine-grained causal inference technique that has been used for many years. We especially study the problems in state replacement.
- We propose a novel causal inference model that is symmetric and comparative. We declare the goals of the model, which reflect the user's intention when reasoning about software behavior by comparison.
- We propose a novel realization of the model. It leverages dual slicing to ensure relevance of the causes and limit the scope of causality testing. While it makes use of state replacement to determine causality, a novel execution model and its approximation are developed to avoid the undesirable entangling of the state from both executions.
- We implement and evaluate a prototype. We apply the causal inference engine toward failure explanation for 15 real world bugs, including all the reported bugs for `tar`, `make`, and `grep` in a one year period. Comparison against the causal inference engine from the most recent improved delta debugging [115] and dual slicing techniques shows that our technique has substantially improved the efficiency and effectiveness of failure explanation.

The ideas presented in this chapter have previously been published by the author in the proceedings of ICSE 2013 [115] (©2013 IEEE).

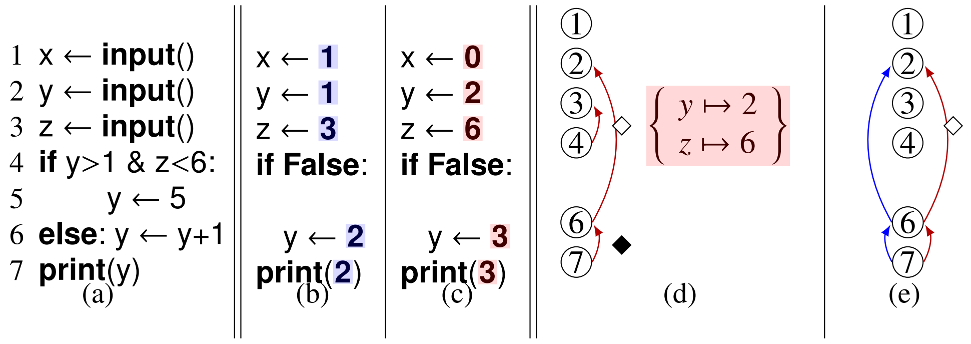


Figure 5.1.: (a) A program. (b-c) executions with differing input. (d) CSM. (e) dual slice. Symbols ◇ and ◆ denote the cause point and effect point, respectively. The set in (d) represents the causal state set.

Delta debugging is a classic debugging technique that can minimize failure inducing inputs [133] or the faulty internal program state essential to reproducing a failure [28, 132]. The original work first contrasts a buggy execution with a similar correct execution to determine state differences [28, 132]. It then performs *Causal State Minimization* (CSM) to determine the minimal subset of state differences essential to reproducing the failure. CSM involves performing the correct execution up to a point of interest preceding the failure, called *the cause point*, replacing a subset of program state with state from the buggy execution, and continuing this patched execution to determine whether the failure can be induced. If so, the subset is called a *causal state set* or *cause set*. The technique makes use of a generalized binary search to enumerate and test different subsets until it identifies the minimal cause set. We recently combined delta debugging with the more precise execution alignment techniques in Chapter 3 and Chapter 4 to improve its robustness, precision, and efficiency. By applying CSM inductively, a causal chain or *summary* of a failure can be computed, comprising a sequence of the minimal causal state sets computed for a sequence of execution points leading from the root cause to the failure [114, 115].

Example. Consider the simple program presented in Figure 5.1. This program reads three integers, re-defines one of them, and then prints it. In the execution of (b), the user inputs 1, 1, 3 and the program prints 2. In contrast, in the execution (c), the user inputs 0, 2, 6 and the program prints 3. Suppose that execution (c) is buggy. Given the buggy output 3 on line 7, called the *effect point*, we apply CSM to determine what state on line 4, called the *cause point*, actually caused the buggy output. The cause and effect points are respectively marked in the figure as empty and filled diamonds in (d).

Note, here the term "*buggy*" is a generalized notion as there is not a faulty statement per se. *Any* behavioral difference between the executions may be considered buggy and we are interested in what caused these differences. The discussion and the technique are universally applicable for cases where true faults cause the behavioral differences.

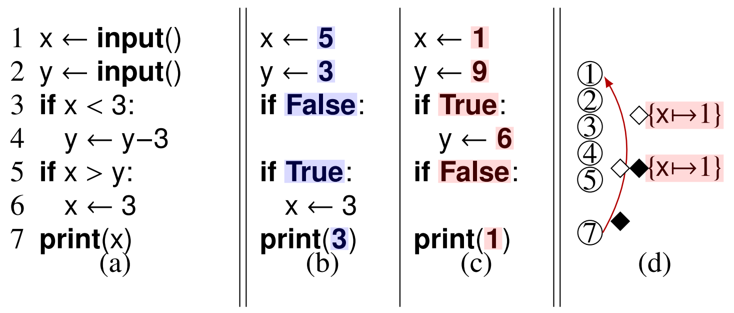
CSM repeatedly replays execution (b) up to line 4. Each time, it then replaces a subset of state with state from execution (c) to identify a subset sufficient to produce $y \mapsto 3$ within execution (b). For instance, replacing (on line 4) the variable/value mappings $y \mapsto 1$ and $z \mapsto 3$ in execution (b) with $y \mapsto 2$ and $z \mapsto 6$ from execution (c) yields $y \mapsto 3$ on line 7. Thus, the process identifies that the values of y and z are buggy on line 4 in execution (c), leading to the buggy output. Figure 5.1d presents the causal state set on line 4 along with relevant program dependences for comprehension. The computation continues in order to determine whether a smaller causal set can be identified. If not, the identified minimal set will be reported.

If we desire a summary of the failure, the current cause point becomes the new effect point and the identified causal state set becomes the new target buggy state. The algorithm then continues to compute the causal state set for a preceding new cause point, until no such sets can be computed [115].

Limitations. Delta debugging [28, 132] and its recent improvements [114, 115] all use CSM. While prior research demonstrated the effectiveness of these techniques, we find that inherent limitations of CSM often lead to low quality failure summaries. Next, we discuss these limitations in detail and motivate the need for a new causal inference engine.

Confounding caused by partial state replacement: The first problem with CSM is that *replacing only a subset of the state in an execution can induce new behavior that was not present in **either** of the original executions*. We call this problem the *confounding of partial state replacement*. The introduced new behavior can affect the validity of a causality test. Particularly, a causal chain may terminate prematurely because key buggy state is excluded due to confounding, or it may contain additional state that does not pertain to the failure. In the worst case, the entire chain may not even be relevant for explaining the failure. From our experiments, 11 of the 15 real bugs suffered from this problem.

For example, consider the program presented in Figure 5.1. Previously, we showed that CSM can determine that {y↦2,z↦6} is the causal state set on line 4. Suppose CSM further considers a smaller subset {y↦2}. When replacing the value of y in execution (b) with that from (c), the condition of the **if** statement becomes **True**. This redefines y on line 5, rendering the target state y↦3 uninducible. Because of that condition, CSM finds replacing the values of both y and z necessary. However, z is unrelated to the original behavioral difference. The only contribution of z in both executions is its use on line 4, which had the value **False** in *both* executions. Ideally, only the definition of y↦2 should be blamed for the failure.



```
1 x ← input()     x ← 5          x ← 1
2 y ← input()     y ← 3          y ← 9
3 if x < 3:       if False:      if True:
4   y ← y−3                        y ← 6
5 if x > y:       if True:       if False:
6   x ← 3           x ← 3
7 print(x)        print(3)       print(1)
      (a)             (b)            (c)
```

Figure 5.2.: Missing causes by execution omission. (a) program. (b-c) executions with differing input. (d) CSM result.

Execution omission: The second problem is that *CSM may miss important causal state in the presence of execution omission errors [140]*, where the buggy target state is produced because statements were *not executed* due to the bug. In such cases, the computed failure summaries are usually incomplete. The root cause of the problem is that *CSM is **asymmetric***, meaning the buggy and correct executions have asymmetric roles in the process: CSM reasoning is based on modifying state only in the correct execution; its final results only include information from the buggy execution.

Figure 5.2 presents an example. The correct execution in (b) follows the **False** branch of line 3, then the **True** branch of line 5, and prints 3, whereas the “buggy” execution in (c) follows the **True** branch of line 3, then the **False** branch of line 5, and prints 1. Suppose that initially the effect point is line 7 and the cause point is line 5. CSM determines that replacing the value of x is sufficient to induce the buggy target state in (b), so it identifies $x \mapsto 1$ as the only buggy state at the cause point. However, the buggy output $x \mapsto 1$ on line 7 in (c) is due to the undesirable omission of line 6, which is partially determined by the buggy state of $y \mapsto 6$. Missing $y \mapsto 6$ in the cause set leads to an incomplete summary of the failure.

Suppose the computation continues backward with a new effect point on line 5 and new cause point on line 3. CSM determines that replacing the value of x on line 3 is sufficient to induce the buggy target state $x \mapsto 1$ on line 5. Figure 5.2d shows the result of this analysis. This implies $x \mapsto 1$ is the sole root cause of the bug. However, replacing the value of x on line 3 in (b) cannot induce the final failure although it can induce $x \mapsto 1$ on line 5, because line 3 evaluates to **True** in the patched execution. Hence, line 4 produces $y \mapsto 0$ and leads to $x \mapsto 3$.

In our experiments, 5 of the 15 real bugs face this problem.

Efficiency: CSM may demand a large number of reexecutions. The number of state differences can be as large as the size of the allocated memory [115]. The number of possible subsets that need to be tested for causality is potentially combinatorial in terms of the full set. To combat this, existing approaches use *delta debugging* [132] to perform a generalized binary search over the subsets. However, the number of reexecutions can still be quadratic in the size of all used memory. Even the most recent implementation of CSM [115] may take a few hours to reason about a failure while the original execution time is just a few milliseconds.

5.2 Comparative Causality

In this chapter, we propose a more effective and precise causal inference model called *comparative causality* (CC). This model focuses on *symmetrically* reasoning about two executions, one buggy and one correct[1], in order to explain why they both differ from eachother. It also enables efficient and practical implementation. In the following, we first define a number of notations and concepts. Then we study the intended properties of the new model. Here we assume we can properly align the control flow and the variables/memory regions of the two executions for fine-grained comparison using the work presented in Chapter 3 and Chapter 4:

- **Execution point:** We use a superscripted label l^e to denote a point in execution e. Symbol $l^{(e_1,e_2)}$ denotes a point that appears in both executions e_1 and e_2, determined by the given control flow alignment as described in Chapter 3. It is also called an *aligned point*.
- **State difference:** we use $\{\mathsf{x} \mapsto (v_1,v_2)\}$ to denote that a variable x has value v_1 in e_1 and value v_2 in e_2, with $v_1 \neq v_2$.

Problem Statement: Given a set of state differences Δ at an aligned execution point $l_{\blacklozenge}^{(e_1,e_2)}$ and a preceding aligned point $l_{\diamond}^{(e_1,e_2)}$, we want to find a set of state differences at $l_{\diamond}^{(e_1,e_2)}$ that is *relevant*, *sufficient*, and *minimal* for inducing Δ.

The preceding execution point is the *cause* point and the latter one the *effect* point. We demand aligned points because state comparison is not meaningful at non-aligned points. An inducing state difference in the cause point is called a *cause*; a state difference in Δ is called an *effect*.

5.2.1 Property One: Relevance

The causes identified by CC must be *relevant* to the target effects. Intuitively, a difference d is relevant to a later difference d_s if d_s is (transitively) produced from d through a

[1]How to acquire a correct execution given only the buggy execution can be found in a survey [113].

sequence of differences. It represents the notion that “buggy state must be derived from preceding buggy state (except at the root cause)”.

Consider the example presented in Figure 5.1. The state difference {z ↦(3,6)} on line 3 is not relevant to {y ↦(2,3)} on line 7 even though there is a dynamic dependence path from line 3 to line 7, because the difference of z is neutralized on line 4, which yields **False** in both runs. In contrast, The difference {y ↦(1,2)} on line 2 is relevant to {y ↦(2,3)} on line 7. More formally:

Definition 5.2.1 (Relevance) *A state difference $\delta_\diamond$ at $l_\diamond^{(e_1,e_2)}$ is* relevant *to a target state difference $\delta_\blacklozenge$ at a later effect point $l_\blacklozenge^{(e_1,e_2)}$ if either of the following conditions is satisfied.*

1. *There exists a dynamic program dependence path from $\delta_\blacklozenge$ to $\delta_\diamond$ in e_1 (e_2) where all the statement computations along the path yield different results from the other execution e_2 (e_1).*

2. *There exists a state difference δ_x in an aligned point in between $l_\diamond^{(e_1,e_2)}$ and $l_\blacklozenge^{(e_1,e_2)}$ such that $\delta_\diamond$ is relevant to δ_x and δ_x is relevant to $\delta_\blacklozenge$.*

Condition (1) expresses the requirement that a difference cannot be neutralized within an execution in order to be relevant. Note that this property is symmetric to both executions as relevance can be determined by a dependence path in *either* execution. This allows us to precisely capture relevance in the presence of execution omission. Consider the example in Figure 5.2. The state difference {y ↦(3,6)} on line 5 is relevant to {x ↦(3,1)} on line 7, due to the dependence path $y@5 \leftarrow \texttt{True}@5 \leftarrow 6 \leftarrow 7$ in (Figure 5.2b). Observe that there is no dependence between $y@5$ and $x@7$ in the failing execution (Figure 5.2c) due to the omission of line 6. The underlying intuition is that omission is an asymmetric concept regarding one execution. An omitted statement regarding one execution implies that it appears in the opposing execution. With our symmetric definition, omissions are conceptually precluded. Condition (2) expresses that relevance can be transitive, even across the two executions.

5.2.2 Property Two: Sufficiency

The identified set of causes must sufficiently induce the target effect of *each* of the two executions within its opposing execution. This inducement acts as a new causality test and witnesses the causal relationship between the identified causes and the target state.

The property is *symmetric* as it requires the set of effects in either execution to be induced by the causes. It means that if for all the variables in the cause set, we copy their values from execution e_1 to e_2, we can induce the target effect of e_1 at the effect point in e_2, *and vice versa*.

Consider the example in Figure 5.2. State differences {y ↦(3,6), x ↦(5,1)} on line 5 form a sufficient set regarding the effect {x ↦(3,1)} on line 7. In contrast, the difference {x ↦(5,1)} itself is insufficient because although replacing x's value 5 with 1 in (b) can induce the effect {x↦1} on line 7, replacing x's value 1 with 5 in (c) cannot induce the effect {x↦3}. This symmetry ensures that we capture relevance due to execution omission. More formally,

Definition 5.2.2 (Sufficiency) *A cause set $\Delta_\diamond$ at $l_\diamond^{(e_1,e_2)}$ is* sufficient *for a given target effect set $\Delta_\blacklozenge$ at a later effect point $l_\blacklozenge^{(e_1,e_2)}$ if and only if, in the absence of confounding, copying the state of e_2 in $\Delta_\diamond$ to e_1 at the cause point induces the effect of e_2 in $\Delta_\blacklozenge$ in execution e_1 at the effect point, and vice versa.*

One key condition is that reexecution should be confounding-free. Unfortunately, normal program execution cannot guarantee this. The remainder of this subsection focuses on discussing confounding.

What is confounding? Determining sufficiency involves replacing part of the state in one execution with values from the opposing execution. However, the continuation of the modified execution has state from both original executions entangled, affecting each other and inducing undesirable and unexpected results in causal inference.

Recall in Figure 5.1, we saw that partially changing the state of execution (b) with the single desired cause variable y yielded output different than in either execution (b) or (c). In addition, we found that including z as a cause along with y would yield the target state, although z is not relevant to the output. Both of these are unexpected results that we call *confounding from partial state replacement*. These confounding effects do not just have the ability to *include* arbitrary state within the set of identified causes, they can *exclude* arbitrary state, as well.

