A NOVEL INVARIANTS-BASED APPROACH FOR AUTOMATED SOFTWARE FAULT LOCALIZATION

BY

SWARUP KUMAR SAHOO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Abstract

Software bugs are *everywhere*. Not only do they infest software during development, but they escape our extermination efforts and enter production code. In addition to severe frustration to customers, software failures result in billions of dollars of lost revenue to service providers. The most important steps for debugging and eliminating a software failure are reproducing the failure and finding its root cause, either during development time or during production run. Currently, debugging is a costly, time-consuming and manual process. Automating some of these steps will greatly help developers, reduce costs, increase productivity and software reliability. In this thesis proposal, I propose a novel way of doing automated software bug diagnosis.

Reproducing bug symptoms is a prerequisite for performing automatic bug diagnosis. Do bugs have characteristics that ease or hinder automatic bug diagnosis? As a first step, we conducted a thorough empirical study of several key characteristics of bugs that affect reproducibility at the production site [1]. We manually examined 266 randomly selected bug reports of six server applications and consider their implications on automatic bug diagnosis tools. Our results were promising. From the study, we found that nearly 82% of bug symptoms can be reproduced deterministically by re-running with the same set of inputs at the production site. We further found that very few input requests are needed to reproduce most failures; in fact, just one input request after session establishment suffices to reproduce the failure in nearly 77% of the cases. We describe the implications of the results on reproducing software failures and designing automated diagnosis tools for production runs.

We also propose an automatic diagnosis technique for isolating the root cause(s) of software failures after it is reproduced [2]. We use likely program invariants, constructed using automatically generated good inputs that are close to the fault-triggering input, to select a set of candidate program locations which are possible root causes. We then trim the set of candidate root causes using software-implemented dynamic backwards slicing, plus two new filtering heuristics: dependence filtering, and filtering via multiple failing inputs that are also close to original failing input. Experimental results on 13 reported software bugs of three large open-source servers MySQL, Squid, Apache web server and LLVM C/C++ clang compiler show that we are able to narrow down the number of candidate bug locations to between 5 and 28 locations for 12 out of 13 software bugs, even in programs that are hundreds of thousands of lines long. In our earlier work, we also showed invariants based techniques can be leveraged to effectively detect permanent hardware faults [3].

To My Parents.

Acknowledgments

In my desire for obtaining a Doctoral degree I was guided, encouraged, and supported by several persons. This thesis would not have been possible without their support and help.

Many thanks to my adviser, Dr. Vikram Adve, who spent his valuable time and effort to guide me through this complete research work and numerous paper publications which forms the core of my thesis work. He also carefully read my numerous revisions and suggested changes to improve this thesis. I have learned a lot from him, from algorithm development, program analysis to research methodology. He has helped me reach a level of maturity with his astonishing amount of acumen in research and a gentle, but firm, personality. I also thank him for making my experience at The University of Illinois, Urbana-Champaign highly satisfactory and fruitful. I hope his positive influence will help me in my journey towards becoming an independent researcher.

I am also grateful to my committee members Dr. Sarita Adve, Dr. Darko Marinov, Dr. Madhusudan Parthasarathy, Dr. John Regehr, and Dr. Benjamin Zorn, who offered invaluable suggestions, guidance and support. I am thankful to them for guiding me through my early draft of the dissertation, accommodating me at short notices. They have been a constant source of neat research ideas and have helped me to complete the thesis in a timely manner.

Thanks to the University of Illinois Graduate College for awarding me a Fellowship, providing me with the financial means to start my thesis research work. Also thanks to National Science Foundation for providing funding for my Reasearch Assistantship which helped and enabled me to complete this thesis. I also thank John Criswell who helped me developing some parts of software used in my thesis research work and many research publications. Also thanks to Chase Geigle who helped in conducting some experiments. I would also like to thank other members of our LLVM research group Robert Bocchino, Andrew Lenhart, Stephen Heumann, and Will Dietz for their help and valuable suggestions. And finally, I thank my wife, parents, brothers, sister and numerous other friends who endured this long process with me, always offering unflinching support and love, without which this work would not have been possible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In-field software failures are becoming increasingly common with the increasing complexity of software. In addition to causing severe inconvenience to customers, such software failures result in billions of dollars of lost revenue to service providers [4, 5, 6]. To this end, increasing the reliability of systems is becoming critically important.

Software development is a manual and complex process and is error prone. Hence, in spite of tremendous improvements in software engineering, testing and software reliability [7, 8, 9, 10, 11, 12, 13], not only do the software bugs infest software during development, but they escape our extermination efforts during testing and enter production code.

Debugging is the process of eliminating a software failure by reproducing the same failure, finding the faulty program statements which are root cause of the failure (referred to as bug diagnosis or fault localization) and then rectifying the faulty statements either during development time or during production run. Debugging is a very complex job for various reasons. Reproducing the same failure is very difficult, especially for longer running and multi-threaded applications. Locating the root causes is also extremely difficult as the point of failure symptom may be much later than when the faulty statements are executed. Production run failures are especially difficult to diagnose due to lack of enough information about failures.

Currently, debugging is mostly a manual process which is very costly and time-consuming. The use of automatic software bug diagnosis (also called fault localization) techniques, which automatically identifies the buggy lines in the program responsible for a given failure, is a promising solution for fixing bugs found during development time or production runs. Automatic bug diagnosis has the potential to speed up identification of root causes of failure (both during development and production runs), create reduced test cases for filing bug reports, and automatically repair the software while it is in production. Automating some of the debugging steps will greatly help developers, reduce costs, increase productivity and software reliability.

One challenge for automated diagnosis tools is that they generally require mechanisms to roll back and replay program inputs repeatedly for reproducing the failure in order to evaluate different root causes. Non-deterministic failures are also difficult to reproduce. Also, generating minimal test cases can make bug diagnosis process easier and more accurate. Techniques to simplify the failure reproduction and minimal test input generation processes will greatly help automated bug diagnosis.

In this thesis, we conducted an empirical study of software bugs of six widely used server "infrastructure" programs to identify the characteristics of server bugs which facilitate or hinder debugging activities like minimal test input generation and automated bug diagnosis. We also developed a novel framework to automatically diagnose root causes(s) of software bugs using likely program invariants. We experimentally evaluated our approach on 13 server and compiler bugs.

Next, we describe the motivation and goals of our empirical study of server software bugs along with the important findings and contributions. We then describe the motivation and goals of our automated diagnosis framework followed by the important contributions of this work.

1.2 Empirical Study of Server Bugs

Efficiently reproducing the same failures repeatedly and generating minimal test cases are difficult tasks which can simplify automated bug diagnosis. Server applications make these challenges even more difficult as they run for long periods of time, handle large amounts of data over that time, and perform concurrent processing of input. Do the failures in server applications lend themselves to being automatically reproduced? Are there characteristics of server applications that ease or frustrate attempts to perform automatic bug diagnosis? Are there characteristics of inputs that ease the procedure to find a minimal test case? We aim to answer these questions through an empirical study of real-world bugs in server applications.

We studied the above questions using public bug databases and committed bug fixes (patches) for six large open-source, server "infrastructure" programs that are widely used to build servers and Web server applications: Squid, Apache, Subversion (SVN), MySQL, the OpenSSH Secure Shell Server (`sshd`), and the Tomcat Application Server. Among the several servers we considered for this study, we attempted to include a mix of stateless and stateful servers. We randomly selected and manually analyzed a total of 266 randomly selected bugs and 30 specifically selected concurrency bugs in these six programs to answer these questions[1].

1.2.1 Findings and Contributions

The main findings of our study are as follows. Here, we define an "input" as a logical request from the client to the server that needs to be buffered to be able to reproduce a failure (see Section 3.3):

1. Nearly 77% of the failures due to software bugs can be reproduced with just one input request (excluding session establishment inputs like *login* and *select database* requests).

[1] A detailed spreadsheet with all these bugs can be found at http://sva.cs.illinois.edu/ICSE2010/bug_statistics.xls

Among the remaining bugs with clear information, only 12 out of the 30 multiple-input failures need more than 3 inputs to trigger a symptom.

2. Among the 30 cases that require more than one input request to trigger a fault, in a majority of them (17 cases), all the necessary inputs are likely to occur within a short duration of time.

3. For most of the failures, the time duration between the first faulty input and the symptom is small.

4. For the majority of overall bugs (nearly 63%), the failure symptom is an incorrect output. In two of the applications (Squid and Tomcat), we find many fewer incorrect output symptoms compared to other applications; our analysis leads us to believe that this is because Squid and Tomcat use many more assertions and exception handlers, respectively.

5. Nearly 82% of all the analyzed faults were reported to have deterministic behavior (i.e. the fault triggers the same symptom each time the application is run with the same set of input requests in the same order, on a fixed architecture/OS platform and a configuration). This holds for stateful applications as well as for extensively multi-threaded applications like Apache, MySQL, and Tomcat.

6. Concurrency bugs (e.g., data races or deadlocks) form a small fraction (<2%) of all bugs (even in highly concurrent applications), but they have very different, more complex characteristics: nearly all are non-deterministic, they usually require more inputs to trigger, they have more hangs/crashes and fewer incorrect output symptoms, but about 17% of them show different failure symptoms in different executions for the same inputs.

Our results have several implications for automated bug diagnosis tools:

1. Most bug failures can be automatically reproduced by replaying just the last input. Furthermore, most failures, including both single and multi-input failures, are triggered within a short time after receiving the first faulty input. In cases where multiple inputs are needed to trigger a fault, the required inputs are also likely to be clustered together in the input stream. Therefore, tools can optimize their search for inputs that reproduce the failure by first trying a small number of the most recent inputs received before the failure was detected. Systems can buffer only a small suffix of the inputs for each server instance.

2. New techniques are needed to detect the internal data corruptions that cause incorrect outputs as they are the most common symptom. Inserting more assertions or exceptions manually or automatically appears to be a promising approach.

3. Isolating fault-triggering inputs for concurrency bugs and reproducing failures of such bugs reliably is significantly harder than for other bugs, but tools can exploit the fact that there are fewer incorrect output errors and many more hangs/crashes (i.e., failure detection is somewhat easier).

One perhaps surprising implication of the present study is that a simple restart/replay approach is likely to work for most bugs in real-world server programs, as described in Section 3.7. Based on these results, we also propose a new low-cost technique to reproduce failures during production and development. Automated bug diagnosis tools can buffer a suffix of the input requests to the server and upon failure, *restart the server and replay the suffix of inputs* to attempt to reproduce the failure. A few preliminary experiments suggest this approach may be able to reproduce server failures without checkpointing the server state. Checkpointing-based rollback and replay is not practical as lightweight checkpointing requires operating system support [7] and multithreaded processes also complicate the checkpointing procedure. Simple restart/replay approach may be a more feasible approach than checkpointing.

1.3 Automated Diagnosis of Software Bugs

After reproducing a failure, the next step for eliminating a software failure is to find its root cause. Given a program and a "failing input," i.e., an input for which the program encounters a failure, *bug diagnosis* or *fault localization* is the problem of identifying one or more possible locations in a program that must be changed to prevent the failure in future executions. This is a very important, costly and time-consuming manual step in debugging software failures. Automating this *fault localization* process can significantly increase programmer productivity and reduce software development costs.

There has been much previous work on this problem [10, 11, 12, 14, 15, 16, 17, 18, 13, 19, 20], but the overall problem is far from solved, and, to our knowledge, *no such tools are used in real world development.* Simple statistical techniques such as Tarantula [17, 18] turn out to be useful only in relatively few bugs, as our results corroborate (see Section 6.2.4). Some tools [21] are slow and may not be suitable for large applications. Other tools [15] may report too many source lines as potential root causes of the bug (i.e., report too many false positives). Some tools [22, 23, 20] often fail to report source lines containing the root cause (i.e., often have false negatives). Some tools [16] also may be impractical due to special hardware requirements. Others use systematic but unscalable techniques like comparing the memory states of passing and failing runs [20] or SMT solver-based approaches [24]. In fact, for many of these approaches, it is not clear how they will scale to large, realistic programs. A recent study [25] has shown that existing fault localization approaches, in practice, do not help programmers much. Currently, there is no practical and usable technique that can pinpoint the root cause of failures in large programs realistically and efficiently without generating a large number of false positives. The goal of our work is to address the drawbacks of the existing approaches and improve the state of the art in fault localization.

Our work on fault localization extends previous work that uses invariant-based diagnosis to narrow down the root causes of bugs. Informally, *likely program invariants* are program properties that are observed to hold in some set of successful executions but may not necessarily hold for all executions [26]. Invariant-based approaches first extract likely invariants in some manner, and then run the program on an input that triggers the bug. Any invariants that are violated in this failing execution are assumed to be potential root causes [15]. Invariants can be very useful in identifying candidate root cause(s) as they provide an efficient way to compare passing and failing runs.

We propose a sophisticated approach combining likely program invariants with novel filtering techniques to narrow down a possible set of locations that have to be changed to eliminate a failure ("*candidate root causes*"). Current invariant-based approaches [15] often report too many false positives (i.e. too many extra source lines as potential root causes). An earlier work by Pytlik et.al. [27] used invariants for fault localization of Siemens benchmark suite without much success possibly due to the fact that they used general test inputs to generate invariants. We extend the previous invariants-based work in two ways. First, instead of generating invariants from many randomly selected inputs, we use systematic algorithms based on delta debugging (ddmin) [21] to automatically construct a small set of inputs which are very similar to the failure-triggering input but that do not trigger the fault and train invariants using these inputs. This process yields stricter invariants that are better able to find root causes (though this can result in more false positives). Second, we employ several techniques to filter out the false positive invariants as tighter invariants can yield more false positives. We use dynamic backward slicing to filter out failed invariants that do not influence the observed failure symptom. Additionally, we have developed two novel heuristics to further narrow down the root causes of failure. First heuristics called *dependence filtering* removes a failed invariant from the candidate root causes, if it is dependent upon another failed invariant without any intervening passing invariant. Second, we can focus on invariants that fail for each of the inputs from a small set of failure-inducing inputs which are similar to the original input.

We evaluated our approach using three large open-source, server "infrastructure" applications: Squid, Apache, MySQL and LLVM C/C++ clang compiler. We used eight previously reported bugs from server applications and five clang compiler bugs to evaluate our approach. To the best of our knowledge, our evaluation uses larger applications than any previous approaches except one [12]. We used an automated procedure based on *ddmin* to construct the passing and failing inputs that are "close" to the failing input for Squid and MySQL. For clang compiler bugs, we used C-Reduce [28], which is a tool to minimize an input which produces a given software failure. to automatically generate similar passing and failing inputs. For Apache, we manually constructed passing and failing inputs using a systematic algorithm.

1.3.1 Findings and Contributions

To summarize, our main contributions are:

1. We present a novel invariants-based approach for fault localization, based on training the invariants with a small set of automatically constructed "good inputs" that are close to a given failing input. Our results show that this approach yields a more precise set of root causes than previous work. Moreover, we are the first to perform this combination in software without special-purpose hardware support.

2. We present novel heuristics for reducing false positives in the diagnosis results. These heuristics have valuable synergy with the core invariant strategy because they leverage the other analysis techniques, including input construction and dynamic dataflow and dependence analysis.

3. We evaluate our approach for real bugs in a larger and more realistic set of applications than all previous work except one [12].

4. Our results show that the approach is extremely effective at reducing the set of possible root causes, even in large programs, and that each of the filtering steps (except

multiple-input filtering) makes an important contribution in reducing this set. For the eight server bugs, the slicing step removes nearly 75% of the false candidates, dependence filtering removes an additional 60% of the remaining ones on average, and using multiple faulty inputs removes a few more.

5. Overall, we reduce the number of candidate root causes to a range of only 5–17 program expressions per bug, even in programs of hundreds of thousands of lines of code.

6. As compared to Tarantula [17] and Ochiai [18] approaches, which have been used for similar comparisons in previous work, we find that their approach is slightly better in the two best cases, but is extremely inconsistent, with thousands of candidate locations in 6 out of 8 bugs.

7. Additionally, we evaluated our approach on few compiler bugs since they tend to have different characteristics and generally more complex than bugs of other applications. For the five clang bugs, slicing step removes nearly 80% of the false candidates and dependence filtering removes nearly 50% of the remaining ones on average. Overall, we reduce the number of candidate root causes to a range of only 17–28 per bug except 1 bug where we failed to find the root cause.

Invariants can be leveraged for solving a wide variety of problems. In our earlier work, we used invariants based technique to successfully detect permanent hardware faults [3]. Hence, implementation costs of our diagnosis tool can be partially shared by leveraging invariants for solving other related problems.

In future, we plan to work on additional algorithms and filtering techniques like SMT solver and value-replacement based approaches to reduce the candidate set of root causes further. Our current implementation can not effectively handle missing code bugs like missing initialization. We also plan to explore other varieties of invariants at more program points and invariants on more types of program values so that we can diagnose more variety of bugs including concurrency bugs. We also plan to develop our automated input generation algorithms further and make them more robust. We also plan to extend our work to automatically isolate root cause(s) for failures during production runs. Finally, we would like evaluate with more applications and bugs.

The rest of the document is organized as follows. Chapter 2 presents the earlier work related to our bug diagnosis framework. Chapter 3 describes the details of our empirical study of server software bugs along with the results of the study. Chapter 4 gives the overview of our developed automated bug diagnosis framework and further describes how initial candidate set of root cause(s) are selected using automatically generated inputs and invariants. Chapter 5 explains the details of various filtering techniques we use like dynamic backward slicing, dependence filtering and multiple-input filtering. Chapter 6 presents our experimental evaluation on 13 server and compiler bugs. Chapter 7 describes the future work we plan to pursue. Finally, Chapter 8 describes the conclusion of our work.

Chapter 2

Related Work

There have been many previous approaches for doing automated software bug diagnosis or fault localization. These approaches can be broadly classified into following categories. We will describe and contrast with some specific work in more detail in later chapters.

Delta debugging [13, 19] is a general technique for fault localization which compares a successful and a failing run to find root causes. The original technique computed the cause-effect chains (causal paths of the failure) by comparing the program states of failing and successful run at each step and determining a minimal subset of program state which needs to be copied to failing run for the failure symptom to go away. The original technique is semi-automated and does not scale to larger programs due to inaccurate mapping of program states. Recent work on program execution indexing and memory indexing [29, 20] has greatly improved delta debugging's diagnostic accuracy. Program execution indexing makes it easier to match dynamic program statements and similarly, memory indexing simplifies the comparison of matching memory locations in two runs. However, these techniques computes causal paths in stead of root causes and previous work [22, 23] shows 45% of the computed causal paths miss root cause. Previous work [22, 23] also states that if a different input is used for a failing run, it can give inaccurate causal paths. Also, when copying state from a correct run to a faulty run, it may be necessary to copy more than the faulty state, thus resulting in false positives. It is also not clear how this approach will scale for larger programs and larger program traces. Triage [12] also used differences in control flow between a successful and a failing run and combined it other techniques to further improve its accuracy.

Statistical techniques [30, 17] are another way to perform fault localization which generally requires a few failing and successful runs. In this technique, correlations between some particular software properties and failure symptom are found by using a set of correct and failed executions. Liblit et.al. [11] developed efficient techniques to use these kind of correlations with much lower overhead. They used sampling to spread the overhead of their instrumentation code across various executions. Tarantula and others [17] exploit differences in control flow between successful and failing runs and measure the correlation between execution of statements and failure symptom. They rank the statements based on a suspicious criteria which gives the likelihood of the corresponding statement being faulty. Jones and Harold [18] experimented with the tarantula approach for Siemens benchmark suite to rank the root causes according to different suspiciousness criteria and found the Ochiai ranking criteria to be the best. Artzi et.al. [31, 32] improved Tarantula approach in two ways for fault localization of PHP applications. First, they automatically generate test cases using a form of concolic testing. Second, they extend the base approach with output mapping and condition modelling. Baudry [33] et al. propose a new technique to select test cases which can maximize diagnostic accuracy for statistical techniques. Renieris et.al. [34] uses a faulty run and large number of successful runs for performing fault localization. It selects the correct run that most resembles the faulty run according to a distance metric criteria and compares the program spectra between these two runs. In particular, they use set spectra with union/intersection model and distance spectra with nearest neighbor model. These statistical techniques are likely to work well when there is control flow difference between successful and failing runs and may not work well in other cases reporting many false positives.

Invariants-based diagnosis frameworks use invariants as a primary technique to detect anomalous program behavior and localize faults. DIDUCE [15] was one of the first work to use invariants for diagnosis. It used a variant of range invariant to find root causes. They refine the invariants as the application runs and report any violations during the same program run. They also rank the statements based on some ranking metric and present them to the users for finding root causes of failures. Later, Harold [35] used range invariants for fault localization of Siemens benchmark programs. Dimitrov et.al. [16] also used three type of invariants for fault localization and augmented it with few other techniques to increase its accuracy as explained later.

Dynamic slicing has also been used for bug diagnosis; Gupta et.al. [36, 37] first used dynamic backward slicing for fault localization and later they also combined both forward and backward slicing in failure inducing chop and multiple points slicing [38, 39] to further increase the precision of diagnosis. Triage [12] combined differences in control flow between successful and failing run with dynamic backward slicing for automated diagnosis. Dimitrov et.al. [16] combined invariants based diagnosis with dynamic forward slicing. They also use an additional heuristic by suppressing the execution of candidate root causes and observing the output of the program to further reduce false positives. However, they require custom hardware support for implementing their technique.

Value replacement or execution perturbation technique [40, 41, 42] was first used by Gupta et.al. for fault localization. They replaced the value of program statements with few values observed during successful runs and then decide which statements might be root cause depending upon the program outcome. Later they extended their technique [43] to find out all root causes when multiple faults are present. They also used some optimizations and parallelization to improve the efficiency of their technique. For larger programs, there will be a very large number of possible combinations of program statements and their all possible alternate values to explore.

Formal methods have also been used for automated bug diagnosis. The *explain* tool [44] by Groce et.al. uses Bounded Model checking tool CBMC and SAT solver to find a counterexample. It then uses a PBS solver to find a successful run that is as similar as possible to the counter example. The differences between the two runs are computed and presented to the user after applying a causal slicing step. BugAssist [24] formulates the program as a boolean trace formula in conjunctive normal form such that every satisfiable assignment of the formula corresponds to one particular execution of the program. It then extends the formula by adding constraints on input values and program post conditions. It uses a partial MAX-SAT solver to determine the smallest set of clauses which can not be satisfied and reports the corresponding program statement as possible root causes. For some larger trace, they also use trace reduction techniques like dynamic slicing. However, we do not believe that these techniques alone will scale to larger programs.

