Crash Scene Investigation: Instrumentation and Postmortem Analysis for Deployed Applications

by

Peter Ohmann

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: 06/08/2017

The dissertation is approved by the following members of the Final Oral Committee:

Ben Liblit, Associate Professor, Computer Sciences
Aws Albarghouthi, Assistant Professor, Computer Sciences
Jeffrey Linderoth, Professor, Industrial and Systems Engineering
Loris D'Antoni, Assistant Professor, Computer Sciences
Thomas Reps, Professor, Computer Sciences

To Kendra, whose love and support made this dissertation possible.

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Ben Liblit. Ben gave me guidance but also the freedom to explore my passions in research. He was the fiercest skeptic when I was over-sure of a new idea, and he was a cheerleader when I lacked confidence in a research direction. To say that he had a profound influence on this thesis would be a colossal understatement. The skills that I have developed as a researcher, this document, and its contained body of work all exist because of his guidance.

I would also like to thank the other members of my dissertation committee: Aws Albarghouthi, Jeff Linderoth, Loris D'Antoni, and Tom Reps. Their feedback has been invaluable in improving the quality of this dissertation.

I have benefited immensely from many collaborations during my thesis work. My many co-authors and collaborators helped mold my understanding of various topics, and supported my development as a researcher. I was privileged to work with: Alexi Brooks, Dave Bingham Brown, Loris D'Antoni, Nate Deisinger, Manav Garg, Pallavi Ghosh, Ben Liblit, Jeff Linderoth, Naveen Neelakandan, and Tom Reps.

I would like to thank the current and former members of the Liblit research group: Arumuga Nainar, Cindy Rubio González, Tristan Ravitch, Mark Chapman, Alisa Maas, Dave Bingham Brown, and Zi Wang. Our countless discussions and problem-solving grinds in group meetings helped me to clarify the scattered ideas that eventually resulted in this dissertation.

I would also like to thank my other colleagues and friends in the Madison Computer Sciences department. Venkatesh Srinivasan has been an amazing friend for me to count on, as well as a brilliant researcher to help sift through new ideas. Ben Welton was my link to the world of computer systems research and always kept my spirits up with our weekly lunches. I would also like to thank the many other graduate students who attended my practice talks and provided advice, encouragement, and fun over these past years, including Jason Breck, Alex Cobian, Drew Davidson, Evan Driscoll, Stephen Lee, Tushar Sharma, and Aditya Thakur. I would be remiss not to acknowledge the amazing staff and resources at the University of Wisconsin, especially in the Computer Sciences department. I would like to especially thank Angela Thorp for her guidance in the complex process that is doctoral study, as well as her encouragement, advice, and willingness to stop to chat, especially in the busiest and most difficult times of these past years.

I dedicate this dissertation to wife, Kendra, who now knows more about computer science than she will ever willingly admit. She was with me through the successes, the struggles, and the occasional uncertainties of this dissertation. She was my English grammar guru, but, most importantly, my best friend on this journey. This dissertation would not have been possible without her love, encouragement, patience, and never-ending support.

To my parents, Roger and Ann, I am eternally grateful. They have supported me unceasingly throughout my entire life, and I could never begin to acknowledge all that they have done for me. The love they have shown me, and the values they have taught me, make up all of the good of the person that I am today.

My brother, Tony, and my sister, Julia, are the most caring siblings one could ask for. I would like to thank them for putting up with my antics, and for making every family gathering fun and memorable. I would also like to thank my brother-in-law, Nick Richards, and his wife, Kayla, for all of the fun times. My in-laws, Brent and Linda Richards, always believed in my potential, and welcomed me into their family; I am grateful for all that they have done for me. I would also like to thank my many aunts and uncles for always supporting me and keeping me laughing.

Finally, I would like to thank my other close fiends who have always stayed loyal to me. Kayla Bates was the only person I knew in Madison when I first moved here, and I am grateful for all of the Madison wisdom she passed along. Seth and Ashia Pollen have been great friends to Kendra and I, and helped make Madison feel like home. Lastly, I would like to thank Matt Schultz, Tyler Ohmann, Mark Scheibelhut, and Eric and Janelle Schulzetenberg for their constant friendship over the years.

This dissertation was supported, in part, by DARPA MUSE award FA8750-14-2-0270; DoE contract DE-SC0002153; LLNL contract B580360; NSF grants CCF-0904371, CCF-0953478, CCF-1217582, CCF-1318489, CCF-1320854, and CCF-1420866; a grant from the Wisconsin Alumni Research Foundation; and a CodeSurfer license generously provided by GrammaTech, Inc. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author, and do not necessarily reflect the views of the sponsoring agencies.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Debugging is a difficult, error-prone, and tedious task. Debugging, testing, and verification can account for 50–75% of a software project's cost [68]; these costs can grow even higher in some cases [64, 165]. Nevertheless, or perhaps partly because of this, complex production software inevitably contains bugs, and post-deployment failures are common. These production failures increase the cost of bug repair substantially, as they are difficult to reproduce and are initially observed by non-experts. Detailed postmortem crash reports can theoretically alleviate this problem. In particular, a developer would greatly benefit from a concrete, reproducible, full execution trace. Sadly, full execution tracing is usually impractical for complex programs. Even for simple code, the overhead of full tracing may only be acceptable during in-house testing.

The first part of this dissertation describes new techniques and tools for lightweight instrumentation of deployed applications. We describe techniques that trace program paths and that gather program coverage data at various granularities. We also present new algorithms for optimizing program coverage instrumentation based on customized requirements. Across all of these approaches, our techniques focus on extremely low-overhead control-flow tracing, and take advantage of readily-available information from failure reports (e.g., stack traces) to avoid redundant tracing and prioritize information to maintain when trace data must be limited due to tight overhead requirements.

The second part of this dissertation defines analysis techniques that operate on post-deployment failure reports. These reports may contain a variety of elements, but usually will not contain a full trace of run-time information from the failing execution. Thus, our analysis techniques are specifically designed to grapple with the imperfect information contained in production-run failure reports (including, but not limited to, our own tracing techniques). Our techniques aid developers in the difficult task of debugging post-deployment failures by restricting the possible program code a developer needs to consider, and by answering user control-flow questions about the failing execution.

1 INTRODUCTION

Modern software is often large and complex. Software testing, particularly for large systems, is expensive and time-consuming to conduct thoroughly, and complete verification of a software system is generally not possible. As a result, modern software is rarely or never bug-free when released to end users. Failures of software that occur after initial release are *post-deployment failures*.

When software fails (whether in the testing lab or post-deployment), a developer must *debug* the application to understand, identify, and repair the fault. Debugging is a difficult, error-prone, and tedious task, even for failures directly observed by a developer in the testing lab. Debugging, testing, and verification can account for 50–75% of a software project's cost [68]; these costs can grow even higher in some cases [64, 165]. Nevertheless, or perhaps partly because of this, complex production software inevitably contains bugs, and post-deployment failures are common. These production failures increase the cost of bug repair substantially, as they are difficult to reproduce and are initially observed by non-experts.

Detailed postmortem crash reports can theoretically alleviate this problem. In particular, a developer would greatly benefit from a report with sufficient information to concretely and completely reproduce the failing execution. Sadly, the most natural method of achieving this goal—tracing all statements executed and external events that occur during the failing execution—is usually impractical or impossible for complex programs deployed at user sites. Many techniques can reduce the cost of execution tracing [21, 22, 25, 98, 182]. Nevertheless, the overhead of these techniques (often many times the original execution time) remains a substantial obstacle to their adoption in post-deployment monitoring scenarios. Even for simple code, the overhead of full tracing may only be acceptable during in-house testing.

Thus, because deployed applications usually cannot gather full traces of run-time information, they must instead leave behind failure reports with varying detail. Crash-reporting systems are becoming more and more commonplace in enterprise software systems [16, 65, 66, 128]. These systems usually report only very limited information from a crash (e.g., stack traces and dumps of register values). While these systems are often very efficient, and have been used for grouping related failure reports or matching reports to known bugs [19, 43], the information they report often leaves significant ambiguity about the failing execution.

Debugging from incomplete failure reports is especially difficult. To counter these challenges, postmortem analysis tools aid developers by disambiguating failing executions. For example, some tools focus on replaying the failing execution [35, 36, 42, 78, 79, 153, 193],

while others attempt to locate potentially-faulty program locations [2, 23, 79, 82, 102, 106, 150, 195], and still others attempt to locate code potentially relevant to the failure condition [5, 6, 72, 83, 94, 180, 184]. However, most existing postmortem debugging tools either expect very dense execution information from the failing run (e.g., a full, repeatable trace) that is too expensive to collect from production runs, or utilize only very basic information (e.g., the failing stack trace). Sadly, stack traces alone leave significant ambiguity about a program's execution (as discussed further in Chapter 5). Developers must weigh run-time overhead (including time, memory, and disk space) against the benefits of traced data for postmortem failure analysis. Prior approaches to post-deployment monitoring [25, 44, 85, 139, 143, 170] routinely give up perfect information, which necessarily results in an *incomplete picture* of any failing execution. Postmortem analysis tools, then, should ideally be flexible to incorporate a wide variety of inexpensive data that may be collected in a production failure report.

This dissertation supports debugging of post-deployment failures by advancing the state-of-the-art in two primary areas: (1) lightweight monitoring of deployed applications, specifically focusing on monitoring just the *control flow* of traced applications, and (2) postmortem analysis based on failure reports of varying detail. One central principle of this work is that these two areas of research are most effective when aware of the strengths and weaknesses of one another. Specifically, tracing should be extremely lightweight while remaining focused on data most useful to postmortem analysis, and should be complementary to other failure report elements; analysis techniques should be targeted specifically to gracefully handle inevitably-imperfect failure reports from post-deployment failures.

1.1 Targeting Deployed Applications

The most effective means of handling bugs in user-facing applications is to ensure that all bugs are fixed prior to release. Many software development processes and pre-deployment tools have been developed to help identify and fix bugs in the testing lab. Testing practices, such as test-driven development [126, 181] and regression testing [138], take great strides in improving software quality. Other tool-based solutions include static (e.g., formal verification [11, 56]) and dynamic (e.g., code coverage [186], profiling [21, 22], and record-and-replay [57, 161]) techniques.

Sadly, despite these efforts, deployed software inevitably contains bugs. In fact, even in applications with thorough test suites and rigorous coding/build standards, bugs inevitably slip through into deployment. We performed a survey of bug reports for `gcc`, the most common

Table 1.1: Bug reports for supported gcc releases

Month	Supported Stable Releases	Bug Report Counts	
		Stable Releases	Long-Term Support Releases
Oct. 2016	5.4; 6.2	45	71
Nov. 2016	5.4; 6.2	47	72
Dec. 2016	5.4; 6.2	43	66
Jan. 2017	5.4; 6.3	38	62
Feb. 2017	5.4; 6.3	45	73
Mar. 2017	5.4; 6.3	37	64

C/C++ compiler in modern use. gcc was initially released in 1987, and has, on average, 3 releases of each supported version per year. Over this time, gcc has grown into a very complex piece of open-source software, managed by a large team of developers who set rigorous standards for testing and release guidelines.

Table 1.1 shows the results of our survey. We examined the public bug database for gcc for the 6-month period between October 2016 and March 2017. At any time, gcc has two current releases that are actively supported, and provides long-term support (major bug fixes) for all gcc releases since 3.3.4 (released in May 2004). For each of the 6 months, we list the current stable releases at that time, and a count of the number of *true* bug reports submitted by users during that time window, broken into those that affect stable releases, and those that affect any supported version. We define a *true* bug report as a report that has been confirmed by developers, and is either unresolved or resolved as fixed. (This excludes reports marked as enhancements, duplicates, or invalid.)

Overall, the results indicate that post-deployment failures are quite common, even for a project as mature as gcc. Stable releases average 1.3 to 1.6 unique bug reports per day in the months surveyed; including long-term support releases increases this average to between 2.0 and 2.6 reports. Through manual inspection, we found that the vast majority of these reports indicate either crashes or failures producing bad output.

Other research supports our findings. A 2008 study by Jones [80] finds that each stage of testing averages only 30%–35% removal efficiency, and that the average defect removal efficiency is about 85% in the United States (i.e., 15% of bugs will make it through to production). Within the companies surveyed, software averaged 5 defects per function point (i.e., functional unit or requirement), and larger applications had significantly higher defect density. Another study by Jones [81] in 2016 confirmed these earlier findings. Here, Jones

found an average of 4.25 defects per function point, but applications of the largest size (> 10,000 function points) were far worse, at 8.25 defects per point on average.

Post-deployment failures would not be problematic if they were easy to debug and fix. However, on the contrary, debugging failures from released software is notoriously difficult. A recent study by Britton et al. [32] found that developers spend 50% of their time debugging, and half of that time (25% overall) specifically on fixing bugs in existing code; this effort amounts to $312 billion per year spent on debugging (including wages and administrative overhead), with half of this debugging time allocated to fixing post-deployment bugs.

Sadly, most of the pre-deployment tools and techniques listed earlier in this section are not applicable or acceptable for use in deployed applications (e.g., due to time or memory overheads, or because failure reports contain insufficient data to run the tools). Recent research has provided tools that are specifically targeted for tracing or analyzing failures from deployed applications. (We discuss this related work in more detail in Section 1.3 and throughout the following chapters.) However, industry adoption of more powerful debugging tools and tracing systems for deployed applications has been very slow [145]. Most systems in use today gather only stack traces, system configuration, or partial memory dumps [16, 43, 52, 65, 66, 122, 128], and developers debug almost exclusively via the most basic, traditional debugger primitives and manually-inserted logging statements [145].

1.2 Tracing and Analyzing Post-Deployment Failures

Recent research suggests that more detailed failure report information substantially increases the effectiveness of debugging and postmortem analysis [42, 78]. In some cases, failure reporting systems may include a more extensive "core dump" file containing a snapshot of the failing program's memory. Typically, a core dump includes the full program stack at termination, and, configurably, some portion of the program's stack or heap data. Nevertheless, both our own analysis experiments (see Chapter 5, Section 5.3) and other postmortem analysis research [153] indicate that core dumps alone leave significant ambiguity in the program's activity during the failing execution.

On the other extreme, one might consider tracing all operations and environment interactions for the entire execution of a failing program. This situation is ideal in that it allows complete reproduction of the failing execution; a recent survey of developers at major software companies indicates that the ability to reproduce a failing execution (via a failing test case with steps to reproduce the failure) is the most valuable possible outcome of a failure report [197].

However, tracing a complete execution is infeasible for nearly all deployed software—and most software in the testing lab as well—due to large execution-time overheads, excessive memory and disk usage, and user privacy concerns. In general, users of deployed applications are likely to expect *negligible* overheads such that the performance of their applications is unaffected by any additional tracing. Naturally, these overhead thresholds will vary considerably by user and application. While recent work by Cornejo et al. [41] suggests that users of highly-interactive applications could tolerate overheads as high as 30%, many existing tools have even higher overheads. Furthermore, variety in overhead requirements for deployed applications suggests that monitoring tools should be *customizable* and able to adapt to user requirements.

To gather additional trace data from an application, one may *instrument* the program by adding additional tracing code to the original program. Undoubtedly, one seeks a delicate balance between the compile-time and run-time costs of instrumentation and the power of analysis for deployed applications. The central thesis of this dissertation is that

> **Small amounts of carefully-chosen runtime data, coupled with postmortem analysis techniques designed for imperfect failure reports, can yield substantial reductions in execution ambiguity from deployed application failures.**

Our defense of this thesis has two key parts: (1) techniques for extremely lightweight instrumentation and tracing customization, and (2) novel postmortem analysis techniques that scale gracefully from complete execution information to inevitably-incomplete failure reports from crashes of deployed applications. We implement these goals in the `CSI` (Crash Scene Investigation) toolkit for instrumentation and analysis of deployed C/C++ applications.

1.2.1 Instrumentation

To address the first part of the thesis, we develop and evaluate program instrumentation techniques and methods of tracing customization. These techniques are realized in our C/C++ compiler, `csi-cc`. We adopt the key principles that instrumentation must:

1. be efficient in both time and space,
2. avoid all kernel interaction, including disk I/O or system calls,
3. be complementary to other failure data (e.g., scale with and hook into existing stack-trace data), and
4. be customizable post-deployment.

All of our tracing is focused on the *control flow* of the program; this focus keeps our instrumentation efficient (as we never need to examine data values in our code), and partially mitigates privacy concerns relating to accidental sensitive data collection by our techniques.

We develop two primary tracing mechanisms that follow these principles, both presented in Chapter 2. The first is an adaptation of a well-studied path profiling approach by Ball and Larus [22] to bounded path tracing. Here, we trade out the traditional array of path execution counts (as used in profiling) for a fixed-size array of recent paths to maintain a suffix of recent execution in each active stack frame. The second instrumentation strategy is a class of in-memory program coverage mechanisms, gathered at various granularities. Specifically, we examine coverage data at functions, call sites, and statements; we find that coverage at call sites strikes a useful balance between the run-time cost of instrumentation versus the denser information provided for our postmortem analyses.

We next address the final of our key principles above, and develop two strategies for tracing customization (utilizing latent instrumentation code embedded in the executable); this work is presented in Chapter 3. We find that a surprisingly straightforward "trampoline function" approach is most efficient in this context, after also examining a more advanced technique based on GNU indirect function call attributes [114].

Finally, in Chapter 4, we develop a new parameterized approach to optimizing program coverage instrumentation that, beyond any existing techniques, is able to optimize instrumentation in cases where coverage requirements and allowed instrumentation points are limited. While prior research optimizes program coverage instrumentation in various contexts [3, 4, 21, 88, 89, 105, 169], our technique is the first to allow customized requirements. The input to our approach includes: (1) the subject program, (2) a set of program points for which we must ensure accurate coverage data in any failure report, and (3) a set of allowed instrumentation points (possibly not all program points). These optimizations apply to many situations in deployed software, including gathering coverage data for code not covered by one's in-house test suite [143], and our new program coverage mechanisms described above.

1.2.2 Analysis

To address the second part of the thesis, we develop and evaluate novel postmortem analysis techniques that operate on partial report data from failing executions, in various forms. As previously stated, a complete replayable execution trace is very valuable here, and recent research attempts to derive failing inputs and thread schedules from failure reports at various levels of detail [35, 36, 42, 78, 79, 153, 193], specifically using symbolic execution. We

discuss some of these techniques in more detail in Section 1.3. These approaches are promising, but can be very expensive in both time and memory cost for analysis; in the general case, replaying an execution via symbolic execution is undecidable. Our analyses instead take a different approach. We specifically focus on providing information that is valid for *any* run of the failing application that could produce the provided failure report data.

Our first analyses, realized in our tool `csi-spotlight` described in Chapter 5, make specific use of our path trace and program coverage data from the previous section. The first analysis technique restricts the set of possible statements and branches that may have been executed during the failing run. We begin from a provided stack trace with optional partial path trace data (per our instrumentation approaches), and use a stack-restricted backward reachability analysis to eliminate code that could not possibly have been executed on the failing run, based also on (optional) provided program coverage data. Our second analysis technique is based on backward program slicing, which is used to compute the possible set of statements that may have transitively affected a particular point of interest during execution (in our case: the point of failure). This restricted set of possibly-relevant statements is called a program *slice*. We constrain the program dependence graph [140], a traditional static representation of a program's possible flows of control and data. Our approach is again a stack-restricted backward reachability analysis, but, in this case, we also eliminate inconsistent *data* flows, based on the *control* flow information fed as input to our analysis. Our technique builds upon work in dynamic program slicing [6], which computes a more accurate slice based on a full trace of program events from a single execution. We instead facilitate *hybrid* slicing: all returned statements may affect the failure point on *some run* consistent with the failure report data.

Our second analysis technique, realized in our tool `csi-grissom` described in Chapter 6, extends the first analysis of `csi-spotlight` by allowing users to pose a much richer class of control-flow queries over the failing program's execution. Obtaining answers to questions about a failing execution is a crucial part of debugging, and `csi-grissom` extends `csi-spotlight` in two important ways. First, we allow users to ask a wide variety of control-flow questions, including, but not limited to, questions about statements that may or may not have executed on the failing run. Second, we support a much larger class of failure report elements, encoded as formal languages over the program's control-flow graph. We also introduce and study a new class of subregular languages, the unreliable trace languages (UTL). If all failure constraints and the user's query can be given as UTL, we present a new algorithm for precisely answering user queries in polynomial time.

Finally, in Chapter 7, we describe the design of a debugging tool, `CSIclipse`, that facilitates exploration of tracing and analysis result data in an integrated development environment. Usefulness for actual users debugging deployed applications is the ultimate test of our analysis techniques. Accordingly, we also lay out a preliminary design for a human subjects study to assess the utility of the `CSIclipse` tool.

1.2.3 The Big Picture

One of our key contentions is that the instrumentation and analysis components of tools for failing deployed applications do best when working together. All of our techniques are designed with this assertion in mind. Our instrumentation approaches are designed to complement common existing failure report elements (e.g., stack traces), while our analysis techniques are specifically designed to scale gracefully in the presence of imperfect failure information from post-deployment failure reports. We assess the trade-offs in instrumentation overheads and analysis precision, as well as the advantages of combining our lightweight instrumentation and analysis approaches, as we conclude in Chapter 8.

1.3 Related Work

This section describes high-level related work that is related to the dissertation generally. Individual chapters discuss prior work that is related to specific approaches, algorithms, and tools.

As stated in the previous section, many prior approaches use symbolic execution to replay executions from failure reports [35, 36, 42, 78, 79, 153, 193]. Rößler et al. [153] use symbolic execution and genetic algorithms to synthesize a sequence of unit test cases that closely match data from a core dump. Cao et al. [35] perform selective recording of return values for functions which are most difficult for a symbolic execution engine to reason about. They show that focusing on difficult functions can substantially reduce tracing overhead while still tracing enough information to deterministically replay many failures. Jin and Orso [78] use symbolic execution to synthesize an execution matching failure trace data at a variety of granularities. Jin and Orso [79] then build on this previous work to generate multiple similar success and failure runs to assist in fault localization. In both cases, the replayed execution then matches whatever conditions are given: just a failure location will result in a failure at the same point, while a full execution trace will result in an exact match. While Jin and Orso reduce tracing overheads substantially relative to full tracing, they still measure

runtime overheads near 50% for two subject programs. Complete replay is potentially very useful for developers, but finding matching failure inputs is undecidable in the general case, and often very computationally expensive, regardless of the density of trace data collected. Such techniques also make difficult trade-offs: with very limited failure report data, the replayed execution is unlikely to closely match the actual failing execution [78]. However, with very dense data, the cost of both tracing and replay is higher. As stated previously, we see replay techniques as complementary to our instrumentation and analysis techniques. Our instrumentation data is less expensive to collect than most previous techniques, but is systematically tied to existing data (e.g., stack traces and core dump data) in a manner that may be useful for replay engines. Our analyses target a different problem in failure understanding, reporting data that is valid for *any* run matching failure report data, helping developers to generalize beyond *one specific* run.

As an alternative, some research sacrifices perfect replay for efficiency. Prior techniques facilitate lightweight checkpoint-and-replay of production-run failures [109, 171], and partial replay of long-running server applications [37]. Cheung et al. [37] log branches and array accesses for symbolic replay, but drop trace data at pre-specified execution checkpoints to reduce overheads and log size, allowing replay of a suffix of the failing execution. Liu et al. [109] take advantage of the fact that memory corruption errors leave behind clear evidence that they occurred, and present an extensible system to scan for various common problems (heap overflows, uses after free, and memory leaks) at each execution region separated by non-replayable system calls. Our approaches do not specifically target memory errors, but we share the motivation for utilizing data that is already present in failing core dumps. These techniques are more efficient than their full-replay counterparts, and share our motivation for prioritizing dense trace information near the failure point. As with full replay, we see this work as complementary to our own, providing partial replay of a single failure, while our analyses generalize to aid developer understanding of the bug underlying the failure. Our analyses could complement a bounded replayable trace with restricted views of global execution information and general user control-flow queries about the failing execution.

Artzi et al. [17] reproduce failures by tracking arguments to methods in a shadow stack, and take advantage of object-oriented properties by shadowing values at the object level. This work is similar to ours in its exploitation of available runtime structure (in this case, object-oriented features), but with a different goal: producing method unit test cases. Manevich et al. [112] use backward data-flow analysis to reproduce failing executions based on only a failure location

and typestate information regarding the failure. While efficient, this approach is limited to solving specific typestate problems with simple types.

Yuan et al. [190] introduce the SherLog tool which performs postmortem analysis from log files. SherLog infers paths that must, may, or cannot have executed between logging points, and derives data-flow information for those paths. Yuan et al. [191, 192] also analyze programs to enhance existing logging and capture additional state for SherLog's analysis. Our analyses do not use free-form log data, though we briefly discuss how log data might be encoded for our `csi-grissom` analysis in Chapter 6. Data values from log messages could complement our lightweight control-flow tracing mechanisms, particularly for use in postmortem analysis techniques, including our own.

Prior research [18, 31, 110, 139] emphasizes adaptive post-deployment instrumentation with data collection aggregated across large user communities. Such approaches are complementary to our own: we focus on gathering very valuable information at very low cost, while these related efforts focus on how best to deploy information-gathering instances. This work is most closely related to our tracing customization work described in Chapter 3; there, we discuss this related work in more detail.

Some prior approaches rely on complete tracing of executions [90] or environment interactions [39]. Intensive data collection of this sort allows for rich functionality, such as deterministic replay [39] or detailed visualization of program paths and user queries in an integrated development environment [90]. However, the resulting overheads may be prohibitive for deployed software. Our techniques throughout this dissertation explicitly trade off detail for efficiency, and demonstrate that much of value can be learned even from impoverished but targeted failure report data.

1.4 Contributions

The primary contributions of this dissertation are frameworks and tools for supporting the challenging and tedious task of debugging deployed applications. More specifically, the techniques and tools described here enable more efficient instrumentation of deployed applications, and more effective analyses of incomplete failure reports arising from failing executions of these applications. The contributions are summarized as follows:

- We develop two complementary classes of program tracing mechanisms—path traces and in-memory program coverage techniques—that augment core memory dumps with lightweight, tunable tracing with performance suitable for deployed applications

(Chapter 2). These mechanisms are realized in an instrumenting C/C++ compiler, `csi-cc`.

- We design and evaluate means for efficient developer and end-user customization of `csi-cc` tracing after deployment (Chapter 3).
- We introduce the problem of determining optimal program coverage instrumentation based on user-specified requirements: desired coverage locations, as well as the set of locations that are valid for instrumentation. We present three approaches to optimization—one fully-optimal and two approximations—that substantially reduce instrumentation required to trace coverage information (Chapter 4). Our approaches specifically target those requirements that arise in tracing of deployed applications, and are the first to optimize coverage instrumentation based on customized requirements.
- We develop two analysis techniques that specifically take advantage of our efficient `csi-cc` tracing mechanisms (Chapter 5). Our first analysis limits the set of potentially-executed statements and branches in a program; our second analysis is a novel technique for static program dependence graph restriction based on partial dynamic trace data. Our analyses, realized in the `csi-spotlight` engine, substantially reduce the relevant code a developer would need to consider when debugging a post-deployment failure.
- We introduce two variants (with and without calling-context sensitivity) of a query recovery problem for answering arbitrary user-specified control-flow queries given a program and a failure report from a run of that program. We develop two analyses based on formal automata encodings that can encode a wide class of failure constraints, and can precisely answer user queries for real-time or batch analyses (Chapter 6). Our analyses are realized in the `csi-grissom` engine.
- We develop and study a new class of subregular languages, the unreliable trace languages, that are particularly suited to answering context-insensitive control-flow queries in polynomial time (Chapter 6). Our `csi-grissom` system answers queries remarkably efficiently when we encode failure constraints and user queries entirely as unreliable trace languages.
- We present the design of a front-end tool, `CSIclipse`, which allows developers to display and navigate the tracing and analysis data from our previous contributions (Chapter 7). We also detail a preliminary design for human subjects evaluation of our techniques.

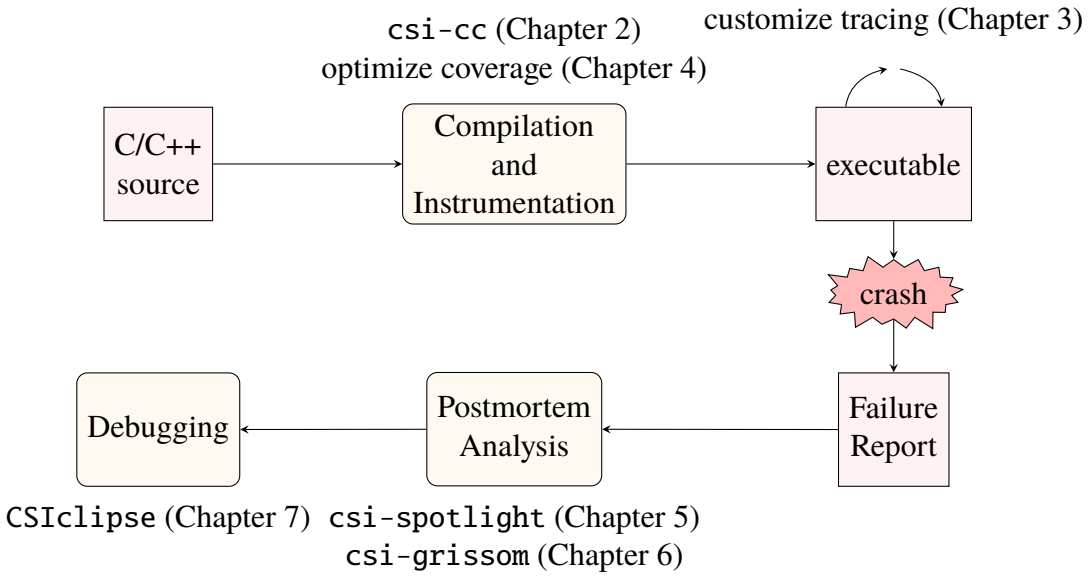


Figure 1.1: CSI system, annotated with contributions

All of these contributions are part of the CSI framework for instrumentation and analysis. Figure 1.1 shows the overall CSI system architecture, and positions our contributions within this framework. We discuss each contribution in further detail in the relevant section(s).

1.5 Dissertation Organization

The remainder of this dissertation is organized into two major parts. We describe techniques for lightweight instrumentation of deployed applications in Part I; this part is broken into three chapters. First, in Chapter 2, we describe four new low-cost tracing methods based on program coverage and path tracing. Then, in Chapter 3, we describe approaches to dynamic customization of program tracing, allowing end-users to change active tracing techniques post-deployment to meet changing overhead or failure report data requirements. Finally, in Chapter 4, we describe novel techniques for optimizing program coverage instrumentation based on instrumentation requirements and limitations.

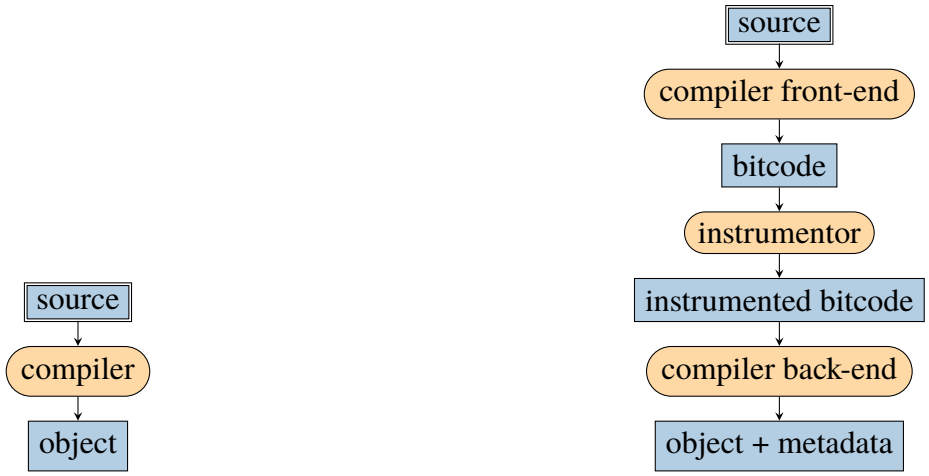
In Part II, we describe our analysis techniques specifically targeting failure reports from deployed applications; this part is again broken into three chapters. In Chapter 5, we describe analysis techniques that reduce the amount of code a developer would need to consider while debugging, by computing a reduction on the possible control and data flows that may have occurred during the failing run based on the provided failure report. Then, in Chapter 6,

we describe a generalized system for answering user control-flow queries based on failure report data that can take advantage of a much wider wide range of possible failure constraints. Finally, in Chapter 7, we describe a front-end tool that can display `CSI` tracing and analysis data within the Eclipse integrated development environment, and lay out a preliminary design for a human subjects study to further assess the quality of our analysis data.

In Part III, we conclude with a summary of the contributions of this dissertation and the important relation between instrumentation and analysis design for deployed applications, lessons for future tool developers, and possible directions for continued research. Part IV contains appendices, with proofs supporting the empirical evaluations in the dissertation, and a full mathematical optimization model for the program coverage optimizations from Chapter 4.

Part I

Instrumentation



(a) Standard compilation **(b) `csi-cc` compilation**

Figure 1.2: Compilation process interposition for `csi-cc`. Sharp-cornered rectangles represent inputs and outputs; rounded rectangles represent computations.

In this part, we describe new techniques for instrumentation and configurable tracing of deployed software. These approaches are realized in the `csi-cc` instrumenting C/C++ compiler. All of the instrumentation techniques described in this part share some common engineering and experimental design features that we describe here.

We use Clang/LLVM 3.5 [101] to compile and instrument programs. Instrumentation operates directly on LLVM bitcode. Thus, we interpose on the normal compilation process (which proceeds directly from source code to object file). Figure 1.2 shows our modifications. We insert an additional step to generate LLVM bitcode from a source file, run our instrumentor and any command-line compile-time optimizations on the bitcode, and then compile the resulting instrumented bitcode to an object file. Note that our instrumentation generates metadata that is necessary for postmortem anlayses; we describe this data in the relevant sections of the following chapters. In all following chapters, we abbreviate the process shown in Fig. 1.2b, instead referring to the entire compilation process as instrumentor.

In all empirical evaluations, we measure overhead as the ratio of times for instrumented and uninstrumented code, meaning that we will also capture any compile-time cost or impacts on base compiler optimizations introduced by our intervention in the standard compilation process. This emulates the true cost a developer would need to pay to use `csi-cc` as a drop-in replacement for existing compilation infrastructure.

All experiments used a quad-core Intel Core i5-3450 CPU (3.10 GHz) with 32 GB of RAM running Red Hat Enterprise Linux 6. Most applications in our evaluation are shared

across the remaining chapters of the dissertation. Unless otherwise stated, we obtained all subject applications from the Software-artifact Infrastructure Repository [55, 154]. All of our applications contain known faults that can be enabled or disabled at compilation time, allowing us to reuse the same subjects for our instrumentation and analysis experiments. However, for instrumentation experiments, we always gathered our compilation and execution times over the non-faulty build of each application. Since developers and users alike would gladly pay extra overhead cost on failing runs (for the diagnostic value of traced data), our process measures the relevant worst-case behavior: extra time and memory cost on non-failing runs.

All of our instrumentation techniques maintain all state in process memory. Thus, we do not report disk or log usage for any of our experiments.

2 TRACING AND COVERAGE

Substantial portions of this chapter are derived from a 2013 conference paper by Ohmann and Liblit [135], and a 2016 journal paper by Ohmann and Liblit [137].

As previously stated, debugging is a difficult, time-consuming, and expensive part of software development and maintenance. Full execution tracing is impractical for modern complex software, but, nevertheless, developers would benefit greatly from seeing concrete traces of events leading to failures, failure-focused views of the program or program state, or suggestions of potentially-faulty statements.

One common and very useful artifact of a failed program execution is a core memory dump. Coupled with a symbol table, a core dump reveals the program stack of each execution thread at the moment of program termination, the location of the crash, the identities of all in-progress functions and program locations from which they were called, the values of local variables in these in-progress functions, and the values of global variables.

In this chapter, we present instrumentation techniques that augment core dumps with lightweight, tunable tracing. We explore four such enhancements: (1) a variant of Ball–Larus path profiling [22], (2) function coverage, (3) statement coverage, and (4) call-site coverage. We also briefly highlight the difference between our instrumentation (which is static) and our run-time tracing (which is customizable). We introduce the notion of a "scheme" to describe a possible tracing configuration, which is a core concept for our customization approaches described in Chapter 3.

Our techniques are realized in `csi-cc`, an instrumenting C/C++ compiler. We evaluate the trade-offs among the above tracing methods, and conclude that pairing our path profiling variant with call-site coverage yields a complementary, realistic, and valuable choice for deployed applications. Our results for this pairing with a realistically-tuned tracing scheme show low overheads (0%–3% execution time, 0%–4% memory) suitable for production use. As we later discuss in Chapter 5, this pairing also results in a substantial reduction of failure execution ambiguity for our analyses, even at this minuscule cost.

Figure 2.1 shows an overview of the `csi-cc` compilation process and associated artifacts. Note that the instrumentor operations abbreviate the process from Fig. 1.2b. Each feature of this diagram is described in the remainder of this chapter.

Figure 2.2 shows an example that we will refer to throughout this chapter. The source code in Fig. 2.2a is taken from flex, one of the applications used in various evaluations

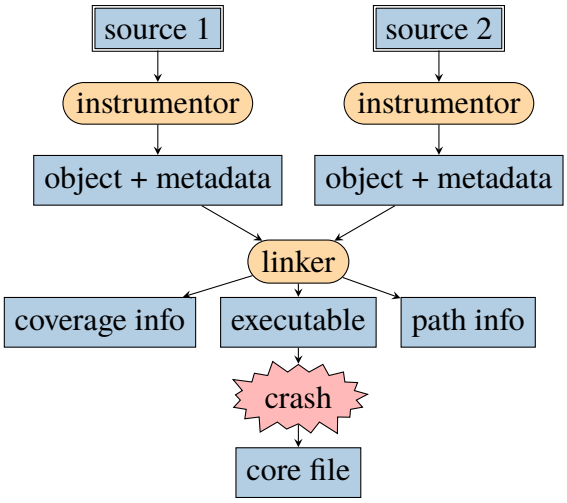


Figure 2.1: Overview of `csi-cc` data collection. Sharp-cornered rectangles represent inputs and outputs; rounded rectangles represent computations.

throughout this dissertation. For the examples in this chapter, we will most often make use of the function's intraprocedural control-flow graph (CFG) representation, shown in Fig. 2.2b.

2.1 Background

In this section, we provide necessary background information for the approaches described in the remainder of this chapter. We begin by describing core dumps and their benefits for postmortem debugging. We then review a well-studied path profiling approach by Ball and Larus [22]; this chapter develops a variant of this approach.

2.1.1 Core Memory Dumps

All widely-used modern operating systems can produce a "core dump" file containing a snapshot of a program's memory. A dump may be saved after abnormal program termination due to an illegal operation (such as using an invalid pointer) or on demand (such as by raising a fatal signal or failing an assertion). This can be useful if the core dump is to be used for postmortem analysis.

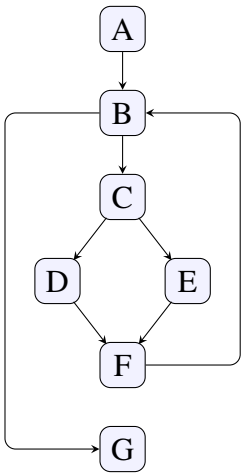
Typically, a core dump includes the full program stack at termination, including the final values of all local variables and program registers, and some portion of the heap. For our purposes, the key elements are the point of failure (the exact location of the program crash), as well as the most-recent call location in each other still-active frame on the stack (i.e., each

```
void add_action(char *new_text) {
  int len = strlen(new_text);

  while (len + index >= size - 10) {
    int new_size = size * 2;
    if (new_size <= 0)
      size += size / 8;
    else
      size = new_size;
    array = realloc_char_arr(array, size);
  }

  strcpy(&array[index], new_text);
  index += len;
}
```

(a) Code example



(b) Control-flow graph

Figure 2.2: Example code

stack frame's return address). Conveniently, core dumps are only produced in the case of program failure. Thus, collecting them imposes zero run-time overhead. Furthermore, any trace data kept in-memory within the program stack will be automatically dumped if and only if the program produces a core dump. These are two key advantages to our use of core dumps for lightweight tracing and postmortem analysis.

2.1.2 Path Profiling

Path profiling is traditionally used to compute path coverage during program testing. The approach we adopt from Ball and Larus [22] is designed to efficiently profile all acyclic, intraprocedural paths. The algorithm first removes back edges to transform the control-flow graph (CFG) of a procedure into a directed acyclic graph (DAG). We represent the transformed CFG as a single-entry, single-exit DAG $G = (V, E, s, x)$ where V is the set of nodes in the graph and $E \subseteq V \times V$ is the set of edges with no directed cycles. Every node in V is reachable by crossing zero or more edges starting at the unique entry node $s \in V$. Conversely, the unique exit node $x \in V$ is reachable by crossing zero or more edges starting from any node. A path p through G is represented as an ordered sequence of nodes $\langle p_1, \ldots, p_{|p|} \rangle$ such that $(p_i, p_{i+1}) \in E$ for all $1 \leq i < |p|$. We define a *complete path* as a path whose initial and final nodes are s and x, respectively. Let C represent the set of all complete paths; note that this

set is finite since G is a DAG. Loops are handled specially, and are discussed later in this subsection.

The overall goal of the Ball–Larus algorithm is to assign a value $\mathit{Increment}(e)$ to each edge $e \in E$ such that

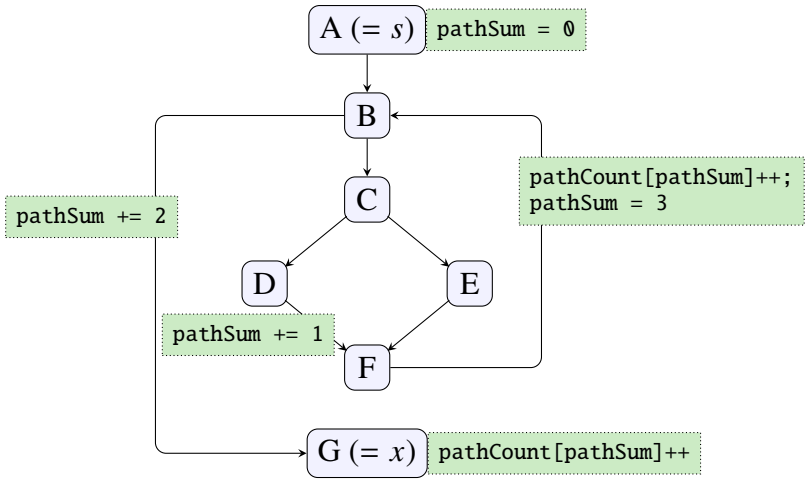
1. each complete path in C has a unique *path sum* produced by summing the *Increment* values for each edge in the path;
2. the assignment is minimal, meaning that all path sums lie within the right-open interval $[0, |C|)$; and
3. the assignment is optimal, meaning that each path requires the minimal number of non-zero additions.

The first step assigns a value to each edge such that all complete path sums are unique and the assignment is minimal. To do so, the algorithm traverses the graph in reverse-topological order. For each $n \in V$ we compute $\mathit{NumPaths}[n]$, the number of paths from n to x. If we number the outgoing edges of n as $e_1, \ldots, e_k$ with respective successor nodes $v_1, \ldots, v_k$, then the weight $\mathit{Weight}(e_k)$ assigned to each outgoing edge of n is $\sum_{j=1}^{k-1} \mathit{NumPaths}[v_j]$. After this step, complete path sums using *Weight* values are unique, and the assignment is minimal.

The next step optimizes the value assignment. This step uses a *maximum-cost spanning tree* (MCST) of G. A MCST is an undirected graph with the same nodes as G, but with an undirected subset of G's edges that form a tree, and for which the total edge weighting is maximized. Algorithms to compute maximum-cost spanning trees are well-known. Remaining non-tree edges are *chord edges*, and all edge weights must be "pushed" to these edges (i.e., propagated forward or backward through G until only chord edges have non-zero weights). The unique cycle of spanning-tree edges containing a chord edge determines its *Increment*.

Instrumentation is then straightforward. We track the path sum in a register or local variable, `pathSum`, initialized to `0` at s. Along each chord edge, e, we update the path sum: `pathSum +=` $\mathit{Increment}(e)$. When execution reaches x, we increment a global counter corresponding to the path just traversed: `pathCount[pathSum]++`.

Cycles in the original CFG create an unbounded number of paths. Control flow across back edges requires creating extra paths from s to x by adding "dummy" edges from s to the back edge target (corresponding to initialization of the path sum when following the back edge) and from the back edge source to x (corresponding to a counter increment when following the back edge). The algorithm then proceeds as before. Because of the dummy



(a) Extra instrumentation code added for path profiling

pathSum	Path
0	ABCEF
1	ABCDF
2	ABG
3	BCEF
4	BCDF
5	BG

(b) Path numbers

Figure 2.3: Path profiling example

edges to x and from s, counter increments and reinitialization of the path sum occur on back edges. We expand our definition of a complete path to include paths that begin at back edge targets or that end at back edge sources.

Figure 2.3 shows possible instrumentation to profile paths in the example function from Fig. 2.2. Figure 2.3a shows the function's CFG annotated with `pathSum` and `pathCount` increments. Each acyclic path completes at either function exit or the loop back edge, and the counter for the path's value is incremented at that point. As shown in Fig. 2.3b, each acyclic path is uniquely numbered. Note that the assignment is clearly minimal, as each acyclic path contains at most one `pathSum` initialization, and one `pathSum` increment.

The preceding overview of path profiling focuses on details relevant to our path tracing variant, described in the following section; see Ball and Larus [22] for the complete, authoritative treatment. There has been a great deal of follow-on work since the original paper [13, 51, 115, 162, 173], and we comment on some of these more recent developments when discussing related work in Section 2.5.

2.2 Instrumentation Approaches

When considering which data to collect and how, several desirable properties guide our choices. Instrumentation must be **efficient** in time and space, and therefore suitable for production use. Data must be held **in memory** until failure, adding no I/O or other system

calls during normal execution. Data size must **scale** with aspects of execution state, such as stack depth or number of program locations. Lastly, results must be **mappable** back to source code, and contain as little ambiguity as possible.

Any core dump already records the return address of each active function at the time of failure. While this has all the above qualities, it may be insufficient on its own. Therefore, we augment core dumps with two novel techniques: path tracing and various forms of program coverage.

Specifically, in this chapter, we consider four tracing mechanisms (path tracing, statement coverage, call-site coverage, and function coverage), which we group into two different high-level methods of tracing. Our path tracing mechanism is an extension of path profiling from Ball and Larus [22]. Coverage mechanisms are all traced similarly, while only the traced program points differ. For each instrumented function, we produce a metadata file used to interpret traced data for reconstruction and postmortem analysis; we describe this metadata individually for each tracing mechanism.

2.2.1 Ball–Larus Inspired Path Tracing

Path tracing records the last N acyclic paths taken through each function on the stack at the time of failure. Like any stack-bound data, path traces are discarded whenever a function returns. We achieve this tracing using a variant of Ball–Larus path profiling. Rather than counting acyclic path executions, we instead record each completed acyclic path in a stack-allocated circular buffer.

However, completed paths alone do not yield an execution suffix. We also need the final "incomplete" path leading up to the failure. Fortunately, given a CFG G, failing node v, and a partial path sum w, we can recover the unique acyclic path that accumulates the value w and ends at v. This property is a natural consequence of the Ball–Larus approach: v and w are the only state maintained while determining acyclic paths, and therefore must constitute the system's entire "memory" of the partial path covered so far.

Theorem 2.1. *For any node $v \in V$, every* (`pathSum`, v) *pair either encodes a unique subpath in G or is infeasible.*

Proof. The proof is straightforward by contradiction. Suppose that two distinct subpaths, p and q, both begin at s, end at v, and share the same value of `pathSum`. G has a unique exit node, x, that is reachable from v via some sequence of edges $\vec{E}$. This sequence of edges need not be unique. Each of those edges has been assigned an increment, and, therefore, we can

```
void add_action (char *new_text) {
  volatile int pathSum = 0;
  volatile int pathTrace[N];
  volatile int pathIndex = 0;

  int len = strlen(new_text);

  while (len + index >= size - 10) {
    int new_size = size * 2;
    if (new_size <= 0) {
      size += size / 8;
      pathSum += 1;
    } else
      size = new_size;
    array = realloc_char_arr(array, size);

    pathTrace[pathIndex] = pathSum;
    pathIndex = (pathIndex + 1) % N;
    pathSum = 3;
  }

  pathSum += 2;
  pathTrace[pathIndex] = pathSum;

  strcpy(&array[index], new_text);
  index += len;
}
```

Figure 2.4: Path tracing instrumentation example. Highlighted code implements path tracing.

compute the sum of the "suffix" sequence of edges to be $w = \sum_{e \in \vec{E}} Increment(e)$. We can "complete the path" for both p and q by connecting each subpath to $\vec{E}$ and getting a total path sum of $w_{final} = w + \texttt{pathSum}$. However, we know that two acyclic paths do not share the same path sum by the proof from Ball and Larus [22]. Therefore, any feasible (pathSum, v) pair encodes exactly one subpath in G. □

Furthermore, no subpath can give rise to more than one possible (pathSum, v) pair, as edge increments are fixed during instrumentation. Therefore, the converse of the above proof is also true: every unique subpath in G is represented by a unique (pathSum, v) pair. We must merely guarantee that an accurate partial path sum is available at every point during execution, since our instrumentation pessimistically assumes that failure can occur at any time.

Figure 2.4 shows appropriate instrumentation for the example from Fig. 2.2. Note that the `pathSum` increments and `pathTrace` stores correspond to the path profiling instrumentation scheme shown in Fig. 2.3. Our implementation of path tracing includes a number of changes relative to standard Ball–Larus path profiling. We move array allocation into the stack, giving one trace (`pathTrace`) per active call. The size of this array determines how many acyclic paths are retained. This size is fixed at build time, defaulting to 10. (We performed preliminary experiments on small applications, varying the buffer size over several orders of magnitude up to 100,000. We find that overhead initially increases anywhere from 10%–40% per order of magnitude. Overhead eventually stabilizes once the array is so large that most of it is unused and therefore never mapped into memory.) Note that, because space for path traces is stack-allocated, it naturally scales directly with the stack depth. Its allocation is also "free" as no explicit allocation is required, and (depending on the choice of trace size) it has minimal impact on the size of a stack frame.

The stack-allocated array serves as a circular buffer. A local variable (`pathIndex`) tracks the current buffer position. At each back edge and function exit, we append the path sum (`pathSum`) for the just-completed path to this buffer. On back edges, the path sum is reinitialized (`pathSum = 3`) to uniquely identify paths beginning at the loop head. Obviously, we cannot instrument functions with more paths than can be counted in a machine integer. This issue rarely affects 64-bit platforms, though Section 2.3.1 notes one exception seen in our experimental evaluation. Instrumentation skips these functions, for which we simply collect no trace data.

We must be able to access the current path sum at any point, not just at the very ends of complete paths. For safety, we forbid the compiler from keeping this value in a register. Rather, both the path sum and the trace array are declared **`volatile`**.

Instrumentation produces a metadata file necessary for future analyses. For each function, we record (1) a full representation of the control-flow graph with edges labeled with path-sum increments; and (2) a mapping from basic blocks to line numbers. The linker aggregates this metadata into a single record for the entire executable: path info in Fig. 2.1.

2.2.2 Program Coverage

Path traces provide very detailed information close to the point of failure in each active stack frame. However, path traces have two major blind spots: old paths that have already rotated out of the circular trace buffer, and interprocedural paths through calls that have already returned.

Program coverage data can easily provide coarser-grained global information, allowing tracing to scale gracefully as the debugging task departs from the active crash stack. Coverage instrumentation uses one global array per instrumented function, and one local array (of the same size) for each stack frame. For a function f, we select a set of statements, which we call *trace points*. These trace points are numbered $0, 1, \ldots, n-1$; these serve as indices into f's local and global coverage arrays. Coverage that we gather is *binarized*, meaning that we record whether each trace point was ever executed (1) locally, in each particular invocation of f corresponding to a stack frame; and (2) globally, for any invocation of f across the entire program's execution. Thus, each trace point corresponds to one local coverage bit per active f stack frame plus one global coverage bit. Taken together, the local and global coverage bits have several desirable properties. The local bits offer up-to-date information for trace points in each still-active function. Space for these bits is stack-allocated, so, like path traces, it naturally scales with the stack depth. Conversely, the global coverage bits summarize data from completed calls that have already left the stack.

Note that many possible sets of trace points exist for a given function. We consider three alternatives in this work. First, one may elect to gather full statement coverage. Naïvely, one trace point could be used for each statement in f. However, one trace point per basic block in f is sufficient. Second, if one is interested in function coverage, one need only select one trace point per function. Any statement guaranteed to execute on any execution of f will do; we use function entry, as selecting function exit may require either multiple trace points or adding a shared exit block. Note that function coverage is unique in that it has no stack-local variant: all functions currently in the active stack are clearly executing.

Call-site coverage is the final coverage form we consider. Here, we have one trace point for each call site in f. Our use of call sites as the program points for which to gather coverage information is somewhat arbitrary. However, the choice is well-matched to its purpose. Call sites mark departures from the visible call stack; these are places where stack-based tracing (such as path tracing) cannot help us. Intuitively, coverage at call sites complements dense stack-local mechanisms where that help is most likely to be useful. We find that call-site coverage works extremely well in practice (see Part II of the dissertation, especially Chapter 5 where we consider analysis performance for each tracing mechanism individually).

Figure 2.5 shows appropriate instrumentation for the example function from Fig. 2.2. The three variants correspond to our three sets of trace points. This example also shows some of the subsumption relationships that hold among the three types of program coverage. Call-site coverage is more precise than function coverage for this particular function: it is able

```
volatile bool add_actionCov = false;

void add_action(char *new_text) {
  add_actionCov = true;

  int len = strlen(new_text);

  while (len + index >= size - 10) {
    int new_size = size * 2;
    if (new_size <= 0)
      size += size / 8;
    else
      size = new_size;
    array = realloc_char_arr(array, size);
  }

  strcpy(&array[index], new_text);
  index += len;
}
```

(a) Function coverage

```
volatile bool add_actionCov[3] = {false, false, false};

void add_action(char *new_text) {
  volatile bool cov[3] = {false, false, false};

  int len = strlen(new_text);
  cov[0] = add_actionCov[0] = true;

  while (len + index >= size - 10) {
    int new_size = size * 2;
    if (new_size <= 0)
      size += size / 8;
    else
      size = new_size;
    array = realloc_char_arr(array, size);
    cov[1] = add_actionCov[1] = true;
  }

  strcpy(&array[index], new_text);
  cov[2] = add_actionCov[2] = true;
  index += len;
}
```

(b) Call-site coverage

Figure 2.5: Program coverage instrumentation example. Highlighted code implements coverage.

```
volatile bool add_actionCov[6] = {false, false, false, false, false, false};

void add_action(char *new_text) {
  volatile bool cov[6] = {false, false, false, false, false, false};

  int len = strlen(new_text);
  cov[0] = add_actionCov[0] = true;

  while (len + index >= size - 10) {
    int new_size = size * 2;
    cov[1] = add_actionCov[1] = true;
    if (new_size <= 0) {
      size += size / 8;
      cov[2] = add_actionCov[2] = true;
    } else {
      size = new_size;
      cov[3] = add_actionCov[3] = true;
    }
    array = realloc_char_arr(array, size);
    cov[4] = add_actionCov[4] = true;
  }

  strcpy(&array[index], new_text);
  index += len;
  cov[5] = add_actionCov[5] = true;
}
```

(c) Statement coverage

Figure 2.5: Program coverage instrumentation example (cont.)

to determine whether the loop was ever taken via tracing the call to `realloc_char_array`. The subsumption relationship, however, does not hold in general, as a function may be a leaf function (i.e., contain no calls) or not be guaranteed to execute a call instruction on every path through the function. However, statement coverage always subsumes both function and call-site coverage. In the examples of Fig. 2.5, only statement coverage distinguishes the direction of the **`if`** statement within the loop.

Local coverage data is stored in a stack-allocated n-element array (`cov`), zero-initialized at function entry. A per-function global n-element array (`add_actionCov`), initialized at program start, holds global coverage information. Immediately following each trace point, i, we store **`true`** into slot i of both the local and global coverage arrays. To preserve ordering, the arrays are declared **`volatile`**.