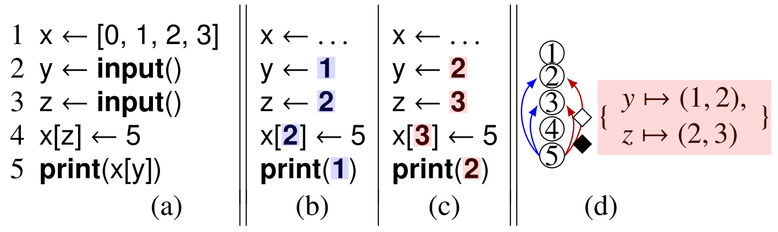
At a high level, these unexpected results occur because partial state replacement *created new behaviors that did not exist in either of the original executions*.

Definition 5.2.3 (Confounding) *Given executions e_1 and e_2 as well as a patched execution e_p constructed from them, a causality test using e_p is* confounded *if either of the following conditions are satisfied:*

1. *An execution point in e_p is not present in e_1 or e_2.*

2. *A data dependence in e_p is not exercised in e_1 or e_2*

Condition (1) corresponds to *control flow confounding* and (2) to *data flow confounding*, which means confounding can occur without exhibiting any new control flow.



```
1 x ← [0, 1, 2, 3]    x ← ...        x ← ...
2 y ← input()         y ← 1          y ← 2
3 z ← input()         z ← 2          z ← 3
4 x[z] ← 5            x[2] ← 5       x[3] ← 5
5 print(x[y])         print(1)       print(2)
        (a)              (b)            (c)
```

Figure 5.3.: Data flow confounding example. (a) program. (b-c) executions with differing input. (d) confounded explanation.

Consider the example in Figure 5.3. This time, the target state is {x[y] ↦(1,2)} with cause and effect points at lines 4 and 5 respectively. Observe that in each execution, the read from and written to elements of x are different. Thus, the only identified cause for the different output should be the differing values of y, which provides the index read from the list. However, when only the value of y is replaced on line 4 in (b), the patched execution reads the new value written to the list on line 4. Thus, the target state is not induced. Observe that in this case, a new data dependence from line 5 to line 4 is exercised.

In later sections, we will examine new execution models that can avoid/mitigate confounding. We argue that the two properties, together with the minimality requirement, are essential for understanding execution differences. They precisely express the programmer's intentions.

5.3 Realizing Comparative Causal Inference

In this section, we discuss the realization of CC. Given a target effect set and a cause point, we leverage a technique called dual slicing to compute a set of candidate causes and only apply causality testing on the candidate set. Dual slicing is a symmetric slicing technique that works on two executions. It first determines control flow and value differences in the two executions through trace comparison and then performs slicing on these differences (in and across both executions). The benefits of using dual slicing are twofold. First, it ensures relevance of the candidates. Second, it is more efficient because causality testing only needs to enumerate subsets of the candidates instead of the full set of state differences as in CSM [115, 132].

After acquiring the dual slice, we then *symmetrically* minimize the causes included in the slice to a minimal subset sufficient for inducing the target state within *both* executions. During the minimization process, one key step is to perform causality testing by state replacement. In order to avoid confounding, we devise an execution model that harnesses a patched execution in such a way that it respects the control flow and dependences in the two original executions while allowing flexibility for reasoning about the effects of state replacement.

5.3.1 Background: Dual Slicing

Dual slicing was first introduced to study concurrency bugs [125] and software vulnerabilities [65]. Figure 5.4 presents the basic dual slicing algorithm. Although it is not part of this chapter's contributions, we present a simplified version of the algorithm for completeness.

```
DUALSLICE($l_\blacklozenge^{(e_1,e_2)}$)
Input:    $l_\blacklozenge^{(e_1,e_2)}$ - the slicing criterion
Output:   $\mathcal{D}$ - the dual slice, a set of deps in either execution

 1: if $e_1 \neq \perp$ then
 2:     for each data dep $dd \leftarrow \{l_\blacklozenge^{(e_1,e_2)} \xrightarrow[e_1]{x} l_\diamond^{(e_1,e_2')}\}$ do
 3:         if $e_2' \equiv \perp$ or $x$ has different values on $l_\diamond$ then
 4:             $\mathcal{D} \leftarrow \mathcal{D} \cup dd \cup$ DUALSLICE($l_\diamond^{(e_1,e_2')}$)
 5:     control dep $cd \leftarrow \{l_\blacklozenge^{(e_1,e_2)} \underset{e_1}{\Longrightarrow} l_\diamond^{(e_1,e_2')}\}$
 6:     if $e_2' \equiv \perp$ or $l_\diamond$ has different branch outcomes then
 7:         $\mathcal{D} \leftarrow \mathcal{D} \cup cd \cup$ DUALSLICE($l_\diamond^{(e_1,e_2')}$)
 8: if $e_2 \neq \perp$ then
 9:     /* operations symmetric to when $e_1 \neq \perp$ */
10: return $\mathcal{D}$
```

Figure 5.4.: Dual slicing

Given a slicing criterion, an execution point that exhibits a state difference, the algorithm returns its dual slice, a set of dynamic dependences from both executions denoting the causality of the difference. Lines 1-7 describe the process of slicing in execution e_1. It first ensures that the current criterion $l_\blacklozenge$ is present in e_1 (line 1). Here, $l_\blacklozenge^{(\perp,e_2)}$ denotes that $l_\blacklozenge$ is not present in e_1. Lines 2-4 traverse each dynamic data dependence edge of the criterion in e_1 with x, the variable involved, denoted as $l_\blacklozenge^{(e_1,e_2)} \xrightarrow[e_1]{x} l_\diamond^{(e_1,e_2')}$. We use variable e_2' to show that $l_\diamond$ may or may not be in the second execution, disregarding the value of e_2. On lines 3-4, if $l_\diamond$ is exclusively in e_1 (i.e, $e_2' \equiv \perp$) and thus is a control flow difference, or even if it is not exclusive but variable x has different values in the two executions, the data dependence is added to the slice. The dual slice of $l_\diamond$ is recursively computed and added to the slice too (line 4). Thus, when $l_\diamond$ is present in both executions and produces the same value, it is not added because it cannot induce the criterion. In lines 5-7, the algorithm traverses the control dependence edge in e_1, denoted as "$\underset{e_1}{\Longrightarrow}$". Similarly, if the guarding predicate is exclusive or has different branch outcomes, the edge gets added and the dual slice of the predicate is recursively computed. Lines 8-9 are symmetric to lines 1-7, describing the process of slicing in execution e_2.

```
1  t ← input()
2  x ← input()
3  y ← input()
4  z ← input()
5  if x + y + z > 3:
6     z ← −10
7  if x + y + z > 0:
8     z ← 5
9  if z < 0 and y > 0:
10    z ← t
11 else: print(z)
```
(a)

```
t ← 0
x ← 1
y ← 0
z ← 4
if True:
  z ← −10
if False:

if False:

  print(-10)
```
(b)

```
t ← 1
x ← 1
y ← 1
z ← 1
if False:

if True:
  z ← 5
if False:

  print(5)
```
(c)

```
y ← input()
z ← input()
if 1+y+z >3:
  z ← −10
if 1+y+z >0:
  z ← 5

print(z)
```
(d)

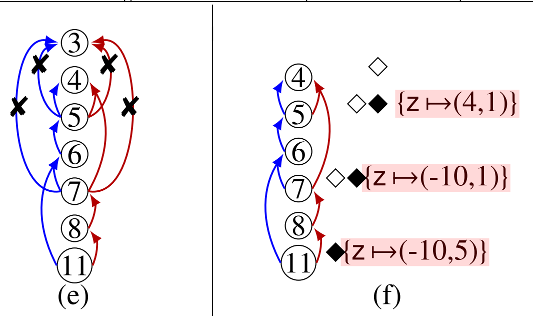


Figure 5.5.: (a) program. (b-c) two runs. (d) program from the dual slice. (e) dual slice. (f) CC explanation.

Example. Consider the program in Figure 5.5a. The dual slice of the two executions, (b) and (c), is presented in Figure 5.5e (including the crossed-out dependences). Part of the computation is represented as follows. We use *dS()* as a shorthand for DUALSLICE(). The superscripts of execution points are elided for brevity when explicit from the context. The box in a step denotes that the next step is to execute the recursive call inside.

$$\begin{aligned} dS(11^{(b,c)}) &= \{11 \xrightarrow[b]{z} 6\} \cup \boxed{dS(6^{(b,\perp)})} \cup \{11 \xrightarrow[c]{z} 8\} \cup dS(8^{(\perp,c)}) && [1] \\ &= \{11 \xrightarrow[b]{z} 6,\ 6 \underset{b}{\Longrightarrow} 5\} \cup \boxed{dS(5^{(b,c)})} \cup \{11 \xrightarrow[c]{z} 8\}... && [2] \\ &= \{11 \xrightarrow[b]{z} 6,\ 6 \underset{b}{\Longrightarrow} 5, 5 \xrightarrow[b]{z} 4, 5 \xrightarrow[c]{z} 4, ...\}... && [3] \end{aligned}$$

At step [1], the control dependence to line 9 is not involved as it has the same branch outcome in the two runs. Also, dual slicing line 6 of execution (b) in step [1] entails slicing line 5 in both executions (step [2]). Line 1 is not included, even though it denotes a difference, as it is not reachable from the criterion. The dual slice captures the behavioral differences of the two executions related to the criterion.

5.3.2 Dual Slices Are Relevant but Not Ideal

Dual slices are represented in terms of dependences, whereas causal inference is conducted on program state. Hence, we first introduce a projection from a dual slice to the corresponding set of state differences at a given execution point so that we can discuss the properties of dual slicing in our context. These properties are unique to the proposed technique and have not been studied before.

Given a dual slice and a cause point $l^{(e_1,e_2)}$, which is an aligned point, we define the cut of the dual slice with respect to the point as follows.

$$\begin{aligned} C(\mathcal{D}, l^{(e_1,e_2)}) = \;& \{x \mapsto (v_1, v_2) \mid l_{\blacklozenge}^{e_t} \xrightarrow[e_t]{x} l_{\diamond}^{e_t} \in \mathcal{D}, \\ & \text{with } l_{\diamond} \prec_{e_t} l \prec_{e_t} l_{\blacklozenge} \text{ or } l_{\blacklozenge} \equiv l, \\ & \text{and } x \mapsto (v_1, v_2) \text{ on } l \text{ with } v_1 \neq v_2\} \end{aligned}$$

It denotes the set of state differences involved in the dual slice on the given cause point. It essentially denotes the set of variables when we cut the dual slice on the cause point. Symbol $l_a \prec_{e_t} l_b$ denotes l_a precedes l_b in execution e_t.

Consider the dual slice in Figure 5.5e. The cut on line 7 is the following. $C(\mathcal{D}, 7^{(b,c)}) =$ {z↦(-10,1), y↦(0,1)}. Note that {t↦(0,1)} is not in the cut.

Theorem 5.3.1 *All the causes in a cut $C(\mathcal{D}, l^{(e_1,e_2)})$ are relevant to the slicing criterion. All relevant causes on $l^{(e_1,e_2)}$) are included in its cut.*

The property suggests that dual slices cover all the causes the programmer needs to inspect. Unfortunately, a dual slice cut may not sufficiently induce the slicing criterion given the confounding-prone regular execution model. That is, replacing the state of all causes in a cut may not induce the failure. Let us revisit the example in Figure 5.1. The dual slice is shown in Figure 5.1e. Its cut on line 4 has only y. However, from the discussion

```
inferCauses($\mathcal{D}$, $l_\diamond^{(e_1,e_2)}$, $l_\blacklozenge^{(e_1,e_2)}$, $\Delta_\blacklozenge$)
Input:   $\mathcal{D}$ - the dual slice          $l_\diamond$ - the cause point
         $l_\blacklozenge$ - the effect point          $\Delta_\blacklozenge$ - the target state
Output:  causes of target at $l_\diamond$

1: $\Delta \leftarrow C(\mathcal{D}, l_\diamond)$
2: $\Delta_{min} \leftarrow \Delta$
3: for each $s \subset \Delta$ by delta debugging do
4:     if $|s| < |\Delta_{min}|$
       $\wedge\ \mathcal{E}^{e_1}[s\downarrow_{e_2} / s\downarrow_{e_1}]_{l_\blacklozenge}^{l_\diamond} \rightsquigarrow \Delta_\blacklozenge\downarrow_{e_2}$
       $\wedge\ \mathcal{E}^{e_2}[s\downarrow_{e_1} / s\downarrow_{e_2}]_{l_\blacklozenge}^{l_\diamond} \rightsquigarrow \Delta_\blacklozenge\downarrow_{e_1}$ then
5:         $\Delta_{min} \leftarrow s$
6: return $\Delta_{min}$
```

Figure 5.6.: Minimizing causes

in Section 5.1, we know that replacing y↦1 with y↦2 in execution (b) does not lead to the target effect due to the confounding from z.

A cut may also not be minimal. It may contain causes that are not essential for inducing the target effect. In Figure 5.5e, the cut on line 7 is {z ↦(-10,1), y ↦(0,1)}, but the minimal sufficient set is just {z ↦(-10,1)}.

These limitations motivate us to realize the proposed CC by performing confounding-free minimization on dual slices.

5.3.3 The Basic Algorithm

In this subsection, we assume a confounding-free execution model and introduce the basic minimization algorithm. We will discuss the execution model in the next subsection.

Figure 5.6 presents the basic approach. Given a precomputed dual slice, the cause and effect points, and the target state, the algorithm returns a minimal set of causes sufficient to induce the target state. The algorithm starts by computing a dual slice cut at the cause point, which is essentially the set of relevant causes. Lines 3-6 minimize the set to only those sufficient for inducing the observed target state of each execution in the other. We leverage the delta debugging algorithm to enumerate subsets of the relevant causes and test their causality. Symbol $\mathcal{E}^{e_1}[s\downarrow_{e_2} / s\downarrow_{e_1}]_{l_\blacklozenge}^{l_\diamond}$ means executing e_1 up to the cause point $l_\diamond$, replacing its variable/value mappings in s with those from e_2, and continuing the execution up to the effect point $l_\blacklozenge$. Symbol $s \downarrow_{e_1}$ denotes the projection of state differences s on execution e_1. If the variables in the target state have the values from e_2, we say that the target state of e_2 was induced, written $\rightsquigarrow \Delta_\blacklozenge \downarrow_{e_2}$.

Note, in contrast to existing CSM approaches [114, 132], our minimization algorithm performs *two symmetric* causality checks. This is necessary to include causes via omission.

5.3.4 Confounding Free Execution Model

Recall that confounding occurs when new control flow or data dependences not in either original execution occur in a patched execution. By Theorem 5.3.1, we know that all the relevant causes are included by the dual slice. This suggests we only need to perform causality testing *within* the dual slice.

Conceptually, the essence of our new execution model is to construct a program containing only the behavior of the dual slice and all reexecutions for causality testing occur on the constructed program. Statement executions not in the dual slice should be prevented in order to minimize confounding.

Illustrative Example. Consider the example in Figure 5.5. Assume we start by using the target state {z ↦(-10,5)} at line 11. Assume the cause point is line 7 and we apply Figure 5.6 to minimize the causes at this point. The cut of the dual slice (Figure 5.5e) involves variables y and z. When we consider variable z with a regular execution model, we reexecute (c) up to the cause point and replace the value of z with -10. It induces the false branch outcome on line 7 but the true branch outcome on line 9, which is different than execution (b). Hence, {z ↦(-10,1)} is not considered a valid cause set.

With our new execution model, conceptually, we construct a program representing the dual slice, as in Figure 5.5d, in which lines 1, 2, 9, and 10 are precluded as they are not in the slice. Also, line 11 is no longer guarded by any predicate. Operands that are in the slice and have identical values in both executions are concretized (e.g. x on lines 5 and 7).

1. When l is not a conditional with $l \notin \mathcal{D}$, skips l.
2. When l is a conditional and it was in both executions with $branch(\mathcal{T}^{e_1}, l) \equiv branch(\mathcal{T}^{e_2}, l)$, unconditionally continue with the same branch as in the original executions.
3. When l is a conditional and it was in both executions with $branch(\mathcal{T}^{e_1}, l) \neq branch(\mathcal{T}^{e_2}, l)$ or l is in only one execution, evaluate the statement according to Rule 4) and follow the computed branch.
4. When l is not a conditional with $l \in \mathcal{D}$, validate that all the operands involved in some data dependence in $\mathcal{D}$ have the same data dependence as they did in the original executions, otherwise terminate and report confounding; For any operand not in any dependences in $\mathcal{D}$, denoted as x, set its value to $val(\mathcal{T}^{e_x}, l, x)$, and continue.