Chapter 3

Empirical Study of Server Bugs

The first step for doing automatic software bug diagnosis is reproducing the same failure. One challenge for automated diagnosis tools is that they generally require mechanisms to roll back and replay program inputs repeatedly for reproducing the failure in order to evaluate different root causes. Unfortunately, using checkpointing for such roll back limits the practical usefulness of such tools. First, checkpointing that is lightweight enough to be used for software diagnosis or debugging requires operating system support [7], which means that such tools cannot work on commodity operating systems that lack such support. Multi-threaded processes also complicate the checkpointing procedure. In addition, diagnosis cannot be done if the fault happens before the checkpoint is taken. It would be very useful if diagnosis tools could simply *restart* the target program and replay a small subset of the inputs to reproduce and diagnose the failures. Exploring this question requires an understanding of real-world bug behavior.

There also exist techniques like Delta Debugging [21] to generate a minimal set of test inputs from some failing test case. In general, such tools are too slow to be deployed in a production environment. Again, the knowledge of what types of inputs and how many inputs are commonly needed to produce failures can help us to build automated tools which use heuristics to efficiently select a minimal test case. However, in order to be widely deployable in production run environments, bug diagnosis tools must be able to quickly isolate the inputs that trigger a fault, reduce the inputs to just those that trigger the fault, and be able to reproduce the fault with reasonable reliability.

We performed an extensive empirical study of characteristics of several server software bugs which have important implications upon automated diagnosis tools. In particular, we wanted to study the characteristics of bugs which can simplify difficult tasks of automated debugging tools like reproducing failures and generating minimal test cases. This chapter presents the details of our empirical study of software bugs for six server applications.

3.1 Goals of The Study

The main goal of this study is to answer and analyze the following questions, focusing on server software:

1. How many inputs are needed to trigger the symptoms of a software bug?
2. How long does it take for the symptom to occur after the first faulty input is used by the application?
3. What kinds of symptoms appear as a manifestation of bugs? Are these symptoms sufficient for creating automatic bug diagnosis tools?
4. What fraction of failures are deterministic with respect to the inputs?

The answers to these questions will have implications for designing automated diagnosis tools for server applications. Server applications have several qualities that make them ideal for such a study. They are widely used and mission critical for many businesses. They process a large amount of input data over extended periods of time, making it important to understand how many server inputs are needed to trigger failures, and how reliably they do so. They are also extremely concurrent, making it important to understand whether bug behavior (both normal errors and concurrent programming errors) is deterministic with respect to the inputs. While these qualities make them challenging, a silver lining is that their inputs are well-structured due to the nature of the protocols they use to communicate with clients.

Application	**Description**	**#LOC**	**Years in Production**	**#total bugs after sampling**	**#bugs Selected**
Squid 3.0.x	caching web proxy	93K	5	170	40
Apache 2.0.x	web server	283K	5.7	65	52
OpenSSH sshd 3.6-3.x, 4.x	secure shell server	27K	5.25	61	54
SVN 1.0.0 - 1.6.0	version control server	587K	5	16	12
MySQL 4.x	database server	1,028K	5.7	90	55
Tomcat 5	servlet container and web server	274K	5	70	53
Total		**2,292K**		**472**	**266**

Table 3.1: **Set of Server Applications and Software Bugs.**

3.2 Methodology

In this section, we describe our methodology for selecting the applications and the bugs we study; we also discuss the limitations of our study. A summary of the applications and software bugs is in Table 3.1.

3.2.1 Application Selection

Our study focuses on widely used, large server applications. We aimed our study at open-source applications that provide both a publicly-accessible bug database and access to the server's source code. When possible, we opted to study servers implementing stateful protocols (i.e., protocols requiring the server to maintain application-specific protocol state during an application-defined session) as these servers are the most likely to require more inputs for failure reproduction. Finally, we needed applications with a sufficient number of production bugs to study.

We considered many widely used Linux servers e.g., those implementing the IMAP, SMTP, FTP, DNS, LDAP, and NIS protocols, and selected six server applications. We found that only a few maintain significant state, and we deliberately included the stateful servers in our study. Three out of six servers in our study (Squid, MySQL, Tomcat), or the applications run on top of Tomcat, maintain significant state that affects reported server bugs. All six servers use TCP as the transport-level protocol for client communication.

Apache web server: The Apache web server communicates with clients via the stateless

HTTP protocol [45] and via HTTP over SSL for authenticated and encrypted communication. While HTTP is stateless, Apache does maintain some information on the threads and processes it uses to handle requests. Apache can also maintain in-memory and on-disk state for caching recently served web pages.

Squid HTTP proxy server: The Squid web proxy server also communicates with clients via HTTP and HTTP over SSL. Like Apache, Squid can cache recently accessed web pages in memory and on disk. Squid's rate throttling features [46] must also maintain some global state.

MySQL database server: MySQL uses a custom protocol for client communication [47]. For each session, the server must maintain state about the authentication status and credentials of the client, temporary tables, prepared statements, and various parameters like query cache size and SQL modes. MySQL also supports clustering and replication, which can maintain a lot of global state.

Tomcat Servlet Container: Tomcat uses the HTTP protocol (optionally over SSL) for client communication. It maintains internal state about which web applications are loaded and provides facilities with which web applications can maintain session state across individual HTTP requests [48]. Tomcat bug reports are often generated by developers of applications running on Tomcat. Such applications can maintain arbitrary amounts of state, e.g., for e-commerce. This state will affect the behavior and reproducibility of failures due to Tomcat bugs. Tomcat also supports clustering and session replication, which maintains a lot of global state.

OpenSSH secure shell server: The OpenSSH server communicates via the stateful SSH protocol [49]. This protocol provides sessions which maintain one or more virtual channels over which programs on the client can communicate with programs on the server [49]. For each client connection, the server must maintain a small amount of state information about each SSH session as well as the state of each virtual channel maintained by the session.

Subversion version control server: SVN can use HTTP or a custom, stateful SVN protocol [50] tunneled through SSH. For this study, we restricted ourselves to the stand-alone server using the SVN protocol. The SVN server maintains a small amount of per-session state [50].

3.2.2 Bug Selection

We selected bugs to analyze for each server as follows.

First, for each application, we selected a recent major version of the software that had been in development and production use for at least a year. We expect these versions to have a more diverse bug sample. In a few cases, a single version of the software did not provide a sufficiently sized sample, so we used multiple versions of the software.

Second, we then selected the set of repaired bugs by using filters provided by the program's Bugzilla database. For all programs except MySQL, we searched for bug reports with a status field of either RESOLVED, VERIFIED, or CLOSED, a resolution field of FIXED, and whose severity was anything other than TRIVIAL or ENHANCEMENT. Since the filters for MySQL are non-standard, we had to adapt our selection criteria to search for bug reports marked FIXED or PATCH APPROVED/QUEUED and with any severity other than FEATURE REQUEST. Fixed bugs will have the most complete and accurate information, but may be biased towards easier-to-fix bugs (see section 3.2.3).

Third, we randomly selected a subset of bugs from the bug list generated in the previous step for each server using a seeded `rand()` function. We skipped this step for `sshd` and SVN since they had fewer bugs. The detailed information about the number of random bugs selected after this step is shown in the fifth column of Table 3.1.

Finally, we removed the bugs in development versions of code from the randomly selected bug list. Since our main goal was in-production bug diagnosis, we only focused on bugs in the externally released versions of code. Squid had many development bugs, so we had to select a much larger random sample compared to other servers. Then, we manually removed trivial bugs like build errors, documentation errors, and feature requests. After this final manual filtering, we were left with 266 bugs from 472 bugs in the earlier step. The last column of Table 3.1 shows the details for each server.

After creating the list of bugs, we analyzed each bug report and its test cases, available patches, and any other external information associated with it. We classified each bug's characteristics by first examining the information in the bug report and then by inspecting the patches and other external information if more information was needed to make a decision. We also used available patches to confirm our analysis of the bug reports. Based on the analysis, we classified each bug based upon the observable failure symptoms the bug caused, the reproducibility of the observable symptoms, and the maximal number of inputs needed to trigger the observable symptoms. For some bugs, we could not classify the bug for some characteristics due to limited information in the bug report. We report, for each characteristic, the number of such "unclassifiable" bugs by placing them in a separate classification as described in Section 3.4.

3.2.3 Limitations

Our methodology's limitations should be considered when using or interpreting our results. Like other empirical studies, our results are limited by the kinds of applications and software bugs we used.

Our study examines a subset of server applications. Our results may not apply to other types of software or to network servers with different architectures e.g., peer-to-peer software. Also, all the servers except one are written in C and/or C++ (Tomcat is written in Java); results may differ for similar servers written in other programming languages.

There is also some potential bug selection bias in our study. First, some samples omit bugs that cause security flaws because their projects e.g., Apache [51] and `sshd` [52], do not record such bugs in their public bug databases. Second, our study omits unfixed bugs from the samples as their bug reports may contain inaccurate information. Since it is possible that unfixed bugs have different properties, it has the potential to introduce bias. Third, the bug samples in our study exclude unreported bugs. We believe this is acceptable because unreported bugs (in a mature server) are likely to be much less frequent than reported bugs. Nevertheless, it is possible that bugs whose failures are difficult to reproduce (which may include non-deterministic bugs and multi-input bugs) are less likely to be reported. Our results are, therefore, only representative of *reported bugs*. On the positive side, our study does not exhibit the bug feature and commit feature biases as described by Bird et. al. [53] as we sampled *all* fixed bugs in the bug database instead of sampling the subset of fixed bugs which are linked with the source code bug fixes or other bug fix information.

Finally, our study relies upon the accuracy and completeness of the bug reports and the materials connected to them. Also, as we have analyzed a large number of bug reports manually, there may be some human error in the results.

3.3 Definitions and Terminology

Here we define several terms as they are used in the context of our work. First, an *input request* to a server is a logical input from the client to the server (we identify logical inputs at the application level as reported in the bug reports); examples include a login/authentication request for establishing a session in SSH/MySQL, any command from an SSH or SVN client, an HTTP request, or a MySQL query. Each request may internally involve more than one communication between the client and server and may span several network packets.

We chose logical inputs because bug reports describe test cases as a set of application-level inputs. Translating inputs from bug reports into low-level network protocol messages would have made the study too tedious and error-prone.

Messages coming from sources other than the client e.g., the file system, back-end databases, DNS queries, are not considered to be input requests. These inputs are responses to requests that the server made on behalf of a client i.e., client input is driving these other inputs. Since there is a causal relationship between client inputs and these other inputs,

Application	**Memory Error Crash**	**Other Crash**	**Assertion violation**	**Hang**	**Incorrect Output**	**Unclear**
Squid	6 (15.00%)	5 (12.50%)	11 (27.50%)	0 (0.00%)	18 (45.00%)	0 (0.00%)
Apache	6 (11.54%)	2 (3.85%)	1 (1.92%)	2 (3.85%)	38 (73.08%)	3 (5.77%)
sshd	0 (0.00%)	5 (9.26%)	0 (0.00%)	3 (5.56%)	44 (81.48%)	2 (3.70%)
SVN	1 (8.33%)	2 (16.67%)	1 (8.33%)	1 (8.33%)	7 (58.33%)	0(0.00%)
MySQL	7 (12.73%)	8 (14.55%)	0 (0.00%)	3 (5.45%)	35 (63.64%)	2(3.64%)
Tomcat	10 (18.87%)	0 (0.00%)	12 (22.64%)	3 (5.66%)	25 (47.17%)	3 (5.66%)
Total	30 (11.28%)	22 (8.27%)	25 (9.40%)	12 (4.51%)	167 (62.78%)	10(3.76%)

Table 3.2: **Classification of Bug Symptoms.**

replaying the client inputs will cause the server to perform the same requests.

We also exclude inputs that do not trigger the fault per se but are used, instead, to recreate a persistent environment in which the failure can be reproduced. Examples of such inputs are SVN checkout and SQL table creation commands (unless they are part of the faulty input set). In production systems, the environment is almost always present before the fault is triggered; these inputs are only in the bug report so that a similar environment can be manufactured at the developer's site for off-line diagnosis.

We define *symptoms* of bugs as any incorrect program behavior which is externally visible or detectable. For example, an incorrect output, segmentation fault, program crash, program hang, or assertion violation is a symptom. We use the term *incorrect output* to describe a symptom in which the server completes an operation but the external output of the program differs from the correct behavior. In our analysis, we say that an incorrect output symptom is detected when incorrect output is externally visible to the user or client. Though detecting incorrect outputs can be challenging, the internal data corruption that causes them can be detected through manually or automatically inserted assertions [26, 54, 55]. Detecting bugs is not a subject of this study; we simply treat incorrect outputs as a symptom.

A failure due to a software bug is observed to be *deterministic* if the fault triggers the same symptom each time the application is run with the same set of input requests in the same order, on a fixed architecture/OS platform and a fixed server configuration. Otherwise, the failure due to a bug is *non-deterministic*. The architecture, OS platform and server configuration are the user-controllable parts of the environment. This definition is reasonable because our goal is reproducibility at the end-user's site, where these parts of the environment and software configuration are fixed. A failure due to a bug is *timing dependent*, if the timing of the input requests in addition to their order determines whether a symptom is triggered and if so, which symptom. This is a special case of a non-deterministic software bug that is input timing dependent. As an example of our terminology, one of the failures in Squid (an assertion violation) occurred when a client sent an input request to the server and then disconnected from the server before the response for the request was written back. This is a timing-dependent bug as the symptom's occurrence depends upon when the disconnect request is sent to the server. The timing of the inputs matters and is under the client's control. In contrast, for a concurrency bug like a data race, the occurrence of the symptom depends upon timing issues that are beyond the client's control (e.g., thread scheduling); we classify such failures as *non-deterministic*, not as *timing dependent* .

Note that our study is conservative with respect to the definition of non-determinism. We classify a bug as non-deterministic if, according to the bug report, the symptom(s) could not be reproduced consistently for any reason (e.g., the same inputs may not be available or parts of the environment might have changed). It is possible that such bugs are deterministic, but we conservatively assume that they are not. Also, our definition of determinism is overly restrictive. We believe that many of the deterministic bugs in our study are actually deterministic across different environments and configurations as, in many cases, failures are reproduced by developers on different systems from the one where the bugs are first detected. But because bug reports often lack sufficient information about the bug's behavior across different environments, we chose to define deterministic to mean reproducibility in a fixed environment.

3.4 Classification of Software Bugs

We now present the results of our analysis of the selected server application bugs. We classify the software bugs based on three characteristics: observed symptoms (Section 3.4.1), reproducibility (Section 3.4.2) and the number of inputs needed to trigger the symptom (Section 3.4.3). Finally, we analyze the characteristics of a few important subclasses of bugs in (Section 3.4.4).

3.4.1 Bug Symptoms

We analyzed the bug reports in detail to determine the symptoms observed for each bug when the failure triggering input requests are sent to the server. For a small number of bugs, two symptoms were possible depending upon which inputs were used to trigger it; for these bugs, we chose to analyze the symptom that was given in the original bug report. The observed symptoms are memory error (segmentation fault, NULL pointer exception, and memory leak), program crash, assertion violation, program hang, and incorrect output. A program crash in this context is an abnormal server termination that is either not caused by a segmentation fault, NULL pointer exception, or memory leak or for which no cause is given in the bug report. We made this distinction because segmentation faults, Java NULL pointer exceptions, and memory leaks all have a memory error as the root cause while other termination errors may have other root causes. Assertion violation symptoms include explicit C/C++ checks and Java exceptions other than NULL pointer exceptions.

Table 3.2 shows the results. Each column shows the number of bugs that result in the corresponding symptom for each application. The last column shows the number of cases in which the bug report did not clearly identify the symptom; there were only 10 such cases

Application	Deterministic	Timing-dependent	Non-deterministic	Unclear from bug report
Squid	28 (70.00%)	3 (7.50%)	7 (17.50%)	2 (5.00%)
Apache	41 (78.85%)	0 (0.00%)	4 (7.69%)	7 (13.46%)
sshd	49 (90.74%)	1 (1.85%)	2 (3.70%)	2 (3.70%)
SVN	11 (91.67%)	0 (0.00%)	1 (8.33%)	0 (0.00%)
MySQL	46 (83.64%)	2 (3.64%)	4 (7.27%)	3 (5.45%)
Tomcat	43 (81.13%)	3 (5.66%)	4 (7.55%)	3 (5.66%)
Total	218 (81.95%)	9 (3.38%)	22 (8.27%)	17 (6.39%)

Table 3.3: **Symptom Reproducibility Characteristics.**

out of 266 bugs. The number in parentheses shows the fraction of the cases resulting in the symptom as a percentage of the total number of bugs.

The results in Table 3.2 indicate that most of the bugs (nearly 63%) result in incorrect output. Incorrect outputs can be caused by memory errors (e.g., buffer overflows, dangling pointers), integer overflow errors, off-by-one errors in loops, logical errors, etc. The results also show that Squid and Tomcat have many more assertion violations (28% and 23%, respectively), but a lower percentage of incorrect output errors compared to the other servers. We suspect this is because Squid has many more assertions in its code than the other servers and because Tomcat uses Java exceptions.

Implications:

These results lead to two important conclusions:

1. New techniques are needed to efficiently detect the data corruption that causes incorrect output errors at run-time so that future diagnosis tools will be able to detect and diagnose the majority of software errors.

2. The results from Squid and Tomcat suggest that adding assertions or automatically generated program invariants may help detect incorrect output errors.

3.4.2 Bug Reproducibility

We did a similar analysis to determine the reproducibility of failures due to server bugs; Table 3.3 summarizes our results. We classified bugs as either deterministic, timing dependent,

Application	# 0-input	# 1-input	# 2-input	# 3-input	# >3-input	Unclear	Max #inputs
Squid	8 (20.00%)	20 (50.00%)	1 (2.50%)	2 (5.00%)	2 (5.00%)	7 (17.50%)	3
Apache	3 (5.77%)	38 (73.08%)	1 (1.92%)	0 (0.00%)	2 (3.85%)	8 (15.38%)	2
sshd	4 (7.41%)	28 (51.85%)	14 (25.93%)	2 (3.70%)	2 (3.70%)	4 (7.41%)	12
SVN	0 (0.00%)	0 (0.00%)	9 (75.00%)	3 (25.00%)	0 (0.00%)	0 (0.00%)	3
MySQL	2 (3.64%)	0 (0.00%)	0 (0.00%)	40 (72.73%)	8 (14.55%)	5 (9.09%)	9
Tomcat	6 (11.32%)	34 (64.15%)	2 (3.77%)	1 (1.89%)	4 (7.55%)	6 (11.32%)	3
Total	23 (8.65%)	120 (45.11%)	27 (10.15%)	48 (18.05%)	18 (6.77%)	30 (11.28%)	12 (max)

Table 3.4: **Maximal Number of Inputs Needed to Trigger a Symptom.**

or non-deterministic. This property is useful, as it will determine whether diagnosis tools can reliably reproduce the failure symptoms. In some bug reports, the failure could not be reproduced or was very difficult to reproduce (as it occurred infrequently). We conservatively classified such bugs as non-deterministic. Some bug reports contained no information at all about reproducing the failure; these are counted separately in the last column of Table 3.3. The numbers in parentheses in Table 3.3 are the corresponding fractions as a percentage of the total number of bugs.

The results are encouraging. More than 82% of the bugs demonstrate deterministic behavior; these results agree with Chandra and Chen's results that nearly 80% of bugs are actually environment-independent [56]. A few bugs (nearly 3%) exhibit timing dependence. Only 8% of the bugs exhibit other non-deterministic behavior.

It should be noted that, in practice, automatic diagnosis tools will normally replay a subset of inputs. While deterministic bugs trigger the same symptom each time when run with the same input sequence, they may also depend upon global state such as memory layout. If using a subset of inputs creates a different global state, then the bug may not trigger the symptom again even though the same root cause is triggered. Section 3.7 discusses this issue in more detail.

Implications:

The above results indicate that:

1. Because most bugs are deterministic, bug diagnosis tools should be able to reproduce them by replaying inputs.

2. A small percentage of bugs are either timing dependent or non-deterministic. Bug diagnosis tools will need to incorporate new techniques (such as time-stamping inputs or controlling thread scheduling) in order to reproduce failures due to these bugs via input replaying.

3.4.3 Number of Failure Triggering Inputs

We now analyze and classify the software bugs based on the number of inputs needed to trigger the failure symptom. As mentioned in Section 3.2, we count each logical input as one input request. We determined the maximal set of inputs needed to trigger the symptom by examining each bug report and the other related external web resources linked to it. When more than one set of inputs could trigger a symptom, we used the largest set to compute the maximal number of inputs. We did not count changes to server configuration files or command line options as inputs. Table 3.4 shows the results of this analysis. The second, third, fourth, fifth, and sixth columns in Table 3.4 show the number of cases in which 0, 1, 2, 3, and more than 3 inputs are needed to trigger the corresponding symptom. The zero input column includes cases such as when the server experiences a failure after start-up but before it processes client input requests. There were a few bugs for which it was clear from the bug report that more than one input was needed to trigger the failure, but the exact number of inputs could not be determined as they were difficult to reproduce. We have counted them as requiring more than 3 inputs and included them in the sixth column. The seventh column, as in the previous section, shows the number of bugs for which the bug report contained too little information to determine whether one or more inputs are needed to reproduce the symptom. The last column shows the maximum number of inputs needed to trigger a failure due to any of the bugs we studied for that application. We used only the bugs for which we know the exact number of inputs to trigger a failure to compute the maximum number of inputs. As before, the numbers in parentheses in the table are the corresponding fractions as a percentage of the total number of bugs.