Note that the instrumentation in Fig. 2.5 shows some clear redundancy. This redundancy is especially prevalent in our statement coverage example; for example, the execution of the trace point `cov[4]` implies the execution of the trace point `cov[2]`. Prior work optimizes the placement of binarized coverage probes [3, 4, 105, 169, 185]. However, none of this prior work directly applies to our instrumentation, where we must (1) ensure correct coverage data in the presence of failing executions, and (2) constrain our optimization techniques based on where we *desire* coverage data in contrast to where we want to *allow* instrumentation. We develop a class of novel optimization approaches to address these shortcomings in Chapter 4.

For each trace point, we record a small amount of static metadata used to identify the trace point during analysis. In practice, our instrumentor records slightly different data depending on the type of trace point used (based on the requirements of our downstream analyses; see Chapter 5). Function coverage need only record the name (or mangled name) of the function. Call-site coverage records: (1) the name of the called function, if known; and (2) the line number of the call site. Statement coverage records the sequence of line numbers occurring in the basic block, and, for reasons discussed further in Chapter 5, Section 5.3.1.2, any calls that occur within the basic block. The linker aggregates this metadata into a single record for the entire executable: coverage info in Fig. 2.1.

2.2.3 Tracing Customization

Note that some of the above techniques can be combined (i.e., we can perform multiple types of tracing within a single function). Furthermore, recall that one of our primary goals was that our instrumentation should be configurable (for overhead or to change focus) without re-compilation or re-deployment. To this end, we distinguish between:

- *program instrumentation*: the static modification of a program's code at compilation time, and
- *program tracing*: the dynamic tracing code that is actually executable during a particular run of the program.

Throughout the remainder of the dissertation, we will refer to a (possibly empty or singleton) set of tracing mechanisms as a tracing *scheme*. For example, the following are all tracing schemes: (1) path tracing alone; (2) path tracing paired with statement coverage; and (3) the combination of path tracing, call-site coverage, and function coverage. Note that program instrumentation determines the set of available tracing schemes that could possibly be available

Table 2.1: Evaluated applications, ordered by size.

Application	Description	Versions	Mean LOC
tcas	Siemens	1	173
schedule2	Siemens	1	373
schedule	Siemens	1	413
replace	Siemens	1	563
tot_info	Siemens	1	564
print_tokens2	Siemens	1	568
print_tokens	Siemens	1	727
ccrypt	Linux utility	1	5,280
gzip	Linux utility	5	8,114
space	ADL interpreter	1	9,563
sed	Linux utility	7	14,314
flex	Linux utility	5	14,946
grep	Linux utility	5	15,460
gcc	C compiler	1	222,196

at run time, while only one tracing scheme will actually be active at any given time during a program run. For the remainder of this chapter, we only consider instrumentation that allows exactly one tracing scheme per function (i.e., tracing cannot be customized at run time). In Chapter 3, we will lift this restriction, and discuss methods of handling tracing customization in the presence of many possible tracing schemes.

2.2.4 Additional Consideration: Thread Safety

Our experimental evaluation in Section 2.3 uses only single-threaded applications, but our instrumentation remains valid with threads. Path tracing only accesses stack-allocated variables, and each thread independently maintains its own path traces. Program coverage writes to globals, but never reads from globals. (We store each coverage bit as a full byte for atomicity.) Thus, even updates to the global coverage arrays have no malign race conditions.

2.3 Experimental Evaluation

To assess the efficiency of our data-collection strategies, we selected a range of subject applications varying in functionality and size. All applications are written in C. Table 2.1 gives additional details about our test subjects. The Siemens applications, flex, grep, gzip,

sed, and space were obtained from the Software-artifact Infrastructure Repository [55, 154]. ccrypt and gcc are real, released applications. These applications correspond with those used in later analysis evaluation (see Chapters 5 and 6), and are selected because they have seeded or real faults that can be enabled or disabled to create failure reports for evaluation. As stated before the start of this chapter, we ran experiments over the non-faulty builds of most applications; gcc used a build with a known fault.

Some of the applications had multiple versions as indicated by the "Versions" column of Table 2.1. Results presented in this section are aggregates across all versions and test suites of each application. In general, results vary little among builds of a given application; we explicitly note any exceptions. We also aggregate results for all applications from the Siemens test suite to simplify presentation. These are very small, simple applications, and results indicate that they have similar results for overheads with all forms of tracing. Again, we note exceptions in the following sections.

2.3.1 Run-Time Overhead

Overhead is the ratio of execution times for instrumented and uninstrumented code. For each version of each application, we ran the test suite over the non-faulty build at least three times and took the geometric mean of the overheads for each test case, and then took the geometric mean over all versions; thus, we avoid over-representing specific versions or long-running test cases. Results appear in Fig. 2.6. Smaller values are better, with 1.0 conveying no instrumentation overhead. We built each application version using our instrumentor, with all non-library functions instrumented with various instrumentation configurations.

We first evaluated each tracing mechanism individually. The first four bars (Function Coverage, Call-Site Coverage, Statement Coverage, and Path Tracing) show these results. Function coverage causes no measurable overhead for our test subjects. Call-site coverage is far cheaper than statement coverage (gathered as basic block coverage). The maximum overhead for call-site coverage among our test subjects was 2.0% (for gcc), while statement coverage has overheads as high as 25.8% (for sed). gcc has thirteen functions with more than 2^{63} acyclic paths; these cannot be instrumented for path tracing. Even so, the cost of full path tracing is surprisingly low, varying across applications from negligible to 10.4% (for gzip). This low cost suggests that our adaptation of the classic path profiling approach is very efficient, substantially reducing our overhead over full path profiling (which, according to initial results by Ball and Larus [22], has approximately 30% average run-time overhead due to large storage requirements and the use of hashing).

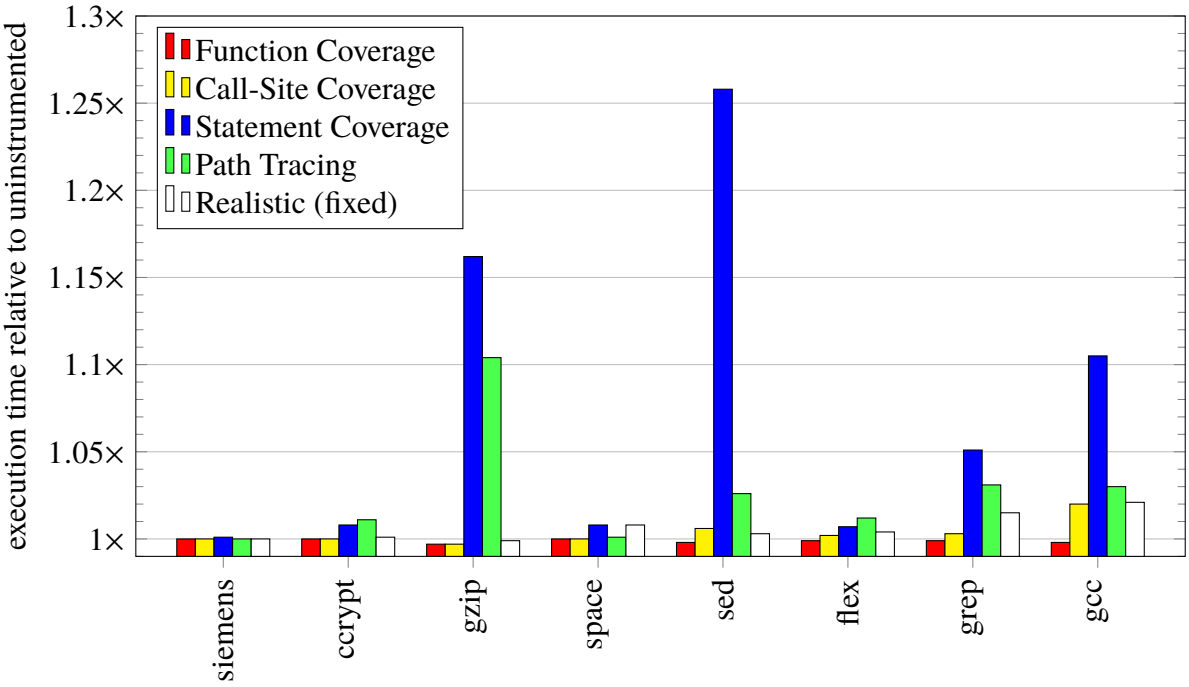


Figure 2.6: Run-time overhead

Taking into account measured overheads and expected orthogonality of benefits, we then considered instrumentation based on a realistic tracing scheme. At a high level, the goal is to combine call-site coverage with selective use of path tracing (where its higher cost but more dense information is likely to be most useful). Specifically, our goal was to simulate a developer activating latent path tracing instrumentation in response to observed post-deployment failures. We enabled all faults for each application, and observed each failing run from each application's test suite. We then instrumented each application with:

- call-site coverage tracing for all non-library functions, and
- path tracing for any function appearing in the crash stack of any failing test case for each application version.

With this configuration, each function is still only instrumented with one tracing scheme: either only call-site coverage, or call-site coverage plus path tracing. Again, this is a realistic configuration if latent instrumentation can be enabled post-deployment in response to observed failures. In this chapter, we assume that this customization is done via re-compilation. In Chapter 3, we will discuss approaches to post-deployment tracing customization.

Results for this customized scheme are shown as "Realistic (fixed)" in Fig. 2.6. The use of the "(fixed)" qualifier is to highlight that this choice of tracing is fixed at compile time, rather than being truly customizable post-deployment (again, we will rectify this problem in Chapter 3). These results indicate that limiting path tracing to functions involved in failures can significantly reduce overhead (especially for gzip and sed). The overhead of a particular application appears to depend on non-trivial factors. For example, larger applications do not necessarily have more overhead. Most applications have comparable overheads for all versions with realistic instrumentation. Both gzip and flex have one version with significantly higher overhead (about 1.5% on average in both cases), while the other versions are negligible. Overheads between sed versions also vary somewhat, ranging from negligible to 2.0%. Averaged across all larger (non-Siemens) applications, the realistic configuration shows a mere 0.5% overhead.

All of the preceding results used non-optimized builds, as this is most conducive to debugging. We also gathered results (not shown in Fig. 2.6) using each of our previous instrumentation schemes but with Clang "-O3" optimization enabled. Our analyses (from Part II of this dissertation) still work correctly on optimized code, due in part to our use of `volatile` declarations as discussed in Section 2.2. Results for optimized code are similar to unoptimized results, and suggest that our instrumentation does not seriously hinder program optimization. For the "Realistic (fixed)" configuration with optimization, average overhead increases to just 1.1% across all larger applications, with a maximum overhead of 3.0% (for gzip). However, debugging optimized code is always tricky. For example, statement reordering can make the execution paths we recover difficult to understand. Prior work on debugging optimized code [75, 168] is directly applicable here.

2.3.2 Memory Overhead

We next measured the ratio of the maximum resident memory size of the running program for instrumented and uninstrumented code. Again, for each version of each application, we ran the test suite over the non-faulty build at least three times and took the geometric mean of the overheads for each test case. Results appear in Fig. 2.7. Smaller values are better, with 1.0 conveying no instrumentation overhead.

Again, the first four bars indicate memory overhead for each tracing mechanism individually. Our results indicate that function coverage and call-site coverage have very small memory footprints. For statement coverage and path tracing, however, extra memory usage is somewhat larger; for sed, overheads reach 8.4% for path tracing and 6.7% for statement coverage.

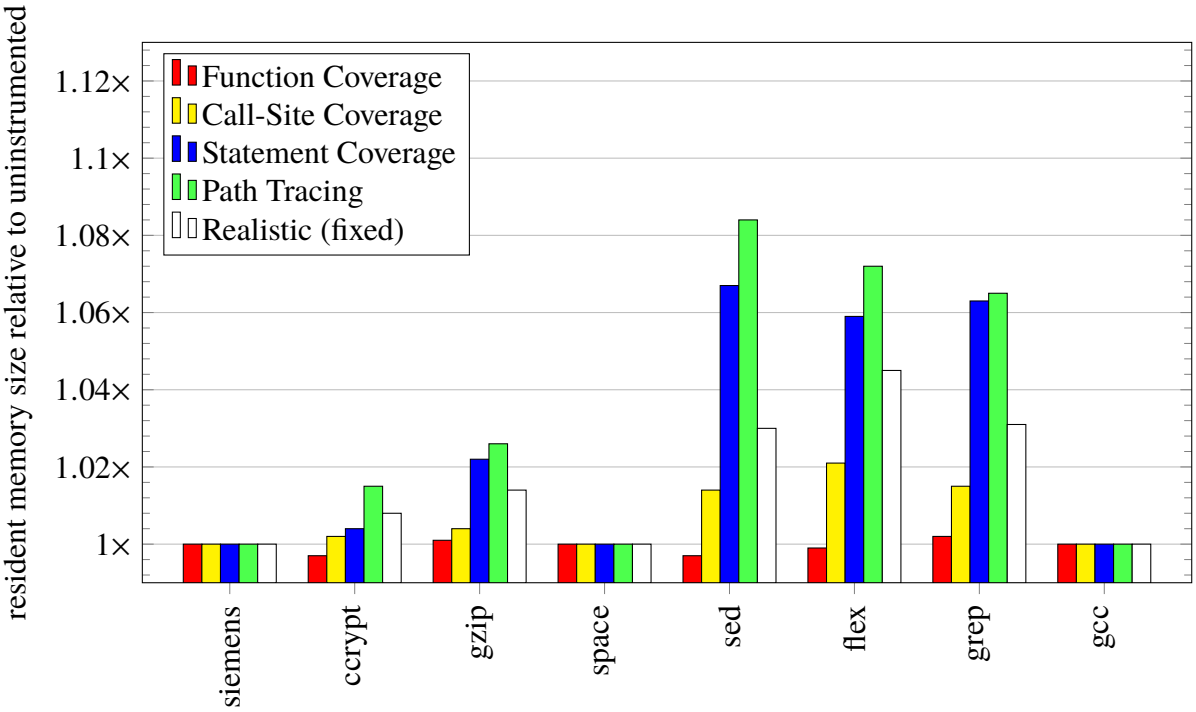


Figure 2.7: Memory overhead

The final bar ("Realistic (fixed)") again corresponds to the scheme proposed in Section 2.3.1: tracing of call-site coverage and/or path tracing, with coverage enabled everywhere and path tracing enabled in functions appearing in failing stack traces. Again, we see that specializing tracing to observed failures can significantly reduce overhead. In the most extreme case (sed), the memory overheads for call-site coverage and path tracing are 1.4% and 8.4% respectively (totaling 9.8%), but the realistic configuration causes only 3.0% overhead. In fact, the realistic configuration shows just 1.8% memory overhead averaged across all larger (non-Siemens) applications.

As with time overhead, we also gathered memory overhead numbers for our tracing configurations with standard "-O3" compiler optimizations enabled. For the most part, the results are again similar to their unoptimized counterparts. Overhead numbers are slightly higher in optimized code. The differences are most pronounced for coverage mechanisms; statement coverage sees its maximum overhead value (for flex) increase from 5.9% to 12.4%. The realistic scheme similarly sees its maximum overhead (again, for flex) increase from 4.5% to 7.6%. Nevertheless, the average overhead for this realistic scheme with compiler optimizations (for the larger applications) is only 2.5%. Thus, our tracing has a very small memory footprint, even for optimized builds.

2.3.3 Discussion

Our results indicate that our path tracing and program coverage approaches have generally small impacts on both the time and memory usage of our test subject applications. Our program coverage results suggest that there may be substantial trade-offs regarding coverage density in our domain. While function coverage has unmeasurably small overhead, it also provides significantly less detail about the program's execution. On the other hand, full statement coverage comes at a high overhead cost, but provides precise detail where its cost can be tolerated. Call-site coverage provides an interesting compromise: its cost is often similar to that of function coverage, but it provides complementary data to dense stack-local path traces. In Part II of this dissertation—particularly in Chapter 5—we will further assess these trade-offs regarding trace data utility for postmortem analysis based on core dumps.

We also examined a particularly interesting combination of path tracing and call-site coverage tracing in the "Realistic (fixed)" tracing scheme. However, note that, while this scheme suggests great benefits from customizing tracing to previously-observed failures, we have thus far required program re-compilation to achieve these benefits. We will revisit this realistic scheme in Chapter 3, where we present new methods that allow developers or end-users to customize tracing schemes post-deployment, without re-compilation.

Overall, our goal was to introduce customizable tracing mechanisms with very little run-time cost that are nevertheless very useful for postmortem analysis. We have succeeded, thus far, in the first part of our goal via targeted core dump enhancement. Our realistic tracing scheme introduces very low overheads (0.5% time and 1.8% memory across the larger benchmarks) suitable for deployed applications. The remaining chapters in this dissertation introduce customizable tracing approaches, improve upon our tracing techniques, and present postmortem analyses that take advantage of our tracing mechanisms and other partial execution information available from post-deployment failure reports.

2.4 Threats to Validity

We attempted to gather fair and generalizable results, but have not formally proven the correctness of our approaches or implementation. Here we discuss threats to the validity of our results, and measures taken to mitigate these risks.

2.4.1 Threats to Internal Validity

One of the central claims of this dissertation is that nearly all deployed software will have bugs throughout its lifetime. Our software is no exception. Bugs in our instrumentation design or implementation could impact the accuracy of overhead results.

We made significant attempts to control other factors (outside of our tracing mechanisms) that could have impacted overhead results. We used a single machine and ran our tests under minimal load. Nevertheless, our instrumentation does impact code size, stack frame size, and (in rare cases where we need to place instrumentation along edges) the control-flow graph of the programs. Thus, it is possible that other factors not directly related to instrumentation may impact our results.

2.4.2 Threats to External Validity

While we attempted to select applications with a wide variety of size and functionality, it is obviously impossible for us to test our approach on all possible programs. Thus, our results may not generalize to all deployed software. In particular, we evaluate only applications written in the C programming language in this work, so the applicability of our approaches to modern object-oriented languages cannot be guaranteed.

Our "Realistic (fixed)" scheme assumes that developers are able to make changes to program instrumentation based on observing previous failures. Where such failure reports are obfuscated or difficult to obtain, developers may struggle to select appropriate functions for instrumentation/tracing. This *realistic* scheme also assumes that developers are able to readily re-deploy software after compiling with new tracing schemes; we consider this lack of customizability a significant concern in many real-world scenarios, and seek to improve on this limitation in Chapter 3.

2.5 Related Work

Recent work extends the original efficient path profiling work by Ball and Larus [22]. Apiwattanapong and Harrold [15] and Vaswani et al. [173] reduce the overhead of path profiling by specializing profiles to a user-specified subset of each function's paths. While we share the desire to reduce the cost of profiling, applying these techniques to path *tracing* would result in failure reports containing traces with "holes" for untraced paths. Other work extends profiled paths by adding calling-context sensitivity [13] or by connecting paths

interprocedurally [115, 164] and across loop back edges [51, 104, 156, 164]. These techniques increase the cost of profiling to gather more detailed profile data. Because we use fixed-length path buffers in this work, and maintain our data within the active program stack, extending paths across back edges would allow us to trace longer paths, while gathering interprocedural paths may allow us to maintain some partial paths from returned stack frames.

Our program coverage instrumentation techniques from Section 2.2.2 are quite basic in that they gather redundant information. Much prior work optimizes the placement of binarized coverage probes [3, 4, 105, 169, 185] or lowers the cost of gathering coverage by dynamic insertion and deletion of probes [38, 85, 117, 118, 141, 169]. Because (1) we are working with applications that are assumed to fail and (2) we specialize the program points for which we require accurate coverage data and/or those that we are allowed to instrument, no existing techniques will apply directly to our mechanisms. In Chapter 4, we discuss new optimization strategies that generalize many of these prior techniques. Many other prior approaches complement our coverage mechanisms; we describe these relationships in detail in Chapter 4.

3 EFFICIENT RUN-TIME CUSTOMIZATION

Substantial portions of this chapter are derived from a 2016 journal paper by Ohmann and Liblit [137].

Any tracing performed for deployed applications must be efficient in both time and memory usage. The data gathered from this tracing should be targeted and a valuable addition to any generated failure reports. However, the precise *amount* of overhead that is acceptable for a particular application will vary considerably. For example, while Cornejo et al. [41] find that users of interactive applications can tolerate running-time overheads as high as 30%, this cost is far too high for time-sensitive computation [61, 141]. Especially in production code, it can be difficult to specify instrumentation overhead requirements beforehand, as these requirements may change over time, or vary for each program instance. Furthermore, while focusing on failure-related code could substantially reduce tracing cost, it is impossible to predict where or when post-deployment failures will occur before release. This variety in overhead requirements for different deployed applications, coupled with the corresponding variety in desired failure report detail, necessitates *customizable* run-time tracing after deployment.

Recall from Chapter 2 that we distinguish between static instrumentation and dynamic run-time tracing (which is, ideally, customizable). There, we introduced the notion of a "scheme" which indicates a possible tracing configuration, based on the statically-available options from instrumentation.

In this chapter, we build upon this idea and evaluate effective means of customizing our tracing mechanisms (from Chapter 2) post-deployment, without re-compilation or re-deployment of the application. We begin by providing a more thorough characterization of tracing schemes. Then, we propose two methods of instrumentation that facilitate customization of tracing schemes via unintrusive binary modification. In both cases, for N possible tracing schemes, we begin by creating N replicas of each instrumented function. Then, the first customization method we consider uses a straightforward **`switch`**-based "springboard" function to dynamically select which replica to call at each instantiation. The second method uses the recently-introduced GNU indirect function attribute to resolve replica selection for each function at program load time.

Through our evaluation of these two customization approaches, we find that even the basic **`switch`**-based style has only a small impact on running time and dynamic memory usage. In particular, a customizable version of our "Realistic" scheme from Chapter 2 still displays just 0%–5% execution time overhead (plus a modest increase in static memory usage) with

customization enabled; this overhead is low enough for most production use. Surprisingly, our more advanced customization method based on load-time indirect function calls has very similar performance to the basic method, suggesting that either would be an efficient choice for post-deployment tracing customization.

3.1 Tracing Schemes

As originally introduced in Section 2.2.3, a set of tracing mechanisms is a tracing scheme, formally defined as follows.

Definition 3.1. *For a set of tracing mechanisms, T, any $s \subseteq T$ is a tracing scheme over T.*

The number of possible tracing schemes grows exponentially with the number of possible tracing mechanisms. Specifically, there are $2^{|T|}$ possible tracing schemes for any set of tracing mechanisms (precisely the power set of T, $\mathcal{P}(T)$). For tracing customization, we are interested in a set of tracing schemes that are available based on instrumentation we perform with `csi-cc`. Such a set of tracing schemes could be any subset of $\mathcal{P}(T)$, or, put another way, could be any element of $\mathcal{P}(\mathcal{P}(T))$. However, we can customize even further. For a given program, P, one need not be constrained to the same set of tracing schemes for each function.

Definition 3.2. *Given:*

- *T, a set of tracing mechanisms, and*
- *F, the set of procedures contained in program P;*

any function $S : F \to \mathcal{P}(\mathcal{P}(T))$ is a tracing configuration over T and P.

Put simply, a tracing configuration is a mapping from each procedure (or function) in a program to the set of tracing schemes that are possible to select for run-time tracing (via post-deployment configuration).

We defined four tracing mechanisms in Section 2.2.3: path tracing (PT), function coverage (FC), statement coverage (SC), and call-site coverage (CC). That is, $T = \{PT, FC, SC, CC\}$. Thus, there are a total of sixteen possible tracing schemes for these mechanisms. Note, however, that some of these schemes will trace redundant information; for example, statement coverage subsumes both call-site coverage and function coverage. Nevertheless, the number of possible schemes can quickly become unwieldy: there are up to ten possibilities for our

mechanisms proposed in Chapter 2, depending on the function being instrumented. This "All Options" tracing configuration is defined as follows. First, for a given function, f:

- $hasCall_f$ is *true* iff f contains at least one call statement,
- $alwaysCalls_f$ is *true* iff f contains at least one call statement that must execute on all paths from f's entry to exit, and
- $multiPath_f$ is *true* iff f contains at least one branch.

Then, the "All Options" tracing configuration, S, is defined over each $f \in F$ as:

$$
\begin{aligned}
pathOptions_f = \quad & \{\{\}, \{PT\}\} \text{ if } multiPath_f, \text{ else } \{\{\}\} \\
coverageOptions_f = \quad & \{\{\}, \{FC\}, \{SC\}\} \cup \\
& \{\{CC\}\} \text{ if } hasCall_f, \text{ else } \emptyset \cup \\
& \{\{FC, CC\}\} \text{ if } hasCall_f \text{ and not } alwaysCalls_f, \text{ else } \emptyset \\
S(f) = \quad & pathOptions_f \times coverageOptions_f
\end{aligned}
$$

where the "×" operation performs the pair-wise union of schemes. In brief, the above states that f's tracing options are each of the following, possibly paired with path tracing: no coverage, function coverage, statement coverage, and (for some functions) call-site coverage and function coverage + call-site coverage. The auxiliary variable *pathOptions* determines whether path tracing should be used in any schemes: since path traces are stored within the program stack (per Section 2.2), they provide no benefit for functions with only one acyclic path. The auxiliary variable *coverageOptions* determines which coverage mechanisms should be included in possible schemes. Function coverage and statement coverage are always tracing options, while call-site coverage is only useful to include if f contains at least one call to be instrumented. The pairing of function coverage and call-site coverage is useful if it is possible to execute f without calling another function at some point during f's execution (in this case only, call-site coverage does not subsume the information from function coverage).

In Chapter 2, we introduced the "Realistic (fixed)" tracing scheme, but lamented that we needed to recompile to enable or disable path tracing and/or call-site coverage for instrumented functions. The following is a *customizable* variant, "Realistic":

$$\{\{\}, \{CC\}, \{CC, PT\}\}$$

With this tracing configuration, we could trace nothing, call-site coverage, or call-site coverage plus path tracing.

3.2 Tracing Customization

There are many ways to implement post-deployment customization based on tracing schemes as defined in Section 3.1. Our approach uses further instrumentation of the program's code. We statically replicate each function, instrument each replica with one possible tracing scheme, and dynamically decide which replica to execute. We consider two alternatives for dynamically selecting replicas: a **`switch`**-based technique that selects the appropriate replica at each call, and a technique that resolves replicas at program load time via indirect function link tables. Figure 3.1 shows an example instrumented via these alternatives, which are discussed in detail in the following sections. Note that our instrumentation is over LLVM bitcode, so Fig. 3.1 actually shows a source-level representation of our transformations. For this example, our tracing configuration, S, is equal to the "Realistic" configuration from Section 3.1:

$$\{\{\}, \{CC\}, \{CC, PT\}\}$$

In both methods, for each function, a global variable, `add_actionInst`, determines which function variant to use on that particular run. Here there are three possibilities: no tracing, call-site coverage tracing, or call-site coverage plus path tracing. These variables are stored in a special section of the data segment where they can easily be changed by direct editing of the program binary. For example, applications can initially ship with all instrumentation turned off. Over time, instrumentation can be activated for selected functions based on previously-observed failures.

3.2.1 Springboard Function

For our first customization method, we first replicate each function, f, into multiple functions: one for each scheme in $S(f)$ (i.e., one for each of f's possible tracing schemes). Each replica is instrumented via the mechanisms from Chapter 2 per its corresponding scheme. The original body of f is changed to a "springboard" that calls the correct variant. We rely on compiler optimizations to make appropriate choices regarding inlining or conversion to jump tables.

Figure 3.1a shows an example for the function from the running example from Chapter 2, Fig. 2.2. Note that this first approach introduces an extra indirection into each function

```
enum {
  INST_NONE,
  INST_CC,
  INST_CC_PT
} add_actionInst;

void add_action(char *new_text) {
  switch (add_actionInst) {
    default:
      add_action_NONE(new_text);
      return;
    case INST_CC:
      add_action_CC(new_text);
      return;
    case INST_CC_PT:
      add_action_CC_PT(new_text);
      return;
  }
}
```

(a) Switch springboard

```
enum {
  INST_NONE,
  INST_CC,
  INST_CC_PT
} add_actionInst;

void (*resolver_add_action ()) () {
  switch (add_actionInst) {
    default:
      return add_action_NONE;
    case INST_CC:
      return add_action_CC;
    case INST_CC_PT:
      return add_action_CC_PT;
  }
}

void add_action(char *new_text)
__attribute__
((ifunc ("resolver_add_action")));
```

(b) GNU indirect function calls

Figure 3.1: Tracing customization example

call. However, all calls that we introduce are *tail calls*, which should theoretically make them excellent candidates for compiler optimizations (e.g., tail-call recursion optimizations, inlining, or conversion to function-pointer jump tables). In our evaluation, we saw these optimization behaviors frequently; Section 3.3 describes overall performance.

3.2.2 GNU Indirect Functions

Our second customization method uses the `ifunc` (indirect function) attribute, which is supported by modern C/C++ compilers for ELF binaries. Like our first method, we first replicate a function, f, and then appropriately instrument each replica. However, there we changed the original f into a springboard function that examined a global variable at each executed *call* to determine f's chosen tracing scheme. In our second customization method, we instead introduce a new "resolver" function that reads f's global switcher variable just *once* at program load time, and then stores a pointer to the appropriate variant in the program's Procedure Linkage Table (PLT). (The PLT is normally used to lazily bind functions from dynamically-linked libraries to their absolute addresses after loading.) The original body of f

is then completely removed, and calls to f are resolved through the PLT, as if f were loaded from a dynamic library.

To obtain this functionality, we indicate our resolver to the compiler via the attribute `ifunc("resolver_foo")`. This attribute, attached to function `foo`, says that `resolver_foo` is a function that returns a pointer to the appropriate variant of `foo` (resolved whenever `foo` is first loaded into memory). Precisely as suggested, calls to `foo` then go through the PLT; that is, `foo` is treated as external, as if it were loaded through a dynamically-linked library. The indirect function attribute was originally designed for scenario-based optimization based on the architecture on which an application is running [114]. For example, certain library routines (e.g., `memcpy`) might use the resolver function to check for certain architectural features, such as block or cache sizes, to select the most efficient implementation to run. In our case, the resolver function is quite different: we examine a *user-space* global switcher variable, rather than making system calls for architectural properties.

Figure 3.1b shows appropriate instrumentation for our running example. Note the structural similarity to the springboard approach from Section 3.2.1. The primary advantage of this approach, as noted in the above discussion, is that the `resolver_add_action` function is called only once, whereas, optimizations aside, the **`switch`** from Fig. 3.1a is checked on every call to `add_action`. However, there are possible disadvantages to this approach as well. First, the basic springboard approach is designed to be optimization-friendly, transforming direct function calls via basic indirection; in contrast, the `ifunc` approach ensures that direct calls will always resolve as calls through function pointers. Second, the use of indirect function resolution means that one cannot modify tracing scheme selection after the first call to a function (i.e., dynamically change tracing *during* a program run); we do not make use of this ability in our evaluation, but burst tracers [25, 70] would suffer from this limitation.

3.3 Experimental Evaluation

We conducted experiments to assess the effect of adding customization atop the tracing mechanisms from Chapter 2. We also investigated the relative strengths of the two customization instrumentation strategies from Section 3.2. The bulk of our experiments essentially mirror those from Chapter 2: we assess the time and memory overheads of our instrumentation, use the same set of test subjects, run experiments over non-faulty builds of applications, and aggregate results across all versions and test suites of each application.

Table 3.1: Customization microbenchmark results. All times are in seconds.

	Running Time			
	O0		O3	
Call Style	springboard	ifunc	springboard	ifunc
direct	3.43	3.26	3.14	3.23
function pointer	3.46	3.30	3.14	3.24

3.3.1 Comparing Customization Mechanisms

We began by comparing the performance of our two customization options on a very small microbenchmark program. The program simply accepts one integer as input from the command line, and then calls the function:

```
int sum(int x, int y) {
  return x + y;
}
```

in a loop based on the input integer. The goal of this benchmark was to spend as much time as possible in dispatch of customized call sites. We compiled the microbenchmark with `csi-cc` using a modified version of the "All Options" tracing configuration from Section 3.1 where we always create all 10 variants of each instrumented function (regardless of redundant tracing). We gathered the running time for the springboard and indirect function approaches over 100,000,000 loop iterations; results scaled linearly for different numbers of iterations, so we report only times for this single iteration count.

Table 3.1 shows our results. We began with our program making a direct call to the `sum` function (the "direct" row in Table 3.1). With no compiler optimization, the indirect function approach is a modest 5% faster on our example. However, real deployed code generally uses optimization, so we also measured running times with Clang's optimization level "-O3." Here, we found that the basic springboard approach is actually *more efficient* than the indirect function call approach. When we examined the compiled binary, we found that our predictions from Section 3.2 were indeed correct: the springboard function is aggressively inlined, and standard tail-call optimizations are performed where stack frame sizes allow. The indirect function approach, in contrast, benefits very little from compiler optimization, and the small time reduction in Table 3.1 appears to come primarily from optimization of the `sum` function.

Next, we wanted to see if results differed for calls that were already indirect prior to our customization. The "function pointer" row in Table 3.1 shows results for runs where

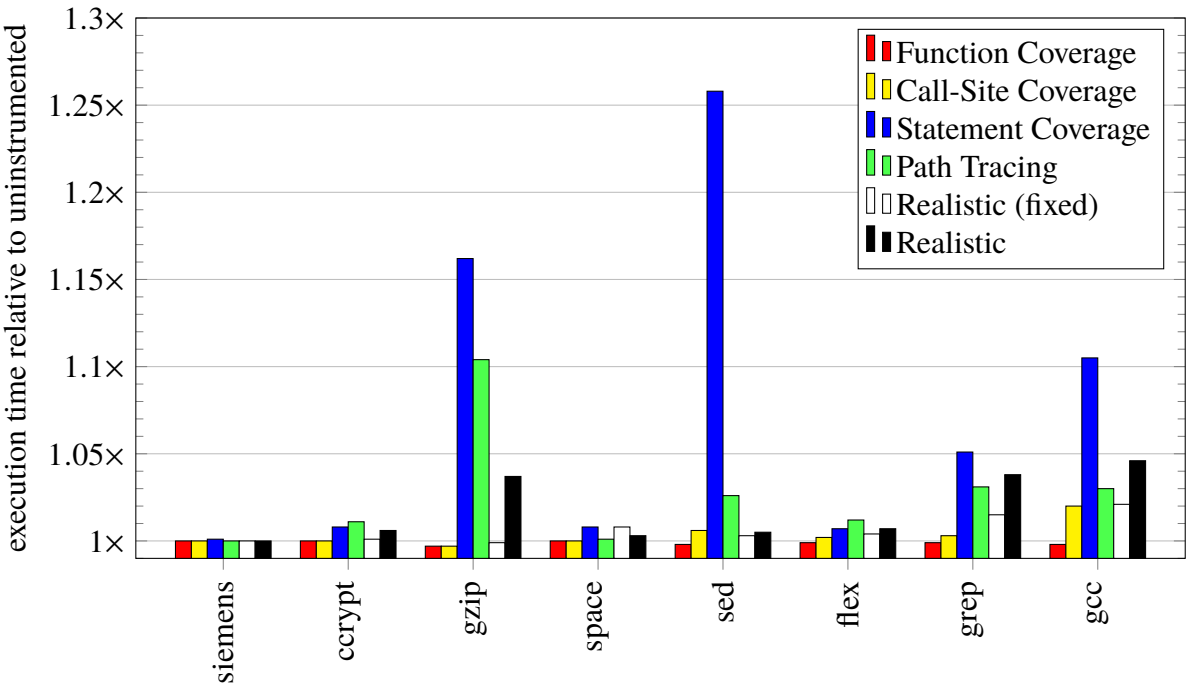


Figure 3.2: Run-time overhead

we changed the direct `sum` call into a call through a function pointer whose value was not visible to the compiler for optimization (i.e., it actually compiles down to a call through a function pointer). We hypothesized that this would give the indirect function call approach an advantage, because completely inlining the called function was no longer possible. However, we again found our results nearly unchanged (aside from a small increase in absolute running times). The indirect function approach is again slightly more efficient without optimization, but, overall, neither approach seems strongly preferred. Calls to replica functions for the springboard approach are again largely inlined, leading to a large speedup when moving to higher levels of compiler optimization.

3.3.2 Run-Time Overhead

We next measured the overhead of introducing customization into our already existing mechanisms for deployed applications. Using the set of benchmark programs from Chapter 2, we built each application with our instrumentor using the new "Realistic" tracing configuration, as defined in Section 3.1. Recall that this configuration allows each function to select run-time tracing from: no tracing, call-site coverage, and call-site coverage plus path tracing.

As in the "Realistic (fixed)" instrumentation from our previous evaluation, we then configured the generated binaries to activate call-site coverage tracing for all non-library functions, as well as path tracing for any function appearing in the crash stack of any failing test case for each application version. However, note that, in this case, we were able to perform this customization without re-compilation of the applications. Hence, we more accurately simulate true customization of tracing via latent instrumentation that is enabled post-deployment in response to observed failures.

Figure 3.2 plots our run-time overhead results from Chapter 2 with an additional "Realistic" bar for this new configuration. Plotted results use the springboard approach to customization, but, as we discuss further in Section 3.3.4, we found no measurable difference in results using the indirect function approach. As in previous experiments, results are largely consistent across application versions. One version of gzip has significantly lower overhead (about 1% on average), while the other versions are around 5%. Overheads between sed versions again vary somewhat, ranging from negligible to 2.5%. Averaged across all larger (non-Siemens) applications, the realistic configuration has only 2.0% overhead, with a maximum overhead of 4.6% for gcc.

Note the difference between the "Realistic" and "Realistic (fixed)" bars in Fig. 3.2. This difference corresponds to the portion of overhead for the realistic configuration that was due to tracing customization. Clearly, customization adds a substantial portion of the run-time overhead for some applications (especially gzip). Overall, though, the "Realistic" configuration has extremely low run-time overhead (< 5% in all cases), and would be suitable for deployed applications.

As in our previous evaluation, plotted results used non-optimized builds. However, as stated previously, real deployed applications are almost universally built with aggressive compiler optimization. We also gathered results for our new "Realistic" configuration with Clang's "-O3" optimizations enabled. Again, results are very similar. In fact, overhead *decreases* to an average of just 1.6% across all larger applications, with a maximum overhead of 4.5% (for gzip). While debugging optimized code can be challenging, it appears that our customization (in conjunction with our tracing options from Chapter 2) does not seriously hinder unrelated compiler optimization.

3.3.3 Memory Overhead

We next measured the effect of our customization instrumentation on the maximum resident memory size of our subject applications. As before, we aggregated ratios for instrumented

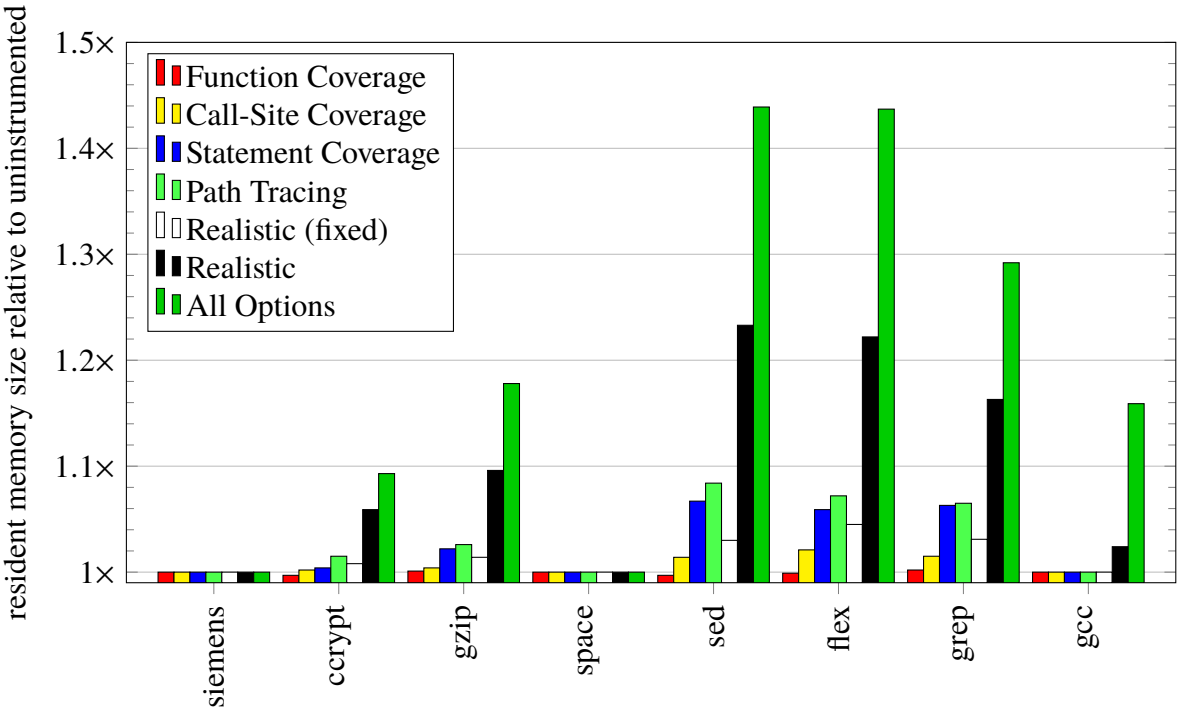


Figure 3.3: Memory overhead

and uninstrumented code across application test suites and versions for the non-faulty build of each application.

Figure 3.3 plots our results, again including a new bar for the "Realistic" configuration with call-site coverage and path tracing enabled as for our run-time evaluation. Here, when comparing with the "Realistic (fixed)" bar, we see that tracing customization appears to take a large toll on static memory usage. Exploring this further, the final bar, "All Options," shows memory overhead for the pathological tracing configuration from Section 3.1 where we instrument for up to 10 tracing possibilities for each function. In our experiment, we then actually enabled no tracing at execution time (i.e., we select the "{}" scheme for all functions). Perhaps as one might expect, the memory cost of customization is quite high. Because we create a new copy of every instrumented version of each function, we can potentially cause an exponential blow-up in code size. For the larger applications, the results of "All Options" instrumentation indicate that this is a potential issue. This scheme does not enable any tracing at run time, so the observed memory overheads are pure code bloat.

Nevertheless, it is important to keep these results in perspective. First, our instrumentation makes rather naïve choices; in a real-world scenario, it may be possible to make more informed decisions about which functions are likely to ever require customization or tracing in the

future. Second, and most importantly, the increased memory usage for customization is a one-time cost: it does not scale throughout program execution. Thus, the *dynamic* memory cost of tracing is more closely related to the "Realistic (fixed)" scheme, which shows just 1.8% average memory overhead for the non-Siemens applications.

We also gathered results with the indirect function customization mechanism and with "-O3" compiler optimization. For the indirect function version of customization, we again are unable to measure any significant differences from the basic springboard approach. With optimization, results are again quite similar to their unoptimized counterparts. The "Realistic" configuration sees only very minimal differences, with a decrease in average static memory overhead from 11.4% to 10.0% for the larger applications. The "All Options" configuration sees a much larger reduction for the non-Siemens applications: average overhead decreases from 22.9% to 16.5%, while maximum overhead (for sed) decreases from 43.9% to 32.5%. Thus, we consider the memory footprint of our customization reasonable, especially with the vast majority of its cost in static consumption.

3.3.4 Discussion

Our results indicate that we can perform post-deployment customization of `csi-cc` tracing schemes efficiently. We further found that the customizable variant of our "Realistic" tracing configuration yields an efficient pairing that provides complementary information for postmortem program analysis (see Part II, in particular Chapter 5, for analysis evaluation).

Our indirect function variant of customization had very little impact on overall instrumentation performance. Per our microbenchmark results in Section 3.3.1, many low-level details regarding compiler optimizations have an impact in this domain. In practice, it seems that even the small overheads from our tracing mechanisms dominate the difference in performance between our two customization options. Overall, it appears that both of our approaches provide efficient customization of our lightweight tracing mechanisms.

Recall that our overall goal was to develop customizable tracing with very little run-time cost that nevertheless provides great benefit for postmortem analysis. In this chapter, we significantly improved the customization aspect of the first part of this goal via two efficient approaches to selecting tracing schemes post-deployment. Our customizable "Realistic" tracing scheme proved remarkably efficient, with only 2.0% run-time and 1.8% dynamic memory overhead across our larger benchmarks. Of course, where higher execution overheads can be tolerated, other tracing mechanisms (e.g., statement coverage) can provide additional

detail about failing executions. If execution constraints are tightened, simple function coverage may be a better choice. To this end, our approach is highly customizable.

3.4 Threats to Validity

The threats to validity for our results are similar to those for our parallel evaluation from Chapter 2, discussed in Section 2.4. The only major changes relate to threats to the internal validity of our findings. As previously, our customization instrumentation code may contain bugs; these could impact our overhead results. However, we again controlled for factors other than our instrumentation that may have impacted overhead results; most notably, we ran experiments on a single lightly-loaded machine. We also spot-checked that our customization correctly selected function variants, and that we were able to recover traced data during execution. We ran a small selection of our test cases under a debugger; all tests were successful. Our customization strategies often have a much more significant impact on the memory layout of a process: we examined and evaluated some of these effects in our microbenchmark experiments, and verified that the output of all program runs for our complete experiments matches that of the uninstrumented version.

3.5 Related Work

Prior work emphasizes adaptive post-deployment instrumentation with data collection from prior runs. For example, Yu et al. [189] use dynamic feedback to reduce tracing in a race detector, while Dwyer et al. [58] use adaptive tracing to reduce the cost of monitoring safety and correctness properties. Arumuga Nainar and Liblit [18] dynamically adapt probe placement for statistical fault localization after deployment, building on the observation that only a very small portion of a program is actually relevant for helping a developer zero in on a bug. These approaches share our motivation for customizing tracing to reduce post-deployment tracing overhead. However, they all target specific problems in post-deployment optimization of a specific tracing mechanism, whereas our customization focuses on low-cost selection of different tracing schemes in production code.

The GAMMA project [31, 139] automates the distribution of data-collection tasks among a user community, reducing the burden on each individual tracing instance. Our work develops low-cost tracing mechanisms, and allows for efficient selection of these mechanisms in individual instances post-deployment. Thus, the two approaches are likely complementary:

we focus on gathering very valuable information at very low cost, while GAMMA focus on how best to deploy information-gathering instances.

Jimborean et al. [77] develop the VMAD system, which allows dynamic toggling of previously-inserted instrumentation code by running the user's program in a virtual environment. Our approach has different priorities. Our purely instrumentation-based customization techniques sacrifice some on-the-fly configurability (e.g., switching between tracing blocks within a single function). However, our techniques appear to be more efficient (Jimborean et al. [77] measure run-time overheads as high as 183% and code-size increases up to 268%), and are in many ways less invasive (we do not require loading a separate virtual machine concurrently with instrumented applications).

The GNU indirect function facilities that we used for one of our customization strategies derive from prior research on scenario-based function multi-versioning [114]. This technique is a good fit for our requirements, as we too require multiple versions of instrumented functions, though for a very different purpose: rather than optimizing for a specific architecture, our function variants are instrumented with different tracing schemes.

4 OPTIMIZING CUSTOMIZED PROGRAM COVERAGE

Substantial portions of this chapter are derived from a 2016 conference paper by Ohmann et al. [134].

In the preceding chapters, we introduced novel tracing mechanisms—path tracing and various granularities of in-memory program coverage—and methods of post-deployment tracing customization. In this chapter, we focus specifically on improving the efficiency of our program coverage mechanisms, with an emphasis on call-site coverage. Our program coverage mechanisms, and more specifically coverage at call sites, provide significant postmortem analysis benefit despite modest tracing cost, as we will see beginning in Chapter 5. However, despite extensive research on optimization of program coverage instrumentation [3, 4, 21, 88, 89, 105, 169], no prior techniques apply directly to our coverage tracing.

Recall that our program coverage data is *binarized*, meaning that coverage data marks each program location as "covered" or "uncovered," rather than counting the number of occurrences of each point during an execution. Binarized program coverage is also used in other prior program analysis research, including research in postmortem program analysis [129] and fault localization [106, 157]. Other prior research gathers program coverage over a limited set of program points post-deployment, either to divide the monitoring task across multiple clients [139], or to only monitor newly-added code or code not covered during in-house testing [143]. All of these cases have in common the fact that they only require accurate coverage data for a limited subset of the statements and/or branches of a program. Furthermore, as we saw in Chapter 2, gathering full statement coverage is often too expensive for post-deployment monitoring. Sparse tracing also means sending less data back to developers in failure reports; as we will see in Chapter 6, large amounts of data in failure reports can slow down analysis time, and this slow-down is wasteful if traced data is redundant or unnecessary. Orthogonal to performance, some code may be *off-limits* for monitoring; for example, one might avoid placing instrumentation in security-sensitive code, tightly optimized code, or code with strict real-time requirements.

In this chapter, we introduce the customized coverage probing problem, which is to select a smaller set of coverage probes given instrumentation restrictions and a set of program points of interest. We first prove that solving this problem to full optimality is NP-hard. Then, we provide three approaches to the problem. Our first approach constructs a mixed-integer linear program (MILP) whose solution identifies the optimal probe set; however, solving this MILP to optimality requires prohibitive analysis time during instrumentation. We therefore offer

two approximation approaches. The first approximation is very inexpensive to compute, but provides no optimality guarantees; the second finds a locally optimal approximation.

While prior research optimizes instrumentation for program coverage, none applies directly to our *customized* coverage scenario. Agrawal [4] and Tikir and Hollingsworth [169] optimize probe placement for binarized statement coverage. Unfortunately, Agrawal's approach is not applicable for incomplete executions (such as those that terminate unexpectedly due to a fatal signal), and this dissertation specifically targets failing deployed applications. Low-cost coverage tracing is especially important in these deployed scenarios, as only small amounts of run-time overhead are tolerable. Furthermore, neither Agrawal nor Tikir and Hollingsworth optimize for constrained coverage requirements. Deployed software calls for *customized* coverage information: coverage at a subset of program locations, such as those not exercised by a test suite [143] or those containing features interesting for debugging (e.g., call sites as per our instrumentation from Chapter 2 and other work by Nishimatsu et al. [129]).

Our approaches in this chapter are the first to optimize based on customized coverage requirements. We find that even coarse approximations can statically eliminate much instrumentation. When naïve instrumentation imposes large overhead, our optimizations significantly reduce both compilation and run-time costs.

4.1 Customized Coverage

Our goal is to determine an optimal instrumentation plan to gather *customized, binarized* program coverage information. More concretely, we are given a single procedure's control-flow graph (CFG), G; some subset of the vertices in G for which the developer desires information, D; and another subset of the vertices in G defining legal observation points, I. The problem is to determine the cheapest set of *probes* to insert into locations from I such that, for any given path p through G, the set of probes encountered along p is sufficient to determine the set of vertices from D that were traversed at least once along p.

4.1.1 Input

The input to the problem is as follows:

- $G = (V, E)$, a directed graph with vertices V and edges E
- $e \in V$, a unique source (or entry) vertex with in-degree 0
- $I \subseteq V$, a subset of vertices that may be probed (instrumented)

- c_i, the cost of probing vertex $i \in I$, where $\forall i \in I, c_i > 0$
- $D \subseteq V$, a set of "desired" vertices that must be "covered"
- $X \subseteq V$, a set of possible termination points

4.1.2 Problem Definition

Definition 4.1. *For $v_1, v_2 \in V$, $v_1 \rightarrow v_2$ denotes the set of all paths from v_1 to v_2 in G. If $v_1 = v_2$, then the trivial path (crossing no edges) is included in this set.*

Definition 4.2. *$V(p)$ denotes the set of all vertices encountered along path p, including the start and end vertices of p. If p is the trivial path from vertex v (crossing no edges), then $V(p)$ is the singleton set $\{v\}$.*

Definition 4.3. *A set of vertices $S \subseteq I$ is called a **coverage set** of D iff:*

$$\forall x \in X \text{ and } \forall p_1, p_2 \in e \rightarrow x,$$
$$V(p_1) \cap S = V(p_2) \cap S \implies V(p_1) \cap D = V(p_2) \cap D$$

That is, two executions ending at the same termination point $x \in X$ that encounter the same S vertices must encounter the same D vertices as well. Contrapositively, paths that encounter different D vertices must be distinguishable by encountering different S vertices as well.

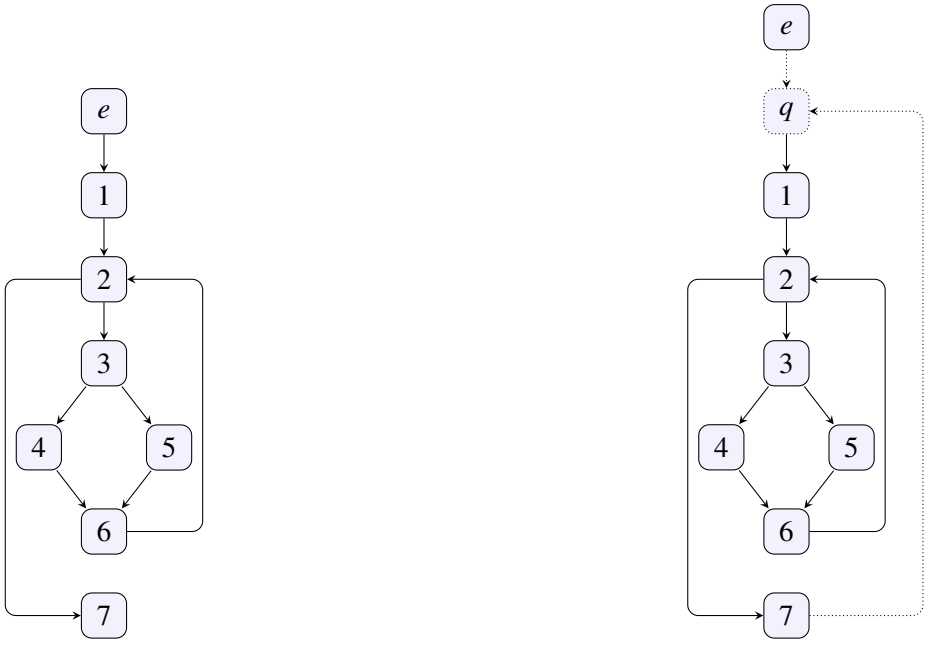
Problem Statement:

The **Customized Coverage Probing Problem** is to find a coverage set S of D such that $\sum_{s \in S} c_s$ is minimal.

4.1.3 Discussion

We desire the lowest-cost coverage set of D. Put another way, the goal is to find the cheapest set of probe vertices ($S \subseteq I$) such that the resulting observations ($V(p_1) \cap S$ and $V(p_2) \cap S$) are sufficient to derive desired coverage information ($V(p_1) \cap D$ and $V(p_2) \cap D$) for all executions, whether terminating normally or abnormally ($\forall x \in X$). If $D \subseteq I$, then a coverage set of D exists, since we can set $S = D$. However, the existence of some S that satisfies Definition 4.3 does not imply that $D \subseteq I$. That is, we may be able to infer coverage information for $d \in D$ without probing d directly.

The cost of probing each vertex (c_i) is input to the problem. To minimize the expected runtime cost of instrumentation, these values should represent the expected execution frequency of each $i \in I$, which could be derived from static heuristics [183] or profile data [21, 89].



(a) Original control-flow graph **(b) Multiple function executions**

Figure 4.1: Example control-flow graph with transformation for multiple function executions

However, other cost functions can minimize the static number of probes inserted, or prioritize instrumentation locations based on real-time requirements or security concerns. In our evaluation (see Section 4.3), we use LLVM's built-in analyses for statically approximating the execution frequency of each basic block.

4.1.4 Encoding Specific Problems

Figure 4.1a shows our running example control-flow graph (originally from Chapter 2, Fig. 2.2) augmented with entry vertex e. Using this graph as $G = (V, E)$, one can obtain optimized *local* statement coverage instrumentation with $I = D = V \setminus \{e\}$. Assuming the program could crash at any statement, we can also let $X = V$. This input corresponds to optimizing instrumentation for the stack-allocated statement coverage array `cov` from Fig. 2.5. This option is also valid for functions known to execute exactly once (e.g., the `main` function). To handle multiple executions of an instrumented function, one performs a simple graph modification: add a new vertex q representing any execution from the rest of the program, as shown in Fig. 4.1b. This change to our intraprocedural CFG indicates that the function may be called zero, one, or multiple times during the full program's execution (where all other functions are collapsed

into the summary node, q). For statement coverage, we then let $I = D = V \setminus \{e, q\}$. Usually, we say that $q \in X$, indicating that execution may terminate outside the current function. This input optimizes coverage instrumentation for global statement coverage; that is, for the statement coverage array `add_actionCov` from Fig. 2.5. If we can statically determine that the program cannot crash in the function (or, outside the context of crashing deployed applications, we only need coverage data for complete executions), we can set $X = \{q\}$. For edge coverage instead of vertex coverage, we split each edge in the CFG. Let V' be the set of new vertices added on these split edges. Full edge coverage is then encoded by $I = D = V'$.