Figure 5.7.: Semantics of $\mathcal{E}[]$.

Again, let us determine the causality of variable z on line 7. We reexecute (c) up to the cause point using the original program. We replace the value of z with -10, *then continue execution with the program in Figure 5.5d.* Since lines 9 and 10 are not in the program, we avoid confounding and can induce the desired target state. Hence {z ↦(-10,1)}} is the minimal inducing cause set. Observe that it allows us to prune the relevant but not necessary cause {y ↦(0,1)}}. Applying Figure 5.6 transitively, we acquire a more concise failure explanation as shown in Figure 5.5f. □

Semantics of the New Execution Model. In the following, we discuss the semantics that allows achieving the effect of executing exclusively within the dual slice without constructing a new program. During minimization, we first reexecute the original program with normal semantics up to the cause point and then continue executing the program with *the new semantics* after state replacement until the effect point.

In the semantics, we assume the runtime availability of the dual slice $\mathcal{D}$ and the traces of the original two executions, denoted by $\mathcal{T}^{e_{1/2}}$. Without losing generality, we assume we are patching e_1 using information from e_2. The value of a variable x at a point l^{e_1} in the original execution e_1 can be queried from the trace by $val(\mathcal{T}^{e_1}, l, x)$. If an execution point l^{e_1} is a conditional statement, $branch(\mathcal{T}^{e_1}, l)$ queries its branch outcome in execution e_1. The semantics is presented in Figure 5.7.

Statement executions not in the dual slice are skipped when they are not conditional statements (Rule 1). When executing conditional statements, we cannot simply skip as we need to select a branch to proceed. Rules 2-3 specify the cases for conditional statements.

In Rule 3, if a conditional had different branch outcomes originally or was present in only one execution, the semantics evaluates the predicate and follows the computed branch. The essence is to allow the flexibility to take either branch based on the predicate evaluation in order to reason about the effect of state replacement when it is in the dual slice. If the statement is not in the dual slice, it does not matter which branch is taken because all non-conditional statements inside the branches must be skipped according to Rule 1. These statements must not be in the dual slice; otherwise, the conditional would have been in the slice according to the dual slicing algorithm.

Rule 4 handles non-conditional statement execution in the dual slice, for all the operands not involved in any dependences in the slice, implying that they must have identical values in the two executions, we concretize them with values from the traces to achieve isolation. For operands involved in some dependence, we ensure no data flow confounding.

This new model will not allow any confounded executions to go through, as can be inferred from the semantic rules.

Theorem 5.3.2 *A dual slice cut is sufficient within the new execution model.*

This theorem ensures that Figure 5.6 must be able to find a minimal sufficient set of causes inducing the target state because in the worst case, the cut is the minimal set. Informally, the theorem holds because reexecution is exclusively within the dual slice and hence replacing all the state in a cut leads to a reexecution equivalent to the part of the dual slice belonging to the opposing execution, and hence the target state.

A Practical Approximation. Unfortunately, the semantics in Figure 5.7 demands a prohibitively expensive implementation. It requires collecting traces with dependences and values. The traces and the dual slice have to be accessed during each reexecution. Each statement has to be instrumented to decide if it is in the dual slice (Rule 1) or perform

1. Rule 2 from Figure 5.7.
2. When l is a conditional and it was in both executions with $branch(\mathcal{T}^{e_1}, l) \neq branch(\mathcal{T}^{e_2}, l)$, evaluate the statement normally and follow the computed branch.
3. When l is a conditional and it was in only one execution e_x, follow the branch that was taken in e_x.
4. Otherwise, evaluate l as in a regular execution model.

Figure 5.8.: Semantics of the approximate execution model.

complex control (Rules 2-4). The overhead could easily be many orders of magnitude, not affordable for repeated reexecutions.

In practice, we observe that control flow confounding is the dominant confounding factor and data flow confounding can only affect the execution by causing control flow confounding in most cases. We hence propose a practical approximation that can completely prevent control flow confounding and mitigate data flow confounding. The approximate model ensures a patched execution can only follow dynamic branches taken by at least one of the original executions. Consequently, it enforces a control flow path composed of segments that occurred in either execution. What we do here is essentially constructing guard rails for the execution so that it can never deviate from the dual slice's control flow. Since data dependences heavily depend on control flow, the approximation can also mitigate data flow confounding. The semantics is presented in Figure 5.8.

Observe that the semantics does not require the runtime of the dual slice or dependence/-value traces for runtime checking, but rather just the control flow trace. This can be very efficiently represented and accessed by using bit streams that simply record the sequence of boolean branch outcomes. It does not skip statements. It hence avoids instrumenting all statements to decide if one can be skipped at runtime.

Theorem 5.3.3 *The approximate execution model is free of control flow confounding.*

The theorem can be inferred from the semantic rules. We implemented the approximate semantics and in practice it was able to suppress all confounding in our experience.

5.4 Evaluation

We implemented our technique using LLVM 3.0. We have also implemented the CSM and dual slicing approaches [114, 115, 125, 132] for comparison. Both implementations reflect the latest published designs [115,125]. The evaluation is in the context of automated debugging. The techniques contrast buggy and correct executions, using explanations for their different behavior as explanations of bugs. First, we compute explanations for a set of real world bugs by chaining together the computed causes. We contrast the explanations computed by the three different techniques. Second, we examine in depth how the problems that CSM faces affect its results in practice.

We used real world bugs taken from the repositories of open source programs. They include all deterministic bugs from `tar`, `grep`, and `make` in a one year period that we were able to reproduce. All the bugs in our study were non-crashing, semantic bugs that produce incorrect outputs. Table 5.1 presents the full set of programs and bugs. The first three columns identify the buggy program, bug ID, and the version of the program that actually contains the bug. The SSLOC column contains the static source lines of code computed with `sloccount`. The Alt. column identifies how a second, correct execution was selected. We used a correct input when the bug report also provided it, otherwise, we used predicate switching [134] to automatically synthesize a correct execution from the failing one. More information on acquiring a correct execution from a given failing execution resides in Sumner's survey paper [113]. We performed all experiments on a 64-bit 2.4GHz CPU with 12GB RAM using one core.

5.4.1 Full Explanation Comparison

Our first experiment uses each of the three techniques to compute an explanation for each bug. For each bug, we first identify the last observable failure and use that as the initial target state. CC and CSM select the last preceding definition of a target effect as the cause point to compute the causes. They also proceed transitively, using the computed causes as the new target state and the current cause point as the new effect point until there are no more causes to identify (e.g. the two executions have no state differences).

We contrast the results of the different techniques through their quality, scale, and efficiency. We measure *quality* through precision and recall with respect to a relevant, sufficient, and minimal explanation of why the correct and buggy executions differed. This is manually checked at each step of the computation. Precision (P) is the proportion of the dynamic statements in the computed explanation for a technique that coincide with the statements in the correct explanation. Recall (R) is the proportion of the dynamic statements in the correct explanation that are also identified by the computed explanation. We have to resort to manual inspection due to the lack of an automated oracle to tell us the ideal explanations for execution differences. As we show later, such ideal explanations are small enough for line by line human inspection.

We have done the following to mitigate threats to validity. First, we cross referenced the computed explanations with the root causes identified by the bug fixes or reports. Second, we calibrated our system using the Siemens suite before our experiments. We computed the explanations for the over 10,000 failing runs in Siemens using the corresponding passing executions of the provided correct versions and validated that these explanations capture the injected faulty statements as the root causes. Third, we also release the experimental results of the real world bugs at the same site for interested readers.

We measure the *scale* of a technique by the number of dynamic statements (Stmts) in the computed explanation. Finally, we measure efficiency in three ways: the number of steps or rounds of causal inference, the clock time required in seconds, and the number of reexecutions needed. Note that the clock time of CC includes dual slicing time. Table 5.1 shows the results. From these, we make several observations.

CC consistently yields the highest quality explanations. Dual slicing generally has good recall but poor precision because it does not minimize. CSM is unpredictable because it can arbitrarily include or exclude causes, however, it frequently fails to identify causes for even a single step of an execution. We shall explore the unpredictability of CSM further in the next section. In contrast, CC yields high precision and high recall for every computed

Table 5.1: Comparison of full explanations. Averages are arithmetic except for P & R, which are geometric. - means that the root cause could not be captured.

Program	ID	Version	SSLOC	Alt.	CC							CSM							Dual Slicing		
					Steps	Time	Tests	Stmts	P	R	Roots	Steps	Time	Tests	Stmts	P	R	Roots	Stmts	P	R
find	1	4.5.7	73k	switch	7	12	15	6	1.0	1.0	X	1	253	1260	0	0	0	-	185	0.03	1.0
gnuplot	2	4.5.0	144k	switch	11	44	33	10	1.0	1.0	X	11	141	469	10	1.0	1.0	X	148	0.06	1.0
gnuplot	3	4.4.0	139k	input	35	200	323	48	1.0	1.0	X	1	51	208	0	0	0	-	464	0.07	1.0
gnuplot	4	4.2.4	134k	input	146	961	337	129	1.0	1.0	-	127	950	1888	121	0.97	0.91	-	368	0.33	1.0
gnuplot	5	4.2.4	134k	switch	24	140	130	33	1.0	1.0	-	31	931	3012	38	0.87	1.0	-	237	0.14	1.0
grep	6	2.5.4	12k	switch	59	114	186	62	1.0	1.0	-	24	8263	1012	23	0.96	0.35	-	153	0.51	1.0
grep	7	2.5.4	12k	switch	45	156	327	69	1.0	1.0	-	33	183	1734	32	1.0	0.46	-	109	0.62	1.0
grep	8	2.5.4	12k	switch	27	49	78	27	1.0	1.0	X	24	168	1546	23	0.96	0.81	-	95	0.26	1.0
make	9	3.81.90	30k	switch	27	342	62	27	1.0	1.0	X	18	416	543	17	1.0	0.63	-	38	0.66	1.0
tar	10	1.22.90	20k	switch	5	22	8	3	1.0	1.0	X	5	50	221	3	1.0	1.0	X	3	1.0	1.0
tar	11	1.22.90	24k	input	30	124	125	48	1.0	1.0	X	1	110	332	0	0	0	-	61	0.79	1.0
tar	12	1.22.90	20k	input	9	53	121	20	1.0	1.0	X	1	66	296	0	0	0	-	1239	0.01	1.0
tar	13	1.22.90	20k	switch	11	43	28	10	1.0	1.0	X	6	439	2117	5	1.0	0.5	-	1270	0.01	1.0
tar	14	1.23	21k	input	17	80	87	23	1.0	1.0	X	5	165	709	15	0.73	0.48	X	25	0.92	1.0
tar	15	1.23	21k	switch	5	22	15	4	1.0	1.0	X	5	228	1283	4	1.0	1.0	X	557	0.01	1.0
				Average	30.5	157.4	125	34.6	1.0	1.0	-	19.53	827.6	1108.7	19.7	0.22	0.26	-	330.1	0.14	1.0

explanation. For the bugs, it captures 11 of 15 root causes whereas CSM *fails* to do so in 11 of 15 cases. Where CC failed to identify root causes, denoted by -, it still explained why the two *executions* differed, thus the precision and recall. In those cases, the second execution was too different to meaningfully explain the bugs as well.

The extra reexecutions for CSM make it slower than CC, even when it computes fewer steps. On average, CSM takes 13.8 minutes to compute an explanation, even though it produces less of the correct explanation. In contrast, CC takes 2.6 minutes on average because the extra dual slice information allows it to avoid considering all memory differences as potential causes. This reduces the number of necessary reexecutions by up to two orders of magnitude.

CC produces more concise explanations than dual slicing. The precision numbers show that CC is more precise than dual slicing, 1.0 vs 0.14. On average, CC produces explanations of 35 dynamic statements, while dual slicing produces 330 statements.

This experiment illustrates that CC produces superior explanations in terms of quality, efficiency, and scale.

5.4.2 Why and How CSM fails

A single incorrect cause at any point of the full chain computation can cascade through the rest of the computation, causing more incorrect causes. It is hence difficult to determine the reasons behind the incorrectness by simply looking at the full chains. Our second experiment examines *why and how* CSM missed or erroneously included causes on a per-step basis. Note that CC does not encounter these problems for the given benchmarks, and dual slicing does not do minimization. Thus, we focus only on CSM for this experiment.

We first computed the causes for each step using CSM as in the first experiment. For each step, we also supply the same (CSM) target state and the same cause point to CC and compare the resulting causes from the two approaches. This allows us to quickly observe any effects from confounding.

In this per-step fashion, we checked the results of CSM for missing causes (M), extra causes (E), or failure to identify any causes (F). These are the ways that the technique can fail. We also checked *why* these failures occurred, including control flow confounding (CFC), data flow confounding (DFC), and execution omission (O). Table 5.2 contains these results.

Table 5.2: CSM difficulties. This includes symptoms: (M)issing causes, (E)xtra causes, and complete (F)ailure. It also lists reasons why: control and data flow confounding (CFC/DFC) or (O)mission.

ID	M	E	F	CFC	DFC	O
1	X	-	X	X	-	-
2	-	-	-	-	-	-
3	X	X	X	X	-	-
4	X	X	-	X	-	X
5	X	X	-	X	-	-
6	X	-	X	X	-	X
7	X	-	-	-	-	X
8	X	X	-	X	-	X
9	X	X	-	X	-	X
10	-	-	-	-	-	-
11	X	X	X	X	-	-
12	X	X	-	X	-	-
13	X	-	X	X	-	-
14	X	X	X	X	-	-
15	-	-	-	-	-	-

CSM suffers from all three problems. It misses causes in almost all benchmarks (12 out of 15), has extra causes in 8 out of 15, and fails to produce any causes for a step in 6 out of 15 cases. These failures resulted both from omission and from confounding, although confounding was the more frequent cause.

Control flow confounding causes errors in most of the CSM explanations. In 11 out of 15 cases, the CSM explanations are directly impacted by control flow confounding. This shows that control flow confounding is a real world challenge that we must address.

Data flow confounding does not directly impact CSM. While close inspection indicates that some data flow confounding occurs, it impacts the executions only through control flow confounding. As CC prevents control flow confounding, the impact of the corresponding data flow confounding is also suppressed. For example, data flow confounding may lead to an incorrect branch, but CC forces the execution back to the correct branch through its execution model.

Together, these fine grained comparisons allow us to see that omission and confounding do indeed impact existing techniques. Furthermore, taken with the results in Table 5.1 they show that CC is resilient when faced with them.

5.4.3 Example of Resulting Explanations

Next, we demonstrate a failure explanation generated by CC and explain how CSM fails to compute that explanation. This chain is for bug 13. Version 1.22.90 of `tar` has a bug when using the `--backup` option. When extracting files from an archive, this option copies any already existing files into a backup directory, preventing these files from being overwritten. When extracting a directory that already exists, however, it *appears* to incorrectly prevent files from being extracted.

We used predicate switching to dynamically patch the buggy execution and derive a correctly behaving execution. Both the buggy and the switched executions first extract some files before trying to extract a directory that already exists. The switched execution renames the extracted directory so that it does not conflict with the existing one, and it correctly extracts files to the new directory without error. However, the buggy execution appears to have not extracted any files at all, even the previously extracted ones.