Appl	**# ≤1-input**	**#>1-input**	**Unclear**	**Max #inputs**
Squid	28 (70.00%)	5 (12.50%)	7 (17.50%)	3
Apache	41 (78.85%)	3 (5.77%)	8 (15.38%)	2
sshd	46 (85.19%)	4 (7.41%)	4 (7.41%)	11
SVN	9 (75.00%)	3 (25.00%)	0 (0.00%)	2
MySQL	42 (76.36%)	8 (14.55%)	5 (9.09%)	5
Tomcat	40 (75.47%)	7 (13.21%)	6 (11.32%)	3
Total	206 (77.44%)	30 (11.28%)	30 (11.28%)	11 (max)

Table 3.5: **Maximal Number of Inputs Needed to Trigger a Symptom Excluding Session Setup Inputs.**

Our results show that most failures can be triggered using a very small number of inputs. Some symptoms can be triggered with zero inputs. Squid had a few such cases (nearly 20%). For example, when run with a certain combination of options or with a particular configuration, Squid failed after starting execution but before processing any client inputs. The majority of failures in Squid, Apache, and Tomcat can be triggered with just a *single input request*, in the case of `sshd` and SVN, using just *two input requests*, for MySQL, using *three input requests*. All of the 2 input request failures in `sshd` and SVN require one input for authentication and session establishment and another final input to trigger the symptom. All of the 3 input request failures in MySQL require two inputs for authentication and database selection, providing a session and execution context in which the final input could trigger the symptom. Table 3.5 lists the number of failures that can be triggered with no more than one input request excluding session establishment inputs. If we exclude the session setup inputs, then nearly 77% of the bugs in all applications need just a *single input request* to trigger the symptom; in all such cases, we have observed that it is always the last input in the corresponding session/connection which triggers the symptom. Among the remaining cases, nearly 11% of the bugs needed more than one input to trigger the symptom, and the remaining 11% of bugs do not have any clear information about the number of inputs in their bug reports.

Furthermore, considering all the bugs for which we could determine the exact number of inputs, all of the failures for Apache, Squid, SVN and Tomcat can be reproduced using at most 2, 3, 2, and 3 input requests (excluding session establishment inputs), respectively; only two failures in `sshd` and one in MySQL required more than 3 non-login input requests. In fact, very few bug failures (12 bugs across all server bugs excluding the unclear cases) needed more than three non-login input requests to reproduce the symptom.

Another result (not listed in the tables) is that, for most of the bug reports we studied, the symptom occurs immediately after the last faulty input is processed. In fact, the only exceptions were hangs and time-outs; for these, the time between the last faulty input and when the symptom can be observed is time-dependent but usually small. This suggests that the error propagation chain for these bugs is usually short and that a symptom can usually be detected immediately after the server processes the faulty inputs. These results agree with previous work [57] that found that error propagation chains in the Linux kernel are short in practice.

Implications:

These results have several implications for automatic bug diagnosis tools:

1. Virtually all failures can be triggered by replaying a small number of inputs.

2. Most of the failures can be simply reproduced by first connecting to the server, creating a session if necessary (through authentication and/or a database select request), and replaying a single input.

3. For most of the bugs, the last input request from the session/connection which triggers the fault can be used to reproduce the symptom.

4. Except for bugs which cause a hang or time-out, the failure symptom for a set of faulty inputs will occur immediately after the last faulty input is processed.

3.4.4 Analysis

Interestingly, when we consider only the subset of non-deterministic or multi-input bugs, many characteristics are very different from the overall bug characteristics. For example, among the 22 non-deterministic bugs, a majority of them (45%) are multi-input bugs and only 40% were single input bugs (remaining cases were unclear). Also, only 23% of the 22 non-deterministic bugs result in incorrect output, most of the failures result in catastrophic failures like crash, segmentation fault, assertion violation and hangs. This implies that majority of non-deterministic bugs need multiple inputs to trigger a failure and many fewer of them result in incorrect outputs as compared to the overall bugs. Of the 30 multi-input bugs, only 40% are deterministic, 27% are timing-dependent, and 33% are non-deterministic. This implies that many fewer multi-input bugs show deterministic behavior compared to the overall bugs.

3.5 Multiple Input Bug Analysis

Bugs that require multiple inputs to trigger a symptom are harder to reproduce and diagnose because the faulty inputs may be interspersed with non-faulty inputs, and the combination of inputs to explore can be large. We did a more detailed study of multi-input bugs to see if there are patterns that can be exploited to reduce the input stream to just the faulty inputs. Specifically, we wanted to see whether the set of inputs for triggering a multi-input failure were likely to be clustered together within an input stream or occur within a short time duration, which can help automatic tools to detect the symptom-triggering inputs more easily. Otherwise, automatic tools will need to use more complex algorithms to track down the faulty inputs.

There were 30 multi-input bugs (5 from Squid, 3 from Apache, 4 from `sshd`, 3 from SVN, 8 from MySQL, and 7 from Tomcat). For servers like Apache, Squid and Tomcat, any bug in Table 3.4 with more than one input is considered a multi-input bug. For `sshd`, SVN, and

MySQL, we considered a multi-input bug to be any bug requiring more than two, two and three inputs, respectively, to trigger the symptom. Our rationale is that all of these two and three-input bugs require one and two inputs, respectively, for establishing a session e.g., authentication and/or database selection, and a final, single input for triggering the failure. These bugs are, in essence, single-input bugs with additional session establishment inputs that occur in a known location within the input stream and can be easily buffered for each session.

We classified each bug into one of three categories: *CLUSTERED*, *LIKELY CLUSTERED* and *ARBITRARY*. A bug is classified as CLUSTERED if the input requests must occur within some bound; this bound is often short. For this category, the faulty inputs are always going to be clustered within the input stream within a short period of time (less than a few minutes). We classify a bug as LIKELY CLUSTERED when we know that the faulty input requests are likely to occur within a short duration for most real-world inputs, but there is no bound. For these cases, there are reasonable cases in which the inputs may not be clustered. Inputs are classified as ARBITRARY if there is nothing to indicate that they must be or usually will be clustered within an input stream in real-world usage. This does not mean that the inputs will not be clustered; it simply means that there is no reason to expect that they are likely to be clustered for a single given input stream.

Tables 3.6 and 3.7 shows our detailed analysis results. The second column shows the bug's ID from the bug database, the third column reports the number of inputs needed to trigger the symptom, and the fourth column succinctly describes the steps needed to trigger the symptom. The fifth column explains why we classified the bug as we did, and the last column shows the classification we assigned to the bug.

We classified 8 of the 30 bugs as CLUSTERED and 9 bugs as LIKELY CLUSTERED. We deemed 13 of the bugs as ARBITRARY. The faulty inputs for the first two categories can be isolated relatively easily. Even for ARBITRARY cases, it should be noted that server applications normally run on multiple installations. To perform a successful diagnosis, we do

Appl	BugID	#Ip	Steps to trigger the bug	Conclusion	CLASS
Squid	1862	3	Send a POST request with an incomplete body, kill the origin server, and send the remaining body of the request.	All the events will happen within the time a complete request is processed, hence will most likely occur within a short time duration.	CLUSTERED
Squid	2276	>1	Send many NTLM authenticator requests.	The requests can be far apart.	ARBITRARY
Squid	500	3	Start downloading a file. Then start downloading the same file concurrently. Abort downloading the first file.	As the requests happen concurrently, they will occur within the duration of first download.	CLUSTERED
Squid	2096	2	Send a first request to a web page. Then, send a Second request to same webpage at nearly the same time.	Both the inputs will occur within a very short duration, before the completion of first request.	CLUSTERED
Squid	1984	>1	Send a lot of requests for a long time. No mention of any particular inputs causing the crash.	The requests can be far apart spread over a long period of time.	ARBITRARY
Apache	17274	2	First, try to authenticate with LDAP server with wrong login-password. Then try to authenticate again with same login.	Incorrect and correct logins will most likely happen within a short duration.	LIKELY CLUSTERED
Apache	33748	>1	Need a random number of inputs to cause the crash.	The requests can be spread over a long period of time.	ARBITRARY
Apache	34618	>1	Send requests which open a connection to LDAP server. First two requests work, but the third may crash server. Sometimes more such connections are needed for crash.	When many requests are needed, they may not occur within a short duration.	ARBITRARY
sshd	1156	3	First, login to a shell. Run any command, e.g. `sleep 1`; Then, Ctrl+c while the command is running.	Last two commands can happen within a short duration for short running commands.	LIKELY CLUSTERED
sshd	1264	12	Execute 11 commands through a library function cmd() which internally opens a new channel and closes it after executing the command.	All the commands will be sent consecutively and will most likely occur within a short duration.	LIKELY CLUSTERED
sshd	1432	3	Three successive failed login attempt.	All the three requests can occur consecutively (e.g. when someone doesn't remember the password).	LIKELY CLUSTERED
sshd	948	5	The bug is triggered by a denied ssh connection blocked by tcp_wrappers. The bug occurs after 5th blocked connection.	Five blocked connections can happen over a long period of time.	ARBITRARY
SVN	1626	3	First, login to server. Run any svn command, e.g. `svn commit`; Then, Ctrl+c while the command is running.	Last two commands will occur within a short duration, before the svn command completes.	CLUSTERED
SVN	2288	3	First, login to server. Then, run `svn lock test.txt;` followed by `svn commit test.txt;`, when there is no write access on root.	As in many cases, users commits after small changes, the last two commands will be within a short duration in such cases.	LIKELY CLUSTERED
SVN	2700	3	First, login to server. Then, run `svn lock test.txt;` followed by `svn commit test.txt;`.	As in many cases, users commits after small changes, the last two commands will be within a short duration in such cases.	LIKELY CLUSTERED
MySQL	1890	>1	Execute a series of INSERT queries in main thread, Execute some SELECT, UPDATE and DELETE queries on the same table in background thread.	According to the report, error occurs always after some queries execute concurrently, likely to be close together.	LIKELY CLUSTERED
MySQL	5034	5	`prepare stmt1 from "select 1 into @arg15"; execute stmt1; execute stmt1;`.	The execute stmt can occur far apart.	ARBITRARY
MySQL	8510	4	`set sql_mode = ONLY_FULL_GROUP_BY; then select round(sum(a)), count(*) from foo group by a;`.	In cases, when sql_mode is not set in the config file, it can be set long before running group by based queries.	ARBITRARY
MySQL	27164	5	Create InnoDB table; Immediately create MyISAM table containing a POINT column; Insert into the table with the POINT column.	In most cases, insert queries will be immediate after table creation.	LIKELY CLUSTERED

Table 3.6: **Analysis of Multiple-Input Software Bugs.**

Appl	BugID	#Ip	Steps to trigger the bug	Conclusion	CLASS
MySQL	4271	5	`prepare stmt1 from <explain complex select>; execute stmt1; execute stmt1;`.	The execute stmt can occur far apart.	ARBITRARY
MySQL	3415	6	In first thread `LOAD DATA INFILE 'file' into mytable;`, in second thread `INSERT INTO mytable2 VALUES(2)`.	As both requests are processed concurrently, they should occur within a short duration.	CLUSTERED
MySQL	15302	4	`load data from master; execute any command.`	As the requests are executed consecutively, they should occur within a short duration.	CLUSTERED
MySQL	186	9	In Master `create temporary table t(a int); reset master;`, in SLAVE `stop slave;reset slave;start slave;`.	reset master can occur after a long time of create.	ARBITRARY
Tomcat	37150	>1	When there is more than 1 simultaneous connection, run a long request like big dir listing.	As the requests are processed concurrently with other connections, they should occur within a short duration.	LIKELY CLUSTERED
Tomcat	27104	>1	Few inputs with clustering and session replication needed to trigger exception.	The inputs can occur independently.	ARBITRARY
Tomcat	37896	>1	When requests are being processed, kill one of the replication servers, all the web servers will fill up the max threads.	All the inputs will occur within socket timeout period, after which the threads will be unblocked.	CLUSTERED
Tomcat	26171	3	Start session through webpage, restart webapp, reload webpage in that session.	The two requests can occur far apart.	ARBITRARY
Tomcat	42497	2	Request a static file, get a 200 response with ETag. Request the same file again, getting a 304 response without ETag.	The two requests can occur far apart.	ARBITRARY
Tomcat	40844	2	Authenticate two users simultaneously with HTTP DIGEST.	The two requests will occur within a short duration.	CLUSTERED
Tomcat	45453	>1	Send requests to make JDBCRealm cache PreparedStatement and preparedRoles. Run two requests allowing two threads to call getRoles simultaneously.	First set of requests can occur far apart.	ARBITRARY

Table 3.7: **Analysis of Multiple-Input Software Bugs (Cont.).**

not need to reproduce the failure at every possible installation; it is sufficient to reproduce it on a single installation. Thus, it is enough for the automated tools to work if the faulty inputs will cluster in at least one instance.

For designing bug diagnosis tools that replay input, the important factor is the time between the *first* faulty input and the symptom; this delay determines the minimum amount of information a replay tool must record in order to be able to catch all of the faulty inputs. Since we have determined that most faulty inputs are clustered together within an input stream, we know that the time between the first faulty input and the last faulty input is small, and since the symptom will occur shortly after the last faulty input (as described in Section 3.4.3), we can conclude that the time period between the first faulty input and the symptom is also small.

Implications:

There are two important implications of our results:

1. Most multi-input bugs (except for those in which the symptom is a hang or time-out) will trigger the symptom shortly after the first input. This means that a replay tool only needs to record a small suffix of the input stream to reproduce the failure.

2. The locality of multiple faulty inputs within an input stream makes it easier to create a reduced test case.

3.6 Study of Concurrency Bugs

Surprisingly, we found only three concurrency bugs out of 160 bugs in the 3 multi-threaded servers (Apache, MySQL and Tomcat) we studied in Section 3.4. This strongly indicates that there are relatively few concurrency bugs in servers relative to the *total* number of reported bugs (similar observations can be drawn from the data in [58]). One possible reason is that servers generally process each request relatively independently, producing fewer inter-thread interactions than other multi-threaded programs with more intricate sharing behavior.

Nevertheless, concurrency bugs do occur and may have different characteristics from other bugs as they usually involve multiple threads and may be difficult to reproduce. We therefore manually selected and analyzed a set of concurrency bugs to determine if they show behavior similar to that of the bugs in Section 3.4. We now present the results of our analysis of these bugs. As before, we classified the concurrency bugs based on three characteristics: observed symptoms, reproducibility, and the number of inputs needed to trigger the symptom.

We selected 30 concurrency bugs from Apache, MySQL and Tomcat by searching the bug databases based on a set of keywords like 'race(s),' 'atomic,' 'concurrency,' 'deadlock,' 'lock(s),' and 'mutex(s)' in a manner similar to Lu et al. [58]. The other servers have few

Appl	**SegFault**	**Crash**	**Assert Vio.**	**Hang**	**Incorrect Output**	**SegFault/ Incorrect Output**	**SegFault/ Assert Violation**	**Crash/ Incorrect Output**	**Crash/ Hang**
Apache	1 (11.11%)	0 (0%)	0 (0%)	4 (44%)	1 (11.11%)	1 (11.11%)	1 (11.11%)	1 (11.11%)	0 (0.00%)
MySQL	1 (9.09%)	0 (0%)	0 (0%)	4 (36%)	5 (45.45%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	1 (9.09%)
Tomcat	1 (10.00%)	1 (10%)	6 (60%)	1 (10%)	0 (0.00%)	1 (10.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Total	3 (10.00%)	1 (3%)	6 (20%)	9 (30%)	6 (20.00%)	2 (6.67%)	1 (3.33%)	1 (3.33%)	1 (3.33%)

Table 3.8: **Classification of Bug Symptoms of Concurrency Bugs.**

Appl.	**Deterministic**	**Timing-dependent**	**Non-deterministic**
Apache	0 (0.00%)	0 (0.00%)	9 (100.00%)
MySQL	2 (18.18%)	2 (18.18%)	7 (63.64%)
Tomcat	0 (0.00%)	0 (0.00%)	10 (100.00%)
Total	2 (6.67%)	2 (6.67%)	26 (86.67%)

Table 3.9: **Symptom Reproducibility Characteristics of Concurrency Bugs.**

concurrency bugs. Out of these 30, 23 were data race bugs and 5 were deadlock bugs (it was not clearly reported for two of the bugs).

Table 3.8 shows the results of classification based on symptoms. Each column shows the number of bugs for the corresponding symptom or combination of symptoms. We include combinations of symptoms because some of these concurrency bugs were reported to produce different, yet specific, symptoms in different executions for the same input (e.g., a segmentation fault in some executions and an incorrect output in others). There are some interesting differences between concurrency and non-concurrency bugs. First, as expected, a much higher fraction of bugs produce hangs; most of them are due to deadlocks. Second, five (17%) of the concurrency bugs produced different, yet specific, symptoms in different executions, as noted above. Third, there are much fewer incorrect outputs with concurrency bugs (20% overall, but 45% in MySQL). It appears that a higher fraction of concurrency bugs produce catastrophic symptoms like crashes or hangs, which can cause service interruption. This observation is similar to that reported by Lu et. al. [58].

Table 3.9 shows our results from classifying bugs as deterministic, timing-dependent, or non-deterministic. Most of the bugs (87% overall, and 100% in Apache and Tomcat) show non-deterministic behavior. This underscores the difficulty of reproducing the symptoms of

Application	**# 0-2 input**	**# 3-8 input**	**# >8-input**	**Unclear**	**Max #inputs**
Apache	0 (0.00%)	0 (0.00%)	6 (66.67%)	3 (33.33%)	20
MySQL	0 (0.00%)	3 (27.27%)	5 (45.45%)	3 (27.27%)	16
Tomcat	0 (0.00%)	0 (0.00%)	6 (60.00%)	4 (40.00%)	15000
Total	0 (0.00%)	3 (10.00%)	17 (56.67%)	10 (33.33%)	15000 (max)

Table 3.10: **Maximal Number of Inputs Needed to Trigger a Symptom for Concurrency Bugs.**

concurrency bugs.

Table 3.10 shows the results of classifying bugs based on the number of inputs required to reproduce the bug (note that the columns differ from Table 3.4). For some cases, it was not clear from the bug reports exactly how many inputs were needed (for example, bug reports mention that they need to run some test cases in stress mode with multiple threads repeatedly for a few minutes to trigger a symptom). We have conservatively classified these cases as requiring more than eight inputs. The fifth and sixth columns are identical in purpose to those of Table 3.4. We used only the bugs for which we know the exact number of inputs to trigger them to compute the maximum number of inputs. As before, the numbers in brackets in the table are the corresponding fractions as a percentage of the total number of bugs. There are several differences with our earlier results for non-concurrency bugs. First, all the studied concurrency bugs need multiple non-login inputs (>2) to trigger a symptom (as a comparison, very few non-concurrency bugs needed multiple non-login inputs for reproduction). Some cases need significantly more inputs, but none of the bugs in Apache and MySQL (for which bug report mentions exact number of fault-triggering inputs) needed more than 20 inputs. Second, many bugs needed executions with multiple threads and multiple client connections for some time to reliably trigger the symptoms. (For most of the bugs, though, the bug reports mention that the symptoms can possibly be triggered using 2 or 3 threads and client connections.)

3.6.1 Implications

These results have several implications for automatic bug diagnosis tools for concurrency bugs.

1. As most of the concurrency bugs are non-deterministic and produce different symptoms, bug diagnosis tools will need to develop new techniques to reliably reproduce the symptoms.
2. A somewhat larger suffix of inputs will be needed to trigger the symptoms compared to non-concurrency bugs.
3. For most of the concurrency bugs, we will also need to use inputs from multiple different client connections. This can significantly complicate the process of reproducing the symptom and minimizing test cases.

Finally, note that our methodology successfully identified both non-deterministic behavior and also the need for multiple inputs in these 30 bugs. The *same methodology* found a very low occurrence of both these behaviors among overall reported bugs, as described in the previous section. This is an important validation of those (perhaps surprising) results for overall reported server bugs in Section 3.4.

3.7 Implications for Designing Automated Bug Diagnosis Tools

Bug diagnosis often involves one or more of the following steps: recording inputs that may lead up to a fault, reducing a small triggering input from the recorded inputs when a fault does occur, and using the inputs to somehow determine the cause of the bug.

New bug diagnosis tools may be able to exploit the bug characteristics we have found to improve their efficiency. We provide potential ways of using our results below.

3.7.1 Recording Server Input

Tools that automatically create reduced test cases for bugs need an recorded input that triggers the bug. However, recording all of the inputs to a server program is infeasible. Servers process vast amounts of inputs over long periods of time; a way to prioritize which inputs to save and which inputs to discard are needed.

Our results indicate that most bugs can be triggered with a single input, that some protocols require an authentication step before playing that single input, and that the inputs needed for reproducing a multiple-input bug are often clustered close together within the input stream. Additionally, we observed (like previous work [57]) that the symptom is often triggered immediately after the faulty inputs have been processed. Also, as discussed in Section 3.5, we found that the time duration from first faulty input to symptom is usually short for most bugs.

The implication is that an automatic bug diagnosis tool should be able to catch most bugs without recording the entire input stream. Rather, it should suffice to record a prefix of the per-session input (to replay session establishment and authentication, if necessary) and a suffix of the input (which will most likely contain all of the inputs needed to trigger the symptom).

However, Bug reports often try to reproduce a failure using a subset of the inputs containing all the failure-triggering inputs instead of the complete input stream. One concern is that the ability for a set of inputs to trigger a symptom may depend upon global state in the server that was affected by processing earlier inputs. For example, one set of inputs may cause the server to allocate memory in such a way that a buffer overflow triggers a segmentation fault while a subset of those inputs generate an incorrect output symptom due to a different heap layout. While a simple test case exists, how do we know that it can be reduced from a larger test case? In other words, can a suffix of inputs which contains all the failure-triggering inputs reliably trigger the same symptom that was triggered by the original input stream? We hypothesize that most deterministic bugs will generate identical symptoms even when earlier non-faulty inputs are removed from the input stream. Such bugs have a small number of failure triggering inputs and appear to have small error propagation chains. Moreover, Servers generally process each request relatively independently. Given this, we believe failure behavior of these server bugs is less likely to be affected by differences in global state.

We did a preliminary experiment to see if the global state of the server often affects whether a set of faulty inputs triggers a symptom. We selected four real bugs from four server applications: a stack buffer overflow bug due to an integer overflow (`sshd`), a NULL pointer dereference bug (Apache), and two heap buffer overflow bugs (Squid and NullHTTPD). For each bug, we created an input stream of 99 good inputs and appended to it the faulty input. We then proceeded to feed the input stream into the server, recorded whether the same symptom as reported in the bug report occurred, removed 10 good inputs from the input stream, and re-ran the experiment. We repeated this process until we had reduced the input stream to just 10 faulty inputs.