In some contexts, one need only gather coverage information for a specific set of statements or edges. For example, consider our call-site coverage mechanism from Chapter 2. In this case, D is the set of basic blocks containing call statements. Other work gathers residual coverage information for deployed applications based on statements or branches not covered during in-house testing [139, 143]. Here, D is the set of non-covered blocks (or edges, using the edge-splitting method described above). Beyond well-studied cases, many others are possible. One might reduce run-time cost by removing some portion of the "hottest" blocks from I as identified by standard profiling. Based on previous results from postmortem analyses (such as those we describe in Part II of this dissertation), one might set D to program locations whose execution status was unknown in the failing execution. After a product release, developers could isolate new failures by setting D to recently changed code. Alternately, one might exclude security-sensitive code or code with real-time requirements from I.

4.1.5 Proof of NP-Hardness

To show that the Customized Coverage Probing Problem is NP-hard, we show that a known NP-complete problem reduces to our problem in polynomial time. Maheshwari [111] proves that determining the optimal placement for traversal markers in acyclic CFGs is NP-complete. Specifically, given an acyclic CFG, $G = (V, E)$, one can enumerate the set of source–sink paths through G, $P(G)$; then, for all $p \in P(G)$, define $E(p)$ as the set of edges along p. Maheshwari proves that determining the minimum-size set of edge traversal markers, $M \subseteq E$, such that each $p \in P(G)$ traverses a unique set of traversal markers, is an NP-complete problem. Formally, the problem is to find $M \subseteq E$ such that:

- for all distinct $p_1, p_2 \in P(G)$, $M \cap E(p_1) \neq M \cap E(p_2)$; and
- for all M' such that $|M'| < |M|$, there exist distinct $p_1, p_2 \in P(G)$ such that $M' \cap E(p_1) = M' \cap E(p_2)$.

Theorem 4.4. *The Customized Coverage Probing Problem is NP-hard.*

Proof. Note that traversal markers and coverage probes yield precisely the same information for acyclic graphs: no edge may occur more than once in any path (because there are no cycles), and the order of any pair of edges is fixed if both may occur along the same path (because there would otherwise be a path containing a cycle). Thus, while Maheshwari's original intent was to prove that minimal probing to distinguish all *paths* in an acyclic CFG is NP-complete, the same proof also shows that minimizing the number of edge probes to obtain *full edge coverage* information is NP-complete.

Transforming a Traversal Marker Placement Problem into a Customized Coverage Probing Problem is straightforward. Given a CFG $G = (V, E)$, we split each edge to instrument for edge coverage as in Section 4.1.4. Then, an optimal solution to the Customized Coverage Probing Problem with input

$$I = D = V' \setminus V \qquad\qquad c_i = 1 \text{ for all } i \in I$$

is also an optimal traversal marker placement. The transformation is polynomial, as splitting each edge is simply an $O(E)$ operation. Therefore, the Customized Coverage Probing Problem is at least as hard as optimal traversal marker placement; hence, the Customized Coverage Probing Problem is NP-hard. □

4.2 Optimization Approaches

We consider three approaches to solving the Customized Coverage Probing Problem. After re-characterizing our definition of a coverage set from Definition 4.3 into a checkable sufficiency condition, we present a MILP that identifies the optimal coverage set. However, since our evaluation indicates that obtaining a fully-optimal solution to our problem is intractable for real use, we also offer two approximation approaches. The first of these is based on CFG dominance relations and is remarkably efficient; however, this approximation provides no optimality guarantees. We then build on this first approach to formulate a second approximation that finds a locally minimal coverage set.

4.2.1 Checking Sufficiency of Coverage Sets

As a first step, we must devise a sufficiency check for a candidate coverage set $S \subseteq I$. That is, we would like a simplified condition (relative to Definition 4.3) to check whether any given set

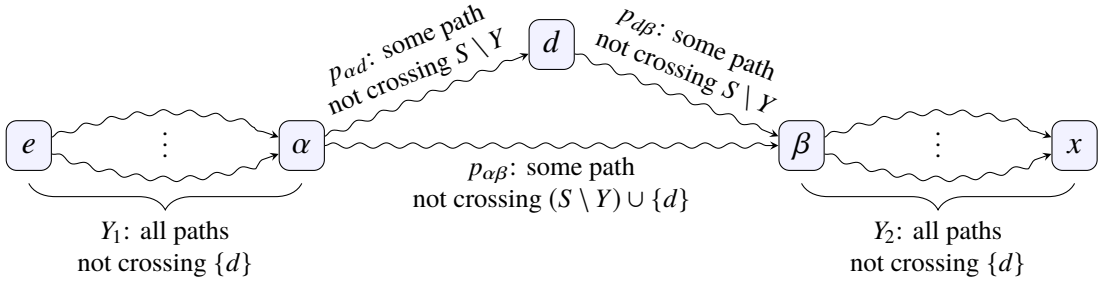


Figure 4.2: Pictorial representation of an ambiguous triangle

S is a coverage set of D. Recall that D is the set of desired vertices, and that S is a coverage set of D if and only if coverage information for S unambiguously allows one to determine coverage information for D on any execution path. Definition 4.3 is given with respect to all paths through G ending at some $x \in X$; for any graph with loops, there may be infinitely many such paths. Intuitively, however, many of these paths are redundant with respect to S's coverage set status. We formalize this intuition by narrowing our search to finding "ambiguous triangles" in which some $d \in D$ may or may not occur between observation vertices α and β. The condition is represented pictorially in Fig. 4.2. As input, we have all items from Section 4.1.1 and some $S \subseteq I$, a candidate coverage set for D.

In Fig. 4.2, wavy lines represent paths crossing 0 or more edges. The core of this approach lies in finding an ambiguous triangle composed of paths $p_{\alpha d}$, $p_{\alpha\beta}$, and $p_{d\beta}$. Such a triangle represents an ambiguous region of execution (between observation points) that demonstrates that S alone is not sufficient to determine if d executed. Conversely, S is a coverage set of D if S allows no ambiguous triangles in G. An ambiguous triangle is a triple (α, β, d) such that:

- α is either the entry vertex or an observation point from S,
- β is either from S or a possible termination location, and
- $d \in D$, and d may or may not occur on paths $\alpha \to \beta$ that contain no other new observations.

To define *new observations* above, we must define sets Y_1, Y_2, and Y. The set Y_1 contains all d-free paths from the entry vertex to α. The set Y_2 contains all d-free paths from β to a possible termination location. (If $d \in X$, then d is excluded as a possible final location, since ending at d implies the observation, and hence execution, of d.) The set Y contains all vertices along paths in $Y_1 \cup Y_2$. These vertices may occur before or after $\alpha \to \beta$ paths, and, hence, any vertices in $Y \cap S$ provide no new information along $\alpha \to \beta$ paths. If $Y_1 = \emptyset$, then d dominates

α on all paths through G; hence, α will not be a part of any ambiguous triangles, as d occurs on *all* paths through α. Conversely, if $Y_2 = \emptyset$, then d post-dominates β with respect to all termination points; hence, β will not be a part of any ambiguous triangles, as d will occur on *all* paths through β.

To formalize the above, we require one new definition:

Definition 4.5. ***(Connected Excluding)*** *For $\Psi \subseteq V$ and $v_1, v_2 \in V$, let $v_1 \xrightarrow{\notin \Psi} v_2$ denote the set of paths from v_1 to v_2 that* do not *cross any edges with a source or target vertex $\psi \in \Psi$. Formally,*

$$v_1 \xrightarrow{\notin \Psi} v_2 \equiv \{p \in v_1 \rightarrow v_2 \text{ such that } V(p) \cap \Psi = \emptyset \text{ or } |V(p)| = 1\}$$

This definition includes trivial paths. That is, for $v \in V$, $v \xrightarrow{\notin \Psi} v$ is nonempty for any $\Psi \subseteq V$, even if $v \in \Psi$. Then, for all (α, β, d) triples where

$$\alpha \in S \cup \{e\} \qquad \beta \in S \cup X \qquad d \in D \setminus S$$

we define the following sets:

$$Y_1 = e \xrightarrow{\notin \{d\}} \alpha \qquad P_{\alpha d} = \alpha \xrightarrow{\notin S \setminus Y} d$$

$$Y_2 = \bigcup_{x \in X \setminus \{d\}} \beta \xrightarrow{\notin \{d\}} x \qquad P_{\alpha\beta} = \alpha \xrightarrow{\notin (S \setminus Y) \cup \{d\}} \beta$$

$$Y = \bigcup_{\pi \in Y_1 \cup Y_2} V(\pi) \qquad P_{d\beta} = d \xrightarrow{\notin S \setminus Y} \beta$$

Then, set S is a coverage set of D if and only if:

$$Y_1 = \emptyset \ \vee Y_2 = \emptyset \ \vee P_{\alpha d} = \emptyset \ \vee P_{\alpha\beta} = \emptyset \ \vee P_{d\beta} = \emptyset$$

for all (α, β, d) triples defined above. Note that these five disjuncts correspond precisely to the five necessary parts of the ambiguous triangle pictured in Fig. 4.2.

If all five of these subpaths exist for any (α, β, d) triple, then we can form paths p_1 and p_2 from Definition 4.3. Specifically, by selecting appropriate

$$y_1 \in Y_1 \quad y_2 \in Y_2 \quad p_{\alpha\beta} \in P_{\alpha\beta} \quad p_{\alpha d} \in P_{\alpha d} \quad p_{d\beta} \in P_{d\beta}$$

we can form the appropriate paths as

$$p_1 = y_1 \circ p_{\alpha\beta} \circ y_2 \qquad\qquad p_2 = y_1 \circ p_{\alpha d} \circ p_{d\beta} \circ y_2$$

where $\circ$ indicates path concatenation. Thus, if all five of the above subpaths exist for any (α, β, d) triple, then S is not a coverage set of D. The equivalence between this "ambiguous triangle" formulation and our original Definition 4.3 is intuitive based on the pictorial representation of triangles in Fig. 4.2; however, the proof is non-trivial, and we defer the full argument to Appendix A.1.

As a brief example, consider the CFG in Fig. 4.1a, and the input configuration $X = \{6\}$ and $D = \{5\}$. Candidate coverage set $S = \{1, 2, 3, 4, 6, 7\}$ is not sufficient to cover D, because the paths $\pi_1 = \langle e, 1, 2, 3, 4, 6, 2, \mathbf{3}, \mathbf{4}, \mathbf{6}\rangle$ and $\pi_2 = \langle e, 1, 2, 3, 4, 6, 2, \mathbf{3}, \mathbf{5}, \mathbf{6}\rangle$ contain the same set of S vertices ($\{1, 2, 3, 4, 6\}$) but π_2 contains $5 \in D$ while π_1 does not. Note the key difference highlighted in **bold**: the instrumentation cannot distinguish a $\langle 3, 4, 6\rangle$ loop iteration from a $\langle 3, 5, 6\rangle$ iteration. In terms of an ambiguous triangle, we have

$$\begin{aligned} \alpha &= 3 & Y &= \{1, 2, 3, 4, 6\} \\ \beta &= 6 & \langle 3, 5\rangle &\in P_{\alpha d} \\ Y_1.\text{vertices} &= \{1, 2, 3, 4, 6\} & \langle 3, 4, 6\rangle &\in P_{\alpha\beta} \\ Y_2.\text{vertices} &= \{2, 3, 4, 6\} & \langle 5, 6\rangle &\in P_{d\beta} \end{aligned}$$

Again, we see the exact same ambiguity. Because of observations on prior and/or future loop iterations (Y_1 and Y_2), the execution of vertex 4 does not preclude the execution of vertex 5, and we have an ambiguous triangle formed from subpaths $\langle 3, 4, 6\rangle$ and $\langle 3, 5, 6\rangle$.

Consider a few special cases. If $S \supseteq D$, then S is always a coverage set of D, as no possible vertex for d exists (i.e., $D \setminus S = \emptyset$). This aligns with Section 4.1.3's claim that if $D \subseteq I$ (the instrumentable set), then I itself is a coverage set of D. If $S = \emptyset$, then S is a coverage set of D iff $\forall d \in D$, d occurs on either all or no paths from the entry to any termination point. In this case, $\alpha = e$, $\beta \in X$, and $Y \cap S = \emptyset$. Thus, by the definition of $p_{\alpha d}$, $p_{\alpha\beta}$, and $p_{d\beta}$, S is not a coverage set of D if d may or may not occur on paths from e to $\beta \in X$.

All of our approaches assume that I is a coverage set of D. In practice, one might consider cases where this is untrue (and ask for *maximal* coverage given a limited I set). Fortunately, because each $d \in D$ is independent, we can find the maximal $D' \subseteq D$ that I can possibly cover by letting $D' = \{d \in D \text{ such that } I \text{ is a coverage set of } \{d\}\}$.

4.2.2 Optimal MILP Formulation

As we prove in Section 4.1.5, obtaining an optimal solution to the Customized Coverage Probing Problem is NP-hard. Using the characterization in Section 4.2.1, we construct a 0–1 mixed-integer linear optimization problem (MILP) whose solution identifies the optimal coverage set. However, due to the intractability of obtaining an optimal solution (see our evaluation comments in Section 4.3), we give only a very brief overview of the formulation here, and defer the complete MILP model description to Appendix B.

Note that, even before selecting an $S \subseteq I$, we can pre-compute all candidate ambiguous triangles as the set of triples of vertices $\mathcal{T} = \{(\alpha, \beta, d) \in (I \cup \{e\}) \times (I \cup X) \times D\}$. Then, because the Y set does not depend on our choice of S, for each $(\alpha, \beta, d) \in \mathcal{T}$ we also pre-compute the additional set of vertices

$$Y_{\alpha\beta d} = \bigcup_{x \in X, \pi \in (e \xrightarrow{\notin\{d\}} \alpha) \cup (\beta \xrightarrow{\notin\{d\}} x)} V(\pi).$$

For each (α, β, d) triple, the set $Y_{\alpha\beta d}$ corresponds exactly to the Y set from Section 4.2.1. That is, it contains all vertices along d-free paths from e to α (Y_1) or β to a termination point (Y_2).

The goal is to find S, a minimal-cost coverage set of D. We first introduce the binary selection variables

$$z_i = 1 \text{ iff } i \in S$$

to represent the selected coverage set. Next, we use five sets of binary variables, one for each path set in the characterization from Section 4.2.1, to force the associated set to be empty:

$$\begin{aligned}
s_{\alpha d} = 1 &\text{ implies that } e \xrightarrow{\notin\{d\}} \alpha = \emptyset \\
t_{\beta d} = 1 &\text{ implies that } \beta \xrightarrow{\notin\{d\}} x = \emptyset \ \forall x \in X \setminus \{d\} \\
u_{\alpha\beta d} = 1 &\text{ implies that } \alpha \xrightarrow{\notin S \setminus Y_{\alpha\beta d}} d = \emptyset \\
v_{\alpha\beta d} = 1 &\text{ implies that } \alpha \xrightarrow{\notin (S \setminus Y_{\alpha\beta d}) \cup \{d\}} \beta = \emptyset \\
w_{\alpha\beta d} = 1 &\text{ implies that } d \xrightarrow{\notin S \setminus Y_{\alpha\beta d}} \beta = \emptyset
\end{aligned}$$

The model is constructed as a network flow problem, with constraints to force the non-existence of each of the above paths. The total number of constraints is non-trivial, but polynomial in the size of $\mathcal{T}$, V, E, and X; again, see Appendix B for the complete formulation.

Recall from Section 4.2.1 that S is a coverage set of D if and only if at least one of these five sets of paths is empty for all $(\alpha, \beta, d) \in \mathcal{T}$. (This appears to be a slightly stronger condition, as the set of (α, β, d) triples should also be constrained by the selection of S, per Section 4.2.1. However, all additional triples in $\mathcal{T}$ are redundant because they indicate α or β selections that are not observations, and therefore can have no meaningful impact on Y or relevant triangle paths.) Now, the five sets of triangle paths are only relevant when $z_d = 0$, because instrumented vertices are always observed. The following constraint forces one of the five paths to be empty, but only when $z_d = 0$:

$$s_{\alpha d} + t_{\beta d} + u_{\alpha\beta d} + v_{\alpha\beta d} + w_{\alpha\beta d} \geq (1 - z_d) \; \forall (\alpha, \beta, d) \in \mathcal{T}$$

In the end, our objective is to minimize cost

$$\sum_{i \in V} c_i z_i$$

subject to the above forcing constraint on program paths, and the relevant constraints for each of the five classes of paths.

This approach is guaranteed to provide a provably optimal coverage set, but unfortunately is too slow in practice. In fact, we were only able to evaluate the fully optimal approach on our smallest test subjects (see Section 4.3). There are a number of reasons for this. The formulation requires pre-computation of the sets $Y_{\alpha\beta d}$, which are of size quartic in the number of vertices in G. Furthermore, even with powerful commercial software, solving a large-scale MILP still relies on enumerating an often large branch-and-bound search tree. Fortunately, safe and fast approximations of the optimal result are possible.

4.2.3 An Inexpensive Approximation

Several prior approaches optimize coverage probes by using the dominance relation among basic blocks [4, 169]. A basic block v dominates a basic block w if and only if $e \xrightarrow{\notin \{v\}} w = \emptyset$. Immediate dominance relations for any single-entry directed graph form a tree, and algorithms for computing dominators are well-known [7, 103]. Figure 4.3 shows the dominator tree for our running example from Fig. 4.1a.

In this section, we develop an inexpensive approximation algorithm based on dominator information, rather than the sufficiency condition from Section 4.2.1. Our approach performs a bottom-up traversal of the dominator tree, "covering" a block's subtrees only as necessitated

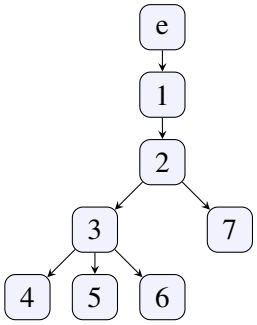


Figure 4.3: Dominator tree for Fig. 4.1a

by the desired set, D. This approach is inspired by the prior work of Agrawal [4] and Tikir and Hollingsworth [169], but supports customized coverage. Most accurately, our approach generalizes these prior approaches, which could be considered special cases of our algorithm for only complete executions [4] and full coverage [4, 169].

A vertex, v, can be "covered" (i.e., guaranteed accurate coverage information for any execution) in two possible ways. First, v itself may be instrumented, so that its coverage is observed directly. Second, we might instrument an appropriate subset of v's dominator-tree descendants such that all executions through v must execute at least one vertex in the descendant set. Clearly, for this approximation, we must instrument all leaves of the tree. Internal block v must be instrumented only if v's dominator-tree children cannot cover v. In our case, v is instrumented if a path exists in G from v to some $x \in X$ that bypasses all of v's covered children in the dominator tree (and such that v does not dominate x). By the definition of dominance, any time a descendant of v executes (including a crashing execution), it implies the execution of v. Intuitively, if the program can halt after executing v without an observation implying v's execution, then v's coverage data is unknown on some execution.

For example, consider vertex 3 in Fig. 4.3, and a crash $x = 7$ (i.e., the program halts in block 7). Here, the subset of dominator children $\{4, 5\}$ is sufficient to cover 3: in Fig. 4.1, all paths from 3 to 7 must pass through some element of $\{4, 5\}$. Likewise, $\{6\}$ would also cover 3: any CFG path from 3 to 7 must pass through 6. Of course, both alternatives assume that I includes the necessary instrumentation points. If I disallows both $\{4, 5\}$ and $\{6\}$ as instrumentation plans, then 3 can only be covered by direct instrumentation of 3 itself.

Figure 4.4 details our algorithm. The global set *willInst* builds up the final result: the set of basic blocks to be probed for coverage. The global set *willCover* tracks which blocks will be guaranteed to have accurate coverage information available. Hence, as vertices are added to *willInst*, *willCover* is updated to reflect the newly covered nodes. The overall goal is to

```
global: willInst, the set of vertices that will be probed
global: willCover, the set of vertices for which coverage information will be available

input: G = (V, E), a single-function control-flow graph
input: e ∈ V, the entry vertex
input: I ⊆ V, vertices that may be probed
input: c : V ↦ ℝ+, costs for vertices
input: D ⊆ V, vertices with desired coverage
input: X ⊆ V, possible ending vertices
output: willInst ⊆ I, a coverage set of D

T = dominator tree for G, with entry vertex e;
ord_T = any bottom-up ordering of T.vertices;

willInst = ∅;
willCover = ∅;
canCover = ∅;
needInst = ∅;
foreach v in ord_T do
    coveredChildren = T.children_of(v) ∩ willCover;
    if ¬exitWithout(v, coveredChildren, X) then
        willCover ∪={v};
        canCover ∪={v};
    else
        canCoverChildren = T.children_of(v) ∩ canCover;
        vNeedInst = exitWithout(v, canCoverChildren, X);
        if v ∈ I ∨ ¬vNeedInst then canCover ∪={v} ;
        if vNeedInst then needInst ∪={v} ;

        if v ∈ D then
            if v ∉ canCover then return FAIL ;
            else cover(v, canCover, needInst, T, X, c) ;

return willInst;
```

Figure 4.4: Dominator-based approximation

make $D \subseteq willCover$, that is, to "cover" all vertices in D. First, we compute T, the dominator tree of G, and any bottom-up (i.e., reverse-topological) ordering of T's vertices, *ord_T*. Then, we iterate over each vertex in *ord_T*, adding those vertices from I that require instrumentation to the set *willInst*. During this iteration, we discover which vertices can only be covered via direct instrumentation (stored in set *needInst*), and which could possibly be covered either via direct instrumentation or via their dominator descendants (stored in set *canCover*).

In the loop, we first find v's dominator children that are already covered based on updates to *willCover* in previous loop iterations (*coveredChildren*). If these vertices are already

Function exitWithout(*v, children, X*)
input: v, the vertex possibly covered
input: *children*, a subset of v's immediate-dominator children that may provide coverage information for v
input: X, possible ending vertices
return $\exists x$ such that $x \in X$ and v does not dominate x and $v \xrightarrow{\notin children} x \neq \emptyset$;

Figure 4.5: Test for an exit bypassing dominator children

sufficient to cover v (i.e., no path exists from v to an exit or crash that bypasses v's covered children, as defined in function exitWithout() from Fig. 4.5), then v is added to *willCover* and *canCover*. Otherwise, we gather all of v's dominator children that possibly could be covered (*canCoverChildren*). Because we process T bottom-up, *canCover* already contains all of v's dominator children that could possibly be covered. If *canCoverChildren* is insufficient to cover v, then v is added to *needInst*. If v can be covered by its dominator children or can be directly instrumented, v is added to *canCover*. At this point, if $v \in D$, we want to find a cheap coverage set for v. However, note that this approximation assumes that a vertex can only be covered by its dominator descendants, which is not always true. If we desire coverage for v but $v \notin$ *canCover*, then the algorithm fails to find a coverage set for D. This situation is rare in practice, and another approach (such as our locally optimal approximation, which we describe in Section 4.2.4) could reduce the size of I in this case. If $v \in$ *canCover*, we find a coverage set for v via a call to cover().

Procedure cover() in Fig. 4.6 walks back down the dominator tree in order to cheaply cover vertex v. First, if v cannot be covered by its dominator children (i.e., $v \in$ *needInst*), then we instrument v to obtain its coverage data. Otherwise, we iterate over all of v's dominator children that can be covered, sorted by cost to try to avoid instrumenting the costliest vertices. (This is a heuristic, but it is effective in practice.) For each child, w, if w is already covered (i.e., $w \in$ *willCover*), we skip it. Otherwise, if w is necessary to cover v (as determined by the call to exitWithout()), we must recursively cover w. After the completion of cover(), v is covered either via direct instrumentation, or via calls to cover() on its descendants. Thus, we pass each vertex as argument v to cover() at most once, since v is added to *willCover* at the conclusion of cover(), and will be excluded from *removableChildren* in future calls.

Our approach is most similar to that of Tikir and Hollingsworth [169], who instrument a basic block, v, whenever v is either a leaf vertex in the dominator tree, or has an outgoing edge (in G) to a block that v does not dominate. This is equivalent to our approach in the

```
Procedure cover(v, canCover, needInst, T, X, c)
    input: v ∈ canCover, the vertex to cover
    input: canCover, a pre-computed set of vertices that could be covered
    input: needInst, a pre-computed set of vertices whose coverage information can only be
           determined by direct probing
    input: T, the dominator tree for the function containing v
    input: X, possible ending vertices
    input: c, costs for vertices

    if v ∈ needInst then
        willInst ∪={v};
    else
        canCoverChildren = T.children_of(v) ∩ canCover;
        assert ¬exitWithout(v, canCoverChildren, X);

        removableChildren = canCoverChildren \ willCover;
        foreach w in removableChildren ordered by c do
            if exitWithout(v, canCoverChildren \ {w}, X) then
                cover(w, canCover, needInst, T, X, c);
            else
                canCoverChildren \={w};

    willCover ∪={v};
```

Figure 4.6: Cover a dominator tree vertex

un-customized special case of $I = D = X = V$, since any such outgoing edge from v targets a possible halting location. However, our approach handles the full range of input from Section 4.1, allowing us to optimize coverage with far more degrees of flexibility. We look for paths to any non-dominated termination point (Fig. 4.5), and only cover vertices where necessitated by D (Fig. 4.6).

4.2.4 Locally Optimal Approximation

The approach in Section 4.2.3 is computationally inexpensive: it calls cover() on each block at most once, and, therefore, traverses each dominator tree vertex at most twice. However, it provides no guarantees on the optimality of the *willInst* set. In fact, as noted in Section 4.2.3, it is possible that the dominator-based approximation will be unable to find any coverage set $S \subseteq I$, even if at least one such set exists. This is the "**return** FAIL" case in Fig. 4.4.

We can compute a locally optimal coverage set in polynomial time by iteratively testing smaller-and-smaller candidate coverage sets via the conditions in Section 4.2.1. By these

input: $I \subseteq V$, vertices that may be probed
input: $c : V \mapsto \mathbb{R}^+$, costs for vertices
input: $D \subseteq V$, vertices with desired coverage
output: $S \subseteq I$, a locally optimal coverage set of D

assert I is a coverage set of D;
S = copy(I);
tryRemove = sort I by c;
foreach i in *tryRemove* **do**
    if $S \setminus \{i\}$ *is a coverage set of* D **then**
        $S = S \setminus \{i\}$;
return S;

Figure 4.7: Locally optimal approximation

conditions, a candidate coverage set S can clearly be checked in polynomial time. For each (α, β, d) triple arising from our current choice of S, we:

1. compute Y, which requires two depth-first or breadth-first search passes (one to gather all possible vertices along Y_1 paths $e \xrightarrow{\notin\{d\}} \alpha$, and one to gather vertices along Y_2 paths $\beta \xrightarrow{\notin\{d\}} x$);
2. check for the existence of any path $\alpha \xrightarrow{\notin S \setminus Y} d$;
3. check for the existence of any path $\alpha \xrightarrow{\notin (S \setminus Y) \cup \{d\}} \beta$; and
4. check for the existence of any path $d \xrightarrow{\notin S \setminus Y} \beta$.

Each of the five connected-excluding tests again requires a single depth-first or breadth-first search. If, for any (α, β, d) triple, the vertex sets for Y_1 and Y_2 from Item 1 are non-empty and a path exists for all Items 2 to 4, then S is *not* a coverage set of D.

A coverage set S is locally minimal with respect to D when S is a coverage set of D, and $\forall S' \subset S$, S' is not a coverage set of D. Figure 4.7 gives the direct approach. We begin with $S = I$ (a coverage set of D by assumption), and iteratively attempt to remove each element of S. Removing vertices from S can never cause S to cover *more* vertices. Thus, if S is a coverage set of D, then $\forall S^{\uparrow} \supset S$, $S^{\uparrow}$ is also a coverage set of D. Contrapositively, if S is *not* a coverage set of D, then $\forall S^{\downarrow} \subset S$, $S^{\downarrow}$ is not a coverage set of D either.

This approach has polynomial time complexity, but performs redundant computation, and is too inefficient for practical use. We improve performance using a number of optimizations and heuristics. We begin by reducing our initial I set by a call to our dominator-based

approximation (Fig. 4.4). If this approximation returns FAIL, then I cannot be proven a coverage set of D using only dominance relations, and we begin with the full user-specified I set. As Fig. 4.7 shows, we heuristically attempt to first remove the costliest vertices from S. We pre-compute $V(Y_1)$ for each (α, d) pair, and $V(Y_2)$ for each (β, d) pair, as these sets are not dependent on the choice of S. We also perform substantial pruning of possible (α, β, d) triples. For example, as Fig. 4.2 illustrates, all possible α vertices must precede d along some path through G, and all β must follow both α and d along some path through G; this pruning both reduces the number of α and β vertices to consider, and could pre-compute some paths for Items 1 to 4 above. Finally, we find that, in practice, when ambiguous triangles exist, they tend to exist in close proximity to the un-covered $d \in D$. In other words, the length of paths in $\alpha \xrightarrow{\notin S \setminus Y} d$, $\alpha \xrightarrow{\notin (S \setminus Y) \cup \{d\}} \beta$, and $d \xrightarrow{\notin S \setminus Y} \beta$ tends to be short. Thus, we prioritize testing α and β vertices crossing the fewest edges from d.

Overall, though, our approach does not fundamentally differ from Fig. 4.7, and our evaluation finds that our locally optimal approach takes substantial time to run, particularly when having to prove (via the search for Items 1 to 4 above for all possible (α, β, d) triples) that a candidate S set is in fact a coverage set of D. Recall that the approach is approximate; any solution S contains no unnecessary blocks to cover D, but other less-costly instrumentation plans may exist. Thus, as stated, in the context of our MILP from Section 4.2.2, this approach results in a locally optimal solution.

4.3 Experimental Evaluation

We evaluated both the compile-time and run-time efficiency of our coverage optimization approaches described in this chapter. We implemented the techniques described in Section 4.2 for C/C++ programs by extending the coverage mechanisms in `csi-cc` from Chapter 2. We used LLVM's built-in `BlockFrequency` analyses to determine costs (c_i for each $i \in I$) as input to our approach. This statically approximates the execution frequency of each block, but is realistic since even a run-time profile approximates post-deployment behavior.

Since our optimizations in this chapter apply intraprocedurally, we ran experiments optimizing our statement coverage ($I = D = V$) and call-site coverage ($I = D =$ {basic blocks containing at least one call site}) mechanisms. Call-site coverage is an example of customized coverage that cannot be optimized by any prior approach. Note the slight difference from call-site coverage as described earlier in this chapter (Section 4.1.4), where we proposed allowing instrumentation anywhere (i.e., $I = V$). In practice, we have found that LLVM's

Table 4.1: Evaluated applications, ordered by size.

Application	Description	Versions	Mean LOC
tcas	Siemens	1	173
schedule2	Siemens	1	373
schedule	Siemens	1	413
replace	Siemens	1	563
tot_info	Siemens	1	564
print_tokens2	Siemens	1	568
print_tokens	Siemens	1	727
ccrypt	Linux utility	1	5,280
gzip	Linux utility	5	8,114
space	ADL interpreter	1	9,563
exif	Linux utility	1	10,611
bc	Linux utility	1	14,292
sed	Linux utility	7	14,314
flex	Linux utility	5	14,946
grep	Linux utility	5	15,460
bash	Linux shell	6	80,443
gcc	C compiler	1	222,196

static cost model is not always a good representation of run-time costs, which means that our approaches may be driven to choose more expensive instrumentation plans. By setting $I = D$, we ensure that our optimizations can only remove probes; they cannot, for example, cover basic block b by inserting new probes into b's dominator descendants (present in I but absent from D) to assure coverage of b. Note that inaccurate cost data is a threat to run-time efficiency, but never to correctness. That is, our approaches can never select a result S that is not a coverage set of D, even if S results in suboptimal run-time performance. In all cases, we optimize coverage assuming that programs may crash at any statement: $X = V$.

Table 4.1 lists our subject programs. Note that we include a larger set of applications than in previous chapters; the additional applications (exif, bc, and bash) do not appear in our analysis evaluations (see Part II, Chapters 5 and 6). However, our optimizations do not impact results for a precise analysis, because we only optimize out coverage instrumentation that can be inferred from remaining coverage probes. Of the new applications, bash comes from the Software-artifact Infrastructure Repository [55, 154], while bc and exif are real-world programs. We used non-faulty builds for most applications, though gcc and exif used builds with a known fault.

Table 4.2: Relative compilation times. "–" marks compilations that did not complete within 3 hours.

	Statement Coverage			Call-Site Coverage		
Application	None	Dominators	Local	None	Dominators	Local
tcas	1.3	1.3	1.3	1.3	1.3	1.3
schedule2	1.4	1.4	2.0	1.3	1.3	1.3
schedule	1.4	1.4	5.1	1.3	1.3	1.5
replace	1.4	1.4	43.2	1.3	1.3	1.3
tot_info	1.3	1.3	14.2	1.3	1.3	1.4
print_tokens2	1.3	1.3	5.4	1.3	1.3	1.3
print_tokens	1.4	1.4	360.6	1.3	1.3	1.7
ccrypt	1.5	1.5	5.1	1.4	1.4	1.6
gzip	1.7	1.6	3,157.8	1.4	1.4	64.7
space	1.6	1.6	24.8	1.5	1.5	1.6
exif	1.5	1.5	22.2	1.5	1.4	12.3
bc	1.6	1.6	365.5	1.5	1.5	18.7
sed	2.0	1.8	–	1.5	1.5	3,744.4
flex	2.1	1.8	–	1.6	1.6	–
grep	1.9	1.7	–	1.4	1.4	12.1
bash	1.5	1.5	–	1.5	1.5	–
gcc	1.8	1.7	–	1.5	1.5	–

We exclude fully optimal coverage results from all experiments. Our MILP optimizer based on Section 4.2.2 and Appendix B either exceeds our compilation time limit (3 hours) or runs out of memory for all but the 3 smallest Siemens benchmarks; even these incur over 1,000× slowdown in compilation time over our baseline.

4.3.1 Optimization and Compile Time

For each version of each application, we first measured the wall-clock time to perform each of our optimization approaches and instrument the program. These results are shown in Table 4.2, relative to a base build with "`clang -O3`": a value of 1.0 indicates no compilation-time overhead. We built each application version at least three times, and divided by base compilation time. We then took the geometric mean to aggregate across all versions (to avoid over-representing specific versions). The "None" columns indicate compilation overhead for instrumenting all $i \in I$ for the selected option, that is, compilation time with no optimization, as in Chapter 2. The "Dominators" columns show overhead for the dominator-

based approximation from Section 4.2.3. The "Local" columns show overhead to obtain a locally minimal solution. In all cases, we instrumented functions to gather *local* coverage data; again, this correspond to optimizing instrumentation for the stack-local coverage array `cov` from Chapter 2, Fig. 2.5.

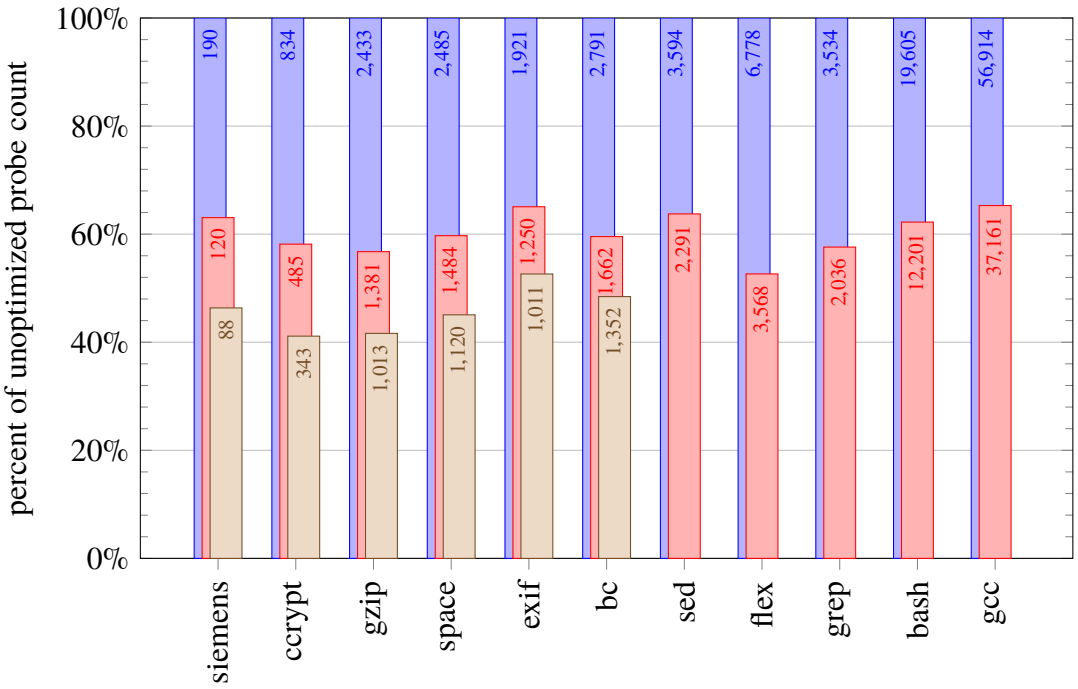
The results are grouped into statement coverage results (gathered as basic block coverage) and call-site coverage results. Without optimization, there is a cost of 1.3×–2.1× to compile benchmarks for statement coverage. However, the dominator-based optimization is extremely inexpensive, often saving time over full instrumentation. We believe that the extra cost results from generating and inserting the required probing code. The overhead of locally minimal optimization varies greatly between benchmarks. We find that large, complex functions take a disproportionately long time to optimize. For example, gzip's base compile time averages under 2 seconds, but computing a locally optimal solution requires over 1.5 hours, dominated by 3 functions that consume over 90% of the total time. As we discuss in Section 4.2.4, the locally optimal approach has polynomial time complexity; however, the description in Section 4.2.1 indicates that, in the worst case, we must consider all possible (α, β, d) triples in order to prove that a particular $S \subseteq I$ is a coverage set of our desired set D.

Restricting the set of desired blocks, D, reduces the number of (α, β, d) triples considered by our locally optimal approach, and should reduce optimization time. Our call-site coverage compilation results confirm this; we find substantially smaller compile-time overheads when compiling for coverage only at call sites. For example, while gzip's compile time increases from 1.5 seconds to just over 2 minutes, this is far below the 3,000× increase we see for optimizing statement coverage. These improvements allow us to compute locally optimal solutions for two additional benchmarks (grep and sed). However, 3 benchmarks still do not complete compilation, and sed exhibits a 3,744× slowdown; thus, scalability remains a concern for our locally optimal formulation.

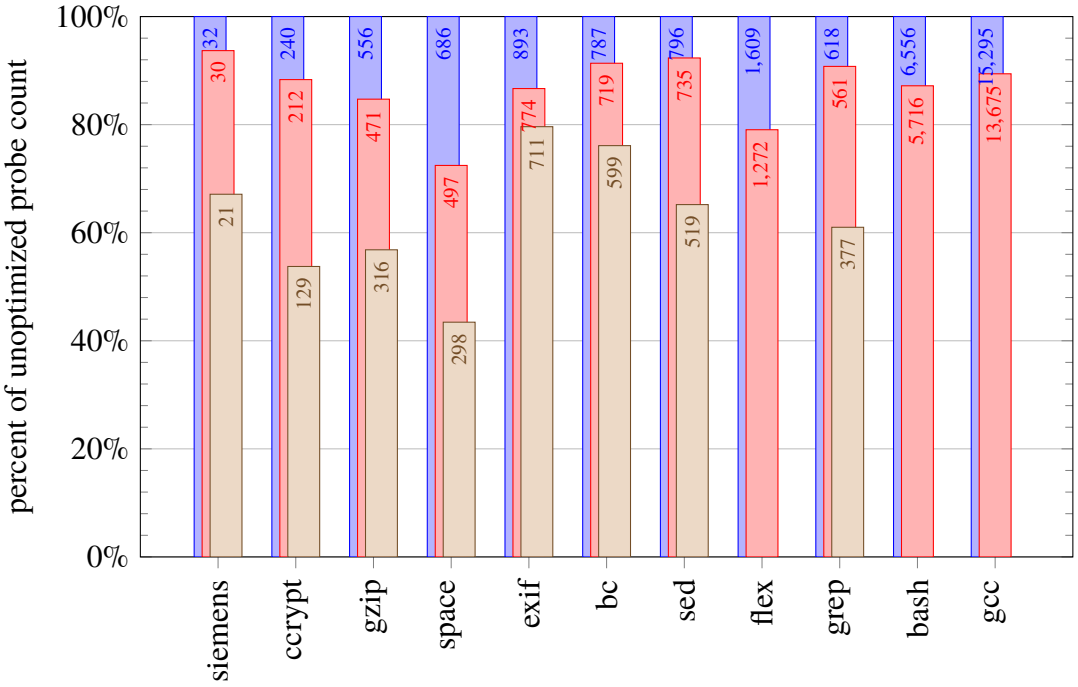
4.3.2 Static Probe Counts

We also gathered the total number of probes inserted by each approach, to examine the static reduction in probe insertions for our optimizations. We took the arithmetic mean of probe counts for different versions of each application. As in previous chapters, we also aggregated results for all Siemens applications; probe reductions are very similar across all Siemens benchmarks of non-trivial size.

Figure 4.8 shows results for statement coverage and call-site coverage. All bars are scaled to 100% for no optimization to show how much relative reduction the locally optimal



(a) Statement coverage



(b) Call-site coverage

Figure 4.8: Coverage probe counts, scaled to 100% for no optimization.

and dominator-based approximations achieve. Stacked bars indicate the percentage of the unoptimized probes still included after the specified optimization. Numbers within each bar are mean counts without scaling.

Figure 4.8a shows results for statement coverage. As noted in Section 4.2.3, since $I = D = X = V$ for statement coverage, our dominator-based approximation is equivalent to the optimizations of Tikir and Hollingsworth [169] in this specific scenario. Those applications that cannot be compiled within our time limit ("–" in Table 4.2) exclude locally optimal results in the figure. Overall, probe reductions are substantial. The Siemens benchmarks average 190 probes with no optimization, but just 120 probes with the dominator-based approximation: 70 probes have been optimized away, on average. In relative terms, these Siemens benchmarks have just 63% as many probes with dominator-based optimization as they do with no optimization. Our locally optimal approach further reduces this count to 88 probes on average, a reduction of 54%. As another example, the dominator-based approximation reduces bc's probe count by approximately 40% relative to unoptimized instrumentation. The locally optimal approach further reduces the probe count, cutting the remaining probes to just below half of the original set. Our dominator-based approximation is inexpensive, but still reduces probe counts by over 40% on average. The locally optimal approach is substantially more expensive (as seen in Table 4.2), but further reduces necessary instrumentation to just 44% of unoptimized instrumentation for completed benchmarks, on average.

The main thrust of our approaches, however, comes in their ability to optimize instrumentation based on *customized* coverage requirements. Figure 4.8b presents results for call-site coverage. Note that prior work cannot optimize coverage probes in this scenario. Here, the unoptimized instrumentation set is much more selective; that is, absolute probe counts for no optimization are much smaller. Reductions are also smaller across all benchmarks, but, for most applications, we still see substantially less instrumentation. The dominator-based approximation reduces probe counts by up to 28% (space), and averages a 15% reduction across all benchmarks. The benefits of the locally optimal approach are very pronounced. For those applications that completed local optimization, we see an average further reduction of 30% from the dominator-based approximation, with total reductions as high as 57% (space, relative to unoptimized).

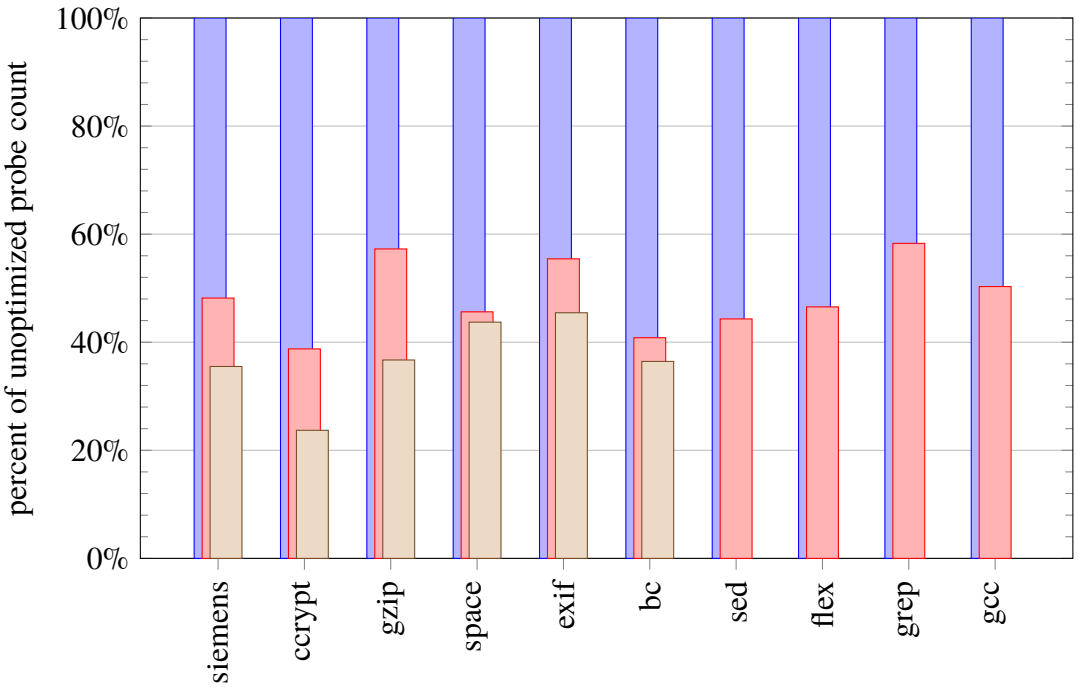
4.3.3 Dynamic Probe Counts

Next, we assessed the *dynamic* impact of our optimizations on run-time performance. Because our optimization approaches are general (and could be applied to other techniques beyond

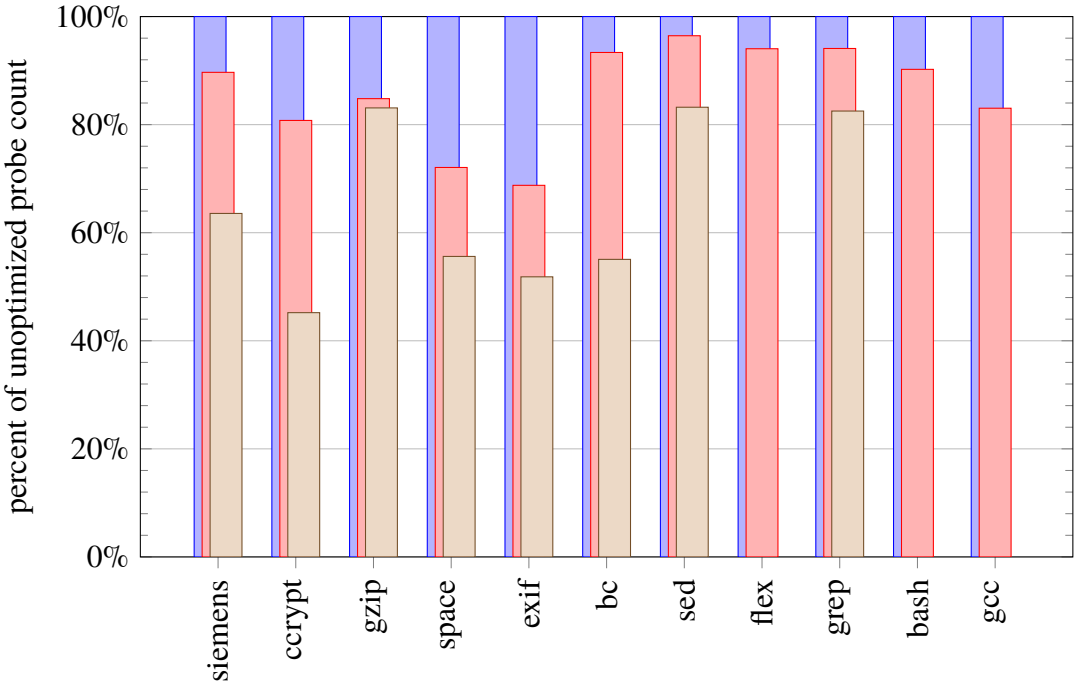
`csi-cc` instrumentation), we first gathered the total number of probe executions at run time. The goal of this experiment is to abstract from the cost of executing each *individual* coverage probe, and instead focus on the reduction in total number of probe executions. As in previous chapters, we ran each application through its corresponding test suite. We gathered the count for each trial, and computed the percentage reduction for each level of optimization. We took the arithmetic mean to aggregate across each complete test suite, and aggregated the resulting values across all versions of each application (to avoid over-representing specific versions or long-running test cases). We again aggregated results for the Siemens applications, which exhibit similar run-time performance.

Figure 4.9 shows our results. All bars are again scaled to 100% for no optimization to show how much relative reduction the locally optimal and dominator-based approximations achieve. Mean counts without scaling are omitted due to space limitations in plots.

Figure 4.9a shows dynamic probe execution reductions for statement coverage. We excluded one test case for gzip that exceeds our 1-hour timeout for extracting probe counts. We also omit bash results, as bash's test suite is highly sensitive to our probe-counting infrastructure with dense statement coverage. Again, $I = D = X = V$, and our dominator-based approximation is equivalent to Tikir and Hollingsworth [169]. For all of the applications, even this approximation results in a substantial drop in overheads; in fact, all applications lose at least 60% of their probe executions, while simultaneously shrinking compile time per Table 4.2. ccrypt sees the largest reduction, executing just 39% of the unoptimized probe count. Although expensive to compute, overhead reductions from further reducing probes via the locally optimal formulation are sometimes substantial. For example, after the dominator-based approximation reduces gzip's probe executions by 43%, our locally optimal approach removes more probes, reducing probe executions to just 37% relative to uninstrumented code. Averaging absolute counts for gzip, the dominator-based approximation reduces probe executions from just over 1,000,000 to 540,000, and the locally optimal approximation further reduces this count to around 300,000. ccrypt is reduced to just 24% of the unoptimized count via locally optimal optimization; in absolute terms, probe executions are reduced from over 24,000 to 6,300 on average. Of course, as mentioned earlier, this run-time performance may come at a cost: gzip's compile time increases from seconds to hours when moving to a locally optimal solution. Overall, the performance of the dominator-based approximation is quite impressive. Our locally optimal approach presents a significant trade-off: it does often remove significantly more probes (see Fig. 4.8a), but at a very high compilation cost (see Table 4.2).



(a) Statement coverage



(b) Call-site coverage

Figure 4.9: Dynamic probe executions, scaled to 100% for no optimization.

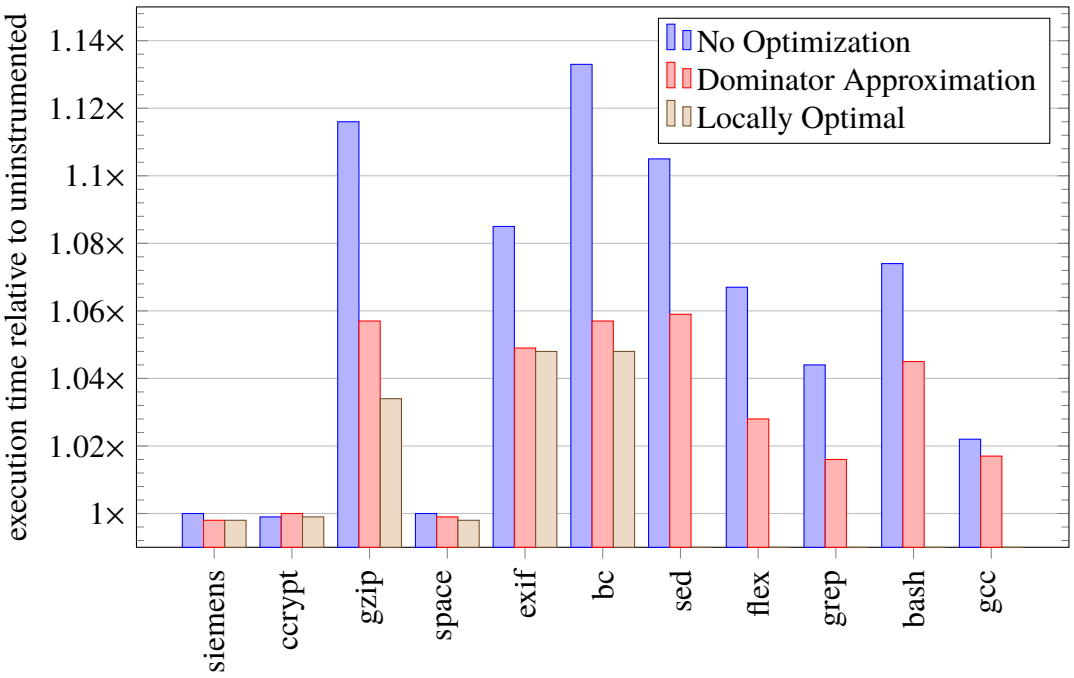


Figure 4.10: Run-time overhead with statement coverage

Figure 4.9b presents call-site coverage results. We exclude 10 (out of 1061) bash test cases due to time-out. Here, unoptimized probe executions are much smaller, and reductions from the dominator-based approximation are less pronounced. Nevertheless, some applications see significant benefit. For example, exif and space both execute just 70% of the unoptimized probe counts. For exif, optimization reduces absolute counts from over 27,000 to approximately 18,000. Our locally optimal approach, however, is very impressive. While some applications see less benefit (e.g., gzip), many applications see enormous reductions. For example, bc and ccrypt both reduce probe executions by 40% beyond the reductions of the dominator-based approximation. In absolute terms, the locally optimal approach reduces dynamic probe executions in bc from over 11,000,000 (no optimization) to approximately 6,000,000.

4.3.4 Running Time

The results from Figure 4.9 do not depend on probe costs, but real impacts on running time depend on the cost to execute each probe. Recall from Chapter 2 that our in-memory coverage instrumentation was already quite efficient, though our denser coverage mechanisms (statement coverage in particular) had overheads that were too high for many deployed scenarios.

We, therefore, measured execution times of each program's test suite, computing overheads relative to "`clang -O3`" as a baseline. Figure 4.10 shows results for statement coverage optimization. Note that these results differ from those plotted for statement coverage in Chapter 2, Fig. 2.6, as we use compiler optimization level "`O3`" here, while previous plots showed results without compiler optimizations. For the larger benchmarks, we observe significant reductions. Mean overhead for the non-trivial benchmarks shrinks from 7.2% to 3.6% using the dominator-based approximation. Although expensive to compute, the locally optimal approach can provide substantial additional benefit. For example, the dominator-based approximation reduces gzip's overhead from 11.6% to 5.7%; our locally optimal formulation reduces this further to just 3.4% relative to uninstrumented code.

For call-site coverage, we were unable to measure a significant difference in running time due to our optimizations. This is surprising, given the reductions shown in Fig. 4.9b. However, `csi-cc` in-memory binary probes are already quite fast, and the majority of our test cases do not run long enough to exhibit large overheads. If we optimized instrumentation for another tool where probes were more costly, we would expect much more pronounced results. Furthermore, the larger numbers of probes used for statement coverage show a correspondingly larger optimization benefit. Finally, as noted previously, there are other advantages to reducing coverage instrumentation aside from run-time overheads. Notably, as we find in Chapter 6, the density of failure report elements can have a significant impact on the efficiency of postmortem analysis; thus, reducing redundant tracing has other downstream benefits.

4.3.5 Discussion

For all results discussed in this chapter, we also assessed the statistical significance of our findings. We conducted a Wilcoxon signed-rank test between test cases with each level of optimization. For all results that show non-trivial differences in overhead, we find sufficient evidence ($p < 0.01$) to reject the null hypothesis that our optimizations have no effect on the metrics we gather for our programs. More specifically, we find that our results are statistically significant ($p < 0.01$) for all static probe count and dynamic probe execution reductions, as well as for each non-zero reduction in run-time overhead for statement coverage from Fig. 4.10.

Overall, our results show that our optimization approaches can significantly reduce the static and dynamic cost of customized coverage instrumentation. In light of its advantages found in previous chapters, we specifically optimized `csi-cc`'s call-site coverage; this is an

instance where prior techniques are unable to reduce necessary instrumentation, and where (prior to optimization) the number of required coverage probes is already substantially reduced from full statement coverage. Even so, our techniques reduce static probe counts by as much as 57% (space in Fig. 4.8b), and dynamic probe executions by up to 55% (ccrypt in Fig. 4.9b). Even when the absolute numbers are not large, these reductions should not be discounted: these may be very important in deployed systems with real-time requirements, or to reduce the size of failure reports for downstream postmortem analyses.