Figure 5.9 shows a simplified version of the relevant code, as well as the explanation by CC, which is slightly shortened for readability. First, predicate switching renames one of the extracted directories from ”dir2” to ”dir”. Next, a call to mkdir() fails in the buggy execution, returning −1 because ”dir2” already exists. In contrast, the call succeeds in the switched execution and returns 0. This difference (0 vs. −1) gets propagated through the variable status back into extract_archive(), where it makes the condition on line 18 **True** only in the failing execution, indicating an error when extracting ”dir2”. So the buggy execution calls undo_last_backup(). This actually *replaces all of the extracted files* with the original backups. As a result, all of the files extracted before ”dir2” *appear* to never

Code Summary

```
 1 int read_header_primitive():
 2    file_name = "dir" vs. "dir2"
 3
 4 int extract_dir(file_name):
 5    tmp = mkdir(file_name)
 6    ...
 7    status = tmp
 8    if status:
 9       if errno == EEXIST && IS_DIR(file_name):
10          pass
11       elif !maybe_recoverable(filename):
12          mkdir_error(file_name);
13    return status
14
15 void extract_archive():
16    ...
17    status = extract_dir(file_name)
18    if status && backup_option:
19       undo_last_backup()
```

Explanation

At 2, file_name **is** **"dir"** vs. **"dir2"**.
At 5, tmp **is** **0** vs. **-1**.
At 7, status **is** **0** vs. **-1**.
At 13, the **return** value **is** **0** vs. **-1**.
At 17, extract_dir returns **is** **0** vs. **-1**.
At 18, (status && backup_option) **is** **False** vs. **True**.
So **"undo_last_backup()"** **is** called, overwriting the extracted files with the original ones.

Figure 5.9.: Example of a derived explanation using our technique

have been extracted, even though they were. In fact, the original bug reports for this failure assumed the files had not been extracted, as well, but our generated explanation clearly shows that they were first extracted and then incorrectly overwritten.

The root cause is that extract_dir() should not fail even if mkdir() fails due to the existence of the directory, because extracting to an existing directory should not cause problems. A `tar` developer can see this from the computed explanation (on the bottom of Figure 5.9) and know how to construct a fix. Indeed, the applied fix set status to 0 on line 10.

Note, CSM cannot construct this explanation. On line 13, confounding prevents further analysis of the bug. First, the condition on line 9 only executes in the failing execution, where it is **True**. CSM replaces the value of tmp at line 7 to produce the failing status at line 13, but the condition on line 9 evaluates to **False** this time because it also requires a failing value for errno. Hence, CSM proceeds to line 12, which reports an unrecoverable error and terminates. This confounding prevents the identification of tmp alone as the cause. Additional confounding not shown here also prevents replacing both errno and tmp from inducing the failing status.

5.4.4 Threats and Limitations

We have shown that CC is effective at explaining why two executions are different, but there are limits to the technique, our evaluation, and what may be inferred from it.

We first note that explaining why a buggy and correct execution differ does not always provide a useful explanation of a bug, as observed in 27% of our generated explanations.

Also, manual examination of execution differences risks human error. Most of the explanations generated by CC are short enough that we can be confident of our inspection.

Finally, again, comparative causality is presently limited to examining deterministic bugs. This inherently follows from exploiting reexecution within the technique.

5.5 Related Work

The most relevant work is causal state minimization (CSM) that was originally introduced by Zeller [132], and subsequently improved by others [28, 114, 115]. In contrast to CSM, our CC model avoids confounding, handles execution omission by symmetric analysis, and is much more efficient.

Ermis et al. have used theorem provers on abstract error traces to infer causes of bugs that remain invariant over the failing executions of a program [41]. While the technique can also infer semantic causes of bugs, the inference process requires many calls to a theorem prover and has not been shown to scale. It also requires a generalized specification of the failure over all possible executions, which can be difficult to provide.

Podgurski and Clarke also examined the notion of *semantic dependences* [96]. Semantic dependences are static dependences indicating when one statement can potentially impact the result of another. In contrast, our approach identifies dynamic dependences that can be blamed for another statement producing a particular result.

Recently, Rößler et al. also noted problems with Zeller's original approach, although they did not delve into what these problems were [103]. They also produce a technique for explaining bugs, but it is based on test generation and requires a strong oracle to evaluate each new test.

Traditional dynamic slicing [76] is a technique that captures dynamic data and control dependences. It has been extensively examined for its usefulness in debugging [119]. Dynamic slices are usually problematically large and suffer from execution omission. Dual slicing is a kind of dynamic slicing technique that *compares two executions* and extracts the differing dependencies between the two [65, 125]. It forms the initial basis of our technique. In contrast, our computed explanations are much smaller due to state replacement and minimization.

Several satisfiability based techniques also strive to precisely localize and potentially explain failures, either within a single program [51, 68] or when comparing correct and incorrect versions [14, 97]. The present limitations in constraint solving, however, have thus far mostly limited these techniques to programs of a few thousand lines of code. In contrast, our technique explains failures in programs with well over 100K lines.

Our technique requires that the executions of interest be reproducible. Tools that aid failure reproduction, for instance, can make this more feasible in practice [9]. Other tools also make it easier to repeatedly analyze the behavior of a program at different points in the history of a failing execution [71].

5.6 Conclusions

We presented a novel causal inference technique called comparative causality. It allows precise and concise explations for the differences between two executions at a very fine granularity. It advances the state of the art in three aspects: it improves *robustness* of underlying state replacement techniques by preventing confounding through novel execution models; it handles execution omission errors by analyzing two executions symmetrically; and it substantially improves efficiency by leveraging dual slicing. Evaluation on a set of real world bugs shows that the proposed technique can generate high quality explanations at low cost.

6 FINDING APPROXIMATELY CORRECT EXECUTIONS

Execution comparison provides a means for developers to better understand a program's behavior. Given two executions of a program, a developer may compare and contrast the control flow paths taken and the values operated upon during these executions in order to better understand how and why the program behaved differently for each. When one of the executions fails and the other is correct, such comparison forms a basis for automated techniques toward debugging like that presented within Figure 2.2. These techniques extract the behaviors and properties of the respective executions that correspond with the correct and incorrect program behaviors. The differences in behavior provide evidence for what parts of a program might be incorrect and why. This helps to reduce the developer burden in finding, understanding, and fixing a bug.

The utility of such techniques depends on precisely which executions they compare against each other [8, 100, 114, 132]. When the correct execution does not follow the same path as the failing one, automated analyses may derive little more than the fact that the executions are different because there are no fine-grained differences from which comparison can extract more information. One might then try using an execution that follows the exact same path as the failing one. Unfortunately, the failing execution's behavior is incorrect, so executions that are similar to it may also behave as if they were incorrect. As a result, the differences between the executions do not provide insight on why the failing execution is buggy. Deciding which executions should be compared is a significant problem when using execution comparison.

The difficulty in the problem arises because of a mismatch between the respective objectives of execution comparison and of debugging in general. Debugging involves comparing the failing execution with the programmer's model or specification of how the execution should have behaved. Differences between the model and the actual behavior of the execution precisely capture an explanation of where and why the program is not behaving correctly, and ideally how the developer may change the underlying program to remove the fault. In contrast, execution comparison finds differences between two actual executions. These differences explain how and why those particular executions behaved differently. For example, if one execution receives an option '`-a`' as input and the other receives an option '`-b`' as input, execution comparison can explain how the semantically different options lead to semantically different program behavior.

The mismatch between debugging and execution comparison arises because the semantic differences that debugging strives to infer should explain the failure. This inherently requires comparing the failing execution against the correct, intended execution. However, only the correct version of the program can generate that correct execution, and the correct program is exactly what automated debugging should help the programmer create. Because this execution is unavailable, we must settle for an execution that is instead *similar* to the correct one. In order for execution comparison to be useful, the execution against which we compare the failing execution must *approximate* the correct one without knowing a priori how the correct one should behave. We call this the *peer selection* problem. Given a particular failing execution, another execution, called the *peer*, must be chosen for comparison against the failing one. This peer must be as similar as possible to the unknown correct execution.

```
1 x = input()
2 if x > 5:
3   print('one')
4 elif x > 0:
5   if x > 1:
6     print('two')
7   else:
8     print('three')
```

(a)

	Executions			
input	2	6	4	1
path	1	1	1	1
	2	2	2	2
	4	3	4	4
	5		5	5
	6		6	8

(b)

Figure 6.1.: (a) A trivial faulty program. The `x>1` on line 5 should instead be `x>2`. This causes the program to fail when the input is `2`. (b) Example executions of the program on different inputs. Input 2 yields a failing execution; the others are correct.

For example, consider the program snippet in Figure 6.1a. The `if` statement on line 5 is incorrect; instead of `x>1`, it should instead be `x>2`. As a result, if the input is `2`, the program prints ‘two’ when it should print ‘three’. If we use execution comparison to debug the failing execution when the input is `2`, we must first find a suitable peer. Figure 6.1b presents some possible correct executions along with the input that yields each execution and the path, or the trace of the statements in the execution.

First consider the execution with input `6`. Of those listed, this execution differs the most from the failing execution. They are different enough that execution comparison provides little insight. They take different paths at line 2 because they have different inputs and because `6>5` is true, but `2>5` is not. However, this does not imply anything about the correct behavior when the input is `2`. Similarly, if we consider input `4`, the execution follows the exact same path as the failing execution except the execution is correct. This also yields no insights on why the original execution failed. Finally, let us consider the input `1`. This execution follows the exact path that the failure inducing input `2` would follow if the program were correct, and it produces the output that the failing execution should have. Comparing this execution with the failing execution tells us that lines 2 and 4 are likely correct because they evaluate the same in both executions. Line 5, however, evaluates differently, so we should suspect it and examine it when debugging the program. Because this execution behaved most similarly to how the failing execution *should have* behaved, it was able to provide useful insight on why the program was faulty.

This chapter considers 5 techniques that can select or create a peer execution when provided a failing one. Their approaches range widely from selecting known test inputs to synthesizing entirely new executions that might not even be feasible with the faulty version of the program. We objectively examine their fitness for selecting peers by using these techniques to derive peers for known bugs in real world programs. We use the patched versions of the programs to generate the actual correct executions, the ideal peers, and compare the generated peers against them to discover which techniques created peers most similar to the correct executions. We examine the strengths and weaknesses of the techniques with respect to properties of the bug under consideration, the failure generated by the bug, and the faulty program as a whole. Developers can use this information to automatically or interactively help choose the most effective peer selection technique for understanding a particular failure. In summary, the contributions of this chapter are:

1. We survey and implement existing techniques that can select peers for execution comparison. We consider their applicability with respect to properties of the program and failure that the developer knows a priori.
2. We objectively examine the fitness of each technique for peer selection on 20 real bugs in 3 real world programs. They represent the full set of reported bugs for these programs during a roughly one year period. Using the corrected versions of the programs as extracted from their source repositories, we generate the expected correct execution and compare the peers from the techniques against it.
3. We examine the real world bugs to infer when the given techniques may or may not be applicable if more information about a bug is already known.

The ideas presented in this chapter have previously been published by the author in the proceedings of ISSTA 2011 [113].

6.1 Selecting Executions

In this section, we review five existing techniques for selecting execution peers. Not all of the techniques were originally designed with peer selection in mind, but those that were not still generate peers either as a side effect or intermediate step toward their respective intended purposes.

6.1.1 Input Isolation

The same input that causes an execution to behave unexpectedly, henceforth called a *failing input*, can sometimes be reduced to produce a valid alternative input. This new and simpler input might yield an execution that does not fail. This insight was used by Zeller in [132] to produce two executions, one passing and the other failing, whose internal states were then compared against each other. Given a failing input, Zeller used his previously developed delta-debugging technique [133] to simplify the input and isolate a single input element such that including this element caused a program to crash, and excluding the element caused the program to succeed. In his technique, the execution *without* the inducing element was used as a peer for the execution including the element. The underlying intuition is that the less the inputs for the two executions differ, the less the executions themselves should differ, as well.

The central idea of the technique is to use a modified binary search over the possible inputs of the program. A subset of the original input is extracted and tested on the program. If the subset of the original failing input also yields a failing execution, then that new input is used instead of the original, effectively reducing the size of the failing input. In contrast, if the new input yields a passing execution, then this acts as a lower bound on the search. Any later input that is tested must at least include this passing input. In this way, the algorithm terminates when the failing input and the passing input are minimally different[1].

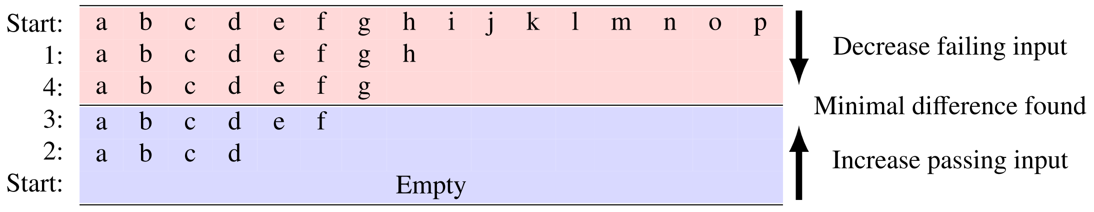


Figure 6.2.: Delta debugging can produce two minimally different inputs, yielding an execution to use as a peer.

Figure 6.2 presents a brief example showing how delta debugging generates inputs for peer executions. Suppose that a program crashes when the characters '`b`' and '`g`' are present in the input. The full input `a..p` thus causes a program to crash, as seen on the first line. We also assume that an empty input yields a passing execution, as denoted on the last line. The approach tries to narrow down exactly which portions of the input are responsible for the crash by selectively removing portions of the input and re-executing the program on

[1]Differences between inputs are locally minimal as in [133].

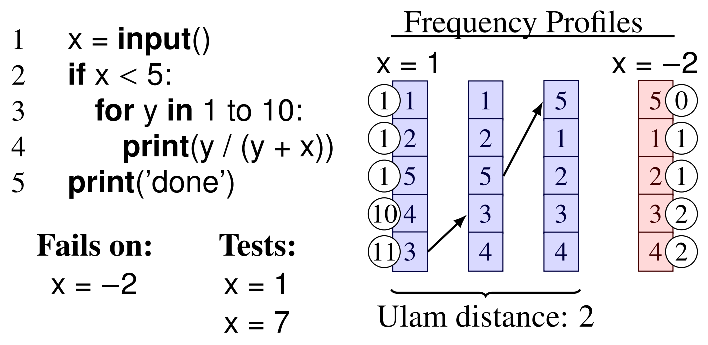
the new input. In the first test, the algorithm selects `a..h`. When the program executes this input, it fails, so the failing input reduces to `a..h` on step 1. On step 2, `a..d` is selected. Because this does not include ‘`g`’, the execution passes, and the passing input increases to `a..d`. Steps 3 and 4 continue the process until the failing input contains `a..g` and the passing input is `a..f`. Because they differ by only ‘`g`’, the inputs are minimally different, and the technique selects the passing execution as a peer.

6.1.2 Spectrum Techniques

Modern program development includes the creation of test suites that can help to both guide the development process and ensure the correctness of a program and its components. One straightforward approach for finding a peer is to simply select an execution from this already existing test suite. This allows the developer to reuse the existing work of generating the tests, but the developer must still select the specific execution from the test suite to use. Execution profiles, or *program spectra*, are characteristic summaries of a program’s behavior on a particular input [56] and researchers have long used them as a means of establishing test suite sufficiency and investigating program failures [36,56,66,100,101]. In particular, spectra have been used to select a passing execution from a test suite with the intent of comparing the execution to a different, faulty one [100]. Thus, spectra provide an existing means of selecting an execution peer.

We specifically focus on the spectrum based peer selection in [100]. This paper introduces the *nearest neighbor model* for selecting a passing peer. For every passing test in a suite, as well as the failing execution itself, we compute a frequency profile that records the number of times each basic block of the program executes during the test. Next, we evaluate a distance function with respect to the failing profile against each passing profile. This determines which profile, and thus which test, is most similar to the failing execution.

A key observation in [100] is that the nature of the distance function is critical to the utility of the overall technique. A Euclidean distance, where each element in the frequency profile is a component in a vector, can lead to illogical results. For example, if one execution has long running loops and another does not, such a distance may ignore similarities between the parts that are the same in both executions. General inconsistencies with distance functions influenced by the concrete number of times a program executes a basic block led the authors to use the Ulam distance, an edit distance between two sequences of the same unique elements. Each profile is first transformed into an ordered sequence of basic blocks of the program, sorted by their execution frequencies. Note that *order* is the discriminating factor, the actual frequencies are not even present in the profile. The distance function between profiles is the edit distance [61], or the number of operations necessary to transform the sorted list of basic blocks from the test into that from the failing execution. The lower the distance, the fewer dissimilarities between the profiles, so the algorithm selects the test with the lowest edit distance as the *nearest neighbor*, and peer.



```
1   x = input()
2   if x < 5:
3     for y in 1 to 10:
4       print(y / (y + x))
5   print('done')
```

Fails on:
x = −2

Tests:
x = 1
x = 7

Figure 6.3.: The nearest neighbor model's peer finding approach. Basic blocks (identified by the numbers inside the boxes) are sorted by frequency (circled numbers), then the test profile with the lowest edit distance to that of the failure belongs to the peer.