For all of the bugs we tested, the same symptom occurred after the faulty input was received regardless of the number of good inputs that preceded it. This suggests that the ability to reproduce a bug by replaying a suffix of the inputs to the server does not greatly depend upon global server state.

3.7.2 Automated Test Case Reduction

Our results may also help develop heuristics for guiding automatic test case reduction. For example, the ddmin minimizing delta debugging algorithm by Zeller and Hildebrandt [21] considers all test inputs as equally likely to trigger a symptom. However, our results indicate that most server bugs are single-input bugs, or require a small number of inputs that are clustered close together. As stated in Section 3.5, for most of the bugs, we find the time duration from first faulty input to symptom is usually short. Therefore, it may be possible to improve the efficiency of test case reduction algorithms by first systematically testing each small suffix of the recorded input stream to see if any of those inputs trigger the symptom before applying a more general algorithm like ddmin. For most of the bugs, this procedure is likely to be faster than ddmin. Also, some of the tools can possibly take benefit of the fact that very few inputs (usually ≤ 3) are needed to trigger the symptom by first searching small subsets before trying more complex algorithms.

3.7.3 Automated Bug Diagnosis Tools

The results of our study may also help improve tools such as Triage [12] that use checkpointing and replay to automatically diagnose bugs during production runs. Triage periodically checkpoints a program during normal execution [12]. When it detects an error, Triage instruments the code with additional detectors that help determine the bug's root cause and re-executes the program from the last checkpoint [12].

Our results have several implications for tools like Triage. First, a system like Triage can reduce the input stream to a much smaller set of inputs (as described previously), speeding up the bug diagnosis process by not replaying irrelevant inputs. Second, our results indicate that symptoms can be triggered by restarting the server and replaying a small number of inputs i.e., *without* checkpointing server state. This alleviates the need for checkpointing, However, techniques like checkpointing may not be practical due to various reasons. First, complexity of taking checkpoints (which needs specialized support from application and/or operating system [12, 7]) is a major issue. Second, multi-threaded process also complicates the checkpointing procedure further. Second, they can impose additional overhead on normal program executions. In addition, diagnosis cannot be done if the fault happens before the checkpoint is taken. Using a restart-replay mechanism will enable us to build a much simpler and much powerful automatic bug diagnosis tool.

3.8 Related Work

Several previous bug characteristic studies have examined the bug reports of both open-source and proprietary software. Gray [59] studied the maintenance reports of customer systems to study trends in system outages; his results show that software is the dominant factor in system outages. Lee and Iyer [60] studied the root causes and error propagation behavior of bugs in the Tandem GUARDIAN90 system and concluded that consistency checks detected slightly more than half of the bugs in software and prevented error propagation for 31% of the bugs [60]. Sullivan and Chillarege [61] studied the root causes and triggers for bugs and determined that most bugs were caused by memory errors. More recent work by Li et. al. [62] aimed to update the results of Sullivan and Chillarege by studying the Mozilla web browser and the Apache web server; they found that memory errors were not the dominate cause of errors in these applications. Furthermore, they found that incorrect program behavior was the most common bug symptom [62]; our study confirms this result. Lu et. al. [58] studied many characteristics of concurrency bugs in open-source software; they found, among other things, that most concurrency bugs can be reliably triggered by controlling the schedule of four or fewer memory accesses. Chandra and Chen [56] studied the bug reports of several versions of Apache, GNOME, and MySQL to determine whether application-agnostic recovery techniques could recover from the majority of bugs detected in production runs. Our study confirms their finding that most bugs are deterministic [56]. As far as we know, no other bug study has examined the number of inputs needed to reproduce bugs or aimed to answer questions about the feasibility of creating automatic diagnosis tools that replay inputs.

3.9 Conclusions

In this chapter, we reported the results of an empirical study of bugs found in open-source servers. Our results show that most server bugs are deterministic and that failures due to most bugs (77%) can be reproduced by replaying a single input request (after session establishment if needed). Even for the remaining multi-input bugs, the set of inputs needed for reproducing failures is usually small and are often clustered together within the input stream. For most of the bugs, the time duration between first faulty input and the symptom is small. Our results also showed that many bugs produce incorrect outputs, indicating that better detectors are needed to flag errors in production runs. Most of the concurrency bugs, though, need multiple inputs to reproduce a symptom and are non-deterministic. Finally, we discuss how the results can be used by automated tools for doing in-production bug diagnosis. One of the key implications of the study is that most of the failures may be reproduced without checkpointing server state.

Chapter 4

Invariant-based Automated Diagnosis of Software Bugs

Finding the root cause is the most important and difficult step for any automated debugging process. Earlier approaches for automated bug diagnosis have some drawbacks - some approaches do not scale to large applications, some require custom hardware support, some may miss root causes, others give too many false positives. We try to address these drawbacks in our framework for automated bug diagnosis. Our work on fault localization extends previous work that uses invariant-based diagnosis to narrow down the root causes of bugs.

Likely program invariants are program properties that are *observed* to hold in some set of successful executions but, unlike sound invariants, may not necessarily hold for all possible executions [26, 15]. Invariant-based approaches first extract likely invariants in some manner, and then run the program on an input that triggers the bug. Any invariants that are violated in this failing execution are assumed to be potential root causes. In this way, we use *likely program invariants* to provide us with a set of possible candidate root causes. By training likely program invariants on “good inputs,” violations of these invariants in a failing run can identify deviations between good and bad runs (i.e., possible bug locations and their consequences). The tradeoff, however, is that they can also be violated on valid behaviors not captured in the training runs (”false positives”) and, if the invariants are too broad, can miss bug locations (”false negatives”). Despite these tradeoffs, likely program invariants are powerful because they can capture properties *of the actual source code*, including properties that violate programmer intent (whereas programmer-written invariants, such as assertions in the code, will only express programmer intent); and, by observing patterns of behavior of program variables, (e.g., ”the variable `month` lies between 1 and 12”), they can capture deep algorithmic properties, which are captured in limited ways, if at all, by static analysis [63].

Previous work on invariant-based approaches has three major short-comings. First, they all use general test inputs for training, and because the coverage from test inputs is often low [64], root causes can be missed as a result. Second, using general test inputs also makes the likely invariants too broad (e.g., value ranges of variables may be too large) and not tailored to the specific failure, which can also make them miss root causes. Third, the previous approaches have not developed sufficiently practical techniques to filter out false positives, i.e., to narrow down the candidate locations of root causes. Some do not do much filtering [15], leaving a large number of possible candidates. Another recent work uses custom hardware support for forward slicing to filter out invariants that do not influence the observed failure symptom [16].

We propose a sophisticated approach combining likely program invariants with novel filtering techniques to narrow down a possible set of ("*candidate root causes*"). Our goal is to make this set as small as possible. We improve over previous work in two key ways. First, we generate likely invariants in a novel way by using a small set of automatically constructed good inputs that are as close to the given failing input as possible for training. We use an automated algorithm based on delta debugging (ddmin) [21] to construct similar inputs. By using inputs that are close to the failing input, rather than arbitrary inputs obtained ahead of time, the invariants that do fail are more likely to highlight the differences between the failing run and similar passing runs, i.e., to identify the root cause. Moreover, by using a *small* set of such inputs, our invariants are much tighter than would be obtained via random inputs, or via inputs used during general program testing. This has the benefit that the invariants at the root cause are more likely to be violated (fewer false negatives). It has the drawback that many other spurious invariants may be violated as well (more false positives). We consider this tradeoff worthwhile because *we consider it extremely important not to miss the root cause*, and because we believe we can use sophisticated filtering techniques to eliminate the false positives. Like our work, DIDUCE constructs invariants tailored to the failing run

– in fact, within the same run itself – but they do not do much filtering [15], to eliminate false positives, leaving a large number of possible candidate root causes.

Second, because tighter invariants can yield more spurious invariant failures, i.e., false positives, we employ several techniques to filter out these false positives. We use dynamic backward slicing to filter out invariants that do not influence the observed failure symptom; unlike previous work [16], we do this entirely in software, and we account for control dependence and not just data dependence. Additionally, we have developed two novel heuristics to further narrow down the root causes of failure. The heuristics are based on two observations. First, if an invariant on one instruction fails, then another instruction dependent on the first one may also be likely to have an invariant failure, but the underlying cause is the first, not the second. We remove the second from the candidate root causes and call this *dependence filtering* step. Importantly, this heuristic leverages the data-flow and control dependence analysis used in the slicing algorithm directly. Second, if multiple similar inputs produce the same failure symptom, they are likely to have the same root cause, and therefore, we can focus on invariants that fail for all such inputs. Again, this heuristic directly leverages (indeed, simply reapplies) the previous steps, but for additional failing inputs.

For each failure, our tool will perform the diagnosis automatically and present back to the programmer the set of program statements with failed invariants after each filtering step as candidate root causes. It will also present the local sub-expression rooted at each candidate root cause since the root cause may be anywhere in that expression. The programmer can investigate the candidate root causes after the final filtering step and their sub-expressions first. If the programmer can't find the root cause among them, the tool will present the programmer with the candidate root causes at the previous filtering step and so on.

In this chapter, we first describe the overall approach of our complete software bug diagnosis framework in Section 4.1 and then describe the basic invariants-based diagnosis to identify first set of candidate root cause(s) in detail in Section 4.2. The procedure to construct inputs to generate invariants is explained in Section 4.3 in more detail. In the next chapter 5, we describe the various filtering techniques we use to reduce the false-positive invariants.

4.1 Overall Invariants-based Approach for Software Bug Diagnosis

Some previous automated bug diagnosis techniques use program invariants [15, 11]. Others, such as Delta Debugging [13, 19], compare a failed execution of the program with a successful execution to identify the root causes of a failure. One drawback of Delta Debugging is that comparing individual dynamic statements or values between a failed and a successful run can be inefficient and unscalable. Instead, we can use program invariants to make an efficient comparison of the program runs. Thus, *our approach essentially combines the two prior approaches.*

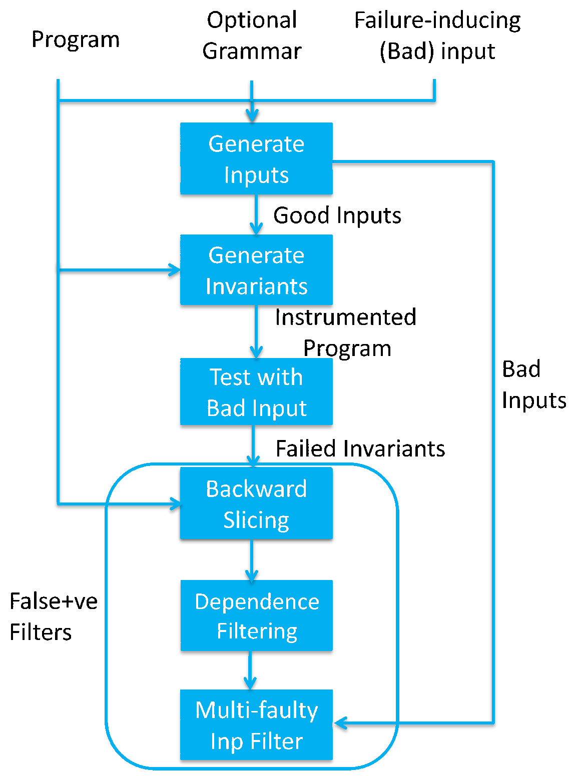

Figure 4.1: **Diagnosis Tool Architecture**

Our approach needs the source code of the program, configuration files, a failing input and a deterministic detector (described in more detail in Section 6.1.2) to accurately identify the root causes. Figure 4.1 gives a broad overview of our approach. The process begins with a program and a failing input, plus an optional grammar specifying the possible tokens in valid program inputs and all possible replacement tokens for each input token, and finally generates a set of failed invariants, which identify locations of the possible root causes. The final step (not shown) maps each location back to a set of source statements. We now describe the approach in more detail.

Definition 1. *Program invariants are program behaviors that are guaranteed to hold for all possible inputs.*

Definition 2. *Program behaviors that are observed to hold for some, but perhaps not all, program inputs are known as likely invariants [26].*

Likely invariants can be extracted by monitoring the execution of the program for some desired set of program inputs; we refer to these as the *training runs.* If we instrument a program to test likely invariants and then execute it with the failing input, any invariants that fail are a strong indicator of differences from successful runs. Thus, we can use locations of failed invariants as an initial set of *candidate root causes* of the bug. Note that true invariants are never violated and so do not help to identify candidate root causes.[1]

The key problem we must address in this work is that this initial set of candidate root causes is often very large, e.g., with hundreds of locations.

Definition 3. *We define all the candiate root cause locations which are not true root causes as false positives.*

Hence, all the candiate root cause locations except the true locations of the root cause will be *false positives.* The challenge is to filter down the set of candidates into as small a

[1]We therefore often omit the qualifier "likely" in the rest of this document; all unqualified uses of "invariant" implicitly mean "likely invariant".

set of locations as possible. We use a series of dynamic analysis steps based on dynamic program slicing and dependence filtering to filter out nearly all the false positives.

We use the example in Figure 4.2 to illustrate the steps of our approach. This is a simplified version of some code in the MySQL database server which contains a buffer overflow bug. The buffer overflow occurs when a *Select* query contains a specific date between 0000-Jan-01 and 0000-Feb-28. The buffer overflow occurs at line 31 when the `type_names` array is indexed with `weekday` for the aforementioned dates. The root cause of this bug is at line 10 in function `calc_daynr`. It uses an unsigned variable `year` to perform computation. When the value of `year` is 0, the decrement statement at line 10 results in a very large number and, consequently, the value of `temp` at line 13 also becomes a very large number. Finally, at line 14, the function returns a negative number; this value is used to calculate the day of the week in function `calc_weekday`, which, in turn, returns a negative value. Hence, when the return value is used as an index in line 31, it causes a buffer overflow and a program crash.

4.1.1 Generating Similar Inputs

The first step in our approach is to generate similar passing and failing inputs from the original failure-inducing input. As explained in Subsection 4.2.2, this step is important for the overall success of our approach. We employ three different types of algorithms to generate similar inputs. The main idea is to systematically remove or modify the parts of the original input using algorithms based on delta debugging (ddmin) [21]. If an optional input grammar specifying the tokens in valid program inputs is given then modification can be done more efficiently at the token level as described in detail in Section 4.3. In this way, we generate many similar good inputs which avoid the original failure. In a similar way, we can generate many similar failing inputs. Our ability generate such similar failing inputs enabled filtering techniques like multiple failing-input filtering as discussed in Subsection 4.1.5. We next describe the overall approach to localize faults using such similar good and bad inputs.

```
long calc_daynr(uint year,uint month,uint day)
{
  long delsum;
  int temp;

  if (year == 0 && month == 0 && day == 0)
    return (0); /* Skip errors */
  delsum= (long)(365L*year+31*(month-1)+day);
  if (month <= 2)
      year--;
  else
    delsum-= (long) (month*4+23)/10;
  temp=(int) ((year/100+1)*3)/4;
  return (delsum+(int) year/4-temp);
}

int calc_weekday(long daynr, bool first_week_day)
{
  return ((int)((daynr+5L+(first_week_day ? 1L : 0L)) % 7));
}

bool make_date_time(TIME_FORMAT *format, M_TIME *t,
                  timestamp_type type, String *str)
{
   str->length(0);
   if (l_time->neg)
     str->append('-');
...
  weekday= calc_weekday(calc_daynr(t->year,t->month,
                                                t->day),0);
  str->append(loc->d_names->type_names[weekday],
      strlen(loc->d_names->type_names[weekday]),
                              system_charset_info);
...
}
```

Figure 4.2: **Source Code Example**

4.1.2 Likely Program Invariants

We use program invariants to find a set of candidate program instructions that are possible root causes of a failure. Given an input that triggers the bug, we obtain a set of good inputs that do not trigger the bug from the failure inducing input using the previous step and we then run the program with these good inputs to find a set of program invariants. The invariants we use for our work are *range invariants* (described in Section 4.2.1). We limit our invariants to load, store or function return values, for reasons explained in Section 4.2. For example, we monitor the return value of functions such as `calc_daynr` and `calc_weekday`, as well as the loads of the three fields of `t` in the arguments to `calc_daynr`. For all good inputs, the return values of the two functions will be positive; this will be captured as likely invariants. When the program is run on the failure-inducing input, these two invariants (along with some other invariants) will fail as the return values of these functions will be negative. This will give us a number of candidate locations which can be root causes.

Overall, this step generated 95 failed invariants for this particular bug in MySQL. While this is far too large to report to programmers, it is still an impressively low number (especially considering that we are using one of the simplest possible types of invariants, ordinary ranges). This is strong evidence that the overall approach based on invariants constructed from nearby good inputs is promising.

4.1.3 Dynamic Program Slicing

Most failed invariants are not root causes of the bug. For example, some failed invariants are on instructions that do not compute any values that affect (directly or transitively) the instruction at which the symptom occurs. We can eliminate any such instruction from the set of possible root causes.

Dynamic backwards slicing [65] is a technique that can precisely determine which instructions affected a particular value in a single execution of a program. We start from the statement which causes the symptom (in this example, the invalid array element access) and find the statements in the execution that could have affected that statement, either via data flow or control flow. This set of statements is exactly the dynamic backward slice. We then remove all the failed invariants which are not part of this slice from the set of possible root causes.

In the part of the code shown in Figure 4.2, there are actually no failed invariants for the failing execution (though there are fairly many in other parts of the program). However, imagine that statements 25 and 26 had generated failed invariants. They would have been false positives since they are not the root cause of the failure. However, these statements are not part of the dynamic backward slice because they do not affect the value at the symptom, and so they would have been filtered out from the candidate set. Note that both function returns are part of the dynamic backward slice and would be included in the candidate set.

For this bug, this step removes about 62% of the failed invariants, reducing the candidate count from 95 to 36.

4.1.4 Dependence Filtering

Another source of false positives is error propagation. If a faulty instruction that is the bug's root cause triggers an invariant failure, then any instruction using the faulty value computed by that instruction may also trigger an invariant failure. Such subsequent invariant failures are not indicative of a root cause; they merely occur because the erroneous value is propagating through subsequent computation.

Using this observation, we have devised a heuristic called *dependence filtering* to further reduce the set of candidate root causes. This heuristic uses the data flow graph and control dependence graph formed during the previous dynamic slicing step to eliminate any instruction with a failed invariant that depends on another instruction with a failed invariant, *with no intervening passing invariants.* This policy is explained in Section 5.2.1.

For the example bug, when we build the dynamic dependence graph starting from the failing array indexing statement, the failed invariants of `calc_daynr` and `calc_weekday` will lie on one path to the symptom. Since the result of `calc_daynr` is directly used by `calc_weekday` with no intervening instructions that have invariants, the error at `calc_weekday` is likely to be propagated from `calc_daynr`, and we exclude the return value of `calc_weekday` as a candidate.

This step removes about 55% of the 36 remaining candidate root causes, leaving only about 16 candidates.

4.1.5 Filtering Using Multiple Failing Inputs

Note that once the invariants have been computed, the remaining steps above (finding failing invariants; dynamic backward slicing; dependence filtering) all happen using a single failing input. However, we can use generated inputs from the first step which are similar to the original input and also cause the program to fail with exactly the same symptom to further reduce the candidate set. Since these new bad inputs result in the same failure, it is very likely that the root cause will be the same. We can therefore *repeat the last three steps above* for each such failing input. The root cause should be included in the final candidates generated for all these inputs. We can then take the intersection of these sets to find the final set of candidates. For this bug, this step reduced the final set of candidates by 25% from 16 to 12.

The tool will first present the set of candidate locations from the final step to the developer for analysis. If developer cannot find the root cause among them, then tool will present the other candiate root causes in the order depending upon the number of other failing inputs in which they appear. If the root cause cannot still be identified, then the tool will present root causes from previous steps in this order - dependence filtering, slicing and likely invariants.

Overall, our diagnosis has narrowed down the possible location of the bug *to 16 locations in a program of over 1 million total lines of code* for this bug, since root cause of this bug cannot be found in final step, but can be found among the candidate locations of dependence filtering step. Moreover, for each of these locations, we have fairly detailed information about the ranges of program values seen in the successful executions, and the corresponding actual values from the failing runs, which fall outside those ranges. This is an extremely effective result because it should vastly simplify the task of the programmer in debugging the error.

4.1.6 Feedback to the Programmer

Each candidate invariant in the final or non-final set represents an instruction that may indicate the location of a root cause of the failure. Recall, however, that we only track invariants on load, store and return instructions. This means that the actual location of a bug may be anywhere in the maximal sub-expression rooted at a load, store or return that does not itself contain any load, store or return (We call this a *pure, local sub-expression*). Therefore, the tool additionally computes the maximal pure, local sub-expression rooted at each candidate root cause and presents it back to the programmer.

For the example bug, the invariant on the return value of `calc_daynr` indicates the location of the root cause, but the true location is somewhere in the body of that function. (Note that the entire function body is pure and local, after register promotion of local stack variables.) Therefore, the tool would present to the programmer the subset of the body of the function that is used to compute the return value in the failing run, i.e., lines 6, 8, 9, 10, 13 and 14. Of course, the negative return value in the failing run is a huge clue that `temp` at line 14 must be a large value. `temp` is computed from `year`, and `year` is computed by decrementing the incoming parameter value. This can only produce a large value by overflowing. Thus, the decrement of `year` is the true root cause.

4.1.7 Program Analysis and Instrumentation Infrastructure

Our system uses the the LLVM Compiler Infrastructure [66] to analyze and transform programs. LLVM supports a number of languages, including C and C++. Its intermediate representation (the LLVM IR) represents a program as a set of functions; each function is a set of basic blocks with an explicit, statically known control-flow graph. Scalar variables are kept in Single Static Assignment (SSA) form [67], making definition-use chains explicit. Other variables (e.g., C arrays and structs) are kept in non-SSA form; scalar fields of structures and arrays elements are accessed using RISC-like load and store instructions.