4.4 Threats to Validity

The threats to validity for our results are again similar to those in previous chapters. To mitigate threats to external validity, we expanded our selection of benchmarks from previous evaluations. However, all applications were still written in C (though our implementation in `csi-cc` applies to C and C++ programs), so we cannot guarantee that our approaches will produce similar results on more modern object-oriented languages. To apply our optimizations, a client must provide a number of inputs, including: a set of desired basic blocks, a set of instrumentable basic blocks, and a cost map for the program's control-flow graph. This last item, in particular, may be challenging to obtain in some contexts (and can never be fully accurate with respect to all possible executions of a program); however, as we note in Section 4.3, an inaccurate cost map can only limit gains in efficiency, and is never a threat to correctness (i.e., will never cause our approaches to remove probes that are in fact necessary to cover the desired set).

As in previous chapters, we are measuring very small overhead ratios for our test subjects, and variance in our measurements is a threat to internal validity. As before, we ran all experiments on a lightly-loaded machine, and ran many trials for each application. We do, in fact, see some variance in our results (mostly in execution-time measurements); however, we assessed the statistical likelihood of our results (see Section 4.3), and found that our measured overhead differences are very unlikely to occur by chance ($p < 0.01$ for all non-trivial reductions). While we have not formally proven the correctness of our implementations, we did take steps to verify probe sets returned by each optimizer. We spot-checked a small selection of results for each approach on the smaller benchmarks, and verified that we returned valid coverage sets. The locally optimal approach is, by design, a coverage set verifier, so we used it to verify sets returned from the fully optimal implementation and the dominator-based approximation.

4.5 Related Work

Closely related work optimizes placement of binarized coverage probes. Agrawal [4] optimizes probes by forming "superblocks" from sets of basic blocks based on dominance and post-dominance relations. Later, Agrawal [3] extends this work to interprocedural dominance relations; Li et al. [105] and Xu et al. [185] extend these optimizations beyond superblocks. Tikir and Hollingsworth [169] also optimize coverage probe placement via dominators, but use a faster, simpler approach (and no post-dominance information) for online instrumentation. Our approximation in Section 4.2.3 is inspired by many of these prior approaches. However, because we support multiple crash points (via input parameter X), we cannot directly take advantage of post-dominator information as per Agrawal [3, 4]. Furthermore, because we allow customization of desired and instrumentable locations (via input parameters D and I, respectively), we develop a generalization of these existing approaches, facilitating many coverage optimization scenarios that are not supported by any prior work. Particularly for local coverage data (i.e., without the transformation from Fig. 4.1b), dominance and post-dominance relations do not capture the tightest coverage set definition from Definition 4.3; in essence, the use of dominance information for coverage optimization *assumes* multiple function executions are possible. Thus, our approaches can optimize instrumentation more aggressively than any prior work.

Binarized coverage only needs to observe each probed location one time: once a coverage value becomes true, it stays true. Building on this insight, prior work optimizes coverage gathering via dynamic insertion and deletion of probes [38, 85, 117, 118, 141, 169]. These techniques are complementary to our own: we optimize *where* to insert probes, while such techniques address *how* and *when* to insert probes. While they may be useful in conjunction with `csi-cc`, note that these approaches can be somewhat more invasive. Specifically, all such techniques require either a virtual machine or other run-time monitoring support to insert or remove instrumentation dynamically. Kasikci et al. [85] achieve this monitoring through extremely lightweight code breakpoints which remove the relevant instrumentation code when triggered; such a minimally-invasive approach with low run-time cost may be an good match.

Many commercial tools gather program coverage over test suites (e.g., [20, 62, 74, 166]). These tools gather complete program coverage, whereas our work allows a developer to *focus* tracing to reduce overheads and/or limit possible instrumentation.

Prior work has optimized instrumentation to gather *frequency counts* of statements or edges, often to identify program "hot spots." Knuth and Stevenson [89] optimize frequency counter placement for program statements, and Knuth [88] optimizes instrumentation for edge

counts. Ball and Larus [21] formalize these classic approaches, and generalize the counting problems for vertices and edges. While these classic approaches run in polynomial time, their solutions cannot be used for binarized instrumentation: they rely on Kirchoff's current law, which does not hold of binarized indicators. Furthermore, while coverage data can obviously be derived from count data, many use cases that we consider would not make use of the more detailed count information. The cost of gathering counts is higher than the cost of gathering coverage. In contexts where counts are not required, binarized coverage has a number of advantages: (1) each probe is less expensive to execute, (2) each probe requires only one bit of storage, whereas integers for counts require more storage and are susceptible to overflow, (3) instrumentation is easily made thread-safe on all architectures, and (4) probes can be removed after they are first triggered.

As noted at the start of this chapter, prior work reduces the run-time cost of gathering coverage post-deployment by tracing a limited set of program entities. For example, Pavlopoulou and Young [143] monitor coverage for code that remains uncovered by a program's test suite. The GAMMA project [31, 139] spreads coverage-gathering tasks across large user communities to reduce the overhead burden of each individual tracing instance. In the terms of this chapter, each individual entity only traces a subset of the overall desired set, D. Our optimizations are directly applicable here, further reducing required coverage probes for each deployed instance.

Prior work optimizes the set of test cases required to achieve a specific coverage criterion [4, 26, 113, 187]. Early approaches to test-suite minimization have much in common with approaches to minimizing coverage probes. Most notably, these techniques can also make use of dominance information to determine which test paths will achieve the same coverage criteria. However, our problem is different in that desired coverage information must be guaranteed for *any* run, rather than attained from a minimal *set* of runs.

Other work has developed the idea of relative coverage in the context of web services [24, 59, 116]. Our work can also facilitate customizing coverage metrics to context-dependent targets, but deals with optimizing coverage probe placement rather than how to gather and present this data to users of a web service.

Part II

Analysis

In this part, we describe new postmortem analysis approaches that operate on imperfect failure data from deployed software. These approaches are realized in two analysis systems, `csi-spotlight` and `csi-grissom`. Both of these analysis systems share some common engineering and experimental design features that we describe here.

As in Part I, all experiments used a quad-core Intel Core i5-3450 CPU (3.10 GHz) with 32 GB of RAM running Red Hat Enterprise Linux 6. Subject applications are the same as those used in Part I, Chapter 2, and are mostly from the Software-artifact Infrastructure Repository [55, 154]. Some versions of our subject applications have multiple faults which can be enabled separately; this results in substantially more total build variants for some applications than in previous chapters. In all experiments, we only enabled one fault at a time.

All analyses obtain failure data from applications instrumented with `csi-cc`, using various mechanisms described in Part I of this dissertation. All experiments use Clang/LLVM 3.5 [101] without compiler optimization, as unoptimized code is most conducive to debugging. Our trace data is valid with optimization (and efficient, per Part I results); our analyses produce correct data over optimized programs as well.

Recall from Chapter 2 that `csi-cc` traced data is stored in-memory, and, thus, is left behind in core dumps produced when applications fail unexpected or are forced to abort. We extract this data with a debugger (in our case: `gdb` [1]) based on the embedded metadata pictured in Fig. 2.1. For test cases where core dumps were already produced, we used the generated core file. If a test case produced bad output without crashing, we used the output tracing tool of Horwitz et al. [73] to identify the first character of incorrect output, and forced the application to abort at that point. This infrastructure results in a core dump containing `csi-cc` trace data for each failing test case in the test suite of each fault for each version of each application.

5 ACTIVE NODES, EDGES, AND SLICES

Substantial portions of this chapter are derived from a 2013 conference paper by Ohmann and Liblit [135], and a 2016 journal paper by Ohmann and Liblit [137].

When failures occur in production, detailed postmortem information is invaluable but difficult to obtain. Based on any information available in a production failure report, developers can benefit from postmortem analysis tools that help narrow debugging focus to relevant portions of the failing program code. One major challenge in analyzing post-deployment failures is sparsity in traced information. In fact, our tracing mechanism designs in Part I were motivated by the fact that dense execution tracing is infeasible for deployed applications. There, we took advantage of readily-available information in core memory dumps from failing applications, enhancing this data with lightweight, customizable tracing. However, our techniques were far from complete tracing. Thus, any postmortem analysis technique targeting post-deployment failures must readily adapt to incomplete data in failure reports.

Ideally, a failure report would contain all information necessary to completely reproduce the failing execution. Prior work with symbolic execution derives inputs and/or thread schedules that match a failed execution [35, 36, 42, 78], and some of that work begins with core dumps [153, 178, 193]. However, finding inputs for executions matching failure data is undecidable in general, very computationally expensive, and may present other concerns regarding user privacy (depending on data included in failure reports). Our analyses take a different approach.

In this chapter, we present two analysis techniques that serve as preprocessing tools for debugging based on post-deployment failure reports. These analyses specifically use our lightweight path tracing and program coverage mechanisms presented in Part I, along with stack information available in core dumps. Our first analysis technique determines the set of potentially-executed control-flow graph (CFG) nodes and edges in any failing run that could produce the input failure data. Our second analysis is a unique hybrid program slicing restriction that narrows the set of statements that may have impacted the value of a user-specified set of program variables during any run matching the failure data.

Our analyses are realized in the `csi-spotlight` analysis engine. We use this engine to evaluate the postmortem analysis benefit of each of our tracing mechanisms (originally described in Chapter 2); we examine each independently, and then we examine the "Realistic" pairing described in previous chapters. We find that there are significant trade-offs between tracing overhead and analysis benefit among our tracing mechanisms, but specifically highlight

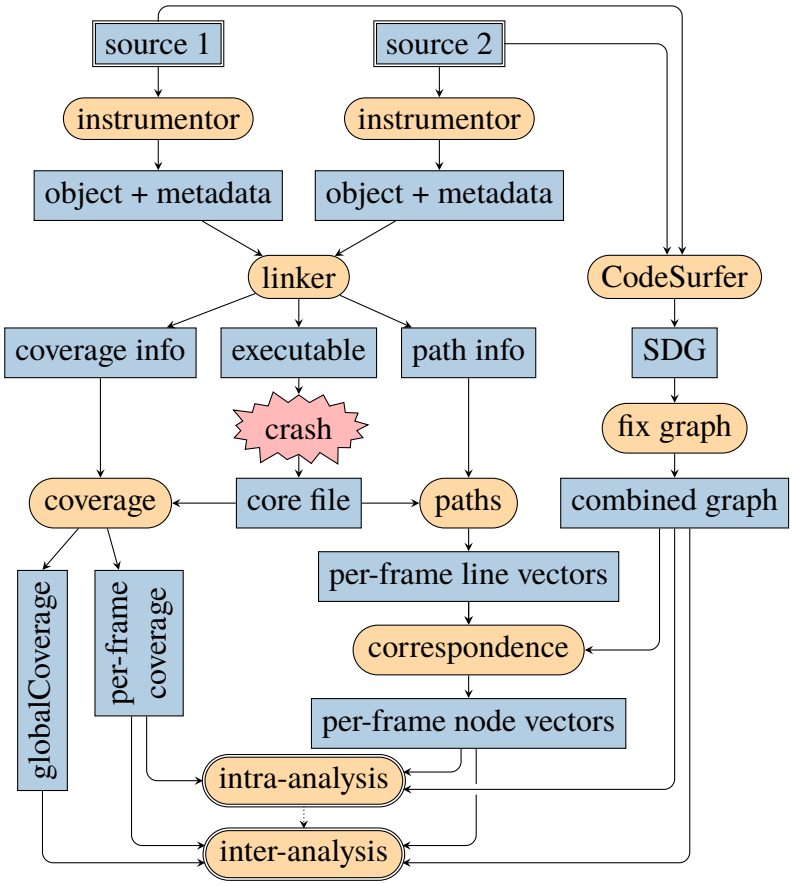


Figure 5.1: Overview of `csi-spotlight`. Sharp-cornered rectangles represent inputs and outputs; rounded rectangles represent computations.

the benefits of our "Realistic" pairing of path tracing and call-site coverage; despite its low cost (per Chapters 2 and 3), this pairing results in substantial reductions in execution ambiguity via our analysis. In our evaluation, we see interprocedural slice reductions as high as 78%, and active node and edge reductions as high as 71% for evaluated applications.

Figure 5.1 shows the `csi-spotlight` architecture, including how we use the artifacts from `csi-cc` instrumentation. Note that the `csi-cc` design from Fig. 2.1 appears unmodified in the top-left corner. Each other feature of this diagram is described in the remainder of this chapter.

5.1 Background

In this section, we provide necessary background for the analyses described in the remainder of this chapter. Specifically, we briefly outline program slicing, which serves as the basis of one of our analyses.

Program slicing with respect to program P, program point n, and variables V determines all other program points and branches in P which may have affected the values of V at n. The original formulation by Weiser [180] proposed the *executable static slice*: a reduction of P that, when executed on any input, preserves the values of V at n. In this chapter, we are concerned with non-executable or *closure slices*, which are the set of statements that might transitively affect the values of V.

Ottenstein and Ottenstein [140] first proposed the program dependence graph (PDG), a useful program representation for slicing. The nodes of a PDG are the same as those in the CFG, and edges represent possible transfer of control or data. A *control-dependence edge* is labeled either *true* or *false* and always has a control predicate or function entry as its source. An edge $n_1 \rightarrow n_2$ means that the result of the conditional at n_1 directly controls whether n_2 executes. (A node may have multiple control-dependence parents in the case of irregular control flow such as **`goto`**, **`break`**, or **`continue`** statements.) A *data-dependence edge* is labeled with a variable v, and has a variable definition at its source and a variable use at its target. A backward static slice over a PDG with slicing criterion (n, V) contains the reverse transitive closure from node n (and its predecessors along incoming data-dependence edges labeled with some $v \in V$), following control-dependence and data-dependence edges.

Our definition of the system dependence graph (SDG), an interprocedural dependence graph, is drawn from Horwitz et al. [72]. The SDG combines all PDGs, and adds a number of new nodes and edges. Each call is now broken out into three types of nodes: a call-site, actual-in, and actual-out nodes. (We treat globals as additional parameters, following Horwitz et al. [72].) A special actual-out node is created for the return value. Each PDG is also augmented with formal-in and formal-out nodes corresponding to formal parameters and the return value, as well as global variables used or defined in the procedure. Interprocedural control-dependence edges are added from each call site to the called procedure's entry node. Interprocedural data-dependence edges are added for all appropriate (actual-in, formal-in) and (formal-out, actual-out) pairs, including the return value. Finally, summary edges from actual-in to actual-out nodes are computed; these represent transitive data dependence summarizing the effects of each procedure call. Details on the computation of these edges can be found in Horwitz et al. [72].

A static slice considers all possible program inputs and execution flows. While debugging, one prefers a slice that is constrained to a particular execution. Korel and Laski [93] first proposed dynamic slicing as a solution to dataflow equations over an execution history. We are interested in closure dynamic slices similar to those proposed by Agrawal and Horgan [6]. The authors propose four variants of dynamic slicing, given a full trace of the CFG nodes executed on a program run. The first variant simply marks all executed nodes, and performs a static slice over that subset of the graph. The second recognizes that each executed node has exactly one control-dependence parent and one reaching definition for each variable used in the statement. Therefore, this variant slices using only dependence edges actually observed as active during the execution. The third approach recognizes that different instances of each node may have different dependence histories. Therefore, this approach creates a *dynamic* dependence graph by replicating each statement each time it occurs in the execution trace, attaching only the active dependence edges for that instance of the statement. Agrawal and Horgan's final approach only replicates nodes with unique transitive dependencies.

Dynamic slicing can be very expensive, potentially requiring data equivalent to a full execution trace. To make matters worse, one must trace all memory accesses due to pointer variables, arrays, and structures to have a completely accurate dynamic slice in the general case [5, 94]. Kamkar et al. [83] reduce the cost of dynamic slicing by requiring only a complete call trace of the analyzed run, while Zhang and Gupta [196] compact the size of the dynamic dependence graph. However, despite these advances, the cost of fully-accurate dynamic slicing remains too high for production use. Venkatesh [176] and Binkley et al. [28] formalize the semantics of program slicing, and discuss the distinctions and orderings among the different types of program slices.

5.2 Analysis Approaches

We develop two analyses that take advantage of our new tracing mechanisms from Part I of this dissertation. Our first analysis restricts the feasible execution set of CFG nodes and edges based on dynamic information from a failing run. Our second analysis builds on a novel static program dependence graph restriction algorithm, which can be used without knowledge of slicing criteria to allow future restricted static program slicing.

For both analyses, we assume that we are given an SDG that is also overlaid with the control-flow edges in each procedure (as the PDG contains all nodes from the CFG by our definition). In the remainder of this chapter, we refer to a graph with both CFG and PDG

edges as a *combined graph*. Both analyses are defined with respect to `csi-cc` data collected as per Chapter 2. For combined graph, G, we assume that this data has been extracted from the core file and is named and organized as follows:

- ***stack*** = $\langle frame_1, frame_2, \ldots, frame_n \rangle$: a representation of the stack at program termination. Each $frame_i$ has the following fields:
  - **coverage** = $\{n_1 : b_1, n_2 : b_2, \ldots, n_{|coverage|} : b_{|coverage|}\}$: a mapping from trace points to Boolean values. All n_i are nodes in the combined graph for $frame_i$'s function, and each b_i indicates the value of the coverage bit (*true* or *false*) corresponding to n_i's trace point from `csi-cc` instrumentation. A particular coverage mechanism contributes entries to **coverage** only if it was actively traced on the failing execution; optimized coverage instrumentation (e.g., via Chapter 4) only has entries for those points that were actually instrumented.
  - **path** = $\langle p_1, p_2, \ldots, p_{|path|} \rangle$: a vector of nodes from $frame_i$'s combined graph. This vector indicates the execution suffix for this frame; in our case, this suffix is gathered via the path tracing mechanism from Chapter 2. This vector always contains at least one entry: either the final crash location (for the innermost frame on the stack) or the location of the still-in-progress call to the next inner frame (for all other frames), even if path tracing was not active in $frame_i$'s tracing scheme.
- ***globalCoverage*** = $\{f_1 : coverage_1, f_2 : coverage_2, \ldots, f_{|globCov|} : coverage_{|globCov|}\}$: a mapping from each function in G to **coverage** maps as defined previously, regardless of the state of the program stack. Boolean values in each $coverage_i$ indicate the value of the global coverage bit (*true* or *false*); that is, they indicate whether the corresponding trace point was ever executed across the entire failing execution. Trace points for a particular coverage mechanism, m, over a particular function, f_i, are contained in *globalCoverage* only if m was in the selected tracing scheme for f_i on the failing run. Hence, if some f_i had no coverage tracing enabled, its $coverage_i$ is empty.

5.2.1 Restriction of Execution Paths

Our first analysis determines the set of CFG nodes and edges which could not have executed given the crashing program stack and tracing data collected. This analysis involves only computing static control-flow graph reachability based on the path and coverage data. As the analysis is very lightweight, it could be used before debugging to filter portions of the program structure shown to a programmer.

Procedure intra_active_nodes(G_f, *path, coverage*)
input: a single-function combined graph G_f
input: a vector of nodes $path = \langle path_1, \ldots, path_{|path|} \rangle$ representing a path in G_f
input: a mapping $coverage = \{n_1 : b_1, n_2 : b_2, \ldots\}$ from trace points in G_f to Boolean
output: a set of nodes *retain*

coverage_reduce(G_f, *coverage*, $path_{|path|}$);
retain = *path*.nodes ∪ cfg_backward_reachable(G_f, $path_1$);

Figure 5.2: Intraprocedural active node analysis

Procedure coverage_reduce(G_f, *coverage, last*)
input: a single-function combined graph G_f
input: a mapping $coverage = \{n_1 : b_1, n_2 : b_2, \ldots\}$ from trace points in G_f to Boolean
input: a node *last* representing the last executed node in f

unusedPoints = $\{n_i$ such that $(n_i : false) \in coverage\}$;
G_f.nodes −= *unusedPoints*;
G_f.nodes = cfg_forward_reachable(G_f, f.entry) ∩ cfg_backward_reachable(G_f, *last*);

Figure 5.3: Coverage reduction

Let P be a program with combined graph G. While the control-flow statements and edges in G represent all possible control flows on any execution of P, they are a static over-approximation of those active in any possible run of P. A full execution trace for a specific run r can precisely yield the set of executed statements and edges in G. With this information, one might reasonably restrict G to a subgraph G_r containing only the CFG nodes and edges active during r, and use the restricted subgraph during debugging or subsequent r-specific analyses.

If the complete execution trace is unavailable, but possible execution flows can be safely over-approximated, then the graph G_r can likewise be approximated, giving a subgraph that is larger than ideal, but still smaller than G. In our case, we have ***stack*** and ***globalCoverage*** as defined above (based on path traces and program coverage data, as described in Chapter 2). This trace data is incomplete and ambiguous: many runs can produce the same data. Our goal is to use this trace data to determine the set of possibly-*active nodes* and the set of possibly-*active edges* on *any* run that is consistent with the trace data.

Figure 5.2 shows the algorithm for intraprocedural active nodes analysis. This algorithm determines all potentially-active nodes, indicated by the output set, *retain*. We first run the procedure coverage_reduce() shown in Fig. 5.3. This procedure eliminates all trace points in the function that were not executed in a particular activation record, as well as any other

input: a whole-program combined graph G
input: a vector of frames *stack*
input: a mapping *globalCoverage* from functions to coverage maps
output: a set of nodes *retain*

forall (f: *coverage*) in *globalCoverage* **do**
    G_f = fragment of G representing function f;
    coverage_reduce(G_f, *coverage*, f.exit);

retain = $\emptyset$;

foreach *frame* in $\langle stack_{|stack|}, \ldots, stack_1 \rangle$ **do**
    G' = temporary copy of G restricted to *frame*.function;
    retain′ = intra_active_nodes(G', *frame*.path, *frame*.coverage);
    retain ∪= *retain*′;
    end_call = call node located at $frame.\text{path}_{|frame.\text{path}|}$;
    calls = {$n \in$ *retain*′ such that n is a call};
    if *end_call* ∉ cfg_backward_reachable(G', $frame.\text{path}_1$) and
      end_call ∉ $\{frame.\text{path}_1, \ldots, frame.\text{path}_{|frame.\text{path}|-1}\}$ **then**
        calls −= *end_call*;
    foreach *call* in *calls* **do**
        retain ∪= nodes interprocedurally forward-reachable from
                call.target without crossing any return edges;

Figure 5.4: Interprocedural active node analysis

program points that could not have executed given that the trace points did not execute. The procedure has two phases. First, it extracts the set of nodes that did not execute according to coverage data, *unusedPoints*, and removes these from G_f. Second, it determines the set of nodes forward-reachable from function entry and the set of nodes backward-reachable from the function's end (in this case, the crash point). Any node not in the intersection of these two sets either (1) only executes if an eliminated trace point executes or (2) only occurs after the crash point. Then, continuing with Fig. 5.2, all nodes in the path trace must be kept, along with any nodes backward-reachable from the first path entry ($path_1$). All other nodes can be eliminated. Though not shown, determination of active edges is identical; the only difference is that we track edges crossed rather than nodes visited for each stage.

The interprocedural algorithm in Fig. 5.4 is largely an extension of the intraprocedural algorithm, with some complexities to deal with stack data. We apply the logic from Fig. 5.3 to every procedure in the entire application, now using ***globalCoverage***. After this, for each frame on the stack, we execute the intraprocedural algorithm over a mutable copy of the

procedure's CFG, G'. This temporary copy is necessary because intra_active_nodes() will remove nodes from G' via a call to coverage_reduce(), and the final result of the interprocedural algorithm must respect the *retain* sets of all invocations of each procedure on the stack (in the case of recursion) and all possible invocations through transitive calls. To incorporate possible execution flows outside the visible stack, we collect the set of possibly-executed calls (excluding the final call, *end_call*, corresponding to the crash location for the relevant stack frame), and determine the set of CFG nodes that may have executed during those calls. This set is determined as all forward-reachable CFG nodes from the entry of each called function. This reachability analysis crosses call edges (to get full interprocedural information) but not return edges (to preserve context-sensitivity). Instead, we assume that the intraprocedural CFG contains an intraprocedural edge corresponding to the call and return for each call site. As with the intraprocedural variant, gathering active edge information is nearly identical. Here, an additional requirement is that we also maintain a set of possible return edges (which, for this analysis, can be derived directly from the set of possible call edges). After all frames have completed, we can eliminate nodes and/or edges which were eliminated for all frames (i.e., were never added to *retain*).

5.2.2 Static Slice Restriction

Our second analysis is a novel technique for program dependence graph (PDG) restriction based on an early dynamic program slicing algorithm originally proposed by Agrawal and Horgan [6]. Note, however, that we are not actually computing a dynamic slice: during analysis, the slicing criteria (program point and variables of interest) may not yet be known. Rather, we restrict the static PDG to respect the failing execution data. This analysis can be a preparatory step for multiple future slice queries for any given slicing criteria.

Let P be a program with combined graph G. As with the CFG in Section 5.2.1, dependence edges in G are a static over-approximation of those active in any possible run of P. Suppose in this case that one knew exactly which control-dependence and data-dependence edges were actually used during a specific run r. One might reasonably restrict G to a dependence subgraph G_r containing only the dependence edges active during r, and use the restricted subgraph during subsequent r-specific analyses. For example, a backward static slice over G_r would yield an r-restricted dynamic slice for any program point of interest. This technique corresponds to approach 2 in Agrawal and Horgan [6].

As in the CFG case, our path traces and program coverage data from Chapter 2 allow us to over-approximate the exact set of dependence edges active in r, yielding a safe over-

input: a single-function combined graph G_f
input: a vector of nodes $path = \langle path_1, \ldots, path_{|path|} \rangle$ representing a path in G_f
input: a mapping $coverage = \{n_1 : b_1, n_2 : b_2, \ldots\}$ from trace points in G_f to Boolean
output: a restricted version of G_f with respect to *path* and *coverage*

coverage_reduce(G_f, *coverage*, $path_{|path|}$);
retain = intra_control_retain(G_f, *path*) ∪ intra_data_retain(G_f, *path*, ∅);
G_f.pdg_edges ∩= *retain*;

Figure 5.5: Intraprocedural dependence graph reduction

approximation of the ideal G_r. Specifically, we compute a *trace-restricted dependence graph* that retains every dependence edge that could possibly have been active in *any* run that is consistent with the trace data.

For our formulations, we are given a combined graph, G, as defined earlier. In Figs. 5.6 to 5.8, "$\rightarrow$" always refers to a control-dependence (not control-flow) edge, while "$\rightarrow_v$" refers to a data-dependence edge defining v. For the high-level descriptions of the algorithms given here, we collapse all actual-in and actual-out nodes into their associated call nodes for ease of presentation.

5.2.2.1 Intraprocedural Restriction

Figure 5.5 shows the overall process of computing intraprocedural PDG restrictions, which proceeds in several phases. This algorithm resembles that in Fig. 5.2 for active CFG nodes, but determining active dependence edges is somewhat more complex. To begin, coverage information is used to prune the reachable nodes in the combined graph per Fig. 5.3, described earlier. Next, we identify the control-dependence and data-dependence edges that must be retained. This process is more complex than simple reachability required for CFG nodes and edges; details for each of control-dependence and data-dependence edges appear in Figs. 5.6 and 5.7, respectively. Lastly, we remove all dependence edges not selected for retention.

Figure 5.6 shows the process for determining the retained set of control-dependence edges. The goal is to identify the immediate control-dependence parent of each node in *path* and each node potentially executed prior to *path*. The vector *unattributed* holds path entries for which the algorithm has yet to determine the most direct controlling node. The outer **foreach** loop walks backward (beginning from the crash point) through the entries in *path*. The inner loop begins with the entry immediately prior to the current node, again walking backward through *path*. During this inner-loop search, if a node is encountered that controls the execution of the outer-loop node, then the control-dependence edge between those nodes was "active" in

```
Function intra_control_retain(G_f, path)
    input: a single-function combined graph G_f
    input: a vector of nodes path = ⟨path_1, ..., path_|path|⟩ representing a path in G_f
    output: a set of nodes retain

    unattributed = path;
    retain = ∅;
    foreach (n, i) in (path_|path|, |path|), ..., (path_1, 1) do
        foreach p in path_{i-1}, ..., path_1 do
            if p → n is a control-dependence edge in G_f then
                retain ∪= {p → n};
                remove slot i from unattributed;
                break;
    reachable = cfg_backward_reachable(G_f, path_1);
    retain ∪= {_ → n such that n ∈ reachable};
    retain ∪= {q → n such that q ∈ reachable ∧ n ∈ unattributed};
```

Figure 5.6: Intraprocedural control-dependence retention

the traced execution, and thus must be retained. Once such a node is found, the outer-loop node has found its directly-controlling conditional; it is removed from *unattributed* and the search for that node ends. After attributing control-dependence parents to as many path entries as possible, the algorithm determines the set of nodes backward-reachable from the first entry in the trace ($path_1$). These nodes have no additional dynamic information: any control-dependence edge from a reachable node could have been active in some run producing this trace. Finally, all remaining unattributed nodes from *path* must retain all incoming control-dependence edges from reachable nodes.

Determining the retained set of data-dependence edges, detailed in Fig. 5.7, follows a similar process, albeit with some additions. Here, each node must determine active data-dependence parents for each variable used at that node. The algorithm first determines which variables must be defined and may be used by each node in the combined graph. For brevity in presentation, *mustDef* and *mayUse* are computed as sets of (node, variable) pairs, but will also be interpreted as mappings from nodes to sets of variables. Each entry of the *unattributed* vector again corresponds to a node from the path trace, but now maps each node to a set that tracks all unattributed variable uses at that node. The *calleeExclusions* parameter is unused by the intraprocedural analysis. The nested loops step backward through *path*, as in control-dependence retention. In this case, the outer loop finishes with a path entry only once it has attributed each variable used (or potentially used, in the case of pointers) at that node.

Function intra_data_retain(G_f, *path, calleeExclusions*)
- **input:** a single-function combined graph G_f
- **input:** a vector of nodes $path = \langle path_1, \ldots, path_{|path|} \rangle$ representing a path in G_f
- **input:** a set of variables *calleeExclusions* unused at call site $path_{|path|}$
- **output:** a set of nodes *retain*

$mustDef = \{(n, v)$ such that $n \in G_f$.nodes $\wedge$ n must define $v\}$;
$mayUse = \{(n, v)$ such that $n \in G_f$.nodes $\wedge$ n may use $v\}$;
$unattributed = \langle mayUse[path_i]$ for i in $1, \ldots, |path| \rangle$;
$unattributed_{|path|}$ −= *calleeExclusions*;
$retain = \emptyset$;
foreach (n, i) in $(path_{|path|}, |path|), \ldots, (path_1, 1)$ **do**
- **foreach** p in $path_{i-1}, \ldots, path_1$ **do**
  - **if** $unattributed_i = \emptyset$ **then break**;
  - **if** $p \rightarrow_v n$ is a data-dependence edge in G_f for some $v \in unattributed_i$ **then**
    - $retain \cup= \{p \rightarrow_v n\}$;
    - **if** $v \in mustDef[p]$ **then**
      - $unattributed_i$ −= $\{v\}$;

$reachable$ = cfg_backward_reachable(G_f, $path_1$);
$retain \cup= \{\_ \rightarrow_v n$ such that $n \in reachable\}$;
forall (n, i) in $(path_1, 1), \ldots, (path_{|path|}, |path|)$ **do**
- $retain \cup= \{q \rightarrow_v n$ such that $q \in reachable \wedge v \in unattributed_i\}$;

Figure 5.7: Intraprocedural data-dependence retention

Otherwise, at each inner loop step, data-dependence edges are retained for any variables not yet attributed. Summary data-dependence edges (from the appropriate actual-in to actual-out nodes) should be added to *retain* whenever a call node is encountered. The path trace does not contain data-flow information. Thus, in the case of pointers with multiple possible variable targets, the analysis cannot be certain which dependence for v was active. Therefore, the algorithm considers a used variable v attributed only if the source must always define v. Lastly, we conservatively add all possible data-dependence edges to unattributed variable uses, much as Fig. 5.6 did for control-dependence edges leading to unattributed nodes.

5.2.2.2 Interprocedural Restriction

Figure 5.8 gives the steps for interprocedural restriction. The formulation closely mirrors the interprocedural slicing method given in Horwitz et al. [72], which is also later used to slice over the restricted dependence graph. First, we use ***globalCoverage*** information to remove

input: a whole-program combined graph G
input: a vector of frames *stack*
input: a mapping *globalCoverage* from functions to coverage maps
output: a restricted version of G with respect to *stack* and *globalCoverage*

forall (f: *coverage*) in *globalCoverage* **do**
    G_f = fragment of G representing function f;
    coverage_reduce(G_f, *coverage*, f.exit);

retain = $\emptyset$;
formals = $\emptyset$;

foreach *frame* in $\langle stack_{|stack|}, \ldots, stack_1 \rangle$ **do**
    G' = temporary copy of G restricted to *frame*.function;
    call = call node located at *frame*.path$_{|frame.\text{path}|}$;
    coverage_reduce(G', *frame*.coverage, *call*);
    actuals = variables for actual arguments for *call*;
    connected = $\{call \rightarrow_v f$ such that $v \in actuals \ \wedge \ f \in formals\}$;
    unconnected = $\{v \in actuals$ such that $\nexists \, call \rightarrow_v \_ \in connected\}$;
    retain $\cup$= *connected*;
    retain$'$ = intra_control_retain(G', *frame*.path) $\cup$ intra_data_retain(G', *frame*.path, *unconnected*);
    retain $\cup$= *retain*$'$;
    formals = $\{formal$ such that $formal \rightarrow \_ \in retain'\}$;

worklist = all call nodes n such that *retain* contains any intraprocedural dependence edge from n;
retain $\cup$= edges interprocedurally backward-reachable from *worklist*
        without crossing any edges from calls to formal-ins;
G.pdg_edges $\cap$= *retain*;

Figure 5.8: Interprocedural dependence graph reduction

unexecuted trace points from each function, as well as any other nodes execution-dependent on those program points.

Next we process each ***stack*** frame, beginning with the crashing function. This phase identifies active dependence edges within and between stack procedures; transitive dependencies from called (and returned) procedures are captured with summary edges. For each frame, we create G', a temporary subgraph of G containing only nodes from the frame's function. As with the active nodes analysis from Fig. 5.4, interprocedural restriction must respect the *retain* sets of all possible invocations of each procedure. We then remove unused trace points via a call to coverage_reduce(). At this point, we need to connect this frame to the previous frame by retaining data-dependence edges from formal-in nodes to actual variables from the call. For the innermost frame, this step has no effect. For other frames, *connected* will contain those edges to formal-in nodes that correspond to (transitively) potentially-used

formals in the previous stack frame; these must be retained. *unconnected* contains any actuals not connected to a useful formal. Note that here the intraprocedural restriction algorithms are used as subroutines. We now use the third parameter to intra_data_retain: the algorithm does not consider unused actuals to be "unattributed," as incoming data-dependence edges for these variables were unused.

The final step of the algorithm retains dependence edges from transitive calls beginning from the stack frames. A *worklist* is populated with all calls not corresponding to the crash point in this frame. All dependence edges backward-reachable in the SDG from the *worklist* nodes (including edges corresponding to function returns but excluding those corresponding to function calls) must be retained. These edges correspond to transitive interprocedural dependencies for previously-returned calls. The algorithm does not need to "re-ascend" to calling procedures because summary edges are included in both phases.

5.2.2.3 Additional Considerations and Relationship to Dynamic Slicing

Slices over a restricted graph, like those of Agrawal and Horgan [6] and Horwitz et al. [72], are *closure slices*. These over-approximate the set of statements that may have affected the variable values at the chosen slice point, but are not necessarily executable or equivalent to the original program.

Unlike Agrawal and Horgan, our dependence graph restriction algorithms are not actually computing dynamic slices: they are not "slicing from" any particular program point. In fact, one way to define our analysis is as partial-trace dynamic slicing from every point along our execution suffix. The choice of static-slice criteria is orthogonal to this restriction. Every static slice taken over the restricted graph is consistent with the trace data, modulo the loss of accuracy (as in Agrawal and Horgan's approach 3) when a node is executed multiple times with different incoming dependence edges. Our dependence graph is static, so these dynamically-distinct nodes are necessarily collapsed into one static node.

It would be possible for our algorithms to unroll all traced paths into the SDG and track individual dependencies. This approach for a full execution trace produces what is known as a dynamic dependence graph, and is equivalent to Agrawal and Horgan's approach 3; our approach would produce a partial-dynamic dependence graph. While it can yield smaller dynamic slices, this approach also makes the SDG significantly larger and more complex to understand. Despite advances in compressing dynamic dependence graphs (e.g., Zhang and Gupta [196] and the final approach by Agrawal and Horgan [6]), graph sizes remain quite large, increasing the time and mental effort for a developer to sift through graph data to find

Table 5.1: Evaluated applications, ordered by size.

Application	Description	Variants	Mean LOC
tcas	Siemens	41	173
schedule2	Siemens	9	373
schedule	Siemens	9	413
print_tokens2	Siemens	10	568
print_tokens	Siemens	7	727
ccrypt	Linux utility	1	5,280
gzip	Linux utility	20	8,114
space	ADL interpreter	34	9,563
sed	Linux utility	31	14,314
flex	Linux utility	53	14,946
grep	Linux utility	19	15,460
gcc	C compiler	1	222,196

a reasonable slice point. Thus, we do not work with dynamic dependence graphs for our analysis results.

None of our tracing mechanisms from Part I track updates to memory locations as would be necessary for fully-accurate interprocedural dynamic slicing [5]. This also means that we restrict static *data* dependencies using only *control-flow* trace data. We accept a potential loss of accuracy that comes with static alias analysis for globals and pointer variables when crossing procedure boundaries.

5.3 Experimental Evaluation

We conducted experiments to assess the effectiveness of our algorithms and, in tandem, the utility of the information from our tracing mechanisms described in Part I. Table 5.1 gives details about our test subjects. These applications are nearly identical to those used to evaluate our tracing mechanisms in Chapter 2, except that we exclude two Siemens subjects (replace and tot_info) that exhibit significant issues in mapping failure data onto our combined graphs. (We describe this process in upcoming Section 5.3.1.2.) As noted in the introduction to this part of the dissertation, some applications have multiple versions and multiple faults which can be enabled separately; the "Variants" column of Table 5.1 counts unique builds across all versions and all available faults. Of the Software-artifact Infrastructure Repository subjects: space contains real faults, sed contains both seeded and real faults, and the remaining subjects

(the Siemens applications, flex, grep, and gzip) contain only seeded faults. ccrypt and gcc are real, released versions with real faults.

Results presented in this section are aggregates across all versions, bugs, and test suites of each application. In general, results vary little among builds of a given application; we note any exceptions below. We also aggregate results for all applications from the Siemens test suite to simplify presentation in figures. These are very small, simple applications, and results indicate that they have similar results for our analyses. We note any exceptions in our textual descriptions. All analysis runs in this chapter complete within 10 minutes, so we omit analysis run times.

5.3.1 Analysis Implementation

This section describes details related to our implementation and evaluation of the analyses described in Section 5.2 of this chapter.

5.3.1.1 Implementation Details

CodeSurfer 2.2p0 [14] produces our combined graphs. These match the SDG description given in Section 5.1, and are overlaid with CFG edges. All CFG nodes (i.e., all nodes except for those representing "hidden" formal and actual parameters such as global variables) have associated source-code location information.

Our analysis implementation follows that given in Section 5.2. However, there we simplified presentation by collapsing both global and local formals and actuals into their associated call node. Formals and actuals are separate nodes in CodeSurfer SDGs, and our analysis treats them separately; thus, retention can distinguish between used and unused formal and actual parameters, the *unconnected* set (Fig. 5.8) is composed of nodes (rather than variables), and summary edges exist from actual-in nodes to actual-out nodes (which are relevant for intraprocedural analysis).

5.3.1.2 Sources of Ambiguity

We encountered a number of sources of ambiguity in our analysis framework. Note that ambiguity in our *results* is expected: our tracing mechanisms from Part I intentionally sacrifice full-trace detail to reduce run-time overhead. However, because we use two different pieces of software (Clang and CodeSurfer) to determine statement locations for path trace entries and coverage trace points, additional ambiguity from minor disagreements in program

```
    if (code != REG && !exp_equiv_p (p->exp, p->exp, 1, 0))

    if (elt) elt = elt->first_same_value;
```

(a) Multiple expressions on a single line

```
if (GET_CODE (op) == CONST_INT
    && width <= HOST_BITS_PER_WIDE_INT && width > 0)

p = lookup (arg1, safe_hash (arg1, GET_MODE (arg1)) % NBUCKETS,
            GET_MODE (arg1));
```

(b) Single statements across multiple lines

Figure 5.9: Examples of matching ambiguity

representations is inevitable. Much of the ambiguity in matching program locations stems from the fact that line numbers are the smallest granularity at which we can reliably match Clang AST nodes to CodeSurfer graph nodes. We also find disagreements in the selection of line numbers to assign to particular program points. Naturally, our matching approach must always be conservative with respect to our analyses to ensure that our results safely under-approximate (but never over-approximate) the optimal reduction we can achieve.

We are generally unable to individually match different expressions occupying the same source line, as Clang and CodeSurfer may break or order the expressions differently. This means that the start, end, and size of expressions can differ, as well as the number of nodes or operations involved. Consider the two code lines shown in Fig. 5.9a. In the first line, the LLVM bitcode contains significantly more instructions than the number of expressions in the CodeSurfer CFG (dereferences, the unary "`!`" operation, etc.). The ordering of the actual parameters in the call to `exp_equiv_p` and evaluation of their expressions need not correspond, so we also cannot count on an ordered many-to-one relationship.

This in-line ambiguity is particularly problematic for path traces and statement coverage data, but also has a small impact on call-site coverage matching. For path traces, we handle the ambiguity by matching each path trace entry with a set of nodes matching the given line number (rather than a single node). This set can be refined based on existing CFG edges to previous or from following entries in the trace; nevertheless, significant ambiguity is common. Specifically, we are never able to distinguish paths through a single line, such as the **`if`** statement given on the second line of Fig. 5.9a. Our analyses from Section 5.2 suffer further from this problem because we often miss out on opportunities to remove a node from the *unattributed* sets (Figs. 5.6 and 5.7) due to not knowing if an assignment definitely executes

on a particular line. Ambiguity due to the matching of LLVM line numbers to CodeSurfer nodes reduces the precision of our analysis in the correspondence and coverage stages of Fig. 5.1. Statement coverage, similarly, cannot always distinguish between multiple basic blocks that occur in the same line. However, as mentioned in Chapter 2, Section 2.2.2, the set of called functions is also recorded for each basic block in our statement coverage metadata; this can often help to distinguish blocks occurring purely within the same line. Call-site coverage is unable to distinguish between calls to the same function on the same line.

Our tools must also grapple with statements that span multiple lines. This is because Clang and CodeSurfer builds do not necessarily agree about whether to assign the line number(s) for a statement or expression to the first line, last line, or (in CodeSurfer's case) all relevant lines. Figure 5.9b shows two examples of statements demonstrating this issue. This issue arises most frequently with conditional expressions, ternary expressions, and calls. Actual parameters (unless they contain other expressions that necessitate their own line number, such as another call) tend to be assigned the line number of the call statement (which is usually either the first or last line of the entire statement). Because of the great uncertainty in matching multi-line expressions, we intentionally introduce ambiguity into the combined graph to safely match Clang's output. We collapse all line numbers within each set of nodes corresponding to a multi-line expression into a single set, which we assign to all nodes of the expression. This change impacts path traces, statement coverage, and call-site coverage. For path traces, this collapsing further increases ambiguity regarding which expression each path trace entry refers to on a particular line. For statement coverage, it necessitates that we only remove nodes for which all trace points corresponding to that line have "*false*" as their coverage bit. For call-site coverage, we must ensure that call nodes are only removed if they are either the only call site on the line (for indirect calls), or the only call to the specified function (for direct calls).

Other intricate issues also necessitate some further minor introduction of ambiguity into the combined graph. For example, LLVM 3.5 assigns the line number of the close of the statement block (i.e., the closing "`}`" brace) to the conditional of a **`do`**–**`while`** statement. Naturally, this character has no semantic value, so it will not appear in the CodeSurfer graph. Thus, we must include all line numbers up to the most recent statement within the loop in the set of line numbers for the loop-guard conditional. These changes, as well as changes necessary for multi-line expressions, are referred to as the fix graph stage in Fig. 5.1. Finally, in flex, gcc, and one version of grep, we had to modify one source-code line by eliminating a line break at the start of an **`if`** statement that otherwise caused irreconcilable disagreement between Clang and CodeSurfer line numbers.

Unfortunately, this ambiguity is quite common in the applications we examined. In fact, all four code lines from Fig. 5.9 are taken from one single function of gcc. Nevertheless, our analysis results show that we can significantly reduce ambiguity in the failing execution despite this ambiguity in our analysis framework.

5.3.2 Analysis Effectiveness

We evaluated the benefit of our analyses described in this chapter. We used core dumps produced by each failing test case for each faulty application variant, and ran each of our three analyses: active nodes, edges, and slices. For intraprocedural results, we ran each analysis over every function on the stack that has at least one ambiguous branch on a path from function entry to the crash point.

We also varied which tracing mechanisms were enabled. In all cases, the tracing mechanisms specified were enabled for all functions in each application, without coverage optimization per Chapter 4. Path traces are purely intraprocedural tracing; therefore, restricting tracing to functions appearing in crashing stacks (the "Realistic" configuration from Chapters 2 and 3) does not result in any loss of information. As mentioned in Chapter 2, gcc has thirteen functions with more than 2^{63} acyclic paths that cannot be instrumented for path tracing; however, all program coverage remains available for these functions. Due to memory constraints, we were unable to gather complete analysis results for gcc. Specifically, we excluded six gcc functions that we could not analyze on a 32 GB RAM machine with our memory-based analysis: `assign_parms`, `expand_expr`, `fold`, `fold_truthop`, `rest_of_compilation`, and `yyparse`. gcc's large size also prevented us from constructing the whole-program combined graph. Therefore, we omit interprocedural analysis results for gcc.

5.3.2.1 Restriction of Execution Paths

The restriction algorithms in Section 5.2.1 can eliminate CFG nodes and edges that could not possibly have been active during a given run. Figures 5.10 and 5.11 show results (intraprocedural and interprocedural, respectively) for "active edges" as a percentage of all CFG edges. We show only results for edges here, as "active nodes" show very similar patterns. These numbers are relative to context-sensitive, stack-constrained, backward reachability, indicated by the "None" bar. For the intraprocedural analysis, we count backward-reachable nodes and edges from the frame's crash point. For the interprocedural analysis, we work back from the crash point of the innermost stack frame. Smaller numbers here are better: values

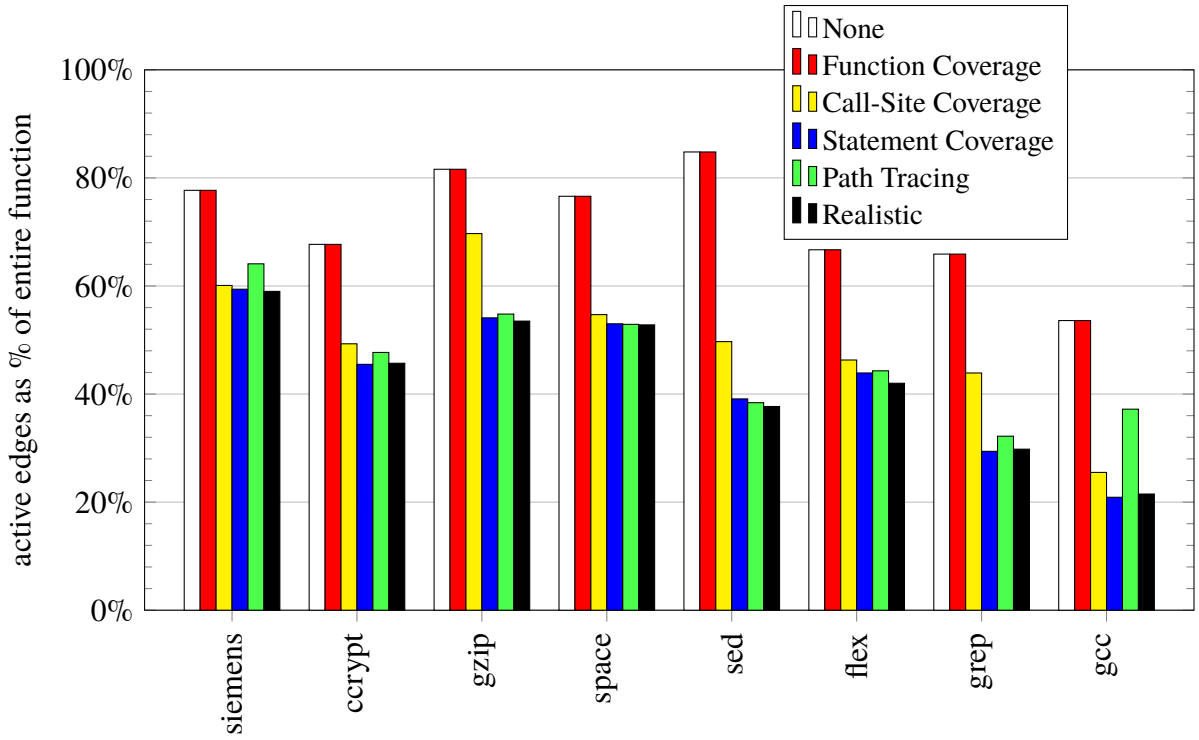


Figure 5.10: Intraprocedural active edges

close to the "None" result indicate little reduction, while values closer to 0% mean that our analysis eliminated many inactive edges.

Figure 5.10 shows intraprocedural results. Here, we measured the set of possibly-active CFG edges as a percentage of all CFG edges in each stack frame's function. As previously stated, we ran our analysis over every function on the stack that has at least one ambiguous branch on a path from function entry to the crash point. We first measured the reductions for each tracing mechanism individually. Reductions for the smaller Siemens applications are modest across all tracing mechanisms. Execution ambiguity is generally very low for these applications due to the small size of most functions. Results for larger applications, however, are much more impressive. Note that, by our analysis formulation from Section 5.2.1, function coverage does not contribute to intraprocedural analysis (as all functions in the crashing stack are clearly already executing). Our other three tracing mechanisms all perform well, though complete statement coverage obtains the best results for all applications except sed (which achieves a 1% better reduction with path tracing). The high cost of statement coverage, however, motivates consideration of the "Realistic" scheme from Chapters 2 and 3: the combination of path traces and call-site coverage. Results indicate that the two mechanisms are complementary. For example, gcc sees an additional 20% reduction due to the combination.

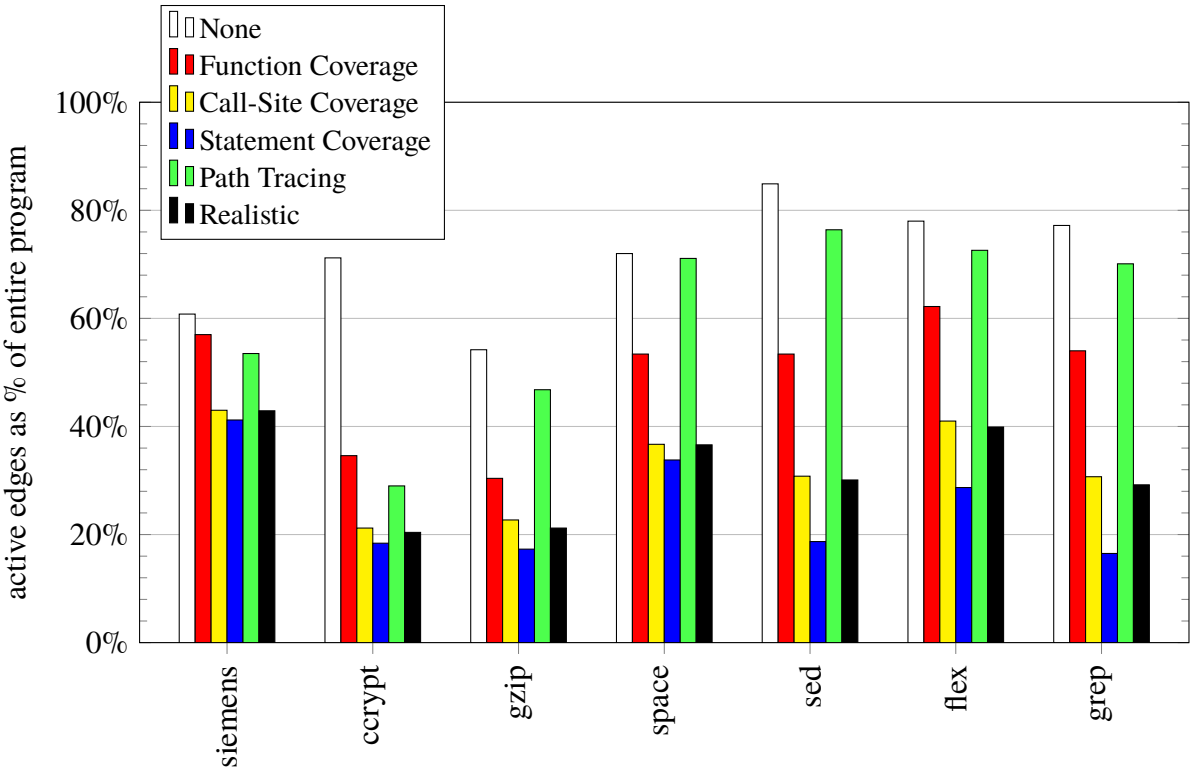


Figure 5.11: Interprocedural active edges

The realistic configuration is the optimal choice for all but two of the applications (ccrypt and gcc), and averages 41% reduction, with a maximum reduction of 54% (sed), across the larger applications. It achieves these reductions at significantly less tracing cost than statement coverage, per results from Chapter 3, Section 3.3.

Figure 5.11 shows interprocedural results for active edges. Here, the plot shows the set of possibly-active CFG edges as a percentage of all edges in the entire program (excluding external libraries). Again, reductions for smaller applications are modest. There are, however, some exceptions: one version of print_tokens sees an average 46% interprocedural reduction in active edges. However, in general, as with intraprocedural analysis, execution ambiguity is very low, often with only one stack frame besides `main`. Considering the larger applications, however, results are again much more impressive. Some patterns are clear. Coverage data is the dominating factor for interprocedural analysis. This is not surprising: coverage maintains global-scope information not available to path tracing. However, path traces do still contribute to the reduction for our "Realistic" result in all larger applications except space (which generally has very little ambiguity *within* the failing stack). Comparing our coverage mechanisms, it is clear that the coarse-grained global information provided by function coverage often

still leaves a great deal of execution ambiguity that can be rectified by the finer-grained coverage mechanisms. Statement coverage provides a clear benefit for some applications (e.g., flex, grep, and sed), but, for others, the reductions obtained for the inexpensive "Realistic" configuration are comparable.

Overall, results for the combination of path tracing and call-site coverage are quite impressive, with average reductions as high as 71% (ccrypt, interprocedural). Most applications are uniform across versions, but versions of sed have active edge reductions ranging from 38%–66% in the intraprocedural case, and 51%–85% in the interprocedural case. space versions vary from 9%–56% intraprocedurally and 6%–54% interprocedurally. In general, for complex applications, we find that a stack trace alone leaves great ambiguity as to which code was active. Our lightweight tracing mechanisms from Part I of this dissertation significantly reduce this ambiguity with negligible impact on performance. Furthermore, our analysis is able to derive substantial reductions in active nodes and edges, despite the imperfect data traced by our mechanisms.

5.3.2.2 Static Slice Reduction

Our SDG restriction algorithms from Section 5.2.2 can compute a restriction of the static SDG based on traced data. Per Section 5.2.2.3, the computed restriction is independent of (and can be computed prior to selecting) the slicing criteria. For our evaluation, we compute interprocedural static slices backward from the crash point in the innermost stack frame; intraprocedural slices work backward from the crash point in each function in the crash stack. In both cases, we slice for all variables used at the slice point. All interprocedural slices are callstack-sensitive [29, 73, 96], using the approach from Krinke [96]. Results show, to a large extent, very similar patterns to those for active edges.