Consider the example in Figure 6.3. For simplicity, it uses frequency profiles over the statements instead of basic blocks. The program listed here has 5 statements which should all execute if the program behaves properly. Line 1 reads a number, `x`, from input. If `x` is less than 5, execution enters a loop that computes and prints a new value. Finally, the program prints 'done' when it is complete. Observe, however, that when `x=-2`, the program crashes on the second execution of statement 4 due to integer division by 0. This gives the failing execution the sorted frequency profile shown on the far right. Basic block IDs are shown in the cells, while their execution frequencies are shown in adjacent circles. Note that the failing execution omits statement 5 and misses the final check of the loop guard. Thus, statements 5 and 3 are out of place. A correct input of `x=1` instead executes the full loop and statement 5, giving the leftmost frequency profile. To compute the Ulam distance between the two, we count the minimum number of blocks that need to be moved. In this case, it is 2. Because the only other test `x=7`, has a distance of 3, the technique chooses input `x=1` as the input of the peer execution in this example.

6.1.3 Symbolic Execution

Both input isolation and spectra based approaches derive peers from existing input. This may not always be possible. For instance, if there are no passing test cases then the spectrum based approach will fail. With no *similar* passing cases, it may simply yield poor results. One alternative is to generate new inputs for a program with the goal of creating inputs that inherently yield executions similar to an original failing run. This goal is achievable through recent work in test generation using *symbolic execution* [8, 22, 23, 48, 105, 129].

When a program executes symbolically, it does not merely execute any operation that might use or depend on input data. Instead, it builds a formula reflecting what the result of the operation could be for *any* possible input value. When it executes a branching instruction like an if statement, the symbolic execution takes all possible paths, constraining the possible values along those paths appropriately. For example, in Figure 6.4, the variable `x` receives an integer value from `symbolic_input()` on line 1. Because the input is symbolic, `x` represents all possible numeric inputs at that point. On line 2, execution reaches an `if` statement and must consider both branches. Thus, along the path where the condition is true, the execution adds the constraint `x>5` to the formula for `x`. Along the other path, it adds the constraint `x<=5`.

The sequence of branch constraints along a path, or *path condition*, determine what conditions on the input must hold for the path to be followed. By solving these constraints at a certain point in the symbolic execution, concrete inputs sufficient to reach that point can

```
x = symbolic_input()
if x > 5:
  print('greater')
elif x > 0:
  if x % 2 == 0:
    print('even')
  else:
    print('even')
```

Figure 6.4.: Buggy code snippet. The last print statement should print “odd”.

be discovered. This is the common underlying approach to test generation using symbolic execution. Precise mechanisms, however, differ between approaches. For instance, programs may have infinitely long or infinitely many different execution paths, so no approach covers them all. The way a particular test generation system chooses to explore program paths impacts the generated tests. In the context of peer selection, we desire to generate executions similar to an already existing path. One example of such a strategy is presented in [8].

The test generation approach in [8] creates new tests that are similar to a failing execution in order to aid fault localization in PHP programs. It first uses the failing execution as a ‘seed’ to the process, collecting its symbolic path condition. It then generates additional paths by negating the conjuncts along the path and solving to generate inputs for those paths as well. The process then continues, always prioritizing the generation of tests along paths that measure as ‘similar’ to the original failing path condition.

A simplified version of the algorithm is reproduced in Figure 6.5. In fact, this is a general test generation algorithm, but as we shall discuss later, we can customize the function selectNextTest() to direct it toward executions similar to the failing one. Line 2 initializes a set of candidate tests to possibly add to the test suite. To start, this holds a test for each branch encountered in the failing execution such that the test takes the alternate path at that branch. E.g., if the failing execution took the true path, the test would take the false path for that branch. Lines 3-6 iteratively select a test to add to the test suite and generate new candidates until a time limit expires or there are no more paths to explore. A more

```
GENERATENEWTESTS(seed)
Input:   failing - a failing test
Output:  a set of new tests
1: tests ← Ø
2: toExplore ← GETNEIGHBORINGTESTS(failing)
3: while toExplore ≠ Ø and time has not run out do
4:     test ← SELECTNEXTTEST(toExplore)
5:     tests ← tests ∪ test
6:     toExplore ← toExplore ∪ GETNEIGHBORINGTESTS(test)
7: return tests
```

```
GETNEIGHBORINGTESTS(seed)
Input:   seed - a previously selected test
Output:  tests with paths branching from that of seed
1: neighbors ← Ø
2: $c_1 \wedge c_2 \wedge \ldots \wedge c_n$ = GETPATHCONDITION(seed)
3: for all $i \in 1, 2, \ldots, n$ do
4:     path ← $c_1 \wedge c_2 \wedge \ldots \wedge \neg c_i$
5:     neighbors ← neighbors ∪ SOLVE(path)
6: return neighbors
```

Figure 6.5.: A test generation algorithm directed by the objective function SELECTNEXTTEST().

complete presentation of the algorithm also handles such optimizations as not following the same path more than once. We refer the reader to the original papers for the details, which are not relevant to the discussion here.

As previously mentioned, this is a general test generation algorithm. The way it selects tests to further explore and add to the test suite is controlled by the objective function SELECTNEXTTEST(). This function guides the test selection toward whatever goals a developer may have in mind. In [8], the goal was similarity with a failing execution, so they additionally developed a heuristic objective function using path conditions to select such tests. Given a sequence of branch conditions, or path condition, encountered in a program, their similarity score is the number of the same conditions that evaluated to the same values in both executions.

Consider again the program in Figure 6.4. The program has a bug such that when line 1 assigns the value 1 to `x`, the program executes the faulty print statement on line 8. The path condition for this execution is $\langle 2_F \wedge 4_T \wedge 5_F \rangle$, meaning that the condition on line 1 evaluated to false, the condition on line 4 evaluated to true, and the last condition on line 5 evaluated to false. If the candidate tests in *toExplore* are the inputs -1, 2, and 10, then their respective path conditions are $\langle 2_F \wedge 4_F \rangle$, $\langle 2_F \wedge 4_T \wedge 5_T \rangle$, and $\langle 2_T \rangle$. Observe, input -1 yields the same value for condition 1, so its similarity is 1. Input 2 has similarity 2, and 3 has similarity 0 for the same reason. Because input 2 has the highest similarity, it is the next test added to the test suite and further explored.

We decide if the generated inputs lead to passing executions with an oracle. We further select from the pool the one that is most similar to the failing run using the same path condition based criterion.

6.1.4 Predicate Switching

Instead of generating input that will force a buggy program to yield a strictly correct, passing execution, some approaches for selecting peers relax the notion of correctness. This allows peers that may represent slightly incorrect executions or normally infeasible executions. Instead of finding some new input for the program, these techniques can patch the program or even the execution itself at runtime to create a new execution that behaves similarly to a failing one. Because incorrectness is undesirable in a peer, these approaches must be careful in how they modify a program's behavior.

One such dynamic patching technique is *predicate switching* [134]. First designed for fault localization, it generates peers as a part of its core process. The authors of this paper noted through experiments that less than 10% of an execution's instructions relate to computation of values produced as output. As a result, they inferred that they could correct most faults by fixing only the control flow of an execution. That is, by forcing an execution to follow the desired control flow, it would behave, to an observer, exactly the same as a correct program. This insight was extended to a technique for constructing peers when given a failing execution.

One way to alter the control flow of a program is to switch the result of evaluating a branch condition, or predicate. For example, if the tenth predicate of a program originally evaluated to true, predicate switching would dynamically patch the execution and flip the outcome to false, forcing the execution along the false path instead of the true path. Predicate switching combines this with various search strategies for such a dynamic patch, considering each individual predicate instance as a candidate. If the observable behavior of a patched execution matches the expected behavior of the failing execution, then the search found a patch, and this patched execution can be returned as a peer.

```
1 x = [101, 102, 103]
2 for i in 0 to 3:
3   if x[i] % 2 == 0:
4     print(x[i])
5 print('done')
```

Figure 6.6.: Buggy code snippet. Index 3 is invalid when accessing the array x.

The code snippet in Figure 6.6 presents a small example where predicate switching is useful. This program creates a list with 3 elements, but its loop body executes for indices 0 through 3 inclusive. When the execution accesses index 3 of the list on line 3, it throws an exception, terminating the program. The sequence of predicates encountered within the failing execution is $\langle 2^1_T, 3^2_F, 2^3_T, 3^4_T, 2^5_T, 3^6_F, 2^7_T \rangle$, with superscripts as timestamps, distinguishing the different instances of the same static predicate. The program crashes after it executes 2^7_T. Predicate switching switches each predicate in order starting with 2^1 until it finds a patch. The second iteration of the loop, when $i = 1$ should print out '102', so the algorithm will not find a valid patch until after that iteration. When the technique switches the fifth predicate, 2^5_T, to false, the program prints 'done' and successfully ends. This is the expected behavior of the correct program, so the search ends with the dynamic patch of 2^5 providing the peer.

Note that the result does not perfectly match the control flow of the correct program. Namely, the correct program also expects a third iteration of the loop to complete, even though it does not do anything. Because the last iteration was not necessary for generating the expected observable behavior of the execution, predicate switching considers that loss an acceptable approximation for the peer.

6.1.5 Value Replacement

Value replacement [62, 63] is a technique related to predicate switching in that it also modifies a program's behavior dynamically to create an approximately correct execution. While predicate switching works on the boolean domain of branch conditions, value replacement replaces other values within a program as well, allowing it to dynamically patch value based errors in addition to control flow errors.

The underlying abstraction used by value replacement is the *value mapping* for a statement, or the set of values used by a particular dynamic instance of a statement during an execution. By changing the value mapping for a statement, value replacement can cause a failing execution to behave observably correctly. Indeed, predicate switching is a special case of value replacement, wherein it only considers the values at branching statements. The difficulty of the technique lies in finding appropriate value mappings to use at a given statement. Unlike predicate switching, which only needs to consider the values true and false, replacing all values at a statement gives intractably many different options for what the value mapping in a patch might be. It is not knowable *a priori* which of these options might correctly patch the execution. To make a solution feasible, value replacement only considers the values observed in the test suite of a program, as these are at least known to be valid alternatives. The first step of the technique constructs a *value profile* that holds the value mappings for a statement observed in any of the tests of a test suite. For each dynamic statement in the failing execution, the previously observed value mappings are then used to generate dynamic patches.

The value profile for a statement may still have many different mappings. Instead of directly applying each of the value mappings, the technique only looks at a subset of the value mappings. For a statement, these mappings use the values from the profile that are:

the minimum less than the original value, the maximum less than the original value, the minimum greater than the original value, and the maximum greater than the original value.

For example, consider the statement `z = x + y` where `x` and `y` have the values 1 and 3 in the failing execution, respectively. If the value profile contains `x = 0,1,2,3` and `y = 0,1,2,3`, then the potential value mappings of the statement consists of {x=0,y=0}, {x=0,y=2}, {x=2,y=0}, {x=2,y=2}, {x=3,y=0}, and {x=3,y=2}.

6.2 Analytical Comparison

In this section, we classify the techniques and discuss their applicability. We empirically compare the techniques in Section 6.3. Table 6.1 summarizes these techniques. Column `Class` classifies the techniques based on how they select peers. Column `Oracle` lists the power of the testing oracle that must be available for applying the technique. Column `Test Suite` notes whether or not a given technique requires a test suite.

Table 6.1: Common features among peer selection approaches.

Technique	Class	Oracle	Test Suite
Input Isolation	input synthesis	complete	no
Spectra Based	input selection	test suite	yes
Symbolic Execution	input synthesis	complete	no
Predicate Switching	execution synthesis	1 test	no
Value Replacement	execution synthesis	1 test	yes

Classification The first three techniques either generate new inputs or use existing inputs to select peer executions. We call them input based techniques. These technique can only generate executions that exercise feasible paths in the faulty program (recall that a feasible path is an execution path driven by a valid input).

The remaining two techniques synthesize executions from the failing execution. They may generate executions that are not feasible under the original faulty program.

Oracle Both input isolation and symbolic execution require complete oracles in theory. These techniques check whether or not arbitrary executions pass or fail, so the oracle must work on any execution. In fact, input isolation also needs to determine whether or not an execution fails in the same way as the original, e.g. crashing at the same point from the same bug. In practice, implementations of the techniques use approximate oracles instead. For example, simple oracles can be composed for certain types of failures (e.g. segfaults). Spectra based selection examines the passing cases from an existing test suite, so an oracle must be able to determine whether or not each individual test passes or fails. Such an oracle is weaker than a complete oracle, because it is usually a part of the test suite. The execution synthesis techniques only check that the patched executions are able to observably mimic the expected behavior. This means that an oracle must be able to check for the expected behavior of a single execution. This is the weakest oracle.

Test Suite Both the spectra based and value replacement approaches require a test suite to function. As noted earlier, test suites are commonly used during development, but the efficacy of these particular approaches will depend on how the tests in the test suite relate to the passing and failing executions. For example, if the test executions do not follow control flows similar to those in the failing execution, the spectra based approach will yield a poor peer. On the other hand, if statements within tests never use the values that the failing execution needs to patch and correct its own behavior, then value replacement will fail entirely.

6.3 Experiment

We implemented all the techniques using the LLVM 2.8 infrastructure and an additional set of custom libraries, requiring about 11,000 lines of C and C++ and 1,500 lines of python in addition to using the KLEE infrastructure for symbolic execution [22]. We ran all tests on a 64-bit machine with 6 GB RAM running Ubuntu 10.10.

For input isolation, we performed delta debugging both on command line arguments passed to the faulty program and on files that the program uses during the buggy execution. For our spectrum based selector, we computed frequency profiles over the test suite for each program that existed at the time the bug was not yet fixed. Each of our analyzed programs contains a test directory in its repository that contains regression tests for bugs that developers previously fixed. These suites contain 150–1,000 tests.

We built our symbolic execution based selector using a modified version of the KLEE engine. Instead of generating tests with a breadth first branch coverage policy, we modified the search heuristics to initially follow the path of the failing execution as much as possible, then explore paths that branch off from the original execution. If the constraints along one path are too complex, the system gives up on solving for input on that path and continues test generation on other paths. Once the system finishes (or the time limit expires), it selects the passing test with the a path condition most similar to that of the failing run as the peer.

We performed an empirical study of how these techniques behave for executions of a set of real bugs. The study examines (1) the overhead incurred by using each technique in terms of time required, (2) how accurately the peers generated by each technique approximate the correct execution, and (3) how the approximations from different techniques compare to each other and in which cases one may be more appropriate than another.

Evaluating the second objective is challenging. Given the selected peer, an execution of the existing buggy program, we must objectively measure how similar it is to the intended correct execution. The more similar, the better the peer approximates the behavior of the correct execution. This necessitates both knowing what the correct execution should be and having a way to compare it with the chosen peer. In order to know what the correct execution should be, we limit our study to bugs that maintainers have already patched in each of the above programs. To reduce the possible bias caused by bug selection, we extracted the bugs for each program by searching through a one-year period of the source repository logs for each program and extracting the correct versions where the logs mentioned corrections and fixes for bugs. We also extracted the immediately preceding version of the program to capture the faulty program in each case. Excluding those that were not reproducible or objectively detectable, we have 10 versions for `tar`, 5 for `make`, and 5 for `grep`. By running the failing version of the program on the input that expresses each bug, we generate the execution for which we desire a peer. By running the corrected version, we generate the perfect peer against which we must compare the selected peers. The precise mechanism for comparing the executions is discussed in Section 6.3.1.