4.2 Constructing Candidate Root Causes

This section discusses how we use invariants to construct a candidate set of possible root causes of a given failure. We assume that the program includes a set of *deterministic detectors* for some classes of bugs which may exhibit non-deterministic behavior. For example, memory safety errors can manifest in non-deterministic ways, but we can use detectors like SAFECode [55], Valgrind [68] or SoftBound [69] to ensure that they provide the same symptom every time they are triggered. These detectors are only used offline after a failure is detected; the detectors are not used, and therefore add no overhead, in production runs of the program. For incorrect output errors, we assume that programmer-inserted or automatically generated program assertions can be used for detection (programmer-inserted assertions are often available at development time). Also, these detectors can be used online during development time, since overhead is not an issue during development time for programs compiled with assertions included. For many applications, comparison with a different version of the application, other equivalent software or known output may work as a detector for incorrect output errors. An input grammar, only specifying the possible tokens in valid program inputs and all possible replacement tokens for each input token, is optional. If such a grammar is available, it can simplify and enable better input generation using our algorithms based on the *ddmin* [21] approach. We use these algorithms to construct inputs for generating invariants, as described in Section 4.3.

The input to our diagnosis is a source program, required configuration files, the "failing input" that triggered the failure, and the failure symptom detected by some deterministic detector (we may also need "expected output" for detection of incorrect output bugs in some applications).

When a program fails on a particular input but succeeds for an input that is very similar (i.e., close) to the failing input, there are likely to be relatively few differences in behavior between those two executions; the key insight behind the work on Delta Debugging is that these differences can help narrow down the root cause(s) of the failure. Our goal is to perform this narrowing automatically by using invariants for the comparisons to identify candidate root causes, followed by various filtering techniques to reduce false-positives. If we can construct a set of such similar good inputs, then we can automatically extract behavior patterns that are common to these inputs. If the execution with the failing input violates any of these behavior patterns, that violation is a powerful indicator of the possible root cause(s) of the failure. Currently, we measure similarity between inputs by using edit distance between two inputs. Two inputs are said to be more similar or closer, if the edit distance is lesser compared to others.

4.2.1 Likely Range Invariants

In this work, we use a simple form of program invariants — likely range invariants. A *likely range invariant*, or range invariant for short, is a range of values computed by some program instruction in the monitored runs. For example, if an instruction `load int* %p` is always observed to return a value in the range between -5 and 10 (denoted [-5,+10]) in some set of training executions, then that range represents a range invariant for that instruction. A range invariant can be *tested* in future runs by instrumenting the program to compare the value computed by the instruction to see whether or not it falls within the range.

Range invariants are attractive because they capture concisely the observed values of program variables and instructions. They are also very efficient to generate and monitor with linear-time overhead as they only require recording (or monitoring) program values independent of other values. Although we expect that other kinds of data and control invariants are likely to be useful in identifying candidate root causes, we leave it to future work to explore such choices.

Since programs can have a large numbers of instructions, we limit our monitoring of invariants to load, store, and function return instructions. For example, an assignment statement, `x = A[i] + b * B[j];` may have an error anywhere in the expression but we only monitor the loads of `A[i], b, B[j]` and the store to `x`. This means we cannot narrow down the location of an error to anything smaller than an entire subexpression terminating in a load or store, but we expect this granularity to be ample for programmers to pinpoint the root causes of a failure. It may not be obvious why we monitor loads as well as stores when loads almost always return values from previous stores. We do so because it allows us to detect errors like incorrect memory address computation (for example, errors in array index expressions, such as `A[i]` or `B[j]` above). We have not yet experimentally evaluated which values or instructions are better in identifying the root cause(s) in practice.

We extract range invariants from training runs using a set of good inputs. We generate invariants only for those parts of the code that are executed by at least one of the good inputs. We then instrument the program to test those invariants and re-execute the instrumented program using the failing input. If a particular range invariant is violated, i.e., the relevant instruction produced a value outside the observed range, we assume that the relevant instruction may contain the bug, i.e., we consider it to be a candidate root cause of the failure.

4.2.2 Motivation for Using Similar Good Inputs

The technique used to extract likely invariants can affect the accuracy and the number of candidate root causes. When a large number of arbitrary program inputs are used to extract range invariants (e.g., all inputs used during software testing), many of the ranges obtained are likely to be wide. Also, coverage from using a random set of training inputs or general test inputs is often low [64]. As a result, many program statements are not executed at all resulting in insufficient training. *This makes it less likely that such a range invariant will be violated during a particular failing run,* which means that the root cause of a failure may not be included in the initial set of candidate root causes. This was the approach taken by previous systems [15, 16]. Pytlik et.al. [27] used invariants generated from test inputs for fault localization of Siemens benchmark suite without much success.

Instead, we need "good" inputs that (a) result in execution whose control flow and intermediate program values are as close to that of the failing run as possible and, (b) are few in number, and use those to extract range invariants. Using close good inputs has two benefits. First, it ensures that the execution behavior of the training runs will be as close to the failing run as possible, and so tends to isolate the differences (even if differences are minor) better that might be the causes of the failure. Moreover, it allows us to use a very small set of training inputs (e.g., only 10 as opposed to thousands, as used in prior work [26]). For example, consider a bug in MySQL that is triggered by an uncommon `Date` field in a Query:

SELECT DATE_FORMAT("0000-01-01",'%W %d %M %Y') as a;

Constructing invariants using a general set of inputs would include queries for INSERT, DELETE, SELECT, JOIN, etc., most of which are likely to have very different behaviors from the failing input. One would need a large number of such inputs to be representative of program behavior. As mentioned earlier, when a large number of arbitrary program inputs are used to extract range invariants, many of the ranges obtained are likely to be wide and likely to miss the root cause. Using a set of inputs with very similar execution behavior including program control flow and intermediate values (e.g., variations of previous query with different `Date` fields) that do not trigger the error requires a much smaller set of inputs to represent the behavior we are seeking.

Using only a few such good inputs makes the invariants narrower (e.g., smaller ranges for range invariants), which is both good and bad. It is good because it makes it more likely that the different behavior will result in a failed invariant. It is bad because it may possibly cause a large number of invariants to fail. Since we aim to identify the root cause(s) of a failure, it is important not to miss the root cause in the first step itself and so the benefit far outweighs the disadvantage. The key goal of this work, therefore, is to compensate for any possible disadvantage, i.e., to filter out the false positives, reducing the final set of reported candidates as much as possible without eliminating the true root causes. Filtering techniques we employ are described in detail in Chapter 5.

As explained earlier, success of our approach depends upon generating good inputs with similar execution behavior like program control flow and intermediate values as that of the failing input and training invariants using those inputs. Hence, the first step of our overall process is to construct a set of "good" inputs that exhibit as similar execution behavior as possible to that of the failing input used for diagnosis. In this work, we create inputs with similar execution behavior by constructing a set of "good" inputs that are as close to the failing input as possible. We define "good" inputs and "close" good inputs as follows.

Definition 4. *We define "good" inputs as those inputs which avoids the original failure symptom of the failing input.*[2]

Definition 5. *We define an input as "close" good input, if the difference between the input and failing input in terms of edit distance is as small as possible.*

If the inputs are very close, their execution behavior is likely to be similar. So, *the overall*

[2]We expect that the approach would work even if such inputs trigger other unknown failures because our main requirement is to isolate differences that avoid triggering the failure being diagnosed. We have not encountered this situation so far.

goal of this step is to create a certain number of good inputs that are as "close" to the failing input as possible. We systematically modify parts of a small failing input to construct such similar "good" inputs.

4.3 Constructing Program Inputs Similar to the Failing Input

We have designed three input generation algorithms that construct good inputs (they can also be used to generate additional failure-triggering, or bad inputs by modifying the criteria they use for selecting which inputs to return). These algorithms use different approaches to generate inputs - one uses a deletion based grammar-independent approach described as *inp-gen-delete* in Algorithm 1, and the other uses a replacement based grammar-dependent approach described as *inp-gen-replace* in Algorithm 2. Both *inp-gen-delete* and *inp-gen-replace* take an input (e.g., the input that triggers the bug) and the number of desired good inputs and return a set of good (or failure-inducing) inputs. We designed a third approach to generate good and bad inputs for compiler bugs by using C-Reduce [28] tool.

4.3.1 Deletion Based Grammar-Independent Approach

The deletion approach initially uses *ddmin* [21] to generate a few good inputs. *ddmin* is a systematic procedure for generating minimal inputs by systematically searching subsets of the input and their complements for various sizes of subsets. As a side effect, it can also produce good and bad inputs while testing various subsets for failure symptoms. Since, in many cases, *ddmin* produces too few good inputs, we use *good-inp-gen* (a variation of the original *ddmin* algorithm) to produce more good inputs. *Good-inp-gen* takes as input the most similar input produced by ddmin (similarity based on edit-distance). This function starts by removing single characters and then removes exponentially more characters from

Algorithm 1 Delete-based Input Generation

```
// Generates num Good Input starting from a failing input using deletion-based approach
function INP-GEN-DELETE(input, num)
    GoodInput ← a good input from ddmin algorithm
    return good-inp-gen (GoodInput, num)
end function

// Generates num Good Input starting from a good input
function GOOD-INP-GEN(input, num)
    Inititialize queue GoodInpQ with input
    // Repeat till Queue of good inputs is empty or we get num good inputs
    while GoodInpQ $\neq \emptyset \wedge$ size of GoodInpQ $<$ num do
        nextGoodInp ← removeFront (GoodInpQ)
        add nextGoodInp to GoodInputList
        find-good-inp (nextGoodInp, length (nextGoodInp))
    end while
    sort GoodInputList based on edit-distance
    return first num elements of InputList
end function

// Generates Good Inputs by starting from n subsets of inp
// and then recursively increasing subset sizes till n
function FIND-GOOD-INP(inp, n)
    Split inp into n subsets of equal size
    for each subset s do
        if s is good input then
            enqueue(GoodInputQ, s)
        end if
        if complement of s is good input then
            enqueue(GoodInputQ, complement of s)
        end if
    end for
    if $n/2 > 1$ then
        find-good-inp (inp, n/2)
    end if
end function
```

the input as it tries to find more good inputs. Like *ddmin*, this also tries both a substring and its complement to increase the likelihood of quickly finding good inputs. To determine whether the inputs are good or bad, we currently use a known correct version of the application. Once it has generated a sufficient number of good inputs, *good-inp-gen* sorts the inputs in the order based on its edit-distance from the original failure-inducing input and returns a pre-defined number of elements from the front of the list. These inputs should be the closest inputs to the original input as they are the closest in length (note that *good-inp-gen* only generates inputs smaller than the original input).

4.3.2 Replacement Based Grammar-Dependent Approach

Both *ddmin* and *good-inp-gen* do not require knowledge of the input grammar, so they are easy to deploy. However, by utilizing the input grammar, we can create a more powerful input generator. An example of such a generator is *inp-gen-replace* which needs an input grammar specifying the possible tokens in valid program inputs and all possible replacement tokens for each input token. This generator uses the input grammar to identify tokens for each terminal in the input. It then systematically generates many variations for each token depending upon the type of the token (note that, in Algorithm 2, there is a $mk < type > Variations()$ function to generate inputs for each type of token). For example, *mkStrVariations()* systematically divides strings into 2^i parts for various values of i and then replaces each character with some other characters, which runs in time proportional to the size of the string. Similarly, *mkIntVariations()* systematically creates different integer values which differs by 2^i from original value for various values of i. For grammar symbol tokens which are not of any special type, we replace them with their matching tokens depending upon the grammar specification. For example, a token for an aggregate function in MySQL like MAX can be replaced by matching tokens like MIN, AVG, STD, SUM, and COUNT etc. Tokens of similar string functions like LOWER, UPPER, REVERSE, RTRIM, and TRIM can be replaced by one another. Matching date functions like DAYNAME, DAYOFMONTH, DAYOFWEEK,

Algorithm 2 Replacement-based Input Generation

```
// Generates num Good Input starting from a failing input using replacement
function INP-GEN-REPLACE(input, num)
    GoodInput ← a good input from ddmin algorithm or the original input
    return inp-replace-input-gen (GoodInput, num)
end function

// Generates num Good Input starting from a good input
// Creates variations for each token depending upon type
// Finally creates inputs by combining variations
function INP-REPLACE-INPUT-GEN(input, num)
    GoodInputs ← $\emptyset$
    for each token t in the input do
        if t is string then
            // Create variants by recursively replacing subsets
            newInputs ← mkStrVariations (input, t)
        else if t is integer then
            // Create variants by replacing with different integers
            newInputs ← mkIntVariations (input, t)
        else if t is float then
            // Create variants by replacing with different floats
            newInputs ← mkFloatVariations (input, t)
            ...
        else if t is grammar symbol then
            // Create variants by replacing with matching tokens
            newInputs ← mkSymVariations (input, t)
        end if

        newGood ← subset of newInputs that are good inputs
        GoodInputs ← GoodInputs ∪ newGood
    end for

    sort GoodInputs based on edit-distance
    return first num inputs in GoodInputs
end function
```

DAYOFYEAR, LAST_DAY, MONTHNAME, and QUARTER can be rreplaced by one another. Similarly, logical operators can be replaced by other similar operators. After variants of each input token are created, we generate inputs by combining the different replacement values for each token. Currently, we generate inputs each of which contain variations of only one token, as exponential blow-up is possible if we consider variations of many tokens in each inputs.

While building an input generator that utilizes an input grammar (specifying tokens and their replacements) may seem onerous, we believe that it can be made practical. First, it should be possible to build "input generator generator" tools (in the same vein as tools like Lex, Yacc and Bison) to make the creation of input generators semi-automated. Such a tool would take, as input, a grammar and a set of functions to generate new strings by replacing individual tokens; the output would be a program that generates inputs for that grammar. Second, many programs use standardized input grammars (e.g., all web servers use the HTTP/FTP protocol grammar; many databases use the SQL grammar; compilers use grammar for C/C++ etc.). Once an input generator has been created for a specific grammar, it can be reused by many applications using that grammar. Programs with complex inputs, like MySQL, or compilers have explicit tokenizers which drive their parsers, and it may not be very difficult to modify and use this for our input construction algorithm. Finally, creating an input generator is a one-time effort. For large applications that are used over many years, the short-term effort to create an input generator should yield long-term savings in bug diagnosis time and effort.

We built input generators for a subset of the inputs for MySQL and squid using the previous algorithms. For this we modified the applications so that after the input tokenization, the tokens are directly fed into our input generators rather than being processed by the application. We maintain a list for each group of matching tokens. Depending upon the token type, we create an array of variants for each input token. After all the tokens are processed, we combine the variants to constructs different inputs. We then use a different, but correct version of the application for checking the output of the buggy version to classify the inputs into three following different categories and order them based on edit-distance:

1. Bad: The variants which produces the original symptom of the test case.
2. Good: The variants which does not produce the original symptom or errors other than those in the original test case.
3. Unknown: The variants which does not produce the original symptom, but give other different kind of errors. We check this using different error codes returned by the server.

These algorithms are likely to produce many good inputs since the number of good inputs are likely to much more than failure-inducing inputs (Both algorithms filter out invalid inputs by examining the output of the execution). One reason why these algorithms work well is also the number of good inputs are usually much more than failing inputs, hence a systematic search of variations of inputs can easily unearth similar good inputs. We can also use a variation of these algorithms to construct some failure-inducing bad inputs as well (which can be used to reduce the number of candidate root causes in a filtering step, as described in Section 5.2.2). We also observe that multiple input generators can be used in a single debugging session to provide a variety of inputs.

4.3.3 Input Construction for Compiler Bugs

Compilers are a very complicated and important piece of software usually consisting of hundreds of thousands of lines of code. They consist of various modules like tokernizer, parser, intermediate code generator, code optimization passes, back-end code generator and static analyzer etc. which closely interacts with one another. Errors in compilers can introduce latent bugs into the applications compiled by them which may not be clearly evident to the developers and may cause security/reliability issues. Hence, it is critical to fix the compiler bugs as quickly as possible.

Compiler bugs tend to have different characteristics as compared to bugs from other applications. Unlike server inputs, which tends to execute isolated parts of the source code, compiler inputs generally execute code from many different modules. Compilers involve many subtle interaction between different passes. Compilers also tend to involve more pointer-intensive code as well as many sophisticated algorithms. Compiler inputs also use a much more complex grammar. Hence, compiler bugs tend to be more complex and difficult to fix as compared to other applications. Compiler bugs like "wrong output code bugs" are even much more complex to diagnose. First, it is difficult to generate reduced test cases for such bugs. Second, it is not clear what should serve as the starting point for fault localization. Since, it is extremely hard to isolate wrong parts of the output, we may have to start from the complete output for dynamic backward slicing rendering it ineffective. As compiler bugs are harder, reducing the time fix these bugs is also more beneficial.

On the positive side, compilers also have a very well-define grammar. We could have directly applied the previous two algorithms to generate good and bad inputs for compiler bugs. However, unlike SQL or HTTP inputs, grammar for compiler inputs is much more complex. It would require a lot more effort to automate the input generation for compiler bugs. It was also not clear, if the previous two algorithms would directly work for compiler bugs or we would require more domain-specific techniques to be effective for compiler bugs. All these concerns led us to leverage an existing tool to automatically generate good and bad inputs for compiler bugs.

Hence for generating inputs for C/C++ compiler bugs, instead of implementing these algorithms from scratch we used an existing input-reduction tool, C-Reduce [28], for generating good and bad inputs. C-reduce tool is also based on *ddmin*, but is a much more powerful tool.

C-Reduce tool is an instance of generalized delta debugging algorithms. Generalized delta debugging [28] is a powerful framework consisting of a search algorithm, transformation operator, validity checking function and fitness function to reduce test cases. For transformation function to create variants, ddmin uses text-based substring removal strategy whereas hierarchical delta debugging removes AST-based subtree. In contrast, C-Reduce employs a much richer set of transformations which operate on many scattered parts of program at each step. For searching through the variants, C-Reduce employs a non-greedy strategy and allows transformations, which may currently increase size but ultimately leads to reduced sizes, to avoid getting stuck at a local minima. C-Reduce also avoids generating input programs with undefined or unspecified behavior by using validity checking function, which uses compiler warnings and tools like Valgrind, clang static analyzer, KCC [70] and Frama-C [71] to detect any undefined or unspecified behavior. Finally, C-Reduce uses a test script to determine whether a variant of a test case is successful (i.e. whether the variant triggers the same symptom as the original test case without any unspecified or undefined behavior) or unsuccessful.

At a high level, C-Reduce repeatedly invokes a collection of pluggable transformations till it reaches a global fixed point. A pluggable C-Reduce transformation is an iterator that points to the next transformation opportunity and walks through a test case performing source-to-source modifications [28]. C-reduce repeatedly applies these set of transformations till they run out of opportunities to reduce the test case any further. C-reduce implements five types of transformations.

In the first category of transformation, C-Reduce successively removes 1 or more contiguous lines of program code from the test case. The number of lines to remove starts with the number of lines in the original test case and halved at each step till it reaches 1 line.

Second transformation includes "peephole" optimizations which modify a contiguous block of tokens like changing identifiers and constants, removing blocks of code, removing operator and operands etc.

Third type of transformation makes localized modifications, but on non-contiguous blocks of tokens like removing matching parentheses/ curly braces without changing any of the code inside and replacing ternary operators (like ?:) with code from one of its branches.

Fourth transformation invokes external pretty-printing commands like GNU indent to reformat the code in test case to make it more readable.

As a fifth transformation, C-Reduce applies 30 source-to-source compiler-like transformations [28] to create reduced failure-inducing programs including:

- Scalar replacement of aggregates.
- Removing a level of indirection from pointer-typed variables.
- Factoring a function call out of a complex program expression.
- Combining multiple type definitions into a single compound definition.
- Making a function-scoped variable a global one.
- Renaming function, parameter or variable with shorter and better names.
- Function inlining of small functions.
- Applying copy propagation optimization.
- Converting unions into structs.

It should be noted that we have not changed anything inside the core of the C-Reduce tool. We only modified the test script used to determine if the variant is successful or unsuccessful. For this work, we tried to diagnose bugs of clang compiler [72] which is a C language family frontend for LLVM compiler infrastructure [73]. The test script uses gcc and the buggy version of clang to make this decision. We modified it to keep track of the successful and unsuccessful variants of the test case as the C-Reduce applies different transformations to create variants. Similar to the previous approaches, we classify the variant into three categories:

1. Bad: The variants which produces the original symptom of the test case.

2. Good: The variants which does not produce the original symptom or errors other than those in the original test case.

3. Unknown: The variants which does not produce the original symptom, but give other different kind of errors.

Finally, we order the variants based on the edit-distance from the minimal test-case and select the first few of the good inputs for generating invariants and first few of the bad inputs for multiple failing-inputs filtering during the fault localization process.

4.3.4 Related Work for Input Construction

There have been some earlier approaches like Fuzzing techniques [74] and input minimization tool like DDmin [21] which can generate variants of a given input. While doing input minimization, *DDmin* tool also generates some "good" and "bad" inputs during its search process. Some of our approaches are based on variations of *DDmin* approach.

Artzi et.al. [31, 32] developed a technique to automatically generate test cases using a form of concolic testing which they used for fault localization of PHP applications based on statistical techniques. They also explored the impact of size and selection of test cases on accuracy of fault localization for statistical techniques. We generate inputs by modifying input according to an input grammar. This strategy is less general, but more scalable for larger programs. Their work combined new test case generation techniques with known statistical fault localization. However, their technique can still be adapted for generating similar inputs for invariant generation, if their approach can scale to larger applications and longer executions and if our approach does not work for those applications.

Fuzzing techniques [74, 75] have previously been used to randomly modify inputs to create new inputs for blackbox testing. Other extensions of fuzz testing [76, 77] have used grammar to generate well-formed inputs. Probabilistic weights can also be assigned to individual production rules to better direct random test generation. Grammars also include rules for corner cases like large inputs and zero values to generate better tests for corner cases. They can also encode applications specific knowledge to guide test generation process better. In contrast, we only use the set of input tokens and their possible replacements to generate inputs which makes our approach more practical and general.

Recent advances in symbolic execution and dynamic test generation techniques [78, 79] have given rise to a new form of testing called whitebox fuzzing technique [80] for discovering new software bugs. In this approach, named SAGE, the program is symbolically executed and input constraints are generated corresponding to each conditional statement. These input constraints are then systematically negated and solved using constraints solvers to generate new test inputs which exercise different paths. This process is repeated using a search criteria for coverage maximization to discover new bugs quickly. SAGE also implements many optimizations which enable it to scale to longer execution traces with hundreds of millions of instructions. All these fuzzing techniques have been developed for better and faster test generation and bug detection. However, it is possible to suitably modify these techniques to create similar good and bad inputs. We would like to evaluate and combine some of these approaches with our developed approaches in future.