Intraprocedural slicing results are shown in Fig. 5.12, where bars indicate the slice size for each stack frame's function as a percentage of all PDG nodes that have a source-code representation (i.e., that map to a line number). Note that a line can have more than one node. For example, for a call with multiple parameters, we count each actual parameter separately, as some may be included in the slice while others are not. The "None" bar represents the slice size for a backward static slice from the crashing location in each active stack frame without the benefit of our dependence graph restriction. Smaller numbers are again better: values close to "None" indicate little reduction in slice size, while values closer to 0% mean that slices were much smaller with our restriction analysis than without. Smaller applications again see less benefit. However, larger applications show much better results. Considering

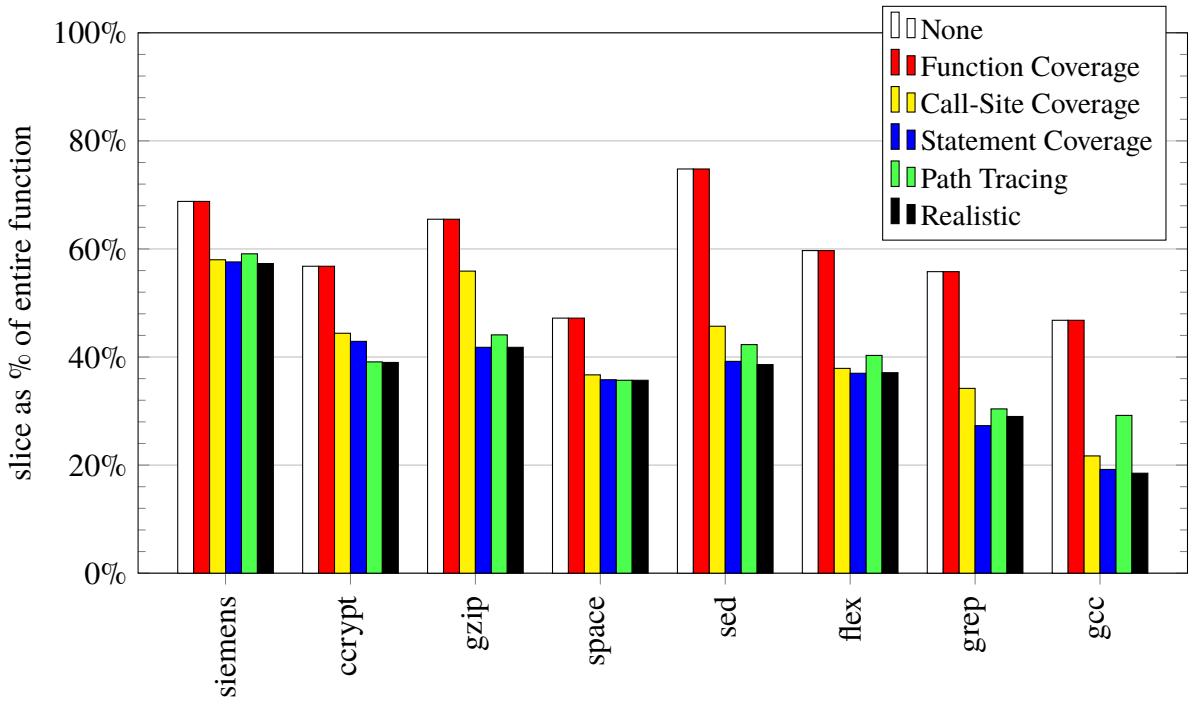


Figure 5.12: Intraprocedural slicing

each tracing mechanism individually, statement coverage again has the strongest results. However, the combination of path tracing and call-site coverage again performs extremely well, with the best results for all applications except grep (for which it lags behind statement coverage by a mere 1.5% reduction). Intraprocedural slice reductions average 40% across all larger applications, with a maximum reduction of 53% for gcc, the largest application in our experiments.

Figure 5.13 shows interprocedural slicing results. Here, bars indicate the slice size as a percentage of all SDG nodes in the entire program (excluding external libraries). The "None" bar represents the slice size for a callstack-sensitive backward slice from the crashing location without the benefit of our dependence graph restriction.

As in all previous cases, the Siemens applications see only a small benefit, though there are some exceptions: one version of schedule has an average interprocedural slice reduction of 73%, but the absolute slice sizes in this particular case are small, so the absolute ambiguity is not large. Results improve substantially for larger applications, with interprocedural slice reduction showing better results (53%–78% reduction, "Realistic" trace data) than the intraprocedural variant. Coverage data is again the dominating factor in interprocedural analysis. Here, however, the benefit of statement coverage over the combination of call-site

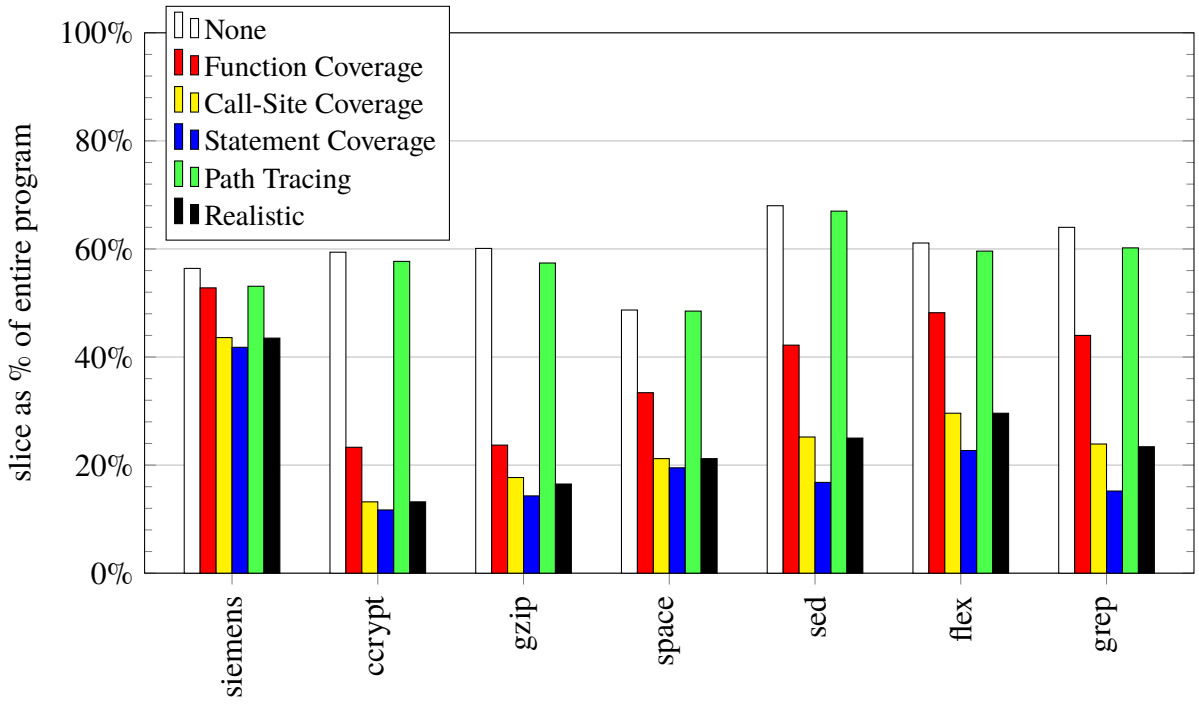


Figure 5.13: Interprocedural slicing

coverage and path traces is much less pronounced. Even for grep (the application with the largest discrepancy), statement coverage further reduces slice size by only 14% beyond the realistic tracing scheme. Overall, the realistic scheme obtains the majority of the benefit of statement coverage at a much lower cost. space is the only larger application with highly varied results, ranging from 6%–46% intraprocedurally and 9%–62% interprocedurally.

Overall, the results for path traces and call-site coverage are again very impressive, especially interprocedurally. Even for flex, the worst among the large applications, our approach cuts interprocedural slice sizes in half. The best results, for ccrypt, show a 78% reduction.

5.3.3 Discussion

Overall, our results indicate that our `csi-spotlight` analysis techniques can substantially reduce ambiguity in the failing program's execution using the lightweight, imperfect trace data from Part I of this dissertation. Our slice restriction results are particularly interesting, in that we are able to substantially reduce ambiguity in a program's data flow using only inexpensive control-flow trace data.

There are clearly substantial trade-offs regarding coverage in our domain. While function coverage has unmeasurably small overhead, its postmortem analysis benefit is often significantly smaller than other options. Statement coverage comes at a high overhead cost, but is useful where its cost can be tolerated. Call-site coverage provides most of the benefit of statement coverage at significantly less cost; thus, it is likely the best choice in many real-world deployed scenarios.

Recall that we previously hypothesized (beginning in Chapter 2) that path traces and call-site coverage data would complement one another, as path traces densely trace near the failure point within the active stack, while call-site coverage focuses tracing on calls: generally the most significant point of ambiguity in interprocedural analysis. Our results in this chapter support this hypothesis, and we see substantial reduction from this scheme, despite its low tracing overhead cost. For interprocedural slice reductions, our most substantial result (78% reduction for ccrypt) comes at a cost of only one half percent of execution time overhead (per "Realistic" results in Chapter 3, Fig. 3.2). On average, we see an interprocedural slice reduction of 63% for the larger (non-Siemens) applications by pairing path tracing and call-site coverage; this reduction comes at an average run-time overhead of just 2.0%. Note that path traces have the additional benefit of providing a detailed (though incomplete) partial trace leading up to the point of failure. We further address the important question of *presentation* for trace and analysis data in Chapter 7.

Overall, our goal was to show that postmortem analysis can adapt to (and significantly benefit from) tracing mechanisms with low enough cost for production use. We have now found success in both aspects of this goal via lightweight, customizable tracing and postmortem analysis that adjusts to various levels of detail in postmortem data. Our "Realistic" scheme proved widely applicable. However, if higher execution overheads can be tolerated, statement coverage proved helpful for many applications. If execution constraints are tightened, simple function coverage can still yield significant benefit in many scenarios.

5.4 Threats to Validity

As in previous chapters, we attempted to gather fair and generalizable results, but have not formally proven the correctness of our analysis implementations. Here, we discuss threats to the validity of our results, and measures taken to mitigate these risks.

5.4.1 Threats to Internal Validity

As stated in previous chapters, bugs in our algorithm design or software implementation could impact the correctness or accuracy of analysis results. We hand-verified a selection of analysis results over our smallest applications, and added basic checks for consistency (e.g., agreement between intraprocedural and interprocedural results).

Section 5.3.1.2 notes changes we required in order to match static information between the two tools we used for our analysis experiments. The intentional addition of this ambiguity into our combined graphs makes our results a safe over-approximation of the optimal result, but could impact the relationship between the results of different tracing methods. In particular, it may have a larger effect on some tracing methods when compared with others. From a subjective and cursory inspection, this ambiguity seems to impact our path traces most significantly.

5.4.2 Threats to External Validity

We use the same subjects as experiments from previous chapters. As before, our results may not generalize to all deployed software, and particularly to software written in modern object-oriented languages (since all of our applications are written in C).

Many applications had seeded faults, raising typical concerns as to whether such faults are realistic [144]. All applications include test suites with both failing and successful runs; we distinguished these runs by comparing the result with that produced by a non-buggy reference version of the same application. In real deployed applications, it may be more difficult to identify failures or to obtain failing core dumps (either due to not recognizing failures until later in execution, or due to security concerns). While this difference does not directly impact the utility of our traced information, it could make it more difficult for a developer to select appropriate functions for instrumentation/tracing, or impact the "distance" between the failure and the fault. Longer fault-propagation distances likely *increase* the benefit of traced information and analyses that can handle imperfect failure data, but the overall impact is not clear from our lab experimental setting.

5.5 Related Work

Several prior efforts use symbolic execution to reconstruct execution information from incomplete failure data [35, 36, 42, 78, 79, 153, 193]. Replay via symbolic execution can be

very expensive and is undecidable in the general case; further, our analyses expect only very limited trace data from a failing run, which makes full replay much more challenging. Our analysis goal is also different: we present data valid for *any* execution producing matching failure data, whereas replay techniques derive inputs for *one* complete execution that produces matching data. We see related work on symbolic execution based on core dumps as possible beneficiaries of the restriction analyses we perform.

Clause and Orso [39] track environment interactions for replay and minimization of failing executions. While we do not incorporate environment trace data, this approach may provide additional valuable sources of information that could be used in conjunction with the analyses described here. Manevich et al. [112] use backward dataflow analysis to reproduce failing executions based on only a failure location and typestate information regarding the failure. While efficient, this approach is geared toward solving specific typestate problems with simple types (e.g., tracking `NULL` values for null-pointer dereferences). Our approach uses denser information, but targets a wider range of failures: anything that can be made to dump core.

Gupta et al. [67] compute slices within a debugger; ordered break points and call/return traces restrict the possible paths taken. While Gupta et al. focus on interactive debugging, our approach is intended for deployed applications. This focus imposes different requirements, leading to different solutions. We expect run-time tracing with minimal overhead relative to a completely uninstrumented application, not merely relative to an application running in an interactive debugger. Gupta et al. use complete break-point and call/return traces, while we have only the bounded trace data from Part I of this dissertation. Takada et al. [163] offer near-dynamic slicing by tracking each variable's most recent writer. All of our tracing mechanisms gather *control-flow* data; in the presence of pointers and arrays, lightweight dynamic data-dependence tracing in the style of Takada et al. could be a useful addition. However, dataflow tracing is expensive, and would need to be adapted for deployed use: the authors report a 3.4× increase in execution time. Call-mark slicing [129] marks calls that execute during a given run, then uses this information to prune possible execution paths, thereby shrinking static slices. The first phase of our interprocedural slice restriction algorithm uses a similar strategy. However, we support a wider range of trace data: global coverage information as well as segregated coverage and path trace information for each stack frame.

6 ANSWERING CONTROL-FLOW QUERIES

Substantial portions of this chapter are derived from a 2015 workshop paper by Ohmann et al. [132] and a 2017 conference paper by Ohmann et al. [130].

When programs crash at user sites, traced data inevitably results in an incomplete picture of any failing execution; tracing must balance run-time costs with benefits for debugging and postmortem analysis. This incompleteness is present not only for our tracing mechanisms from Part I of this dissertation, but for other post-deployment monitoring techniques as well [25, 44, 85, 135, 139, 143, 170].

While our analyses from Chapter 5 proved effective in reducing execution ambiguity and slice sizes based on `csi-cc` trace data, they cannot directly take advantage of other constraints from existing and future tracing tools. Furthermore, the active nodes, edges, and slicing analyses tackle specific analysis tasks, and cannot answer general user-or-tool-specified *queries* about the failing execution. However, obtaining answers to questions about the failing run is a key part of debugging. LaToza and Myers [99] find that developers commonly ask *reachability* questions, often based entirely on the program's control flow. For example, developers asked about possible transitive function callers/callees and possible calling contexts. Additional cases might arise when debugging deployed applications. If developers suspect missed initialization, they might ask, "Is it possible that `init()` did not run?" or, "Could `x` have been used here before being initialized there?" Developers might also ask whether an application has set the appropriate privilege level, locked appropriate resources, or opened appropriate streams. Some queries may be non-interactive (e.g., a batch analysis job that runs overnight), similar to the analyses from Chapter 5. Such analyses could allow time for precise answers, since users are not directly waiting for an answer to a single query. Other questions may be interactive (e.g., during an active debugging session), and require very fast answers.

In this chapter, we develop an analysis framework that allows developers to ask yes/no control-flow questions (e.g., may a particular statement have executed on the failing run?), and answers whether the query is *Possible* or *Impossible* based on failure report data. We formalize and evaluate three different underlying solvers that present trade-offs in expressiveness for failure constraints, precision in analysis results, and scalability. The first of these is very expressive and precise, and accepts any failure constraints or user queries that can be defined as symbolic visibly-pushdown automata (s-VPA) [47]. However, this solver has substantial analysis costs for large programs and/or failure reports. Our second solver is based on symbolic finite automata (s-FA) [174, 175], and supports constraints and queries defined as regular

languages. For our final solver, we define a new class of subregular languages, the unreliable trace languages, that are particularly suited to answering control-flow queries in polynomial time. We also describe encodings for our `csi-cc` trace data for each of our solvers, along with various other common features of failure reports, including stack traces and call sequences.

Our framework is realized in the `csi-grissom` analysis engine. We evaluate our techniques by extracting best-effort program coverage data (labeling each program statement as "Yes," "No," or "Maybe") based on different granularities of `csi-cc` failure report data. Experimental evaluation shows that there are substantial trade-offs in analysis efficiency and precision. Notably, we find that we can answer a broad class of queries remarkably efficiently when encoding all failure constraints and user queries as unreliable trace languages.

6.1 Example

This section describes the structure of `csi-grissom` using the illustrative example given in Fig. 6.1. Section 6.3 defines each shown element more formally.

The left side of Fig. 6.1 shows a source code skeleton and its corresponding control-flow graph (CFG). This is a somewhat more complex example than that used in previous chapters. Here, each node refers to a statement in the program, solid lines refer to intraprocedural control flow, dotted lines indicate calls, and dashed lines denote returns. Call and return edges are labeled with the call site and the called procedure. The CFG includes three procedures: `main` consists of nodes $\{A, B, C, D, E, F, G\}$, `foo` consists of $\{H, I, J, K, L\}$, and `bar` consists of $\{M, N\}$. Upon receiving a failure report like the one at the top of Fig. 6.1, we allow a developer to pose queries like the one shown on the right of Fig. 6.1.

The example failure report contains two elements: a failing stack trace (the program crashed at statement N in function `bar`, which was called from function `foo` at statement J, etc.), and a single log message from the failing run. For the purposes of this example, assume that we have statically determined that the message "Processed" could only be printed by either statement B or C. Note that our system is far less bound to `csi-cc` data specifically, in comparison to our analyses from Chapter 5. In Section 6.6, we discuss how to encode the elements shown in Fig. 6.1, the various types of `csi-cc` trace data from Part I of this dissertation, and other common failure report elements.

The right end of Fig. 6.1 shows a user query as prose. We formally define our class of queries in Section 6.3. In brief, we allow any control-flow query that can be expressed as a visibly-pushdown language [12]. Our system answers the query with respect to the provided

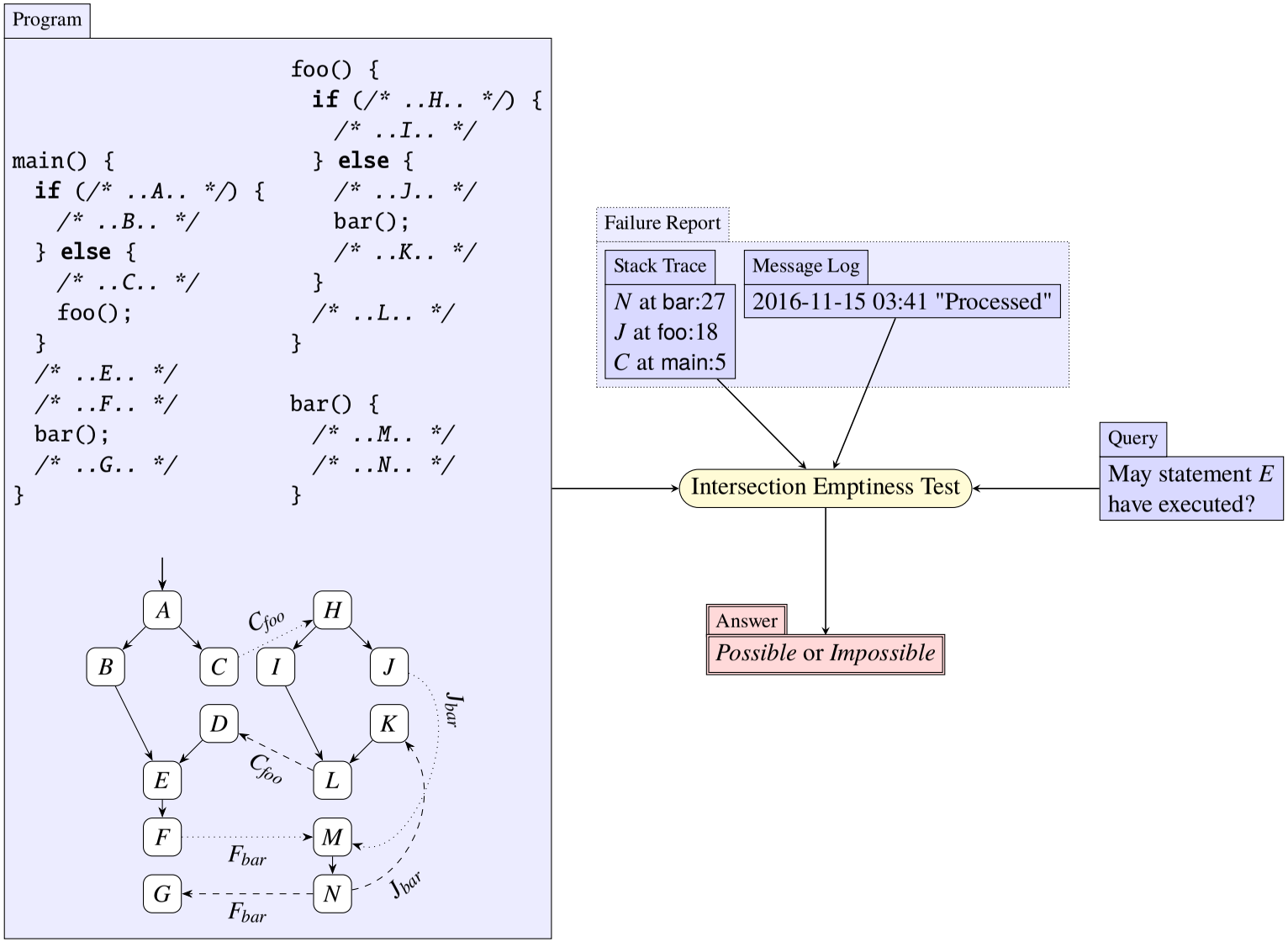


Figure 6.1: Overall system architecture and concrete example. Example inputs consisting of a program, failure report, and query flow into one of our main analysis algorithms, yielding an answer consisting of a single possible/impossible judgment.

CFG and failure report. That is, queries implicitly begin with: "On at least one path through the CFG consistent with the failure report, ..."

In each of our approaches, we first encode the CFG, each failure report element, and the query as formal languages whose member strings correspond to sequences of statements and call/return edge labels from the CFG. If the intersection of all of these languages is non-empty, then there exists such a sequence that (1) could occur in the CFG, (2) is consistent with the entire failure report, and (3) satisfies the query. Thus, we respond that the query is *Possible*. Otherwise, we answer *Impossible*. An imprecise solver may always answer *Possible* to a query. In Fig. 6.1, a precise solver should answer *Impossible*, as there is no path through the CFG that passes through node E before resulting in the final crashing stack. The example contains neither loops nor recursion, and statement E must always follow the return from the in-progress call to `foo` seen in the stack trace. Thus, a precise solver will ensure that the language of the CFG respects calling context by matching labels on call and return edges (simulating a program stack).

Our three analysis approaches in this chapter vary in the machinery used to implement this high-level strategy. Specifically, the languages for each element are encoded differently by each technique, and, therefore, the methods of computing intersections and checking for language emptiness differ as well.

6.2 Background and Definitions

In this section, we provide necessary background and definitions for the approaches described in the remainder of this chapter. We first provide formal definitions for CFGs and program traces, which we use to define our context-sensitive and context-insensitive query recovery problems. We then review the automata models we use for our first two analysis approaches.

We begin by defining a *tagged alphabet*. Given a set of symbols S, we use $(_S$ to denote the set of open symbols $\{(_s \text{ for all } s \in S\}$ and $)_S$ to denote the set of close symbols $\{)_s \text{ for all } s \in S\}$. The tagged alphabet of S is the set $\widehat{S} = S \cup (_S \cup)_S$.

6.2.1 Control-Flow Graphs and Program Traces

Our definition of a control-flow graph in this chapter is more rigorous than those used in previous chapters. This is because we represent interprocedural flow through calls and returns (unlike the intraprocedural analysis of Chapter 4), and require a formal semantics for program traces (unlike Chapter 5, which built upon existing dependence graph representations).

Definition 6.1. *A control-flow graph (CFG) is a tuple* $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$ *where:*

- N *is a finite set of nodes;*
- n_0 *is the entry node;*
- $\mathbb{L}$ *is a finite set of function names;*
- $E_i \subseteq N \times N$ *is a set of internal (intraprocedural) edges;*
- $E_c \subseteq N \times (N \times \mathbb{L}) \times N$ *is a set of function-call edges, such that for every* $(n, (n_1, \alpha), n') \in E_c$, $n = n_1$*; and*
- $E_r \subseteq N \times (N \times \mathbb{L}) \times N$ *is a set of function-return edges.*

In the following, we say that G operates over N and $\mathbb{L}$. Concretely, all edges in E_c are of the form $(n, (n, \alpha), n')$ denoting that control transfers from node n to n' through a call identified by function α. Notice that the call site is also part of the edge label. Similarly, edges in E_r are of the form $(n, (n_1, \alpha), n')$ where control transfers from node n to n' through a function return from α called from site n_1. In the CFG from Fig. 6.1, edges are annotated with the calling node, subscripted by the called function label from $\mathbb{L}$ for brevity.

Let $E = E_i \cup E_c \cup E_r$ denote the set of all edges. We define the semantics of a control-flow graph G using configurations over $N \times (N \times \mathbb{L})^*$, where G is in configuration $c = (n, \langle l_1 \cdots l_k \rangle)$ if the current node is n and the call stack contains the labels $l_1 \cdots l_k$ where each $l_m \in N \times \mathbb{L}$. The initial configuration is $c_0 = (n_0, \varepsilon)$ where ε is the empty stack. A sequence of edges $\pi = \langle e_1 \cdots e_j \rangle$, where all $e_\_ \in E$, forms a *valid context-sensitive path* in the CFG, G, iff there exists a sequence of configurations $\langle (n_0, \bar{l}_0) \cdots (n_j, \bar{l}_j) \rangle$ such that for every $i \geq 1$:

1. If $e_i \in E_i$ and $e_i = (v, v')$, then $n_{i-1} = v$, $n_i = v'$, and $\bar{l}_{i-1} = \bar{l}_i$;
2. If $e_i \in E_c$ and $e_i = (v, (v, \alpha), v')$, then $n_{i-1} = v$, $n_i = v'$, and $\bar{l}_i = \bar{l}_{i-1} \circ (v, \alpha)$;
3. If $e_i \in E_r$ and $e_i = (v, (v_1, \alpha), v')$, then $n_{i-1} = v$, $n_i = v'$, and $\bar{l}_{i-1} = \bar{l}_i \circ (v_1, \alpha)$.

Informally, a valid context-sensitive path only allows return edges to be triggered for the last unmatched function call. This definition allows paths to terminate at any node reachable from n_0, and in configurations that contain a non-empty stack, i.e., with pending function calls.

A sequence of edges $\pi = \langle e_1 \cdots e_j \rangle$, where all $e_\_ \in E$, forms a *valid context-insensitive path* in G iff there exists a sequence of configurations $\langle n_0 \cdots n_j \rangle$ such that for every $i \geq 1$:

1. If $e_i \in E_i$ and $e_i = (v, v')$, then $n_{i-1} = v$, $n_i = v'$;
2. If $e_i \in E_c$ and $e_i = (v, (v, \alpha), v')$, then $n_{i-1} = v$, $n_i = v'$;
3. If $e_i \in E_r$ and $e_i = (v, (v_1, \alpha), v')$, then $n_{i-1} = v$, $n_i = v'$.

Informally, a valid context-insensitive path does not force function calls and returns to be well-matched, effectively not distinguishing between E_i, E_c, and E_r.

Definition 6.2. *Define the projection* $proj(\langle e_1 \cdots e_j \rangle)$ *of a path as* $\langle n_0\, proj(e_1) \cdots proj(e_j) \rangle$*, where for each edge, e:*

$$proj(e) = \begin{cases} v' & \text{if } e \in E_i \text{ and } e = (v, v') \\ (_{v,\alpha} v' & \text{if } e \in E_c \text{ and } e = (v, (v, \alpha), v') \\ )_{v_1,\alpha} v' & \text{if } e \in E_r \text{ and } e = (v, (v_1, \alpha), v') \end{cases}$$

Informally, the projection of a path is the sequence of CFG nodes, calls, and returns traversed by the path. Formally, it is a sequence over the tagged alphabet $\widehat{N \cup (N \times \mathbb{L})}$.

For the remainder of this chapter, we use $N_{\mathbb{L}}$ to denote the alphabet $N \cup N \times \mathbb{L}$ and $\widehat{N_{\mathbb{L}}}$ to denote the alphabet $\widehat{N \cup (N \times \mathbb{L})}$.

Definition 6.3 (Traces)**.** *Let G be a CFG. Given a valid context-sensitive path* π *in G,* $proj(\pi) \in \widehat{N_{\mathbb{L}}}$ *is a valid context-sensitive trace in G. Given a valid context-insensitive path* π *in G,* $proj(\pi) \in \widehat{N_{\mathbb{L}}}$ *is a valid context-insensitive trace in G.* $L_s(G)$ *and* $L_i(G)$ *denote the set of all context-sensitive and context-insensitive traces in G, respectively.*

The following theorem is immediate from our definitions and shows that every context-sensitive trace is also a context-insensitive trace.

Theorem 6.4. *For every CFG G,* $L_s(G) \subseteq L_i(G)$.

6.2.2 Symbolic Automata

Symbolic visibly-pushdown automata (s-VPAs) describe languages of nested words over large or infinite alphabets [47]. Nested words are linear encodings of words with hierarchical structure, such as traces of procedural programs. An s-VPA operates over a tagged alphabet, $\widehat{\Sigma}$, and manipulates a stack. The s-VPA pushes onto the stack only when reading symbols in $(_\Sigma$, and pops only when reading symbols in $)_\Sigma$.

This partition of symbols is a visibly-pushdown alphabet, since it instructs the automaton on how to manipulate the stack: call symbols push onto the stack, while return symbols pop from the stack (similar to function calls and returns in a program trace). A nested word over the alphabet Σ is a sequence of symbols from $\widehat{\Sigma}^*$. To avoid explicit transition on all possible

symbols, s-VPA transitions carry predicates describing sets of symbols from Σ. The set of predicates must be closed under Boolean operators, and checking satisfiability of a predicate must be decidable [47]. We let $\mathbb{P}_X$ represent the set of predicates in our algebra with free variables X. For example, if our alphabet is the set of integers $\{1, \ldots, 100\}$, the predicate $\psi(x, y) = (x \neq y \wedge x \leq 10 \wedge y \geq 5)$ is in $\mathbb{P}_{x,y}$. In this chapter, we use the algebra where predicates are unions of intervals over integers, and we allow equalities between variables. Concretely, each symbol in our alphabet—e.g., a node in the CFG—is mapped to a unique integer. This algebra has all of the required properties above.

The class of *visibly-pushdown languages* recognized by s-VPAs maintain some of the expressiveness of context-free languages (including the ability to express matched calls and returns), but with important properties of regular languages. In particular, they are closed under intersection [12]. (Recall from our informal problem description in Section 6.1 that we check intersection-emptiness for the languages of our CFG, failure report, and query.)

Definition 6.5. *A symbolic visibly-pushdown automaton is a tuple* $A = (Q, q_0, P, F, \Sigma, \delta_i, \delta_c, \delta_r)$ *where:*

- Q *is a finite set of states,*
- $q_0 \in Q$ *is the initial state,*
- P *is a finite set of stack symbols,*
- $F \subseteq Q$ *is the set of final (i.e., accepting) states,*
- Σ *is a set of input symbols,*
- $\delta_i \subseteq Q \times \mathbb{P}_x \times Q$ *is a finite set of internal transitions,*
- $\delta_c \subseteq Q \times \mathbb{P}_x \times Q \times P$ *is a finite set of call transitions, and*
- $\delta_r \subseteq Q \times \mathbb{P}_{x,y} \times P \times Q$ *is a finite set of return transitions.*

A transition $(q, \varphi, q') \in \delta_i$, where $\varphi(x) \in \mathbb{P}_x$, when reading a symbol a such that $\varphi(a)$ is *true*, starting in state q, updates the state to q'. A transition $(q, \varphi, q', p) \in \delta_c$, where $\varphi(x) \in \mathbb{P}_x$, and $p \in P$, when reading a symbol $(_a$ such that $\varphi(a)$ is *true*, moves from state q to q' and pushes the symbol (p, a) on the stack. A transition $(q, \varphi, p, q') \in \delta_r$, where $\varphi(x, y) \in \mathbb{P}_{x,y}$, is triggered when reading an input $)_b$, starting in state q, and with $(p, a) \in P \times \Sigma$ on top of the stack such that $\varphi(a, b)$ is *true*; the transition pops the top stack element and moves to state q'.

A run of A is a sequence of configurations over $Q \times (P \times \Sigma)^*$, where A is in configuration $c = (q, \langle (p_1, \sigma_1) \ldots (p_k, \sigma_k) \rangle)$ if the current state is q with stack $\langle (p_1, \sigma_1) \ldots (p_k, \sigma_k) \rangle$. The initial configuration is $c_0 = (q_0, \varepsilon)$ where ε is the empty stack. An accepting run for A is defined similarly to valid context-sensitive paths through a CFG. Informally, an accepting run is a sequence of configurations where transitions match return symbols to the most recent

unmatched call symbol, and where all predicates on transitions are satisfiable. A nested word, w, is accepted by A iff there exists an accepting run for w in A. We use $L(A)$ to denote the set of all words accepted by A.

In the figures and descriptions of this chapter, we largely follow the style of D'Antoni and Alur [47], and label transitions "$I\colon \varphi_x$," "$C\colon \varphi_x/p$," and "$R\colon \varphi_{x,y}/p$" for Internal, Call, and Return respectively. Here, φ_x denotes a unary predicate on the input symbol x (for internal and call transitions), while $\varphi_{x,y}$ denotes a binary predicate (for return transitions) with y as the current input symbol and x as the input symbol for the matched call transition. For calls and returns, p is the pushed or popped stack symbol.

An s-VPA that contains only internal transitions is a Symbolic Finite Automaton (s-FA) [174, 175]. An s-FA can still operate over alphabet $\widehat{\Sigma}$ for a set of symbols Σ and not use a stack. An s-FA that operates over a finite alphabet accepts a *regular* language. Regular languages can be recognized by (non-symbolic) finite state automata (FSA). All alphabets in our formulations are finite; thus, we use the terms s-FA and FSA interchangeably when describing our approaches, though our implementation (see Section 6.8) uses FSAs.

6.3 Problem Definition

In this section, we formally define our query recovery problem in two variants (context-sensitive and context-insensitive) which impact the problem complexity and precision of results. We use the word "constraint" to describe a language. A constraint C is s-VPA-definable (resp. s-FA-definable) if there exists an s-VPA (resp. s-FA) A_C such that $L(A_C) = C$. Concretely, we will always provide constraints as effective models such as automata or regular expressions.

The inputs of our problem are:

- G, a control-flow graph
- $\{FP_1, \ldots, FP_n\}$, a set of s-VPA-definable *failure constraints* describing the failure report. These can be given in various forms such as s-VPAs, s-FAs, or regular expressions.
- R, an s-VPA-definable *query constraint* describing a property we want to check against our failure report. For example, per Fig. 6.1, we might want to ask whether the node E was traversed, given our failure report.

Definition 6.6 (Context-sensitive recovery)**.** *Given a control-flow graph G with nodes in N and labels in $\mathbb{L}$, a set of s-VPA-definable* failure constraints $\{FP_1, \ldots, FP_n\}$ *of nested words*

over the alphabet $N_{\mathbb{L}}$, and an s-VPA-definable query constraint *R of nested words over the alphabet $N_{\mathbb{L}}$, the context-sensitive query recovery problem is to check whether*

$$L_s(G) \cap \bigcap_i FP_i \cap R = \emptyset.$$

If we assume that $L_s(G)$ is s-VPA-definable, then the query recovery problem is clearly decidable, as we can use known algorithms [47] to check intersection-emptiness over the provided set of s-VPAs. In Section 6.4, this procedure forms our first analysis approach, coupled with a description of how we encode CFGs as s-VPAs. Sadly, this approach has high computational complexity (as detailed in Section 6.4), and we also consider a friendlier alternative.

Definition 6.7 (Context-insensitive recovery)**.** *Given a control-flow graph G with nodes in N and labels in $\mathbb{L}$, a set of s-FA-definable* failure constraints $\{FP_1, \ldots, FP_n\}$ *of words over the alphabet $\widehat{N_{\mathbb{L}}}$, and an s-FA-definable* query constraint *R of words over the alphabet $\widehat{N_{\mathbb{L}}}$, the context-insensitive query recovery problem is to check whether*

$$L_i(G) \cap \bigcap_i FP_i \cap R = \emptyset.$$

This definition only uses s-FAs. Again, if we assume that $L_i(G)$ is s-FA-definable, then we can clearly solve our problem using known intersection-emptiness algorithms for s-FAs [48]. This constitutes our second analysis approach in Section 6.4, coupled with a description of how we encode CFGs as s-FAs.

As one might expect, any solver for the context-insensitive query recovery problem can be used to provide an unsound but complete algorithm for the context-sensitive problem. Observe that a nested word over the alphabet $N_{\mathbb{L}}$ is also a word over the alphabet $\widehat{N_{\mathbb{L}}}$. The proof then follows naturally from our definitions and Theorem 6.4:

Theorem 6.8. *Given*

- *a control-flow graph G with nodes in N and labels in $\mathbb{L}$,*
- *a set of s-VPA-definable* failure constraints $\{FP_1, \ldots, FP_n\}$ *of nested words over the alphabet $N_{\mathbb{L}}$, and*
- *an s-VPA-definable* query constraint *R of nested words over the alphabet $N_{\mathbb{L}}$,*

if $\{FP'_1, \ldots, FP'_n\}$ *and* R' *are s-FA-definable languages of words over the alphabet* $\widehat{N_{\mathbb{L}}}$*, such that for all* i*,* $FP_i \subseteq FP'_i$ *and* $R \subseteq R'$

$$L_i(G) \cap \bigcap_i FP'_i \cap R' = \emptyset \quad \Longrightarrow \quad L_s(G) \cap \bigcap_i FP_i \cap R = \emptyset$$

Proof. Assume, to demonstrate a contradiction, that $L_i(G) \cap \bigcap_i FP'_i \cap R' = \emptyset$, but $L_s(G) \cap \bigcap_i FP_i \cap R \neq \emptyset$. This assumption implies that there exists a word $w \in \widehat{N_{\mathbb{L}}}^*$ such that $w \in L_s(G)$, $w \in FP_i$ for all i, and $w \in R$. Since, for all i, $FP_i \subseteq FP'_i$ and $R \subseteq R'$, we have that $w \in FP'_i$ for all i, and $w \in R'$. From Theorem 6.4, we know that $L_s \subseteq L_i$; therefore $w \in L_i$. Hence, $w \in L_i(G) \cap \bigcap_i FP'_i \cap R'$, which means that this set cannot be empty: a contradiction. □

The choice of including or excluding context-sensitivity in the query recovery problem partly depends on the types of constraints provided. For example, consider a selection of the example queries given at the start of this chapter. Constraints involving whether an application was executing at the appropriate privilege level during a failure would be s-VPA-definable, but not s-FA-definable, if the privilege is established by the calling context (e.g., by a function such as `doPrivileged()`). However, s-FA are perfectly capable of expressing the query "Is it possible that `init()` did not run?" where calling context is not necessary. There are also computational-complexity implications for including context sensitivity, as we note while describing our first two analysis approaches in the following section.

6.4 Two Automata-Based Analyses

Our first two analysis approaches, described in this section, answer queries by directly checking intersection emptiness of automata. Recall from the previous section that, since all other problem input is definable as automata by assumption, we need only describe our encoding of CFGs into each form of automata. We consider two encodings of a control-flow graph, G (from Definition 6.1), as an automaton. The first preserves calling-context sensitivity by encoding G as an s-VPA. Due to the complexity of checking intersection-emptiness for s-VPAs, we also consider a second approach that encodes G as an s-FA. The s-FA encoding cannot ensure calling-context sensitivity, reducing precision but improving scalability. Our encodings are similar to those from prior work in model checking to encode transition systems [10].

We also briefly comment on the computational complexity of our analyses, which is based on our choice of CFG encoding and the encodings of constraints provided. Table 6.1 summarizes our complexity results, including a novel result of polynomial-time complexity

Table 6.1: Complexity of the best known algorithm for computing intersection emptiness for various classes of languages

	s-VPA	s-FA	UTL
Context-Sensitive CFG	EXPTIME	EXPTIME	EXPTIME
Context-Insensitive CFG	EXPTIME	PSPACE [95]	PTIME

for a new subclass of regular languages, the unreliable trace languages, described in the next section of this chapter.

6.4.1 Symbolic Visibly-Pushdown Automata

Given a CFG, $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$, we can perform a straightforward translation of the edges in G to transitions in an s-VPA, $A = (Q, q_0, P, F, \Sigma, \delta_i, \delta_c, \delta_r)$, that accepts nested words over alphabet $N_{\mathbb{L}}$. We begin by introducing final states for each node in the CFG; that is, for every $n \in N$, $n \in Q$ and $n \in F$. Then, we introduce a unique initial state q_0, and

$$q_0 \xrightarrow{I:x=n_0} n_0 \in \delta_i$$

For each $(n_1, n_2) \in E_i$,

$$n_1 \xrightarrow{I:x=n_2} n_2 \in \delta_i$$

For each $(n_1, (n_1, \ell), n_2) \in E_c$,

$$n_2' \in Q$$
$$n_1 \xrightarrow{C:x=(n_1,\ell)/0} n_2' \in \delta_c$$
$$n_2' \xrightarrow{I:x=n_2} n_2 \in \delta_i$$

For each $(n_1, (n_c, \ell), n_2) \in E_r$,

$$n_2' \in Q$$
$$n_1 \xrightarrow{R:x=y=(n_c,\ell)/0} n_2' \in \delta_r$$
$$n_2' \xrightarrow{I:x=n_2} n_2 \in \delta_i$$

States and transitions in A match the nodes and edges of G, plus one additional state and transition for each return site and each procedure entry. These additional transitions allow us to express failure report constraints and queries over procedure entries and return points.

Figure 6.2 shows the s-VPA encoding for the CFG from Fig. 6.1. Note that our construction has a somewhat unusual translation for stack symbols: our CFG automaton has exactly *one* stack symbol ("0") for the entire program, and instead relies on equality predicates on return

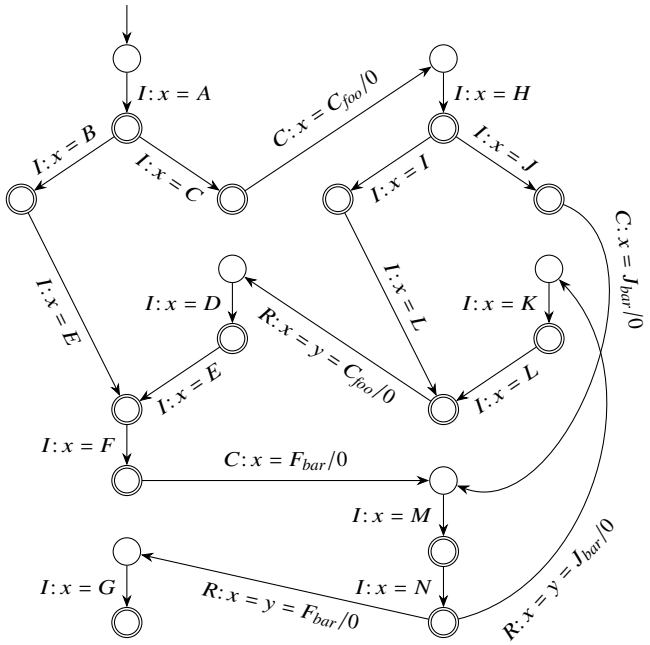


Figure 6.2: s-VPA encoding for the CFG from Fig. 6.1

transitions for calling-context sensitivity. Perhaps the most intuitive encoding would use one stack symbol per call site, simulating addresses pushed onto a program's run-time stack. However, standard product-set constructions for s-VPA intersection [48] can generate an exponential increase in stack symbols; we briefly examined this intuitive encoding (see Section 6.8), but found that it was much less efficient in practice. We similarly minimize unique stack symbols when encoding failure report elements; we describe these in Section 6.6.

We have now described how we encode $L_s(G)$ as an s-VPA. Since the context-sensitive query recovery problem from Definition 6.6 requires that all the languages FP_i and R are s-VPA-definable, the problem is now trivially solved; we answer *Impossible* if $L_s(G) \cap \bigcap_i FP_i \cap R = \emptyset$, and *Possible* otherwise. However, this process is expensive. Intersection emptiness for s-VPA can be solved in exponential time. Moreover, since reachability is known to be co-PTIME, it is unlikely that a PSPACE algorithm exists for computing s-VPA intersection emptiness. Table 6.1 thus lists this complexity for any recovery problem using at least one s-VPA.

6.4.2 Symbolic Finite Automata

Given a CFG, $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$, our translation to an s-FA, $A = (Q, q_0, F, \Sigma, \delta_i)$, is again a straightforward enumeration of the edges in G; in this case, however, our translation

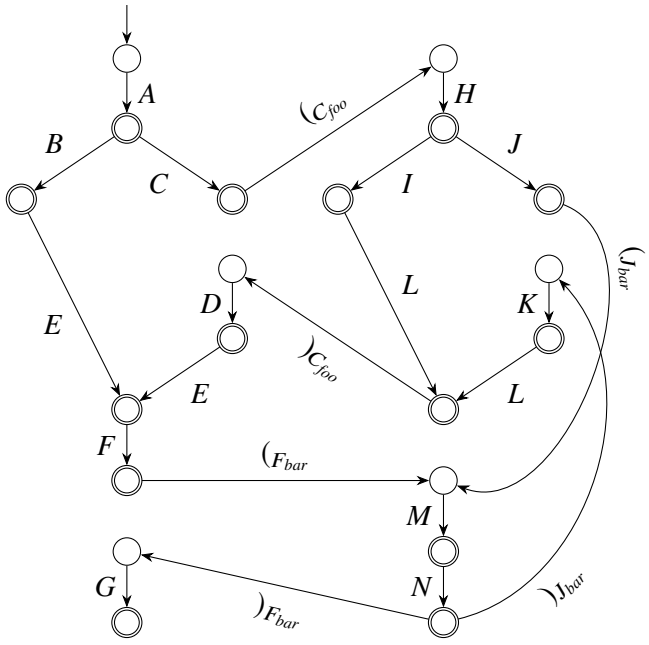


Figure 6.3: FSA encoding for the CFG from Fig. 6.1

results in a calling-context-insensitive representation of G, since we are unable to express call/return pairing relations. Here, A accepts words over $\widehat{N_{\mathbb{L}}}$. Our translation is essentially identical to that given in Section 6.4.1, except that call and return transitions are instead replaced by internal transitions on the corresponding tagged symbol from $\widehat{N_{\mathbb{L}}}$.

Recall that an s-VPA containing only internal transitions over a finite alphabet recognizes a regular language, which can be recognized by a (non-symbolic) FSA. We omit the "I: x =" prefix on transitions in all context-insensitive automata figures. Figure 6.3 shows the FSA encoding for the CFG from Fig. 6.1.

This translation encodes $L_i(G)$ as an s-FA, completing the input for the context-insensitive query recovery problem from Definition 6.7; our analysis answers *Impossible* iff $L_i(G) \cap \bigcap_i FP_i \cap R = \emptyset$. Since our alphabet, $\widehat{N_{\mathbb{L}}}$, is finite, the intersection emptiness problem for s-FAs has complexity PSPACE [95]. In our evaluation, we find that we can answer context-insensitive queries more efficiently, though large numbers of constraints still cause performance issues.

In the following section, we describe a new algorithm that can solve the context-insensitive query recovery problem in polynomial time when all the languages FP_i and R fall in a restricted class of languages called *unreliable trace languages* (UTL). Table 6.1 summarizes these complexity results.

6.5 Unreliable Trace Languages

This section introduces a new class of subregular languages: the *unreliable trace languages*. The class is so named because the accepted languages correspond with traces of observed program points that are "unreliable" in that they do not guarantee that we have seen *every instance* of each observed point. Section 6.5.2 gives a polynomial-time algorithm for checking intersection-emptiness of the language induced from a context-insensitive CFG with n unreliable trace languages. Our approach does not encode the CFG and failure constraints as automata (as we did in Section 6.4); instead, our analysis operates directly over the CFG and tests for intersection-emptiness over context-insensitive traces as in Definition 6.3.

6.5.1 Formal Characterization

For an alphabet Σ, unreliable trace languages are of the class

$$UTL = \left\{ \Sigma^* \, \sigma_1 \, \Sigma^* \, \sigma_2 \, \Sigma^* \, \ldots \, \Sigma^* \, \sigma_n \, \Sigma^* \quad \text{for } n \geq 0 \text{ and such that all } \sigma_i \in \Sigma \right\}$$

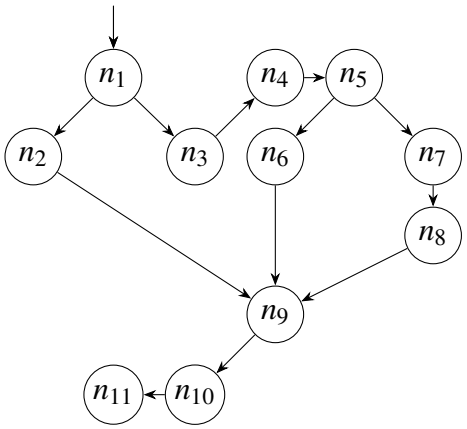
So, for example, the unreliable trace language (defined over $\Sigma = \widehat{N_{\mathbb{L}}}$ from Fig. 6.1)

$$\Sigma^* \, M \, \Sigma^* \, L \, \Sigma^* \, M \, \Sigma^*$$

indicates that accepted strings must contain an M at some point before an L which itself precedes another occurrence of M. However, other occurrences of M or L may occur at any point before, during, or after this sequence. Note that we can represent an unreliable trace language by a vector of symbols from Σ; the above example corresponds to the vector $\langle M, L, M \rangle$. The remainder of this chapter uses this compact vector notation.

The unreliable trace languages are a sub-class of the piecewise-testable languages [146, 159], which can be characterized as any Boolean combination of unreliable trace languages [87]. While very restricted, unreliable trace languages are able to express many possible failure report elements, including program coverage data from `csi-cc` and other sources [85, 139, 143, 170], as well as sampled observations [107]; with some loss of precision, they can also encode many other common failure report elements, including `csi-cc` path traces (see Section 6.6).

Unreliable trace languages are closed under concatenation, but not under other Boolean operations such as negation, complementation, conjunction, and disjunction. Unfortunately, there are constraints that cannot be expressed (even imprecisely) as unreliable trace languages. For example, they cannot express disjunction. Thus, the printed log message from Fig. 6.1 is inexpressible since the CFG node that printed it is ambiguous. The property of PTIME



$n_1.\text{nodes} = \{A\}$ $\quad$ $n_7.\text{nodes} = \{J\}$

$n_2.\text{nodes} = \{B\}$ $\quad$ $n_8.\text{nodes} = \{(_{J_{bar}}\}$

$n_3.\text{nodes} = \{C\}$ $\quad$ $n_9.\text{nodes} = \{D, E, F, K, L, M, N, (_{F_{bar}},)_{C_{foo}},)_{J_{bar}}\}$

$n_4.\text{nodes} = \{(_{C_{foo}}\}$ $\quad$ $n_{10}.\text{nodes} = \{)_{F_{bar}}\}$

$n_5.\text{nodes} = \{H\}$ $\quad$ $n_{11}.\text{nodes} = \{G\}$

$n_6.\text{nodes} = \{I\}$

Figure 6.4: Condensation of the CFG from Fig. 6.1

decidability for our context-insensitive recovery problem (see Definition 6.7) is tightly tied to this language class. In Appendix A.2, we consider two natural extensions of UTL that would allow us to precisely encode a larger class of failure report elements; we prove that both result in NP-hard recovery problems.

6.5.2 Checking Emptiness

We check intersection-emptiness directly over the program's context-insensitive CFG, G (from Definition 6.1), and do not encode $L_i(G)$ using automata. Our input consists of:

- $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$, a control-flow graph;
- $crash \in N$, the stopping point in the innermost stack frame from the failing run; and
- $constraints = \langle c_1, \ldots, c_m \rangle$, a vector of UTL constraints where each c_i is a vector over $\widehat{N_{\mathbb{L}}}$. These constraints correspond to each FP_i and the query, R, from our problem input in Section 6.3.

We answer *Possible* if there exists a context-insensitive trace in G, $proj(\pi)$, such that $proj(\pi)_{|proj(\pi)|} = crash$ and each c_i is a subsequence of $proj(\pi)$ (i.e., $L_i(G) \cap \bigcap_i (c_i) \neq \emptyset$). Otherwise, we answer *Impossible*.

We first split all labeled edges in G, adding a new node named from the tagged symbol of the split edge label, forming graph G'. For the example CFG from Fig. 6.1, this results in 6 new nodes: $(_{C_{foo}}$, $(_{F_{bar}}$, $(_{J_{bar}}$, $)_{C_{foo}}$, $)_{F_{bar}}$, and $)_{J_{bar}}$. We then collapse all strongly-connected components of G' to form the condensation of G', *sccG*. Note that *sccG* is always a directed acyclic graph (DAG). Each node of *sccG* is labeled with its set of collapsed nodes. Figure 6.4 shows the condensation for the CFG from Fig. 6.1.

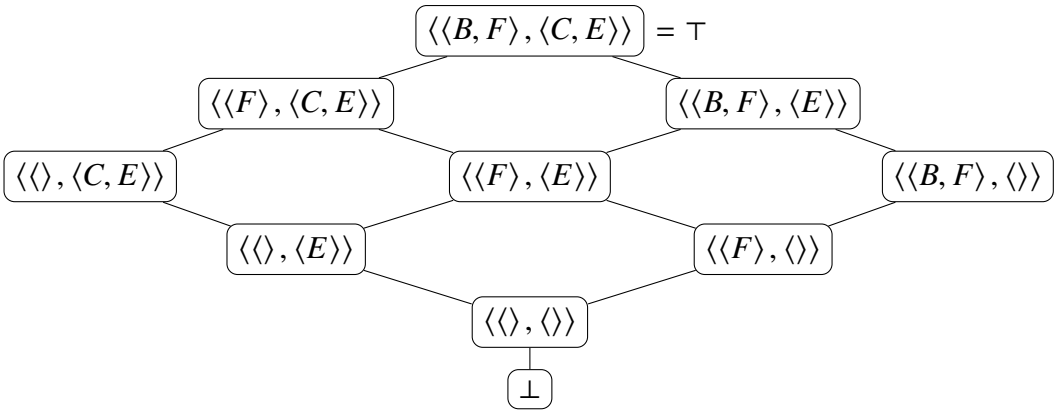


Figure 6.5: Example lattice for UTL emptiness analysis

Intuitively, our goal is to walk forward through G, consuming symbols from each vector in *constraints* in order as we cross the corresponding nodes and edges. If every vector in *constraints* has been completely consumed on some path to *crash*, then we have found a possible execution that is consistent with the program and failure report. Shifting our attention from the original CFG, G, to the condensation graph, *sccG*, has two key effects. First, passing through a nontrivial condensation node *scc* consumes all *scc*.nodes symbols appearing in the unconsumed prefix of any constraint. This technique is valid because each node in a nontrivial strongly-connected component can reach all other nodes in the same component as often as we like. Second, the lack of cycles in *sccG* means that if we do not consume some symbol when passing through the corresponding node, we will never have any future opportunity to do so. Thus, we proceed greedily: consume as many symbols as possible, as early as possible, because we will never get the chance to do so again.

More formally, checking intersection emptiness for $L_i(G)$ from Definition 6.3 (i.e., the context-insensitive CFG language) with a set of unreliable trace language *constraints* (given as vectors per Section 6.5.1) can be cast as a standard iterative data-flow analysis problem over *sccG*, with one caveat related to the definition of our meet ($\sqcap$) operation. The data-flow facts are vectors of vectors over $\widehat{N_{\mathbb{L}}}$, forming a finite lattice where the maximal element $\top$ = *constraints*, and there exists a unique minimal element $\bot$ that indicates that the provided constraints are *Impossible* to satisfy. The second smallest element is the vector of m empty vectors, which indicates that all constraints have been satisfied. Elements are ordered such that $f1 \sqsubseteq f2$ iff for all i, $f1_i$ is a trailing substring of $f2_i$ (i.e., $f2_i = v \mathbin{\shortparallel} f1_i$ for some vector v where "$\shortparallel$" is vector concatenation). Informally, this indicates that fact $f1$ has consumed more of each initial constraint than $f2$ has. Figure 6.5 shows an example lattice for *constraints* = $\langle\langle B, F\rangle, \langle C, E\rangle\rangle$.