6.3.1 Measuring Execution Similarity

We need a way to objectively compare the similarity of these executions. We do this by measuring along two different axes. First, we measure the *path similarity* of the two executions through execution indexing [130]. Indexing builds the hierarchical structure of an execution at the level of each executed instruction. Executions can be very precisely compared by comparing their structures. It has been used in a number of applications that demand comparing executions [70, 115]. For example, in Figure 6.7a, the instructions 2, 3, and 5 correspond across the two executions as denoted by the filled circles, but instruction 4 in the correct execution does not correspond to anything in the failing execution, as it only appears in one of the executions.

Note that execution indexing first requires that we know which static instructions correspond between the two executions. Because the executions come from two different versions of a program, some functions may have changed, with instructions being added or removed between the two versions. To determine which instructions correspond, we extracted the static control dependence regions into a tree and minimized a recursive tree edit distance [16] to find the largest possible correspondence between the two programs. For example, Figure 6.7b shows a program where a developer inserted an `if` statement guarding a new instruction. An edit distance algorithm will show that the fewest possible changes between the two functions require that lines 2 and 5 correspond across program versions, and the `if` statement on line 3 along with the nested control dependence region on line 4 are new additions to the correct version.

```
1 def foo():
2   print "one"
3   if guard:
4     print"two"
5   print "three"
```

Executions	
Failing	Passing
2●	●2
3●	●3
5●	○4
	●5

(a) Matching dynamic instructions across executions

```
1 def baz():                1 def baz():
2 ● print "one"             2 ● print "one"
3                           3 ○ if guard:
4                           4 ○
                              print"two"
5 ● print "three"           5 ● print "three"
failing program             correct program
```

(b) Matching static instructions across versions

Figure 6.7.: Matching of instructions across executions and program versions. Matches are denoted by ●, while mismatches are denoted by ○.

We score path similarity using the formula

$$100 \times \frac{2I_{both}}{I_f + I_c} \tag{6.1}$$

where I_{both} is the number of dynamic instructions that match across the executions, I_f is the total number of instructions in the failing execution, and I_c is the number of instructions in the correct execution. This scores the path similarity of two executions from 0 to 100, where 0 means that the paths are entirely different, and 100 means that the paths execute exactly the same instructions.

Second, we measure the *data similarity* of the executions. We compare the memory read and write instructions that correspond across the two executions, examining both the target location of the memory access and the value that it reads or writes. If these are equivalent across the executions, then the accesses match, otherwise they represent a data difference between the programs. The similarity of the accesses is then scored, again using formula 6.1, but only considering the instructions that access memory. One difficulty performing this comparison, as noted by others [99], is that comparing pointers across two executions is difficult because equivalent allocation instructions may allocate memory at different positions in the heap. This makes it difficult to determine both (1) when the targets of memory accesses match, and (2) when values of read or written pointers match. To enable this, we use *memory indexing* [115], a technique for identifying corresponding locations across different executions, to provide canonical IDs for memory locations. We also use shadow memory and compiler level instrumentation to mark which words in memory hold pointer values. Thus, when comparing the targets of accesses, we compare their memory indices. When comparing the values of accesses, we use the memory index of that value if the value is a pointer; otherwise we use the actual value in memory.

These two approaches allow us to objectively determine both how similar the paths taken by the peers and correct executions are as well as how similar the values they use and create are along those matching paths.

6.3.2 Results

Using these representatives of the peer selection techniques, we considered the collected bugs. Table 6.2 summarizes some details for the bugs. Column `BugID` uniquely identifies each bug across our results. `Program` lists the program to which each bug belongs, along with the date that a developer committed and documented a fix for the bug in the source repository in column `Patch Date`. The table classifies the fault in the program that induces a failure in `Fault Type`. For instance, in bug 15 on `make`, the program calls `strcpy()` instead of `memmove()`, resulting in a corrupted value, so the fault has the description ‘wrong function called’. Finally, column `Failure Type` classifies what the user of a program is able to directly observe about a failure. These observations primarily fall into two categories: the program either computed an unexpected value (denoted value), or it performed an unexpected action (behavior). For example, the incorrect value in bug 5 is an integer representing Unix permissions that the program computes incorrectly, while the

Table 6.2: Bugs used in the study along with their failures and the faults that cause them.

Bug ID	Program	Patch Date	Fault Type	Failure Type
1	tar	23 Jun 2009	missing guard	behavior
2	tar	30 Jul 2009	missing function call	value
3	tar	30 Jul 2009	weak guard	behavior
4	tar	5 Aug 2009	missing function call	behavior
5	tar	4 Oct 2009	wrong formula	value
6	tar	7 Oct 2009	design error	behavior
7	tar	17 Mar 2010	design error	behavior
8	tar	27 Mar 2010	incorrect guard	loop
9	tar	28 Jun 2010	call at wrong place	behavior
10	tar	23 Aug 2010	incorrect guards	behavior
11	make	3 Jul 2010	design error	behavior
12	make	6 Oct 2009	design error	behavior
13	make	30 Sep 2009	design error	behavior
14	make	23 Sep 2009	design error	behavior
15	make	1 Aug 2009	wrong function called	value
16	grep	14 Dec 2009	design error	behavior
17	grep	4 Mar 2010	corner case	behavior
18	grep	4 Mar 2010	corner case	behavior
19	grep	14 Mar 2010	corner case	behavior
20	grep	25 Mar 2010	incorrect goto	value

incorrect behavior of bug 14 is that `make` does not print out errors that it should. Bug 8 caused `tar` to execute an infinite loop, which is a specific type of incorrect behavior.

Overhead

First, consider the overhead of the techniques, presented in Table 6.3. For each bug in column `BugID`, we ran the five peer selection techniques, noting the `Time` in seconds required to select a peer and the number of `Tests` or reexeutions of the program that the technique used. Symbolic execution based approaches do not necessarily terminate on their own; they explore as many of the infinite program paths as possible within a time bound. Thus, this metric does not apply to symbolic execution; we always stopped it at 4 hours. Similarly, we stopped any other approach at 4 hours and considered that attempt to find a peer unsuccessful. Two approaches, the spectrum based approach and value replacement, both first aggregate data about an existing test suite. This effort amortizes over the develop-

Table 6.3: Peer selection work for each bug. For each technique, `Tests` holds the number of reexecutions of the program required by a technique before finding a peer. `Time` is the total time required to find a peer in seconds. *Average and StdDev for Predicate Switching without the infinite loop outlier are 119 and 101.

Bug ID	**Input Isolation**		**Spectrum**		**Symbolic Execution**	**Predicate Switching**		**Value Replacement**		
	Tests	Time	Profile Time	Selection Time	Time	Tests	Time	Tests	Profile Time	Selection Time
1	51	0.7	144	1233	N/A	22412	178	637381	148	5883
2	44	0.2	144	1236	N/A	-	65	-	143	2934
3	213	23.2	145	1257	N/A	25023	204	-	143	>4 hours
4	9	0.2	147	1299	N/A	26880	302	-	143	7032
5	14	10.1	226	1516	N/A	-	74	-	212	3279
6	40	0.6	228	1517	N/A	-	239	-	210	8592
7	208	3.1	262	1890	N/A	27252	228	836191	240	8054
8	27	50.4	251	1877	N/A	24132	12122	-	251	>4 hours
9	55	50.2	255	1866	N/A	31878	143	-	247	2691
10	6	0.1	315	2695	N/A	-	230	-	306	3381
11	5	0.1	35	841	N/A	12581	33	-	1112	>4 hours
12	6	0.1	36	811	N/A	21228	66	819332	1021	13753
13	6	0.1	36	758	N/A	16504	48	674999	1092	11167
14	7	0.1	38	757	N/A	-	230	-	1143	>4 hours
15	9	0.2	35	761	N/A	15599	50	-	648	>4 hours
16	20	0.1	1.1	248	N/A	-	7.1	-	18.7	65
17	5	0.1	4.5	188	N/A	26	0.1	4232	35.2	16.4
18	7	0.1	4.5	197	N/A	252	3.4	8446	25	28.1
19	5	0.1	2.6	218	N/A	196	3.4	232	31.8	6
20	13	40	3.4	627	N/A	39	2.1	59	41.8	1.2
Average	38	9	116	1090	N/A	16000	711*	432202	361	3386
StdDev	61	17	108	676	N/A	11545	2688*	675610	401	4244

ment of a program and is not considered part of the time required to find a single peer. It is an additional, auxiliary cost.

Input isolation is consistently the fastest approach, always finishing in under a minute, which is highly desirable. Next, we note that the average time for predicate switching includes an outlier. Bug 8 is an infinite loop in `tar`, and we use a five second timeout to detect such failures in our system. Thus, every test costs one to two orders of magnitude more in that test case. Discarding that outlier, predicate switching is the next fastest with an average of 2 minutes to find a peer or determine that it cannot. The next fastest is the spectrum based approach, using over 18 minutes to find a peer on average. Finally, value replacement is the slowest, taking about an hour on average to find a peer. From the table and the high standard deviation, however, we can see that it frequently takes 3-4 hours to find a peer in practice.

These results are more intuitive when considering what bounds each approach. Input isolation uses delta debugging over the execution input, so it has a running time polynomial in the size of the input in the worst case and logarithmic in normal cases. The spectrum based approach computes an edit distance over metadata for each statement in a program, so it is polynomially bounded by program size. In contrast, predicate switching and value replacement are bounded by the length of the execution in either branch instructions or all instructions respectively, so they can take longer in theory. Looking at the `Tests` columns, both approaches execute the buggy program several thousand times when finding a peer, whereas input isolation executes it only 38 times on average. In practice, if executions take several seconds to run, then predicate switching and value replacement will become more costly to use, just as we observe for bug 8.

Fitness

While the speed of peer selection is important for providing convenience and scalability, Figure 6.1 shows that a peer is not useful for debugging unless it forms a good approximation of the correct execution. To that end, we collected the peers generated by each

Table 6.4: Peer selection similarity scores for each bug. For each selection technique, `Path` denotes the path similarity score for each bug. `Data` denotes the data similarity score for each bug.

Bug ID	**Input Isolation**		**Spectrum**		**Symbolic Execution**		**Predicate Switching**		**Value Replacement**	
	Path	Data	Path	Data	Path	Data	Path	Data	Path	Data
1	0.2	67.8	98.6	59.3	0.2	69	99.6	67.5	99.7	65.4
2	99.0	56.1	84.0	56.2	0.2	67	-	-	-	-
3	0.2	69.7	87.6	59.4	0.2	68	87.6	56.3	-	-
4	97.4	69.6	95.9	59.0	0.2	69	97.7	66.7	-	-
5	99.3	60.0	98.7	59.7	0.5	69	-	-	-	-
6	0.1	9.1	0.1	9.1	0.1	9.1	-	-	-	-
7	0.1	37.5	0.1	37.5	0.1	9.1	0.1	37.5	0.1	37.5
8	0.2	66.8	95.9	54.4	0.2	67	99.5	67.2	-	-
9	0.2	67.2	75.4	58.9	0.2	67	74.8	58.3	-	-
10	0.1	21.9	0.1	21.9	0.1	22	-	-	-	-
11	74.7	56.2	66.2	52.4	93	50	97.8	61.9	-	-
12	67.1	52.4	0.1	13.9	15	55	66.9	59.2	66.8	59.2
13	78.4	63.3	43.2	63.3	75	65	38.6	61.2	78	61
14	0.1	13.8	0.1	13.9	0.1	14	-	-	-	-
15	52.3	61.4	66.2	52.4	82	76	97.8	61.7	-	-
16	0.1	0	0.1	0	0.1	0	-	-	-	-
17	13.6	46.5	0.7	70	0.6	66	1.1	81	74	63
18	14.4	49.6	93	67	0.6	66	55.2	63	71	63
19	75.8	64.5	93	64	0.6	67	97	64	3.1	69
20	30.4	60.9	0.2	70	0.2	67	2.6	59	31	82
Average	33	49	50	47	13	52	65	62	49	63
StdDev	44	21	42	20	30	25	33	8.7	51	15

technique. We traced the control flow and data accesses of the executions and compared these traces using the metric from Section 6.3.1. Table 6.4 shows the results. For each technique, we present the path similarity (`path`) and data similarity (`data`), along with the average and standard deviation. We further discuss our observations, each highlighted and tagged with the symbol O_i.

Predicate Switching First note that **(O_1):** *predicate switching performs best in terms of both path and data similarity*, on average scoring in the 60's with the lowest variance in both similarity metrics.

Since predicate switching often generates infeasible paths under the original faulty program, we observe that in many cases **(O_2):** *correct executions resemble infeasible paths*

```
1 int create_placeholder(char *file, int *made){
2   ...
3    while ((fd = open (file)) < 0)
4      if (! maybe_recoverable (file, made))
5        break;
6 }
```

tar, extract.c: create_placeholder

Figure 6.8.: Bug 8 requires unsound techniques to get a strong peer.

under the original faulty program. Consequently, input based techniques are less effective because they only generate feasible paths for the faulty program. For example, consider bug 8 of tar, presented in Figure 6.8. This function tries to open a file at a given path location. If the call to open fails, the execution enters the loop to attempt fixing the error with a call to maybe_recoverable. If this call returns the constant RECOVER_OK, then the error was fixed and the function calls open again. The failing execution enters an infinite loop because the condition on line 4 should be compared against the constant RECOVER_OK instead of being negated. In this buggy form, maybe_recoverable returns a nonzero error code every time, so the loop never terminates. Techniques relying on sound behavior cannot modify the execution to escape the loop, so they select executions that never enter it. Because predicate switching can select peers from unsound or infeasible executions, it is able to iterate through the loop the correct number of times and then break out of it. Thus, it matches the correct path very closely with a score of 99.5.

(O_3): *Predicate switching may fail to patch a failing run even though it often leads to peers with good quality if it manages to find one.* In comparison, since input based techniques are not constrained to patching the failing run. They will eventually select a closest peer (even with bad quality).

(O_4): *Predicate switching may coincidently patch a failing run, returning a peer with poor quality.* For example, in bug 17 for grep, predicate switching yields a peer with a path similarity of only 1.1, even though it produced the correct output. The path of the peer execution actually deviates from the correct path early in the execution, but the deviated path still produces the expected output. Figure 6.9 presents the location of the

```
1 void mb_icase_keys (char **keys, size_t *len){
2   ...
3 incorrectpatch:
4   for (i = j = 0; i < li ;) {
5     /* Convert each keys[i] to lowercase */
6   }
7 }
```

grep, grep.c: mb_icase_keys

Figure 6.9.: Execution synthesis on bug 17 yields an erroneous patch.

incorrectly patched predicate. When `grep` starts, it converts user requested search patterns into a multibyte format when ignoring character case, but predicate switching toggles the condition on line 4, preventing patterns from being converted. That `grep` produced correct output is merely coincidental. The unconverted search patterns happen to yield a successful match when the converted ones did not, even though the buggy portion of the program is in an unrelated component (see Figure 6.10). Thus, the comparison of the failing run with the peer does not isolate faulty state but rather some random differences. However, this is an unlikely enough coincidence that predicate switching still scores best on average.

Finally, **(O_5):** *predicate switching is less likely to provide a good peer when used on bugs that exhibit value based failures*. For three of the bugs that exhibit value based failures, where the fault leads to the incorrect value through computation instead of control flow, predicate switching yields a poor peer (bug 20) or no peer at all (bugs 2 and 5). Developers can use this insight to use a more general approach like the spectrum techniques when dealing with value failures.

Value Replacement Value replacement shares much in common with predicate switching. Hence, **(O_6):** *value replacement scores high when it finds a peer but is less likely to find a peer*. This shortcoming is twofold: (1) it takes so long that we had to cut its search short, and (2), it can only mutate values of an execution as guided by values observed in the test suite. If the desired value is not seen in the test suite, value replacement cannot generate a peer. Sometimes, this leads to peers with very poor quality (bug 19) or even no peers (bugs 9 and 10).

```
void parse_bracket_exp_mb (void) {
bug19:
  wt = get_character_class(str, case_fold);
  ...
bug17:
  if (case_fold)
    remap_pattern_range(pattern);
  ...
bug18:
  if (case_fold)
    canonicalize_case(buffer);
}
```

`grep`, dfa.c: parse_bracket_exp_mb

Figure 6.10.: Bugs 17, 18, and 19 are related.