Zhang et.al.[81] have used predicate switching for fault localization. They try to forcibly switch predicates at runtime and check which predicate switch can alter the control flow so that the program successfully completes execution. They use this information to identify the root causes. They use Predicate switching to generate similar successful executions as compared to the failing execution without actually generating good inputs. Such similar successful executions can possibly be used for our invariant generation process. We leave it to future work to determine if this approach is feasible and compare it our current approach.

4.4 Summary of Basic Diagnosis Framework

In this chapter, we described the overall approach of our complete software bug diagnosis framework. We then described the details of our basic invariants-based diagnosis framework which identifies the first set of candidate root cause(s). Finally, we described algorithms to construct inputs which are similar to the original failure-inducing input. These inputs are used to generate invariants and also used later in multiple failing-input filtering step.

Chapter 5

Framework to Reduce the Set of Candidate Root Causes

After the basic invariants-based diagnosis system generates the invariants using the automatically constructed inputs and then determines the set of failed invariant to provide us with a set of candidate root causes, we use a sequence of techniques to reduce that set to a much smaller set. First, we use dynamic backward slicing of the failure symptom to remove any candidate root causes which does not affect the symptom in any way. We then apply two new heuristics, one using data-flow and control dependence and another one using multiple-failure inducing inputs to further reduce the candidate set of root causes. In this chapter, we describe these three filtering techniques in more detail.

5.1 Dynamic Backwards Slicing

We can first discard invariants on values that *do not* affect the behavior of the faulting instruction. We use dynamic backwards slicing starting from the failure symptom to find these values. Dynamic backwards slicing traces a program's execution and then finds the precise set of instructions that directly or indirectly influence the result of a given value in that execution [65]. For each execution of the program that triggers the failure symptom, we compute the dynamic backward slice of the symptom. Only failing invariants for instructions on this slice are retained. Our dynamic slicing system handles both data-flow and control-flow dependences when computing the dynamic backwards slice. The decision to support control-dependence allows our tool to find failed invariants on values that control whether critical pieces of code relevant to the bug are executed or not.

Most dynamic backwards slicing systems have two phases [65]: in the first phase, they instrument the code to record, in a trace file at run-time, sufficient information to find the backwards slice of any value. In the second phase, an analysis uses the execution trace to create a program dependence graph that can be used to find which computations affected the result of a value within that specific program execution. Since an execution trace is used, dynamic backwards slicing is precise; it will not include values that did not influence the computation in question [65].

We have implemented a dynamic backwards slicing compiler pass called *Giri*. Giri is very similar to Zhang and Gupta's No Preprocessing with Caching (NPwC) algorithm [65] in that it does not precompute a complete program dependence graph from the trace; instead, it consults the trace only on demand to compute the dynamic backwards slice of a variable and caches the result in memory for subsequent queries.

Giri takes advantage of the LLVM IR to reduce the size of the trace file. LLVM IR represents a program as a set of functions; each function is a set of basic blocks with an explicit, statically known control-flow graph [66]. Scalar variables are kept in Single Static Assignment (SSA) form [67], making the definition-use chains explicit [66]. Other variables (e.g., C arrays and structs) are kept in non-SSA form; scalar struct fields and array elements are accessed using RISC-like load and store instructions [66].

Giri instruments code so that it records three different pieces of information during the first phase: (a) basic block exits; (b) memory accesses and their addresses; and (c) function calls and returns. Using these values and the LLVM program representation (which holds all scalar temporaries in Static Single Assignment (SSA) form), Giri can easily construct the precise dynamic backward slice in the second phase.

In the first phase, Giri's instrumentation works as follows. First, Giri adds an instruction to each basic block that creates a record in the trace when the basic block has finished execution. These trace records are used in the second phase to determine which branches were taken dynamically during the program's execution. Note that recording the execution of each basic block (as opposed to recording the execution of each individual instruction) suffices since all of the instructions in a basic block must have executed before the last instruction in the basic block.

Second, Giri instruments the program to record the memory locations accessed by all loads, stores, and select C library functions. During the second phase, when it encounters a load instruction while finding the backwards slice, Giri can use the trace to find the store that created the value that the load read from memory. Giri can then continue backtracking from the store to find instructions influencing the loaded value. Note that consulting the trace is not required for finding the sources of values used by most LLVM instructions. Since most instructions operate on SSA scalar values, the input operands to those instructions can be determined from the explicit SSA graph in the LLVM IR representation of the program.

Finally, Giri instruments a program to record the execution of function calls and function returns. Having this information in the trace allows Giri to match up formal and actual arguments and caller/callee return values during the second phase for both direct and indirect function calls.

In the second phase, using these values, plus the LLVM program SSA representation (which holds all scalar temporaries in Static Single Assignment form), Giri can easily construct the precise dynamic backward slice. For SSA values in each instruction, it is straightforward to find the sources which will affect the instruction. For load values, we can use the memory trace to find the corresponding stores whose values are accessed by the load instruction. For the *Phi* instructions, the basic block trace is used to find the value which is the source for that instance of the dynamic instruction. Similarly, for the function call return values and formal arguments, function call-return trace is used. Giri also calculates dynamic control dependences as follows: when Giri first adds instructions from a dynamic execution of a basic block to the backwards slice, it uses static control-dependence analysis [67] to determine which instruction may force the execution of that basic block. Giri will then use the basic-block trace to accurately find the most recent execution of any instruction in that set which determines whether this dynamic instance of the basic block is executed and add it to the backwards slice.

By using the LLVM IR to reduce the trace size, our tool's traces are relatively small (43 to 144 MB in our experiments), making the use of more sophisticated optimizations unnecessary. To speed up the second phase, we perform one important optimization on the trace file. Giri performs a forward scan through the trace, recording the memory regions that have been modified by store instructions. If a load instruction is encountered that accesses a memory location not defined by a store (e.g., a non-modified global variable), it is given a special marking. This marking is used during backwards searches of the trace file to prevent tracing back to the beginning of the trace for loads without a defining store. Other optimizations are possible but are not implemented presently.

5.2 Filtering False Positives

In practice, dynamic backward slicing reduces the number of candidate root causes by a large amount (60% or more) but still leaves a fairly large number to be analyzed. We propose two new techniques to filter out false positives from the remaining set. One of these takes advantage of the dynamic tracing information, which allows us to filter invariants using the dynamic dependence graph (DDG), which is the union of dynamic data flow graph and control dependence graph. The second one takes advantage of our input construction capability to identify candidates that appear to explain the symptom for multiple failing inputs that are only slightly different from the original one.

5.2.1 Dependent Chains of Failures

We observe that the result of an anomalous computation (i.e., one that violates an invariant) can cause dependent instructions to produce anomalous values. For example, if the variable `x` fails a range invariant because it is negative when it was expected to be non-negative, then the result of a subsequent computation using `x` (such as `2*x`) that was expected to be non-negative will be negative, too, potentially failing its range invariant. The real culprit, however, is the first failure; the second failure is just a false positive and not an independent buggy location.

We can filter out such false positives by identifying dependent chains of failing invariants using the DDG. The DDG is a graph $G(N, E)$, where the set of nodes N is the set of instructions executed in a given run of a program, and E is the set of edges $n_i \rightarrow n_j$ such that n_i is an instruction that generates a value that is an operand of n_j or n_j is control dependent on n_i. Graph G must be acyclic since it represents data flow and control dependence within a dynamic sequence of instructions. Conceptually, this graph can be constructed easily from the dynamic trace together with the program, since together they identify the complete set of CPU operations, loads and stores, and the dependencies between them.

We can use a standard depth-first-search traversal on this graph to look for cases where a failed invariant is dependent on a previous failed invariant. However, not all instructions have invariants computed for them. We therefore look for direct paths between two failed invariants without any intervening failed invariants, $n_1 \rightarrow n_2 \rightsquigarrow ... \rightsquigarrow n_{k-1} \rightarrow n_k$, $k \geq 1$, where (a) n_1 and n_k both have invariants that failed; and either (b) $k = 1$, or (b′) all instructions n_i, $2 \leq i \leq k-1$, do not have any passing invariants at all. In other words, there are no intervening invariants between n_1 and n_k at all. We then simply drop n_k from the set of candidate root causes under the argument that it "went bad" simply because n_1 went bad, as discussed above, i.e., it was not an independent bug location. Of course, n_1 may itself get dropped if it is, in turn, dependent on some previous instruction that had a failed invariant. Observe that the key case we are excluding with this rule is when there is a *passing* invariant between n_1 and n_k. Intuitively, we are "trusting" the invariants to distinguish correct values from erroneous ones. If an instruction has a passing invariant, we assume that it did not produce an erroneous value, and therefore a later dependent instruction that has a failed invariant may be an independent source of an error. Of course, in practice, likely invariants are not strong enough to ensure such reasoning is sound. By retaining the later dependent instruction in this case, we are biasing the design towards retaining candidate root causes unless we have high confidence that they are not a true root cause. In practice, computing the DDG explicitly turns out to be unnecessary. Recall that the dynamic backward slicing algorithm, Giri, traverses the union of the DDG and the control dependence graph of the execution. We piggyback directly on Giri to identify dependence chains. Giri essentially uses depth-first search (DFS) on the union of the two graphs to construct the backward slice. Every time Giri visits an instruction I, we check if I had an invariant and, if so, whether that invariant failed or passed. If I failed an invariant, we traverse I's parents in the depth-first-search tree (which corresponds to the chain of users) until we arrive at an ancestor instruction, A_I, that has a passing invariant. We mark all intervening instructions between A_I and I for later filtering. We also stop this ancestor traversal if we arrive at a failed invariant and mark it for later filtering. This ensures that we never need to process any instruction (i.e., mark it and recursively traverse its parents) more than once during the ancestor traversals, though we may arrive at it as many times as there are incoming operands (i.e., incoming edges in the DDG). This ensures that the total time for this dependence filtering pass is linear in the number of edges in the DDG.

At the end of this algorithm, we filter out all remaining candidate instructions that were marked for filtering. In practice, this pass is also very effective at eliminating false positives. It is unfortunately possible, however, that this pass may occasionally eliminate a true root cause itself. In particular, it is possible that I_2 depends on I_1, both have failed invariants, but I_1 was actually a false positive while I_2 was a true root cause.

However, we expect this behavior to be uncommon with sufficient training. This scenario primarily occurs when the true root cause gets its values from an instruction which is a false-positive. In our experiments, we have found that only a tiny fraction of instructions with failed invariants remain after dynamic backward slicing. In other words, the probability that two dependent instructions have failed invariants without any intervening passing invariant and the second failure (root cause) is not a consequence of the first is very low. Sufficient training should make such cases an unlikely occurance.

5.2.2 Using Additional Failure-inducing Inputs

Although we found that the number of remaining candidate root causes after dependence filtering tends to be fairly low (e.g., in the range of 7-18, reduced from hundreds originally), we can do better without new mechanisms. We observe that, using essentially the same methodology that we used for nearby good inputs, we can construct additional failure-inducing inputs that are close to the original one *and produce the same failure symptom as the original one.* For each such input, since the failure symptom is the same *and* the inputs are very similar to the original one by construction, the true root cause of the failure is very likely to be the same.

This insight leads to the following simple strategy. Construct a set of additional failure-inducing inputs close to the original one that produce the same failure symptom. Repeat the previous analysis for each such new input: (1) identify all failing invariants when executing with that input; (2) filter out invariants not on the dynamic backward slice of the failure symptom; and (3) filter out invariants using the dependence filtering step above. This process gives a set of candidate invariants for this failing input. Compute the intersection of these sets from all the failing inputs. That intersection yields the set of invariants that are candidate root causes for all failing inputs. The resulting set represents the final output of our analysis.

5.3 Generating Results for the Programmer

Each candidate invariant in the final set represents an LLVM instruction that may indicate the location of a root cause. Recall, however, that we only track invariants on load, store and return instructions. The actual locations of a root cause may be anywhere in the subexpression rooted at a load, store or return from which wrong values flow through several statements before reaching the failed invariants. This means that we must identify the entire subexpression rooted at a load, store or return for the programmer.

We identify the operations involved in the subexpression using a simple backward traversal of the Dynamic Dependence Graph starting from the candidate root cause till we reach other load, store, return, and function arguments. Since, our current implementation do not monitor invariants on function arguments, the true root cause may actually be somewhere earlier along the data-flow and control dependence chain though we did not encounter any such case in our experiments. We could also place invariants on these subexpression values to remove parts of the subexpression that are not the root cause, but we do not have this implemented at present. Once all LLVM instructions within the subexpression are identified, our analysis maps them back to source statements using debug information.[1]

5.4 Summary of Filtering Techniques

In this chapter, we described the various filtering steps we use in our software bug diagnosis framework. We first described the details of how we implemented dynamic backward slicing. We then described the intuition for using dependence filtering and the implementation details of this filtering step. Finally, we described the multiple failing-input filtering step.

[1]Many compilers have inaccurate debug information when aggressive optimization is enabled. For such compilers, accuracy can be improved by disabling optimizations.

Chapter 6

Experimental Results of Our Fault Localization Framework

In this chapter, we describe in the detail how we apply our invariants-based approach to diagnose server and compiler bugs. Fault localization of server bugs may seem complicated as the servers normally run for a very long period of time, process a very large amount of data during that time period and mostly do the processing concurrently. However, as our empirical study in Chapter 3 found out, there are many characteristics of server bugs which also facilitate fault localization by reducing the impact of large amount of processed data and long running time of servers on fault localization. As our study implies, most server bugs could be reproduced with 3 or fewer inputs. Moreover, we usually need only a small suffix of inputs to reproduce failures after restart. So we can just restart the application and replay a small suffix of the inputs to reproduce the failure and perform fault localization.

Compiler bugs tend to have different characteristics as compared to other bugs and they are also usually more complex to diagnose. Hence, we also use five compiler bugs to further evaluate the effectiveness of our approach. We now describe the methodology of our fault localization experiments and the experimental results.

6.1 Experimental Methodology for Diagnosis of Server Bugs

In this section, we describe the experimental methodology we used to evaluate the effectiveness of our approach for server bugs.

6.1.1 Applications

In our experiments, we used three widely used large server programs: Squid HTTP proxy server, MySQL database server, and Apache HTTP web server. MySQL, which has about a million lines of code, is by far the largest application used to evaluate automatic diagnosis algorithms in the literature to date, except in the Triage work [12]. All three servers communicate with clients using well-defined protocols with well-defined grammars that are amenable to our input construction algorithm.

6.1.2 Software Bugs and Failure Detectors

We selected 8 real software bugs for these three server programs. We were limited in two ways. First, we needed bugs that we could reproduce using information in the bug reports for versions of the software that we could compile with LLVM on a current Linux system. Second, our current implementation cannot effectively handle most concurrency bugs as we currently need a deterministic detector. Our implementation also cannot handle missing code bugs (usually uninitialized variable bugs) because the root cause may not be effectively identified by range invariants on program values. We expect they will need control flow invariants, which are outside the scope of this work. Hence, out of 12 such bugs, we omitted four that are presently outside the scope of our tool.

Table 6.1 describes the bugs we used in detail. The first column provides a name for each bug for identification, using which we refer to the bugs later. The second column shows the name of the application which contains the bug. The third column provides the number of lines of code (in units of 1000 lines). The fourth column shows the unique number of source lines of code that are executed for the failure-inducing input which reproduces the failure due to this bug. We first use program trace to determine the LLVM instructions which are executed during the failing run and then use LLVM debug metadata information to map them to their corresponding program source lines. For some LLVM instructions, debug metadata may be missing due to optimizations. We count one source line for each such instruction. We present the number of executed source lines as it will indicate the effectiveness of our approach. But, the complexity of fault localization for a programmer will however depend upon the size or lines of code of each application. The next column in the table shows the failure symptom of the bug: three bugs were memory safety errors; the rest were incorrect output bugs. The last column provides a brief description of each bug.

The selected bugs have different root causes. For example, some bugs occur due to integer overflow or loss of precision. One buffer overflow bug occurred due to the use of an unsigned value for some computation which causes an integer overflow which after some computations leads to a use of negative index for an array access. Another buffer overflow bug occurred when a buffer length was incorrectly computed for some specific inputs. In another bug, using an incorrect MySQL object type for a MySQL item causes a segmentation fault. Another bug occurred because the wrong algorithm was used to convert a microsecond field in string format into an integer. In another case, incorrect truncation in a long series of computation leads to loss of precision and wrong result.

The root causes for many bugs are far from the symptom. This is especially true for incorrect output bugs where the symptom occurs after 10 to 100s of intervening function calls. These bugs can be much harder to debug. However, our approach is agnostic to the distance of root cause from symptom unlike other related work like triage [12]. Hence, our approach is likely to be equally effective irrespective of distance of root cause from symptom.

We do not use automated detectors for our experiments. For each memory safety error, we manually inserted program assertions at the program point where a tool like SAFECode would have detected a memory safety error. For each incorrect output error, we manually inserted an assertion at the program point that communicates the incorrect output to the client. In both cases, the assertions serve both as detectors and as starting points for dynamic backwards slicing. It is possible to automatically add these types of assertions at the program points which communicates to external world. For incorrect output errors, comparison with

Bug name	Application	LOC	LOC executed in bug	Symptom	Bug Description
Squid-len	Squid 2.3	70K	6927	buffer overflow	Incorrect computation of buffer length leads to buffer overflow
SQL-int-overflow	MySQL 5.1.30	1019K	9822	buffer overflow	Use of unsigned var causes integer overflow which finally leads to buffer overflow
SQL-convert	MySQL 5.1.30	1019K	9982	Incorrect output	Wrong algorithm to convert microsecond field to integer results in incorrect value
SQL-aggregate	MySQL 5.1.23a	1054K	11308	buffer overflow	Use of a negative number along with an aggregate function results in seg fault
SQL-precision	MySQL 5.0.18	949K	7874	Incorrect output	Loss of precision in a sequence of computation results in wrong value
SQL-mul-overflow	MySQL 5.0.18	949K	7835	Incorrect output	Overflow during decimal multiplication results in garbage output
SQL-dataloss	MySQL 5.0.15	937K	9835	Incorrect output	Loss of data when inserting big values in a table
Apache-overflow	Apache 2.2	225K	6052	buffer overflow	For particular value of output size overflow occurs

Table 6.1: **Server Software Bugs Used in the Experiments.**

a different version of the application, other equivalent software or known output may work as a detector for many applications. We can also use programmer-inserted assertions for detection (these are often available at development time).

6.1.3 Good and Bad Input Construction

Our proposed overall diagnosis approach begins with these steps:

1. Compute a minimal or small failure-inducing input from the original failing input, that fails with exactly the same symptom. These are often available from bug reports during development time.

2. Construct a small set of good inputs that are close to the minimal failing input.

3. Construct a small set of additional failure-inducing inputs that are close to the minimal failing input, and fail with exactly the same symptom.

For step 1, we already had fairly small failure-inducing inputs available from the bug reports for six of the eight bugs. For the remaining two bugs, we slightly reduced the inputs by hand by simplifying the input, but this reduction is not critical to our results. For steps 2 and 3, we constructed 8 good and 8 bad inputs for each MySQL and Squid server bug using the grammar-dependent and grammar-independent input construction algorithms described in Section 4.3. In case one of the algorithms did not generate enough number of inputs, we used more inputs from the other algorithm to gather 8 good and bad inputs in total for each bug. This is extremely small number of inputs compared with previous work that has relied on likely invariants, and works only because we construct the inputs to be as close to the failing input as possible. For Apache server bug, we could not automatically generate inputs since they required very particular input values and we have not yet implemented it to specially handle such cases.

However, before we developed algorithm to automatically generate inputs, we had developed a systematic procedure to create good and bad inputs. We used this procedure to generate inputs for Apache bug. Additionally, for comparision with automatically generated inputs, we also used this to generate inputs for the other 7 MySQL and Squid bugs. The steps 2 and 3, use exactly the same procedure in our systematic procedure, but simply retain only good inputs and failure-inducing inputs respectively. We had developed the systematic procedure on paper (which was also similar to automatic generation) and applied it manually. We describe this systematic procedure next.

Each of these two steps in the systematic procedure works through small, local changes on the (conceptual) parse tree of the input, as defined by the input grammar.[1] We use four primitive operations on the nodes of the tree: (a) replacing all or part of the token for a terminal with a different, legal token; (b) deleting all or part of the token for a terminal; (c) adding new terminals, with appropriate parent nodes (non-terminals); and (d) replacing the subtree rooted at a non-terminal with a pre-selected default production for that non-terminal.

The systematic procedure we follow in step 2 is described in Algorithm 3. Function *inp-gen-manual* takes the original failing input and constructs a specified number of good inputs. To construct the first good input, *find-one-good-inp* function tries the four operations in the

[1]We do not handle the case where the failure-inducing input is syntactically invalid.

Algorithm 3 Manual Systematic Input Generation

```
1: Four primitive operations on the nodes of the parse tree:
2: (a) Replacing all or part of the token for a terminal with a different token
3: (b) Deleting all or part of the token for a terminal
4: (c) Adding new terminals, with appropriate parent nodes (non-terminals)
5: (d) Replacing subtree rooted at a non-terminal with a pre-selected default production
   for it.
6:
7: function INP-GEN-MANUAL(faulty-input, num)
8:     GoodInput = find-one-good-inp(faulty-input)
9:     return GoodInput ∪ good-inp-gen (GoodInput, num - 1)
10: end function
11:
12: function FIND-ONE-GOOD-INP(input)
13:     for each operation x in this ordered set (a, b, c, d) do
14:         Apply operation (x) on all terminals and non-terminals of input.
15:         if 1 good input is found then
16:             return first good input found
17:         end if
18:     end for
19: end function
20:
21: function GOOD-INP-GEN(input, num)
22:     Apply operations (a), (b) on all terminals of input and add generated good inputs
    to GoodInpQ.
23:     return GoodInpQ when (GoodInpQ == num).
24:     Apply all possible operation (c) on input and add generated good inputs to Good-
    InpQ.
25:     return GoodInpQ when (GoodInpQ == num).
26:     Apply operation (d) on all non-terminals of input and add generated good inputs to
    GoodInpQ.
27:     return GoodInpQ when (GoodInpQ == num).
28: end function
```

order listed, until one good input is generated. We must run the program with each possible choice to evaluate if it is a good input or not. Once we have one good input, *good-inp-gen* function generates additional good inputs by trying both (a) and (b) first, and then (c) and then (d) on the first good input, until we have as many as we need. We use only 8 good inputs for each bug.