Function consume(*scc, inFact*)
  input: *scc*, a node from the condensation graph
  input: *inFact*, the input fact from the predecessors of *scc* as a vector of vectors over $\widehat{N_{\mathbb{L}}}$ (indicating remaining constraints)
  outFact = $\langle\rangle$;
  foreach *constraint* $\in$ *inFact* **do**
    start = 1;
    while $constraint_{start} \in scc$.nodes **do** *start*++ ;
    if *start* > 2 and *scc* is trivial **then return** $\perp$;
    outFact $\shortparallel$= $\left\langle \langle constraint_{start} \ldots constraint_{|constraint|} \rangle \right\rangle$;
  return *outFact*

Figure 6.6: Update constraints satisfied by consuming as much of each constraint as possible

The standard [123] data-flow equations for a node *scc* are

$$in(scc) = \begin{cases} Init & \text{if } n_0 \in scc\text{.nodes} \\ \bigsqcap_{P \in Pred(scc)} out(P) & \text{otherwise} \end{cases}$$

$$out(scc) = F_{scc}(in(scc))$$

Instantiating this for our particular problem,

$$Init = constraints$$

$$f_1 \sqcap f_2 = \text{merge}(f_1,\ f_2)$$

$$F_{scc}(in(scc)) = \text{consume}(scc,\ in(scc))$$

Figure 6.6 shows function consume(). This function iterates through each element of the input vector, marking as many observations as possible as being now satisfied by passing through this strongly-connected component of *sccG*. For example, consume$\left(n_9, \left\langle \langle E \rangle, \langle F, E, F, G \rangle \right\rangle\right)$ returns $\left\langle \langle\rangle, \langle G \rangle \right\rangle$. Trivial strongly-connected components may only consume one element of each constraint vector, because paths through G may only traverse that node once. Note that no later information can ever invalidate prior satisfaction of a constraint.

Figure 6.7 shows function merge(). Given a pair of input facts, this procedure selects the minimal fact via our definition of $\sqsubseteq$ above. That is, it finds the fact that is an element-wise trailing substring of the other. If neither fact qualifies, then the provided constraints are

Function merge(*f1, f2*)
  input: *f1*, a data-flow fact as a vector of vectors over $\widehat{N_{\mathbb{L}}}$
  input: *f2*, a data-flow fact as a vector of vectors over $\widehat{N_{\mathbb{L}}}$
  assert $|f1| = |f2|$;
  if $\forall i \in 1 \ldots |f1|$, $f1_i = \alpha_i \mathbin{+\!\!+} f2_i$ *for some* α_i **then**
    return *f2*
  else if $\forall i \in 1 \ldots |f2|$, $f2_i = \alpha_i \mathbin{+\!\!+} f1_i$ *for some* α_i **then**
    return *f1*
  else return $\bot$;

Figure 6.7: Merge information from predecessor facts

mutually unsatisfiable, and we return $\bot$. Since *sccG* is a DAG, this situation indicates that no possible path through *sccG* can satisfy all constraints, because each symbol from $\widehat{N_{\mathbb{L}}}$ appears in at most one strongly-connected component. For example,

$$\text{merge}\big(\langle\, \langle B, F\rangle, \langle C, E\rangle \,\rangle, \langle\, \langle F\rangle, \langle C, E\rangle \,\rangle\big) = \langle\, \langle F\rangle, \langle C, E\rangle \,\rangle$$
$$\text{merge}\big(\langle\, \langle F\rangle, \langle C, E\rangle \,\rangle, \langle\, \langle B, F\rangle, \langle E\rangle \,\rangle\big) = \bot$$

Note that merge() is not precisely the meet ($\sqcap$) operation for our lattices, like the one shown in Fig. 6.5. There, $\langle\langle F\rangle, \langle C, E\rangle\rangle \sqcap \langle\langle B, F\rangle, \langle E\rangle\rangle = \langle\langle F\rangle, \langle E\rangle\rangle$, while merge() returns $\bot$. We thus stop immediately upon encountering mutually-unsatisfiable facts. This differs from standard data-flow analysis, but it is correct for our problem: we want to recognize discrepancies in constraint satisfaction, rather than abstracting to a state where both are satisfied. Furthermore, merge() is not associative; for example, per Fig. 6.5 and Fig. 6.7,

$$\text{merge}\big(\text{merge}\big(\langle\, \langle\rangle, \langle\rangle \,\rangle, \langle\, \langle F\rangle, \langle\rangle \,\rangle\big), \langle\, \langle\rangle, \langle E\rangle \,\rangle\big) = \text{merge}\big(\langle\, \langle\rangle, \langle\rangle \,\rangle, \langle\, \langle\rangle, \langle E\rangle \,\rangle\big) = \langle\, \langle\rangle, \langle\rangle \,\rangle$$
$$\text{merge}\big(\langle\, \langle\rangle, \langle\rangle \,\rangle, \text{merge}\big(\langle\, \langle F\rangle, \langle\rangle \,\rangle, \langle\, \langle\rangle, \langle E\rangle \,\rangle\big)\big) = \text{merge}\big(\langle\, \langle\rangle, \langle\rangle \,\rangle, \bot\big) = \bot$$

We return to the issue of associativity later in this section.

The design of both the consume() and merge() functions crucially relies on a "now-or-never" principle. Three key observations underly this principle: (1) UTL constraints only encode what *did* happen during the failing execution, and never encode what *didn't* happen; (2) each symbol from $\widehat{N_{\mathbb{L}}}$ appears in at most one node of *sccG*; and (3) *sccG* is acyclic. Recall that the goal is to find a path satisfying as many constraints as possible. Thus, while running consume() on a given node, *scc*, from *sccG*, we must consume as many symbols as possible from each constraint vector *now* (i.e., at *scc*), or we will *never* get the chance to do so again (because no symbol in *scc*.nodes appears in any other strongly-connected component, and *sccG* is acyclic). Furthermore, any time merge() encounters incomparable data-flow facts, it

can be certain that no future consume() call will ever fully satisfy all of the constraint vectors for both facts. Again, future satisfaction is impossible because *sccG* is acyclic, so each node in *sccG* is only visited (i.e., passed to consume()) once along any path. For example, looking to Fig. 6.4, note that the constraints $\langle B \rangle$ and $\langle C \rangle$ are unsatisfiable. To satisfy these constraints, our analysis would need to find a path through *sccG* containing both n_2 and n_3; no such path exists, and we would discover this point while running merge() on incoming facts for n_9.

Note that our data-flow analysis is operating over a DAG; therefore, we topologically order the nodes in *sccG* and only compute the data-flow fact for each node once. Thus, we will need to perform merge() and consume() at most once for each strongly-connected component. Suppose that S is the number of nodes in *sccG*, C is the size of *constraints*, and L is the length of the longest constraint vector in *constraints*. In the worst case, merge() operates on a number of predecessor facts on the order of S (all nodes in *sccG*), each of which contains C constraints of length L. We can process all predecessors in one merge() operation, which will find a predecessor vector that is an element-wise trailing substring of all others. This operation requires a linear scan over S predecessor facts, each with C constraints, each of size L; this operation thus requires $S \times C \times L$ total comparisons. Next, the call to consume() for a given strongly-connected component, *scc*, must find the first element of each $c \in C$ that is not a member of *scc*.nodes. In the worst case, each constraint still contains L elements, so this operation requires $C \times L$ set-membership checks. Thus, merge() and consume() are each polynomial in the size of *sccG*, *constraints*, and the longest constraint vector. Since we perform at most S calls to each, the whole approach is polynomial.

Recall that merge(), as defined in Fig. 6.7, is not associative. Fortunately, for any strongly-connected component, *scc*, we can always select *scc*'s predecessor that satisfies the most constraints by processing all of *scc*'s predecessors in one operation, as mentioned earlier. Furthermore, if we process *sccG* in a topological order, as previously suggested, the "now-or-never" principle (which relies on the fact that each symbol from $\widehat{N_{\mathbb{L}}}$ appears in at most one strongly-connected component) ensures that we can never encounter a situation where any pair of predecessors of a strongly-connected component have incomparable data-flow facts, but there nevertheless exists a path through *sccG* that satisfies all constraints.

For the final result, we answer *Possible* if there exists some scc_i such that $out(scc_i)$ is the vector of empty vectors (i.e., all constraints have been satisfied) and $crash \in scc_i$.nodes. Otherwise, we answer *Impossible*. For partial correctness proofs (including a proof of soundness with respect to Definition 6.7), see Appendix A.3.

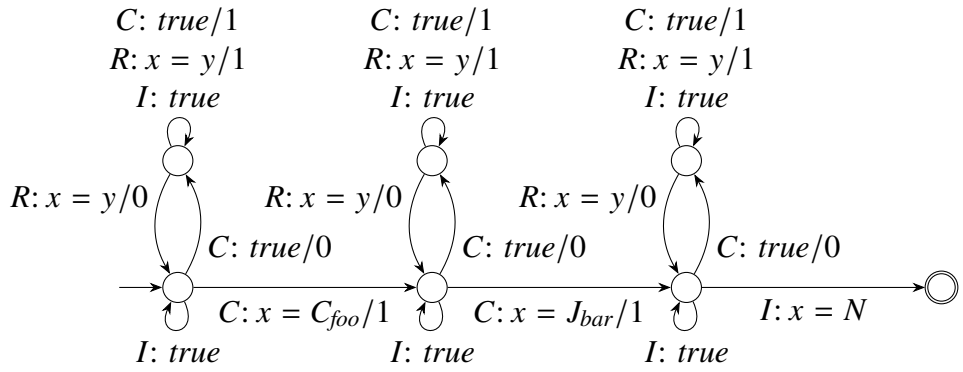
6.5.3 Constraint Negation

As stated previously, unreliable trace languages are not closed under negation in general. However, for *single-element* UTL vectors, we can support negated constraints using the emptiness procedure outlined in the previous section. Note that a single-element constraint, $\langle n \rangle$ (for some $n \in \widehat{N_{\mathbb{L}}}$), asserts that symbol n must occur at least once in any valid context-insensitive trace over G. To negate this constraint, we must ensure that n occurs in *no* valid traces over G. The simplest way to accomplish this goal is to remove n's corresponding node or edge from G. This removal effectively modifies $L_i(G)$, rather than adding an additional constraint into the *constraints* vector. Note that we must modify G prior to the construction of *sccG*, because removing a node or edge could change the structure of strongly-connected components in G.

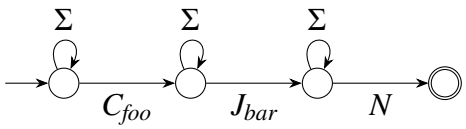
This property of negation does *not* extend to UTL generally. Specifically, negating a constraint with more than one element corresponds to subtracting sub-paths from G (rather than nodes or edges); thus, the same approach to removing nodes and edges from G does not apply in the general case. For example, we cannot negate the constraint $\langle C, E \rangle$ over the example CFG from Fig. 6.1 by removing either node C or node E (or, in fact, any combination of edges) from the graph, because doing so would also remove some valid context-insensitive paths from $L_i(G)$. In fact, Gabow et al. [63] show that the closely-related "impossible pairs constrained path problem" is NP-complete. (This problem, given a CFG, G, and a set of pairs of nodes over G, finds a path through G containing at most one node from each pair.) For both automata-based approaches from Section 6.4, the formal language classes are closed under negation, and, hence, negating any constraint is trivially possible. We further discuss encodings for single-element constraints (which are used to encode program coverage data) in Section 6.6.2.

6.6 Encoding Failure Reports

We now describe how to encode `csi-cc` trace data and other common failure report elements as s-VPA, FSA, and unreliable trace languages. Each failure element supplies a *constraint* that limits which runs can be consistent with the failure. All elements are defined over a CFG, $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$. Unless stated otherwise, all FSA constraints are identical to their s-VPA counterparts but with all transitions internal over $\widehat{N_{\mathbb{L}}}$.



(a) Represented as an s-VPA



(b) Represented as an FSA

Figure 6.8: Encodings of crashing stack $[C_{foo}, J_{bar}, N]$ as automata

6.6.1 Crashing Stack

Many failure reports include a stack trace at the point of failure, including core dumps prior to enhancement with `csi-cc` tracing. A crashing stack trace consists of a sequence of symbols

$$[\ell_0, \ell_1, \ldots, \ell_n, crash]$$

where each $\ell_i \in N \times \mathbb{L}$ and $crash \in N$. Each ℓ_i indicates a call that remains on the program stack at the time of the failure. Note that we require call edge labels (rather than call sites from N) because even calls through function pointers are unambiguous when the call remains on the final crashing call stack. Intuitively, a stack trace constrains the ordered, unmatched calls on any corresponding execution. Concretely, we represent this constraint as

$$L(stack) = \mathbb{W}\ \ell_0\ \mathbb{W}\ \ell_1\ \mathbb{W}\ \ldots\ \ell_n\ \mathbb{W}\ crash$$

where $\mathbb{W}$ corresponds to a "well-matched" region of execution (corresponding to the rest of the program's execution between the calls that appear in the crashing program stack).

Automata For visibly-pushdown languages, $\mathbb{W}$ perfectly encodes matched sequences of calls and returns. Accepted runs end with unmatched calls exactly matching those in the final crashing stack. Figure 6.8a shows the s-VPA encoding of the example crashing stack from Fig. 6.1. Note that here we again minimize stack symbol usage (as we did for CFG

encoding in Section 6.4.1). In this case, we require two symbols, 0 and 1. Whenever a 0 symbol is on the stack, we are in an "unmatched" state, and must match returns with all call transitions pushing 1 symbols before returning to a well-matched state based on the final call stack. Regular languages cannot express matched call/return sequences, so we replace $\mathbb{W}$ with a simple Σ^* self-loop. Figure 6.8b shows the corresponding FSA for our running example.

Unreliable Trace Languages The FSA encoding of stacks very nearly expresses an unreliable trace language. The corresponding unreliable trace language is

$$\langle C_{foo}, J_{bar}, N \rangle$$

This vector does not directly encode that execution must halt after reading N as the final symbol. Instead, this is managed via the parameter *crash* in Section 6.5.2. This difference does not impact the precision of results.

6.6.2 Statement Coverage

Some production-run failure reports may include statement coverage data. For example, our `csi-cc` instrumentation from Part I of this dissertation gathers coverage data at various granularities, both globally and within stack frames. Other tools allow developers to gather coverage data for untested code [139, 143]. Like our `csi-cc` data, some tools gather coverage at limited program locations (e.g., call sites) to aid analysis tools but pay lower tracing cost [67, 129]. Binarized statement coverage data consists of a set of independent observations, where each is a binary indicator (valued *true* or *false*). So, for example,

$$\{s_1\colon \mathit{true}, s_2\colon \mathit{false}\}$$

indicates that statement s_1 executed at least once during the failing run, and statement s_2 did not execute. Note that this characterization precisely matches extracted `csi-cc` data for local **coverage** maps and global ***globalCoverage*** maps per Section 5.2. We impose one constraint for each covered or uncovered statement. For our current encodings, we only use ***globalCoverage*** data for our interprocedural analysis; local **coverage** maps are used for intraprocedural analysis over single-function CFGs.

One could encode *false* entries via an automaton similar to that in Fig. 6.9a. However, note that this automaton simply ensures that s_2 appears in no accepted strings (representing paths through the CFG). Thus, in practice, we use a much simpler approach, and remove s_2 from the CFG prior to encoding other failure constraints.

(a) Encoding of "s_2: *false*" **(b) Encoding of "E: *true*"**

Figure 6.9: Encodings of coverage data entries as s-VPAs

Automata For "s_1: *true*", we must ensure that any accepted string contains at least one occurrence of s_1. If we let $s_1 = E$, we are encoding precisely the query from Fig. 6.1. Fig. 6.9b shows the s-VPA to enforce this constraint.

Unreliable Trace Languages The above *true* constraint (for s-VPAs and FSAs) expresses an unreliable trace language. Specifically, the constraint we want to impose for "E: *true*" corresponds to the vector $\langle E \rangle$. For *false* entries, we can support negated single-element vectors as UTL constraints per Section 6.5.3.

6.6.3 Path Traces

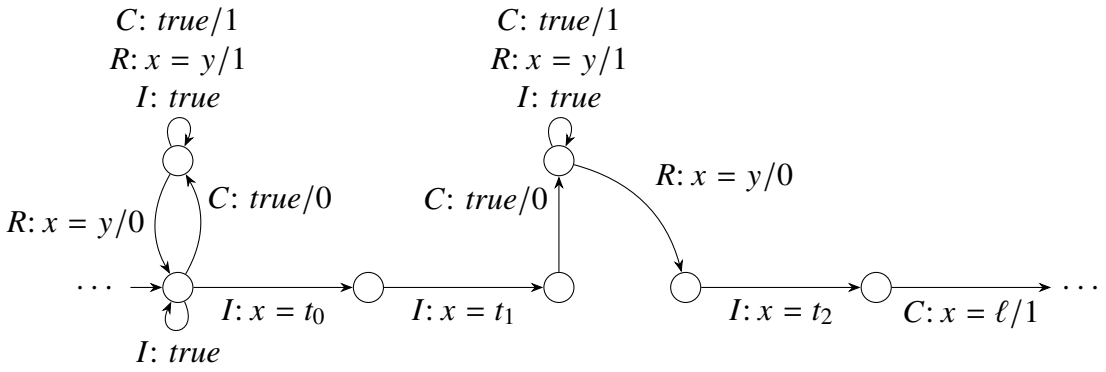
Our `csi-cc` tracing from Part I of this dissertation includes stack-local, bounded path traces. Per extracted failure data from Chapter 5, Section 5.2, path traces extend the stack trace previously described: each frame has at least the call that remains on the active stack at the crash (per our stack trace encoding), but may also contain a vector of nodes leading up to the call. Formally, a stack with path traces is a sequence of vectors

$$[\langle p_{0_0}, \ldots, p_{0_{|p_0|}}, \ell_0 \rangle, \langle p_{1_0}, \ldots, p_{1_{|p_1|}}, \ell_1 \rangle, \ldots, \langle p_{n_0}, \ldots, p_{n_{|p_n|}}, \ell_n \rangle, \langle p_{c_0}, \ldots, p_{c_{|p_c|}}, crash \rangle]$$

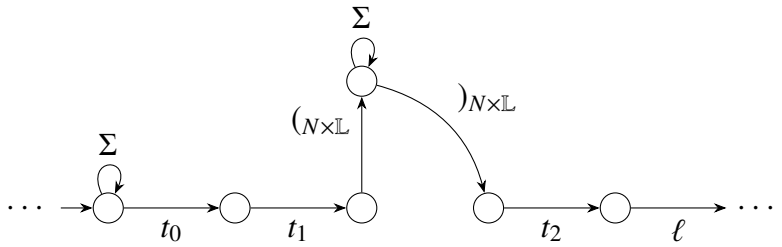
For our example in this section, we will describe encoding for just one stack frame including its path trace; frames are still connected via call transitions on each $\ell_i \in N \times \mathbb{L}$, precisely as in Section 6.6.1. Our running example from Fig. 6.1 is not complex enough to demonstrate all details of path trace encoding. Instead, we use the following generalized vector representing a stack frame with its path trace containing three nodes:

$$\langle t_0, t_1, t_2, \ell \rangle$$

For this example, suppose that t_1 is a call site, and $\ell \in N \times \mathbb{L}$ indicates the active call to the next stack frame.



(a) Represented as an s-VPA



(b) Represented as an FSA

Figure 6.10: Encodings of crashing stack frame with path trace $\langle t_0, t_1, t_2, \ell \rangle$ as automata

Automata Figure 6.10a shows the s-VPA encoding for the above constraint. Note that this figure corresponds to just one frame of the crashing program stack (indicated by ellipsis). We begin with a well-matched region ($\mathbb{W}$) to indicate any execution within the frame prior to path trace data. Then, our automaton only accepts strings containing the path trace entries, which much be in sequence with no gaps for other execution. The transition after t_1 is an exception. Since t_1 is a call site, we expect another well-matched region of execution before continuing with the intraprocedural path trace. Note that path tracing does not record the target of indirect calls (e.g., through function pointers), so we do not specify the label of the called function.

As before, since regular languages cannot express context sensitivity, our FSA encoding replaces each well-matched call region ($\mathbb{W}$) with an unspecified call, return, and Σ^* self-loop. Thus, note that matched substrings between t_1 and t_2 are not guaranteed to preserve calling context. Figure 6.10b shows the corresponding FSA.

Unreliable Trace Languages Unlike basic stack traces, path trace data introduces elements into the crashing stack that cannot be modeled precisely with UTL. Specifically, consecutive transitions (such as those for t_0 and t_1 in Fig. 6.10b) that previously had no Σ^* self loop

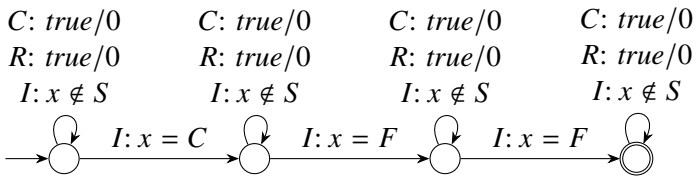


Figure 6.11: Call trace "$[C, F, F]$" as an automaton, where $S = \{C, J, F\}$

can no longer be precisely encoded, since UTL cannot ensure that these statements occur consecutively. For the example frame, the UTL encoding is $\langle t_0, t_1, t_2, \ell \rangle$.

6.6.4 Call Traces

Call traces record occurrences of specific call sites during a program's execution. These traces come in many forms, from logs of every call and return during a program's execution [44, 155, 182] to data gathered in short bursts [25]. Here we focus on traces over some set of call sites $S = \{s_1, \ldots, s_n\}$ (not necessarily every call site in the program), where each $s_i \in N$. Bursty traces are encoded similarly, and add unconstrained execution (i.e., Σ^*) before and after each burst. A call trace is of the form:

$$[c_1, c_2, \ldots, c_m]$$

where each $c_i \in S$. Crucially, such a trace indicates that we have seen *every* instance of each c_i during the traced execution.

As an example, consider the call trace $[C, F, F]$ with respect to the CFG from Fig. 6.1. Assume that $S = \{C, J, F\}$ (i.e., all call sites). This failure constraint cannot possibly be satisfied in any context-sensitive paths through G.

Automata To encode the above constraint, we form an automaton that accepts precisely the observed sequence of calls, with no other instances of symbols from S. Figure 6.11 shows the s-VPA encoding for $[C, F, F]$. Note that this call trace constraint is regular, since it does not encode the matching relations between calls and returns, but, rather, simply their order.

Unreliable Trace Languages Call trace constraints are *reliable* traces, because they indicate that we observe all occurrences of each traced call. Unfortunately, as we prove in Appendix A.2.2, we cannot precisely encode reliable traces and use any known polynomial-time solver to obtain query answers. Instead, we can encode the "$[C, F, F]$" constraint as the

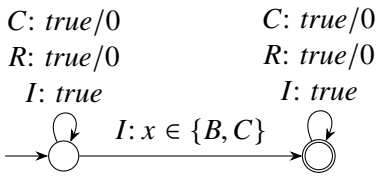


Figure 6.12: Ambiguous coverage data entry "$\{B, C\}$: *true*" as an automaton

unreliable trace language $\langle C, F, F \rangle$ with some loss of precision. Specifically, this constraint no longer enforces the requirement that we have seen all instances of the call sites from S.

6.6.5 Ambiguous Observations

Failure report data may be ambiguous. For example, when a program logs some of its activity, an analysis tool may be unable to determine precisely which output statement printed the message. Here, *exactly one* of the output statements executed on the failing run. In fact, all of the above constraints have generalized variants, where each observation is a *set* of symbols, rather than a single symbol. Here, we consider the specific case of ambiguous coverage data.

Ambiguous binarized statement coverage data consists of a set of independent, Boolean-valued observations of statement groups. For example, for $S_1, S_2 \subseteq N$,

$$\{S_1\text{: } \mathit{true}, S_2\text{: } \mathit{false}\}$$

indicates that at least one statement from S_1 executed during the failing run, and at least one statement from S_2 did not execute. The failure report from Fig. 6.1 contains a log message that was written by either statement B or statement C. This corresponds to the ambiguous coverage observation "$\{B, C\}$: *true*".

Automata For "S_1: *true*," all accepted strings must contain at least one occurrence of some $s \in S_1$. Figure 6.12 shows the s-VPA to enforce the constraint "$\{B, C\}$: *true*."

The encoding for "S_2: *false*" is less elegant, and potentially requires an exponential number of states in the size of S_2. We create one unambiguous *true* coverage automaton for each $s \in S_2$ per Section 6.6.2. Then, using this set of automata, A, the desired encoding is

$$\neg\Big(\bigcap_{a \in A} a\Big)$$

Unreliable Trace Languages Recall that unreliable trace languages (defined in Section 6.5.1) do not allow disjunction. In fact, ambiguous constraints (such as ambiguous

coverage data) cannot be expressed, even with loss of precision, as unreliable trace languages. Put another way, since unreliable trace languages do not support character classes in constraints, the above coverage constraints can only be expressed as Σ^*. In Appendix A.2.1, we prove that precisely encoding ambiguous observations results in NP-hard complexity for context-insensitive query recovery.

6.7 Best-Effort Coverage Analysis

As stated at the start of this chapter, one might use our solvers to answer a variety of interactive and non-interactive queries in many postmortem analysis and debugging scenarios. Here, we describe one example of a non-interactive batch analysis: recovery of best-effort program coverage data based on failure reports provided. Later in this chapter, we will use this analysis for evaluation of our three solvers. Our *best-effort coverage problem* (BECP) is defined as follows.

Given a CFG, $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$, and a failure report, $\{FP_1, \ldots, FP_n\}$ containing s-VPA-definable languages, an answer to the BECP is a partition of N into subsets *Yes*, *No*, and *Maybe*. The set *Yes* contains nodes from N that necessarily exist on all paths through G that are consistent with the failure report. Conversely, *No* contains nodes that cannot exist on any such path, while *Maybe* contains all nodes from N that are not in either *Yes* or *No* (most precisely: nodes that exist on some but not all paths consistent with the failure report). Thus, an imprecise solver for BECP may always place all n into *Maybe* for all $n \in N$.

Our context-sensitive and context-insensitive query recovery problems (Definition 6.6 and Definition 6.7, respectively) can be used to solve BECP. Inputs G and $\{FP_1, \ldots, FP_n\}$ are given as above. Then, for each node $n \in N$, we pose two queries:

(Q1) May n have executed during the failing run?

(Q2) May n *not* have executed during the failing run?

These queries are encoded precisely as statement coverage per Section 6.6.2. Recall that each of these queries can return an answer of *Possible* or *Impossible*. We then run our solver for each query:

$$possibleYes_n = true \text{ if (Q1) is } Possible\text{, otherwise } false$$
$$possibleNo_n = true \text{ if (Q2) is } Possible\text{, otherwise } false$$

Table 6.2: Evaluated applications

			Mean Count Across Variants			
Application	Type	Variants	LoC	Functions	Basic Blocks	SCCs
tcas	Siemens	41	173	12	163	163
schedule2	Siemens	9	373	25	268	228
schedule	Siemens	9	413	23	229	169
replace	Siemens	31	563	27	437	245
tot_info	Siemens	23	564	18	231	129
print_tokens2	Siemens	10	568	27	429	395
print_tokens	Siemens	7	727	25	385	318
ccrypt	Linux utility	1	5,280	116	1,677	1,409
gzip	Linux utility	20	8,114	135	3,704	2,411
space	ADL interpreter	34	9,563	158	3,895	3,389
sed	Linux utility	31	14,314	219	6,612	3,481
flex	Linux utility	53	14,946	184	6,408	3,992
grep	Linux utility	19	15,460	177	7,121	2,886
gcc	C compiler	1	222,196	2,267	142,121	51,668

Our solution to BECP is then:

$$Yes = \{n \in N \text{ such that } possibleYes_n \wedge \neg possibleNo_n\}$$
$$No = \{n \in N \text{ such that } possibleNo_n \wedge \neg possibleYes_n\}$$
$$Maybe = \{n \in N \text{ such that } possibleYes_n \wedge possibleNo_n\}$$

In practice, we only pose the above queries for each basic block, b, from G, and we represent each basic block by the first statement in that block. If *Yes*, *No*, and *Maybe* do not partition N after this process, the provided failure report data is inconsistent. (This does not occur in any of our experiments.) A solver for BECP is more precise if it can soundly increase the size of the *Yes* and *No* sets, and thereby decrease the size of the *Maybe* set.

6.8 Experimental Evaluation

Our empirical evaluation assesses the precision and scalability of each of our solving techniques. Specifically, we compared the time it takes our system to solve BECP (from the previous section) when encoding the CFG and all failure constraints as s-VPA, each of these elements as FSA, and by answering unreliable trace language queries over the context-insensitive CFG. We also compared the precision of analysis results when encoding constraints using each of

Table 6.3: Mean number of constraints intersected

		Call-Site Coverage Constraints	
Application	Stack	False	True
tcas	2	8	7
schedule2	2	17	24
schedule	2	13	22
replace	2	30	20
tot_info	2	12	14
print_tokens2	2	27	40
print_tokens	2	29	26
ccrypt	2	164	25
gzip	2	381	23
space	2	390	108
sed	2	546	113
flex	2	590	132
grep	2	448	89
gcc	2	12,649	585

the formulations. (Recall that a solver is more precise if it answers *Impossible* to more user queries, and, hence, classifies more nodes into the *Yes* and *No* sets for BECP.)

Our s-VPA solver builds upon D'Antoni's symbolic automata library [46], while our FSA solver uses OpenFst [8]. The analysis infrastructure and our solver for unreliable trace languages (UTL) from Section 6.5.2 consist of 5,016 lines of Python code. We instrumented subject applications with `csi-cc` to gather failure reports. We used Clang to produce our CFGs; thus, unlike in Chapter 5, we do not suffer from ambiguity in matching our trace data to our CFG, since `csi-cc` is also built atop Clang.

We used the same set of failures from Chapter 5, this time including failures for the replace and tot_info applications because we were able to map all failure data back to our CFGs effectively. However, we wanted to examine the scalability of our more precise analyses described in this chapter; many of our techniques required substantially more time and memory, making a complete evaluation of all failures infeasible. Thus, we randomly selected one failing run for each fault of each application version, and generated a failure report by extracting `csi-cc` trace data and the stack trace from the core dump produced by that run. Thus, the number of analysis runs for each solver is identical to the number of variants for each application. Table 6.2 again shows our evaluated applications, and also provides information about the CFGs for each application: the number of functions, basic blocks, and

Table 6.4: Incomplete analyses using stacks only

		Time, Memory		
Application	Variants	s-VPA	FSA	UTL
ccrypt	1	1, 0	0, 0	0, 0
gzip	20	20, 0	0, 0	0, 0
space	34	34, 0	0, 0	0, 0
sed	31	28, 3	0, 0	0, 0
flex	53	50, 3	0, 0	0, 0
grep	19	14, 5	0, 0	0, 0
gcc	1	0, 1	1, 0	1, 0

strongly-connected components (SCCs). We gathered these statistics prior to incorporating any failure constraints which cause us to split basic blocks or SCCs.

6.8.1 Crashing Stack Trace

Our first set of experiments sought to answer two research questions:

(RQ1) How well does each solver scale to larger programs?

(RQ2) How precisely can each solver answer queries over very sparse failure reports?

To answer these questions, we ran BECP for each application failure, where we extracted failure reports containing only crashing program stacks; that is, we did not encode any traced `csi-cc` data extracted from failing core dumps. Thus, each analyzed failure involves intersecting the query language with exactly 2 other languages: the language of the program's CFG and the failing stack trace. This is shown in the "Stack" column of Table 6.3.

To answer (RQ1), we limited solvers to 3 hours of running time and 30 GB of memory; we then tracked whether each analysis run completed all BECP queries within the time limit, ran out of time, or ran out of memory. Table 6.4 shows these results, omitting rows for applications that solved all queries for all solvers. For each solver, we report the comma-separated pair of the number of analysis runs that ran out of time and ran out of memory.

The s-VPA solver times out for all larger applications. This solver completes at least one query for many of these applications, but note that checking emptiness is significantly more complex for an s-VPA than for an FSA (where it is simply unconstrained state reachability). Hence, we find that **precisely answering queries with s-VPA does not scale to larger programs.** gcc's CFG has over 140,000 basic blocks, resulting in over 280,000 queries. No

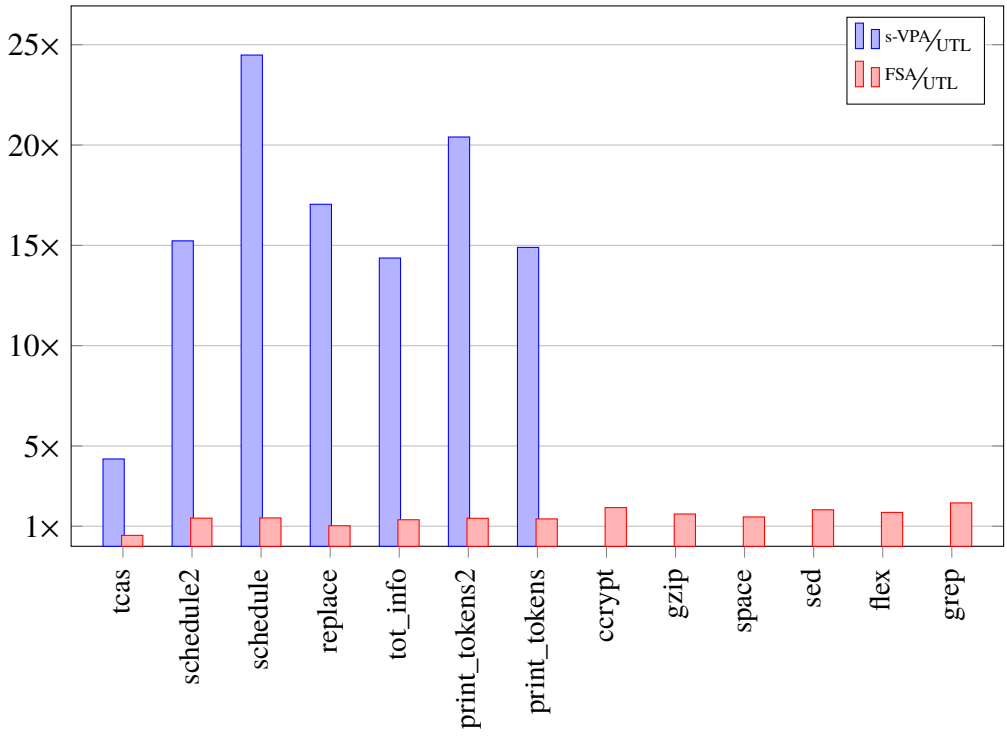


Figure 6.13: UTL-relative analysis time using stacks only

solver completed all gcc queries within the 3-hour time limit. Both the FSA solver and the unreliable trace languages (UTL) solver, however, are able to answer queries in this case. The FSA and UTL solvers averaged 7.88 and 5.19 seconds, respectively, to answer each query for those that completed within the time limit.

Next we compared analysis times for those analysis runs that multiple solvers completed. Figure 6.13 shows these results, plotted as analysis time relative to the UTL solver, averaged across each application. While each s-VPA bar summarizes fewer runs than the corresponding FSA bar, note that all runs that completed with the s-VPA solver also completed with the FSA solver. Missing s-VPA bars toward the right side of Fig. 6.13 echo our earlier finding that s-VPAs do not scale to large programs. Even for small programs, the s-VPA approach takes much more time, with slow-downs up to 25× for schedule (where all solvers completed all failures, per Table 6.4). **The FSA solver is slightly slower than the UTL solver on all non-trivial applications, even when incorporating only the crashing stack as the lone failure constraint.** The largest slow-down here is 2.2× for grep.

Finally, to answer (RQ2), we measured the improvement in precision from using the more expressive s-VPA solver (and, thereby, solving the context-*sensitive* query recovery problem from Definition 6.6). Per Section 6.6.1, s-VPAs allow a more precise encoding of the

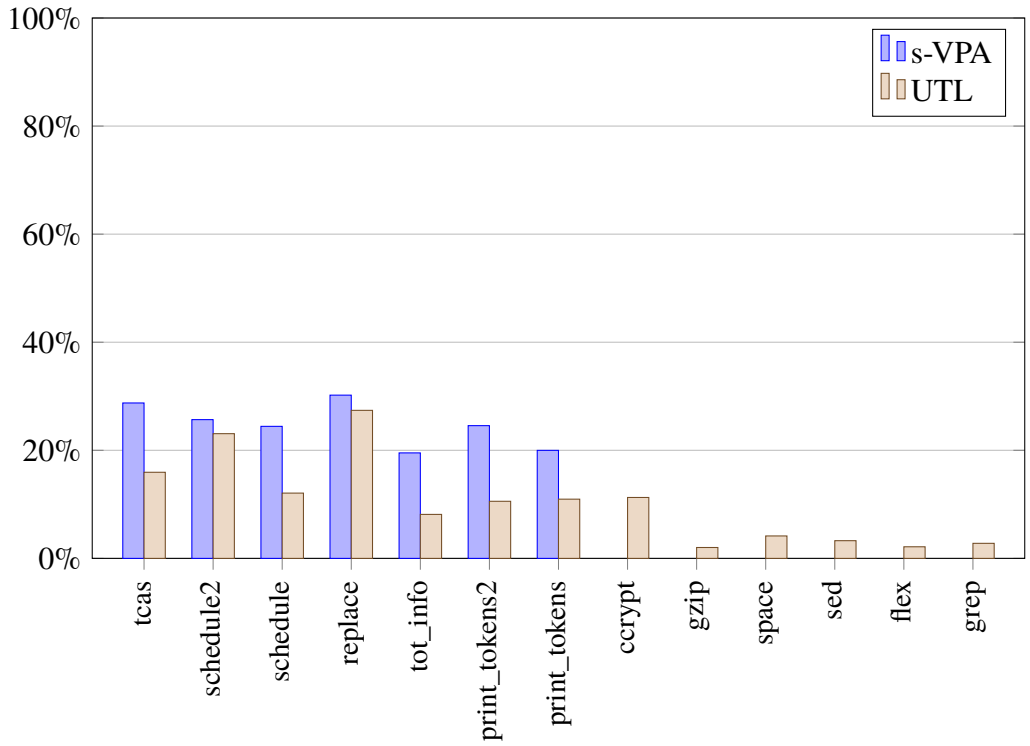


Figure 6.14: Basic blocks categorized as *Yes* or *No* using stacks only

conditions producing a failing stack than the regular language encoding. Figure 6.14 shows the mean percentage of basic blocks that each solver classified into the *Yes* or *No* sets for each failing run. We only show results for the s-VPA and UTL solvers here; FSA results are always identical to those of the UTL solver, as the language encoding of the CFG and crashing stack are identical for these solvers. Two patterns are clear. First, **for those failures that can tolerate its cost, the s-VPA solver gives substantially more precise results.** For example, for tot_info, classified basic blocks grow from 8.1% to 19.5%: a 2.4× improvement. Second, **information about the crashing stack alone is not enough to reduce execution ambiguity significantly for the larger programs.** This agrees with our findings from Chapter 5.

6.8.2 Crashing Stack And Call-Site Coverage

For our second set of experiments, we wanted to answer the following research questions:

(RQ3) How well does each solver scale to more dense failure reports (with many constraints)?

(RQ4) How precisely can each solver answer queries over dense failure reports?

Table 6.5: Incomplete analyses using stacks and call-site coverage

		Time, Memory		
Application	Variants	s-VPA	FSA	UTL
schedule2	9	0, 9	2, 3	0, 0
schedule	9	0, 8	0, 5	0, 0
replace	31	0, 28	0, 0	0, 0
tot_info	23	0, 17	0, 0	0, 0
print_tokens2	10	0, 10	0, 10	0, 0
print_tokens	7	0, 7	1, 4	0, 0
ccrypt	1	0, 1	0, 1	0, 0
gzip	20	0, 20	4, 7	0, 0
space	34	0, 34	0, 34	0, 0
sed	31	3, 28	2, 26	0, 0
flex	53	18, 35	0, 52	0, 0
grep	19	0, 19	1, 18	0, 0
gcc	1	0, 1	1, 0	1, 0

We used failure reports containing the crashing program stack and `csi-cc` global call-site coverage. We optimized instrumentation with the dominator-based approximation from Chapter 4. We then encoded the resulting coverage data for each failing run as described in Section 6.6.2. The "Call-Site Coverage Constraints" columns of Table 6.3 show the number of constraints resulting from the *true* and *false* coverage entries for each failing run, averaged by application. Thus, the sum of these two columns indicates the total number of call sites instrumented, averaged by application. Note that the number of instrumented sites is very similar, but not identical, to the number of coverage probes in our coverage optimization experiments from Chapter 4, Fig. 4.8; this difference arises because enabling faults can change a program's control flow, thereby impacting optimal placement of probes. As in Section 6.8.1, we solved BECP for one failing run per (application, version, fault) triple, and limited solvers to 3 hours of running time and 30 GB of memory.

We first examined (RQ3). Table 6.5 gives the number of analysis runs that ran out of time or memory for each application. Here, both the s-VPA and FSA solvers ran out of either time or memory analyzing most failure reports, even for the smaller Siemens applications. Memory issues are especially prevalent, as intersecting large numbers of constraints results in a worst-case exponential increase in the size of automata. Thus, **neither automata-based solver scales well to dense failure reports**. On the other hand, **the UTL solver was able to solve BECP for every single failure except for gcc,** where it averaged 8.12 seconds to

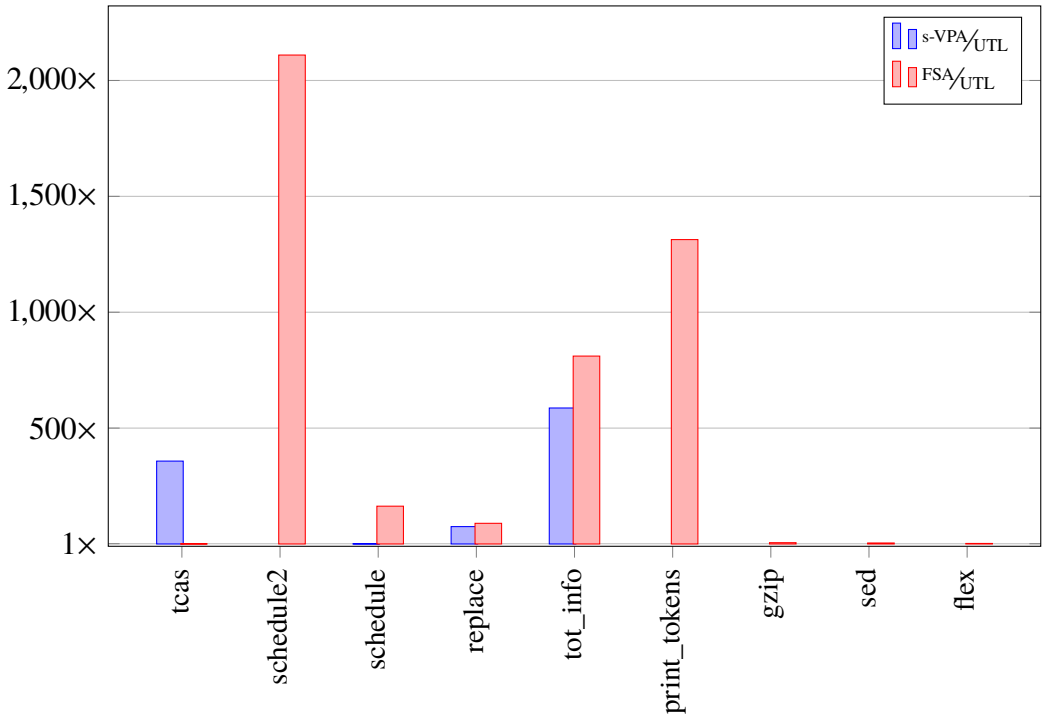


Figure 6.15: UTL-relative analysis time using stacks and call-site coverage

answer each query that completed within the time limit. The gcc failure had 585 *true* coverage entries in its failure report (per Table 6.3), so an automata-based solver would need to compute an intersection over 587 automata (including the CFG and crashing stack) to answer even a single query. In our experiments, the FSA solver exceeded the 3-hour time limit while performing intersections for gcc, but would certainly have exceeded the 30 GB memory limit if given more time.

For a detailed look into (RQ3), we again examined the total analysis time for those runs that completed with each solver. Figure 6.15 plots these results, again relative to UTL solve time. Note that s-VPA and FSA bars cannot be directly compared, because the s-VPA solver did not solve all of the failures solved by the FSA solver. (This discrepancy is especially relevant for schedule, replace, and tot_info results, where the s-VPA solver was slower, but timed out on longer-running examples.) Overall, these results indicate that **the UTL solver is dramatically faster.** The only exceptions are for the small number of runs that completed for the FSA solver in gzip, sed, and flex. Here, the particular failures occurred very early in each application, so the majority of the CFG was quickly placed into the *No* set by each solver. For all of the more complex failures that completed with both solvers (including those for the

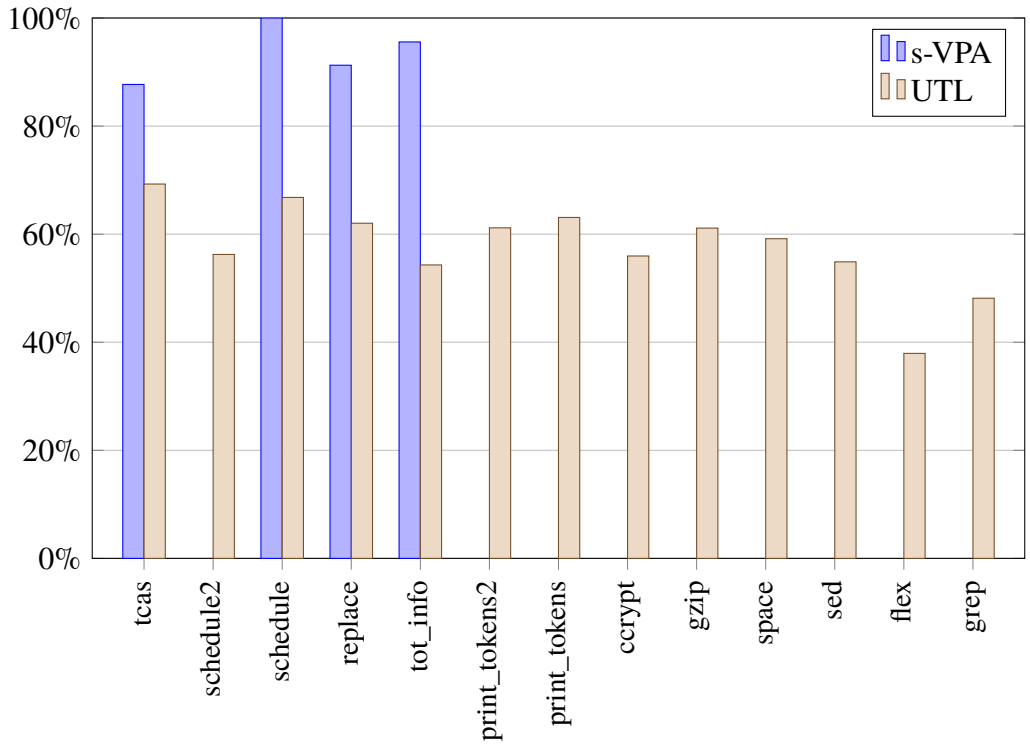


Figure 6.16: Basic blocks categorized as *Yes* or *No* using stacks and call-site coverage

smaller test subjects), UTL outperforms FSA, often by several orders of magnitude. These results are unsurprising, given the number of timeouts recorded in Table 6.5.

Finally, we investigated (RQ4). Figure 6.16 shows the percentage of basic blocks classified in the *Yes* or *No* sets by solving BECP over failure reports containing the crash stack plus coverage data. Recall that a solver is more precise if it classifies more blocks as *Yes* or *No* (rather than *Maybe*). Again, all failure report data is recorded with equal precision for FSA and UTL, so we display only the latter. The plot affirms that **we gain significant precision by encoding constraints to respect context sensitivity (via the s-VPA solver).** However, recall that we were able to solve very few failure report analyses with this solver. Another important result is that **precision improves significantly, even for imprecise solvers, when we incorporate more detailed failure reports.** For example the s-VPA solver disambiguates the execution of 20% of the basic blocks in print_tokens failures using only stack data (Fig. 6.14). Using `csi-cc` call-site coverage data, the UTL solver disambiguates 63% of basic blocks (Fig. 6.16) despite being unable to respect calling-context sensitivity during analysis.

Table 6.6: Resource usage. Times are in seconds and memory usage is in megabytes.

	Running Time (s)		Memory (MB)	
Application	minimal	natural	minimal	natural
schedule	18	28	250	1,176
replace	50	236	1,194	4,460
print_tokens2	46	218	958	6,786

6.8.3 Impact of Minimizing Stack Symbols

Recall from Sections 6.4.1 and 6.6 that our s-VPA encodings attempt to minimize the number of stack symbols used for encoding our CFGs and failure constraints. Our final set of experiments examined whether this design choice significantly impacts analysis performance. We randomly selected one failure from each of three randomly-selected Siemens applications. We then ran our s-VPA analysis over each failure in two variants. First, we used our solver precisely as in our primary evaluation (with a small number of stack symbols). Second, we used a variant of our s-VPA solver with the natural encoding that uses one stack symbol for each call site in the CFG; here, we replaced each 0 stack symbol in the encoding from Section 6.4.1 with a unique integer for the calling node, and removed the equality predicate "$x = y$" on all return transitions. We measured the execution time and memory usage for solving our best-effort coverage problem using only the failing stack trace (i.e., no `csi-cc` coverage data).

Table 6.6 shows our results. Columns labeled "minimal" use the s-VPA encoding from our primary experiments; "natural" columns use the encoding with one stack symbol per call site. Even on these small applications, there is a clear benefit to minimizing stack symbols. The natural stack-symbol-based encoding sees higher memory usage due to larger automata from product-set construction for intersections, as well as longer running times (to check emptiness over the larger automata). In the most extreme case (print_tokens2), running time increases by 4.7× and memory use increases by 7.1×.

The two encodings should produce identical results for all queries. The automata represent the same languages; they differ in whether we use return predicates ("minimal") or stack symbols ("natural") to ensure calling-context sensitivity. We verified that all results matched.

6.8.4 Discussion

In our primary evaluation, we set a 3-hour timeout for solvers to complete *all* queries for a given failure (2 per basic block). Nearly all subjects in Tables 6.4 and 6.5 that fail due to timeout complete at least one query. The only exceptions are in Table 6.5, where all s-VPA runs on sed and flex, and FSA on gcc, reach the 3-hour time limit, but would have exceeded the 30 GB memory limit if allowed to complete more intersections. Per Section 6.8.1, both the FSA and UTL solvers handle queries over gcc (the largest subject) within seconds using a stack trace. Per Section 6.8.2, the UTL solver can even answer gcc queries with the full failure report (with over 13,000 constraints) in seconds. The s-VPA solver answers queries for most failures on the larger applications using only a stack trace; the largest non-gcc applications (sed, flex, and grep) averaged 2 minutes per query. For batch analyses (e.g., extracting coverage data for an entire program), 3 hours is quite reasonable, and vastly outperforms a developer reconstructing this information manually. Our solvers are also fast enough to answer individual, interactive queries, such as the examples from the start of this chapter.

Overall, our results indicate that we can answer many control-flow queries based on failure data in reasonable time, and point to different use cases for our three different solvers. The UTL solver answers queries remarkably efficiently, even when we include very large failure reports (with over 13,000 failure constraints for gcc). However, it sacrifices some expressiveness in the types of failure report elements that it can precisely encode (see Section 6.6). Thus, we consider the UTL solver to be an excellent choice for answering *interactive* queries during a debugging session, where speed trumps precision. In contrast, the s-VPA solver can answer queries more precisely in many cases, but requires more time to do so, and struggles with memory constraints on very large failure reports. Thus, we consider the s-VPA solver to be well-matched to longer, overnight batch analyses. The FSA solver is a compromise between these extremes. As with s-VPA, the FSA solver needs to constrain the size of failure reports to avoid memory issues in its automata intersections, but allows more expressiveness than the UTL solver, while often maintaining similar efficiency for answering interactive queries.

In this chapter, we evaluated our techniques by solving the best-effort coverage problem, defined earlier. While this is only one example scenario run as a batch analysis, we nevertheless can garner insights into the scalability of our techniques and the impacts of failure report features. Our results suggest that failure reports with a limited number of constraints are good targets for improving precision by using the s-VPA and FSA solvers. This suggestion means that detailed failure reports may need to intelligently drop some constraints to make use of these solvers. Gathering more expensive failure data—e.g., call *traces* rather than call

coverage (see Section 6.6.4)—often results in a smaller number of constraints, though at a higher tracing cost [25, 44, 155, 182]. This move to more expensive data would improve the performance and precision of the FSA and s-VPA solvers because the complexity of checking automata intersection emptiness depends on the *number* of constraints intersected. The fact that our analyses are much more efficient using only a crashing stack trace suggests that reducing the number of failure constraints is important in practice; this includes optimizations like those from Chapter 4.

In the context of the `CSI` system in general, our results once again indicate that lightweight core dump enhancement can effectively improve postmortem analysis results at only a small cost to end-users. As an example, in gzip, ambiguity in executed statements on the failing run (i.e., the size of the *Maybe* set) decreases from 98% of the entire CFG (stack only) to just 39% using only global call-site coverage data and our most efficient (UTL) solver.

6.9 Threats to Validity

The threats to validity for our results are similar to those for our evaluation in Chapter 5, discussed in Section 5.4. Since we evaluate the same set of applications with a subset of the same failures, the only major changes relate to our mitigation of threats to the internal validity of our findings. Notably, we do not suffer from the ambiguity issues seen in Chapter 5, because both our `csi-cc` failure data and our CFGs are generated by Clang/LLVM.

We also validated our results in two ways. First, each of our analyses should safely approximate complete, directly-observed coverage. We spot-checked analysis results against full coverage data (gathered as `csi-cc` statement coverage) for a selection of failing runs. All analyses safely approximated the complete information: in every case that we checked, if an analysis reported "Yes" for a basic block, then that block did actually run; similarly, blocks reported as "No" did not run. Our second approach to validation compares the solvers to each other, one basic block at a time. Note that our three solvers have theoretical relationships that should be reflected in our results. Because all of our queries and coverage constraints are precisely definable in UTL, and, per Section 6.6.1, our encoding of the crashing stack is equivalent for FSA and UTL, the results of FSA and UTL should be identical in our experiments. The s-VPA solver can encode any s-VPA-definable constraint (subsuming all definable constraints for FSA and UTL), more precisely encodes the crashing stack (see Section 6.6.1), and ensures well-matched calls and returns for accepted strings; hence, its result should never be less precise than FSA and UTL. For the subset of blocks where multiple

solvers' results are available to compare, FSA and UTL always agree while s-VPA either agrees or is strictly more precise. That is, s-VPA never answers "Maybe" where FSA or UTL answered "Yes" or "No." This perfectly matches the expected theoretical relationships among the three solvers.

6.10 Related Work

LaToza and Myers [99] find that developers commonly ask reachability questions while debugging. This supports the usefulness of our technique, as many of the examples they consider are control-flow questions that we support.

EXPLORER [60] uses demand-driven pointer analysis to allow users to pose interprocedural control-flow queries over a program's call graph. Our system allows queries at the statement (rather than procedure) level, and answers queries with respect to all runs *consistent with a given dynamic failure report*, rather than (statically) all runs *whatsoever*. EXPLORER is also specifically geared toward refining call graphs for modern object-oriented languages; as our techniques in this chapter assume a pre-built CFG, our techniques may be complementary for analyzing modern programs with many dynamically-bound calls.

Our weakening of s-VPA constraints to unreliable trace languages resembles work by Place et al. [147] for automatically separating regular languages by piecewise-testable languages. UTL is a strict subset of the piecewise-testable languages. Thus, the class UTL may be useful in other contexts, such as language separation. Other related work approximates context-free languages with regular languages [40, 119, 127]; thus, it may be possible to automate the process of weakening failure constraints to UTL.