Spectrum Techniques The next strongest technique is the spectrum based approach, which matched paths as well as value replacement, but was not able to match data accesses as well. **(O_7):** *The spectrum based approach works better with a larger test suite, which on the other hand requires more time to search for the peer.* This is evidenced by its strongest scores for `tar`, which had the largest test suite of the programs we tested and the high computation cost. In other cases, as in both `make` and `tar`, finding a desirable peer may be difficult or impossible because no similar test exists.

Of particular interest, however, are the similarity scores in the 90's for bugs 18 and 19 of `grep`. Both of these bugs are related to the component that was patched to fix bug 17. The peer selected for both of those bugs was actually the test added to make sure bug 17 did not reappear as a regression. As shown in Figure 6.10, the three bugs are within the same function, dealing with the same set of variables. As a result, the regression test closely resembles the correct executions regarding bugs 18 and 19, making itself a good peer. Thus, **(O_8):** *the spectrum based approach is able to adapt.* Once a test for a buggy component is added to the test suite, related bugs may have a stronger peer for comparison. This is particularly useful in light of the fact that bugs tend to appear clustered in related portions of code [77].

Input Isolation Input isolation is by far the fastest, but there does not appear to be an indicator for when or if it will return a useful peer. Its tendency to yield paths of low similarity to the correct executions makes it generally less desirable for peer selection. This happens because **(O_9):** *input similarity and behavioral similarity do not necessarily correspond, so making small changes to input will likely create large changes in the behavior of a program.* For example, consider bug 8 for `tar`. When extracting an archive with a symbolic link, using the `-k` option causes `tar` to enter an infinite loop. This option, however, significantly alters the behavior of the program, preventing it from extracting some files. Input isolation selects a peer by generating new input where the `k` has simply been removed. This avoids the error but also radically changes the behavior of the program such that the peer is highly dissimilar to the correct execution. This effect is undesirable because the differences between the failing run and the peer may not be relevant to the fault but rather just semantic differences resulting from the input difference.

In spite of this, its ability to exploit *coincidental similarity* between runs on passing and failing inputs allows input isolation to create better peers for some bugs than any of the other approaches. For example, consider bug 5 of `tar`. Input isolation produced the peer with the highest path and value similarity for this bug. The original command line to produce the failure included the arguments `-H oldgnu`, telling `tar` to create an archive in ‘oldgnu’ format. Input Isolation determined that the most similar passing run should instead use the arguments `-H gnu`. Coincidentally, ‘gnu’ is another valid file format, and creating an archive in that format follows almost the same path as the oldgnu format. Such behavior is difficult to predict but quite interesting.

Symbolic Execution We observe **(O_{10}):** *the symbolic execution approach performs poorly, generating only very short executions that diverge substantially from the failing runs.* The main reason is that the approach tries to model path conditions of entire failing runs in order to search for their neighbors. However, the original failing runs are too complex for KLEE to handle, causing the system to time out. According to the Figure 6.5, the technique degenerates to producing inputs for short executions, which it can manage. Note that good performance was reported for the same algorithm for web applications in [8], but web applications are usually smaller and have very short executions. We note that good code coverage can be achieved for our subject programs if simply using KLEE as a test generation engine. But we observe the generated executions are nonetheless very short, attributed to the path exploration strategy of the test generation algorithm. In contrast, peer selection requires modeling relatively much longer executions.

Threats to Validity The above experiments only consider three client programs; `tar`, `make`, and `grep`; and may not generalize to other programs. In particular, these programs are all medium sized GNU utilities. Experimentation on a broader variety of programs can lessen this threat. Additionally, we have only considered five representative approaches for peer selection, but other techniques that fall into the same classification exist and may yield different results [46, 123]. Our implementations are also derived only from published work and may differ slightly from the real designs of their published systems. However, the case analysis examining the problems with each type of peer selection stands even for new techniques and supports the empirical results.

Finally, we note that we only evaluate these techniques for peer selection, we do not dispute their utility in solving other problems.

6.4 Related Work

We have examined methods for selecting or synthesizing executions for use in execution comparison. Many techniques related to peer selection stem from existing areas of software engineering.

Execution comparison itself has a long history in software engineering, from earlier approaches that enabled manually debugging executions in parallel to more automated techniques that locate or explain faults in programs [28, 125, 132]. These systems require that a failing execution is compared with an execution of the same program. This chapter augments these approaches by suggesting that the desired execution should approximate the correct one. This approximation aids the existing techniques in explaining failures.

Many peer selection techniques come from automated debugging analyses. Delta debugging, used for input isolation, is well known for reducing the sizes of failing test cases and several other uses [133]. Zeller's work on execution comparison also utilized delta debugging to locate faulty program state [28], using either input isolation or spectrum based selection techniques to select a peer execution. Spectrum based approaches for fault localization [36, 56, 66, 100, 101] have long exploited existing test suites in order to identify program statements that are likely buggy. Symbolic execution and constraint solving are popular approaches for automatically generating tests [22, 48, 97, 105, 129]. By building formulae representing the input along paths in an execution, they are able to solve for the input required to take alternative paths. Some systems also use search heuristics to guide the tests toward a particular path for observation [8, 97, 129], appropriately generating a peer as we did in our experiments. Execution synthesis techniques like predicate switching and value replacement mutate an existing failing execution in unsound ways to see if small changes can make the executions behave correctly [63, 134]. Information about how and where this dynamic patch occurred can be used for fault localization. Some effort has been put into finding corrections to executions that require multiple predicate switches [123]. More recently, the observation that unsound approximations of executions provide useful information about real program behaviors has even been explored for test generation [120].

Our system for comparing execution traces uses execution and memory indexing to identify corresponding instructions and variables across the different executions [115, 130]. Other trace comparisons exist, but they do not allow strictly aligning both data and control flow as necessary for examining suitability for execution comparison [57, 99, 137]. Existing work also looks at the static alignment and differencing of static program, but these approaches emphasize efficiency over precision or high level alignment over low level alignment in comparison to the edit distance approach used in this paper [7, 58]. Edit distances have been used to align the abstract syntax trees of multiple programs [45], but not the control flow, which is necessary for aligning traces.

6.5 Conclusions

This chapter introduces peer selection as a subproblem of using execution comparison for debugging. Given a failing execution, that failing execution must be compared against a peer that approximates the correct, intended execution in order to yield useful results. We have surveyed and implemented five existing peer selection techniques, comparing how well the peers that they generate approximate the correct execution and how quickly they can find these peers. The results provide insight into the advantages and disadvantages of the different techniques.

7 RELATED WORK

Coping with debugging has been a long standing issue for software developers. Beyond the core categories of fault localization and slicing, there have been further commonalities and trends in debugging research that relate to the techniques presented in this dissertation.

7.1 Comparison Based Debugging

Our contrasts a correct execution with a buggy execution and explains why they differ. Previous approaches have also used comparison based approaches that look at how buggy and correct executions differ.

Notably, we have already seen that fault localization techniques look at the criteria such as whether or not a statement was more likely to be executed in buggy runs of a program than in correct runs [2, 66]. Renieris and Reiss have also used criteria such as how frequently each instruction was executed to identify correct executions that are similar to a failing execution. They then used the same spectra to identify differences between the two executions that suggested where the buggy statement may reside [100]. Ball et al. used a pool of correct executions combined with counterexamples from a model checker to identify the first step within the failing counterexample that did not exist within any correct trace as well. The transitions into failure then identify possible causes of errors [13].

Some techniques focus instead on comparing different versions of a program or different implementations of solutions to localize possible bugs. Relative debugging from Abramson introduced a debugging interface like gdb augmented with primitives for examining the values of variables in two executions at once. Using these primitives, a developer could manually execute both programs up to the same point in each implementation and use the comparison primitives to determine whether or not variables had the same values in both executions [1]. In contrast, constraint based techniques like Darwin and golden implementation driven debugging look at the common constraints within executions of different programs or different versions of a program on the same input. These techniques then analyze the path conditions of the different implementations to identify conjuncts that are likely associated with the failure [14, 97].

The automated comparison based techniques presented thus far focus on localizing bugs rather than explaining how bugs propagate through an execution. In contrast, we previously showed how Zeller's original work as well as dual slicing use comparison based techniques to explain failures, albeit with less precision or correctness than the technique presented in this dissertation [28, 125, 132].

7.2 Interactive Debugging Interfaces

Not all research has focused on purely manual or purely automated debugging techniques. Instead, some research looks at debugging assistants that try to guide developers by either asking them questions or presenting possible questions that may be answered through further analyses.

Algorithmic debugging is one of the earliest such debugging assistants of which we are aware. It would identify possible conditions related to the search for a bug and ask the developer whether or not those conditions were true. Using the developer feedback, algorithmic debugging could direct the search for a buggy procedure within a program [108]. In contrast, more recent work by Dillig et al. provides an interactive assistant that does not try to explain why or where a bug may be but rather to classify bug reports as actual bugs or false alarms [39].

Whyline was a debugging interface designed to guide the developer through the use of slices for understanding why a bug occurred [74]. Ko et al. recognized that developers often guide their understanding of program behavior through an interrogative process, so Whyline provided an interface of possible questions that the developer might ask at different points within an execution. These questions then fed into static and dynamic slicing algorithms that could guide the developer further backward through the program.

Of these techniques, Whyline is the most closely related to our approach, even though we do not focus on the user interface aspect of debugging. Whyline guides the developer by posing possible questions that can guide the developer backward through the slice. In contrast, the technique presented in this paper determines which questions are useful to ask through the dependences in the final explanation of a bug. Thus, the techniques are complementary and pose a possible direction for future work.

7.3 Structuring Slice Information

Instead of guiding the developer through the slice via questions, some debugging techniques abstract away or deprioritize portions of slices that may not be as immediately interesting to the developer. Developers can still explore the entire slices, but the goal of these techniques is to direct the developer toward interesting portions of the slice first.

One such initial tool was the Program Slice Browser [35]. This browser provided an interface for traversing static program slices, but it also provided a coarse grained hierarchical abstraction of slices. Thus, for instance, the dependences between functions would first appear in the slice, and the developer could zoom into the dependences within a function on demand.

This tool is related to hierarchical dynamic slicing, which instead operates on dynamic slices [124]. However, hierarchical dynamic slicing offers abstractions at the level of control structures such as loops as well. These are important to address in the context of dynamic slicing because long running loops can introduce additional dependences into a slice during each loop iteration.

Similar to dual slicing, thin slicing used the observation that not all dependences and definitions within a slice are actually relevant to answering questions using a slice [111]. It uses the observation that dependences may be introduced into a slice that do not actually affect a value of interest. For instance, a buggy value may be inserted into a container type and read from a container type, thus introducing dependences on the container type itself, even though only the buggy value is of interest. Thin slicing hierarchically prunes the likely uninteresting values from the slice and instead allows the developer to explore them on demand.

Zhang et al. used profiles of the values at different instructions within a program to establish a ranked confidence that the value produced at each instruction within a buggy execution was correct [135]. These confidence rankings then guide a developer's search by deprioritizing or pruning away the statements from the slice that are likely correct.

All of these tools provide interfaces and abstractions that help developers to navigate and understand slices. As such, they are orthogonal to the work presented in this paper and future work may combine them to examine the impact on resulting slices.

7.4 Explanations from Distance Metrics

One of the most related lines of work to that presented in this dissertation is Groce's work on explaining failures using distance metrics [50]. In this work, Groce created a bounded static model of a buggy program using CBMC and searched for a correct execution in this model that had the most similar control flow and data flow to the buggy execution. The approach then computed a backward dynamic slice over the differences between the two executions to produce an explanation that was similar although not equivalent to a dual slice.

The search for an execution as similar as possible to the failing execution is inspired by the counterfactual model of causality [87]. However, as we observed in Chapter 6, the differences between an execution similar to the failing one can simply lead to an explanation of the semantic differences in their input. To explain a failure, correct the execution needs to reflect the intended behavior of the failing one.

The slicing technique presented in the distance metrics work is also able to prune away statements that produce the same values in both executions, as in dual slicing. However, it does not always allow dependences that only exist in one execution to feed back into the slice of the other. Thus, for bugs whose explanations include execution omission, the resulting slice may not be able to identify the missing behaviors that should have occurred.

8 CONCLUSIONS

8.1 Contributions

This dissertation makes contributions in the areas of execution comparison and automated debugging. It first defines problems and solutions that enable dynamic analyses to efficiently and precisely compare properties of multiple executions. Using these, it further constructs techniques that improve the efficiency and precision of generating explanations for software failures.

Canonical Identities for Execution Points. This dissertation surveys and contrasts the already existing approaches for identifying execution points, thus identifying when some approaches may be preferable over others. In addition, this comparison shows that existing approaches do not provide a solution for comprehensively, scalably, and efficiently identifying execution points across multiple executions, which fine-grained execution comparison requires. Instead, a new approach derived from identifying statically equivalent points in control flow graphs does provide such a platform for identifying execution points. The results in this dissertation show that the new technique, PEPID, can identify execution points as a program runs with about 25% overhead in execution time on average.

Canonical Identities for Memory Locations. Just as execution comparison benefits from identifying equivalent execution points across executions, it also benefits from identifying equivalent locations in memory. This allows the comparison of equivalent memory locations to determine precisely what variables and values differ across two executions. Exploiting the EPIDs developed within Chapter 3, this dissertation presents a technique for identifying all memory by using the EPID at which a program allocates a region of memory as well as the offset within an allocated memory region. These new identifiers for memory locations facilitate the identification of corresponding memory across executions. The identifiers also enable the meaningful replacement of the values in memory within one execution with values in corresponding memory from another execution, even when some of those values are pointers to memory within the second execution.

Automated Explanations of Software Failures. Dynamic slices are large in part because they traverse and include dependences that are not relevant to explaining a bug. Even so, dynamic slices also omit dependences necessary to explain a bug because the dependences involve instructions not executed in the buggy runs. These unnecessary dependences can be removed and the missing ones added back by identifying those variables and values within the slice that *cause* the bug at an execution point. Existing techniques, however, have problems with both efficiency and correctness. They can take a long time to identifies causes even at a single step of an execution, and the identified causes can both include and exclude information they should not.

This dissertation shows that execution comparison can indeed be used to identify the causes of a bug. By comparing the failing execution of a program with another execution that approximates the intended correct behavior of the failing one, execution comparison can identify those variables and values reflecting the missed behavior and dependences in traditional dynamic slices. Through state replacement and specialized reexecution of both the failing and correct runs, execution comparison can identify a minimal set of causes for a bug at a particular execution point. As a result, the technique for computing explanations of software failures presented within this dissertation is practical for explaining real world bugs. The technique computes explanations in under three minutes on average and explains how the root causes of bugs propagate to failures in 73% of the examined bugs. Previous techniques worked for only 27% of these bugs.

Approximating Correct Executions. The quality of an explanation for a single failing execution also depends on how well the second (correct) execution reflects the intended behavior of the failing one. This is in contrast to statistical techniques for fault localization. Statistical techniques need to compare a pool of multiple failing executions against correct ones that are similar to the failing ones in order to avoid statistical confounding [11, 49]. This dissertation examines existing approaches for finding such executions that are similar to the intended behavior. We find that some approaches, in particular predicate switching, are both more effective and more efficient at approximating the intended behavior than others, but it is also possible for those techniques to fail at providing any approximation at all.

In addition, we observe that these are only *approximations* of the intended behavior. The approximations may still be dissimilar from the intended behavior, and in such cases, explaining why the two executions differ still will not explain the bug itself, as seen in 27% of the cases in Chapter 5.

8.2 Future Work

Identifying Causes Without Reexecution. One of the core limitations of the technique presented in this paper is that it requires repeated reexecution of the failing and correct executions with deterministic results. Real world programs, however, do not necessarily exhibit deterministic behavior. Programs that exploit concurrency or asynchronous behaviors are affected nondeterministically by timing. Programs that interact with the environment provided by the operating system may be affected by nondeterministic environmental state. Explaining the bugs within these programs cannot rely on repeated deterministic reexecution. Finding a way to explain execution differences without reexecution can avoid this limitation and potential also reduce the overhead associate with reexecution itself.