For constructing additional failure-inducing inputs from the minimal failure-inducing input in step 3, we use the same procedure, except that we accept a choice if it causes the program to fail with exactly the same symptom in the same location.

6.2 Experimental Results of Diagnosis of Server Bugs

In this section, we evaluate the effectiveness of our approach. In particular, we investigate (a) whether our approach can find the true root causes of bugs; and (b) how many false positives it generates. For the purpose of this evaluation, we use the following definition of root cause: for each bug, we use the correct patch in the bug reports to identify the minimal statements which should be changed or deleted to remove the failure symptom (The only cases where new lines of code are added are to replace some existing lines, since we do not include missing-code bugs in our experiments.) We say all these statements are part of the root cause, and we say our approach is successful if it can identify all these statements. This is a conservative definition since there can be many other places which could be changed to fix the same bug. But, even if we find one or more of such statements we count them as false-positive.

We used LLVM compiler [66] to compile the programs, to extract, check violations of invariants and to compute dynamic backward slices. We used "-g" option to compile the applications to obtain debug metadata information for source line mapping. We conducted the experiments on Linux 2.6 on a machine with a dual-core Intel ® Core ™ 2 Duo CPU running at 3 GHz with 4 MB cache and 8 GB of RAM.

The time for doing the complete diagnosis depends on the trace size and program size. For these bugs, the total diagnosis time varied a lot depending upon the application - from around 8 minutes upto 4 hours. For two bugs, we believe it took an excessively long time because our slicing implementation is not optimized. We expect to see many fold improvements in the execution time after a few optimizations.

The detailed experimental results using automated input generation are presented in Table 6.2. The first column in the table lists the bug numbers from Table 6.1. The second column shows the the total number of invariants that are generated and inserted by our system. The third column shows the total number of failed invariants by the original failure-triggering input which form the initial set of candidate root causes. For memory bugs or assertion violations where failure occurs soon after root cause, there may not be any invariant present between the root cause and failure symptom to catch divergent behavior. Failure location can give a strong clue about root cause for such cases. Hence, for such kinds of bugs, we also count the program statement at the failure location as a candidate root cause and include it in this column. The fourth column shows the number of failed invariants that are on the dynamic backward slice. The fifth column shows the number after the dependence filtering step and the sixth column shows the number after all the filtering steps including multiple failing-inputs filtering. The final column shows the maximum number of source lines corresponding to all the expression trees which the developer may need to analyze to fix the bug depending upon which latest step contains the root cause. The last column shows whether the candidates after the final multiple inputs filtering step includes the root cause or not.

Note that a single candidate sub-expression at the LLVM level may span multiple lines of source code. This means that the number of source statements is, in general, larger than the final candidate locations. We believe, however, that the difficulty of the programmer's final task is best measured by the number of candidate locations they have to understand, and not simply the total number of lines of source code. In particular, the number of source lines is

Bug Name	#Invs	Failed Invs	Slice	Dependence filter	Multiple faulty inputs	Src-expr-tree	Root Cause in final step?
Squid-len	3358	357	30	9	9	49	Yes
SQL-int-overflow	5917	95	36	16	12	48	No
SQL-convert	5942	93	27	9	6	64	Yes
SQL-aggregate	6847	156	44	14	8	28	Yes
SQL-precision	4566	130	34	18	17	89	Yes
SQL-mul-overflow	4652	83	13	7	5	26	Yes
SQL-dataloss	5836	153	35	17	11	152	Yes

Table 6.2: **Number of Reported Root Causes Using Automatically Generated Inputs for Server Bugs**

highly dependent on the idioms used by the programmer rather than the fundamental nature of the candidate computations: a relatively simple sub-expression may span two or more lines of code or a complex computation may be expressed in a single compact statement. For this reason, we measure the effectiveness of our analysis primarily by the number of candidate locations and secondarily by the number of source statements they represent.

6.2.1 Number of Candidate Root Causes

Overall, the analysis tool is able to narrow it down to only 5 to 17 candidate root cause expressions for all 7 bugs in two programs. From the results, it is clear that dynamic backward slicing is effective in reducing the number of candidate root causes, consistently removing more than 75% of the candidate set. In two cases, it reduces them by factors of 6 (*SQL-mul-overflow* bug) and 10 (*Squid-len* bug).

The dependence filtering step is also very effective in reducing the candidate root causes. On an average, it removes nearly 60% of the remaining candidate root causes, in one case (*Squid-len* bug) removing 70% of them.

The last filtering step, using multiple faulty inputs, has less consistent benefits in reducing false-positives. In some cases (*Squid-len* and *SQL-precision*) it has only a small or no effect at all. In other cases, the benefit is quite substantial (*SQL-int-overflow, SQL-convert, SQL-*

aggregate, and SQL-dataloss) bugs, especially considering that it comes after a series of other highly effective filters.

6.2.2 Efficacy at Finding True Root Causes

When using all of the filtering techniques, our tool could narrow down the root causes among the final filtering step for 6 of the 7 bugs as shown in Table 6.2. Our approach could not narrow-down the root cause for *SQL-int-overflow* bug among the program expressions in the final filtering step. Failure in *SQL-int-overflow* bug occurs when a specific date value (between 0000-Jan-01 and 0000-Feb-28) is used in a DATE_FORMAT query as mentioned in Section 4.2.1. In such cases, an integer overflow occurs due to the use of an unsigned variable which finally leads to a use of negative index in array access. However, the root cause is still included in the candidate set after the dependence filtering step for this bug. Hence, developers need to analyze 16 candidate program expressions expressions to find the exact root cause. We found that diagnosis for this bug is very sensitive to input. When trained with the set of automatically generated inputs, few invariants whose values flow to an invariant (identifying root cause) are not properly trained. Hence, for one of the failing inputs, one of these invariants fails and the root cause gets filtered out due to data-flow filtering for this failing input. That is why even though original failing input captures the root cause till the dependence filtering step, it gets filtered out after the multiple failing-input filtering. When we tried a different set of program inputs, then the final step was able to identify the root cause, essentially because the training runs produced a better set of invariants. Interestingly, for this bug, in an early version of our system that had *insufficient* training, dependence filtering removed the root cause because it was directly dependent on a false positive. But this did not happen and none of the bugs missed the root cause before the last "multiple faulty inputs" filtering step, since we improved the automated input generation procedure.

There are at least two strategies to deal with such cases, which are essentially "false negatives." According to our current design, the programmer can investigate the final suggestions first, and when all are eliminated, ask the tool for other suggestions. The tool could present the penultimate set of 16 locations (4 additional locations compared to initial 12 locations) in this case, and the user would investigate four more locations to find the bug. Second, the tool could be made more effective with a better input generation algorithm which can identify the root cause for all cases. Also, if we can verify whether or not a candidate location is truly a root cause, perhaps using automatic patching [43] or some other techniques, then we can iteratively try other training inputs to try to find the correct root cause. (Of course, such a verification phase would help us eliminate most false positives as well!)

Note that our approach is more likely to identify root causes of bugs, including *SQL-int-overflow* bug, than invariants extracted using general inputs. For example, using our input construction algorithm, we are able to find the root cause for *SQL-convert* bug, where a large input field (a *username*) with many special characters (as in a ftp request like

$ftp : //usernam\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash : pass@ftp.cs.wisc.edu$)

causes segmentation fault. Each special character needs three times the buffer size compared to normal characters, however the code incorrectly allocates lesser bugger size leading to a buffer overflow. If we use a general set of training inputs, the invariant ranges on the buffer length would be wider and bug-independent, making them less effective. More specifically, if any HTTP training input contained a larger *username* field than the failure-inducing input, the analysis would miss the root cause. However, our invariants with a better and focused training set could find the root cause through invariants on buffer length.

It is also important to note that our approach missed the most ideal invariant failure which could have accurately pinpointed the root cause in *SQL-dataloss* bug. This happens because the value on which invariant fails is not used in any computation, hence it gets filtered out. However fortunately, a different, but related invariant fails whose program expression statements include the actual root cause.

Bug Name	**#Invs**	**Failed Invs**	**Slice**	**Dependence filter**	**Multiple faulty inputs**	**Src-expr-tree**	**Root Cause in final step?**
Squid-len	3340	350	29	8	7	42	Yes
SQL-int-overflow	5924	95	34	15	10	18	Yes
SQL-convert	5930	91	24	8	5	59	No
SQL-aggregate	6841	152	42	13	6	17	Yes
SQL-precision	4542	141	32	17	15	55	Yes
SQL-mul-overflow	4531	78	10	5	4	21	Yes
SQL-dataloss	5985	97	28	12	9	48	Yes
Apache-overflow	2233	89	37	11	10	29	Yes

Table 6.3: **Number of Reported Root Causes Using Inputs Generated by Manually Using Systematic Procedure for Server Bugs**

6.2.3 Comparison with Systematic Manually Generated Inputs

The detailed experimental results using manually generated inputs from our systematic procedure are presented in Table 6.3. The results using manually generated inputs are similar, though little better than using automatically generated inputs. Overall, the analysis tool is able to narrow down to only 4 to 15 candidate root cause expressions for all 8 bugs in three programs using manually generated inputs. For these inputs also, dynamic backward slicing is very effective in reducing the number of candidate root causes, consistently removing more than 60% of the false positives. The dependence filtering step also effectively removes nearly 60% of the false positive candidate root causes. The last filtering step, using multiple faulty inputs, has less consistent benefits. In some cases (Squid-len and SQL-precision bugs) it has only a small effect. In other cases, the benefit is quite substantial (SQL-int-overflow, SQL-convert, and SQL-aggregate bugs), especially considering that it comes after a series of other highly effective filters.

Like SQL-int-overflow bug for automatically generated inputs, our approach could not identify the root cause for SQL-convert bug (in which a wrong algorithm to convert microsecond field to integer results in an incorrect value) after the final filtering step using the manually generated inputs. However, the root cause is still included in the candidate set after the dependence filtering step. We found that diagnosis for this bug is also very sensitive to input like SQL-convert bug. For this bug also, when we tried a different set of inputs, then the final step was able to identify the root cause. Interestingly, we missed the root cause for SQL-int-overflow bug using automatically generated inputs, but successfully identified the root cause when diagnosed with manually generated inputs. Similarly, we missed the root cause for SQL-convert bug using manually generated inputs, but successfully identified the root cause when diagnosed with automatically generated inputs. This reinforces our motivation that the inputs used for our invariants-based approach are critical to the success of this approach. Our approach is more likely to identify the root cause of bugs, including SQL-int-overflow and SQL-convert bugs, than invariants extracted using general inputs.

6.2.4 Comparison with Tarantula and Ochiai

The Tarantula [17] and Ochiai [18] statistical approaches, while simplistic in nature, are powerful methods that allow for reproduction in a reasonable amount of time and serve as a good baseline for software bug diagnosis software. These two approaches function by performing an analysis of executed program statements for a given set of inputs. Specifically, they record for each input

1. whether or not the faulty behavior was induced and
2. the statements which were executed.

This results in a vector for each program statement whose elements are 1 if the statement executed for a given input and 0 if it did not. These vectors are compared against a success/failure vector whose elements are 1 if the program exhibited the failure and 0 if it did not. A similarity coefficient is computed using each approaches' namesake formula, where each a_{ij} is a counter for the number of elements in the source line vector with value i whose corresponding element in the success/failure vector has value j. For example, a_{11} for a statement indicates the number of failing runs in which this statement gets executed at least

Bug Name	**Tarantula**	**Ochiai**	**Our Approach**
Squid-len	[6 - 5266]	[2 - 5255]	49
SQL-int-overflow	[34 - 34]	[34 - 34]	48
SQL-convert	[55 - 9409]	[55 - 9409]	64
SQL-aggregate	[776 - 9847]	[694 - 9762]	28
SQL-precision	[1 - 19]	[1 - 16]	89
SQL-mul-overflow	[5680 - 6878]	[5678 - 6876]	26
SQL-dataloss	[8 - 6060]	[8 - 6060]	152
Apache-overflow	[28 - 5372]	[1 - 5357]	29

Table 6.4: **Tarantula/Ochiai Results**

once. Similarly, a_{10} indicates the number of successful runs in which this statement gets executed at least once.

$$tarantula = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}} + \frac{a_{10}}{a_{10}+a_{00}}}$$

$$ochiai = \frac{a_{11}}{\sqrt{(a_{11}+a_{01}) * (a_{11}+a_{10})}}$$

Program statements are then ranked in decreasing order by this similarity score. The basic principle serves to identify which program statements are highly correlated with program failures. We include Ochiai in our comparison because this similarity metric has been found to give better empirical results [18]. We have implemented these approaches using the LLVM compiler system, leveraging the trace files generated by Giri mentioned in Section 5.1. Trace files in Giri keeps a record for each executed basic block. For each statement, we can use the records for its basic block in trace file to determine whether the statement got executed at least once and then it is straightforward to compute a_{ij} values since we know the outcome of each run.

Table 6.4 details our findings. Listed for each of our bugs is the range of source lines a programmer would have to inspect to find the root cause. This was computed by determining the similarity metric for each source line, ranking them, and then counting all statements that had a higher ranking than the corresponding source line. In most cases, the Tarantula or Ochiai formulas will calculate many source lines as having the same similarity score. We do not order the statements with same score in any way similar to the way in which we don't order our final set of candidate root causes. In the absence of any other technique, the developer will examine them in any random order. Thus, we rather only give a range, where the best case occurs if the root cause is listed at the top of the list of ties, and the worst case when it is listed at the bottom in any random order.

We see that the statistical correlation method of bug diagnosis' effectiveness varies dramatically depending on the type of bugs analyzed. In particular, when analyzing bugs that involve program control-flow divergence (such as SQL-int-overflow and SQL-precision bugs), we see that the Tarantula and Ochiai approaches perform notably well, but in cases where the same root cause is executed under both successful and failure-inducing inputs (such as SQL-convert, SQL-aggregate, and SQL-mul-overflow bugs), we see results that are much less impressive, as one would expect. In such cases, statistical techniques fail to distinguish between true root cause and other statements since most of them are tied with the true root cause with same similarity score. This highlights an important shortcoming of the correlation-based approaches: they fail to accurately account for cases where there is an absence of control flow divergence that leads to failure-inducing behavior. In our sampling of all bugs, the statistical methods only performed well on 2 bugs—our approach provides much more consistent behavior, regardless of the particular type of bug in question.

6.3 Experimental Methodology for Diagnosis of Clang Compiler Bugs

We selected five bugs from clang bug reports of certain clang versions with hundreds of thousands of lines of code. We started with some versions of clang and searched for bug reports (except enhancement bugs) in those versions which had proper patches available and proper test inputs available. We did not consider bugs which were wrong code bugs for these experiments. For wrong code bugs, not only root cause(s) are more complicated, but also it is not clear what should serve as the starting point for our analysis and dynamic slicing, since it is not straightforward to isolate wrong parts of the generated code. Hence, dynamic slicing may become ineffective, if we start slicing from the complete output. Thus we may need more techniques and filtering for such complex class of bugs. Similar to server bugs, we also did not consider missing code bugs (like uninitialized variable bugs) as the root cause may not be effectively identified by range invariants on program values. We leave it to future work to address such classes of bugs. We also excluded bugs which did not have inputs suitable for driving our C-Reduce based automated input generation process or for which C-Reduce failed to produce good inputs. We selected five such clang bugs. The details of the selected bugs have been shown in 6.5. Four of the selected bugs have assertion violation as their symptom and one bug exhibits memory error. For these bugs, the statement which causes assertion violation or which causes the memory error serves as the starting point for our diagnosis.

For these set of bugs, we start with the inputs from bug reports. We ran the C-Reduce tool using the inputs from bug reports to generate a set of inputs and then selected upto 8 good and bad inputs from C-Reduce tool for these experiments.

6.4 Experimental Results of Clang Bugs

In this section we present results for fault-localization of clang bugs. The detailed experimental results using automated input generation are presented in Table 6.6. Similar to server bugs result, the first column in the table lists the bug names from Table 6.5. The second column shows the total number of invariants that are generated and inserted by our system. The third column shows the total number of failed invariants by the original failure-triggering input. The fourth column shows the number of failed invariants that are on the dynamic backward slice. The fifth column shows the number after the dependence filtering step and

Bug Name	Application	LOC	LOC executed in bug	Symptom	Bug Description
Clang-lifetime	Clang-2.9-trunk	650K	35329	memory error	Accessing temporary outside their lifetime
Clang-convert	Clang-2.9-trunk	650K	46507	assert violation	Incorrectly handling conversion from pointers to boolean
Clang-scope	Clang-2.9-trunk	650K	31174	assert violation	Mishandles few cases by falsely assuming global scope specifier is present
Clang-redefine	Clang-2.9-trunk	650K	48019	assert violation	Incorrect handling when one function is redefined
Clang-cast	Clang-2.9-trunk	650K	34597	assert violation	Incorrectly handles certain types (no-op) of casts

Table 6.5: **Clang Compiler Bugs Used in the Experiments.**

Bug Name	#Invs	Failed Invs	Slice	Dependence filter	Multiple faulty inputs	Src-expr-tree	Root Cause in final step?
Clang-lifetime	15294	246	35	17	17	128	Yes
Clang-convert	20566	283	49	26	24	178	Yes
Clang-scope	13452	213	43	19	18	118	Yes
Clang-redefine	20961	243	46	30	28	169	Yes
Clang-cast	14941	-	-	-	-	-	No

Table 6.6: **Number of Reported Root Causes Using Automatically Generated Inputs for Clang Bugs**

the sixth column shows the number after the filtering using multiple failure-inducing inputs. The final column shows the maximum number of source lines corresponding to all the expression trees which the developer may need to analyze to fix the bug. The last column shows whether the candidates after the final multiple inputs filtering step includes the root cause or not.

Overall, the analysis tool is able to narrow it down to only 17 to 28 candidate root cause expressions for 4 out of 5 bugs in clang. From the results, it is clear that dynamic backward slicing is effective in reducing the number of candidate root causes, consistently removing nearly 80% of the candidate set of root causes. The dependence filtering step is also very effective in reducing the remaining false positives like server bugs. On an average, it removes nearly 50% of the remaining candidate root causes. The last filtering step, using multiple faulty inputs was not very effective for clang bugs.

Our approach failed to find the root cause for one of the bugs (Clang-cast). Figures 6.1

```

VisitCastExpr(CastExpr *ce) {
....
  else if (ce->getCastKind() == CK_NoOp) {
    skipProcessUses = true;
  }
....
}

Visit(Stmt *s) {
  skipProcessUses = false;
  Visit(s);
  if (!skipProcessUses)
    ProcessUses(s);
}
```

Figure 6.1: **Part of the Buggy Code of the Clang-cast Bug**

```

VisitCastExpr(CastExpr *ce) {
....
  else if (ce->getCastKind() == CK_NoOp ||
           ce->getCastKind() == CK_LValueBitCast) {
    skipProcessUses = true;
  }
....
}

Visit(Stmt *s) {
  skipProcessUses = false;
  Visit(s);
  if (!skipProcessUses)
    ProcessUses(s);
}
```

Figure 6.2: **Correct Version of the Code to Fix the Clang-cast Bug**

and 6.2 show a simplified part of the buggy code and the correct version of the code which fixes the failure symptom of Clang-cast bug. For some unusual types of casts in the input program, the buggy part of the code does not correctly handle them and after a series of steps it results in an assertion violation. The root cause of this failure is the missing condition in else-if statement of *VisitCastExpr* function at line 4 in Figure 6.1. *Visit(s)* at line 12 calls the *VisitCastExpr* function. But, due to the missing condition in the else-if statement, the buggy code fails to set *skipProcessUses* flag to true. Then, inside the *Visit* function, it incorrectly calls *ProcessUses* function since *skipProcessUses* flag is now false. Inside the *ProcessUses* function, it sets a variable named *lastDR* to zero and then an assertion on *lastDR* fails. Our approach could not identify the root cause because the invariants failed to capture the anomalous behavior in the buggy part of the code. This is possibly happening due to the fact that we are using a limited type of invariants on only integer and floating point type values or our generated inputs were not good enough for training. Other type of invariants like control-flow invariants may be able to capture the control flow divergence and help in accurately identifying the root cause. We would like to determine what additional techniques may be needed for fault localization of such bugs and implement them in future.

6.5 Limitations

In summary, the analysis tool is able to identify only 5 to 17 candidate root cause expressions for all 8 bugs in three server programs, one of which is millions of lines long. For clang compiler bugs, tool could isolate root causes to only 17-28 program expressions for 4 out of 5 bugs. These results indicate that bug-specific likely invariants and dynamic backwards slicing, combined with our novel filtering techniques - dependence filtering, and multiple faulty inputs filtering, can be extremely effective in helping programmers diagnose the root causes of failures.

Nevertheless, our current implementation also has some limitations. First, it cannot effectively handle missing code bugs like missing initialization. We expect such bugs can be handled by a different variety of invariants, namely, control flow invariants. We have also not yet evaluated concurrency bugs. It is not clear how effectively our framework can handle concurrency bugs, if such a bug can be reliably reproduced.

Second, various stages of the approach also depend on robust input generation, as seen for bugs which were diagnosed with one set of inputs but with another set, the root causes were dropped by the "multiple faulty inputs" filter. Hence, input generation is very critical to the success of our approach.

Third, we do not yet generate invariants for function arguments, which can make some of our expression trees unnecessarily large, if we go beyond function arguments while building expression trees. Hence, We do not go back beyond function arguments while computing expression trees to avoid making them too large. It is therefore possible to miss the root cause in the source-level expression trees, even if the root cause is included in the final set of candidate locations. We did not encounter any such case in our experiments, but adding invariants for function arguments should solve this issue, which we plan to do in future.

Finally, our technique also could not identify root cause for one clang compiler bug. We may need new techniques like more types of invariants or invariants on other types of values to address such cases.

6.6 Related Work

Delta debugging [13, 19] compares a successful and a failing run for fault localization, but does not scale to larger programs due to inaccurate mapping of program states. Recent work on program execution indexing and memory indexing [29, 20] has greatly improved delta debugging's diagnostic accuracy. However, these techniques computes causal paths in stead of root causes and previous work [22, 23] shows 45% of the computed causal paths miss root cause. Previous work [22, 23] also states that if a different input is used for a failing run, it can give inaccurate causal paths. Also, when copying state from a correct run to a faulty run, it may be necessary to copy more than the faulty state, thus resulting in false positives. It is also not clear how this approach will scale for larger programs and larger program traces. However, we follow a different approach, and it may be possible to use these techniques for checking which of our candidate root causes are actually responsible for the failure.