Gabow et al. [63] and Lal et al. [97] find paths through CFGs that reach desired nodes. Specifically, Lal et al. allow users to specify a stack trace, ordering pairs among CFG nodes, and general data-flow analysis properties. They then return a shortest path through the CFG that satisfies all constraints. Lal et al. also describe uses of their framework in fault localization. While Lal et al. incorporate other *data*-flow constraints, we dramatically broaden the class of *control*-flow constraints, and offer time/precision trade-offs via three underlying solvers. Sadly, our frameworks are not immediately compatible as we use different underlying formalisms (s-VPA versus weighted pushdown systems [152]). Our use of CFG condensation graphs is similar to that of Gabow et al. [63], who use standard reachability techniques to find a path satisfying a set of compulsory nodes that must be crossed on any matching execution. If we allowed only program coverage information (encoded as per Section 6.6.2),

this approach would suffice for our problem, and is essentially equivalent to UTL emptiness from Section 6.5.2. In effect, this restriction reduces our problem to a basic forward or backward reachability analysis over the CFG condensation graph. However, our UTL solver allows a broader class of constraints, which is why it maintains vectors of constraints, rather than simply marking compulsory nodes along the path. Our full suite of approaches is much more general, and allows an expansive family of constraints and queries.

Many prior approaches [35, 36, 42, 78, 79, 153, 193] use symbolic execution to replay failures based on varying styles of failure data. As stated in previous chapters, matching failures via symbolic execution is expensive, and undecidable in general. We answer queries about *any* run matching failure data, while replay produces *one specific* run that may or may not completely match the traced failure. In addition, most prior approaches are limited to specific categories of failure data [35, 42, 78]. In contrast, our approaches in this chapter can immediately take advantage of any s-VPA-definable constraint.

7 FRONT-END TOOLING AND HUMAN SUBJECTS EVALUATION

An earlier version of these designs was presented in a workshop paper by Ohmann and Liblit [136]. The extended work described in this chapter, while not yet published, includes contributions by Manav Garg and Pallavi Ghosh.

The preceding chapters address two important issues in debugging from post-deployment failure reports. Chapters 2 to 4 present mechanisms to efficiently gather more detailed trace data from failures. Chapters 5 and 6 show that combining this lightweight data with postmortem analyses that adapt to incomplete failure traces can substantially reduce ambiguity in the failing execution. Both of these issues, however, deal exclusively with how to *gather or extract* more detailed failure data. Prior work by Parnin and Orso [142] examines fault localization tools that present developers with an ordered list of suspect source code lines. The authors find that mere counts of suspect lines are not always indicative of a tool's actual impact on debugging performance, because developers need to also develop an *understanding* of failing behavior. Usefulness for actual users debugging deployed applications is the ultimate test of our analysis techniques; one must also consider how data is *presented* to developers.

To that end, this chapter describes the design of a front-end tool, `CSIclipse`, for displaying `CSI` tracing and analysis data. `CSIclipse` is a plugin for the popular Eclipse integrated development environment (IDE) [167]; our plugin presents `CSI` data to developers in a convenient but thorough manner. `CSIclipse` builds on `csi-cc` tracing and `csi-grissom` analysis to achieve this goal. Specifically, `CSIclipse` provides support for

1. viewing `csi-cc` trace data and `csi-grissom` analysis results,
2. annotating source code with execution coverage information based on our best-effort coverage analysis from Chapter 6, Section 6.7, and
3. stepping through `csi-cc` path trace data.

After describing the architecture of `CSIclipse`, we then briefly outline a preliminary design for a human subjects study to assess the utility of `CSI` instrumentation and analysis data as presented through `CSIclipse`.

7.1 CSIclipse

In this section, we present the design of `CSIclipse`. We begin by reviewing the formats of instrumentation and analysis data from previous chapters that are used as input to `CSIclipse`. Then, we detail how this data is presented in the various aspects of our plugin.

7.1.1 Input Data

Recall from Chapter 5, Section 5.2, that we extract `csi-cc` trace data from core dumps, and organize the data as follows:

- ***stack*** = $\langle frame_1, frame_2, \ldots, frame_n \rangle$: the failing stack where each $frame_i$ has fields:
  - **coverage** = $\{n_1 : b_1, n_2 : b_2, \ldots\}$: a mapping from CFG nodes to Boolean values indicating whether each n_i was executed at least once in $frame_i$'s execution.
  - **path** = $\langle p_1, p_2, \ldots, p_{|p|} \rangle$: a vector of CFG nodes indicating a partial execution suffix leading up to the final crash point within $frame_i$. Even for a core dump with no `csi-cc` trace data, **path** contains one entry: the final CFG node from the failing stack trace.

- ***globalCoverage*** = $\{f_1 : coverage_1, f_2 : coverage_2, \ldots, f_{|globCov|} : coverage_{|globCov|}\}$: a mapping from each function in the program's CFG to a **coverage** map as defined above. Here, Boolean values indicate whether each node was ever executed globally, that is, at any point during the entire program's execution, regardless of the failing stack.

Using any of our solvers from Chapter 6, we can then obtain three global sets—*Yes*, *No*, and *Maybe*—by solving our best-effort coverage problem from Section 6.7 using ***globalCoverage*** and just the final entry of each $frame_i$.**path** above (i.e., the failing stack trace). We also use this analysis for each $frame_i$ to obtain stack-local sets $frame_i$.Yes, $frame_i$.No, and $frame_i$.Maybe. For each frame, we run our best-effort coverage analysis over the intraprocedural CFG for $frame_i$'s function using $frame_i$.**coverage** and $frame_i$.**path**. For intraprocedural analysis, our use of **path** is much more straightforward than our encoding from Chapter 6, Section 6.6: we immediately add each node in **path** to $frame_i$.Yes, and instead use $\mathbf{path}_1$ as the crashing point (similar to intraprocedural active nodes analysis from Chapter 5, Fig. 5.2). Thus, the complete result of our analysis is:

- global sets *Yes*, *No*, and *Maybe*
- local sets $frame_i$.Yes, $frame_i$.No, and $frame_i$.Maybe for each $frame_i$ in ***stack***

`CSIclipse` loads instrumentation and analysis data in a simple comma-separated value (CSV) format, with one record for each file in the program's source, and one record for each frame in the failing stack trace. Each record consists of:

- the name and path for a source file, f
- a *Yes* set containing lines of f
- a *No* set containing lines of f
- a *Maybe* set containing lines of f
- a function name (for stack frames only)
- a path trace containing lines of f (for stack frames only)

While we designed this format specifically for ease of integrating `csi-cc` and `csi-grissom` data, `CSIclipse` will operate with any properly-formatted input data. Thus, other tools that can produce any portion of this data can use `CSIclipse` for visualization. The most notable advantage to this is that `CSIclipse` is not bound to a particular programming language, despite the fact that both `csi-cc` and `csi-grissom` only support C/C++ programs.

Encoding our analysis data as `CSIclipse` input is straightforward. First, since `CSIclipse` expects line numbers rather than CFG nodes, we need to convert each node in all of our trace data and analysis results into its corresponding line number. Thus, our *Yes*, *No*, and *Maybe* sets each contain a (possibly-overlapping) set of line numbers. Then, we split our global *Yes*, *No*, and *Maybe* sets into sets for each unique source file, and create one CSV record for each file (function name and path trace are empty for these records). Finally, for each $frame_i$, we create one record consisting of $frame_i$'s defining file and function name, along with $frame_i$.Yes, $frame_i$.No, $frame_i$.Maybe, and $frame_i$.**path**.

7.1.2 Design

As stated previously, `CSIclipse` is a plugin for the Eclipse IDE. Our plugin is designed for use during an active debugging session, and, hence, integrates with existing features in the Debug perspective of Eclipse. We also rely on underlying tooling within Eclipse for accessing information about the process currently being debugged: in the case of `CSI`, the C/C++ Development Tooling. Screenshots from this section are taken with Eclipse version 4.6.3 running on Red Hat Enterprise Linux 6. Throughout this section, we will use an example crashing run from the program flex; this failure was used for evaluation in previous chapters. We instrumented the program with `csi-cc` for path tracing and call-site coverage, then analyzed the core dump from the failing execution with `csi-grissom`, producing input data as detailed in the previous section.

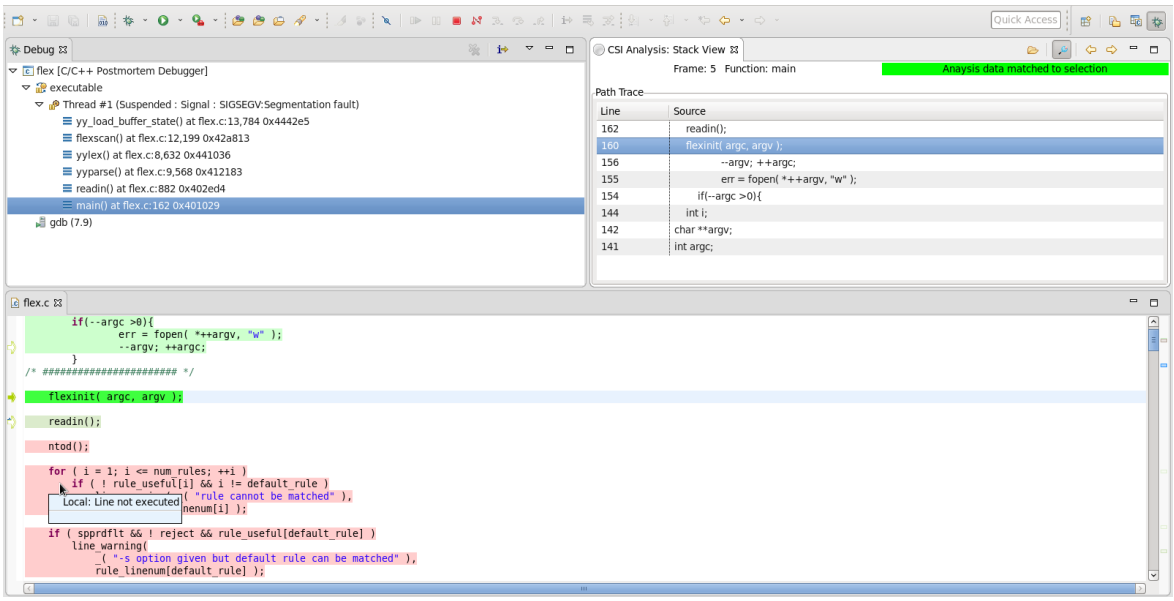


Figure 7.1: Highlighting and stepping through frame-local CSI data

Figure 7.1 shows a user actively debugging the flex failure. Note that in this and other screenshots, we do not display many standard views developers would use during debugging (e.g., local variables, registers, and disassembly) to avoid clutter and to focus on CSIclipse features. Here, the developer loaded the failing executable and core dump via the Eclipse postmortem debugging facilities; the crashing stack data in the standard debugging view (in the top-left corner of the window) indicates that the program crashed at line 13,784 of flex.c, in function yy_load_buffer_state. In the top-right corner of the screenshot, we see the CSI stack view. Here, the developer has already loaded input data via the 📂 button. Note that CSIclipse only maintains data for a single crash report, corresponding to analysis results for a single failing program run. Upon loading a new crash report, the old data is overwritten.

The stack view shown in Fig. 7.1 complements existing Eclipse views. When the developer selects a stack frame context from the standard Debug view, the CSIclipse stack view displays path trace information for that frame, if available: the sequence of lines in the frame's execution suffix, along with a preview of their source content. Selecting a trace line seeks to that line in the Eclipse editor; the developer can also step forward ⇨ and backward ⇦ through the trace. By default, clicking on any CSIclipse entry brings the user to a scratch read-only version of the source file for debugging. This behavior is configurable, and allows our tools to maintain correspondences among instrumentation metadata, analysis results, and source code line numbers. CSIclipse creates this copy the first time a user selects a particular file. Toggling the local annotations button 🔎 to the "on" position (as in the screenshot) enables

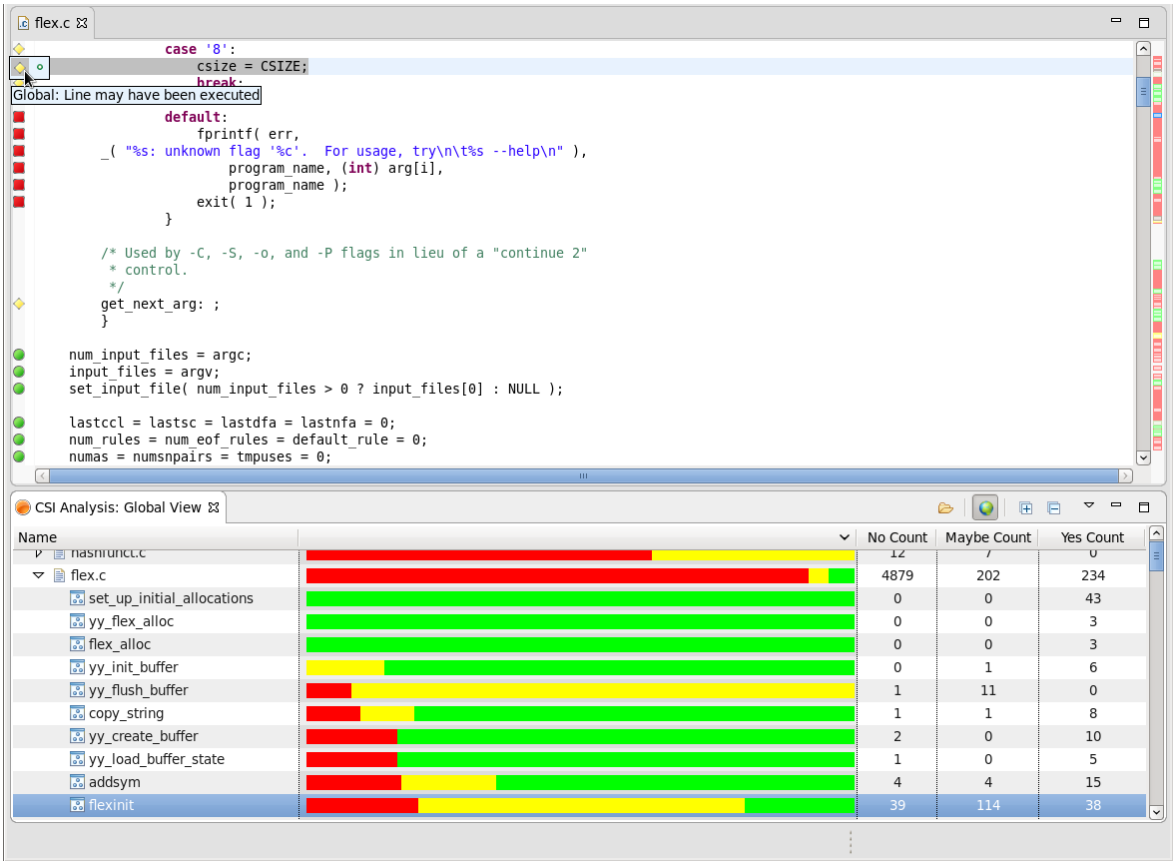


Figure 7.2: Annotations for global execution data

source code highlighting. `CSIclipse` highlights each source line in the relevant function with that line's execution status for the selected stack frame: green for *Yes*, red for *No*, and yellow for *Maybe* (by default). The currently-selected path trace entry is also brightly highlighted and indicated with a filled arrow. In a realistic debugging scenario, a developer is likely to start from the failure point, and step backward through the provided path trace. In this example, a user is currently stepping through the path trace gathered for main's stack frame.

While looking through main's execution, the developer will notice that the function flexinit was called on line 160, but is no longer on the active stack; thus, no path trace data is available for that call. However, flexinit is a complex initialization function, and contains relevant context information for the failure (which occurred during initialization). Figure 7.2 shows the `CSIclipse` global view, which is useful when the debugging task departs from the active stack. The view presents a tree rendering of all projects in the developer's Eclipse workspace, annotated with global information from loaded `CSI` data. The tree gives *Yes*, *No*, and *Maybe* line counts for each function, along with a visual representation of the ratios of these three

sets as a colored bar. The view also aggregates this data for containing elements; for example, each file is annotated with the summary of *Yes*, *No*, and *Maybe* data for all functions in that file. Selecting any element from the view again opens a buffer (read-only by default), centered on the selected function. When the developer toggles the global annotations button "on" (as in the screenshot), `CSIclipse` adds source annotations based on global data. Each marker indicates the line's execution status (*Yes*, *No*, or *Maybe*) for the entire program's execution. These markers are less invasive than their local counterparts (as small icons in the side-bar of the source file). Being standard Eclipse markers, global annotations are also visible in the file overview to the right of the scroll bar. This has a number of advantages: it allows a developer to quickly assess how much of a given file's code was used (visually complementing the summarized information in the global view), helps to identify large regions of executed or unexecuted code, and facilitates navigation to interesting code regions. These global markers also do not overlap with local annotations; the two classes can provide useful information when used together. For example, a line annotated locally *No* and globally *Yes* indicates that the containing function executed previously. A function whose final statement is marked as globally *No* never executed outside the crashing stack context.

7.2 Preliminary Design for Human Subjects Evaluation

In this section, we outline a preliminary design for evaluation of the `CSI` toolchain with human subjects. This procedure is inspired by previous studies in software engineering and program analysis research [90, 91, 100, 108]. While we have not performed this experiment, we present it here to guide future work. The study aims to answer three research questions:

1. Does `CSI` data (as presented by `CSIclipse`) impact the *time* for developers to fix bugs from failure reports?

2. Does `CSI` data (as presented by `CSIclipse`) impact the *quality* of bug fixes?

3. Does `CSI` data (as presented by `CSIclipse`) impact developer *perceptions* of debugging from failure reports?

Item 1 addresses the most intuitive question: can developers save time on post-deployment debugging by using `CSI`? This question is undoubtedly important, but far from the only important factor for debugging tools. Recalling that `CSIclipse` is specifically geared toward helping developers better understand failing runs (and complement existing failure data),

Item 2 looks at the quality of bug fixes. Underlying this question is the hypothesis that a developer who better understands failing behavior is less likely to contribute incomplete fixes or miss edge cases. Finally, Item 3 asks how developers perceive the contributions of `CSI`, and whether it lessens common frustrations with postmortem debugging. Because our techniques described throughout this dissertation target debugging post-deployment failures, the relevant beneficiaries of our tools are industrial software developers (rather than software designers or students learning to write code). We assume significant prior debugging experience.

The study uses a two-condition, within-subjects design with two debugging tasks. Each task consists of an application and three failure reports pertaining to bugs for that program; each failure report consists of a written summary and a core dump from the failing run. All participants complete both tasks. In both conditions, participants use the Eclipse IDE with support for postmortem debugging via `gdb` (through standard Eclipse tools); the second condition adds the `CSIclipse` plugin (from Section 7.1) to Eclipse. We counterbalance by randomizing both the order of tasks and the pairing of experimental conditions to tasks.

First, participants undergo refresher training on `gdb` features in Eclipse. Then, participants attempt each task in a laboratory setting for a fixed time duration (e.g., 20 min). Prior to the task that uses `CSI` data, we present a brief overview of `CSIclipse`. After completing each task, participants take a survey about their experiences on that task, with questions such as:

- How difficult was the application's code to understand? *(Scaled 1–10)*
- *[For each failure report]* If you attempted it, how difficult was the bug in this failure report? *(Scaled 1–10)*
- *[For each failure report]* If you completed it, how successful do you believe that you were in fixing the bug in this failure report? *(Scaled 1–10)*
- Please estimate how much time you spent on each of the following activities during this task. (If time was spent doing more than one activity simultaneously, count it for both.) (1) Understanding the source code, (2) Understanding the failure, (3) Coding (fixing bugs, writing tests, etc.), (4) Running the code or tests, and (5) Trying to get the provided tools to work as expected.

After the completion of both tasks, a semi-structured interview gathers further comparisons between tasks and asks specifically about the strengths and weaknesses of `CSIclipse`. Here, we pose questions such as:

- Which of the tasks was more difficult? Why was it more difficult?
- In what situations was the `CSI` data useful? In what situations was it not useful?
- What did the `CSIclipse` tool *not* do that it should have done?

All of these surveys address our third research question: they provide insight into whether a developer *perceives* that debugging was more successful with `CSI` data, regardless of whether or not this is actually the case.

To address the first two research questions, we also ask participants to mark particular failure reports as "Complete" when finished with them, and gather participant modifications to the code at the study's completion. Then, we use the time data to calculate the total time that each participant spent on each bug, and normalize by the number of bugs attempted (so as not to penalize for bugs that a participant did not attempt to fix). The difference in fix times for our two conditions (if significant) provides evidence for our first research question. However, as stated earlier, the quality of fixes is also relevant in our context. Thus, assuming that a fix does indeed correct execution for the original failing input, an independent experimenter assesses the quality of each fix on a three-point scale: 0 indicates that the bug was not attempted, 1 indicates an incorrect or superficial fix (e.g., hard-coded return values), 2 is a near-perfect fix (perhaps missing an edge case), and 3 indicates a completely-correct fix. Note that this assessment can (and should) be blind to our experimental condition (i.e., whether `CSIclipse` was used for the fix). If these quality values measurably improve when participants use `CSIclipse`, we find evidence to answer our second research question.

In closing this section, we briefly comment on our study population. Recall that the target for `CSI` instrumentation and analysis is industrial software developers with significant prior debugging experience. Thus, unlike many prior studies [91, 160], our setup precludes recruiting exclusively university students. Specifically, students without significant experience on large industrial software projects are not within the target demographic for this study. An acceptable condition would likely be that participants have at least six months of non-academic software development experience, and an acceptable level of familiarity with `gdb` as a debugging tool.

7.3 Related Work

Others [71, 120, 121] have developed Eclipse plugins to evaluate and display program coverage information. Often, such tools operate in the context of running or automatically generating JUnit test cases for Java programs. We draw inspiration from these tools on how to present coverage information in the Eclipse IDE. However, our analyses (1) work in the context of a single failure, (2) must handle partial coverage data (i.e., cases where multiple executions could possibly lead to a provided failure report), (3) must recover this data from

external failing executions (rather than Eclipse-local test executions), (4) segregate global and stack-local best-effort coverage data, and (5) also include path trace information. These concerns lead to different design choices. First, we use an explicit annotation for unknown (i.e., *Maybe*) execution data, as some ambiguity is the normal case, even after `csi-grissom` analysis. Second, we load execution data from external sources in CSV format; this also allows `CSIclipse` to integrate with other instrumentation and analysis tools. Finally, we separately analyze and display global versus local data; this allows us to take advantage of the full range of `CSI` tools from this dissertation, and adapt to different debugging contexts.

Many prior tools present program traces to developers [9, 86, 125]. The Path Projection toolkit [86] visualizes program paths from static analysis reports in an IDE, and provides users with a variety of tools with which to explore, understand, and compare different error paths. Our traces differ in some important respects: they are incomplete, and are derived from incomplete data from deployed applications. Nevertheless, these are mature tools with support for visualizing "nested" stack frame traces. `CSIclipse` could potentially improve trace visualization by adopting similar approaches, or perhaps integrate with existing tools.

Whyline [92] is a debugging tool that allows programmers to ask "why" and "why not" questions about a program's execution in the context of an interactive debugging session. The tool uses a combination of static and dynamic analysis to suggest and answer relevant questions. `CSIclipse` similarly aids developers in understanding failing program executions, but operates in a different context. We assume that failure data is recovered from deployed software, and, thus, both our instrumentation and analysis techniques intentionally sacrifice detail to improve tracing efficiency. Program slicing (in the context of debugging) extracts portions of a program's code that are relevant to a point of interest. Whyline and many other tools [14, 69, 76] allow a developer to perform static (generalizing all possible executions) or dynamic (restricted to a single failing execution) slicing within an IDE. Our `csi-spotlight` analysis from Chapter 5, Section 5.2.2, performs dependence graph restriction (loosely: partially-dynamic program slicing), and tools such as `CSIclipse` may benefit from displaying this data. While some foundational research suggests that slicing meshes with how developers debug in practice [179], more recent work suggests that most developers are unaware of slicing tools [145], and that even dynamic slices are often too large for practical use [194].

Recently, more software engineering researchers have evaluated tool effectiveness by performing studies with human subjects [90, 91, 100, 108]. Our preliminary study design from Section 7.2 is inspired by these prior studies. Our design could be used for evaluating the effectiveness of the `CSI` system or other similar tools.

Part III

Conclusions

8 CONCLUSIONS

In this dissertation, we described the tools and techniques that make up the instrumentation and postmortem analysis system, `CSI`, which specifically targets the requirements of deployed applications. Overall, we showed that very low-overhead tracing can yield large benefits for postmortem analysis of post-deployment failures. The coordination of customizable run-time tracing with analysis techniques that can tolerate imperfect failure data was key to our designs.

In Part I, we developed and evaluated new program instrumentation techniques and methods of tracing customization with overheads suitable for deployed applications. In Chapter 2, we began by introducing two lightweight tracing mechanisms—path traces and in-memory program coverage—that enhance readily-available information in core memory dumps from crashing applications; overheads were only 0.5% for running time and 1.8% memory use for a realistic combination of our path traces with coverage at call sites. In Chapter 3, we built on this success and investigated methods to allow users and developers to adjust tracing on-the-fly (without re-compilation and re-deployment). This additional flexibility came at a cost of just 2.0% running time on average, and slowed no application we studied by more than 4.6%. Finally, in Chapter 4, we presented broadly-applicable optimization techniques for gathering program coverage based on customized requirements post-deployment. Particularly for dense instrumentation, our optimizations significantly reduced both compilation and run-time costs. Even for our call-site coverage mechanism (which already focuses tracing significantly: targeting only call statements), we reduced static instrumentation by up to 57%, and dynamic executions of coverage probes by up to 55% in our experiments.

In Part II, we described and evaluated new postmortem analyses that work with incomplete trace data from post-deployment failures, including our own techniques and other common failure report elements. Our techniques focus on presenting results that are relevant for any failing run matching the provided failure data, and are thereby relevant for both very sparse failure reports (e.g., only stack traces) and very dense reports (e.g., near-full execution traces). In Chapter 5, we described two families of analyses that restrict the relevant code that a developer would need to consider while debugging, using our trace data from Part I. Our first analysis restricted the set of control-flow graph nodes and edges that were potentially active during matching failing runs, while our second analysis computed a trace-restricted program dependence graph to facilitate hybrid static-dynamic slicing over incomplete failure reports. Despite working with limited failure data, we see active node and edge reductions as high

as 71%, and interprocedural slice reductions as high as 78%. In Chapter 6, we presented a system for answering arbitrary control-flow queries over incomplete failure reports. This system went beyond our prior analyses both in the queries we supported and the failure report elements we were able to encode. We also introduced a new subclass of regular languages, the unreliable trace languages, that are especially suited to answering user queries in polynomial time. Finally, in Chapter 7, we presented a front-end tool to display tracing and analysis data.

At a high level, our results clearly indicate that instrumentation and postmortem analysis techniques for deployed applications work best in coordination with one another. Our techniques manifest this cooperation in numerous ways. In Chapters 2 and 5, we saw that combining dense stack-local tracing (path traces) with interprocedural-focused coverage data (at call sites) proved both an efficient and valuable combination. For example, one of our evaluated applications, `space`, crashes within a loop in a complex function containing many branches and a large **`switch`** statement. The bug is a missing `exit` statement within one **`switch`** case. Our `csi-spotlight` analysis is able to provide the complete branch trace within the crashing function, reducing the possible set of executed statements by over 65%. This benefit comes at a tracing-time overhead of just 0.6% relative to uninstrumented code (per Chapter 3, Fig. 3.2). In fact, the analysis of this `space` failure is a relatively simple, intraprocedural example. Interprocedurally and in aggregate, for just 0%–5% execution time overhead and 0%–4% dynamic memory overhead, we reduce interprocedural slice sizes by 51%–78%, per Chapter 5.

Our tracing mechanisms compensate for one another's weaknesses for use in analysis: path tracing provides dense data, but only at a limited distance from the failure point, while call-site coverage is coarse-grained, but extends to both local and global scope. Our analyses are structured, rather than simply adapted, to accept the imperfect failure reports that inevitably arise from post-deployment crashes. This design is especially evident in our `csi-grissom` analysis from Chapter 6, which accepts a large range of failure constraints from many tracing tools (including our own from `csi-cc`). Here, we see a clear example of analysis scaling based on failure report detail: our s-VPA solver precisely answers queries over small failure reports (e.g., using only a stack trace), while our unreliable trace language solver gives up some precision in our base analysis for a large overall benefit in precision by handling reports with over 13,000 failure constraints (Chapter 6, Fig. 6.16).

8.1 Lessons for Tool Developers

This section describes some key lessons relevant for developers who build tools for debugging deployed applications. These lessons derive from our findings throughout this dissertation.

Everything is "best effort" Failure data is rarely or never perfect, especially for deployed applications. Successful post-deployment tool developers should think about the meaning of trace data and/or how analysis data can be interpreted in the presence of ambiguity and imperfection. In our approaches, we always maintain soundness: we provide best-effort results that reduce developer burden, but our results always generalize any single run that could have produced the provided failure data.

Ambiguity using multiple tools Large instrumentation and analysis systems are built from multiple tools, and different tools inevitably manifest differences in program representations. We most notably grapple with this problem in Chapter 5, where differences in compilation-time debug information between Clang and CodeSurfer cause disagreements in mapping our trace data to CFG nodes. We largely avoid this problem with our `csi-grissom` analysis from Chapter 6 by producing analysis CFGs from Clang as well; however, even there, we need to map stack trace line numbers to CFG nodes, and multiple nodes may share the same line number. These mapping challenges provide further evidence that postmortem analysis tools should prepare for imperfect failure data (in this case: ambiguity in failure data).

Instrumentation changes have subtle impacts Seemingly simple factors can have disproportionate impacts when tracing deployed applications and measuring single-digit overheads. For example, effects on architecture-level caching behavior and branch prediction can become dominant (and very relevant) at low overheads. Our path tracing mechanism from Chapter 2 makes seemingly small changes to the classic path profiling approach from Ball and Larus [22]. Nevertheless, we see much smaller overheads, partly because we meticulously designed our instrumentation approaches, even down to how we implement modulus operations for circular buffer wrap-around at the machine-code level. Cache effects also should not be discounted: we move buffers into the stack and always access elements consecutively.

8.2 Future Directions

This dissertation shows that postmortem analysis can draw great benefit from even imperfect tracing data from post-deployment executions, especially if that tracing is chosen purposefully. These successes also reveal several future directions for continued research, and we consider some of the most promising venues in this section.

8.2.1 Instrumentation

Chapters 2 and 4 identify projects related to our path tracing and optimized coverage mechanisms that could be adopted into `csi-cc`. For example, prior work [15, 173] specializes path profiling to a subset of all paths through a procedure; coupled with an analysis that determines the most "interesting" paths for failure analysis, these techniques may apply in our scenario. Similarly, other techniques extend path profiles beyond single-procedure acyclic paths [13, 51, 104, 115, 156, 164, 164]; the trade-offs between tracing cost and path trace detail would be interesting to explore. For binarized coverage, existing work [38, 85, 117, 118, 141, 169] dynamically inserts and deletes probes after they are first triggered. The overhead savings here are often very large; an unintrusive method of dynamic code modification would complement our optimizations from Chapter 4.

A key open question stemming from Chapter 4 is how to improve the efficiency of our coverage optimizations. One primary target for improvement is the search for ambiguous triangles in our locally optimal formulation. Despite the optimizations discussed in that section, our approach performs a large amount of redundant computation by re-exploring many sub-paths in the ambiguous triangles search. We have not proven a lower bound on the complexity of this search, and a dynamic-programming solution or other memoization techniques could improve performance. The fully-optimal approach from Chapter 4 may also benefit from these improvements if adapted to use the sufficiency condition involving ambiguous triangles from Section 4.2.1 more directly.

Another possible direction is to consider special cases where coverage optimization can be much more efficient. For example, the "superblock" approach that combines a procedure's dominator and post-dominator trees, originally from Agrawal [4], appears to be near-optimal for reducing coverage instrumentation at all statements over complete executions. Ball and Larus [21] categorize instances of optimization for statement counts (i.e., where probes are counters rather than Boolean-valued indicators) as undirected-cycle-breaking problems, and analyze the complexity of these instances. No similar categorization exists for binarized

coverage, and the complexity of important special cases (e.g., coverage instrumentation within loops) remains an open question.

Our coarse-grained coverage mechanisms proved a useful complement to dense, stack-local path traces. However, one might consider moving beyond coverage at call sites, and instead capture occasional calling-context snapshots; prior work in precisely encoding call contexts may be applicable [30, 162]. All of our tracing techniques from Part I focus on control flow, though our PDG restriction analysis from Chapter 5 also disambiguates data flow. To leverage aspects of data flow in our tracing, analyses such as those from Yuan et al. [192] to identify "most-useful" variables may provide a starting point.

Our instrumentation from Chapter 3 proved efficient in supporting post-deployment tracing customization. However, there we assumed that a developer would manually select tracing schemes for each function in the program. This selection could be a daunting task, as it can be difficult to predict which trace information will be particularly valuable for unknown failures; the value of trace data certainly is not uniform across applications or procedures. We describe some *static* filtering in Chapter 3 that reduces redundant tracing. Future work might consider *dynamic* improvements to tracing, perhaps beginning with all functions tracing no data, then gradually refining tracing based on previously-observed failures, building on our "Realistic" configuration from Chapters 2 and 3. Additional tracing, in principle, should reduce the *entropy* [158] or *ambiguity* of analysis results over future failing runs; however, many open questions remain, including how to calculate this entropy measure, and how to estimate the cost of enabling specific run-time tracing.

8.2.2 Postmortem Analysis

While our analyses from Chapters 5 and 6 efficiently reduce ambiguity in failing executions and answer a wide range of control-flow queries based on a failure report, all approaches are defined over just a single failure report. Future work could aggregate data from multiple failures. For example, building on our PDG restriction analysis from Chapter 5, unions of slices [27, 124] (though not necessarily valid slices [50]) could help to generalize from a single failing run, while intersections of slices could highlight failure "hot spots." Many real-world software products use heuristics to cluster related failure reports [45, 65, 122], and much research also exists on the topic [43, 52–54, 148, 177]. Traced data or analysis results could improve the accuracy of clustering techniques.

Chapter 6 introduces the unreliable trace languages (UTL), and gives an algorithm to answer user queries in polynomial time over a failure report consisting of UTL constraints. This

polynomial-time decidability appears to be tightly linked to the UTL class (see Appendix A.2). However, other extensions to UTL may be possible, and the theoretical relationship between UTL and polynomial-time answers to queries over control-flow graphs remains open for exploration. We discuss negation for a very limited subclass of UTL in Section 6.5.3 of this chapter: specifically, negation for single-statement observations. This allowed us to answer user-level queries as *Yes*, *No*, or *Maybe* (see the best-effort coverage problem from Section 6.7) when solvers only answer individual queries as *Possible* or *Impossible*. Whether one could answer queries in polynomial time while allowing negation for a larger subclass of UTL remains an open question.

Formally, this classification of *Yes*, *No*, and *Maybe* closely resembles the problem of evaluating a formula with respect to a partial (i.e., incomplete) model. For the latter, the best possible result is given by the formula's *supervaluational meaning* [33, 151, 172, 188]. For our query recovery problems, the control-flow graph and failure report encodings play the role of an incomplete model, and the user's query corresponds to the formula to be evaluated. Our language-based definitions from Chapter 6 have analogs in formal logic; for example, regular languages are definable in monadic second-order logic [34]. The depth and implications of this connection remain open for exploration.

Recent work [35, 36, 42, 78, 79, 153, 193] uses symbolic execution to synthesize inputs matching failure data at various granularities. Our combination of lightweight tracing and flexible postmortem analysis proved effective in significantly reducing failing execution ambiguity. Whether our analyses are used as a pre-pass over the program's CFG or our trace data is used directly as constraints to a symbolic execution engine, our techniques throughout this dissertation could prove useful in synthesizing failing inputs via symbolic execution. For example, Jin and Orso [78] treat traced information as a series of goals; one could likely encode `csi-cc` data in a similar framework.

Fault localization using post-deployment failure data has been a very popular research topic [2, 23, 79, 82, 102, 106, 150, 195]. The techniques in this dissertation—whether for a single failure or in aggregate—might similarly hold promise in the realm of fault localization. Future work could consider techniques based on slice-based fault localization [102], or investigate whether nuggets of traced data can serve directly as features correlated with failing execution.

All analysis techniques from Chapters 5 and 6 operate over single-threaded applications. All tracing from Part I of this dissertation is safe and valid for concurrent execution, but the data does not specifically target concurrency primitives. Since parallel applications are

becoming pervasive in the modern era, analyzing threaded or distributed applications is an important future research direction.

A study similar to the human subjects evaluation outlined in Chapter 7 would be a complement to the experiments from this dissertation. We have shown that our tracing and analysis techniques can effectively reduce ambiguity in program executions from post-deployment failure reports. Detailed studies of how developers debug in practice, and the impact of tools like `CSIclipse`, would greatly benefit developers and tool designers. In the long term, powerful tracing and analysis tools combined with studies of how to best present their data could significantly improve the debugging experience for post-deployment failures.

Part IV

Appendices

A.1 Ambiguous Triangle Correctness Proof

In this appendix, we show that our two coverage set characterizations from Chapter 4 are equivalent; that is, we show that coverage sets as defined in Definition 4.3 are equivalent to the ambiguous triangles characterization from Section 4.2.1. More specifically, we prove the contrapositive by negating our original definitions (which corresponds with how they are actually used algorithmically): We show that there exists at least one ambiguous triangle formed of paths from $P_{\alpha\beta}$, $P_{\alpha d}$, and $P_{d\beta}$—with Y_1 and Y_2 sets as defined in Section 4.2.1—if and only if there exist two paths p_1 and p_2 that differ only in a desired node, violating Definition 4.3.

For this proof, we are given a control-flow graph, $G = (V, E)$, with entry vertex, e, and input sets S, D, and X. From Definition 4.3, note that S is not a coverage set of D iff:

$$\begin{aligned}&\exists x \in X \text{ and } \exists p_1, p_2 \in e \rightarrow x \text{ such that}\\ &V(p_1) \cap S = V(p_2) \cap S \ \wedge \ V(p_1) \cap D \neq V(p_2) \cap D\end{aligned}$$

From Section 4.2.1, further note that the triple (α, β, d) where

$$\alpha \in S \cup \{e\} \qquad \beta \in S \cup X \qquad d \in D \setminus S$$

forms an ambiguous triangle for G, S, and X iff:

$$Y_1 \neq \emptyset \ \wedge Y_2 \neq \emptyset \ \wedge P_{\alpha d} \neq \emptyset \ \wedge P_{\alpha\beta} \neq \emptyset \ \wedge P_{d\beta} \neq \emptyset$$

for (α, β, d) (using the additional path sets defined in Section 4.2.1).

Theorem A.1. *For a given control-flow graph, $G = (V, E)$, with entry vertex, e, and input sets S, D, and X:*

$$\exists \text{ ambiguous triangle } (\alpha, \beta, d) \iff S \text{ is not a coverage set of } D$$

We are thus proving a dual implication, and, in the remainder of this appendix, we cover each implication in a separate section. Specifically, Theorem A.1 follows directly from Lemma A.2 and Lemma A.5 in the following sections. We abuse notation slightly, and use

$V(\dots)$ to indicate the set of vertices encountered along a single path or a set of paths. Thus, for a set of paths P,

$$V(P) = \bigcup_{p \in P} V(p)$$

A.1.1 First Implication

In this section, we prove that ambiguous triangles correspond to two actual paths violating Definition 4.3.

Lemma A.2. *For a given control-flow graph, $G = (V, E)$, with entry vertex, e, and input sets S, D, and X:*

$$\exists \text{ ambiguous triangle } (\alpha, \beta, d) \implies S \text{ is not a coverage set of } D$$

Note that this section uses destructive assignment for computed paths y_1 and y_2 to simplify presentation.

Proof. The proof is constructive, and forms appropriate paths p_1 and p_2 violating Definition 4.3 given an ambiguous triangle (α, β, d). The construction proceeds in four steps, as follows.

Step 1: Pick triangle Select any

$$p_{\alpha\beta} \in P_{\alpha\beta} \qquad p_{\alpha d} \in P_{\alpha d} \qquad p_{d\beta} \in P_{d\beta}$$

from the ambiguous triangle computation such that there is only one occurrence of d along path $p_{\alpha d} \circ p_{d\beta}$. (Note that this is always possible: simply remove all nodes between the first and last occurrence of d on any chosen path with multiple occurrences of d.) Thus, for the remainder of the proof, we will consider the "Connected Excluding" predicate along $p_{\alpha d}$ and $p_{d\beta}$ to be "$\xrightarrow{\notin S \setminus Y \cup \{d\}}$," except, of course, that we allow d to occur as the last vertex in $p_{\alpha d}$ and the first vertex in $p_{d\beta}$.

Step 2: Form y_1 Next, we form the appropriate $y_1 \in Y_1$ for the selected paths. We start with

$$y_1 = \pi \in Y_1 \text{ such that } V(\pi) \supseteq V(p_{d\beta}) \cap V(Y_1)$$

Note that such a path must exist. All of the vertices in π can occur along $e \xrightarrow{\notin\{d\}} \alpha$ paths, because all vertices in π are selected from $V(Y_1)$. Furthermore, the vertices can all occur in the same path, because they occur in order along $p_{d\beta}$ (where the "Connected Excluding" predicate is *more* strict: it is "$\xrightarrow{\notin\{d\}}$" for y_1, but "$\xrightarrow{\notin S\setminus Y\cup\{d\}}$" for $p_{d\beta}$).

Then

$$\forall v \in V(p_{\alpha\beta} \cup p_{\alpha d}) \cap V(Y_1) :$$
$$\text{update } y_1 = y_1 \circ \pi \text{ where } \pi \in \alpha \xrightarrow{\notin\{d\}} v \xrightarrow{\notin\{d\}} \alpha$$

Again, note that the required paths must always exist. The $\alpha \xrightarrow{\notin\{d\}} v$ path exists because it already occurs in $p_{\alpha\beta}$ or $p_{\alpha d}$. The $v \xrightarrow{\notin\{d\}} \alpha$ path exists because $v \in V(Y_1)$.

Step 3: Form y_2 Finally, we form the appropriate $y_2 \in Y_2$ for the selected paths. Similar to the y_1 case, we start with

$$y_2 = \pi \in Y_2 \text{ such that } V(\pi) \supseteq V(p_{\alpha d}) \cap V(Y_2)$$

By a parallel argument from the previous case, such a path must exist. All of the vertices in π can occur along $\beta \xrightarrow{\notin\{d\}} x$ paths (for at least one $x \in X$), because all vertices in π are selected from $V(Y_2)$. Furthermore, the vertices can all occur in the same path (ending at some $x \in X$), because they occur in order along $p_{\alpha d}$ (where the "Connected Excluding" predicate is *more* strict: it is "$\xrightarrow{\notin\{d\}}$" for y_2, but "$\xrightarrow{\notin S\setminus Y\cup\{d\}}$" for $p_{\alpha d}$).

Then

$$\forall v \in V(p_{\alpha\beta} \cup p_{d\beta}) \cap V(Y_2) :$$
$$\text{update } y_2 = \pi \circ y_2 \text{ where } \pi \in \beta \xrightarrow{\notin\{d\}} v \xrightarrow{\notin\{d\}} \beta$$

As before, the required paths must always exist. The $\beta \xrightarrow{\notin\{d\}} v$ path exists because $v \in V(Y_2)$. The $v \xrightarrow{\notin\{d\}} \beta$ path exists because it already occurs in $p_{\alpha\beta}$ or $p_{d\beta}$.

Step 4: Form p_1 and p_2 We can now form the required paths p_1 and p_2 that violate Definition 4.3. They are:

$$p_1 = y_1 \circ p_{\alpha\beta} \circ y_2$$
$$p_2 = y_1 \circ p_{\alpha d} \circ p_{d\beta} \circ y_2$$

Both paths contain the same set of observation vertices:

$$\Big(V(y_1) \cup V(y_2) \cup \{\alpha, \beta\}\Big) \cap S$$

However, $d \notin V(p_1)$ but $d \in V(p_2)$. □

A.1.2 Second Implication

In this section, we prove that any two paths that violate Definition 4.3 imply the existence of a closely-related ambiguous triangle, as defined in Section 4.2.1. We begin by defining new terminology in order to prove a necessary sub-lemma, before proceeding to prove the primary lemma of this section.

Definition A.3. *For vertex v and path π, occurrences$(v, \pi) = \{i$ such that $\pi_i = v\}$.*

Lemma A.4. *If there exists $p_1, p_2 \in e \rightarrow x$ such that:*

- $V(p_1) \cap S = V(p_2) \cap S$
- $d \notin p_1$
- $|occurrences(d, p_2)| > 1$

then there exists $p_3 \in e \rightarrow x$ such that:

- $V(p_1) \cap S = V(p_2) \cap S = V(p_3) \cap S$
- $|occurrences(d, p_3)| = 1$

Proof. Consider each index $i \in occurrences(d, p_2)$. We begin by finding j, the index of the first observation preceding index i in p_2:

$$1 \leq j < i \text{ such that } p_{2_j} \in S \cup \{e\} \text{ and}$$
$$\forall j' \text{ such that } j < j' < i, j' \notin S \cup \{e\}$$

We then find k, the index of the first observation succeeding index i in p_2:

$$i < k \leq |p_2| \text{ such that } p_{2_k} \in S \cup \{x\} \text{ and}$$
$$\forall k' \text{ such that } i < k' < k, k' \notin S \cup \{x\}$$

Next we find any:

$$a \text{ such that } p_{1_a} = p_{2_j}$$
$$b \text{ such that } p_{1_b} = p_{2_k}$$

Note that vertices for a and b must exist, since $V(p_1) \cap S = V(p_2) \cap S$.

Case 1: If $b < a$, we can construct:

$$p_3 = \langle p_{1_1} \dots p_{1_a} \rangle \circ \langle p_{2_j} \dots p_{2_k} \rangle \circ \langle p_{1_b} \dots p_{1_{|p_1|}} \rangle$$

Because $p_{1_1} = e$ and $p_{1_{|p_1|}} = x$, we know that $p_3 \in e \rightarrow x$. Further, note that $\langle p_{2_j} \dots p_{2_k} \rangle$ contains exactly one occurrence of d, because we selected j to be the first observation vertex index prior to i, and k to the be first observation vertex index following i. In effect, we are "stitching" part of path p_2 (specifically, a sequence that contains exactly one occurrence of d) into p_1 (which contains no occurrences of d, by assumption). At this point, we have found an appropriate p_3, and not need consider further indices from $occurrences(d, p_2)$.

Case 2: If $a \leq b$, we note the following possible substitution:

$$\langle p_{2_j} \dots p_{2_k} \rangle \longmapsto \langle p_{1_a} \dots p_{1_b} \rangle$$

which would effectively "remove" the occurrence of d at p_{2_i} by "stitching in" a part of p_1. Note that any such substitution would not remove any observation vertices from p_2, because j and k are the most directly adjacent observations to i in p_2.

If we perform all of the above substitutions for every index in $occurrences(d, p_2)$, we would end up with a path containing no occurrences of d. Hence, if no index i fits the $b < a$ case above, we can construct p_3 by applying *all but one* of the above substitutions to our original p_2, leaving exactly one occurrence of d. □

With this sub-lemma available, we now proceed to prove the primary lemma of this section.

Lemma A.5. *For a given control-flow graph, $G = (V, E)$, with entry vertex, e, and input sets S, D, and X:*

$$S \text{ is not a coverage set of } D \implies \exists \text{ ambiguous triangle } (\alpha, \beta, d)$$

Proof. The proof is again constructive, and forms an ambiguous triangle (α, β, d) from two paths p_1 and p_2 that show that S is not a coverage set of D (via Definition 4.3). At a high level, the proof selects appropriate α and β vertices that occur in both p_1 and p_2, which creates a resulting Y set. Importantly, the selected α and β are such that the three sub-path "legs" of the resulting triangle (paths $p_{\alpha\beta}$, $p_{\alpha d}$, and $p_{d\beta}$ as pictured in Fig. 4.2 from Section 4.2.1) contain only vertices from p_1 and p_2, and no vertices from $S \setminus Y$.

We begin by assuming that we are given a pair of paths, p_1 and p_2, that serve as witness to the violation of Definition 4.3. Then, by Definition 4.3, we know that:

$$d \notin V(p_1) \qquad d \in V(p_2)$$
$$V(p_1) \cap S = V(p_2) \cap S$$

Further, we assume that d occurs *exactly once* in p_2. By Lemma A.4, such a single-d p_2 is guaranteed to exist if any p_2 exists that satisfies the constraints above. The general approach of this proof is to find an appropriate α and β such that the set $Y = V(Y_1) \cup V(Y_2)$ contains all vertices in $V(p_1) \cap S$, while maintaining the necessary reachability relations between α, β, and d to form the triangle.

The construction proceeds in three steps, as follows.

Step 1: Initialize the triangle Initialize the working triangle (not yet an ambiguous triangle by our definition from Section 4.2.1) as follows:

$$p_{ex} = p_1$$
$$p_{ed} = \langle p_{2_1} \dots p_{2_n} \rangle \text{ such that } p_{2_n} = d$$
$$p_{dx} = \langle p_{2_n} \dots p_{2_{|p_2|}} \rangle$$

Note that, at this stage, $V(p_{ex}) \cap S = V(p_{ed} \circ p_{dx}) \cap S$ by construction. This further implies both

$$V(p_{ed}) \cap S \subseteq V(p_{ex}) \cap S$$
$$V(p_{dx}) \cap S \subseteq V(p_{ex}) \cap S$$

Step 2: Find α Find vertex $v \in S \cup \{e\}$ such that $p_{ex_i} = p_{ed_j} = v$ and where j is maximized. That is, we find the latest occurrence of a shared vertex between p_{ex} and p_{ed}. Then, let

$$\alpha = v$$

After this change, we now have an incomplete triangle formed by α and d (we have yet to select β). Thus, we can define the following paths:

$$p_{\alpha x} = \langle p_{1_i} \dots p_{1_{|p_1|}} \rangle$$
$$p_{\alpha d} = \langle p_{2_j} \dots p_{2_n} \rangle$$

We can now compute the set Y_1. We have at least two (possibly identical) paths in Y_1: the subpath from e to α from p_{ex} (i.e., $\langle p_{1_1} \dots p_{1_i} \rangle$), and the subpath from e to α from p_{ed} (i.e., $\langle p_{2_1} \dots p_{2_j} \rangle$). Note that this adds all earlier vertices in p_{ex} and p_{ed} to $V(Y_1)$, since they now exist along some $e \xrightarrow{\notin \{d\}} \alpha$ path.

The first key observation is that $V(p_{\alpha d}) \cap S = \emptyset$, since we chose j to be the latest observation in p_{ed}. Thus, all vertices in $V(p_{ed}) \cap S$ will be included in $V(Y_1)$ after this step. Recall that $V(p_{ex}) \cap S = \big(V(p_{ed}) \cup V(p_{dx})\big) \cap S$ by our initial triangle construction. Thus, we can further conclude that $V(p_{ex}) \cap \big(S \setminus V(Y_1)\big) = V(p_{\alpha x}) \cap \big(S \setminus V(Y_1)\big) = V(p_{dx}) \cap \big(S \setminus V(Y_1)\big)$.

Step 3: Find β Now, we simply do a similar procedure in reverse for β. We find vertex $v \in S \cup X$ such that $p_{\alpha x_k} = p_{dx_\ell} = v$ and where k is minimized. That is, we find the earliest occurrence of a shared vertex between $p_{\alpha x}$ and p_{dx}. Then, let

$$\beta = v$$

After this change, we now have a complete ambiguous triangle formed by α, β, and d. Thus, we can define the following paths:

$$p_{\alpha\beta} = \langle p_{1_i} \dots p_{1_k} \rangle$$
$$p_{d\beta} = \langle p_{2_n} \dots p_{2_\ell} \rangle$$

We can now compute the set Y_2. We have at least two (possibly identical) paths in Y_2: the subpath from β to x from $p_{\alpha x}$ (i.e., $\langle p_{1_k} \dots p_{1_{|p_1|}} \rangle$), and the subpath from β to x from p_{dx} (i.e., $\langle p_{2_\ell} \dots p_{2_{|p_2|}} \rangle$). Note that this adds all later vertices in $p_{\alpha x}$ and p_{dx} to $V(Y_2)$, since they now exist along some $\beta \xrightarrow{\notin\{d\}} x$ path.

To show that the triple (α, β, d) forms an ambiguous triangle, it will suffice to show that $V(p_{ex}) \cap \big(S \setminus V(Y_1 \cup Y_2)\big) = \emptyset$, since, by our initial assumption about paths p_1 and p_2 from Definition 4.3, $V(p_{ex}) \cap S = V(p_{ed} \circ p_{dx}) \cap S$. Recall that, after Step 2, we concluded that $V(p_{\alpha x}) \cap \big(S \setminus V(Y_1)\big) = V(p_{dx}) \cap \big(S \setminus V(Y_1)\big)$. Therefore, since we chose k to be the earliest shared observation in $p_{\alpha x}$, we observe that $V(p_{\alpha\beta}) \cap \big(S \setminus V(Y_1)\big) = \emptyset$. Trivially, then, $V(p_{\alpha\beta}) \cap \big(S \setminus V(Y_1 \cup Y_2)\big) = \emptyset$. Recall that p_{ex} is equivalent to $(y_1 \in Y_1) \circ p_{\alpha\beta} \circ (y_2 \in Y_2)$. Thus, $V(p_{ex}) \cap \big(S \setminus V(Y_1 \cup Y_2)\big) = \emptyset$. We have therefore constructed an ambiguous triangle (α, β, d) by assuming the existence of two paths, p_1 and p_2, that violate Definition 4.3. □

A.2 Proofs of NP-hardness for Two Generalized Trace Language Classes

Substantial portions of this appendix are derived from a 2017 technical report by Ohmann et al. [131].

Unfortunately, the unreliable trace languages (described in Section 6.5.1) are a very tight class. Specifically, the useful property of polynomial-time decidability for queries in unreliable trace languages appears not to generalize beyond this restrictive class. Here, we prove that two small extensions of the unreliable trace languages (to more generalized language families describing program trace properties) result in language intersection problems with NP-hard complexity.

Throughout this section we will often use regular expressions to define regular languages, though most of our techniques are defined over automata. The equivalence of regular expressions and finite-state automata is well-known, and, for each of the trace languages we

define, this conversion is trivial and requires no more than a constant-factor increase from the number of regular expression terms to the number of FSA transitions.

A.2.1 Allowing Ambiguity

The first generalization that we consider relaxes the requirement that each constraint be composed of a sequence of *characters*, instead allowing a sequence of *character classes*. Specifically, we consider the class of languages:

$$\mathcal{A} = \big\{\Sigma^* C_1 \Sigma^* C_2 \ \ldots \ \Sigma^* C_n \Sigma^* \\ \text{for } n \geq 0 \text{ and such that all } C_i \subseteq \Sigma\big\}$$

Theorem A.6. *The context-insensitive query recovery problem from Definition 6.7 is NP-hard if all $FP_i \in \mathcal{A}$ and $R \in \mathcal{A}$.*

Proof. The proof is via a straightforward reduction from Boolean SAT on formulae in conjunctive normal form (CNF) [84].

We are given a CNF formula:

$$f = (p_{1,1} \vee p_{1,2} \vee \ldots) \wedge (p_{2,1} \vee p_{2,2} \vee \ldots) \wedge \ldots$$

We begin by converting each conjunct into a regular expression as follows:

$$f_i = p_{i,1} \vee p_{i,2} \vee \cdots \vee p_{i,n}$$
$$\Downarrow$$
$$r_i = \Sigma^* \ [p_{i,1} \, p_{i,2} \ \ldots \ p_{i,n}] \ \Sigma^*$$

Our alphabet, Σ, is comprised of all literals in all of these conjunct formulae, along with their negations. That is, from the single conjunct

$$\pi_1 \vee \overline{\pi_2} \vee \pi_3$$

we would add the following characters to Σ:

$$\pi_1, \ \overline{\pi_1}, \ \pi_2, \ \overline{\pi_2}, \ \pi_3, \ \overline{\pi_3}$$

Note that all $r_i \in \mathcal{A}$; that is, all r_i are generalized trace languages by our extended definition. Furthermore, all r_i are languages over Σ.