Improving Approximations of Correct Behavior. The quality of the produced explanations depends on how well the correct execution mimics the intended behavior. While it producing such approximations may be possible for programs like system utilities, the quality of such approximations may be poor, and the techniques may not work for other types of applications like server, mobile, or web based applications. Furthermore, producing the approximations can take longer on average than producing the explanations themselves. Exploring faster ways of producing these approximations and ways of more closely mimic-ing the intended behavior of a failing execution both have the potential to improve the practicality and utility of the framework presented within this dissertation.

Abstractions for Presenting Explanations. The explanations computed within by the presented framework work at the instruction and basic block level of the analyzed programs, but this may not be desirable. For languages like Python or ECMAScript, a single statement or operation within the source code of a program may yield multiple implicit instructions and even multiple function calls. A developer may not expect an explanation in terms of these implicit instructions but rather in terms of the high level operations used within source code. In addition, some semantic operations may produce multiple instructions that do not match the level at which a developer understands a program. For instance, a program may sum of all of the elements within a list. If an explanation of a bug involves that summation, the explanation may capture the addition of every element of the list even if only one is relevant. Finding better ways to capture and rarify the intended meaning of a program could abstract away the irrelevant or low level details that do not concern the developer and make automated explanations more useful.

Other Uses for Explaining Execution Differences. Comparing executions has proven useful in security analysis [65], testing and impact analysis [99], and program comprehension [30]. The efficiency and precision of the techniques presented within this dissertation for explaining execution differences may improve the existing solutions within these areas or perhaps allow for entirely new solutions.

LIST OF REFERENCES

LIST OF REFERENCES

[1] David Abramson, Ian Foster, John Michalakes, and Rok Sosič. Relative debugging: A new methodology for debugging scientific applications. *Communications of the ACM*, 39(11):69–77, November 1996.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Hiralal Agrawal. On slicing programs with jump statements. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 302–312, New York, NY, USA, 1994. ACM.

[4] Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience*, 23(6):589–616, June 1993.

[5] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul London. Incremental regression testing. In *Proceedings of the 1993 Conference on Software Maintenance*, pages 348–357, Washington, DC, USA, 1993. IEEE Computer Society.

[6] Dong H. Ahn, Bronis R. de Supinski, Ignacio Laguna, Gregory L. Lee, Ben Liblit, Barton P. Miller, and Martin Schulz. Scalable temporal order analysis for large scale debugging. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 44:1–44:11, New York, NY, USA, 2009. ACM.

[7] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1):3–36, March 2007.

[8] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 49–60, New York, NY, USA, 2010. ACM.

[9] Shay Artzi, Sunghun Kim, and Michael D. Ernst. ReCrash: Making software failures reproducible by preserving object states. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 542–565, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] George K. Baah, Andy Podgurski, and Mary Jean Harrold. The probabilistic program dependence graph and its application to fault diagnosis. *IEEE Transactions on Software Engineering*, 36(4):528–545, 2010.

[11] George K. Baah, Andy Podgurski, and Mary Jean Harrold. Mitigating the confounding effects of program dependences for effective fault localization. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 146–156, New York, NY, USA, 2011. ACM.

[12] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[13] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 97–105, New York, NY, USA, 2003. ACM.

[14] Ansuman Banerjee, Abhik Roychoudhury, Johannes A. Harlie, and Zhenkai Liang. Golden implementation driven software debugging. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 177–186, New York, NY, USA, 2010. ACM.

[15] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[16] Philip Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1-3):217–239, June 2005.

[17] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 197–206, New York, NY, USA, 1993. ACM.

[18] Michael D. Bond, Graham Z. Baker, and Samuel Z. Guyer. Breadcrumbs: Efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 13–24, New York, NY, USA, 2010. ACM.

[19] Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, pages 97–112, New York, NY, USA, 2007. ACM.

[20] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, September 1973.

[21] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 480–490, Washington, DC, USA, 2004. IEEE Computer Society.

[22] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[23] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10:1–10:38, December 2008.

[24] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 121–130, New York, NY, USA, 2011. ACM.

[25] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44, Washington, DC, USA, 2009. IEEE Computer Society.

[26] Jong-Deok Choi and Sang Lyul Min. Race frontier: Reproducing data races in parallel-program debugging. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 145–154, New York, NY, USA, 1991. ACM.

[27] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[28] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.

[29] James S. Collofello and Larry Cousins. Towards automatic software fault location through decision-to-decision path analysis. *International Workshop on Managing Requirements Knowledge*, 0:539, 1987.

[30] Bas Cornelissen and Leon Moonen. Visualizing similarities in execution traces. In *Proceedings of the Third Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pages 6–10, 2007.

[31] Michael A. Cusumano. Reflections on the toyota debacle. *Communications of the ACM*, 54(1):33–35, January 2011.

[32] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. Debugger canvas: Industrial experience with the code bubbles paradigm. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1064–1073, Piscataway, NJ, USA, 2012. IEEE Press.

[33] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 121–134, New York, NY, USA, 1996. ACM.

[34] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Failure and fault analysis for software debugging. In *Proceedings of the 21st International Computer Software and Applications Conference*, COMPSAC '97, pages 515–521, Washington, DC, USA, 1997. IEEE Computer Society.

[35] Yunbo Deng, Suraj Kothari, and Yogy Namara. Program slice browser. In *Proceedings of the 9th International Workshop on Program Comprehension*, IWPC '01, pages 50–, Washington, DC, USA, 2001. IEEE Computer Society.

[36] William Dickinson, David Leon, and Andy Podgurski. Pursuing failure: The distribution of program failures in a profile space. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 246–255, New York, NY, USA, 2001. ACM.

[37] Madeline Diep, Sebastian Elbaum, and Matthew Dwyer. Trace normalization. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, ISSRE '08, pages 67–76, Washington, DC, USA, 2008. IEEE Computer Society.

[38] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[39] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 181–192, New York, NY, USA, 2012. ACM.

[40] Igor Douven. Abduction. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2011 edition, 2011.

[41] Evren Ermis, Martin Schäf, and Thomas Wies. Error invariants. In *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 187–201. Springer Berlin Heidelberg, 2012.

[42] Min Feng and Rajiv Gupta. Detecting virus mutations via dynamic matching. In *Proceedings of the 2009 IEEE International Conference on Software Maintenance*, ICSM '09, pages 105–114, 2009.

[43] Min Feng and Rajiv Gupta. Learning universal probabilistic models for fault localization. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 81–88, New York, NY, USA, 2010. ACM.

[44] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987.

[45] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, November 2007.

[46] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 947–954, New York, NY, USA, 2009. ACM.

[47] Margaret Ann Francel and Spencer Rugaber. The relationship of slicing and debugging to program understanding. In *Proceedings of the 7th International Workshop on Program Comprehension*, IWPC '99, pages 106–, Washington, DC, USA, 1999. IEEE Computer Society.

[48] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[49] Ross Gore and Paul F. Reynolds, Jr. Reducing confounding bias in predicate-level statistical debugging metrics. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 463–473, Piscataway, NJ, USA, 2012. IEEE Press.

[50] Alex Groce. *Error explanation and fault localization with distance metrics*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005.

[51] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8(3):229–247, June 2006.

[52] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. Synergy: A new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 117–127, New York, NY, USA, 2006. ACM.

[53] Liang Guo, Abhik Roychoudhury, and Tao Wang. Accurately choosing execution runs for software fault localization. In *Proceedings of the 15th International Conference on Compiler Construction*, CC'06, pages 80–95, Berlin, Heidelberg, 2006. Springer-Verlag.

[54] Tibor Gyimóthy, Árpád Beszédes, and Istán Forgács. An efficient relevant slicing method for debugging. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, pages 303–321, London, UK, UK, 1999. Springer-Verlag.

[55] Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S. Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 129–142, Washington, DC, USA, 2008. IEEE Computer Society.

[56] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '98, pages 83–90, New York, NY, USA, 1998. ACM.

[57] Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. Semantics-aware trace analysis. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 453–464, New York, NY, USA, 2009. ACM.

[58] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 234–245, New York, NY, USA, 1990. ACM.

[59] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[60] Hwa-You Hsu, James A. Jones, and Alex Orso. Rapid: Identifying bug signatures to support debugging activities. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 439–442, Washington, DC, USA, 2008. IEEE Computer Society.

[61] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, May 1977.

[62] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. Fault localization using value replacement. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 167–178, New York, NY, USA, 2008. ACM.

[63] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. Effective and efficient localization of multiple faults using value replacement. In *Proceedings of the 2009 IEEE International Conference on Software Maintenance*, ICSM '09, pages 221–230, 2009.

[64] Lingxiao Jiang and Zhendong Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 184–193, New York, NY, USA, 2007. ACM.

[65] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 347–362, Washington, DC, USA, 2011. IEEE Computer Society.

[66] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[67] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[68] Manu Jose and Rupak Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 437–446, New York, NY, USA, 2011. ACM.

[69] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 327–336, New York, NY, USA, 2010. ACM.

[70] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 110–120, New York, NY, USA, 2009. ACM.

[71] Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. Expositor: Scriptable time-travel debugging with first-class traces. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 352–361, Piscataway, NJ, USA, 2013. IEEE Press.

[72] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[73] Juergen Klein. Francis Bacon. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2011 edition, 2011.

[74] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[75] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, December 2006.

[76] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[77] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 83–93, New York, NY, USA, 2004. ACM.

[78] Arun Lakhotia. Graph theoretic foundations of program slicing and integration. Technical Report CACS TR-91-5-5, University of Southwestern Louisiana, 1993.

[79] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[80] Feng Li, Wei Huo, Congming Chen, Lujie Zhong, Xiaobing Feng, and Zhiyuan Li. Effective fault localization based on minimum debugging frontier set. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '13, pages 1–10, Washington, DC, USA, 2013. IEEE Computer Society.

[81] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 141–154, New York, NY, USA, 2003. ACM.

[82] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.

[83] Barbara Liskov. The power of abstraction. In *ACM Turing Award Lectures*. ACM, New York, NY, USA, 2007.

[84] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. SOBER: Statistical model-based bug localization. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 286–295, New York, NY, USA, 2005. ACM.

[85] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[86] John M. Mellor-Crummey and Thomas J. LeBlanc. A software instruction counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, pages 78–86, New York, NY, USA, 1989. ACM.

[87] Peter Menzies. Counterfactual theories of causation. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2009 edition, 2009.

[88] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. The design of bug fixes. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 332–341, Piscataway, NJ, USA, 2013. IEEE Press.

[89] Todd Mytkowicz, Devin Coughlin, and Amer Diwan. Inferred call path profiling. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 175–190, New York, NY, USA, 2009. ACM.

[90] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[91] Peter G. Neumann. Risks to the public. *ACM SIGSOFT Software Engineering Notes*, 32(3):20–24, May 2007.

[92] Tuan Anh Nguyen, Christoph Csallner, and Nikolai Tillmann. GROPG: A graphical on-phone debugger. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1189–1192, Piscataway, NJ, USA, 2013. IEEE Press.

[93] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 177–184, New York, NY, USA, 1984. ACM.

[94] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 25–36, New York, NY, USA, 2009. ACM.

[95] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[96] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[97] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: An approach to debugging evolving programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(3):19:1–19:29, July 2012.

[98] Ganesan Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(5):455–490, September 2002.

[99] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 241–252, Washington, DC, USA, 2006. IEEE Computer Society.

[100] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ASE '03. IEEE Computer Society, 2003.

[101] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC '97/FSE-5, pages 432–449, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[102] Michiel Ronsse, Koen De Bosschere, Mark Christiaens, Jacques Chassin de Kergommeaux, and Dieter Kranzlmüller. Record/replay for nondeterministic program executions. *Communications of the ACM*, 46(9):62–67, September 2003.

[103] Jeremias Rößler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. Isolating failure causes through test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 309–319, New York, NY, USA, 2012. ACM.

[104] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 139–152, New York, NY, USA, 2013. ACM.

[105] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[106] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[107] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[108] Ehud Y. Shapiro. Algorithmic program diagnosis. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 299–308, New York, NY, USA, 1982. ACM.

[109] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[110] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 432–441, New York, NY, USA, 1999. ACM.

[111] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 112–122, New York, NY, USA, 2007. ACM.

[112] Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft Research, Redmond, Wash, 1993.

[113] William N. Sumner, Tao Bao, and Xiangyu Zhang. Selecting peers for execution comparison. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 309–319, New York, NY, USA, 2011. ACM. doi:10.1145/2001420.2001458.

[114] William N. Sumner and Xiangyu Zhang. Algorithms for automatically computing the causal paths of failures. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, FASE '09, pages 355–369, Berlin, Heidelberg, 2009. Springer-Verlag.

[115] William N. Sumner and Xiangyu Zhang. Memory indexing: Canonicalizing addresses across executions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 217–226, New York, NY, USA, 2010. ACM. doi:10.1145/1882291.1882324.

[116] William N. Sumner and Xiangyu Zhang. Comparative causality: Explaining the differences between executions. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 272–281, Piscataway, NJ, USA, 2013. IEEE Press.

[117] William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. Precise calling context encoding. *IEEE Transactions on Software Engineering*, 38(5):1160–1177, September 2012.

[118] Gregory Tassey. Economic impacts of inadequate infrastructure for software testing. Technical Report RTI Project Number 7007.011, National Institute of Standards and Technology, 2002.

[119] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[120] Petar Tsankov, Wei Jin, Alessandro Orso, and Saurabh Sinha. Execution hijacking: Improving dynamic analysis by flying off course. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 200–209, Washington, DC, USA, 2011. IEEE Computer Society.

[121] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. DIABLO: A reliable, retargetable and extensible link-time rewriting framework. In *International Symposium on Signal Processing and Information Technology*, pages 7–12. IEEE, 2005.

[122] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 15–26, New York, NY, USA, 2011. ACM.

[123] Tao Wang and Abhik Roychoudhury. Automated path generation for software fault localization. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 347–351, New York, NY, USA, 2005. ACM.

[124] Tao Wang and Abhik Roychoudhury. Hierarchical dynamic slicing. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 228–238, New York, NY, USA, 2007. ACM.

[125] Dasarath Weeratunge, Xiangyu Zhang, William N. Sumner, and Suresh Jagannathan. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 253–264, New York, NY, USA, 2010. ACM.

[126] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[127] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[128] W Eric Wong and Vidroha Debroy. A survey of software fault localization. Technical Report UTDCS-45-09, University of Texas at Dallas, 2009.

[129] Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, pages 359–368, June-July 2009.

[130] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 238–248, New York, NY, USA, 2008. ACM.

[131] Hongtao Yu and Zhiyuan Li. Fast loop-level data dependence profiling. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 37–46, New York, NY, USA, 2012. ACM.

[132] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. ACM.

[133] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

[134] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 272–281, New York, NY, USA, 2006. ACM.

[135] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 169–180, New York, NY, USA, 2006. ACM.

[136] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 94–106, New York, NY, USA, 2004. ACM.

[137] Xiangyu Zhang and Rajiv Gupta. Matching execution histories of program versions. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 197–206, New York, NY, USA, 2005. ACM.

[138] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 319–329, Washington, DC, USA, 2003. IEEE Computer Society.

[139] Xiangyu Zhang, Armand Navabi, and Suresh Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 47–58, Washington, DC, USA, 2009. IEEE Computer Society.

[140] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. Towards locating execution omission errors. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 415–424, New York, NY, USA, 2007. ACM.

[141] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 263–271, New York, NY, USA, 2006. ACM.

VITA

VITA

William Nicholas Sumner was born in Midland, Michigan on October 16, 1982. He graduated *summa cum laude* from Hope College in 2005 with a Bachelor of Science in Computer Science and a Bachelor of Arts in German. He then attended Purdue University from 2005 through 2013, studying dynamic program analysis with Dr. Xiangyu Zhang. He joined Simon Fraser University as an assistant professor in the fall of 2013.