The first work to use invariants for bug diagnosis was DIDUCE [15]; It used a variant of range invariant to find root causes. But, we increase the likelihood of identifying the rootcause by training with similar good inputs. In addition, we support a number of filtering techniques. In contrast, DIDUCE does not use any filtering techniques and may result in more false-positives. Statistical techniques like Tarantula and others [17, 18] use differences in control flow between successful and failing runs and rank the statement based on a suspiciousness criteria. Liblit et.al. [11] developed more efficient methods to use statistical techniques with much lower overhead. However, these techniques can result in too many false positives in the sense that developer may have to analyze too many irrelevant program statements before finding the actual root cause. Pytlik et.al. [27] used invariants for fault localization of Siemens benchmark suite without much success possibly due to the fact that they used general test inputs to generate invariants.

Dynamic slicing has also been used for bug diagnosis; failure inducing chop [38] used both forward and backward slicing. Triage [12] combines differences in control flow between successful and failing runs with dynamic backward slicing for automated diagnosis. We improve upon these techniques by combining dynamic slicing with invariants similar to the work by Dimitrov et.al. [16]. We improve upon Dimitrov et.al. [16] in that our techniques to reduce false positives are not ad hoc, our system does not require special hardware support, and our system utilizes control dependence which, while it can increase false positives, is needed for diagnosing certain bugs [37].

BugAssist's [24] approach of using MAX-SAT solvers for fault localization is more systematic than ours. This approach formulates the program as a boolean trace formula in conjunctive normal form such that every satisfiable assignment of the formula corresponds to one particular execution of the program. It then extends the formula by adding constraints on input values and program post conditions. It uses a partial MAX-SAT solver to determine the smallest set of clauses which cannot be satisfied and reports the corresponding program statement as possible root causes. Execution perturbations [43, 81] have also been used to find root causes for programs. Angellic debugging [82] approach is similar in spirit to value replacement or execution perturbations approaches which searches different values for program expressions which can fix failing test without breaking any passing tests. However, in stead of running the application with different values for a statement, Java PathFinder model checker is used to check if different values for a program statement can fix the failing tests while not breaking any passing runs. But, all these techniques have only been applied to small programs. We do not believe that these techniques alone will scale to larger programs as they are computationally very expensive. However, our approach is orthogonal to these approaches and can be combined with them to improve diagnosis.

Yuan et.al. have developed a tool called SherLog [83] which can analyze source code using information provided by run-time logs to help diagnosis of production run failures. They also developed a tool called ErrLog [84] which adds logging statements into the source code to help in failure diagnosis. Though we use very different techniques for diagnosis, failure information from such tools can possibly help our approach in performing better diagnosis.

This paragraph describes related work on input generation and selection techniques for fault localization. Baudry [33] et al. propose a new technique to select test cases which can maximize diagnostic accuracy for statistical techniques. This technique is specific to statistical technique-based approaches and won't be applicable to our invariants-based system. Artzi et.al. [31, 32] also developed a technique to automatically generate test cases using a form of concolic testing which they used for fault localization of PHP applications based on statistical techniques. They also explored the impact of size and selection of test cases on accuracy of fault localization for statistical techniques. Their work combined new test case generation techniques with known statistical fault localization. Our emphasis is the opposite: our focus is more on effective fault localization techniques than on input generation. We generate inputs by modifying input according to an input grammar. This strategy is less general, but more scalable for larger programs. We also proposed using similar inputs for generating invariants which can increase diagnosis accuracy by localizing faults which may be missed by using general test inputs. However, their technique can still be adapted for generating similar inputs for invariant generation, if our approach does not work for some applications.

Finally, our work, to the best of our knowledge, performs diagnosis on much larger applications than any previous automated diagnosis techniques other than Triage [12].

A recent study [25] has shown that existing fault localization approaches, in practice, do not help programmers much. Currently, there is no practical and usable technique that can pinpoint the root cause of failures in large programs realistically and efficiently without generating a large number of false positives. We expect our approach would improve the state of the art in fault localization and make it more suitable for developers.

6.7 Conclusions

In this chapter, we evaluated our proposed new approach for automated bug diagnosis which combines invariant-based diagnosis, delta debugging, and dynamic backward slicing to compute an accurate set of possible root causes. We evaluated and compared our approach using two types of generated inputs - automatically constructed similar inputs using our developed algorithms and manually generated inputs using a systematic procedure. Using automatically generated inputs, the overall procedure is able to narrow down the root causes of bugs in large, real-world server applications to only 5–17 locations and for clang compiler bugs to only 17–28 locations, even in programs with hundreds of thousands of lines of code. Our approach also performs better than the popular Tarantula [17] and Ochiai [18] statistical approaches in localizing faults.

Chapter 7

Future Work

In this chapter, we discuss a set of possible directions to pursue for future work.

First, though our approach is now able to handle many classes of bugs, we are yet to evaluate on more difficult classes of bugs like "clang compiler wrong output code bugs" and concurrency bugs etc. For example, "compiler wrong output code bugs" not only have more complicated root cause(s), but also may make some of our filtering techniques like dynamic backward slicing completely ineffective. It is not clear what should serve as the starting point for dynamic slicing, since it is not straightforward to isolate wrong parts of the generated code for wrong code bugs. If we start slicing from the complete output, dynamic slicing may become ineffective. Hence, we plan to evaluate and develop additional techniques for such classes of bugs for which our proposed approach may be ineffective.

Second, we plan to develop on a few more systematic techniques to reduce false positives further for various classes of bugs. These stand-alone approaches will not scale to large program sizes, but they may be able to handle large programs in combination with our earlier approaches. In one possible approach, we plan to investigate new program analysis techniques to further reduce false positives. For example, we would like to investigate the use of SMT solvers to remove more false positive candidate root causes. Similar approaches [24] have used solvers like SAT and MAX-SAT solvers for fault localization. This approach formulates the program as a boolean trace formula in conjunctive normal form such that every satisfiable assignment of the formula corresponds to one particular execution of the program. It then extends the formula by adding constraints on input values and program post conditions. It uses a partial MAX-SAT solver to determine the smallest set of clauses which cannot be satisfied and reports the corresponding program statement as possible root causes. But, these stand-alone approaches will not work for larger programs. We plan to combine this technique with our approach. In particular, we would like to ask a question to SMT solver like, "Given a failure-inducing input and an execution path till a particular dynamic statement instance, is there a different value for that statement such that the failure symptom will not occur again?". Based on the answer, we can categorize the candidates into three sets - program statements for which changing their value can avoid failure, statements for which no value can avoid failure, and statements for which solver can't find an answer within a certain time duration. It is computationally expensive for a solver to answer such questions. But, this may be a feasible approach since our framework already provides a small set of candidate root causes. Successful runs have a lot of information which can be helpful to the solvers to answer such questions. Also, we can limit the control flow divergence from the the original control flow to increase its scalability. However, it is not clear how the solvers can exploit such information to increase its efficiency and scalability and further research is needed to find an answer.

Third, in another possible approach, we plan to use value-replacement techniques [42, 43] to remove false positives. The basic idea behind these techniques is to replace the value of program statements with different values and let the program run. Then a decision is made regarding which statements might be root cause depending upon the program outcome. The program has to be rerun for each possible combination of program statements and their all possible alternate values. For larger programs, this technique can't be used alone as there might be a very large number of such possible combinations. However, it may be feasible to use value replacement techniques if we can combine with our diagnosis framework. Our diagnosis framework provides two important advantages which can enable its use. Our framework already provides a very small set of program expressions after the filtering steps, which are candidate root causes. After combining our invariants-based approach with three filtering techniques, we have less than hundred program expressions which are candidate root causes instead of hundreds of thousands of program expressions. We also have a set of possible alternate values for each program statement from the successful runs and invariant ranges. Thus, we have to search a much smaller set of possible combination of program statement and its alternate values. We can also make a more exhaustive and informed search of possible values for each statement. This approach may work better for larger programs than applying value-replacement as a stand-alone approach.

Next, we have only used one kind of invariants in our framework which will not be able to localize faults for all classes of bugs. For example, diagnosis of bugs like missing initialization or more complex bugs like compiler wrong code bugs may need new invariants like *def-use invariants* or multi-variable invariants. We would like to develop and explore a broader class of invariants (e.g., loop-based invariants, address-based invariants, multi-variable relational invariants and def-use invariants) which can capture a wider variety of bugs. In addition, we are only tracking integer and floating-point type of values, we will miss the bugs which does not involve any of these type of values. So, we would like to generate invariants involving other type of values (including address values) to widen the scope of our tool.

We also generate invariants on load, store and function return values. Since, we trace back till load, store and function arguments while generating program expressions of candidate root causes, we can miss the root causes if the wrong value from root cause flows through function arguments. If we monitor invariants on function arguments as well, then we can safely end at function arguments while building expression trees of candidate root causes without missing the root cause. Also, since we do not generate invariants on intermediate values, we have to report a much larger set of program expressions, in place of a small set of program statements. We would like alleviate these drawbacks by using invariants on more program points and variables. Another way to handle the second issue and improve the effectiveness of our approach is to recursively add invariants on all the statements in the candidate program expressions and repeat the diagnosis procedure. This way, we may be able to reduce the size of program statements the developer needs to examine.

Our automatic input generation step is also not yet very robust. We plan to develop it further so that it can handle more variety of bugs and applications. We also plan to evaluate the effectiveness of different approaches for input generation like grammar-independent, grammar-dependent, replacement and deletion based algorithms. Also, the C-Reduce [28] tool is designed to generate minimal inputs, not specifically designed to generate inputs for better diagnosis. It is possible to make modifications to this input generation process to generate inputs more suitable our fault localization approach.

We also evaluated our approach with only thirteen software bugs. Hence, we would like to evaluate our techniques with more bugs of similar and other varieties as well as other applications. We also plan to evaluate our approached on unfixed bugs to reduce any kind of bias in our experiments. We would also like outside users to use and evaluate our approach.

Finally, Production run failures are especially difficult to diagnose for various reasons. As others have noted [12, 85], performing off-site analysis of production run failures at development sites has several limitations: 1) it is difficult to reproduce failures at the development site due to differences in the environment, 2) customers have privacy concerns over what information can be released for off-site diagnosis, and 3) the same bug may generate a different failure at multiple production sites; it is cumbersome for the development team to investigate every failure that occurs without any automated feedback regarding the root cause of the failure. Hence, performing as much diagnosis as possible at the production site will be beneficial. We would like to work on extending our approach in future so that it can diagnose failures during production run as well. For this, we may also need to develop deterministic detector for failures (especially for wrong output bugs since our current approaches may not work for production run failures).

Chapter 8

Conclusions

In this proposal, we reported the results of our empirical study of server software bugs. Our results show that most server bugs are deterministic and that failures due to most bugs (77%) can be reproduced by replaying a single input request (after session establishment if needed). Even for the remaining multi-input bugs, the set of inputs needed for reproducing failures is usually small and are often clustered together within the input stream. Our results also showed that many bugs produce incorrect outputs, indicating that better detectors are needed to detect errors during production runs. Most of the concurrency bugs, though, need multiple inputs to reproduce a symptom and are non-deterministic. One of the key implications of the study is that most of the failures may be reproduced without checkpointing server state.

We also proposed a new approach for automated bug diagnosis that combines invariant-based diagnosis, delta debugging, and dynamic backward slicing to compute an accurate set of possible root causes. We proposed use of automatically constructed good inputs which are similar to the failure-inducing input to develop better invariants that are more likely to find root causes. We also developed algorithms to automatically generate such similar inputs. We also proposed two new heuristics that significantly reduce the false positives. The overall procedure is able to narrow down the root causes of bugs in large, real-world server applications to only 5–15 locations and bugs in clang compiler to 17-28 program locations, even in programs with hundreds of thousands of lines of code.

In future, we plan to work on additional program analysis techniques to reduce the candidate set of root causes further. We also plan to explore other types of invariants at more program points so that we can increase the diagnostic capability of the tool and can diagnose more variety of bugs. We also plan to make input construction process more robust.

Our developed diagnosis framework and proposed approach will greatly help software programmers, reduce software development costs, increase programmer productivity and software reliability.

References

[1] S. K. Sahoo, J. Criswell, and V. Adve, "An empirical study of reported bugs in server software with implications for automated bug diagnosis," in *ICSE*, 2010.

[2] ——, "Using likely invariants for automated software bug diagnosis," in *To appear in Proc of Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[3] S. K. Sahoo, M. Li, P. Ramachandran, S. V. Adve, V. S. Adve, and Y. Zhou, "Using likely program invariants to detect hardware errors," in *International Conference on Dependable Systems and Networks*, 2008.

[4] R. N. Charette, "Why Software Fails," *Spectrum, IEEE*, vol. 42, no. 9, pp. 42–49, September 2005.

[5] "NIST: National Institute of Standards and Technology. Software errors cost U.S. economy $59.5 billion annually: NIST assesses technical needs of industry to improve software-testing," June 2002.

[6] D. Scott, "Making smart investments to reduce unplanned downtime," *Tactical Guidelines TG-07-4033, Gartner Group*, March 1999.

[7] S. M. Srinivasan, S. Kandula, S. K, C. R. Andrews, and Y. Zhou, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *USENIX Annual Technical Conference, General Track*, 2004.

[8] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: detecting atomicity violations via access interleaving invariants," in *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[9] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *PLDI*, 2006.

[10] M. Bond, "Diagnosing and tolerating bugs in deployed systems," Ph.D. dissertation, University of Texas at Austin, 2008.

[11] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[12] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou, “Triage: diagnosing production run failures at the user’s site,” in *SOSP*, 2007.

[13] A. Zeller, “Isolating cause-effect chains from computer programs,” in *SIGSOFT ’02: FSE*, 2002.

[14] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas, “Accmon: Automatically detecting memory-related bugs via program counter-based invariants,” in *Proc. ACM/IEEE Int’l Symposium on Microarchitecture (MICRO)*, 2004. [Online]. Available: citeseer.ist.psu.edu/zhou04accmon.html

[15] S. Hangal and M. S. Lam, “Tracking down software bugs using automatic anomaly detection,” in *Proceedings of the International Conference on Software Engineering*, 2002.

[16] M. Dimitrov and H. Zhou, “Anomaly-based bug prediction, isolation, and validation: an automated approach for software debugging,” in *ASPLOS*, 2009.

[17] J. A. Jones and M. J. Harrold, “Empirical evaluation of the tarantula automatic fault-localization technique,” in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE ’05, 2005.

[18] R. Abreu, P. Zoeteweij, and A. J. van Gemund, “An evaluation of similarity coefficients for software fault localization,” in *12th Pacific Rim International Symposium on Dependable Computing(PRDC)*, 2006.

[19] H. Cleve and A. Zeller, “Locating causes of program failures,” in *ICSE*, 2005.

[20] W. N. Sumner and X. Zhang, “Memory indexing: canonicalizing addresses across executions,” in *FSE*, 2010.

[21] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Transactions on Software Engineering*, 2002.

[22] W. N. Sumner and X. Zhang, “Algorithms for automatically computing the causal paths of failures,” in *FASE*, 2009.

[23] ——, “Automatic failure inducing chain computation through aligned execution comparison,” in *Technical Report 08-023, Purdue University*, 2008.

[24] M. Jose and R. Majumdar, “Cause clue clauses: error localization using maximum satisfiability,” in *PLDI*, 2011.

[25] C. Parnin and A. Orso, “Are automated debugging techniques actually helping programmers?” in *ISSTA*, 2011, pp. 199–209.

[26] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *IEEE Trans. Software Eng.*, 2001.

[27] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss, "Automated fault localization using potential invariants," *In Proceedings of the Workshop on Automated and Algorithmic Debugging*, 2003.

[28] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellision, and X. Yang, "Test-case reduction for c compiler bugs," in *Submission*, 2012.

[29] B. Xin, W. N. Sumner, and X. Zhang, "Efficient program execution indexing," in *PLDI*, 2008.

[30] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-13, 2005.

[31] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical fault localization for dynamic web applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10, 2010.

[32] ——, "Directed test generation for effective fault localization," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10.

[33] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06, 2006.

[34] M. Renieres and S. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003*, 2003.

[35] R. Abreu, A. González, P. Zoeteweij, and A. J. C. van Gemund, "Automatic software fault localization using generic program invariants," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08, 2008.

[36] X. Zhang and R. Gupta, "Cost effective dynamic program slicing," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, ser. PLDI '04, 2004.

[37] X. Zhang, N. Gupta, and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults," *Empirical Software Engineering*, 2007.

[38] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *ASE*, 2005.

[39] X. Zhang, N. Gupta, and R. Gupta, "Locating faulty code by multiple points slicing," *Software: Practice and Experience*, vol. 37, no. 9, pp. 935–961, 2007.

[40] D. Jeffrey, V. Nagarajan, R. Gupta, and N. Gupta, "Execution suppression: An automated iterative technique for locating memory errors," *ACM Trans. Program. Lang. Syst.*, vol. 32, pp. 17:1–17:36, May 2008.

[41] N. Gupta and R. Gupta, "Expert: Dynamic analysis based fault location via execution perturbations," *Parallel and Distributed Processing Symposium, International*, p. 332, 2007.

[42] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *Proceedings of the 2008 international symposium on Software testing and analysis*, ser. ISSTA '08, 2008.

[43] ——, "Effective and efficient localization of multiple faults using value replacement," in *ICSM*, 2009.

[44] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, pp. 229–247, June 2006.

[45] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616 (Draft Standard), Internet Engineering Task Force, Jun. 1999, updated by RFC 2817. [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt

[46] "Squid User's Guide," Website, http://www.deckle.co.za/squid-users-guide/Main_Page.

[47] "MySQL Internals ClientServer Protocol," Website, http://forge.mysql.com/wiki/MySQL_Internals_ClientServer_Protocol.

[48] M. Bond and D. Law, *Tomcat Kick Start.* Sams, 2002.

[49] T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Protocol Architecture," RFC 4251 (Proposed Standard), Internet Engineering Task Force, Jan. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4251.txt

[50] "SVN Protocol Description," Website, http://svn.collab.net/repos/svn/trunk/www/webdav-usage.html.

[51] "Apache Bugs Database," Website, https://issues.apache.org/bugzilla/.

[52] "OpenSSH sshd Bugs Database," Website, https://bugzilla.mindrot.org/.

[53] C. Bird *et al.*, "Fair and Balanced?: Bias in Bug-fix Datasets," in *ESEC/FSE '09*, 2009, pp. 121–130.

[54] J. W. Nimmer and M. D. Ernst, "Automatic generation of program specifications," in *Proc. ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, 2002.

[55] D. Dhurjati, S. Kowshik, and V. Adve, "SAFECode: Enforcing alias analysis for weakly typed languages," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006, pp. 144–157.

[56] S. Chandra and P. M. Chen, "Whither generic recovery from application faults? a fault study using open-source software," in *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 97–106.

[57] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, "Characterization of linux kernel behavior under errors," in *DSN*, 2003, pp. 459–468.

[58] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008, pp. 329–339.

[59] J. Gray, "A census of tandem system availability between 1985 and 1990," *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 409–418, Oct 1990.

[60] I. Lee and R. Iyer, "Faults, symptoms, and software fault tolerance in the tandem guardian90 operating system," in *FTCS-23*, Jun 1993, pp. 20–29.

[61] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability - a study of field failures in operating systems," in *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*. Montreal, Que., Canada: IEEE Computer Society Press, 1991, pp. 2–9. [Online]. Available: citeseer.ist.psu.edu/sullivan91software.html

[62] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *ASID*, 2006.

[63] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, ser. OSDI'00. Berkeley, CA, USA: USENIX Association, 2000, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251229.1251230

[64] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in *International Symposium on Empirical Software Engineering and Measurement*, 2009.

[65] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 319–329. [Online]. Available: http://dl.acm.org/citation.cfm?id=776816.776855

[66] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proc. Conf. on Code Generation and Optimization*, San Jose, CA, USA, Mar 2004, pp. 75–88.

[67] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Trans. on Prog. Langs. and Systs. (TOPLAS)*, pp. 13(4):451–490, October 1991.

[68] J. Seward, “Valgrind, an open-source memory debugger for x86-gnu/linux.” [Online]. Available: http://developer.kde.org/~sewardj/

[69] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “Softbound: highly compatible and complete spatial memory safety for c,” *SIGPLAN Not.*, vol. 44, no. 6, pp. 245–258, 2009.

[70] C. Ellison and G. Rosu, “An executable formal semantics of c with applications,” in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL ’12, 2012.

[71] P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti, “Experience report: Ocaml for an industrial-strength static analysis framework,” in *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ser. ICFP ’09, 2009.

[72] “clang: a C language family frontend for LLVM,” Website, http://clang.llvm.org/.

[73] “LLVM,” `http://llvm.cs.uiuc.edu`, 2006.

[74] D. L. Bird and C. U. Munoz, “Automatic generation of random self-checking test cases,” *IBM Systems Journal*, vol. 22, no. 3, pp. 229 –245, 1983.

[75] J. E. Forrester and B. P. Miller, “An empirical study of the robustness of windows nt applications using random testing,” in *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4*, ser. WSS’00, 2000, pp. 6–6.

[76] “Protos,” Website, http://www.ee.oulu.fi/research/ouspg/protos/.

[77] “Spike,” Website, http://www.immunitysec.com/resources-freesoftware.shtml.

[78] P. Godefroid, N. Klarlund, and K. Sen, “Dart: directed automated random testing,” in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI ’05, 2005.

[79] C. Cadar, D. Dunbar, and D. Engler, “Klee: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI’08, 2008.

[80] P. Godefroid, M. Y. Levin, and D. A. Molnar, “Automated whitebox fuzz testing,” in *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008. [Online]. Available: http://www.truststc.org/pubs/499.html

[81] X. Zhang, N. Gupta, and R. Gupta, “Locating faults through automated predicate switching,” in *ICSE*, 2006.

[82] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011.

[83] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS '10, 2010.

[84] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12. USENIX Association, 2012.

[85] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. C. Hunt, "Debugging in the (very) large: ten years of implementation and experience." in *SOSP*, 2009.