Next, we need to construct a language to enforce the principle of excluded middle. In terms of our trace language, this corresponds to recognizing only strings with exactly one of each literal (from our original CNF formula) or its negation. To do so, we construct our principle of excluded middle regular expression:

$$L(M) = [\sigma_1\ \overline{\sigma_1}][\sigma_2\ \overline{\sigma_2}] \ldots [\sigma_{|\Sigma|}\ \overline{\sigma_{|\Sigma|}}]$$

from each $\sigma_i \in \Sigma$. This language recognizes those strings assigning "true" or "false" to each literal appearing in the original CNF formula. Note that $L(M)$ corresponds exactly to $L_i(G)$ for a control-flow graph, G, that is a sequence of branches. Now, recall that each r_i enforces one of the original conjuncts of f. Thus, if

$$\bigcap_i (r_i) \cap L(M) \neq \emptyset$$

the original formula (f) is satisfiable. Each of the above steps is a straightforward linear transformation from our original formula into a regular language, and, hence, the whole transformation is clearly polynomial. Therefore, determining intersection-emptiness for this generalized class of trace languages is NP-hard. □

A.2.2 Constrained Paths

The second generalization that we consider relaxes the requirement that each constraint have no detail on what happens during execution between observation points. Specifically, if our original alphabet is Σ_o, we consider the class of languages:

$$\mathcal{B} = \Big\{ C^*\,\sigma_1\,C^*\,\sigma_2\,C^*\ \ldots\ C^*\,\sigma_p\,C^* \\ \text{for } p \geq 0,\ C \subseteq \Sigma_o \text{ and such that all } \sigma_i \in \Sigma_o \Big\}$$

For the proof, we require a slightly extended alphabet:

$$\Sigma = \Sigma_o \cup \{\#\}$$

where the symbol "#" is not present in Σ_o, and is used only as a marker in the proof.

Theorem A.7. *The context-insensitive query recovery problem from Definition 6.7 is NP-hard if all $FP_i \in \mathcal{B}$ and $R \in \mathcal{B}$.*

Proof. The proof is via reduction from the decision version of the Shortest Common Supersequence Problem (SCSP) on any alphabet (including a binary alphabet), which is proven NP-complete by Räihä and Ukkonen [149]. The following input characterizes a SCSP over an arbitrary alphabet:

- a size, z
- a set of sequences $S = s_1, s_2, \ldots, s_n$ such that for all $s_i \in S$, all characters s_{i_j} are members of alphabet $\Sigma_o = \Sigma \setminus \{\#\}$.

Solving the SCSP requires that one find a string R such that $|R| \leq z$ and for all $s_i \in S$, s_i is an ordered subsequence of R (that is, s_i is obtained by deleting zero or more elements from R).

We begin by converting each s_i to a regular expression:

$$\begin{aligned} s_i &= s_{i_1}\, s_{i_2}\, \ldots\, s_{i_m} \\ &\Downarrow \\ r_i &= \Sigma^*\, s_{i_1}\, \Sigma^*\, s_{i_2}\, \Sigma^*\, \ldots\, \Sigma^*\, s_{i_m}\, \Sigma^* \end{aligned}$$

Note that all $r_i \in \mathcal{B}$—that is, all r_i are generalized trace languages by our extended definition (with $C = \Sigma$)—and this conversion is a simple enumeration of the sequence (and, hence, is clearly polynomial). The intersection

$$\bigcap_i r_i$$

is always non-empty, as it always contains the concatenation of sequences $s_1\, s_2\, \ldots\, s_n$. However, any string in this intersection that contains no more than z characters from Σ_o serves as a witness for the original SCSP. Now, consider the language

$$L(Z) = (\Sigma_o^*\, \#)^z\, \Sigma_o^*$$

This language recognizes exactly those strings with z "#" characters. Furthermore, $L(Z) \in \mathcal{B}$ (it is a generalized trace language with $C = \Sigma_o$). Finally, consider the language

$$L(V) = (\#\, \Sigma_o^?)^*$$

This language requires that every character from Σ_o be immediately preceded by a "#" character. Furthermore, note that $L(V)$ corresponds exactly to $L_i(G)$ for a control-flow graph, G, that contains an infinite loop containing a large switch branching to nodes labeled with each $\sigma \in \Sigma_o$, all branching back together into the single entry node labeled "#". More concretely, $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$ where:

$$N = \Sigma \qquad n_0 = \# \qquad \mathbb{L} = \emptyset$$
$$E_i = \{(\#, \sigma) \, \forall \sigma \in \Sigma\} \cup \{(\sigma, \#) \, \forall \sigma \in \Sigma\}$$
$$E_c = \emptyset \qquad E_r = \emptyset$$

The language

$$L(Z) \cap L(V)$$

contains those strings with exactly z "#" characters, where each "#" is optionally followed by a single character from Σ_o. Hence, this language contains at most z characters from Σ_o.

Now, if the language

$$L(R) = \bigcap_i (r_i) \; \cap L(Z) \cap L(V)$$

is non-empty, then a supersequence of size $\leq z$ exists for the original input sequences. This supersequence is comprised of the ordered sequence of characters from Σ_o (i.e., excluding all instances of "#") in any witness for $L(R)$. Note that $L(R)$ is an intersection of generalized trace languages (i.e., languages from $\mathcal{B}$) with a feasible control-flow graph language $L_i(G)$, and the above procedure checks this intersection for emptiness to obtain a solution to the original SCSP. All transformations in generating $L(R)$ are polynomial. Therefore, determining intersection-emptiness for this generalized class of trace languages is NP-hard. □

A.3 Proofs of Correctness For Unreliable Trace Language Intersection

Substantial portions of this appendix are derived from a 2017 technical report by Ohmann et al. [131].

In this section, we present partial correctness proofs for our approach to checking intersection-emptiness from Section 6.5.2. Specifically, we prove that our procedure—given

a control-flow graph, $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$, and a vector of unreliable trace constraint vectors V that correspond to a set of failure report elements FP and a query R—answers *Possible* if and only if the language intersection $L_i(G) \cap \bigcap_j FP_j \cap R$ (by Definition 6.7) is non-empty.

Theorem A.8. *If our data-flow analysis procedure reports Possible, then the context-insensitive intersection $L_i(G) \cap \bigcap_j FP_j \cap R$ is non-empty.*

Proof. Note that for any node $n \in sccG$, either $in(n) = \bot$ or $in(n) = out(m)$ for some immediate predecessor m of n. If our procedure reports *Possible*, then there exists a path p through $sccG$ where $n_0 \in p_1.\text{nodes}$, $in(p_i) = out(p_{i-1})$ for all $i \in 2 \ldots |p|$, and $out(p_{|p|})$ is a vector of empty vectors. We will use this path through $sccG$ to form a witness to the non-empty intersection.

Consider the difference between $in(p_i)$ and $out(p_i)$. This difference contains the set of constraint observations that were consumed when passing through p_i. For each constraint FP_j, the difference corresponds to a sequence of consumed symbols, c_j. We then form the string $s_i = c_1 \shuffle c_2 \ldots \shuffle c_{|c|}$. This string is not necessarily a substring of any trace in $L_i(G)$. Fortunately, all symbols in s_i correspond to nodes or edge labels from G that are mutually reachable from one another (i.e., they are in the same strongly-connected component). We can form a new string, s'_i, that corresponds to a partial trace from $L_i(G)$, and contains s_i as a subsequence. Therefore, s'_i contains, in order, all symbols consumed by every constraint at p_i. If the resulting s'_i is empty, we update s'_i to contain a single arbitrary node from $p_i.\text{nodes}$.

For every adjacent pair (s'_i, s'_{i+1}), the final symbol in s'_i, α, belongs to strongly-connected component p_i (from our path p) and the first symbol from s'_{i+1}, β, belongs to strongly-connected component p_{i+1} (also from the path p). Thus, there exists some partial trace w_i from $L_i(G)$ where $w_{i_1} = \alpha$ and $w_{i_{|w_i|}} = \beta$. The following string is a witness to the non-empty intersection $L_i(G) \cap \bigcap_j FP_j \cap R$:

$$s'_1 \shuffle w_1 \shuffle s'_2 \shuffle w_2 \shuffle \ldots \shuffle s'_{|s|}$$

□

Theorem A.9. *If $L_i(G) \cap \bigcap_j FP_j \cap R$ is non-empty, then our data-flow analysis procedure reports Possible.*

Proof. If we know that the intersection $L_i(G) \cap \bigcap_j FP_j \cap R$ is non-empty, then there must exist some witness string w over $\widehat{N_{\mathbb{L}}}$ within the intersection. The context-insensitive condensation

of G, $sccG$, induces a mapping from symbols in $\widehat{N_{\mathbb{L}}}$ to nodes in $sccG$. We can thus, given a witness string w, construct a sequence of substrings $w_1, w_2, \ldots, w_k$ where each w_i consists of exactly those symbols in w that mapped to the same strongly-connected component p_i.

The induced sequence of strongly-connected components $p = \langle p_1, p_2, \ldots, p_k \rangle$ describes a path through $sccG$. Note that all symbols appearing in all unreliable trace language vectors V_j appear in the witness w, so the consume() function will make no progress in any node $n \in sccG$ where n does not occur in p. That is, consume(n, f) = f for n that do not occur in p. This means that for all $i \in 2 \ldots |p|$, $in(p_i) = out(p_{i-1})$. Further, no merge() operation ($\sqcap$) can possibly return $\bot$.

Thus, we simply need to show that, given path p over $sccG$, our data-flow approach consumes all of the observations so that $out(p_{|p|})$ is a vector of empty vectors. Concretely, $F_{p_{|p|}}(F_{p_{|p|-1}}(\ldots F_{p_1}(V)))$ is the vector of $|V|$ empty vectors. Consider a single constraint $V_j = \langle V_{j_1}, V_{j_2}, \ldots, V_{j_m} \rangle$. Each symbol V_{j_k} appears in exactly one strongly-connected component, p_i, from the path p. For a given pair of observations V_{j_a}, V_{j_b} with $a < b$, V_{j_a} appears in some p_α, and V_{j_b} appears in some p_β. Note that V_{j_a} appears before V_{j_b} in w, because the sequence of symbols in V_j is a subsequence of w. Since $sccG$ is a condensation of G, we therefore know that $\alpha \leq \beta$.

Now, all symbols from V occur somewhere within the path p, and moreover, they occur in order (as demonstrated). Recall (from Section 6.5.2) that consume(p_i, $in(p_i)$) will greedily consume the longest prefix, x, of each vector in $in(p_i)$ where each symbol $x_n \in p_i$.nodes. Each requirement vector is consumed independently in parallel as we call consume() over each p_i, and, as previously demonstrated, our approach will never merge facts to $\bot$ in the given problem instance. This allows us to conclude that the data-flow procedure consumes all symbols from each $V_j \in V$, and $out(p_{|p|})$ is a vector of empty vectors as desired. We then report "Possible". □

B FULLY-OPTIMAL COVERAGE OPTIMIZATION

Substantial portions of this appendix are derived from a 2016 technical report by Ohmann et al. [133].

Obtaining an optimal solution to the Customized Coverage Probing Problem is an NP-hard global optimization problem, as shown in Section 4.1.5. In this appendix, we provide a detailed description of how we construct our optimal solution as a 0–1 mixed-integer linear optimization problem based on the sufficiency condition for a coverage set from Section 4.2.1. The full MILP itself is shown in Fig. B.1; we describe each piece in its construction, specifically focusing on some of the key "widgets" making up the different constraints in our formulation.

In our descriptions, we assume that we are given input per Chapter 4, Section 4.1.1:

- $G = (V, E)$, a directed graph with vertices V and edges E
- $e \in V$, a unique source (or entry) vertex with in-degree 0
- $I \subseteq V$, a subset of vertices that may be probed
- c_i, the cost of probing vertex $i \in I$, where $\forall i \in I, c_i > 0$
- $D \subseteq V$, a set of desired vertices
- $X \subseteq V$, a set of possible termination points

We build upon the sufficiency condition (and use the ambiguous triangle terminology) from Section 4.2.1.

To begin, for notational convenience, we define all possible ambiguous triangles for all possible sets $S \subseteq I$ as the set of triples of vertices

$$\mathcal{T} = \left\{ (\alpha, \beta, d) \in (I \cup \{e\}) \times (I \cup X) \times D \right\}$$

Then, for each $(\alpha, \beta, d) \in \mathcal{T}$, we define an additional set of vertices corresponding to set Y from Section 4.2.1. These are vertices occurring on paths from e to α or from β to a terminal vertex that do not cross vertex d:

$$Y_{\alpha\beta d} = \bigcup_{x \in X, \pi \in e \xrightarrow{\notin \{d\}} \alpha \cup \beta \xrightarrow{\notin \{d\}} x} V(\pi)$$

The sets $Y_{\alpha\beta d}$ can be constructed by simply checking basic graph connectivity, and we define the numerical parameter

$$a_{\alpha\beta di} = \begin{cases} 1 & \text{if } i \in Y_{\alpha\beta d} \\ 0 & \text{otherwise} \end{cases}$$

which is provided as input to our model.

The goal is to find S, a minimal-cost coverage set of D. We first introduce the binary selection variables

$$z_i = 1 \text{ iff } i \in S$$

to represent the selected coverage set. Next, we use five sets of binary variables, one for each path set in the characterization of a coverage set from Section 4.2.1, that, when set to 1, will force its set of paths to be empty:

$$\begin{aligned}
s_{\alpha d} = 1 \text{ will imply that } e &\xrightarrow{\not\in\{d\}} \alpha = \emptyset \\
t_{\beta d} = 1 \text{ will imply that } \beta &\xrightarrow{\not\in\{d\}} x = \emptyset \ \forall x \in X \setminus \{d\} \\
u_{\alpha\beta d} = 1 \text{ will imply that } \alpha &\xrightarrow{\not\in S \setminus Y_{\alpha\beta d}} d = \emptyset \\
v_{\alpha\beta d} = 1 \text{ will imply that } \alpha &\xrightarrow{\not\in (S \setminus Y_{\alpha\beta d}) \cup \{d\}} \beta = \emptyset \\
w_{\alpha\beta d} = 1 \text{ will imply that } d &\xrightarrow{\not\in S \setminus Y_{\alpha\beta d}} \beta = \emptyset
\end{aligned}$$

Recall from Section 4.2.1 that S is a coverage set of D if and only if at least one of these five sets of paths is empty for all $(\alpha, \beta, d) \in \mathcal{T}$. To force this condition, we thus introduce the constraint:

$$s_{\alpha d} + t_{\beta d} + u_{\alpha\beta d} + v_{\alpha\beta d} + w_{\alpha\beta d} \geq (1 - z_d) \ \forall (\alpha, \beta, d) \in \mathcal{T}$$

The key widget in our formulation is the ability to model, for $G = (V, E)$, whether or not there exists a path between vertices k and ℓ (i.e., whether $k \rightarrow \ell \neq \emptyset$). From basic network flow theory, $k \rightarrow \ell \neq \emptyset$ if and only if the inequality system

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ij} = \begin{cases} 1 & i = k \\ 0 & i \neq k, \ell \\ -1 & i = \ell \end{cases} \tag{B.1}$$

$$x_{ij} \geq 0 \quad \forall (i, j) \in E \tag{B.2}$$

has a solution. Farkas' Lemma, or basic linear programming duality theory [49], states that the above system does *not* have a solution if and only if there exist dual multipliers $\xi \in \mathbb{R}^{|V|}$

such that

$$\xi_i - \xi_j \geq 0 \quad \forall (i,j) \in E \tag{B.3}$$

$$\xi_k - \xi_\ell \leq -1. \tag{B.4}$$

Note that, in this case, we can safely bound the dual multipliers in the range $[-1, 1]$.

We use this widget as the basis of building our model. Note that we will require multipliers for many choices of starting nodes k and ending nodes ℓ. Specifically, for a fixed (α, β, d) triple, we must enforce the non-existence of one of five different sets of paths (from Section 4.2.1), so we again define five sets of variables. These variables are associated with the existence of a particular vertex $i \in V$ along each of the five paths; thus, for each class of variables, we require one variable for each $i \in V$. For each $(\alpha, \beta, d) \in \mathcal{T}$, the following are the vertex (dual) multipliers for the linear system (B.3)–(B.4).

$\theta_i^{\alpha\beta d}$: dual multipliers associated with $e \xrightarrow{\notin\{d\}} \alpha = \emptyset$

$\eta_i^{\alpha\beta d}$: dual multipliers associated with $\beta \xrightarrow{\notin\{d\}} x = \emptyset \; \forall x \in X \setminus \{d\}$

$\pi_i^{\alpha\beta d}$: dual multipliers associated with $\alpha \xrightarrow{\notin S \setminus Y_{\alpha\beta d}} d = \emptyset$

$\mu_i^{\alpha\beta d}$: dual multipliers associated with $\alpha \xrightarrow{\notin (S \setminus Y_{\alpha\beta d}) \cup \{d\}} \beta = \emptyset$

$\lambda_i^{\alpha\beta d}$: dual multipliers associated with $d \xrightarrow{\notin S \setminus Y_{\alpha\beta d}} \beta = \emptyset$

Returning to the higher-level goal, recall that each of the s, t, u, v, and w variables serves as a forcing variable, ensuring that a particular subpath from the ambiguous triangle is $\emptyset$. To take the simplest example, recall that if $s_{\alpha d} = 1$, we wish to enforce that $e \xrightarrow{\notin\{d\}} \alpha = \emptyset$. Thus, we must remove vertex d from the flow network given by (B.1). In the dual formulation, the equivalent operation is to remove the inequality (B.3) for all $(i, d) \in E$ and for all $(d, j) \in E$. Loosely, we model this constraint by removing all incoming and outgoing edges for d from G for these paths. Finally, our model should exclude $e \to \alpha$ paths only when $s_{\alpha d} = 1$ (recall: $s_{\alpha d}$ directly implies the condition $e \xrightarrow{\notin\{d\}} \alpha = \emptyset$). Thus, we need only enforce the forcing dual flow constraint (B.4) when $s_{\alpha d} = 1$. Algebraically, replacing the upper bound in (B.4) with $1 - 2s_{\alpha d}$ will serve this purpose, since, as previously stated, all dual multipliers are bounded

between $[-1, 1]$. Putting this logic together gives us the dual flow system

$$\begin{aligned} \theta_i^{\alpha\beta d} - \theta_j^{\alpha\beta d} &\geq 0 && \forall (\alpha, \beta, d) \in \mathcal{T}, \forall (i, j) \in E \mid i \neq d, j \neq d \\ \theta_e^{\alpha\beta d} - \theta_\alpha^{\alpha\beta d} &\leq 1 - 2s_{\alpha d} && \forall (\alpha, \beta, d) \in \mathcal{T} \end{aligned}$$

The remaining dual flow systems for η, π, μ, and λ are defined in a similar fashion. However, there are some important and subtle differences. One complication arising in the definition of the variables π (which corresponds to $P_{\alpha d}$), μ (which corresponds to $P_{\alpha\beta}$), and λ (which corresponds to $P_{d\beta}$), is that we must exclude the set of vertices $S \setminus Y$ from the appropriate flow networks. Specifically, we want constraint (B.3) to be redundant when either $i \in S \setminus Y_{\alpha\beta d}$ or $j \in S \setminus Y_{\alpha\beta d}$. Note that $i \in S \setminus Y_{\alpha\beta d}$ is equivalent to $z_i - a_{\alpha\beta di} z_i = 1$, since z_i indicates that $i \in S$ and $a_{\alpha\beta di}$ indicates that $i \in Y_{\alpha\beta d}$. Thus, for example, in the flow system for the variables π, constraint (B.3) is modified to be of the form

$$\pi_i^{\alpha\beta d} - \pi_j^{\alpha\beta d} \geq -(z_i - a_{\alpha\beta di} z_i) - (z_j - a_{\alpha\beta dj} z_j) \quad \forall (\alpha, \beta, d) \in \mathcal{T}, \forall (i, j) \in E.$$

If $t_{\beta d} = 1$, we wish to enforce that there is no path from β to any termination point that avoids passing through d. That is, $\beta \xrightarrow{\notin \{d\}} x = \emptyset$ for all $x \in X \setminus \{d\}$. To model this requirement, we introduce a special sink vertex χ with new edges (x, χ) for each $x \in X \setminus \{d\}$. Note that we *cannot* make this transformation once over the original CFG, G, since the incoming edges for χ depend on our choice of d. After the transformation, there will be at least one path in $\beta \to x$ for some $x \in X$ if and only if the expanded network has a path from $\beta \to \chi$. That is,

$$\bigcup_{x \in X \setminus \{d\}} (\beta \to x) \neq \emptyset \quad \text{if and only if} \quad \beta \to \chi \neq \emptyset$$

Thus, in the flow system for the variables η, constraints (B.3)–(B.4) are modified to be of the form

$$\begin{aligned} \eta_i^{\alpha\beta d} - \eta_j^{\alpha\beta d} &\geq 0 && \forall (\alpha, \beta, d) \in \mathcal{T}, \forall (i, j) \in E \mid i \neq d, j \neq d \\ \eta_\beta^{\alpha\beta d} - \eta_\chi^{\alpha\beta d} &\leq 1 - 2t_{\beta d} && \forall (\alpha, \beta, d) \in \mathcal{T} \\ \eta_x^{\alpha\beta d} - \eta_\chi^{\alpha\beta d} &\geq 0 && \forall (\alpha, \beta, d) \in \mathcal{T}, \forall x \in X \mid x \neq d \end{aligned}$$

The vertex χ is only relevant for these η constraints, and, thus, is not in V (for purposes of any other constraints).

In the end, our objective is to minimize cost

$$\sum_{i \in V} c_i z_i$$

subject to the top-level constraint

$$s_{\alpha d} + t_{\beta d} + u_{\alpha \beta d} + v_{\alpha \beta d} + w_{\alpha \beta d} \geq (1 - z_d) \;\; \forall (\alpha, \beta, d) \in \mathcal{T}$$

which, as stated earlier, asserts that one of the five subpaths forming an ambiguous triangle is $\emptyset$. As proven in Appendix A.1, this further implies that $S = \{i \in V \text{ such that } z_i = 1\}$ is a coverage set of D.

With all of the above in place, we put all constraints together, resulting in the full MILP shown in Fig. B.1. Note that the input is precisely that from Section 4.1, along with the precomputed set $\mathcal{T}$, and the precomputed Y set represented by numerical parameter a. All constraints are defined over all triples in $\mathcal{T}$. From the optimal model instance satisfying the constraints from Fig. B.1 (i.e., the instance that minimizes the cost function), we can then extract the optimal coverage set as $S = \{v \in I \text{ such that } z_v = 1\}$.

$$\min \sum_{i \in V} c_i z_i$$

subject to

$$\begin{aligned}
s_{\alpha d} + t_{\beta d} + u_{\alpha\beta d} + v_{\alpha\beta d} + w_{\alpha\beta d} &\geq 1 - z_d && \forall (\alpha, \beta, d) \in \mathcal{T} \\
\theta_i^{\alpha\beta d} - \theta_j^{\alpha\beta d} &\geq 0 && \forall (\alpha, \beta, d) \in \mathcal{T}, \\
& && \forall (i, j) \in E \mid i \neq d, j \neq d \\
\theta_e^{\alpha\beta d} - \theta_\alpha^{\alpha\beta d} &\leq 1 - 2s_{\alpha d} && \forall (\alpha, \beta, d) \in \mathcal{T} \\
\eta_i^{\alpha\beta d} - \eta_j^{\alpha\beta d} &\geq 0 && \forall (\alpha, \beta, d) \in \mathcal{T}, \\
& && \forall (i, j) \in E \mid i \neq d, j \neq d \\
\eta_\beta^{\alpha\beta d} - \eta_\chi^{\alpha\beta d} &\leq 1 - 2t_{\beta d} && \forall (\alpha, \beta, d) \in \mathcal{T} \\
\eta_x^{\alpha\beta d} - \eta_\chi^{\alpha\beta d} &\geq 0 && \forall (\alpha, \beta, d) \in \mathcal{T}, \\
& && \forall x \in X \mid x \neq d \\
\pi_i^{\alpha\beta d} - \pi_j^{\alpha\beta d} &\geq -(z_i - a_{\alpha\beta d i} z_i) - (z_j - a_{\alpha\beta d j} z_j) && \forall (\alpha, \beta, d) \in \mathcal{T}, \\
& && \forall (i, j) \in E \\
\pi_\alpha^{\alpha\beta d} - \pi_d^{\alpha\beta d} &\leq 1 - 2u_{\alpha\beta d} && \forall (\alpha, \beta, d) \in \mathcal{T} \\
\mu_i^{\alpha\beta d} - \mu_j^{\alpha\beta d} &\geq -(z_i - a_{\alpha\beta d i} z_i) - (z_j - a_{\alpha\beta d j} z_j) && \forall (\alpha, \beta, d) \in \mathcal{T}, \\
& && \forall (i, j) \in E \mid i \neq d, j \neq d \\
\mu_\alpha^{\alpha\beta d} - \mu_\beta^{\alpha\beta d} &\leq 1 - 2v_{\alpha\beta d} && \forall (\alpha, \beta, d) \in \mathcal{T} \\
\lambda_i^{\alpha\beta d} - \lambda_j^{\alpha\beta d} &\geq -(z_i - a_{\alpha\beta d i} z_i) - (z_j - a_{\alpha\beta d j} z_j) && \forall (\alpha, \beta, d) \in \mathcal{T}, \\
& && \forall (i, j) \in E \\
\lambda_d^{\alpha\beta d} - \lambda_\beta^{\alpha\beta d} &\leq 1 - 2w_{\alpha\beta d} && \forall (\alpha, \beta, d) \in \mathcal{T} \\
\\
z_i &\in \{0, 1\} && \forall i \in V \\
s_{\alpha d}, t_{\beta d}, u_{\alpha\beta d}, v_{\alpha\beta d}, w_{\alpha\beta d} &\in \{0, 1\} && \forall (\alpha, \beta, d) \in \mathcal{T} \\
\theta_i^{\alpha\beta d}, \pi_i^{\alpha\beta d}, \mu_i^{\alpha\beta d}, \lambda_i^{\alpha\beta d} &\in \mathbb{R} && \forall (\alpha, \beta, d) \in \mathcal{T}, \\
& && \forall i \in V \\
\eta_i^{\alpha\beta d} &\in \mathbb{R} && \forall (\alpha, \beta, d) \in \mathcal{T}, \\
& && \forall i \in V \cup \{\chi\}
\end{aligned}$$

Figure B.1: The complete MILP formulation

REFERENCES

[1] GDB: The GNU Project Debugger. `http://www.gnu.org/software/gdb`. Accessed: April 14, 2017.

[2] Agrawal, H., J.R. Horgan, S. London, and W.E. Wong. 1995. Fault localization using execution slices and dataflow tests. *Proceedings of the Sixth International Symposium on Software Reliability Engineering (ISSRE)*, 143–151.

[3] Agrawal, Hira. 1999. Efficient coverage testing using global dominator graphs. *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, 11–20. ACM.

[4] Agrawal, Hiralal. 1994. Dominators, super blocks, and program coverage. *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 25–34. ACM.

[5] Agrawal, Hiralal, Richard A. DeMillo, and Eugene H. Spafford. 1991. Dynamic slicing in the presence of unconstrained pointers. *Symposium on Testing, Analysis, and Verification*, 60–73.

[6] Agrawal, Hiralal, and Joseph R. Horgan. 1990. Dynamic program slicing. *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 246–256. ACM.

[7] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, techniques, and tools*. Addison-Wesley.

[8] Allauzen, Cyril, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A general and efficient weighted finite-state transducer library. *CIAA*.

[9] Alsallakh, Bilal, Peter Bodesinsky, Alexander Gruber, and Silvia Miksch. 2012. Visual tracing for the Eclipse Java debugger. *16th European Conference on Software Maintenance and Reengineering (CSMR)*, 545–548. IEEE.

[10] Alur, Rajeev, Ahmed Bouajjani, and Javier Esparza. 2015. Model checking procedural programs. *Handbook of Model Checking*. Springer.

[11] Alur, Rajeev, Thomas A. Henzinger, and Moshe Y. Vardi. 2015. Theory in practice for system design and verification. *SIGLOG News* 2(1):46–51.

[12] Alur, Rajeev, and P. Madhusudan. 2006. Adding nesting structure to words. *Proceedings of the 10th International Conference on Developments in Language Theory (DLT)*, vol. 4036 of *Lecture Notes in Computer Science*, 1–13. Springer.

[13] Ammons, Glenn, Thomas Ball, and James R. Larus. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 85–96. ACM.

[14] Anderson, Paul, Thomas W. Reps, and Tim Teitelbaum. 2003. Design and implementation of a fine-grained software inspection tool. *IEEE Trans. Software Eng.* 29(8): 721–733.

[15] Apiwattanapong, Taweesup, and Mary Jean Harrold. 2002. Selective path profiling. *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 35–42. ACM.

[16] Apple Inc. 2006. CrashReporter. Tech. Rep. TN2123.

[17] Artzi, Shay, Sunghun Kim, and Michael D. Ernst. 2008. Recrash: Making software failures reproducible by preserving object states. *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP)*, 542–565. Springer-Verlag.

[18] Arumuga Nainar, Piramanayagam, and Ben Liblit. 2010. Adaptive bug isolation. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 255–264. ACM.

[19] Ashok, B., Joseph M. Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. 2009. DebugAdvisor: a recommender system for debugging. *Proceedings of the 7th meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*, 373–382. ACM.

[20] Atlassian. Clover. `https://www.atlassian.com/software/clover`. Accessed: Jan 4, 2016.

[21] Ball, Thomas, and James R. Larus. 1994. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.* 16(4):1319–1360.

[22] ———. 1996. Efficient path profiling. *Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 46–57. IEEE.

[23] Bandyopadhyay, Aritra, and Sudipto Ghosh. 2012. Tester feedback driven fault localization. *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST)*, 41–50. IEEE.

[24] Bartolini, Cesare, Antonia Bertolino, Sebastian G. Elbaum, and Eda Marchetti. 2009. Whitening SOA testing. *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 161–170. ACM.

[25] Berris, Dean Michael, Alistair Veitch, Nevin Heintze, Eric Anderson, and Ning Wang. 2016. XRay: A function call tracing system. Tech. Rep., Google Inc.

[26] Bertolino, Antonia. 1993. Unconstrained edges and their application to branch analysis and testing of programs. *Journal of Systems and Software* 20(2):125–133.

[27] Beszédes, Árpád, Csaba Faragó, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. 2002. Union slices for program maintenance. *18th International Conference on Software Maintenance (ICSM)*, 12–21. IEEE.

[28] Binkley, David, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. 2006. A formalisation of the relationship between forms of program slicing. *Sci. Comput. Program.* 62(3):228–252.

[29] Binkley, David, Nicolas Gold, and Mark Harman. 2007. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.* 16(2):8.

[30] Bond, Michael D., and Kathryn S. McKinley. 2007. Probabilistic calling context. *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented Programming Systems and Applications (OOPSLA)*, 97–112. ACM.

[31] Bowring, Jim, Alessandro Orso, and Mary Jean Harrold. 2002. Monitoring deployed software using software tomography. *Proceedings of the ACM workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2–9. ACM.

[32] Britton, Tom, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible debugging software. *University of Cambridge-Judge Business School*.

[33] Bruns, Glenn, and Patrice Godefroid. 2000. Generalized model checking: Reasoning about partial state spaces. *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR)*, 168–182. Springer.

[34] Büchi, J Richard. 1960. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly* 6(1-6):66–92.

[35] Cao, Yu, Hongyu Zhang, and Sun Ding. 2014. SymCrash: selective recording for reproducing crashes. *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 791–802. ACM.

[36] Chen, Ning, and Sunghun Kim. 2015. STAR: stack trace based automatic crash reproduction via symbolic execution. *IEEE Trans. Software Eng.* 41(2):198–220.

[37] Cheung, Alvin, Armando Solar-Lezama, and Samuel Madden. 2011. Partial replay of long-running applications. *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) and the 13th European Software Engineering Conference (ESEC)*, 135–145. ACM.

[38] Chilakamarri, Kalyan-Ram, and Sebastian G. Elbaum. 2006. Leveraging disposable instrumentation to reduce coverage collection overhead. *Softw. Test., Verif. Reliab.* 16(4):267–288.

[39] Clause, James, and Alessandro Orso. 2007. A technique for enabling and supporting debugging of field failures. *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, 261–270. IEEE.

[40] Cordy, Brendan J., and Kai Salomaa. 2007. On the existence of regular approximations. *Theor. Comput. Sci.* 387(2):125–135.

[41] Cornejo, O., D. Briola, D. Micucci, and L. Mariani. 2017. In the field monitoring of interactive applications. *Proceedings of the 39th International Conference on Software Engineering (ICSE), NIER Track*.

[42] Crameri, Olivier, Ricardo Bianchini, and Willy Zwaenepoel. 2011. Striking a new balance between program instrumentation and debugging time. *Proceedings of the sixth conference on Computer systems (EuroSys)*, 199–214. ACM.

[43] Cui, Weidong, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. 2016. RETracer: triaging crashes by reverse execution from partial memory dumps. *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 820–831. ACM.

[44] Dallmeier, Valentin, Christian Lindig, and Andreas Zeller. 2005. Lightweight defect localization for Java. *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, 528–550. Springer.

[45] Dang, Yingnong, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. *34th International Conference on Software Engineering (ICSE)*. IEEE.

[46] D'Antoni, Loris. The symbolic automata library. `https://github.com/lorisdanto/symbolicautomata`. Accessed: April 15, 2016.

[47] D'Antoni, Loris, and Rajeev Alur. 2014. Symbolic visibly pushdown automata. *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, vol. 8559 of *Lecture Notes in Computer Science*, 209–225. Springer.

[48] D'Antoni, Loris, and Margus Veanes. 2016. Minimization of symbolic tree automata. *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 873–882. ACM.

[49] Dantzig, George. 1963. *Linear programming and extensions*. Princeton, NJ: Princeton University Press.

[50] De Lucia, Andrea, Mark Harman, Robert Hierons, and Jens Krinke. 2003. Unions of slices are not slices. *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE.

[51] D'Elia, Daniele Cono, and Camil Demetrescu. 2013. Ball-larus path profiling across multiple loop iterations. *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 373–390. ACM.

[52] Dhaliwal, Tejinder, Foutse Khomh, and Ying Zou. 2011. Classifying field crash reports for fixing bugs: A case study of mozilla firefox. *Proceedings of the 27th International Conference on Software Maintenance (ICSM)*, 333–342. IEEE.

[53] Dickinson, William, David Leon, and Andy Podgurski. 2001. Finding failures by cluster analysis of execution profiles. *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, 339–348. IEEE.

[54] DiGiuseppe, Nicholas, and James A. Jones. 2012. Concept-based failure clustering. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, 29:1–29:4. ACM.

[55] Do, Hyunsook, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10(4):405–435.

[56] D'Silva, Vijay, Daniel Kroening, and Georg Weissenbacher. 2008. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems* 27(7):1165–1178.

[57] Dunlap, George W., Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*. USENIX.

[58] Dwyer, Matthew B., Alex Kinneer, and Sebastian Elbaum. 2007. Adaptive online program analysis. *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, 220–229. IEEE.

[59] Eler, Marcelo Medeiros, Antonia Bertolino, and Paulo Cesar Masiero. 2011. More testable service compositions by test metadata. *Proceedings of the 6th International Symposium on Service Oriented System Engineering (SOSE)*, 204–213. IEEE.

[60] Feng, Yu, Xinyu Wang, Isil Dillig, and Calvin Lin. 2015. EXPLORER: query- and demand-driven exploration of interprocedural control flow properties. *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA)*, 520–534. ACM.

[61] Fischmeister, Sebastian, and Patrick Lam. 2010. Time-aware instrumentation of embedded software. *IEEE Transactions on Industrial Informatics* 6(4):652–663.

[62] Free Software Foundation. Gcov: a test coverage program. `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`. Accessed: Jan 3, 2016.

[63] Gabow, Harold N., Shachindra N. Maheswari, and Leon J. Osterweil. 1976. On two problems in the generation of program test paths. *IEEE Trans. Software Eng.* 2(3): 227–231.

[64] Gauf, Bernie, and Elfriede Dustin. 2007. The case for automated software testing. *Journal of Software Technology* 10(3):29–34.

[65] Glerum, Kirk, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (very) large: Ten years of implementation and experience. *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 103–116. ACM.

[66] Google Inc. Breakpad. `https://chromium.googlesource.com/breakpad/breakpad`. Accessed: Dec 2, 2013.

[67] Gupta, Rajiv, Mary Lou Soffa, and John Howard. 1997. Hybrid slicing: integrating dynamic information with static analysis. *ACM Trans. Softw. Eng. Methodol.* 6(4): 370–397.

[68] Hailpern, B., and P. Santhanam. 2002. Software debugging, testing, and verification. *IBM Syst. J.* 41(1):4–12.

[69] Hammacher, Clemens, Kevin Streit, Sebastian Hack, and Andreas Zeller. 2009. Profiling Java programs for parallelism. *Proceedings of the 2nd International Workshop on Multi-Core Software Engineering (IWMSE)*, 49–55.

[70] Hirzel, Martin, and Trishul Chilimbi. 2001. Bursty tracing: A framework for low-overhead temporal profiling. *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 117–126.

[71] Hofer, Joachim. 2010. eCobertura. `http://ecobertura.johoop.de/`.

[72] Horwitz, S., T. Reps, and D. Binkley. 1988. Interprocedural slicing using dependence graphs. *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 35–46. ACM.

[73] Horwitz, Susan, Ben Liblit, and Marina Polishchuk. 2010. Better debugging via output tracing and callstack-sensitive slicing. *IEEE Trans. Softw. Eng.* 36(1):7–19.

[74] IBM Rational. 2003. PureCoverage. `ftp://ftp.software.ibm.com/software/rational/docs/v2003/purecov/index.htm`.

[75] Jaramillo, Clara, Rajiv Gupta, and Mary Lou Soffa. 2000. FULLDOC: A full reporting debugger for optimized code. *Proceedings of the 7th International Symposium Static Analysis (SAS)*, vol. 1824 of *Lecture Notes in Computer Science*, 240–259. Springer.

[76] Jayaraman, Ganeshan, Venkatesh Prasad Ranganath, and John Hatcliff. 2005. Kaveri: Delivering the Indus Java program slicer to Eclipse. *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE)*, vol. 3442 of *Lecture Notes in Computer Science*, 269–272. Springer.

[77] Jimborean, Alexandra, Luis Mastrangelo, Vincent Loechner, and Philippe Clauss. 2012. VMAD: an advanced dynamic program analysis and instrumentation framework. *Proceedings of the 21st International Conference on Compiler Construction (CC)*, vol. 7210 of *Lecture Notes in Computer Science*, 220–239. Springer.

[78] Jin, Wei, and Alessandro Orso. 2012. BugRedux: reproducing field failures for in-house debugging. *Proceedings of the International Conference on Software Engineering (ICSE)*, 474–484. IEEE.

[79] ———. 2013. F3: fault localization for field failures. *International Symposium on Software Testing and Analysis (ISSTA)*, 213–223. ACM.

[80] Jones, Caper. 2008. Measuring defect potentials and defect removal efficiency. *CrossTalk: The Journal of Defense Software Engineering* 21(6):11–13.

[81] ———. 2016. Exceeding 99% in defect removal efficiency (DRE) for software. *Namcook Analytics, LLC*.

[82] Jones, James A., and Mary Jean Harrold. 2005. Empirical evaluation of the Tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 273–282. ACM.

[83] Kamkar, Mariam, Peter Fritzson, and Nahid Shahmehri. 1993. Three approaches to interprocedural dynamic slicing. *Microprocessing and Microprogramming* 38(1-5): 625–636.

[84] Karp, Richard M. 1972. Reducibility among combinatorial problems. *Proceedings of a symposium on the Complexity of Computer Computations*, 85–103. The IBM Research Symposia Series, Plenum Press, New York.

[85] Kasikci, Baris, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi. 2014. Efficient tracing of cold code via bias-free sampling. *USENIX Annual Technical Conference (ATC)*, 243–254. USENIX.

[86] Khoo, Yit Phang, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. 2008. Path projection for user-centered static analysis tools. *Proceedings of the 8th ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 57–63. ACM.

[87] Klíma, Ondrej, and Libor Polák. 2008. Hierarchies of piecewise testable languages. *Proceedings of the 12th International Conference on Developments in Language Theory (DLT)*, vol. 5257 of *Lecture Notes in Computer Science*, 479–490. Springer.

[88] Knuth, Donald E. 1968. *The art of computer programming, Volume I: Fundamental algorithms*. Addison-Wesley.

[89] Knuth, Donald E, and Francis R Stevenson. 1973. Optimal measurement points for program frequency counts. *BIT Numerical Mathematics* 13(3):313–322.

[90] Ko, Andrew J., and Brad A. Myers. 2010. Extracting and answering why and why not questions about Java program output. *ACM Trans. Softw. Eng. Methodol.* 20(2).

[91] Ko, Andrew Jensen, Thomas D. LaToza, and Margaret M. Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20(1):110–141.

[92] Ko, Andrew Jensen, and Brad A. Myers. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. *30th International Conference on Software Engineering (ICSE)*, 301–310. ACM.

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

[94] ———. 1990. Dynamic slicing of computer programs. *Journal of Systems and Software* 13(3):187–195.

[95] Kozen, Dexter. 1977. Lower bounds for natural proof systems. *18th Annual Symposium on Foundations of Computer Science*, 254–266. IEEE.

[96] Krinke, Jens. 2004. Context-sensitivity matters, but context does not. *4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, 29–35. IEEE.

[97] Lal, Akash, Junghee Lim, Marina Polishchuk, and Ben Liblit. 2005. BTrace: Path optimization for debugging. Tech. Rep. 1535, University of Wisconsin-Madison.

[98] Larus, James R. 1999. Whole program paths. *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 259–269. ACM.

[99] LaToza, Thomas D., and Brad A. Myers. 2010. Developers ask reachability questions. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 185–194. ACM.

[100] ———. 2011. Visualizing call graphs. *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 117–124. IEEE.

[101] Lattner, Chris, and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.

[102] Lei, Yan, Xiaoguang Mao, Ziying Dai, and Chengsong Wang. 2012. Effective statistical fault localization using program slices. *COMPSAC*, 1–10. IEEE.

[103] Lengauer, Thomas, and Robert Endre Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.* 1(1):121–141.

[104] Li, Bixin, Lulu Wang, Hareton Leung, and Fei Liu. 2012. Profiling all paths: A new profiling technique for both cyclic and acyclic paths. *Journal of Systems and Software* 85(7):1558–1576.

[105] Li, J. Jenny, David M. Weiss, and Howell Yee. 2008. An automatically-generated run-time instrumenter to reduce coverage testing overhead. *Proceedings of the 3rd International Workshop on Automation of Software Test (AST)*, 49–56. ACM.

[106] Liblit, Ben. 2014. The Cooperative Bug Isolation Project.

[107] Liblit, Ben, Alexander Aiken, Alice X. Zheng, and Michael I. Jordan. 2003. Bug isolation via remote program sampling. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[108] Lin, Yun, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-based debugging. *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. IEEE/ACM.

[109] Liu, Tongping, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: fast and precise error detection via evidence-based dynamic analysis. *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 911–922. ACM.

[110] Madsen, Magnus, Frank Tip, Esben Andreasen, Koushik Sen, and Anders Møller. 2016. Feedback-directed instrumentation for deployed javascript applications. *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 899–910. ACM.

[111] Maheshwari, Shachindra N. 1976. Traversal marker placement problems are NP-complete. Tech. Rep. CU-CS-092-76, University of Colorado, Boulder.

[112] Manevich, Roman, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. PSE: Explaining program failures via postmortem static analysis. *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 63–72. ACM.

[113] Marré, Martina, and Antonia Bertolino. 2003. Using spanning sets for coverage testing. *IEEE Trans. Software Eng.* 29(11):974–984.

[114] Mars, Jason, and Robert Hundt. 2009. Scenario based optimization: A framework for statically enabling online optimizations. *Proceedings of the 7th International Symposium on Code Generation and Optimization (CGO)*, 169–179. IEEE.

[115] Melski, David, and Thomas W. Reps. 1999. Interprocedural path profiling. *Proceedings of the 8th International Conference on Compiler Construction (CC)*, 47–62. Springer.

[116] Miranda, Breno, and Antonia Bertolino. 2014. Social coverage for customized test adequacy and selection criteria. *9th International Workshop on Automation of Software Test (AST)*, 22–28. ACM.

[117] Misurda, Jonathan, Bruce R. Childers, and Mary Lou Soffa. 2011. Jazz2: a flexible and extensible framework for structural testing in a Java VM. *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ)*, 81–90. ACM.

[118] Misurda, Jonathan, James A. Clause, Juliya L. Reed, Bruce R. Childers, and Mary Lou Soffa. 2005. Demand-driven structural testing with dynamic instrumentation. *27th International Conference on Software Engineering (ICSE)*, 156–165. ACM.

[119] Mohri, Mehryar, and Mark-Jan Nederhof. 2001. Regular approximation of context-free grammars through transformation. *Robustness in language and speech technology*, 153–163. Springer.

[120] Morrison, G. C., Cornelia P. Inggs, and W. C. Visser. 2012. Automated coverage calculation and test case generation. *South African Institute of Computer Scientists and Information Technologists Conference, (SAICSIT)*, 84–93. ACM.

[121] Mountainminds GmbH & Co. KG and Contributors. 2012. EclEmma: Java code coverage for Eclipse. `http://www.eclemma.org/`.

[122] Mozilla Foundation. Talkback. `http://talkback.mozilla.org`. Accessed: Nov 20, 2014.

[123] Muchnick, Steven S. 1997. *Advanced compiler design and implementation*. Morgan Kaufmann.

[124] Mulhern, Anne, and Ben Liblit. 2008. Effective slicing: A generalization of full and relevant slicing. Tech. Rep. 1639, University of Wisconsin–Madison.

[125] Myers, Del, and Margaret-Anne D. Storey. 2010. Using dynamic analysis to create trace-focused user interfaces for IDEs. *Proceedings of the 18th International Symposium on Foundations of Software Engineering (FSE)*, 367–368. ACM.

[126] Nagappan, Nachiappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. 2008. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering* 13(3).

[127] Nederhof, Mark-Jan. 2000. Practical experiments with regular approximation of context-free languages. *Computational Linguistics* 26(1):17–44.

[128] NetApp Inc. 2013. savecore. *Data ONTAP 8.2 Commands: Manual Page Reference for 7-Mode*, vol. 1, 533–534.

[129] Nishimatsu, Akira, Minoru Jihira, Shinji Kusumoto, and Katsuro Inoue. 1999. Call-mark slicing: an efficient and economical way of reducing slice. *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, 422–431. ACM.

[130] Ohmann, Peter, Alexander Brooks, Loris D'Antoni, and Ben Liblit. 2017. Control-flow recovery from partial failure reports. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[131] ———. 2017. Supporting proofs for control-flow recovery from partial failure reports. Tech. Rep. 1845, University of Wisconsin–Madison.

[132] Ohmann, Peter, David Bingham Brown, Ben Liblit, and Thomas W. Reps. 2015. Recovering execution data from incomplete observations. *Proceedings of the 13th International Workshop on Dynamic Analysis, (WODA)*, 19–24. ACM.

[133] Ohmann, Peter, David Bingham Brown, Naveen Neelakandan, Jeff Linderoth, and Ben Liblit. 2016. Encoding optimal customized coverage instrumentation. Tech. Rep. 1836, University of Wisconsin–Madison.

[134] ———. 2016. Optimizing customized program coverage. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 27–38. ACM.

[135] Ohmann, Peter, and Ben Liblit. 2013. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. *28th International Conference on Automated Software Engineering (ASE)*. IEEE/ACM.

[136] ———. 2015. CSIclipse: presenting crash analysis data to developers. *Proceedings of the 2015 Workshop on Eclipse Technology eXchange (ETX)*, 7–12. ACM.

[137] ———. 2016. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. *Automated Software Engineering* 1–40. Online first.

[138] Onoma, Akira K., Wei-Tek Tsai, Mustafa H. Poonawala, and Hiroshi Suganuma. 1998. Regression testing in an industrial environment. *Commun. ACM* 41(5):81–86.

[139] Orso, Alessandro, Donglin Liang, Mary Jean Harrold, and Richard Lipton. 2002. Gamma system: continuous evolution of software after deployment. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 65–69. ACM.

[140] Ottenstein, Karl J., and Linda M. Ottenstein. 1984. The program dependence graph in a software development environment. *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, 177–184. ACM.

[141] Pankumhang, Tosapon, and Matthew Rutherford. 2015. Iterative instrumentation for code coverage in time-sensitive systems. *8th International Conference on Software Testing, Verification and Validation (ICST)*, 1–10. IEEE.

[142] Parnin, Chris, and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers? *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 199–209. ACM.

[143] Pavlopoulou, Christina, and Michal Young. 1999. Residual test coverage monitoring. *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM.

[144] Pearson, Spencer, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. *Proceedings of the 39th International Conference on Software Engineering (ICSE)*.

[145] Perscheid, Michael, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2017. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* 25(1):83–110.

[146] Pin, Jean-Eric. 1997. Syntactic semigroups. *Handbook of formal languages*, 679–746. Springer.

[147] Place, Thomas, Lorijn van Rooijen, and Marc Zeitoun. 2013. Separating regular languages by piecewise testable and unambiguous languages. *38th International Symposium on the Mathematical Foundations of Computer Science (MFCS)*. Springer.

[148] Podgurski, Andy, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. 2003. Automated support for classifying software failure reports. *25th International Conference on Software Engineering (ICSE)*, 465–475. IEEE.

[149] Räihä, Kari-Jouko, and Esko Ukkonen. 1981. The shortest common supersequence problem over binary alphabet is NP-complete. *Theor. Comput. Sci.* 16:187–198.

[150] Renieris, Manos, and Steven P. Reiss. 2003. Fault localization with nearest neighbor queries. *International Conference on Automated Software Engineering (ASE)*. IEEE.

[151] Reps, Thomas W., Alexey Loginov, and Shmuel Sagiv. 2002. Semantic minimization of 3-valued propositional formulae. *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS)*. IEEE.

[152] Reps, Thomas W., Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.* 58(1-2):206–263.

[153] Rößler, Jeremias, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. 2013. Reconstructing core dumps. *Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

[154] Rothermel, Gregg, Sebastian Elbaum, Alex Kinneer, and Hyunsook Do. 2006. Software–artifact infrastructure repository. `http://sir.unl.edu/portal/`.

[155] Rountev, Atanas, Scott Kagan, and Michael Gibas. 2004. Static and dynamic analysis of call chains in Java. *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 1–11. ACM.

[156] Roy, Subhajit, and Y. N. Srikant. 2009. Profiling k-iteration paths: A generalization of the Ball–Larus profiling algorithm. *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 70–80. IEEE.

[157] Santelices, Raúl A., James A. Jones, Yanbing Yu, and Mary Jean Harrold. 2009. Lightweight fault-localization using multiple coverage types. *31st International Conference on Software Engineering (ICSE)*, 56–66. IEEE.

[158] Shannon, Claude E, and Warren Weaver. 1948. A mathematical theory of communication. *Bell system technical journal* 27(379-423):623–656.

[159] Simon, Imre. 1975. Piecewise testable events. *Automata Theory and Formal Languages, 2nd GI Conference*, vol. 33 of *Lecture Notes in Computer Science*, 214–222. Springer.

[160] Sjøberg, Dag I. K., Jo Erskine Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and Anette C. Rekdal. 2005. A survey of controlled experiments in software engineering. *IEEE Trans. Software Eng.* 31(9).

[161] Srinivasan, Sudarshan M., Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. 2004. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. *Proceedings of the USENIX Annual Technical Conference (ATC)*, 29–44. USENIX.

[162] Sumner, William N., Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. 2010. Precise calling context encoding. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 525–534. ACM.

[163] Takada, Tomonori, Fumiaki Ohata, and Katsuro Inoue. 2002. Dependence-cache slicing: A program slicing method using lightweight dynamic information. *Proceedings of the 10th International Workshop on Program Comprehension (IWPC)*. IEEE.

[164] Tallam, Sriraman, Xiangyu Zhang, and Rajiv Gupta. 2004. Extending path profiling across loop backedges and procedure boundaries. *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE.

[165] Tassey, Gregory. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project* 7007(011).

[166] Bullseye Testing Technology. BullseyeCoverage. `http://www.bullseye.com/productInfo.html`. Accessed: Jan 3, 2016.

[167] The Eclipse Foundation. The Eclipse integrated development environment. `https://eclipse.org/`. Accessed: May 12, 2017.

[168] Tice, Caroline Mae. 1999. Non-transparent debugging of optimized code. Ph.D. thesis, EECS Department, University of California, Berkeley.

[169] Tikir, Mustafa M., and Jeffrey K. Hollingsworth. 2002. Efficient instrumentation for code coverage testing. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 86–96. ACM.

[170] ———. 2005. Efficient online computation of statement coverage. *Journal of Systems and Software* 78(2):146–165.

[171] Tucek, Joseph, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. 2007. Triage: diagnosing production run failures at the user's site. *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 131–144. ACM.

[172] Van Fraassen, Bas C. 1966. Singular terms, truth-value gaps, and free logic. *The Journal of Philosophy* 63(17):481–495.

[173] Vaswani, Kapil, Aditya V. Nori, and Trishul M. Chilimbi. 2007. Preferential path profiling: compactly numbering interesting paths. *Proceedings of the 34th annual ACM Symposium on Principles of Programming Languages (POPL)*, 351–362. ACM.

[174] Veanes, Margus. 2013. Applications of symbolic finite automata. *18th International Conference on Implementation and Application of Automata (CIAA)*, 16–23. Springer.

[175] Veanes, Margus, Peli de Halleux, and Nikolai Tillmann. 2010. Rex: Symbolic regular expression explorer. *3rd International Conference on Software Testing, Verification and Validation (ICST)*, 498–507. IEEE.

[176] Venkatesh, G. A. 1991. The semantic approach to program slicing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 107–119. ACM.

[177] Wang, Xiaoyin, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. 2008. An approach to detecting duplicate bug reports using natural language and execution information. *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, 461–470. ACM.

[178] Weeratunge, Dasarath, Xiangyu Zhang, and Suresh Jagannathan. 2010. Analyzing multicore dumps to facilitate concurrency bug reproduction. *Proceedings of the 15th Edition of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 155–166. ACM.

[179] Weiser, Mark. 1982. Programmers use slices when debugging. *Commun. ACM* 25(7): 446–452.

[180] ———. 1984. Program slicing. *IEEE Trans. Software Eng.* 10(4):352–357.

[181] Williams, Laurie A., E. Michael Maximilien, and Mladen A. Vouk. 2003. Test-driven development as a defect-reduction practice. *14th International Symposium on Software Reliability Engineering (ISSRE)*, 34–48. IEEE.

[182] Wu, Rongxin, Xiao Xiao, Shing-Chi Cheung, Hongyu Zhang, and Charles Zhang. 2016. Casper: an efficient approach to call trace collection. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 678–690. ACM.

[183] Wu, Youfeng, and James R. Larus. 1994. Static branch frequency and program profile analysis. *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO)*, 1–11. ACM/IEEE.

[184] Xu, Baowen, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* 30(2):1–36.

[185] Xu, Xiaofeng, Yan Chen, W. Eric Wong, and Donghui Guo. 2009. VNM: A novel method to reduce the overhead of program instrumentation. *Proceedings of the WRI World Congress on Software Engineering (WCSE)*, 256–260. IEEE.

[186] Yang, Qian, J. Jenny Li, and David M. Weiss. 2009. A survey of coverage-based testing tools. *Comput. J.* 52(5):589–597.

[187] Yoo, Shin, and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Softw. Test., Verif. Reliab.* 22(2):67–120.

[188] Yorsh, Greta, Thomas W. Reps, Mooly Sagiv, and Reinhard Wilhelm. 2007. Logical characterizations of heap abstractions. *ACM Trans. Comput. Log.* 8(1):5.

[189] Yu, Yuan, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient detection of data race conditions via adaptive tracking. *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 221–234. ACM.

[190] Yuan, Ding, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: error diagnosis by connecting clues from run-time logs. *Proceedings of the 15th Edition of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 143–154. ACM.

[191] Yuan, Ding, Soyeon Park, Peng Huang, Yang Liu, Michael Mihn-Jong Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: Enhancing failure diagnosis with proactive logging. *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 293–306. USENIX Association.

[192] Yuan, Ding, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving software diagnosability via log enhancement. *Proceedings of the 16th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 3–14. ACM.

[193] Zamfir, Cristian, and George Candea. 2010. Execution synthesis: a technique for automated software debugging. *Proceedings of the 5th European conference on Computer systems (EuroSys)*, 321–334. ACM.

[194] Zhang, Xiangyu, Neelam Gupta, and Rajiv Gupta. 2006. Pruning dynamic slices with confidence. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 169–180. ACM.

[195] ———. 2007. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Softw. Eng.* 12(2):143–160.

[196] Zhang, Xiangyu, and Rajiv Gupta. 2004. Cost effective dynamic program slicing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 94–106. ACM.

[197] Zimmermann, Thomas, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss. 2010. What makes a good bug report? *IEEE Trans. Software Eng.* 36(5):618–643.